深度探索Go语言

对象模型与runtime的原理、特性及应用

[redacted] 编著

清华大学出版社

目　录

版权信息

书名：深度探索Go语言：对象模型与runtime的原理、特性及应用
编著：封幼林
出版社：清华大学出版社
出版时间：2022-08-01
ISBN：9787302600855

.

内容简介

本书主要讲解Go语言的一些关键特性的实现原理。Nicklaus Wirth大师曾经说过：算法+数据结构＝程序，语言特性的实现不外乎是数据结构+代码逻辑。

全书内容共分为4部分：第一部分是基础特性（第1～3章），第二部分是对象模型（第4和5章），第三部分是调度系统（第6和7章），第四部分是内存管理（第8和9章）。书中主要内容包括指针、函数栈帧、调用约定、变量逃逸、Function Value、闭包、defer、panic、方法、Method Value、组合式继承、接口、类型断言、反射、goroutine、抢占式调度、同步、堆和栈的管理，以及GC等。

书中包含大量的探索示例和源码分析，读者在学会应用的同时还能了解实现原理。书中绝大部分代码是用Go语言实现的，还有少部分代码是用汇编语言实现的，这些代码都可以使用Go语言官方SDK直接编译。探索过程循序渐进、条理清晰，用到的工具也都是SDK自带的，方便读者亲自上手实践。

本书适合Go语言的初学者，在学习语言特性的同时了解其实现原理。更适合有一定的Go语言应用基础，想要深入研究底层原理的技术人员，以及有一些其他编程语言基础，想要转学Go语言的开发者阅读。

作者简介

封幼林 资深软件工程师，十多年IT从业经验，曾涉足Win32桌面程序开发、Android移动端开发，以及互联网服务器端开发等多个领域。喜欢研究底层技术，用自己的方法探究背后的实现原理。热爱技术交流与分享，创建了微信公众号“幼麟实验室”，致力做一些形象、通透的计算机教程，让开发者“知其然亦知其所以然”。

序一
FOREWORD

常有同学问我，学习技术的原理、机制到底有什么用？现在已经有很多不同的操作系统和编程语言，我们个人不太可能再去实现操作系统或编程语言！的确如此，如果以学习为目的，我们可以实现操作系统或编程语言的简陋原型，但在工作中并没有这样的需求和机会，而学习原理、机制的目的是增进自己对技术问题的判断，同时获得对典型问题和最佳方案的积累，让自己具备分析复杂技术问题和求解正确的技术方案的能力。

对于编程语言，优秀的程序员既能熟练地使用语言的各种特性，快速满足业务领域开发，又可以掌握语言的设计原理和底层机制。既是别人眼中的快刀手，也是面对难题，一击必中的高手。工作中在面对不同技术方案时，可以快速做出最合理的选择。既可以解决当前的问题，又可以让系统长治久安地演进，将来不会推倒重来，而这些分析判断都取决于你对技术原理和机制的理解。

最近几年，Go语言进展迅速，吸引广大的程序员学习和使用。Go语言有很多优秀特性，如goroutine可以让大家轻易写出高并发的服务。语言掌握起来也简单，往往学习两三周，就可以实际投入工作开发，但真正遇到复杂的场景、资源竞争或GC敏感时，缺少对Go语言机制和进程结构的理解，你会很难完成上述挑战。很可能当你使用Go语言多年后，仍然不能写出健壮的核心业务服务。

在《深度探索Go语言——对象模型与runtime的原理、特性及应用》中，封幼林把Go语言主要的核心特性从原理到应用，从底层的汇编代码到Go语言代码，以庖丁解牛般的剖析让读者对Go语言豁然开朗，使语言的原理与机制变得清晰和简单。相信读者在认真学习后，将使自己对Go语言理解与掌握有一个质的飞跃。

左文建
奇安信集团副总裁

序二
FOREWORD

Go语言诞生距今已有十余年，我最开始使用Go语言还是在2012年，当时Go语言的1.0版本刚刚发布，虽然继承了Plan 9的衣钵，却有很多让人诟病的地方。我们当时用Go语言实现了一些HTTP Client和网络爬虫业务，虽然编写过程十分顺畅，但是会遇到goroutine和GC的性能和其他稳定性的问题，于是就变成了一次浅尝辄止的尝试。

随着时间的推移，我再次在业务中使用的Go语言已经到了1.4版本，它的稳定性问题已得到了解决。很快，随着Go 1.5版本的发布，GC性能问题也不复存在，Go语言终于成长为一门优秀的开发语言。而随着最近几次版本的新特性——泛型的加入，Go语言在表达能力上获得更进一步的提升，未来十分可期。

我大部分时间在用Go语言写服务器端程序，但也用Go语言写过客户端程序，写过PoC，写过DSL，写过JIT，甚至写过嵌入式程序的通信界面，Go语言现在对我来讲已经成为相当称手的工具。选择Go语言进行开发意味着快速、便捷、高性能，甚至它已经成为云原生的代名词。

在我最初接触Go语言的时候，当时唯一一本Go语言的书籍就是许式伟老师编写的《Go语言编程》，可以说是大家用中文学习Go语言的唯一途径，而现在则不断有很棒的中文书籍问世。《深度探索Go语言——对象模型与runtime的原理、特性及应用》直接从底层开始，为大家介绍需要的汇编基础知识，紧接着从指针、函数、goroutine逐步深入，不断剖析Go语言原理，让大家获得最贴近实现原理的知识。拨开运行时的迷雾，不必猜测编写的Go语言代码运行时的行为，真正地让大家掌握Go语言全部的精髓。可以毫不夸张地说，这是一本Go语言的High-End图书。

书中作者先用示例代码描述原理和概念，然后辅以图例说明，最后使用对应生成的汇编代码予以佐证，可以说是学习Go语言底层知识的最佳途径。我阅读Go语言源代码特别喜欢直接在Go语言源码中进行Hack，得益于Go语言的编译速度，Hack完毕后进行编译，然后测试修改结果也十分迅速，这无疑提升了学习速度。建议大家不要怕源代码，只有在源代码中才能洞悉设计者的真正意图，才能理解设计所面临的工程问题和解决方案的精妙之处。

相信大家看完本书后，一定会受益匪浅，水平得到质的提升！

张旭红
金山办公Exline技术副总监，掘金技术社区前技术总监

序三
FOREWORD

不知从什么时候起，Go语言圈子里突然多了一个看上去很可爱的蛋壳形象，把Go语言的底层实现与枯燥的runtime知识变成了妙趣横生的动画，呈现在编程世界里，赢得了大家的喜爱，让人有一种“旧时王谢堂前燕，飞入寻常Gopher家”的感觉。

一件事情，一门技术，如果能让大多数人觉得有意思，那再去学习它就不是什么难事了。

作者输出的文章和动画让我对作者本身也产生了一些兴趣，因为我也是一个Gopher和技术写作者，深知把庞杂的底层知识给别人讲明白是一件多么有挑战的事情。这些透彻的文章，以及看了令人舒心的技术动画，到底是怎么制作出来的呢？

在微信上与作者做过简单的交流之后，得知了作者自身多年的开发经验，以及底层与C++的研发背景，这些谜团便揭开了。在我的研发生涯中碰到的大多数C/C++工程师，对于底层和高并发知识都能如数家珍，但能够将这些积累与他人说清道明并不是每个人都能做得到的。既能学会又能讲清之人少之又少，作者就是其中之一，会使用VideoScribe的工程师也不是那么多。

现代的软件工程师，无论是应用工程师，还是基础设施工程师，都会对底层、高并发知识有浓厚的兴趣，这不是没有理由的，因为这些知识能够帮助我们定位出大部分日常开发中碰到的性能问题，理解并解决所有线上遇到的高并发环境下才会触发的Bug。这些知识也是每个有追求的互联网公司的工程师所必备的专业素质，多读书，多写代码，多积累，最终才能让我们一步一步成为一个合格的技术专家，这些与公司内的头衔无关，是真正的技术硬实力。

本书的内容主要是Go语言的底层知识，相比其他写底层的书而言可能没有覆盖到“所有”底层细节，但其覆盖到的内容都是细之又细，相信大家在阅读本书或阅览作者制作的Go语言系列动画时一定能够有所收获。

曹春晖
《Go语言高级编程》作者

序四
FOREWORD

得知《深度探索Go语言——对象模型与runtime的原理、特性及应用》即将出版上市，我感到非常高兴，更开心的是作者邀请我为此书写推荐序。

我和封幼林的相识，是通过幼麟实验室。幼麟实验室从2020年5月开始，持续以图解的形式讲解计算机和Go语言的相关知识，至今已经发布了一系列与Go语言相关的视频。内容涉及Go语言的slice、map、内存对齐、函数栈帧、闭包、defer和panic等基础特性，还有反射、goroutine、调度系统、Mutex、channel，以及GC等复杂问题，都以简单易懂的形式呈现出来。对广大Gopher来讲，是非常不错的参考学习资料。

本书是作者在图解视频和知乎系列文章的基础上，更加系统地重新创作而成。我们从本书的副标题“对象模型与runtime的原理、特性及应用”，就能看出本书的侧重点，对于想要深入了解这部分内容的读者很有参考价值。

本书从反汇编开始，结合图示讲解和源码分析，非常系统地探索了Go语言的基础特性、对象模型、调度系统和内存管理模块。在讲解Go语言底层知识的同时，作者的探索方法也很值得学习借鉴，让我们知其然也知其所以然。特别是对想要亲自动手探索语言底层实现的读者来讲，简直就是福音。

杨文
Go夜读发起人

前言
PREFACE

近几年来，Go语言作为一门服务器端开发语言越来越受欢迎，简洁易学的语法加上天生的高并发支持，还有日益完善的社区，让很多互联网公司开始转向Go语言。随着Go语言生态日趋成熟，各种组件框架如雨后春笋般涌现，市面上相关的书籍也多了起来，但是其中大部分是以应用为主，对于语言特性本身探索一般不太深入。笔者希望能够有一本讲解语言特性及实现原理的书，这也是写作本书的动机。

笔者当年刚参加工作的时候，使用的第一门开发语言是C++。虽然之前在学校用过C语言和汇编语言，但在接触到C++的一些面向对象特性时还是困惑了很久。直到有一天发现了《深度探索C++对象模型》，作者Stanley Lippman当年在贝尔实验室工作，是世界上第1个C++编译器——cfront的实现者，他从一个语言实现者的高度，对一些关键特性的实现原理及其背后的思考进行了详细阐述，使笔者受益匪浅。后来因为工作的原因，笔者开始使用Go语言，因为有了C/C++相关的基础，所以学习起来更加高效。尤其是当年学习C++对象模型，让笔者认识到语言特性也是通过数据结构和代码实现的，所以就按照自己的方式一边学习一边探索。第一次萌生要写点东西的念头是在给从PHP转Go语言的妻子讲完接口动态派发的实现原理后，用她的话来讲就是有种豁然开朗的感觉，并鼓励笔者把这些东西整理一下。后来我们就在微信公众号上以幼麟实验室的名义发布了一系列视频和文章，主要分析语言特性的底层实现。在一年多的时间里，幼麟实验室受到了广大网友的好评与支持，清华大学出版社的赵佳霓编辑也是在此期间联系了笔者，希望笔者能够把自己的探索研究整理成书。因为写作本书的关系，让笔者能够更系统地思考，收获颇多。希望本书能够帮助各位读者，解决大家学习Go语言中遇到的一些困惑。

本书主要内容

第1章介绍x86汇编的一些基础知识，包括通用寄存器、几条常用的指令，以及内存分页的实现原理等。

第2章介绍指针的实现原理，包括指针构成、相关操作，以及Go语言的unsafe包等。

第3章围绕函数进行一系列探索，包括栈帧布局、调用约定、变量逃逸、Function Value、闭包、defer和panic等。

第4章介绍方法的实现原理，包括接收者类型、Method Value和组合式继承等。

第5章围绕接口对Go语言的动态特性展开探索，包括装箱、方法集、动态派发、类型断言、类型系统和反射等。

第6章介绍goroutine的实现，包括GMP模型、goroutine的创建与退出、调度循环、抢占式调度、timer、netpoller和监控线程等。

第7章介绍同步的原理及其相关的组件，包括内存乱序、原子指令、自旋锁、Go语言runtime中的互斥锁和信号量，以及sync.Mutex和channel等。

第8章介绍堆内存管理，包括heapArena、mspan等几种主要的数据结构，mallocgc函数的主要逻辑，以及GC的三色抽象、写屏障等。

第9章介绍栈内存管理，包括goroutine栈的分配、增长、收缩和释放等。

阅读建议

本书写作过程主要使用了Go 1.16及之前的几个版本，为了避免后续版本可能发生的不兼容问题，相关示例建议使用Go 1.13～Go 1.16编译运行。

阅读本书不需要精通汇编语言、操作系统，但是需要对进程、线程这类基本概念有所了解。毕竟Go语言可直接构建生成系统原生的可执行文件，如果想要深入理解一些语言特性的实现原理，还

是建议学习并实践一下多线程编程、IO多路复用这类关键技术。

第一部分主要包括指针和函数，笔者希望大家能够通过这部分内容，对运行时栈及函数栈帧的相对寻址方式有深入的理解，为后续探索打下坚实的基础。

第二部分想要表达对Lippman大师的崇高敬意，至今难忘初次阅读《深度探索C++对象模型》时那种“初闻大道，喜不自胜”的心情。按照Lippman大师的解释，对象模型应该是编译器对自定义数据类型的建模，指导了对象内存布局及其他一些数据结构和代码的生成。只有理解了语言特性的实现原理，才真正是磨刀不误砍柴工。

第三部分从服务器端程序开发的角度，梳理了如何从最初的多进程、多线程，逐渐发展到现在的协程。runtime的调度逻辑还是比较复杂的，但是最核心的思想就是IO多路复用与协程的结合，让每个任务有自己独立的栈，而同步的核心就是确立Happens Before条件。

第四部分从堆和栈两方面，梳理了内存管理的实现。内存分配方面应重点关注主要的数据结构。至于GC方面，应先理解宏观层面的整体思想和流程，然后去研究一些细节会更加容易。整个runtime实际上是个不可分割的整体，在这里会看到内存管理对类型系统的依赖。

致谢

感谢那些喜爱Go语言的网友对笔者的支持；感谢清华大学出版社的赵佳霓编辑；感谢我的家人，尤其是和我一起讨论技术问题并帮忙整理书稿的妻子，给予我莫大的支持。

由于时间仓促，并且受限于笔者水平，书中难免有不妥之处，请读者见谅，并提宝贵意见。

封幼林
2022年5月

第1章
汇编基础

20世纪90年代，随着Microsoft Windows系统在世界范围流行，Intel公司的x86架构CPU占据了个人计算机的主要市场。近十年来，开源的Linux系统日渐成熟完善，伴随着云计算的热潮，各大互联网巨头纷纷推出了基于x86_64架构的弹性计算服务，x86架构又占据了服务器的主要市场。本章只简单讲解x86汇编语言的必要基础知识，其目的是为后续研究Go语言的底层特性做好准备。熟悉x86架构和x86汇编的读者，可以跳过本章直接阅读后续章节。

文中所使用的寄存器名称，以及示例汇编代码都符合Intel汇编风格，与Go语言自带的反汇编工具有一些差异，在本章的最后会进行简单的对比说明。

1.1 x86通用寄存器

本节简单介绍一下x86架构的通用寄存器，包括32位的x86架构和64位的x86_64架构，后者是由AMD公司首先推出的，也称为amd64架构。因为64位架构是基于32位扩展而来的，保持了向前兼容，所以本节先介绍32位架构，再介绍64位进行了哪些扩展。

1.1.1 32位架构

32位x86架构的CPU有8个32位的通用寄存器，在汇编语言中可以通过名称直接引用这8个寄存器。按照Intel指令编码中的编号和名称如表1-1所示。

表1-1 Intel指令编码中8个通用寄存器的编号和名称

编号	名称	编号	名称
0	EAX	4	ESP
1	ECX	5	EBP
2	EDX	6	ESI
3	EBX	7	EDI

其中编号为0～3的4个寄存器还可以进一步拆分。如图1-1所示，EAX的低16位可以单独使用，引用名称为AX，而AX又可以进一步拆分成高字节的AH和低字节的AL两个8位寄存器。

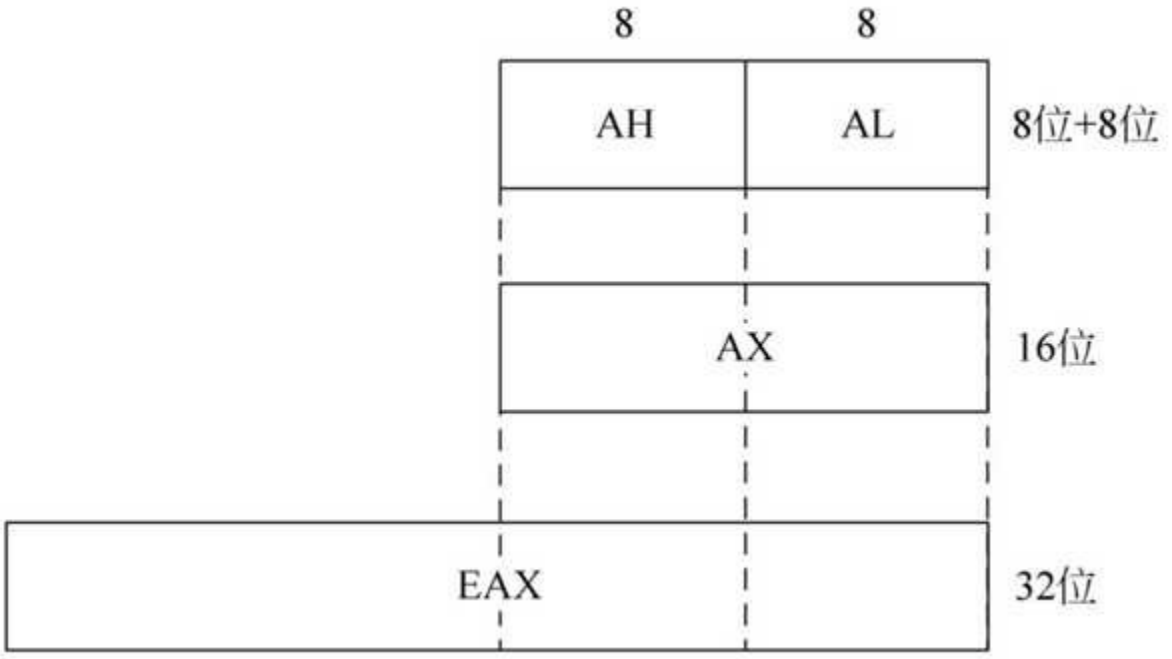

图1-1 EAX寄存器的结构

EAX、ECX、EDX和EBX寄存器都是按照表1-2所示的方式设计的。这种设计让开发者能够非常方便地对不同大小的数据进行操作。

表1-2 编号为0～3的寄存器的结构设计

32 位	16 位	高 8 位	低 8 位
EAX	AX	AH	AL
ECX	CX	CH	CL
EDX	DX	DH	DL
EBX	BX	BH	BL

编号为4～7的4个寄存器，低16位也有独立的名称，但是没有对应的8位寄存器，如表1-3所示。可以认为这4个16位寄存器是为了向前兼容16位的8086，在32位的程序中很少使用。

表1-3 编号为4～7的寄存器的结构设计

32 位	16 位	32 位	16 位
ESP	SP	ESI	SI
EBP	BP	EDI	DI

有些通用寄存器是有特殊用途的：

（1）EAX寄存器会被乘法和除法指令自动使用，通常称为扩展累加寄存器。

（2）ECX被LOOP系列指令用作循环计数器，但是多数上层语言不会使用LOOP指令，一般通过条件跳转系列指令实现。

（3）ESP用来寻址栈上的数据，很少用于普通算数或数据传输，通常称为扩展栈指针寄存器。

（4）ESI和EDI被高速内存传输指令分别用来指向源地址和目的地址，被称为扩展源索引寄存器和扩展目标索引寄存器。

（5）EBP在高级语言中被用来引用栈上的函数参数和局部变量，一般不用于普通算数或数据传输，称为扩展帧指针寄存器。

除了这些通用寄存器之外，还有一个标志寄存器EFLAGS比较重要。汇编语言中用于比较的CMP和TEST会修改标志寄存器里的相关标志，再结合条件跳转系列指令，就能实现上层语言中的大部分流程控制语句，此处不进一步展开。

最后还有一个很重要而且很特殊的寄存器，即指令指针寄存器EIP。指令指针寄存器中存储的是下一条将要被执行的指令的地址，而且汇编语言中不能通过名称直接引用EIP，只能通过跳转、CALL和RET等指令间接地修改EIP的值。

1.1.2 64位架构

64位架构把通用寄存器的个数扩展到16个，之前的8个通用寄存器也被扩展成了64位，每个寄存器的低8位、16位、32位都可以单独使用。寄存器结构设计如表1-4所示。

表1-4 64位架构下16个通用寄存器的结构设计

64 位	32 位	16 位	8 位
RAX	EAX	AX	AL
RCX	ECX	CX	CL
RDX	EDX	DX	DL
RBX	EBX	BX	BL
RSP	ESP	SP	SPL
RBP	EBP	BP	BPL
RSI	ESI	SI	SIL
RDI	EDI	DI	DIL
R8 ～ R15	R8D ～ R15D	R8W ～ R15W	R8B ～ R15B

指令指针EIP被扩展为64位的RIP，但依然不能在代码中直接引用。标志寄存器EFLAGS被扩展为64位的RFLAGS，里面的标志位保持向前兼容。

内存地址也扩展到了64位，实际上目前的硬件只使用了低48位，在1.3节介绍内存分页机制时会进行相关说明。

1.2 常用汇编指令

x86的汇编指令一般由一个opcode（操作码）和0到多个operand（操作数）组成，大多数指令包含两个操作数，一个目的操作数和一个源操作数。为了便于理解上层语言中一些特性的实现，下面简单介绍几条常用的指令。

1.2.1 整数加减指令

x86汇编使用ADD指令进行整数的加法运算，该指令有两个操作数，第1个操作数也叫作目的操作数，第2个操作数也叫作源操作数。ADD指令把两个操作数的值相加，然后把结果存放到目的操作数中。源操作数可以是寄存器、内存或立即数，而目的操作数需要满足可写的条件，所以只能是寄存器或内存，而且两个操作数不能同时为内存。

如下指令将EAX寄存器的值加上16，并把结果存回EAX中，指令如下：

```
ADD EAX, 16
```

整数减法运算通过SUB指令来完成，对操作数的要求和ADD指令一致，不过是从目的操作数中减去源操作数，并把结果存回目的操作数中。

如下指令将ESP寄存器的值减去32，并把结果存回ESP中，就像高级语言中分配函数栈帧时所做的那样，指令如下：

```
SUB ESP, 32
```

包括ADD和SUB在内的很多汇编指令能够接受不同大小的参数，例如通过两个8位寄存器进行int8加法，指令如下：

```
ADD AL, CL
```

通过两个16位寄存器进行int16加法，指令如下：

```
ADD AX, CX
```

x86是一个复杂指令集架构，很多指令像这样支持多种操作数组合，虽然代码中使用同一个opcode名称，但是实际编译后对应的是不同的opcode。上层语言中的数据类型会指导编译器，在编译阶段选择合适的opcode和对应的operand。

1.2.2 数据传输指令

x86有多种数据传输指令，这里只简单介绍最常用的MOV指令。MOV指令主要用来在寄存器之间及寄存器和内存之间传输数据，也可以用来把一个立即数写到寄存器或内存中。第1个操作数称为目的操作数，第2个操作数是源操作数，MOV指令用于把源操作数的值复制到目的操作数中。

把ECX寄存器的值复制到EAX寄存器中，指令如下：

```
MOV EAX, ECX
```

把数值1234复制到EDX寄存器中，指令如下：

```
MOV EDX, 1234
```

因为涉及从内存中读写数据，所以接下来有必要了解一下x86常用的几种内存寻址方式，实际上很多指令会涉及内存寻址，不过跟数据传输放在一起讲解更容易理解。

指令中可以直接给出内存地址的偏移量，又称为位移，也可以通过一项或多项数据计算得到一个地址。

（1）Displacement：位移，是一个8位、16位或32位的值。

（2）Base：基址，存放在某个通用寄存器中。

（3）Index：索引，存放在某个通用寄存器中，ESP不可用作索引。

（4）Scale：比例因子，用来与索引相乘，可以取值1、2、4、8。

经过计算得到的地址称为有效地址，计算公式如式（1-1）所示。

$$\text{Effective Address} = \text{Base} + (\text{Index} \times \text{Scale}) + \text{Displacement} \qquad (1\text{-}1)$$

Base、Index和Displacement可以随意组合，任何一个都可以不存在，如果不使用Index也就没有Scale。Index和Scale主要用来寻址数组和多维数组，这里不继续展开。下面简单介绍基于Base和Displacement的寻址。

（1）位移（Displacement）：一个单独的位移表示距离操作数的直接偏移量。因为位移被编码在指令中，所以一般用于编译阶段静态分配的全局变量之类。

（2）基址（Base）：将内存地址存储在某个通用寄存器中，寄存器的值可以变化，所以一般用于运行时动态分配的变量、数据结构等。

（3）基址+位移（Base+Displacement）：基址加位移，尤其适合寻址运行时动态分配的数据结构的字段，以及函数栈帧上的变量。

如下3条汇编指令分别使用位移、基址和基址+位移这3种寻址方式，指令如下：

```
MOV EAX, [16]
MOV EAX, [ESP]
MOV EAX, [ESP + 16]
```

1.2.3 入栈和出栈指令

1.1节在介绍通用寄存器的时候，提到过ESP寄存器有特殊用途，被CPU用作栈指针。x86的一些指令虽然不直接以ESP为操作数，但是会隐式地修改ESP的值，例如入栈和出栈指令。

入栈指令PUSH只有一个操作数，即要入栈的源操作数。PUSH指令会先将ESP向下移动一个位置，然后把源操作数复制到ESP指向的内存处，代码如下：

```
PUSH EAX
```

等价于：

```
SUB ESP, 4
MOV [ESP], EAX
```

最后这个MOV指令把ESP用作基址进行寻址。

出栈指令POP也只有一个操作数，是用来接收数据的目的操作数。POP指令会先把ESP指向的内存处的值复制到目的操作数中，然后把ESP向上移动一个位置，代码如下：

```
POP EAX
```

等价于：

```
MOV EAX, [ESP]
ADD ESP, 4
```

1.2.4 分支跳转指令

x86的指令指针寄存器EIP始终指向下一条将要被执行的指令，但是汇编代码中并不能通过名称直接引用EIP，所以无法通过MOV之类的指令修改EIP的值。有一系列用于进行分支跳转的指令会隐式地修改EIP的值，例如无条件跳转指令JMP。

JMP指令只有一个操作数，可以是一个立即数、通用寄存器或内存位置，通过这个操作数给出了将要跳转到的目的地址，代码如下：

```
//跳转到地址 32 处
JMP 32
//跳转目的地址经由 EAX 给出
JMP EAX
//跳转目的地址经由内存位置给出
JMP [EAX + 32]
```

跳转操作与过程调用不同，不记录返回地址。除了无条件跳转指令，x86还提供了一组条件跳转指令，根据标志寄存器EFLAGS中的不同标志位来决定是否跳转，此处不一一介绍。

1.2.5 过程调用指令

绝大多数的上层语言提供了函数这一语言特性，在汇编语言中被称为过程。x86的过程调用通过CALL指令实现，该指令和跳转指令一样只有一个操作数，也就是过程的起始地址。可以认为CALL在JMP的基础上多了一步记录返回地址的操作，返回地址就是紧随CALL之后的下一条指令的地址。CALL指令先把返回地址入栈，然后跳转到目的地址执行。

目的地址也可以经由一个立即数、通用寄存器或内存位置来给出。假如下一条指令的地址为32，代码如下：

```
CALL EAX
```

等价于：

```
PUSH 32
JMP EAX
```

子过程执行完成后通过RET指令返回，RET指令会从栈上弹出返回地址，并跳转到该地址处继续执行。

RET指令有两种格式，一种没有操作数，只用来完成返回地址弹出和跳转，另一种有一个立即数参数，在上层语言实现某些调用约定时用来调整栈指针，代码如下：

```
RET 8
```

等价于：

```
RET
ADD ESP, 8
```

远调用（Call far）和远返回在上层语言中基本不会用到，这里不予介绍。

1.3 内存分页机制

1.3.1 线性地址

在DOS时代，应用程序直接访问物理内存，代码中的地址都是实际的物理内存地址。任何程序都有权读写所有的物理内存，稍有不慎就会覆盖掉其他程序的代码或数据，连操作系统内核也无法自保。随着80386芯片的到来，PC进入了保护模式，并且开启了内存分页模式，通过特权级和进程地址空间隔离机制，解决了上述问题。如今，主流的操作系统采用分页的方式管理内存。

在分页模式下，应用程序中使用的地址被称为线性地址，需要由MMU（Memory Management Unit）基于页表映射转换为物理地址，整个转换过程对于应用程序是完全透明的。

1.3.2 80386两级页表

80386架构的线性地址的宽度为32位，所以可以寻址4GB大小的空间，与进程的地址空间大小相对应。地址总线为32位，硬件可以寻址4GB的物理内存。分页机制将每个物理内存页面的大小设定为4096字节，并按照4096对齐。

因为每个页面的大小为4096字节，并且地址总线的宽度为32位，所以每个页面中正好可存储1024个物理页面的地址。完整的页表结构的第一层是1个页目录页面，其物理地址存储在CR3寄存器中，通过页目录页面进一步找到第二层的1024个页表页面。

32位的线性地址被MMU按照10位+10位+12位划分，整个地址转换过程如图1-2所示。前两个10位的取值范围都是0～1023，分别用作页目录和页表的索引。最后的12位，取值范围为0～4095，用作最终的页面内偏移。

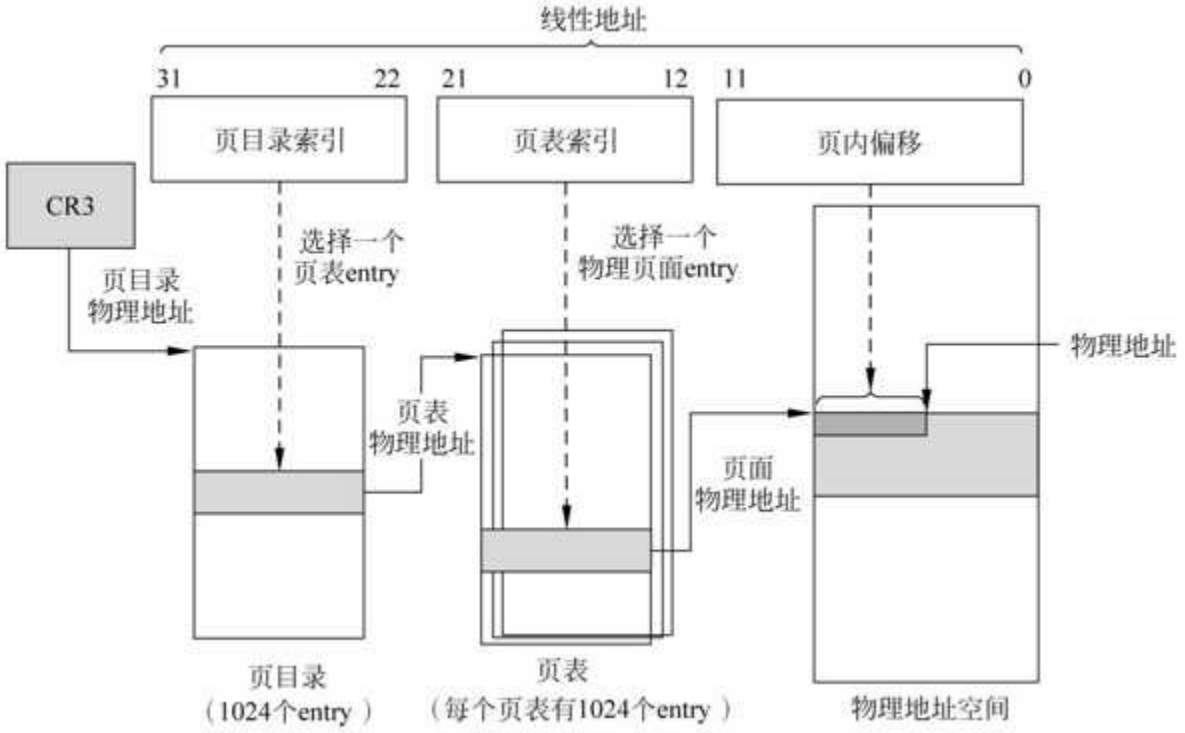

图1-2 80386线性地址到物理地址的转换

1.3.3 PAE三级页表

80386架构的线性地址的宽度为32位，每个进程拥有4GB的线性地址空间。主流操作系统一般按照2：2或3：1的方式进一步将进程的4GB地址空间划分为用户空间和内核空间。因为内核只有一份，

所以内核占用的这组物理页面由所有进程共享，而每个进程独享自己2GB或3GB的用户空间，即所谓的进程地址空间隔离就是通过进程独立的页表实现的，然而硬件32位的地址总线只能寻址4GB的物理内存，在多进程的操作系统上，每个进程实际能够映射到的物理页面远远不足2GB。在这种情况下，Intel推出了物理地址扩展技术（Physical Address Extension，PAE）。

PAE将地址总线拓展到36位，从而使硬件能够寻址多达64GB的物理内存。线性地址的宽度仍然是32位，MMU的页表映射机制需要进行相应调整，以支持从32位线性地址到36位物理地址的映射。

为了支持36位的物理地址，页目录和页表中的地址项被调整为64位，一个页面只能存储512个地址。MMU将32位的线性地址按照2位+9位+9位+12位划分，整个地址转换过程如图1-3所示，在页目录之前又加了一层页目录指针，总共三级页表映射。高两位用来选择一个页目录，接下来的9位用来选择一个页表，再用9位来选择一个物理页面，加上最后12位的偏移值，最终确定一个物理地址。

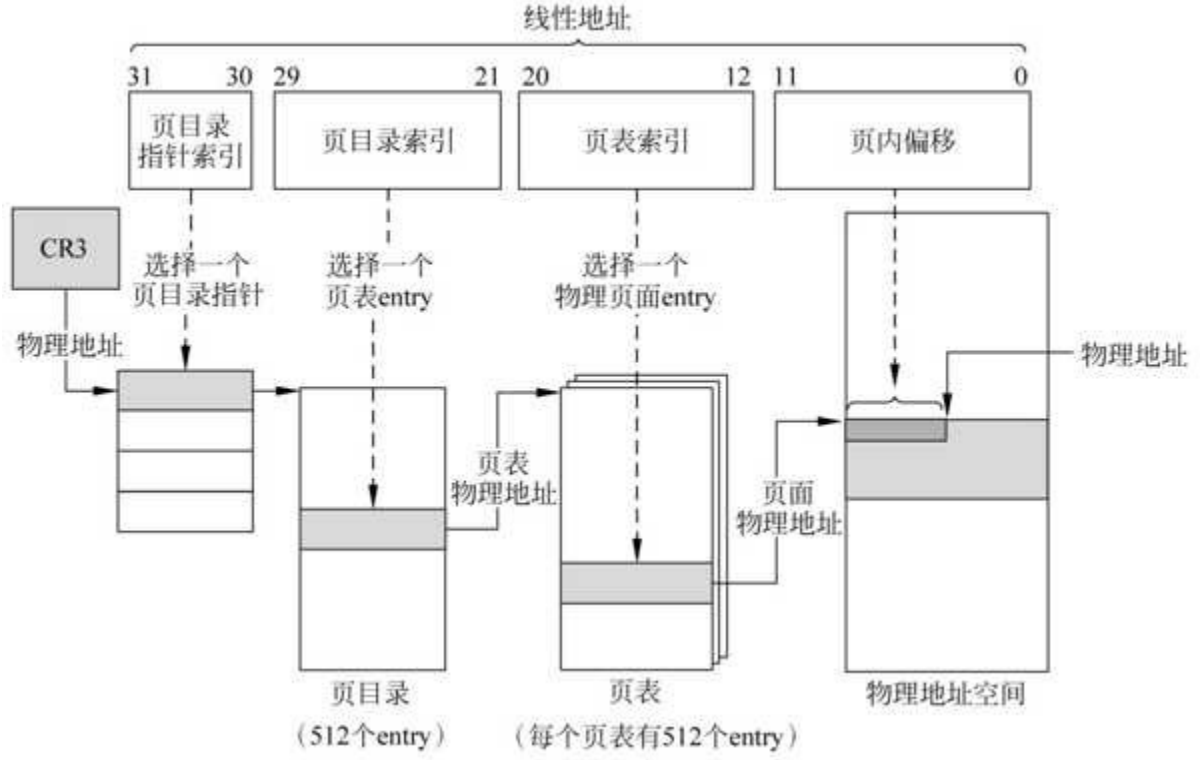

图1-3 PAE线性地址到物理地址的转换

1.3.4 x64四级页表

通过PAE技术，虽然硬件支持的物理内存变大了，但进程的地址空间大小并没有变化。对于某些类型的程序，例如数据库程序，进程2～3GB的用户地址空间成为明显的瓶颈，而且32位的数据宽度也无法满足时下的计算需求，所以64位架构应运而生了。

Intel推出的IA64架构因为与原来的x86架构不兼容，所以没能普及，而AMD公司通过扩展x86推出的x64架构，因为良好的向下兼容性而被广泛采用。常见的x64、x86_64都是指amd64架构，如今的个人计算机基本是基于amd64架构的。

在amd64上，寄存器的宽度变成了64位，而线性地址实际只用到48位，也就是最大可寻址256TB的内存。很少有单台计算机会安装如此大量的内存，所以没有必要实现48位的地址总线，常见的个人计算机的CPU的地址总线实际还不到40位，例如笔者的计算机的Core i7实际只有36位。服务器的CPU的地址总线的宽度会更大，例如Xeon E5系列能达到46位。

amd64在PAE的基础上进一步把页表扩展为四级，每个页面的大小仍然是4096字节，MMU将48位的线性地址按照9位+9位+9位+9位+12位划分，整个地址转换过程如图1-4所示。高9位选择一个页

目录指针表，再用9位选择一个页目录，接下来的两个9位分别用于选择页表和物理页面，最后的12位依然用作页内偏移值。

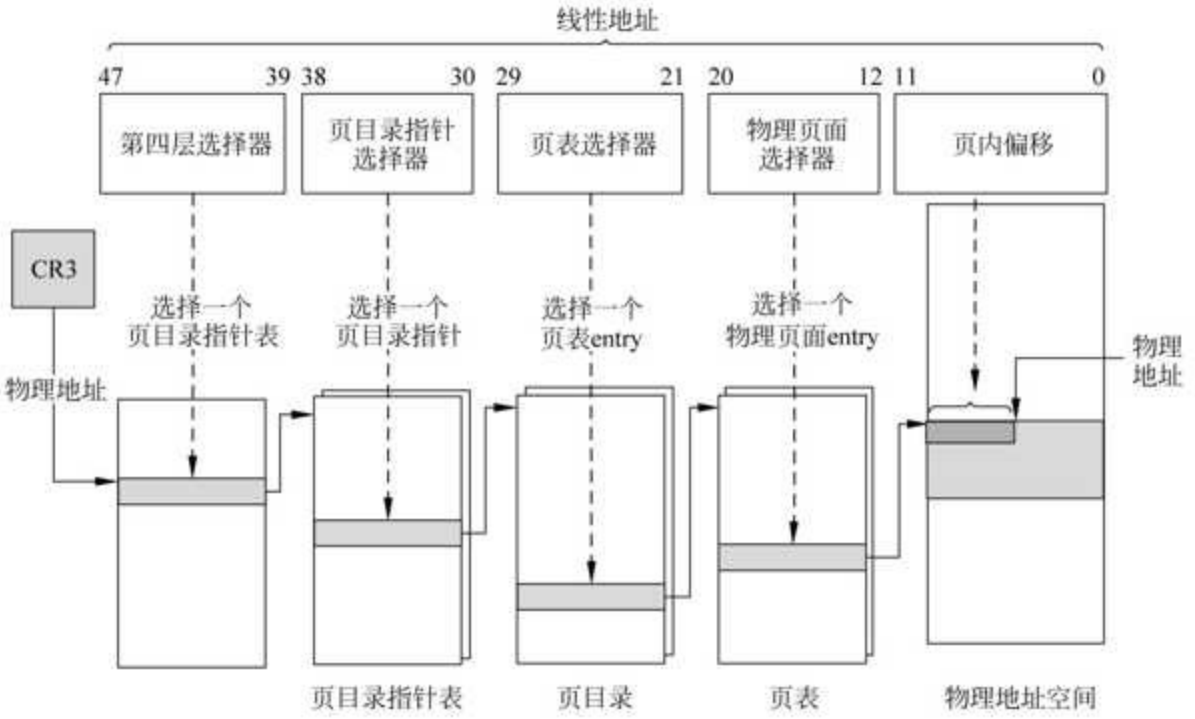

图1-4 amd64线性地址到物理地址的转换

1.3.5 虚拟内存

乍看起来，完整的页表结构会占用大量的内存，例如在80386上就会占用1+1024＝1025个物理页面。因为页目录本身也被用作页表，所以实际上是1024个页面，总共占用4096×1024＝4MB的空间。因为系统空间是所有进程共享的，所以对应的页表也是共享的，而大多数进程并不会申请大量的用户空间内存，用不到的页表也不会被分配，所以进程的页表是稀疏的，并不会占用太多的内存。

进程是以页面为单位向操作系统申请内存的，操作系统一般只是对进程已申请的区间进行记账，并不会立刻映射所有页面。等到进程真正去访问某个未映射的页面时，才会触发Page Fault异常，操作系统注册的Page Fault Handler会检查内存记账：如果目标地址已申请，就是合法访问，系统会分配一个物理页面并完成映射，然后恢复被中断的程序，程序对这一切都是无感的；如果目标地址未申请，就是非法访问，系统一般会通过信号、异常等机制结束目标进程。

当物理内存不够用的时候，操作系统可以把一些不常使用的物理页面写到磁盘交换分区或交换文件，从而能够将空出的页面给有需要的进程使用。当被交换到磁盘的页面再次被访问时，也会触发Page Fault，由Page Fault Handler负责从交换分区把数据加载回内存。程序对这一切都是无感的，并不知道某个内存页面到底是在磁盘上，还是在物理内存中，所以称为进程的虚拟内存。

1.4 汇编代码风格

Go语言使用的汇编代码风格跟最常见的Intel风格和AT&T风格都不太相同，根据官方文档的说法，是基于Plan 9汇编器的风格做了一些调整。本节简单对比Go汇编和Intel汇编的风格差异。

1. 操作数的宽度

在Go汇编中通过指令的后缀来判断操作数的宽度，后缀W代表16位，后缀L代表32位，后缀Q代表64位，不像Intel汇编中有AX、EAX、RAX不同的寄存器名称。例如对于整数自增指令，Intel汇编风格的代码如下：

```
INC EAX
INC RCX
```

对应的Go汇编风格的代码如下：

```
INCL AX
INCQ CX
```

2. 操作数的顺序

对于常见的有两个操作数的指令，Go汇编中操作数的顺序与Intel汇编中操作数的顺序是相反的，源操作数在前而目的操作数在后。

例如Intel汇编的代码如下：

```
MOV EAX, ECX
```

转换成Go汇编的代码如下：

```
MOVL CX, AX
```

3. 地址的表示

有效地址的计算公式如式（1-1）所示，如果要用ESP作为基址寄存器，EBX作为索引寄存器，比例系数取2，位移为16，则可以分别给出两种风格的代码。

Intel汇编的代码如下：

```
[ESP + EBX * 2 + 16]
```

Go汇编的代码如下：

```
16(SP)(BX * 2)
```

4. 立即数格式

Go汇编中的立即数类似于AT&T风格的立即数，需要加上$前缀。

Intel汇编的代码如下：

```
MOV EAX, 1234
```

Go汇编的代码如下：

```
MOVL $1234, AX
```

1.5 本章小结

本章简单介绍了x86架构的通用寄存器、内存寻址方式和比较关键的几组指令。了解了操作系统以页面为单位的内存管理机制，以及分页模式下线性地址到物理地址的映射过程。还简要地对比了Go汇编风格与Intel汇编风格的几点不同。鉴于后续章节中将会经常用到反汇编技术来探索Go语言的特性，所以本章内容旨在让读者掌握必要的汇编基础知识。

第2章
指针

指针凭借其灵活强大的内存操作能力，在C和C++中扮演着非常重要的角色，但也因一些常见的安全问题给人们带来很多困扰。指针在Go语言中被保留了下来，但是影响力似乎大大降低了，出于安全方面的考虑，指针运算等一些重要特性被移除，使指针显得不再那么重要。在学习过程中，有很多人对值类型、指针或引用类型，以及值和地址这些概念感到困惑。本章从指针的构成出发，首先理解指针的本质，然后逐一分析指针的常见操作的实现原理，以及常见的问题和解决方法，最后介绍关于Go语言unsafe包的一些思考和实践。

2.1 指针构成

在Go语言中，声明一个指针变量的示例代码如下：

```
var p * int
```

变量名为p，其中的*int为变量的类型。对*int进一步拆解，*表明了p是一个指针变量，用来存储一个地址，而int是指针的元素类型，也就是当p中存了一个有效地址的时候，该地址处的内存会被解释为int类型。

无论指针的元素类型是什么，指针变量本身的格式都是一致的，即一个无符号整型，变量大小能够容纳当前平台的地址。例如在386架构上是一个32位无符号整型，在amd64架构上是一个64位无符号整型。

有着不同元素类型的指针被视为不同类型，这是语言设计层面强加的一层安全限制，因为不同的元素类型会使编译器对同一个内存地址进行不同的解释。

2.1.1 地址

在Go语言中，一个有效的地址就是一个无符号整型数值，运行阶段用来在进程的内存地址空间中确定一个位置。

在第1章中简单地介绍了x86的几种常用寻址方式，指针一般会用到基址+位移的寻址方式。例如当指针元素的类型为int时，通过指针访问int元素的代码被编译成汇编指令，就是将某个通用寄存器用作基址进行寻址。

在amd64架构下通过go build命令编译一个示例，代码如下：

```
//第 2 章/code_2_1.go
package main

func main() {
    n := 10
    println(read(&n))
}

//go:noinline
func read(p * int) (v int) {
    v = * p
    return
}
```

使用Go自带的objdump工具反编译main.read（）函数，得到的汇编代码如下：

```
$ go tool objdump -S -s 'main.read' gom.exe
TEXT main.read(SB) C:/gopath/src/fengyoulin.com/gom/code_2_1.go
        v = *p
  0x488ee0              488b442408              MOVQ 0x8(SP), AX
  0x488ee5              488b00                  MOVQ 0(AX), AX
        return
  0x488ee8              4889442410              MOVQ AX, 0x10(SP)
  0x488eed              c3                      RET
```

在第一条MOVQ指令中，第1个操作数0x8（SP）表示参数p在栈上的地址，关于函数栈帧布局，将会在第3章中详细介绍，目前只要理解这条指令的作用是把参数p中存储的地址值复制到AX寄存器中即可。

在第二条MOVQ指令中，第1个操作数使用AX作为基址加上位移0，也就是用基址+位移的方式寻址指针p指向的数据，所以这条指令的作用就是把目标地址处的值复制到AX中。

在第三条MOVQ指令中，第2个操作数0x10（SP）表示栈上返回值的地址，所以这条指令的作用就是把AX中存储的值复制到返回值v中。

经过上面三条指令，便可成功地把指针p指向的数据复制到函数返回值空间。

2.1.2 元素类型

指针本身就是个无符号整型，这一点不会因不同的元素类型而有所不同，而元素类型会影响编译器如何对指针中存储的地址进行解释，这一点也可以通过汇编代码进行验证。

把第2章/code_2_1.go中read（）函数修改为read32（）函数，其主要目的是改变参数和返回值的类型，代码如下：

```
//第 2 章/code_2_2.go
//go:noinline
func read32(p *int32) (v int32) {
    v = *p
    return
}
```

修改后的代码重新进行编译和反编译，得到的汇编代码如下：

```
$ go tool objdump -S -s 'main.read32' gom.exe
TEXT main.read32(SB) C:/gopath/src/fengyoulin.com/gom/code_2_2.go
        v = *p
  0x488f30              488b442408              MOVQ 0x8(SP), AX
  0x488f35              8b00                    MOVL 0(AX), AX
        return
  0x488f37              89442410                MOVL AX, 0x10(SP)
  0x488f3b              c3                      RET
```

可以看到第一条用于复制指针存储的地址的指令没有发生变化。第二条指令中的内存寻址单元0（AX）也没有变，而原本后两条MOVQ指令现在变成了MOVL，表明复制的数据长度发生了变化，从8字节变成了4字节。造成这一变化的原因正是指针元素类型从int变成了int32。

2.2 相关操作

本节分析指针常见操作及其底层实现原理，也会介绍指针所引发的那些广受诟病的问题，以及在Go语言中如何解决这些问题。此外，一些指针特性受限于安全问题，在Go语言中不能直接使用，在本节也会探讨一些替代方案及背后的思考。

2.2.1 取地址

指针中存储的是地址，而地址一般通过取地址运算符获得，或者在动态分配内存时由new之类的函数返回。在Go语言中取地址运算符与C语言相比似乎没什么变化，编译器会确保应用取地址运算符的变量类型与指针的元素类型是一致的。下面仍然通过反编译一个简单的函数，来看一下取地址运算符到底做了什么。

在amd64架构下通过go build命令编译一个示例，代码如下：

```
//第 2 章/code_2_3.go
package main

var n int

func main() {
    println(addr())
}

//go:noinline
func addr() (p *int) {
    return &n
}
```

反编译main.addr（）函数得到的代码如下：

```
$ go tool objdump -S -s 'main.addr' gom.exe
TEXT main.addr(SB) C:/gopath/src/fengyoulin.com/gom/code_2_3.go
        return &n
  0x488f90          488d05691f0f00          LEAQ main.n(SB), AX
  0x488f97          4889442408              MOVQ AX, 0x8(SP)
  0x488f9c          c3                      RET
```

其中LEAQ指令的作用就是取得main.n的地址并装入AX寄存器中。后面的MOVQ指令则把AX的值复制到返回值p。

这里获取的是一个包级别变量n的地址，等价于C语言的全局变量，变量n的地址是在编译阶段静态分配的，所以LEAQ指令通过位移寻址的方式得到了main.n的地址。LEAQ同样也支持基于基址和索引获取地址，具体可参考第1章所介绍的x86寻址方式。

在C语言中，不应该将函数内某个局部变量的地址作为返回值返回，虽然编译器允许这样的代码通过编译，但在代码逻辑上却属于明显的Bug。因为函数一旦返回，栈帧随即销毁，这部分内存会被后续的函数栈帧覆盖，所以通过返回的指针读写栈上的数据就可能会造成程序异常崩溃，虽然也有可能不会崩溃，但是基于错误的数据继续运行下去，会变得更加难以调试和排查。

在Go语言中，通过逃逸分析机制避免了此类问题。来看一个示例，代码如下：

```
//第 2 章/code_2_4.go
//go:noinline
func newInt() (p * int) {
    var n int
    return &n
}
```

其中变量n实际上是在堆上分配的，因为n逃逸到堆上，所以即使newInt（）函数返回，函数栈帧销毁，也不会影响后续正常使用n的指针。待到第3章介绍函数时再进一步介绍逃逸。

2.2.2 解引用

通过指针中的地址去访问原来的变量，就是所谓的指针解引用。在2.1.1节已经通过反编译验证了指针的解引用过程，就是把地址存入某个通用寄存器，然后用作基址进行寻址。接下来就介绍一下C语言中与指针解引用相关的几个常见问题，以及这些问题在Go语言中是如何解决的。

1. 空指针异常

所谓空指针，就是地址值为0的指针。按照操作系统的内存管理设计，进程地址空间中地址为0的内存页面不会被分配和映射，保留地址0在程序代码中用作无效指针判断，所以对空指针进行解引用操作就会造成程序异常崩溃，程序代码在对指针进行解引用前，始终要确保指针非空，因而需要添加必要的判断逻辑。

所以遭遇空指针异常并非语言设计方面的缺陷，而是程序逻辑上的Bug。Go语言中对空指针进行解引用会造成程序panic（宕机）。

2. 野指针问题

野指针问题一般是由于指针变量未初始化造成的。众所周知，C语言中声明的变量需要显式地初始化，否则就是内存中上次遗留的随机值。对于未初始化的指针变量而言，如果内存中的随机值非零，就会使指针指向一个随机的内存地址，而且会绕过代码中的空指针判断逻辑，从而造成内存访问错误。

为了解决C语言变量默认不初始化带来的各种问题，Go语言中声明的变量默认都会初始化为对应类型的零值，指针类型变量都会初始化为nil，而代码中的空指针判断逻辑能够避免空指针异常，从而使问题得到解决。

3. 悬挂指针问题

在C语言中，程序员需要手动分配和释放内存，而所谓悬挂指针问题，就是指程序过早地释放了内存，而后续代码又对已经释放的内存进行访问，从而造成程序出现错误或异常。

Go语言实现了自动内存管理，由GC负责释放堆内存对象。GC基于标记清除算法进行对象的存活分析，只有明确不可达的对象才会被释放，因此悬挂指针问题不复存在。

2.2.3 强制类型转换

基于指针的强制类型转换非常高效，因为不会生成任何多余的指令，也不会额外分配内存，只是让编译器换了一种方式来解释内存中的数据。出于安全方面的考虑，Go语言不建议频繁地进行指针强制类型转换。两种不同类型指针间的转换需要用unsafe.Pointer作为中间类型，unsafe.Pointer可以和任意一种指针类型互相转换。

在amd64架构下反编译一个函数，代码如下：

```
//第 2 章/code_2_5.go
//go:noinline
func convert(p * int) {
    q := (* int32)(unsafe.Pointer(p))
    *q = 0
}
```

得到汇编代码如下：

```
$ go tool objdump -S -s 'main.convert' gom.exe
TEXT main.convert(SB) C:/gopath/src/fengyoulin.com/gom/code_2_5.go
        *q = 0
  0x488fa0          488b442408          MOVQ 0x8(SP), AX
  0x488fa5          c70000000000        MOVL $0x0, 0(AX)
}
  0x488fab          c3                  RET
```

把指针的类型强转换为int32后，原本的MOVQ指令变成了MOVL，没有产生任何额外指令，所以转换效率是非常高的。

2.2.4 指针运算

在C语言中，指针和不指定长度的数组，在元素类型相同的情况下是可以等价使用的，指针加上一个整数n等价于取数组中下标为n的元素的地址。指针可以进行加减运算，给操作多维数组带来了很大方便，但也经常会造成内存访问越界问题。

Go语言中的数组必须指定长度，并且是值类型，与指针不再等价，指针运算也不再支持，这些都是出于安全考虑的。数组的长度在编译时期能够确定，编译器可以生成代码检测下标越界问题，而指针则不然，编译器无法确定指针运算的安全边界，所以无法保证其安全性。

Go语言的slice集成了数组和指针的优点，既能像指针那样关联一个可以动态增长的Buffer，又能像数组那样让编译器生成下标越界检测代码，在某些场合可以考虑用slice代替指针运算。

如果还想像C语言中那样直接进行指针运算，就需要借助unsafe.Pointer进行转换。2.2.3节中已经提到unsafe.Pointer可以与任何一种指针类型互相转换，除此之外unsafe.Pointer还可以与uintptr互相转换，而后者可以进行整数运算。

假如有一个元素类型为int的指针p，要把p移动到下一个int的位置，在C语言中可以通过指针的自增运算实现，代码如下：

```
++p;
```

在Go语言中等价的代码如下：

```
p = (* int)(unsafe.Pointer(uintptr(unsafe.Pointer(p)) + unsafe.Sizeof(* p)))
```

在Go语言中实现此功能就显得有些烦琐了，先把p转换为unsafe.Pointer类型，再进一步转换为uintptr类型，然后加上一个int的大小，再转换回unsafe.Pointer类型，最终转换为*int类型。

2.3 unsafe包

本节简单地介绍Go语言的unsafe包，在2.2节中已经用到了unsafe.Pointer进行指针的强制类型转换和指针运算，实际上就是人为地干预编译器对内存地址的解释方式，这些能力对于研究语言的底层实现来讲是不可或缺的。

代码中用好unsafe，能够优化程序的性能，想必很多人都见过经典的类型转换，代码如下：

```
func convert(s []byte) string {
    return *(*string)(unsafe.Pointer(&s))
}
```

Slice Header结构只是比String Header结构多了一个容量字段，相当于内嵌了一个String Header，如图2-1所示。

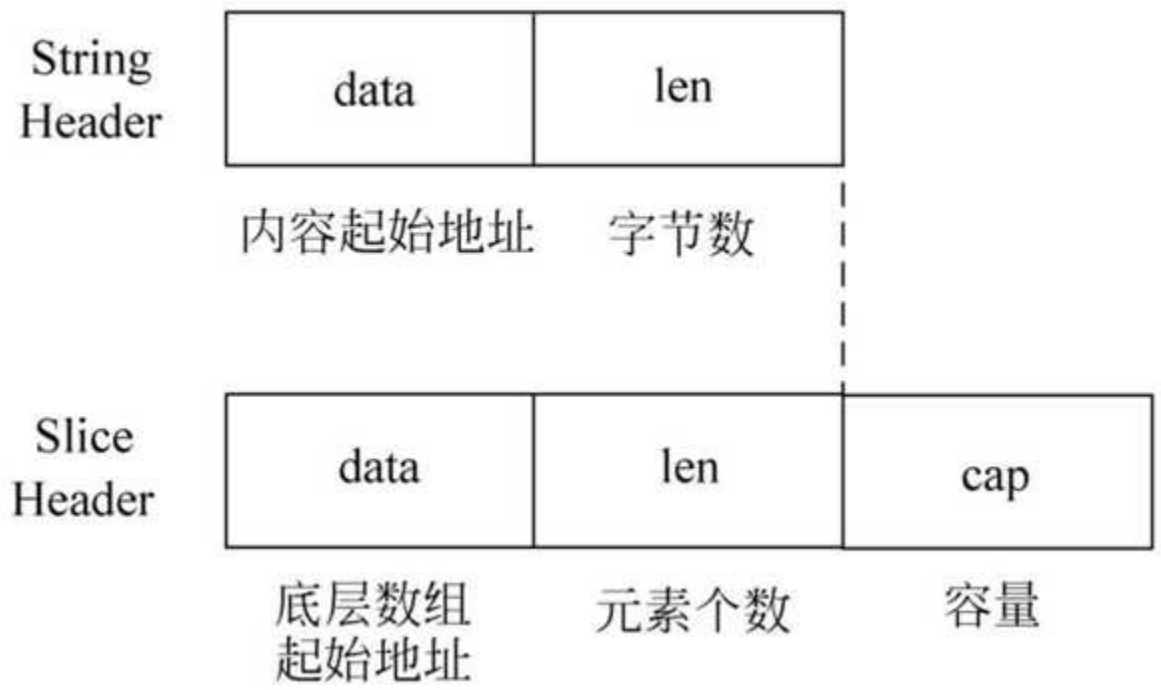

图2-1 String Header和Slice Header的结构

用这种强制类型转换的方式可以避免额外的内存分配，从而减少程序的开销，但是也会带来一些风险。因为按照Go语言的设计思想，string的内容是不可修改的，但是slice元素是可以修改的，基于上述方法得到的string与原来的slice共享底层Buffer，如果不经意修改了slice就可能会造成程序逻辑错误。

根据官方文档的说法，unsafe包包含的操作绕过了Go语言的类型安全机制，使用unsafe包会造成程序不可移植，并且不受Go 1兼容性准则的保护。那么unsafe到底该不该用呢？本节就围绕这个问题进行一些分析研究。

2.3.1 标准库与keyword

本节主要分析unsafe包的本质，到底是标准库还是一组keyword。这个思考源于2.2.4节进行指针运算时用到的unsafe.Sizeof，而sizeof在C语言中是个关键字。先从源码入手，梳理unsafe包都提供了些什么，代码如下：

```
//一个"任意类型"定义
type ArbitraryType int
//指针类型定义
type Pointer * ArbitraryType
//3 个工具函数原型(只有原型,没有实现)
func Sizeof(x ArbitraryType) uintptr
func Offsetof(x ArbitraryType) uintptr
func Alignof(x ArbitraryType) uintptr
```

根据源码中的注释，ArbitraryType在这里只是用于文档目的，实际上并不属于unsafe包，它可以表示任意的Go表达式类型。Sizeof（）函数用来返回任意类型的大小，Offsetof（）函数用来返回任意结构体类型的某个字段在结构体内的偏移，而Alignof（）函数用来返回任意类型的对齐边界，最重要的是这3个函数的返回值都是常量。

基于上述信息，已经可以断定unsafe并不是一个真实的包，unsafe提供的这些能力不是标准库层面能够实现的。指针强制类型转换本来就是在编译阶段实现的，而Sizeof（）函数、Offsetof（）函数和Alignof（）函数返回的是常量值，也就要求返回值必须在编译阶段确定，所以必须由编译器直接支持。可以通过实验进行验证，代码如下：

```
//第 2 章/code_2_6.go
//go:noinline
func size() (o uintptr) {
    o = unsafe.Sizeof(o)
    return
}
```

在amd64平台，反编译size（）函数得到汇编代码如下：

```
$ go tool objdump -S -s 'main.size' gom.exe
TEXT main.size(SB) C:/gopath/src/fengyoulin.com/gom/code_2_6.go
        return
  0x488fb0          48c744240808000000      MOVQ $0x8, 0x8(SP)
  0x488fb9          c3                      RET
```

这条MOVQ指令直接向返回值o中写入了立即数8，也就说明Sizeof（）函数在编译阶段就被转换成了立即数，与C语言中的sizeof并无区别。上述测试方法同样适用于Offsetof（）函数和Alignof（）函数。

既然这些都是由编译器直接支持的，本质上跟keyword一样，为什么Go语言要放到unsafe包中呢？根本原因还是出于安全考虑。直接的任意操作内存的能力可以让程序员写出更高效的代码，但是也因为过于灵活而让编译器无法落实安全检查，从而使程序变得不安全。unsafe这个名字就旨在提醒程序员，内存操作有风险，要谨慎！

2.3.2 关于**uintptr**

很多人都认为uintptr是个指针，其实不然。不要对这个名字感到疑惑，它只不过是个uint，大小与当前平台的指针宽度一致。因为unsafe.Pointer可以跟uintptr互相转换，所以Go语言中可以把指针转换为uintptr进行数值运算，然后转换回原类型，以此来模拟C语言中的指针运算。

需要注意的是，不要用uintptr来存储堆上对象的地址。具体原因和GC有关，GC在标记对象的时候会跟踪指针类型，而uintptr不属于指针，所以会被GC忽略，造成堆上的对象被认为不可达，进而被释放。用unsafe.Pointer就不会存在这个问题了，unsafe.Pointer类似于C语言中的void*，虽然未指定元素类型，但是本身类型就是个指针。

2.3.3 内存对齐

硬件的实现一般会将内存的读写对齐到数据总线的宽度，这样既可以降低硬件实现的复杂度，又可以提升传输的效率。有些硬件平台允许访问未对齐的地址，但是会带来额外的开销，而有的硬件平台不支持访问未对齐的地址，当遇到未对齐的地址时会直接抛出异常。鉴于这些原因，编译器在定义数据类型时，还有runtime在分配内存时，都要进行对齐操作。

Go语言的内存对齐规则参考了两方面因素：一是数据类型自身的大小，复合类型会参考最大成员大小；二是硬件平台机器字长。

机器字长是指计算机进行一次整数运算所能处理的二进制数据的位数，在x86平台可以理解成数据总线的宽度。当数据类型自身大小小于机器字长时，会被对齐到自身大小的整数倍；当自身大小大于机器字长时，会被对齐到机器字长的整数倍。

通过unsafe.Sizeof（）函数和unsafe.Alignof（）函数可以得到目标数据类型的大小和对齐边界，表2-1给出了常见内置类型的大小和对齐边界。

表2-1 常见内置类型的大小和对齐边界

类　型	32位平台		64位平台	
	大小	对齐边界	大小	对齐边界
bool	1	1	1	1
int8、uint8	1	1	1	1
int16、uint16	2	2	2	2
int32、uint32、float32	4	4	4	4
int64、uint64、float64	8	4	8	8
int、uint、uintptr	4	4	8	8
complex64	8	4	8	4
complex128	16	4	16	8
string	8	4	16	8
slice	12	4	24	8
map	4	4	8	8

complex类型由实部和虚部两个float组成，complex64相当于[2]float32，complex128相当于[2]float64，所以对齐边界分别与float32、float64一致。

map多数情况下会被分配在堆上，本地只有一个指针指向堆上的数据结构，而指针的对齐边界自然与uintptr相同。

string和slice的结构定义可参考reflect.StringHeader与reflect.SliceHeader，代码如下：

```
type StringHeader struct {
    Data uintptr
    Len int
}
type SliceHeader struct {
    Data uintptr
    Len int
    Cap int
}
```

它们的对齐边界与其最大的成员，即类型uintptr的对齐边界相同。值得强调的是对于struct而言，每个成员都会以结构体的起始地址为基地址，按自身类型的对齐边界对齐。除此之外，整个struct还

要按照成员中最大的对齐边界进行对齐，所以编译器会按需要在结构体相邻成员之间及最后一个成员之后添加padding，因此需要合理地排列数据成员的顺序，从而使整个struct的空间占用最小化。

来看一个示例，代码如下：

```
//第 2 章/code_2_7.go
type s1 struct {
    a int8
    b int64
    c int8
    d int32
    e int16
}
```

数据类型s1在amd64架构上占用了32字节空间，如图2-2所示，在a和b之间有7字节的padding，目的是让成员b对齐到8。c和d之间有3字节的padding，为的是让成员d对齐到4。又因为整个struct的成员中最大的对齐边界为int64对应的8，所以e之后还有6字节的padding，使整个结构体对齐到8，但是这样总共浪费了16字节空间，空间利用率只有50%。

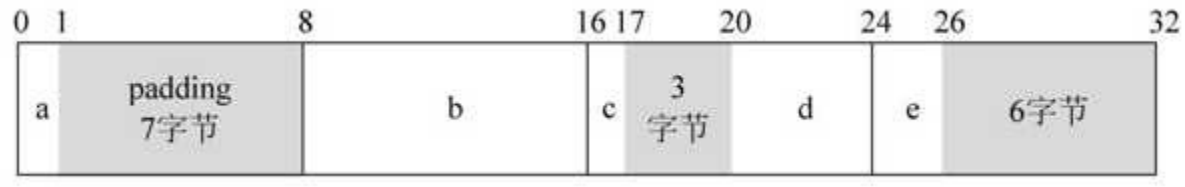

图2-2 s1内存布局

接下来通过调整结构体成员的位置，尽量避免编译器添加padding，调整后的代码如下：

```
//第 2 章/code_2_8.go
type s2 struct {
    a int8
    c int8
    e int16
    d int32
    b int64
}
```

如图2-3所示，数据类型s2和之前的s1有着相同类型的5个数据成员，但是经过人为优化成员的顺序后，编译器没有添加任何padding，整个struct占用了16字节空间，利用率达到100%。

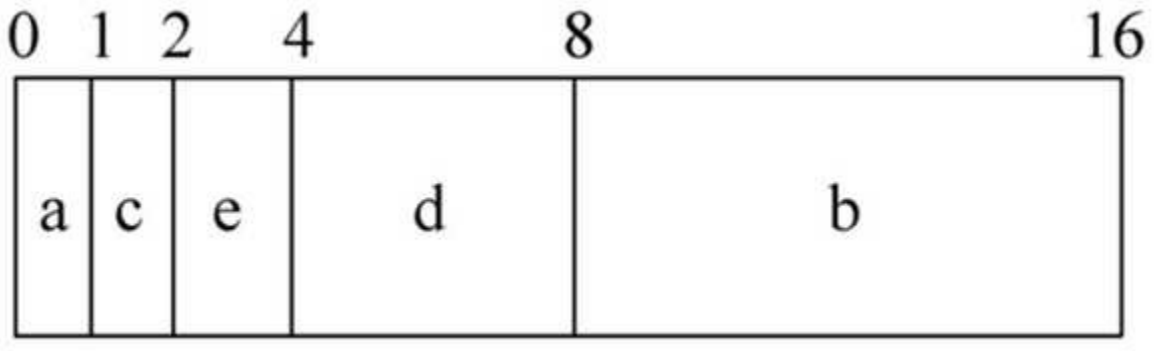

图2-3 s2内存布局

2.4 本章小结

本章首先从指针的构成开始讲解，通过反汇编的方式，展示了编译器如何使用指针存储的地址进行内存寻址，以及元素类型对指令生成的影响。后续又介绍了与指针相关的操作、常见问题和解决方法。最后结合指针强制类型转换的实例介绍了unsafe包，并通过unsafe的实际应用，了解了内存对齐的原理。在接下来对Go语言特性的探索中，unsafe也会起到非常重要的作用。

第3章
函数

函数在主流的编程语言中是一个基础且重要的特性。通过函数对逻辑单元进行封装，使代码结构更加清晰，便于实现代码复用，基于函数的编译链接技术让构建大型应用程序更为方便。也正因为函数太过于基础，所以很多人对于其底层细节并不甚关心，在实际应用中便会遇到一些问题。本章从函数的底层实现开始研究，逐步梳理Go语言中与函数相关的特性，旨在理解其背后的设计思想。

从代码结构来看，层层函数调用就是一个后进先出的过程，与数据结构中的入栈出栈操作完全一致，所以非常适合用栈来管理函数的局部变量等数据。x86架构提供了对栈的支持，本书第1章汇编基础部分介绍了栈指针寄存器SP，以及入栈出栈对应的指令。x86还通过CALL指令和RET指令实现了对过程的支持（汇编语言中的过程等价于Go语言中的函数）。下面就先从CPU的视角，看一下函数调用的过程。

CPU在执行程序时，IP寄存器会指向下一条即将被执行的指令，而SP寄存器会指向栈顶。图3-1为下一条指令即将调用函数f1（）函数的场景。

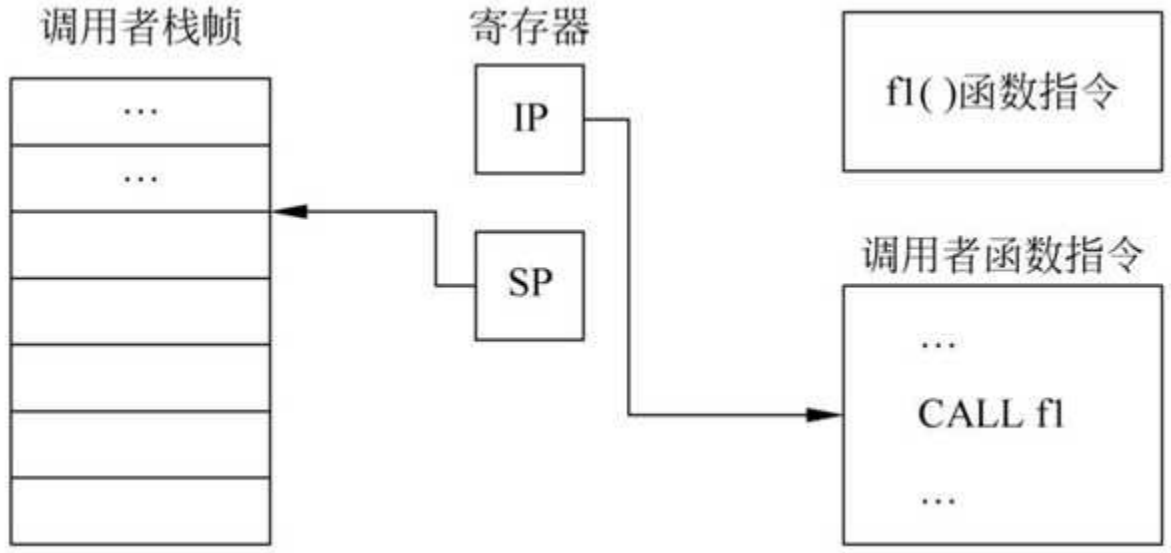

图3-1 函数调用发生前

f1（）函数的调用由CALL指令实现。CALL指令会先把下一条指令的地址压入栈中，这就是所谓的返回地址，然后会跳转到f1（）函数的地址处执行。当f1（）函数执行完成后会返回CALL指令压栈的返回地址处继续执行。由于CALL指令引发了入栈操作和指令跳转，所以SP和IP寄存器的值都发生了改变，如图3-2所示。

当f1（）函数执行到最后时会有一条RET指令。RET指令会从栈上弹出返回地址，然后跳转到该地址处继续执行，如图3-3所示，注意SP和IP寄存器的改变。

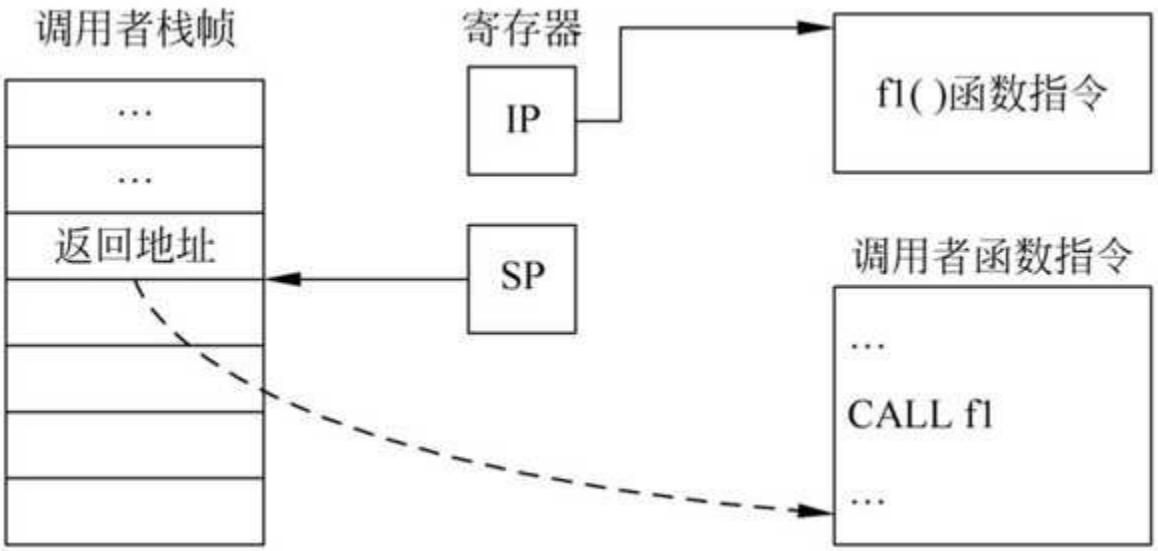

图3-2 CALL指令执行后

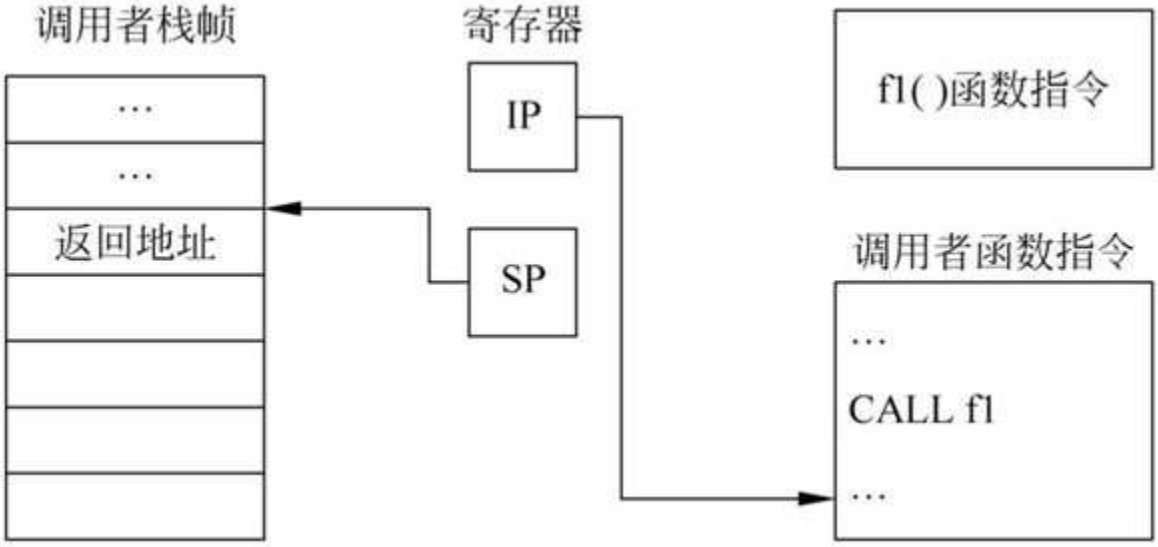

图3-3 RET指令执行后

这里只是简单地演示了一次函数调用中指令流的跳转与返回，更多细节将在本章后续内容中展开。

3.1 栈帧

在一个函数的调用过程中，栈不只被用来存放返回地址，还被用来传递参数和返回值，以及分配函数局部变量等。随着每一次函数调用，都会在栈上分配一段内存，用来存放这些信息，这段内存就是所谓的函数栈帧。

3.1.1 栈帧布局

实际管理栈帧的是函数自身的代码，也就是说由编译器生成的指令负责栈帧的分配与释放。栈帧的布局也是由编译器在编译阶段确定的，其依据就是函数代码，所以也可以说函数栈帧是由编译器管理的。一个典型的Go语言函数栈帧如图3-4所示。

参照上面的函数栈帧布局示意图，从空间分配的角度来看，函数的栈帧包含以下几部分。

（1）return address：函数返回地址，占用一个指针大小的空间。实际上是在函数被调用时由CALL指令自动压栈的，并非由被调用函数分配。

（2）caller’s BP：调用者的栈帧基址，占用一个指针大小的空间。用来将调用路径上所有的栈帧连成一个链表，方便栈回溯之类的操作，只在部分平台架构上存在。函数通过将栈指针SP直接向下移动指定大小，一次性分配caller’s BP、locals和args to callee所占用的空间，在x86架构上就是使用SUB指令将SP减去指定大小的。

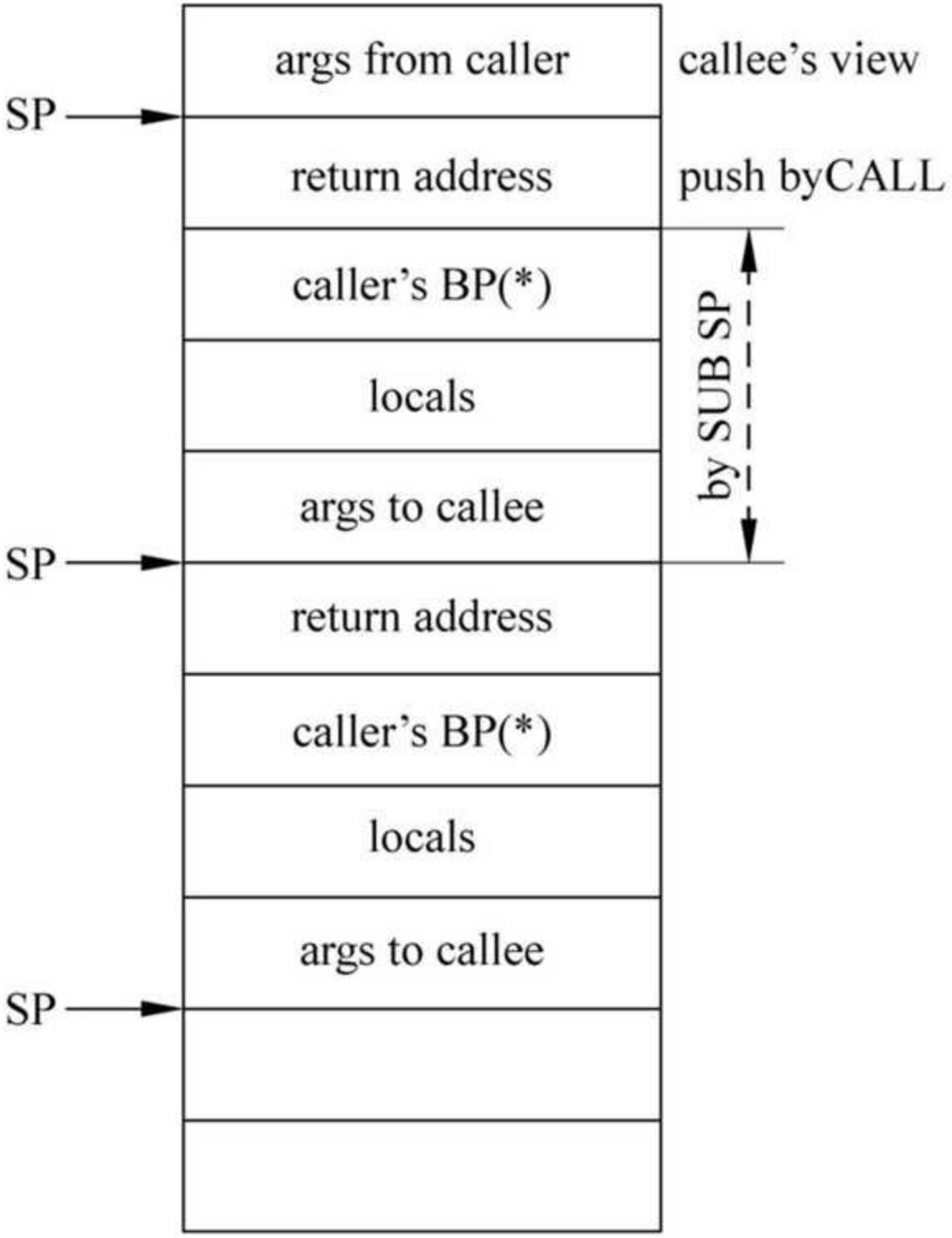

图3-4 Go语言函数栈帧布局示意图

（3）locals：局部变量区间，占用若干机器字。用来存放函数的局部变量，根据函数的局部变量占用空间大小来分配，没有局部变量的函数不分配。

（4）args to callee：调用传参区域，占用若干机器字。这一区域所占空间大小，会按照当前函数调用的所有函数中返回值加上参数所占用的最大空间来分配。当没有调用任何函数时，不需要分配该区间。callee视角的args from caller区间包含在caller视角的args to callee区间内，占用空间大小是小于或等于的关系。

综上所述，只有return address是一定会存在的，其他3个区间都要根据实际情况进行分析。

按照一般代码的逻辑，函数的栈帧应该包含返回值、参数、返回地址和局部变量这4部分。从空间分配的角度来看，返回值和参数是由caller负责分配的，CALL指令将返回地址入栈，然后callee通过SUB指令在栈上分配空间。从空间分配的角度更容易解释内存布局，所以不必纠结于函数栈帧的定义。

下面实际验证一下函数的栈帧布局，看一下各个区间的分布与上文所讲是否一致，代码如下：

```
//第 3 章/code_3_1.go
package main

func main() {
    var v1, v2 int
    v3, v4 := f1(v1, v2)
    println(&v1, &v2, &v3, &v4)
    f2(v3)
}

//go:noinline
func f1(a1, a2 int) (r1, r2 int) {
    var l1, l2 int
    println(&r2, &r1, &a2, &a1, &l1, &l2)
    return
}

//go:noinline
func f2(a1 int) {
    println(&a1)
}
```

注意：在后续的示例代码中都会用println（）函数来打印调试信息，之所以不使用fmt.Printf（）之类的函数，是因为前者更底层，也更“简单”，在runtime中专门用作打印调试信息，不会造成变量逃逸等问题，所以不会带来不必要的干扰。通过调试代码来验证语言特性比较直观，问题是调试代码容易造成干扰，就像物理学中的“测不准原理”，所以要足够谨慎。最稳妥的办法还是直接阅读反编译后的汇编代码，本书中给出的调试代码都经过反编译确认，确保没有造成实质性干扰而得出错误结论。

实际上，代码中的println（）函数会被编译器转换为多次调用runtime包中的printlock（）、printunlock（）、printpointer（）、printsp（）、printnl（）等函数。前两个函数用来进行并发同步，后3个函数用来打印指针、空格和换行。这5个函数均无返回值，只有printpointer（）函数有一个参数，会在调用者的args to callee区间占用一个机器字。

来看一个示例，代码如下：

```
//第 3 章/code_3_2.go
var a, b int
println(&a, &b)
```

这里的println（）函数经编译器转换后的代码如下：

```
runtime.printlock()          //获得锁
runtime.printpointer(&a)     //打印指针
runtime.printsp()            //打印空格
runtime.printpointer(&b)     //打印指针
runtime.printnl()            //打印换行
runtime.printunlock()        //释放锁
```

所以这一组函数调用只需一个机器字的空间，用来向printpointer（）函数传参。在64位Windows 10环境下，编译执行第3章/code_3_1.go得到的输出结果如下：

```
$ ./code_3_1.exe
0xc000107f50 0xc000107f48 0xc000107f40 0xc000107f38 0xc000107f20 0xc000107f18
0xc000107f70 0xc000107f68 0xc000107f60 0xc000107f58
0xc000107f38
```

这3行输出依次是由f1（）函数、main（）函数、f2（）函数中的println（）函数打印的，所以可以以此为参照，画出栈帧布局图。先对3个函数栈帧上各区间的大小进行整理，如表3-1所示。

表3-1 3个函数栈帧上各区间的大小

函数	caller's BP	locals	args to callee	大小
main()	1个指针	4个int：v1～v4	4个int：调用f1	0x48
f1()	1个指针	2个int：l1、l2	1个int：调用println	0x20
f2()	1个指针	无	1个int：调用println	0x10

结合调试输出的变量地址和以上表格，绘制栈帧布局如图3-5所示。图3-5（a）是调用f1（）函数时的栈，图3-5（b）是调用f2（）函数时的栈。通过f1（）函数的调用栈，可以发现函数的返回值和参数是按照先返回值后参数，并且是按照由右至左的顺序在栈上分配的，与C语言时期的参数入栈顺序一致。f1（）函数的参数和返回值占满了整个args to callee区间。

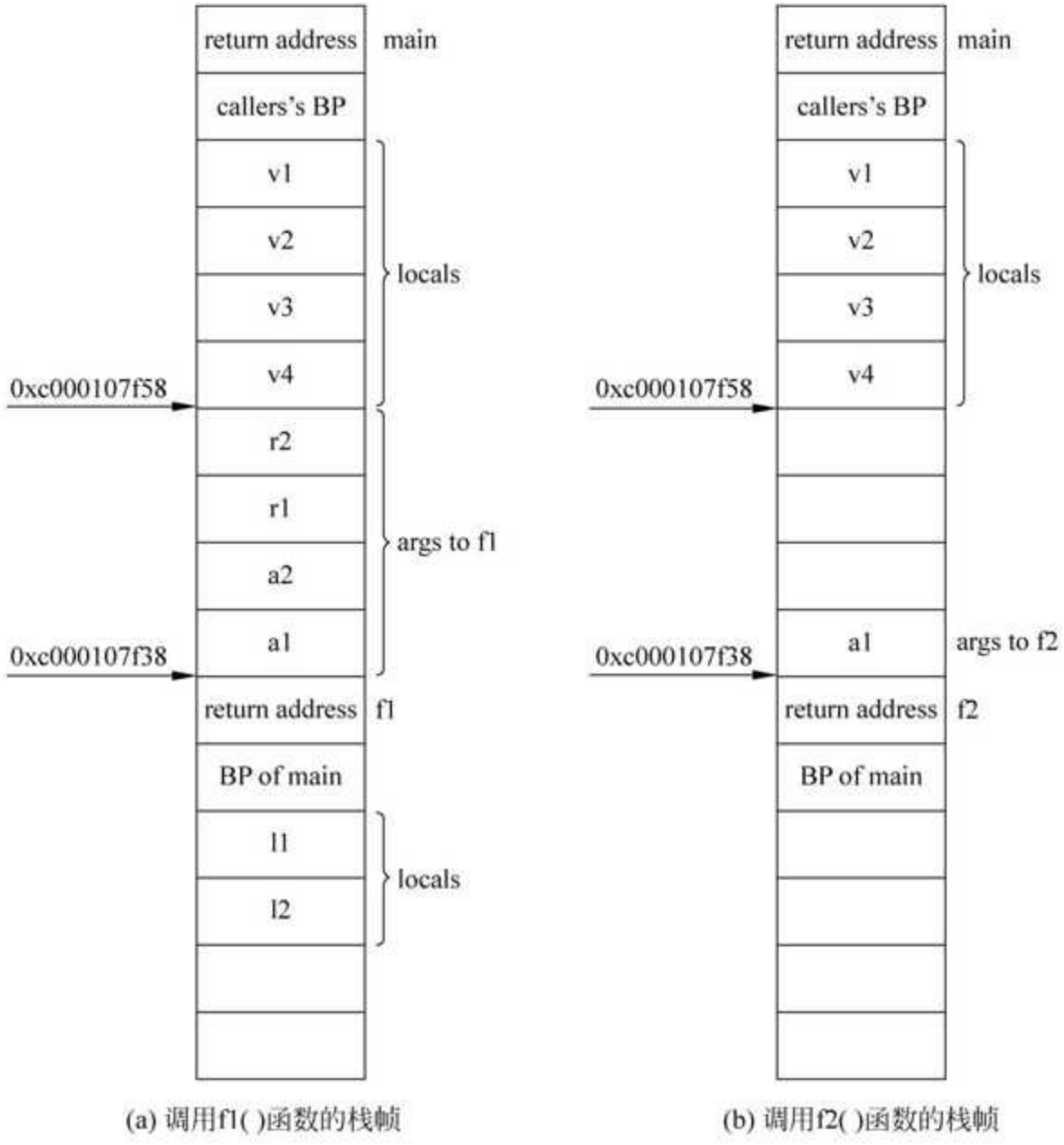

图3-5 main调用f1（）函数和f2（）函数的栈帧布局图

值得注意的是，调用f2（）函数时的栈，在a1和v4之间空了3个机器字。这是因为Go语言的函数是固定栈帧大小的，args to callee是按照所需的最大空间来分配的。调用函数时，参数和返回值看起来更像是按照先参数后返回值，从左到右的顺序分配在args to callee区间中，并且从低地址开始使用的。这点与我们对传统栈的理解有些不同，更符合传统栈原理的一些编译器，如32位的VC++编译器，它使用PUSH指令动态入栈，args to callee区间的大小不是固定的。Go这种固定栈帧大小的分配方式使调试、运行时栈扫描等更易于实现。

3.1.2 寻址方式

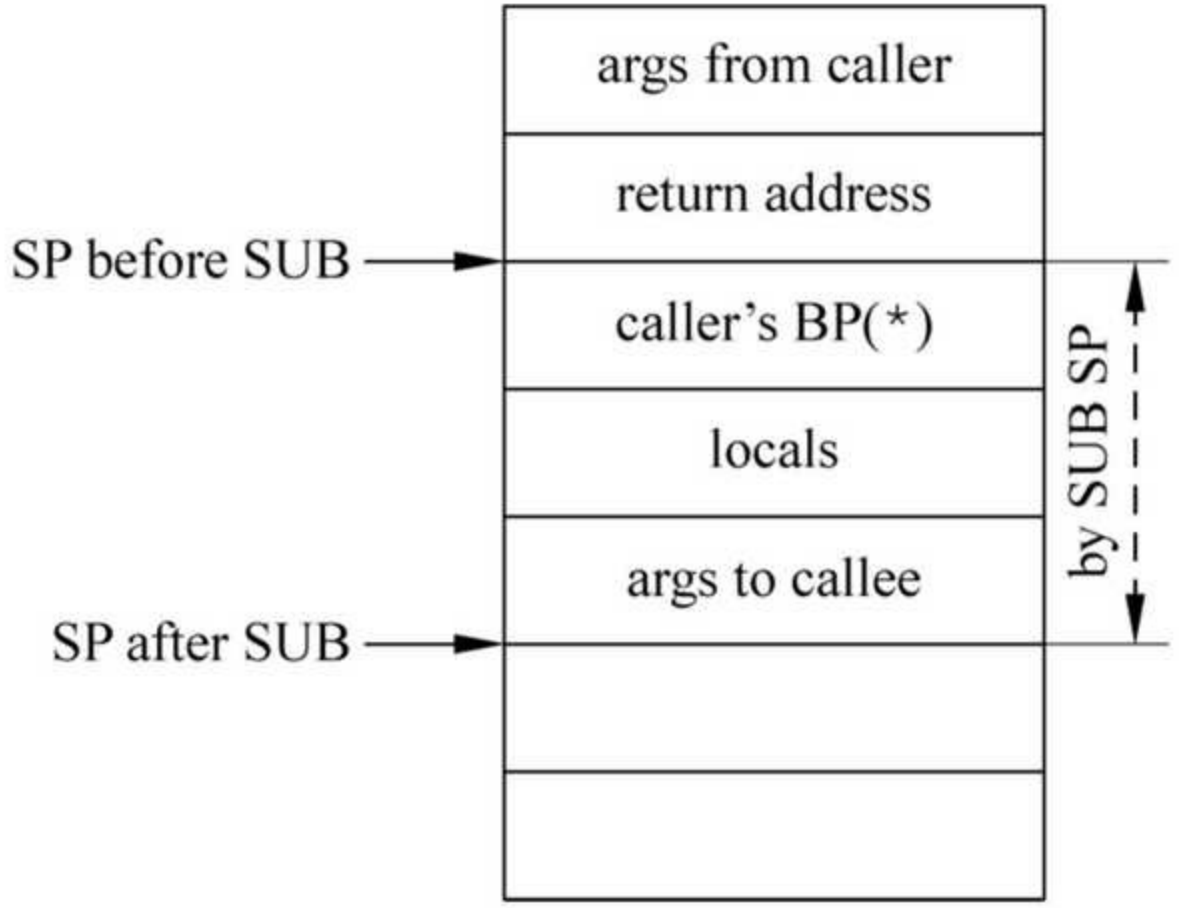

图3-6 SUB指令分配整个栈帧

从栈空间分配的角度来分析Go语言函数栈帧的结构还有另一个好处，即与实际的栈帧寻址一致。函数的prolog通过SUB指令向下移动栈指针寄存器SP来分配整个栈帧，此时SP指向args to callee区间的起始地址，如图3-6所示。

如果把图3-6中整个函数栈帧视为一个struct，SP存储着这个struct的起始地址，然后就可以通过基址+位移的方式来寻址struct的各个字段，也就是栈帧上的局部变量、参数和返回值。

下面实际反编译一个函数，看一下汇编代码中实际的寻址方式。为了尽可能包含函数栈帧的各部分，而又避免汇编代码太过复杂，准备了一个示例，代码如下：

```
//第 3 章/code_3_3.go
package main

func main() {
    fa(0)
}

//go:noinline
func fa(n int) (r int) {
    r = fb(n)
    return
}

//go:noinline
func fb(n int) int {
    return n
}
```

在64位Windows 10下编译上述代码，然后反编译fa（）函数得到的汇编代码如下：

```
$ go tool objdump -S -s 'main.fa' gom.exe                                    //1
TEXT main.fa(SB) C:/gopath/src/fengyoulin.com/gom/code_3_3.go                //2
func fa(n int) (r int) {                                                     //3
  0x488e60          65488b0c2528000000      MOVQ GS:0x28, CX                 //4
  0x488e69          488b8900000000          MOVQ 0(CX), CX                   //5
  0x488e70          483b6110                CMPQ 0x10(CX), SP                //6
  0x488e74          7630                    JBE 0x488ea6                     //7
  0x488e76          4883ec18                SUBQ $0x18, SP                   //8
  0x488e7a          48896c2410              MOVQ BP, 0x10(SP)                //9
  0x488e7f          488d6c2410              LEAQ 0x10(SP), BP                //10
        r = fb(n)                                                            //11
  0x488e84          488b442420              MOVQ 0x20(SP), AX                //12
  0x488e89          48890424                MOVQ AX, 0(SP)                   //13
  0x488e8d          e81e000000              CALL main.fb(SB)                 //14
  0x488e92          488b442408              MOVQ 0x8(SP), AX                 //15
        return                                                               //16
  0x488e97          4889442428              MOVQ AX, 0x28(SP)                //17
  0x488e9c          488b6c2410              MOVQ 0x10(SP), BP                //18
  0x488ea1          4883c418                ADDQ $0x18, SP                   //19
  0x488ea5          c3                      RET                              //20
func fa(n int) (r int) {                                                     //21
  0x488ea6          e89520fdff              CALL runtime.morestack_noctxt(SB) //22
  0x488eab          ebb3                    JMP main.fa(SB)                  //23
```

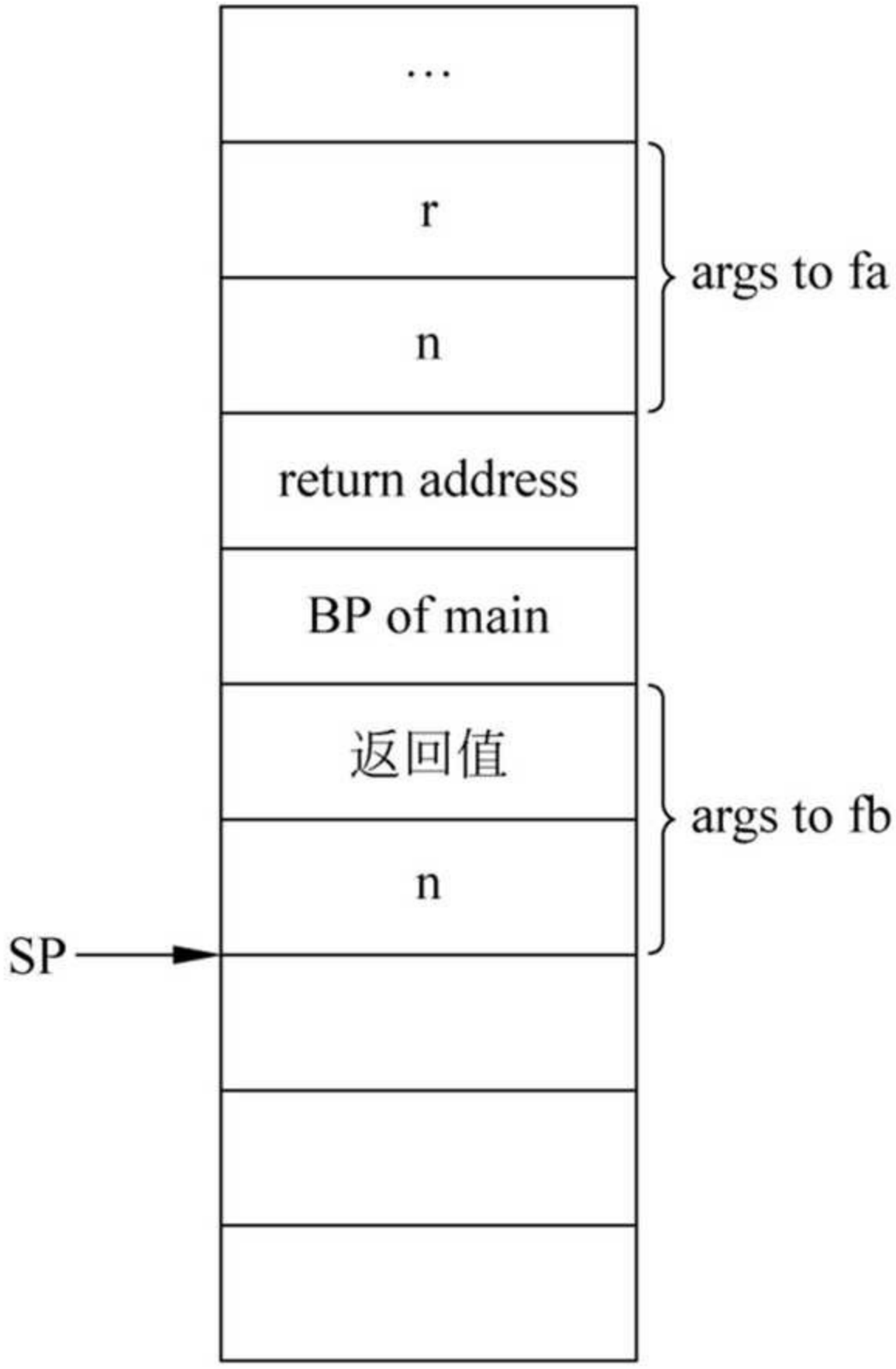
...
r
n
args to fa
return address
BP of main
返回值
n
args to fb
SP

图3-7 函数fa的栈帧布局

不熟悉x86汇编语言的读者先不要被这段代码吓到，只要阅读过本书第1章的汇编基础，看懂这段代码是不成问题的。结合图3-7所示fa（）函数的栈帧布局，这段汇编代码的结构还是很清晰的。

（1）4～7行和最后两行汇编代码主要用来检测和执行动态栈增长，与函数栈帧结构相关性不大，留到第9章栈内存管理部分再讲解。

（2）倒数第4行的RET指令用于在函数执行完成后跳转回返回地址。

（3）第8行的SUBQ指令向下移动栈指针SP，完成当前函数栈帧的分配。倒数第5行的ADDQ指令在函数返回前向上移动栈指针SP，释放当前函数的栈帧。释放与分配时的大小一致，均为0x18，即24字节，其中BP of main占用了8字节，args to fb占用了16字节。

（4）第9行代码把BP寄存器的值存到栈帧上的BP of main中，第10行把当前栈帧上BP of main的地址存入BP寄存器中。倒数第6行指令在当前栈帧释放前用BP of main的值还原BP寄存器。

（5）第12行和第13行代码，通过AX寄存器中转，把参数n的值从args to fa区间复制到args to fb区间，也就是在fa中把main（）函数传递过来的参数n，复制到调用fb（）函数的参数区间。

（6）第14行代码通过CALL指令调用fb（）函数。

（7）第15～17行代码，还是通过AX寄存器中转，把fb（）函数的返回值从args to fb区间复制到返回值r中。

Go语言中函数的返回值可以是匿名的，也可以是命名的。对于匿名返回值而言，只能通过return语句为返回值赋值。对于命名返回值，可以在代码中通过其名称直接操作，与参数和局部变量类似。无论返回值命名与否，都不会影响函数的栈帧布局。

3.1.3 又见内存对齐

在C语言函数调用中，通过栈传递的参数需要对齐到平台的位宽。假如通过栈传递4个char类型的参数，GCC生成的32位程序需要16字节栈空间，64位程序需要32字节栈空间。如果传递大量参数，则这种对齐方式会存在很大的栈空间浪费。

Go语言函数栈帧中返回值和参数的对齐方式与struct类似，对于有返回值和参数的函数，可以把所有返回值和所有参数等价成两个struct，一个返回值struct和一个参数struct。因为内存对齐方式更加紧凑，所以在支持大量参数和返回值时能够做到较高的栈空间利用率。

通过如下示例可以验证函数参数和返回值的对齐方式与struct成员的对齐方式是一致的，代码如下：

```
//第 3 章/code_3_4.go
package main

type args struct {
    a int8
    b int64
    c int32
    d int16
}

//go:noinline
func f1(a args) (r args) {
    println(&r.d, &r.c, &r.b, &r.a, &a.d, &a.c, &a.b, &a.a)
    return
}


//go:noinline
func f2(aa int8, ab int64, ac int32, ad int16) (ra int8, rb int64, rc int32, rd int16) {
    println(&rd, &rc, &rb, &ra, &ad, &ac, &ab, &aa)
    return
}

func main() {
    f1(args{})
    f2(0, 0, 0, 0)
}
```

在64位Windows 10上运行上述程序，得到的输出结果如下：

```
$ ./code_3_4.exe
0xc000039f74    0xc000039f70    0xc000039f68    0xc000039f60    0xc000039f5c    0xc000039f58
0xc000039f50 0xc000039f48
0xc000039f74    0xc000039f70    0xc000039f68    0xc000039f60    0xc000039f5c    0xc000039f58
0xc000039f50 0xc000039f48
```

第一行是用struct作为参数和返回值时的输出，第二行是按照和struct成员一致的顺序直接声明参数和返回值时的输出，可以看到两者的布局完全一致。

现在又有了一个问题：栈帧上的参数和返回值到底是分开后作为两个struct，还是按照一个struct来对齐的？可以通过如下示例进一步验证，代码如下：

```
//第 3 章/code_3_5.go
package main

//go:noinline
func f1(a int8) (b int8) {
    println(&b, &a)
    return
}

func main() {
    f1(0)
}
```

f1（）函数有一个返回值和一个参数，而且都是int8类型，如果返回值和参数作为同一个struct进行内存对齐，则a和b应该是紧邻的，中间不会插入padding。在64位Windows 10上的实际运行结果如下：

```
$ ./code_3_5.exe
0xc000039f70 0xc000039f68
```

可以看到参数a和返回值b并没有紧邻，而是分别按照8字节的边界进行对齐的，也就说明返回值和参数是分别对齐的，不是合并在一起作为单个struct。

上面探索过了参数和返回值的对齐方式，接下来再看一下局部变量是如何对齐的，是不是跟参数和返回值一样，按照声明的顺序等价于一个struct呢？这个问题也可以通过一个示例直接验证，代码如下：

```
//第 3 章/code_3_6.go
//go:noinline
func fn() {
    var a int8
    var b int64
    var c int32
    var d int16
    var e int8
    println(&a, &b, &c, &d, &e)
}
```

在64位Windows 10上运行后得到的输出结果如下：

```
$ ./code_3_6.exe
0xc0000c9f59 0xc0000c9f60 0xc0000c9f5c 0xc0000c9f5a 0xc0000c9f58
```

可以看到编译器对这5个局部变量在栈帧上的布局进行了调整，与声明顺序并不一致，可以将局部变量区间等价成一个struct，代码如下：

```
struct {
    e int8
    a int8
    d int16
    c int32
    b int64
}
```

经过这样调整后，变量布局更加紧凑，编译器没有插入任何padding，空间利用率更高。

这里可以再问一个问题：为什么编译器会对栈帧上局部变量的顺序进行调整以优化内存利用率，但是并不会调整参数和返回值呢？这其实很好解释，因为函数本身就是对代码单元的封装，参数和返回值属于对外暴露的接口，编译器必须按照函数原型来呈现，而局部变量属于封装在内部的数据，不会对外暴露，所以编译器按需调整局部变量布局不会对函数以外造成影响。

3.1.4 调用约定

在进行函数调用的时候，调用者需要把参数传递给被调用者，而被调用者也要把返回值回传给调用者。调用约定就是用来规范参数和返回值的传递问题的。如果基于栈传递，还会规定栈空间由谁负责分配、释放。有了调用约定的规范，在构建应用程序的时候，只要知道目标函数的原型就能生成正确的调用代码，而不需要关心函数的具体实现，这也是编译链接技术的一项必要基础。

截至目前的探索研究，可以对Go语言普通函数的调用约定进行如下总结：

（1）返回值和参数都通过栈传递，对应的栈空间由调用者负责分配和释放。

（2）返回值和参数在栈上的布局等价于两个struct，struct的起始地址按照平台机器字长对齐。

要想真正理解调用约定的意义，还是要了解编译、链接这两个阶段。在C语言中，编译器一般是以源码文件为单位，通过编译生成一个个对应的目标文件，目标文件中就已经是机器指令了。对于不是在当前源码文件中定义的函数，CALL指令处会把函数地址留空，到了链接阶段再由链接器负责在这些预留的位置填上实际的函数地址。给函数传参和读取返回值的指令需要由编译器在编译阶段生成，那如何保证调用者和真正的函数实现能够达成一致呢？那就是调用约定的作用，体现在C语言的函数原型上。函数原型可以通过声明给出，不必同时定义函数体的实现，编译器就是参照函数原型来生成传参相关指令的。

在Go语言中不常见到单独给出的函数声明，基本上连同函数体一起给出，编译器在函数内联优化方面也比C语言更激进。函数的声明和实现总在一起，如何验证编译器能够参照函数声明来生成传参相关指令呢？可以不使用go build命令，而是直接使用go tool compile命令，即只编译不链接。

创建一个add.go文件并写入示例内容，代码如下：

```
//第 3 章/code_3_7.go
package main

import _ "unsafe"

func main() {
    Add(1, 2)
}

func Add(a, b int) int
```

需要注意，Add（）函数只有声明而没有实现。下面对其进行编译，命令如下：

```
go tool compile -trimpath="`pwd`=>" -p main -o add.o code_3_7.go
```

然后反编译add.o文件中的main（）函数，命令如下：

```
go tool objdump -S -s main.main add.o
```

与Add（）函数调用相关的几行汇编代码如下：

```
      Add(1, 2)
0x2c8     48c7042401000000        MOVQ $0x1, 0(SP)
0x2d0     48c744240802000000      MOVQ $0x2, 0x8(SP)
0x2d9     e800000000              CALL 0x2de          [1:5]R_CALL:main.Add
```

可以看到两条MOVQ指令分别复制了参数1和2，证明编译阶段参照函数声明生成了正确的传参指令，也就是调用约定在发挥作用。CALL指令处，十六进制编码e800000000预留了32位的偏移量空间，在链接阶段会被链接器填写为实际的偏移值。

3.1.5 Go 1.17的变化

在本书临近截稿时，Go 1.17版本正式发布了，其中对函数的传参进行了优化。在1.16版及以前的版本中都是通过栈来传递参数的，这样实现简单且能支持海量的参数传递，缺点就是与寄存器传参相比性能方面会差一些。在1.17版本中就实现了基于寄存器的参数传递，当然只是在部分硬件架构上实现了。某些寄存器比较匮乏的平台，如32位的x86，可用的寄存器太少，实际传参时总是有一部分参数要通过栈传递，所以改进的意义不大。即使有16个通用寄存器的amd64架构，可用于传参的寄存器也是有上限的，参数太多时还是要有一部分通过栈传递。

下面我们就用专门设计的代码，结合Go自带的反编译工具，在汇编代码层面看一下1.17版本的函数调用是如何通过寄存器传递参数的。

1. 函数入参的传递方式

首先看一下入参是如何传递的，准备一个示例，代码如下：

```
//第3章/code_3_8.go
package main

func main() {
    in12(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
}

//go:noinline
func in12(a, b, c, d, e, f, g, h, i, j, k, l int8) int8 {
    return a + b + c + d + e + f + g + h + i + j + k + l
}
```

这个in12（）函数有12个输入参数，我们禁止编译器把它内联优化，这样才能通过反编译看到函数调用传参的汇编代码。反编译命令及得到的汇编代码如下：

```
$ go tool objdump -S -s '^main.main$' gom.exe
TEXT main.main(SB) C:/gopath/src/fengyoulin.com/gom/code_3_8.go
func main() {
  0x45aae0        493b6610            CMPQ 0x10(R14), SP
  0x45aae4        7659                JBE 0x45ab3f
  0x45aae6        4883ec20            SUBQ $0x20, SP
  0x45aaea        48896c2418          MOVQ BP, 0x18(SP)
  0x45aaef        488d6c2418          LEAQ 0x18(SP), BP
        in12(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
  0x45aaf4        66c704240a0b        MOVW $0xb0a, 0(SP)
  0x45aafa        c64424020c          MOVB $0xc, 0x2(SP)
  0x45aaff        b801000000          MOVL $0x1, AX
  0x45ab04        bb02000000          MOVL $0x2, BX
  0x45ab09        b903000000          MOVL $0x3, CX
  0x45ab0e        bf04000000          MOVL $0x4, DI
  0x45ab13        be05000000          MOVL $0x5, SI
  0x45ab18        41b806000000        MOVL $0x6, R8
  0x45ab1e        41b907000000        MOVL $0x7, R9
  0x45ab24        41ba08000000        MOVL $0x8, R10
  0x45ab2a        41bb09000000        MOVL $0x9, R11
  0x45ab30        e82b000000          CALL main.in12(SB)
}
  0x45ab35        488b6c2418          MOVQ 0x18(SP), BP
  0x45ab3a        4883c420            ADDQ $0x20, SP
  0x45ab3e        c3                  RET
func main() {
  0x45ab3f        90                  NOPL
  0x45ab40        e8bb86ffff          CALL runtime.morestack_noctxt.abi0(SB)
  0x45ab45        eb99                JMP main.main(SB)
```

上述命令反编译了main（）函数，我们关注的是它调用in12（）函数时是如何传参的。通过这一系列MOVL命令我们可以知道，第1～9个参数是依次用AX、BX、CX、DI、SI、R8、R9、R10和R11这9个通用寄存器来传递的，从第10个参数开始使用栈来传递，如图3-8所示。通过函数头部的栈增长代码，我们还可以发现R14寄存器被用来存放当前协程的g指针了，不过这就是题外话了。

2．函数返回值的传递方式

探索了函数入参是如何传递的，接下来再用另一个例子来探索一下函数的返回值的传递方式，代码如下：

```
//第 3 章/code_3_9.go
package main

func main() {
    out12()
}

//go:noinline
func out12() (a, b, c, d, e, f, g, h, i, j, k, l int8) {
    return 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
}
```

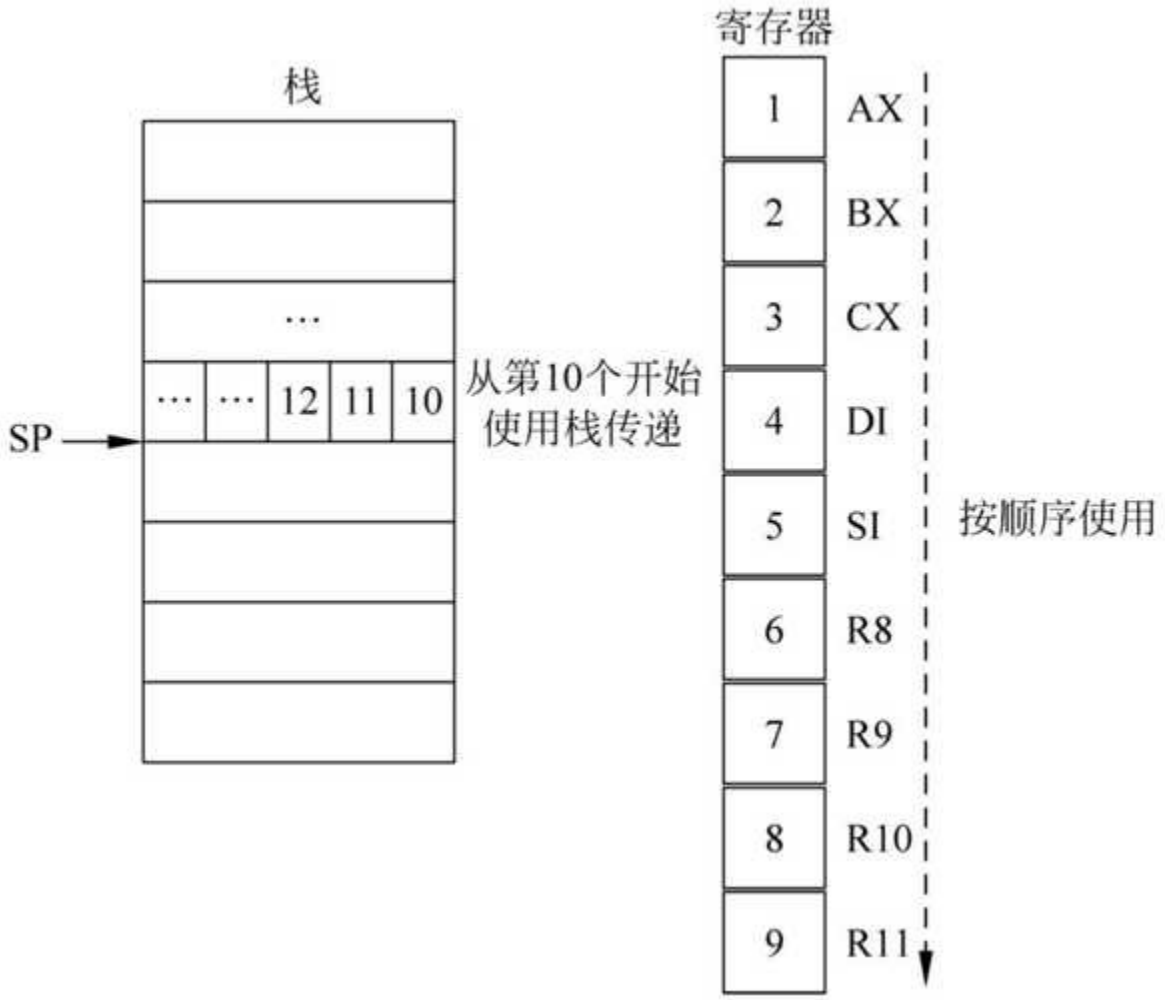

图3-8 Go 1.17中in12（）函数入参的传递方式

out12（）函数会返回12个返回值，我们还是得禁止编译器将其内联优化。这次我们要反编译out12（）函数，代码如下：

```
$ go tool objdump -S -s '^main.out12$' gom.exe
TEXT main.out12(SB) C:/gopath/src/fengyoulin.com/gom/code_3_9.go
        return 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
  0x45ab20        c64424080a          MOVB $0xa, 0x8(SP)
  0x45ab25        c64424090b          MOVB $0xb, 0x9(SP)
  0x45ab2a        c644240a0c          MOVB $0xc, 0xa(SP)
  0x45ab2f        b801000000          MOVL $0x1, AX
  0x45ab34        bb02000000          MOVL $0x2, BX
  0x45ab39        b903000000          MOVL $0x3, CX
  0x45ab3e        bf04000000          MOVL $0x4, DI
  0x45ab43        be05000000          MOVL $0x5, SI
  0x45ab48        41b806000000        MOVL $0x6, R8
  0x45ab4e        41b907000000        MOVL $0x7, R9
  0x45ab54        41ba08000000        MOVL $0x8, R10
  0x45ab5a        41bb09000000        MOVL $0x9, R11
  0x45ab60        c3                  RET
```

如图3-9所示，可以看到与入参相同，前9个返回值使用了同一组寄存器传递，并且是按照相同的顺

序来使用的。从第10个返回值开始，要通过栈来传递，而栈上传参的方式与1.16版本及以前一样。

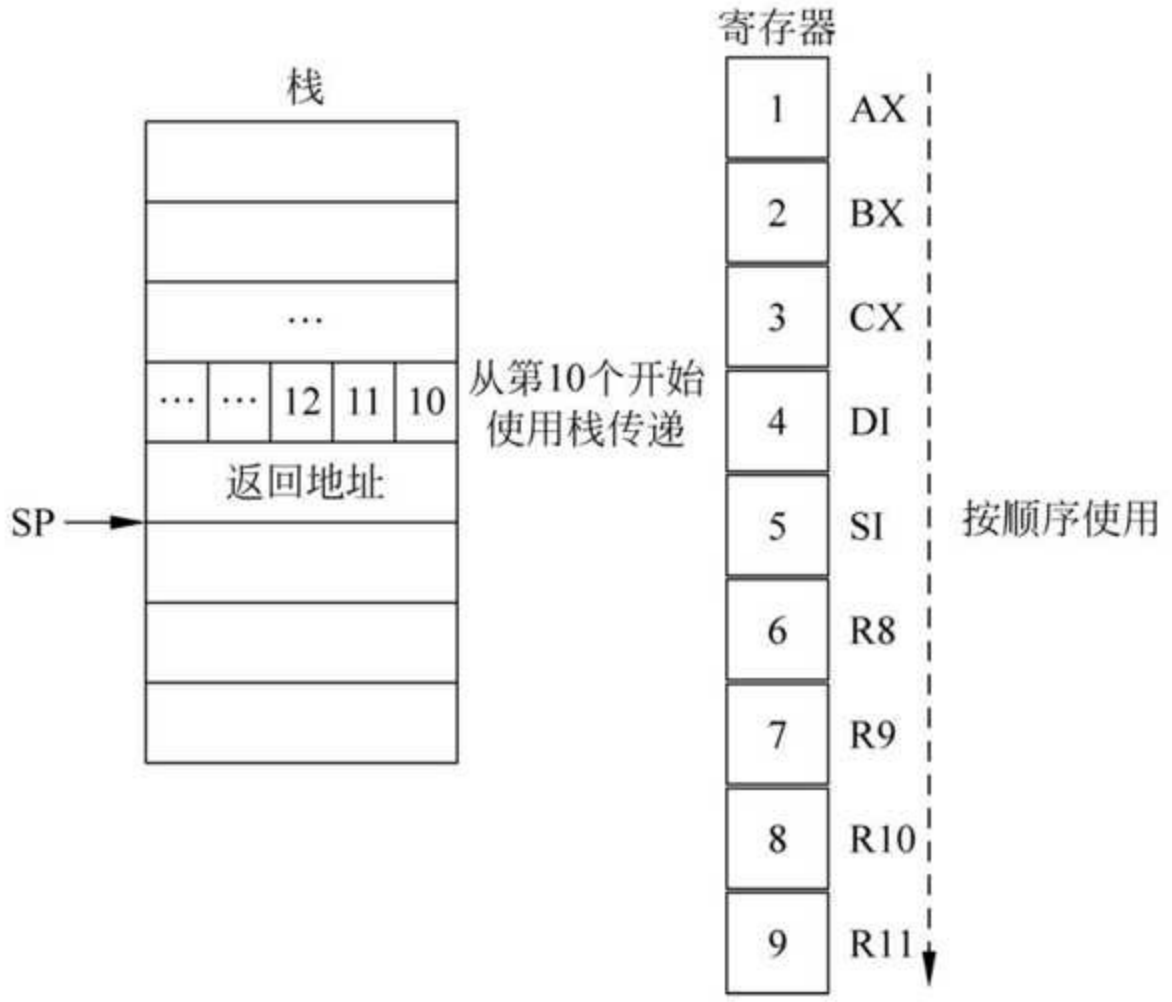

图3-9 Go 1.17中out12（）函数返回值的传递方式

总体来讲，使用9个通用寄存器对传参进行优化，最多只能传递9个机器字大小，而不是9个参数。像string会占用2个机器字，而切片会占用3个。即便如此，对于大部分函数来讲都已经够用了，所以整体优化还是很可观的。笔者在这里就不进行性能测试了，有兴趣的读者可以自行设计用例，使用自带的Benchmark来测评一下。

3.2 逃逸分析

3.2.1 什么是逃逸分析

在解释逃逸分析之前，先来思考一个场景，如果一个函数把自己栈帧上某个局部变量的地址作为返回值返回，会有什么问题？示例代码如下：

```
//第 3 章/code_3_10.go
package main

func main() {
    println( * newInt())
}

//go:noinline
func newInt() * int {
    var a int
    return &a
}
```

按照3.1节对函数栈帧布局的讲解，newInt（）函数的局部变量a应该分配在函数栈帧的locals区间。在newInt（）函数返回后，它的栈帧随即销毁，返回的变量a的地址就会变成一个悬挂指针，caller中对该地址进行的所有读写都是不合法的，会造成程序逻辑错误甚至崩溃。

事实是这样的吗？上述分析有个前提条件，即变量a被分配在栈上。假如编译器能够检测到这种模式，而自动把变量a改为堆分配，就不存在上述问题了。反编译newInt（）函数，看一下结果，代码如下：

```
$ go tool objdump -S -s 'main.newInt$' gom
TEXT main.newInt(SB) /home/fengyoulin/go/src/fengyoulin.com/gom/code_3_10.go
func newInt() * int {
  0x458710              64488b0c25f8ffffff      MOVQ FS:0xfffffff8, CX
  0x458719              483b6110                CMPQ 0x10(CX), SP
  0x45871d              7632                    JBE 0x458751
  0x45871f              4883ec18                SUBQ $0x18, SP
  0x458723              48896c2410              MOVQ BP, 0x10(SP)
  0x458728              488d6c2410              LEAQ 0x10(SP), BP
        var a int
  0x45872d              488d054c980000          LEAQ 0x984c(IP), AX
  0x458734              48890424                MOVQ AX, 0(SP)
  0x458738              e8831efbff              CALL runtime.newobject(SB)
  0x45873d              488b442408              MOVQ 0x8(SP), AX
        return &a
  0x458742              4889442420              MOVQ AX, 0x20(SP)
  0x458747              488b6c2410              MOVQ 0x10(SP), BP
  0x45874c              4883c418                ADDQ $0x18, SP
  0x458750              c3                      RET
func newInt() * int {
  0x458751              e85a97ffff              CALL runtime.morestack_noctxt(SB)
  0x458756              ebb8                    JMP main.newInt(SB)
```

重点关注上述汇编代码中runtime.newobject（）函数调用，该函数是Go语言内置函数new（）的具体实现，用来在运行阶段分配单个对象。CALL指令之后的两条MOVQ指令通过AX寄存器中转，把runtime.newobject（）函数的返回值复制给了newInt（）函数的返回值，这个返回值就是动态分配的int型变量的地址。

如果把第3章/code_3_10.go中newInt（）函数中的取地址运算改成使用内置函数new（），则效果也是一样的，代码如下：

```
//go:noinline
func newInt() *int {
    return new(int)
}
```

根据上述研究，现阶段可以把逃逸分析描述为当函数局部变量的生命周期超过函数栈帧的生命周期时，编译器把该局部变量由栈分配改为堆分配，即变量从栈上逃逸到堆上。

3.2.2 不逃逸分析

3.2.1节演示了逃逸分析，代码示例中将函数的某个局部变量的地址作为返回值返回，或者通过内置函数new（）动态分配变量并返回其地址。其中内置函数new（）有着非常明显的堆分配的含义，是不是只要使用了new（）函数就会造成堆分配呢？进一步猜想，如果对局部变量进行取地址操作会被转换为new（）函数调用，那就不用进行所谓的逃逸分析了。

先验证new（）函数与堆分配是否有必然关系，代码如下：

```
//第 3 章/code_3_11.go
//go:noinline
func New() int {
    p := new(int)
    return *p
}
```

反编译New（）函数，得到的汇编代码如下：

```
$ go tool objdump -S -s '^main.New$' gom
TEXT main.New(SB) /home/fengyoulin/go/src/fengyoulin.com/gom/code_3_11.go
        return *p
  0x458710              48c7442408000000      MOVQ $0x0, 0x8(SP)
  0x458719              c3                    RET
```

MOVQ指令直接把返回值赋值为0，其他的逻辑全都被优化掉了，所以即便是代码中使用了new（）函数，只要变量的生命周期没有超过当前函数栈帧的生命周期，编译器就不会进行堆分配。事实上，只要代码逻辑允许，编译器总是倾向于把变量分配在栈上，因为比分配在堆上更高效。这也就是本节所谓的不逃逸分析，或者说未逃逸分析，这种说法并不严谨，主要是为了突出编译器倾向于让变量不逃逸。

3.2.3 不逃逸判断

本节主要探索编译器进行逃逸分析时追踪的范围，以及在什么情况下就认为变量逃逸了或者确定变量没有逃逸。3.2.1节研究变量逃逸所用的方法，主要通过让函数返回局部变量的地址，使局部变量的生命周期超过对应函数栈帧的生命周期。按照这个规则来猜想，如果把局部变量的地址赋值给包级别的指针变量，应该也会造成变量逃逸。准备一个示例，代码如下：

```
//第 3 章/code_3_12.go
var pt * int

//go:noinline
func setNew() {
    var a int
    pt = &a
}
```

反编译setNew（）函数，在得到的汇编代码中节选关键的几行，代码如下：

```
    var a int
0x488eb4            488d0525db0000              LEAQ runtime.types + 51680(SB), AX
0x488ebb            48890424                    MOVQ AX, 0(SP)
0x488ebf            e8cc34f8ff                  CALL runtime.newobject(SB)
0x488ec4            488b442408                  MOVQ 0x8(SP), AX
```

通过runtime.newobject（）函数调用就能确定，变量a逃逸到了堆上，验证了上述猜想。进一步还可以验证逃逸分析的依赖传递性，准备示例代码如下：

```
//第 3 章/code_3_13.go
var pp ** int

//go:noinline
func dep() {
    var a int
    var p * int
    p = &a
    pp = &p
}
```

反编译dep（）函数，节选部分汇编：从节选的部分代码可以发现，变量p和a都逃逸了。p的地址被赋值给包级别的指针变量pp，而a的地址又被赋值给了p，因为p逃逸造成a也逃逸了，代码如下：

```
$ go tool objdump -S -s '^main.dep$' gom.exe
TEXT main.dep(SB) C:/gopath/src/fengyoulin.com/gom/code_3_13.go
func dep() {
  //省略部分代码
        var a int
  0x493ec4          488d0575a70000        LEAQ tuntime.rodata + 42560(SB), AX
  0x493ecb          48890424              MOVQ AX, 0(SP)
  0x493ecf          e84c97f7ff            CALL runtime.newobject(SB)
  0x493ed4          488b442408            MOVQ 0x8(SP), AX
  0x493ed9          4889442410            MOVQ AX, 0x10(SP)
        var p * int
  0x493ede          488d0d7b670000        LEAQ runtime.rodata + 26208(SB), CX
  0x493ee5          48890c24              MOVQ CX, 0(SP)
  0x493ee9          e83297f7ff            CALL runtime.newobject(SB)
  0x493eee          488b7c2408            MOVQ 0x8(SP), DI
```

假如某个函数有一个参数和一个返回值，类型都是整型指针，函数只是简单地把参数作为返回值返

回，就像下面的inner.RetArg（）函数，代码如下：

```
//第 3 章/code_3_14.go
package inner

//go:noinline
func RetArg(p * int) * int {
    return p
}
```

在另一个包中arg（）函数调用了inner.RetArg（）函数，将局部变量a的地址作为参数，并返回了一个int类型的返回值，代码如下：

```
//第 3 章/code_3_15.go
package main

//go:noinline
func arg() int {
    var a int
    return * inner.RetArg(&a)
}
```

在arg（）函数中并没有把变量a的地址作为返回值，也不存在到某个包级别指针变量的依赖链路，所以变量a是否会逃逸的关键就在于inner.RetArg（）函数。inner.RetArg（）函数只是把传过去的指针又传了回来，而且作为被调用者来讲，它的生命周期是完全包含在arg（）函数的生命周期以内的，所以不应该造成变量a逃逸。

事实到底如何呢？还要通过反编译验证，节选部分关键汇编代码如下：

```
    var a int
0x489034            48c744241000000000      MOVQ $0x0, 0x10(SP)
    return * inner.RetArg(&a)
0x48903d            488d442410              LEAQ 0x10(SP), AX
0x489042            48890424                MOVQ AX, 0(SP)
0x489046            e845b1fdff              CALL funny/inner.RetArg(SB)
0x48904b            488b442408              MOVQ 0x8(SP), AX
0x489050            488b00                  MOVQ 0(AX), AX
0x489053            4889442428              MOVQ AX, 0x28(SP)
```

没错，变量a确实是在栈上分配的，也就说明编译器参考了inner.RetArg（）函数的具体实现，基于代码逻辑判定变量a没有逃逸。虽然代码中通过noinline阻止了内联优化，但是没能阻止编译器参考函数实现。假如通过某种方式能够阻止编译器参考函数实现，又会有什么样的结果呢？

可以使用linkname机制，连同修改后的arg（）函数的代码如下：

```
//第 3 章/code_3_16.go
//go:linkname retArg funny/inner.RetArg
func retArg(p * int) * int

//go:noinline
func arg() int {
    var a int
    var b int
    return * inner.RetArg(&a) + * retArg(&b)
}
```

再次反编译arg（）函数，节选变量a和b分配相关的汇编代码如下：

```
    var a int
0x489034            48c744241000000000    MOVQ $0x0, 0x10(SP)
    var b int
0x48903d            488d059cd90000        LEAQ runtime.types + 51680(SB), AX
0x489044            48890424              MOVQ AX, 0(SP)
0x489048            e84333f8ff            CALL runtime.newobject(SB)
0x48904d            488b442408            MOVQ 0x8(SP), AX
0x489052            4889442420            MOVQ AX, 0x20(SP)
```

变量a依旧是栈分配，变量b已经逃逸了。在上述代码中的retArg（）函数只是个函数声明，没有给出具体实现，通过linkname机制让链接器在链接阶段链接到inner.RetArg（）函数。retArg（）函数只有声明没有实现，而且编译器不会跟踪linkname，所以无法根据代码逻辑判定变量b到底有没有逃逸。

把逻辑上没有逃逸的变量分配到堆上不会造成错误，只是效率低一些，但是把逻辑上逃逸了的变量分配到栈上就会造成悬挂指针等问题，因此编译器只有在能够确定变量没有逃逸的情况下，才会将其分配到栈上，在能够确定变量已经逃逸或无法确定到底有没有逃逸的情况下，都要按照已经逃逸来处理。这也就解释了为什么在上述代码中的变量b逻辑上没有逃逸，却被分配在了堆上。

3.3 Function Value

函数在Go语言中属于一类值（First Class Value），该类型的值可以作为函数的参数和返回值，也可以赋给变量。当把一个函数赋值给某个变量后，这个变量就被称为Function Value。声明一个Function Value变量的示例代码如下：

```
var fn func(a, b int) int
```

其中fn就是个Function Value变量，它的类型是func（int，int）int。Function Value可以像一般函数那样被调用，在使用体验上非常类似于C语言中的函数指针。那么Function Value本质上是不是函数指针呢？

本节会分析Function Value和函数指针的实现原理，还有闭包的实现原理，以及Function Value是如何支持闭包的。

3.3.1 函数指针

熟悉C语言的读者应该有过使用函数指针的经验，函数指针跟本书第2章中所讲的指针类似，存储的都是地址，只不过不是指向某种类型的数据，而是指向代码段中某个函数的第一条指令，如图3-10所示。

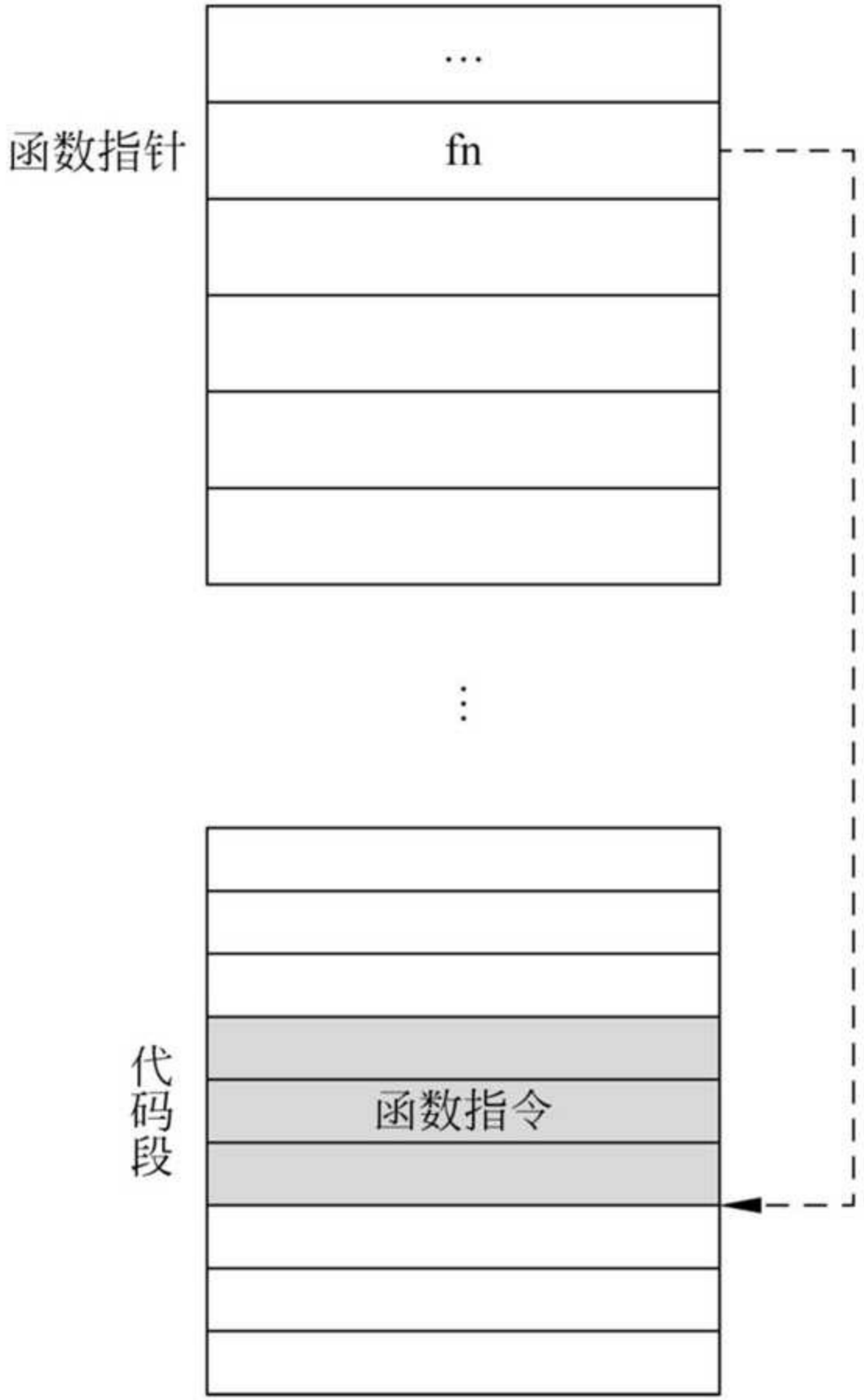
...
函数指针
fn
代码段
函数指令

图3-10 函数指针

准备一个简单的C语言函数指针应用示例，代码如下：

```
//第 3 章/code_3_17.c
int helper(int ( * fn)(int, int), int a, int b) {
    return fn(a, b);
}

int main() {
    return helper(0, 0, 0);
}
```

上述helper（）函数有3个参数，fn是个函数指针。在Linux+amd64环境下，用GCC编译上述代码，命令如下：

```
$ gcc -O1 -o main code_3_17.c
```

编译优化级别O1刚好合适，既不会内联优化掉helper（）函数，又能生成简洁易读的汇编代码。用GDB调试反编译helper（）函数，代码如下：

```
(gdb) disass
Dump of assembler code for function helper:
=> 0x00005555555545fa <+0>:     sub    $0x8,%rsp
   0x00005555555545fe <+4>:     mov    %rdi,%rax
   0x0000555555554601 <+7>:     mov    %esi,%edi
   0x0000555555554603 <+9>:     mov    %edx,%esi
   0x0000555555554605 <+11>:    callq  *%rax
   0x0000555555554607 <+13>:    add    $0x8,%rsp
   0x000055555555460b <+17>:    retq
End of assembler dump.
```

通过上述代码可见，GCC使用DI、SI和DX寄存器按顺序传递了helper（）函数的3个参数。通过函数指针fn进行调用的具体逻辑如下：

（1）mov%rdi，%rax把函数指针fn中存储的地址从rdi复制到rax寄存器。

（2）mov%esi，%edi把esi复制到edi，也就是把helper（）函数的第2个参数作为fn的第1个参数。

（3）mov%edx，%esi把edx复制到esi，也就是把helper（）函数的第3个参数作为fn的第2个参数。

（4）callq*%rax调用rax寄存器中存储的地址处的函数。

通过查阅反编译后的汇编代码，可以确定C语言中的函数指针就是个函数地址。函数指针的类型类似于函数声明，编译器参考这种类型信息并依据调用约定来生成传参等汇编指令。

3.3.2 Function Value分析

有了对C函数指针的了解，再看到Go语言中的Function Value时，第一感觉就是函数指针，不过换了个名字。实际是不是这样呢？还得通过实践来验证。

准备一个go文件并写入，示例代码如下：

```
//第 3 章/code_3_18.go
package main

func main() {
    println(helper(nil, 0, 0))
}

//go:noinline
func helper(fn func(int, int) int, a, b int) int {
    return fn(a, b)
}
```

依然把Function Value的调用隔离在一个函数中，以便于分析。反编译代码如下：

```
$ go tool objdump -S -s '^main.helper$' gom.exe
TEXT main.helper(SB) C:/gopath/src/fengyoulin.com/gom/code_3_18.go
func helper(fn func(int, int) int, a, b int) int {
  0x488e90              65488b0c2528000000      MOVQ GS:0x28, CX
  0x488e99              488b8900000000          MOVQ 0(CX), CX
  0x488ea0              483b6110                CMPQ 0x10(CX), SP
  0x488ea4              763f                    JBE 0x488ee5
  0x488ea6              4883ec20                SUBQ $0x20, SP
  0x488eaa              48896c2418              MOVQ BP, 0x18(SP)
  0x488eaf              488d6c2418              LEAQ 0x18(SP), BP
        return fn(a, b)
  0x488eb4              488b442430              MOVQ 0x30(SP), AX
  0x488eb9              48890424                MOVQ AX, 0(SP)
  0x488ebd              488b442438              MOVQ 0x38(SP), AX
  0x488ec2              4889442408              MOVQ AX, 0x8(SP)
  0x488ec7              488b542428              MOVQ 0x28(SP), DX
  0x488ecc              488b02                  MOVQ 0(DX), AX
  0x488ecf              ffd0                    CALL AX
  0x488ed1              488b442410              MOVQ 0x10(SP), AX
  0x488ed6              4889442440              MOVQ AX, 0x40(SP)
  0x488edb              488b6c2418              MOVQ 0x18(SP), BP
  0x488ee0              4883c420                ADDQ $0x20, SP
  0x488ee4              c3                      RET
func helper(fn func(int, int) int, a, b int) int {
  0x488ee5              e85620fdff              CALL runtime.morestack_noctxt(SB)
  0x488eea              eba4                    JMP main.helper(SB)
```

下面整体梳理一下这段代码：

（1）4～7行和最后两行用于栈增长，暂不需要关心。

（2）第8～10行分配栈帧并赋值caller's BP，RET之前的两行还原BP寄存器并释放栈帧。

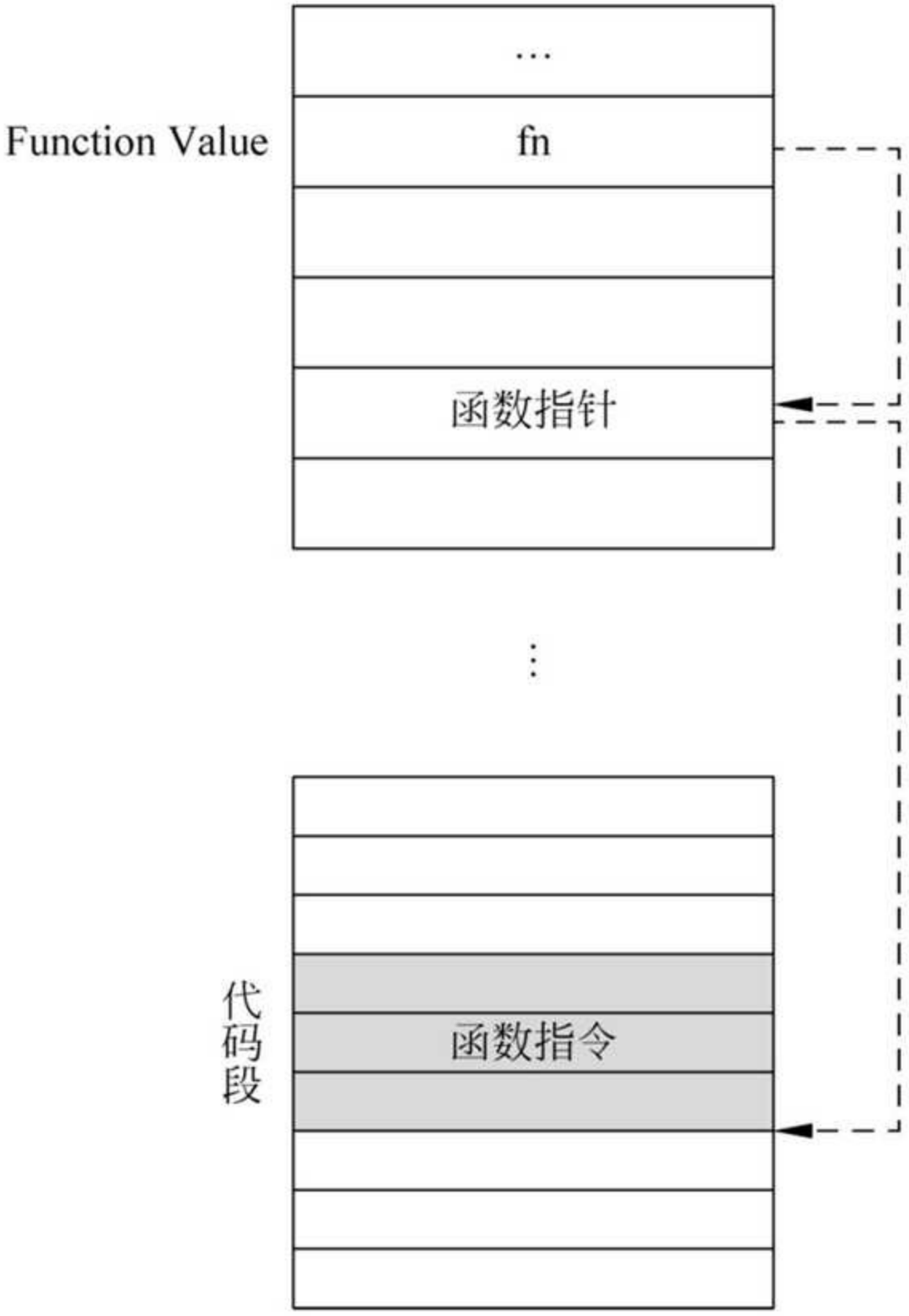

图3-11 Function Value

（3）CALL后面的两行用来复制返回值。

（4）CALL连同之前的6条MOVQ指令，实现了Function Value的传参和过程调用。

只有第4步才是需要关心的地方，进一步拆解：

（1）MOVQ 0x30（SP），AX和MOVQ AX，0（SP）用于把helper（）函数的第2个参数a的值复制给fn（）函数的第1个参数。

（2）MOVQ 0x38（SP），AX和MOVQ AX，0x8（SP）同理，把helper（）函数第3个参数b的值复制给fn（）函数的第2个参数。

（3）MOVQ 0x28（SP），DX把helper（）函数第1个参数fn的值复制到DX寄存器，MOVQ 0（DX），AX把DX用作基址，加上位移0，也就是从DX存储的地址处读取出一个64位的值，存入了AX寄存器中。

（4）CALL AX说明，上一步中AX寄存器最终存储的是实际函数的地址。

通过上述逻辑，可以确定Function Value确实是个指针，而且是个两级指针。如图3-11所示，Function Value不直接指向目标函数，而是一个目标函数的指针。为什么要通过一个两级指针实现呢？目前还真不好解释，先继续向后研究，等到3.3.3节再回过头来解释这个问题。

3.3.3 闭包

说到Go语言的闭包，比较直观的感受就是个有状态的Function Value。在Go语言中比较典型的闭包场景就是在某个函数内定义了另一个函数，内层函数使用了外层函数的局部变量，并且内层函数最终被外层函数作为返回值返回，代码如下：

```
//第 3 章/code_3_19.go
func mc(n int) func() int {
    return func() int {
        return n
    }
}
```

每次调用mc（）函数都会返回一个新的闭包，闭包记住了参数n的值，所以是有状态的。基于目前对函数栈帧的了解，函数栈帧随着函数返回而销毁，不能用来保存状态，研究函数指针和Function Value的时候也没有发现哪里用来保存状态，所以这里就有个问题：闭包的状态保存在哪里呢？

1. 闭包对象

为了搞清楚这个问题，先来尝试一下反编译，从汇编代码中找答案，反编译代码如下：

```
$ go tool objdump -S -s '^main.mc$' gom.exe
TEXT main.mc(SB) C:/gopath/src/fengyoulin.com/gom/code_3_19.go
func mc(n int) func() int {
  0x488ec0              65488b0c2528000000      MOVQ GS:0x28, CX
  0x488ec9              488b8900000000          MOVQ 0(CX), CX
  0x488ed0              483b6110                CMPQ 0x10(CX), SP
  0x488ed4              7645                    JBE 0x488f1b
  0x488ed6              4883ec18                SUBQ $0x18, SP
  0x488eda              48896c2410              MOVQ BP, 0x10(SP)
  0x488edf              488d6c2410              LEAQ 0x10(SP), BP
        return func() int {
  0x488ee4              488d0595640100          LEAQ runtime.types+91008(SB), AX  //1
  0x488eeb              48890424                MOVQ AX, 0(SP)                    //2
  0x488eef              e89c34f8ff              CALL runtime.newobject(SB)        //3
  0x488ef4              488b442408              MOVQ 0x8(SP), AX                  //4
  0x488ef9              488d0d30000000          LEAQ main.mc.func1(SB), CX        //5
  0x488f00              488908                  MOVQ CX, 0(AX)                    //6
  0x488f03              488b4c2420              MOVQ 0x20(SP), CX                 //7
  0x488f08              48894808                MOVQ CX, 0x8(AX)                  //8
  0x488f0c              4889442428              MOVQ AX, 0x28(SP)                 //9
  0x488f11              488b6c2410              MOVQ 0x10(SP), BP
  0x488f16              4883c418                ADDQ $0x18, SP
  0x488f1a              c3                      RET
func mc(n int) func() int {
  0x488f1b              e82020fdff              CALL runtime.morestack_noctxt(SB)
  0x488f20              eb9e                    JMP main.mc(SB)
```

代码中负责栈增长、栈帧分配和操作BP的部分在3.3.2节已经介绍过，此处不再赘述。重点关注return下面注释编号的9行汇编代码就可以了，逐行梳理一下这部分逻辑：

（1）第1～4行代码使用runtime.types+91008作为参数调用了runtime.newobject（）函数，并把返回值存储在AX寄存器中，这个值是个地址，指向分配在堆上的一个对象。

（2）第5行和第6行把main.mc.func1（）函数的地址复制到了AX所指向对象的头部，0（AX）表示用AX作为基址且位移为0。

（3）第7行和第8行把mc（）函数的参数n的值复制到了AX所指向对象的第2个字段，0x8（AX）表示用AX作为基址且位移为8。

（4）第9行把AX的值复制到mc（）函数栈帧上的返回值处，也就是最终返回的Function Value。

根据第2步和第3步的代码逻辑，可以推断出第1步动态分配的对象的类型。应该是个struct类型，第1个字段是个函数地址，第2个字段是int类型，代码如下：

```
struct {
    F uintptr
    n int
}
```

说明编译器识别出了闭包这种代码模式，并且自动定义了这个struct类型进行支持，出于面向对象编程中把数据称为对象的习惯，后文中就把这种struct称为闭包对象。

闭包对象的成员可以进一步划分，第1个字段F用来存储目标函数的地址，这在所有的闭包对象中都是一致的，后文中将这个目标函数称为闭包函数。从第2个字段开始，后续的字段称为闭包的捕

获列表，也就是内层函数中用到的所有定义在外层函数中的变量。编译器认为这些变量被闭包捕获了，会把它们追加到闭包对象的struct定义中。上例中只捕获了一个变量n，如果捕获的变量增多，struct的捕获列表也会加长。一个捕获两个变量的闭包示例代码如下：

```
//第 3 章/code_3_20.go
func mc2(a, b int) func() (int, int) {
    return func() (int, int) {
        return a, b
    }
}
```

上述代码对应的闭包对象定义代码如下：

```
struct {
    F uintptr
    a int
    b int
}
```

2. 看到闭包

通过反编译来逆向推断闭包对象的结构还是比较烦琐的，如果能有一种方法，能够直观地看到闭包对象的结构定义，那真是再好不过了。下面介绍一种方法，将闭包逮个正着。

根据之前的探索，已经知道Go程序在运行阶段会通过runtime.newobject（）函数动态分配闭包对象。Go源码中newobject（）函数的原型如下：

```
func newobject(typ * _type) unsafe.Pointer
```

函数的返回值是个指针，也就是新分配的对象的地址，参数是个_type类型的指针。通过源码可以得知这个_type是个struct，在Go语言的runtime中被用来描述一个数据类型，通过它可以找到目标数据类型的大小、对齐边界、类型名称等。笔者习惯将这些用来描述数据类型的数据称为类型元数据，它们是由编译器生成的，Go语言的反射机制依赖的就是这些类型元数据。

假如能够获得传递给runtime.newobject（）函数的类型元数据指针typ，再通过反射进行解析，就能打印出闭包对象的结构定义了。那如何才能获得这个typ参数呢？

在C语言中有种常用的函数Hook技术，就是在运行阶段将目标函数头部的代码替换为一条跳转指令，跳转到一个新的函数。在x86平台上就是在进程地址空间中找到要Hook的函数，将其头部替换为一条JMP指令，同时指定JMP指令要跳转到的新函数的地址。这项技术在Go程序中依然适用，可以用一个自己实现的函数替换掉runtime.newobject（）函数，在这个函数中就能获得typ参数并进行解析了。

还有一个问题是runtime.newobject（）函数属于未导出的函数，在runtime包外无法访问。这一点可以通过linkname机制来绕过，在当前包中声明一个类似的函数，让链接器将其链接到runtime.newobject（）函数即可。

本书使用开源模块github.com/fengyoulin/hookingo实现运行阶段函数替换，打印闭包对象结构的完整代码如下：

```
//第 3 章/code_3_21.go
package main

import (
    "github.com/fengyoulin/hookingo"
    "reflect"
    "unsafe"
)

var hno hookingo.Hook

//go:linkname newobject runtime.newobject
func newobject(typ unsafe.Pointer) unsafe.Pointer

func fno(typ unsafe.Pointer) unsafe.Pointer {
    t := reflect.TypeOf(0)
    (*(*[2]unsafe.Pointer)(unsafe.Pointer(&t)))[1] = typ //相当于反射了闭包对象类型
    println(t.String())
    if fn, ok := hno.Origin().(func(typ unsafe.Pointer) unsafe.Pointer); ok {
        return fn(typ) //调用原 runtime.newobject
    }
    return nil
}
//创建一个闭包,make closure
func mc(start int) func() int {
    return func() int {
        start++
        return start
    }
}

func main() {
    var err error
    hno, err = hookingo.Apply(newobject, fno) //应用钩子,替换函数
    if err != nil {
        panic(err)
    }
    f := mc(10)
    println(f())
}
```

在64位Windows 10下执行命令及运行结果如下：

```
$ ./code_3_21.exe
int
struct { F uintptr; start *int }
11
```

运行结果第2行的int和第3行的struct定义都是被fno（）函数中的println（）函数打印出来的，最后一行的11是被main（）函数中的println（）函数打印出来的。第3行的struct就是闭包对象的结构定义，闭包捕获列表中的start是个int指针，那是因为start变量逃逸了，第2行打印的int就是通过runtime.newobject（）函数动态分配造成的。如果把闭包函数中的start++一行删除，闭包捕获的

start就是个值而不是指针，本节的最后将解释闭包捕获与变量逃逸的关系。某些读者可能会对fno（）函数中的反射代码感到困惑，读完本书第5章与接口相关的内容就能够理解了。

此时再回过头去看Function Value的两级指针结构，结合闭包对象的结构定义就很好理解了。如果忽略掉闭包对象中的捕获列表部分，剩下的就是一个两级指针结构了，如图3-12所示。

Go语言在设计上用这种两级指针结构将函数指针和闭包统一为Function Value，运行阶段调用者不需要关心调用的函数是个普通的函数还是个闭包函数，一致对待就可以了。

如果每次把一个普通函数赋值给一个Function Value的时候都要在堆上分配一个指针，那就有些浪费了。因为普通函数不构成闭包也没有捕获列表，没必要动态分配。事实上编译器早就考虑到了这一点，对于不构成闭包的Function Value，第二层的这个指针是编译阶段静态分配的，只分配一个就够了。

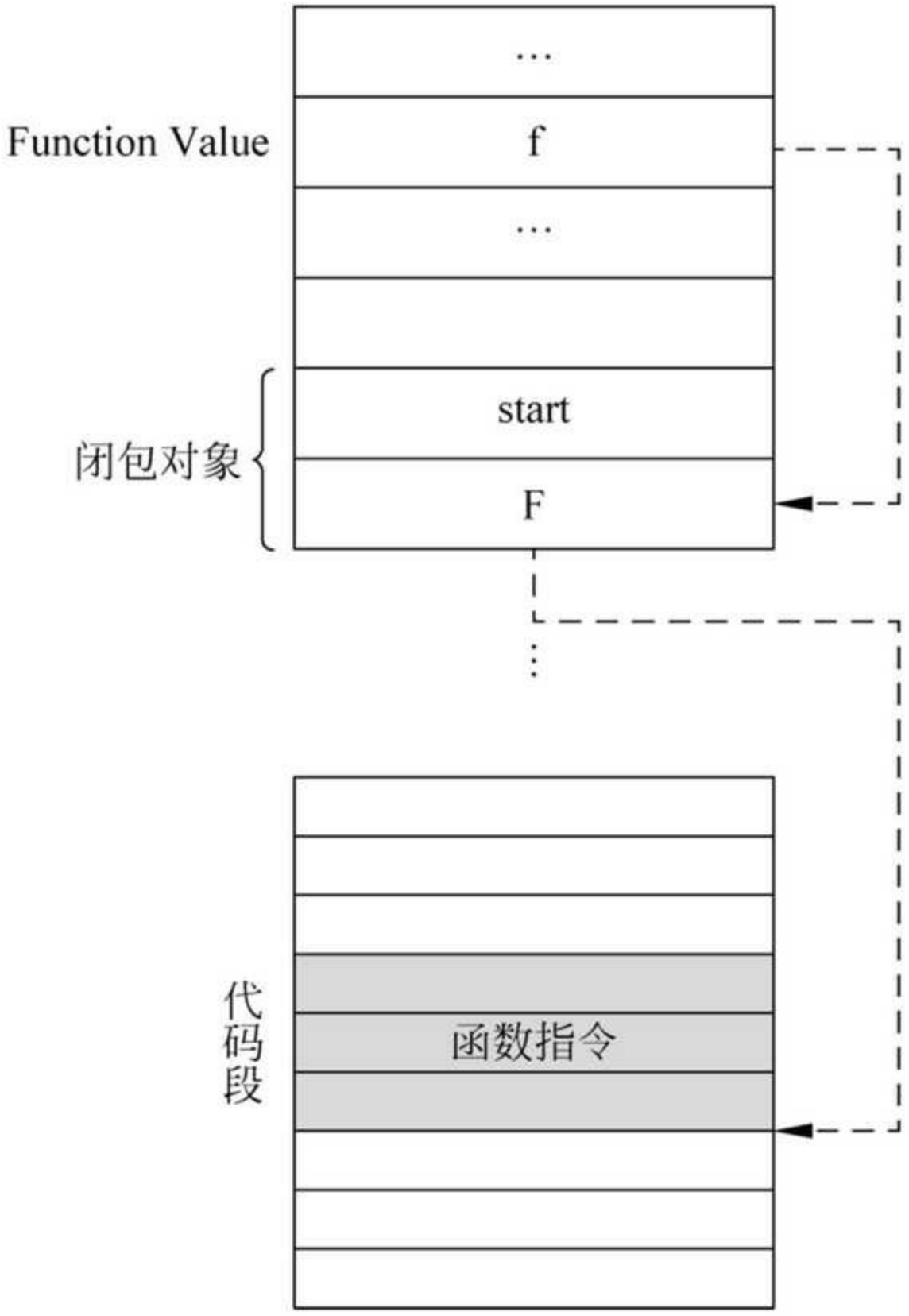

图3-12 Function Value和闭包对象

3. 调用闭包

细心的读者可能还会有个疑问：闭包函数在被调用的时候，必须得到当前闭包对象的地址才能访问其中的捕获列表，这个地址是如何传递的呢？

这个问题确实值得深入研究。调用者在调用Function Value的时候，只是像调用一个普通函数那样传递了声明的参数，如果Function Value背后是个闭包函数，则无法通过栈上的参数得到闭包对象地址。除非编译器传递了一个隐含的参数，这个参数如果通过栈传递，那就改变了函数的原型，这样就会造成不一致，是行不通的。

还是通过反汇编来看一下闭包函数是从哪里得到的这个地址，先来构造闭包，代码如下：

```
//第 3 章/code_3_22.go
func mc(n int) func() int {
    return func() int {
        return n
    }
}
```

根据本节第2部分的探索，可以确定闭包对象的结构定义代码如下：

```
struct {
    F uintptr
    n int
}
```

反编译闭包函数得到的汇编代码如下：

```
$ go tool objdump -S -s "^main.mc.func1$" gom.exe
TEXT main.mc.func1(SB) C:/gopath/src/fengyoulin.com/gom/code_3_22.go
        return func() int {
  0x4b6970              488b4208                MOVQ 0x8(DX), AX
                return n
  0x4b6974              4889442408              MOVQ AX, 0x8(SP)
  0x4b6979              c3                      RET
```

只有3行汇编代码，逻辑如下：

（1）将DX寄存器用作基址，再加上位移8，把该地址处的值复制到AX寄存器中。

（2）把AX寄存器的值复制给闭包函数的返回值。

（3）闭包函数返回。

显然，DX寄存器存储的就是闭包对象的地址，调用者负责在调用之前把闭包对象的地址存储到DX寄存器中，跟C++中的thiscall非常类似。之前有很多读者在反编译Function Value调用代码时，总会看到为DX寄存器赋值，并为此感到疑惑，这就是原因。调用者不必区分是不是闭包、有没有捕获列表，实际上也区分不了，只能统一作为闭包来处理，所以总要通过DX传递地址。如果Function Value背后不是闭包，这个地址就不会被用到，也不会造成什么影响。

4. 闭包与变量逃逸

本节第3部分打印闭包对象结构定义的时候发现跟变量逃逸还有些关系。事实上变量逃逸跟闭包之间的关系很密切，因为Function Value本身就是个指针，编译器也可以按照同样的方式来分析Function Value有没有逃逸。如果Function Value没有逃逸，那就可以不用在堆上分配闭包对象了，分配在栈上即可。使用一个示例进行验证，代码如下：

```
//第 3 章/code_3_23.go
func sc(n int) int {
    f := func() int {
        return n
    }
    return f()
}
```

代码逻辑过于简单，为了避免闭包函数被编译器优化掉，编译时需要禁用内联优化，命令如下：

```
$ go build -gcflags='-l'
```

再来反编译sc（）函数，反编译命令及输出结果如下：

```
$ go tool objdump -S -s '^main.sc$' gom.exe
TEXT main.sc(SB) C:/gopath/src/fengyoulin.com/gom/code_3_23.go
func sc(n int) int {
  0x4b68f0              65488b0c2528000000      MOVQ GS:0x28, CX
  0x4b68f9              488b8900000000          MOVQ 0(CX), CX
  0x4b6900              483b6110                CMPQ 0x10(CX), SP
  0x4b6904              764b                    JBE 0x4b6951
  0x4b6906              4883ec20                SUBQ $0x20, SP
  0x4b690a              48896c2418              MOVQ BP, 0x18(SP)
  0x4b690f              488d6c2418              LEAQ 0x18(SP), BP
        f := func() int {
  0x4b6914              0f57c0                  XORPS X0, X0
  0x4b6917              0f11442408              MOVUPS X0, 0x8(SP)
  0x4b691c              488d053d000000          LEAQ main.sc.func1(SB), AX
  0x4b6923              4889442408              MOVQ AX, 0x8(SP)
  0x4b6928              488b442428              MOVQ 0x28(SP), AX
  0x4b692d              4889442410              MOVQ AX, 0x10(SP)
        return f()                                                      //这一行
  0x4b6932              488b442408              MOVQ 0x8(SP), AX
  0x4b6937              488d542408              LEAQ 0x8(SP), DX
  0x4b693c              ffd0                    CALL AX
  0x4b693e              488b0424                MOVQ 0(SP), AX
  0x4b6942              4889442430              MOVQ AX, 0x30(SP)
  0x4b6947              488b6c2418              MOVQ 0x18(SP), BP
  0x4b694c              4883c420                ADDQ $0x20, SP
  0x4b6950              c3                      RET
func sc(n int) int {
  0x4b6951              e88a56faff              CALL runtime.morestack_noctxt(SB)
  0x4b6956              eb98                    JMP main.sc(SB)
```

首先梳理一下return f（）之前的6行汇编代码：

（1）XORPS和MOVUPS这两行利用128位的寄存器X0，把栈帧上从位移8字节开始的16字节清零，这段区间就是sc（）函数的局部变量区，正好符合捕获了一个int变量的闭包对象大小。

（2）LEAQ和MOVQ把闭包函数的地址复制到栈帧上位移8字节处，正是闭包对象中的函数指针。

（3）接下来的两个MOVQ把sc（）函数的参数n的值复制到栈帧上位移16字节处，也就是闭包捕获

列表中的int变量。

这段代码在栈上构造出所需的闭包对象，如图3-13所示。

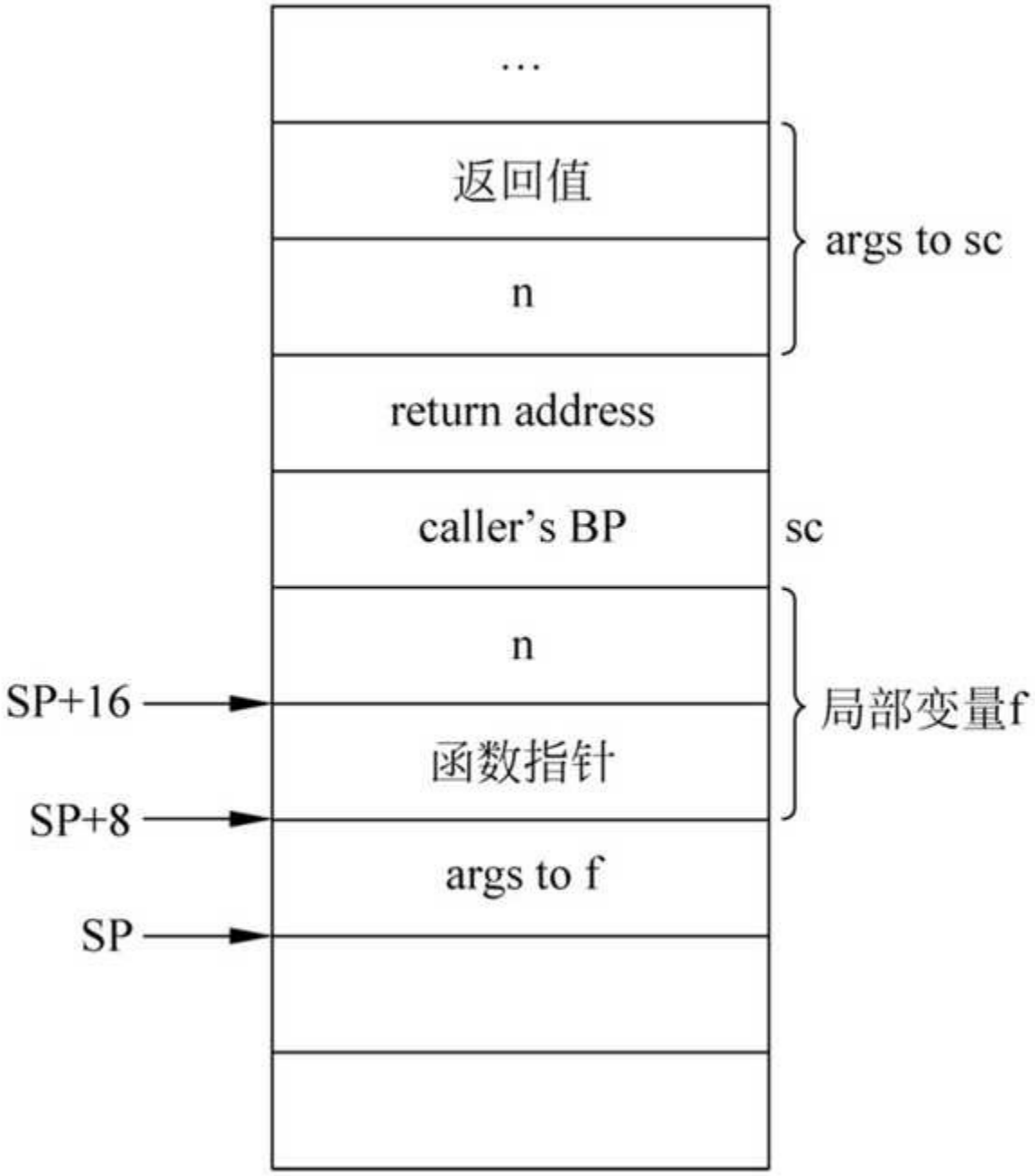

图3-13 sc（）函数中构造的闭包对象f

再梳理一下return之后的5行汇编代码：

（1）MOVQ把闭包函数的地址复制到AX寄存器中，LEAQ把闭包对象的地址存储到DX寄存器中。

（2）CALL指令调用闭包函数，接下来的两条MOVQ把闭包函数的返回值复制到sc（）函数的返回值。

所以，这段代码实际调用了闭包函数，如图3-14所示，闭包函数执行时直接把闭包对象捕获的n复制到f（）函数的返回值空间，然后f的返回值会复制到sc（）函数的返回值空间。

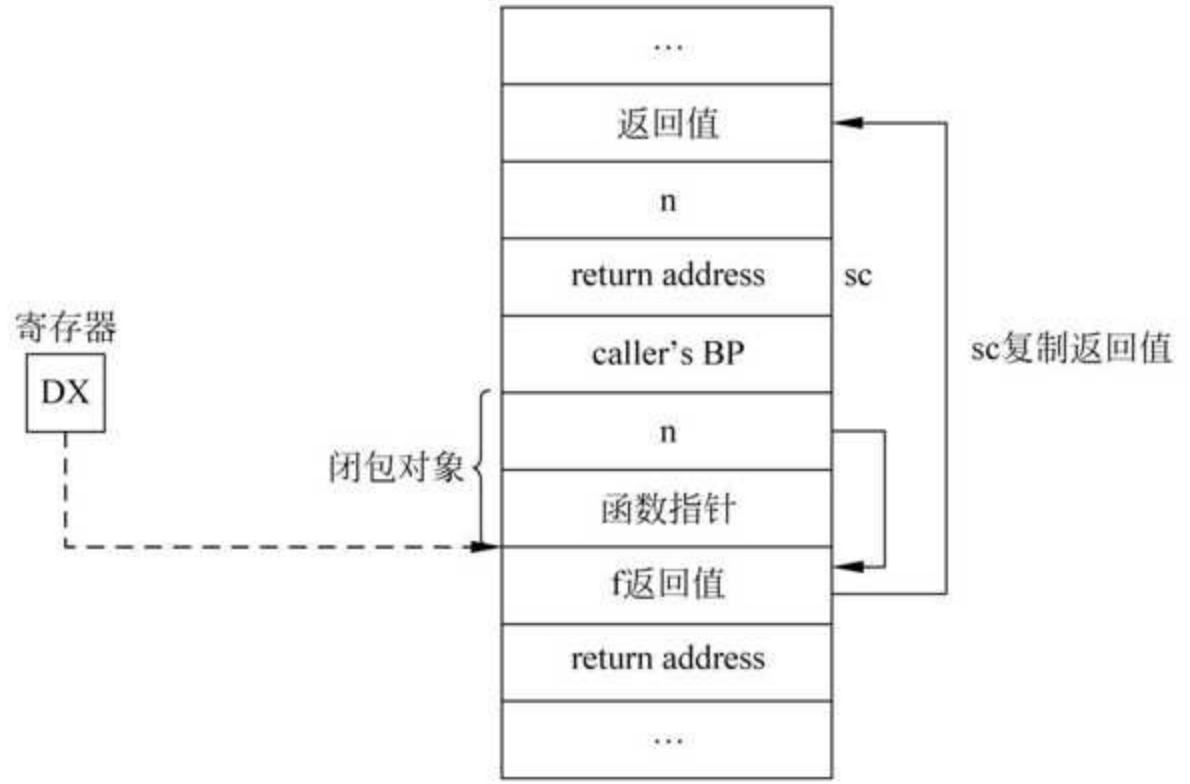

图3-14 调用闭包函数f（）

整体来看，上述代码逻辑除了闭包对象分配在栈上之外，并没有其他的不同，不过还是能够说明逃逸分析在起作用。

还有一个需要探索的问题，就是关于闭包对象的捕获列表，捕获的是变量的值还是地址？这实际上也跟逃逸分析有着密切关系。下面先从语义的角度来看一下，什么时候捕获值和什么时候捕获地址。

根据之前的经验，只要在闭包函数中改动一下捕获的变量，就会变成捕获地址。在第3章/code_3_23示例代码的基础上加一行自增语句，代码如下：

```
//第 3 章/code_3_24.go
func sc(n int) int {
    f := func() int {
        n++
        return n
    }
    return f()
}
```

构建时还是要禁用内联优化，再通过反编译检查闭包捕获的类型，发现确实捕获了变量n的地址。在上一示例代码中，没有修改n的时候，捕获的是值，所以可以这样推断：编译器总是倾向于捕获变量的值，除非有必要捕获地址。

从语义角度来讲，闭包捕获变量并不是要复制一个副本，变量无论被捕获与否都应该是唯一的，所谓捕获只是编译器为闭包函数访问外部环境中的变量搭建了一个桥梁。这个桥梁可以复制变量的值，也可以存储变量的地址。只有在变量的值不会再改变的前提下，才可以复制变量的值，否则就会出现不一致错误。

准备一个示例，代码如下：

```
//第 3 章/code_3_25.go
func sc(n int) int {
    n++
    f := func() int {
        return n
    }
    return f()
}
```

经过反编译验证，其中的闭包会捕获值，因为变量自增发生在闭包捕获之前，在闭包捕获之后变量的值不会再改变。

准备另一个示例，代码如下：

```
//第 3 章/code_3_26.go
func sc(n int) int {
    f := func() int {
        return n
    }
    n++
    return f()
}
```

这里的闭包会捕获地址，因为自增语句使变量的值发生了改变，而这个改变又在闭包捕获变量之后。

事实上，对于上述这种闭包对象未逃逸的场景，如果没有禁用内联优化，编译器大概率会把闭包函数优化掉。上述探索的意义主要在于明确编译器在捕获值上的倾向，也就是只要逻辑允许捕获值，就不会捕获地址。如果都捕获地址，更符合语义层面的变量唯一性约束，那么编译器为什么要尽最大可能性捕获值呢？

结合变量逃逸的依赖传递性来思考就比较容易理解了。如果闭包对象逃逸了，则所有被捕获地址的变量都要跟随着一起逃逸，而捕获值就没有逃逸的问题了，可以减少不必要的堆分配，进而优化程序性能。

3.4 defer

Go语言的defer是个很有意思的特性，可通俗地翻译为延迟调用。简单描述就是，跟在defer后面的函数调用不会立刻执行，而像是被注册到了当前函数中，等到当前函数返回之前，会按照FILO（First In Last Out）的顺序调用所有注册的函数。

需要注意的是，假如跟在defer后面的语句中包含多次函数调用，那么只有最后的那个会被延迟调用，而其他的都会立刻执行。准备示例代码如下：

```
//第 3 章/code_3_27.go
func fn() func() {
    return func() {
        println("defer")
    }
}
```

fn（）函数会返回一个Function Value，那么defer fn（）（）会立刻调用fn（）函数，实际被延迟调用的是fn（）函数返回的Function Value。

被延迟调用的函数的参数也会立刻求值，如果依赖某个函数的返回值，则相应函数也会立刻被调用，示例代码如下：

```
defer close(getChan())
```

在上述代码中的close（）函数被延迟调用，而getChan（）函数则立刻被调用。那么延迟执行到底是如何实现的呢？这个就是本节将要探索的内容，接下来的3.4.1～3.4.3节分别就Go语言1.12、1.13和1.14版本进行研究，因为defer的实现在这几个版本之间发生了较大的变化。

为了统一称谓，后文中将通过defer调用的函数称为defer函数，将使用defer关键字调用某函数的函数称为当前函数，将当前函数通过defer关键字来延迟调用某defer函数这一动作称为注册。

3.4.1 最初的链表

使用1.12版本的SDK构建一个示例，代码如下：

```
//第 3 章/code_3_28.go
package main

func main() {
    println(df(10))
}

func df(n int) int {
    defer func(i *int) {
        *i = 2
    }(&n)
    return n
}
```

反编译得到可执行文件中的df（）函数，节选比较关键的汇编代码如下：

```
0x452fc6          4883ec20              SUBQ $0x20, SP
0x452fca          48896c2418            MOVQ BP, 0x18(SP)
0x452fcf          488d6c2418            LEAQ 0x18(SP), BP
0x452fd4          48c744243000000000    MOVQ $0x0, 0x30(SP)
  defer func() {
0x452fdd          c7042408000000        MOVL $0x8, 0(SP)
0x452fe4          488d05453d0200        LEAQ go.func.* + 58(SB), AX
0x452feb          4889442408            MOVQ AX, 0x8(SP)
0x452ff0          488d442428            LEAQ 0x28(SP), AX
0x452ff5          4889442410            MOVQ AX, 0x10(SP)
0x452ffa          e8c124fdff            CALL runtime.deferproc(SB)
0x452fff          85c0                  TESTL AX, AX
0x453001          751a                  JNE 0x45301d
  return n
0x453003          488b442428            MOVQ 0x28(SP), AX
0x453008          4889442430            MOVQ AX, 0x30(SP)
0x45300d          90                    NOPL
0x45300e          e88d2dfdff            CALL runtime.deferreturn(SB)
0x453013          488b6c2418            MOVQ 0x18(SP), BP
0x453018          4883c420              ADDQ $0x20, SP
0x45301c          c3                    RET
  defer func() {
0x45301d          90                    NOPL
0x45301e          e87d2dfdff            CALL runtime.deferreturn(SB)
0x453023          488b6c2418            MOVQ 0x18(SP), BP
0x453028          4883c420              ADDQ $0x20, SP
0x45302c          c3                    RET
```

汇编代码中调用了两个新的runtime函数，分别是runtime.deferproc（）函数和runtime.deferreturn（）函数。一直到Go 1.12版本，defer的实现都没有太大变化，代码中的defer都会被编译器转化为对runtime.deferproc（）函数的调用。

1. deferproc

Go语言中，每个goroutine都有自己的一个defer链表，而runtime.deferproc（）函数做的事情就是把defer函数及其参数添加到链表中，即本节所谓的注册。编译器还会在当前函数结尾处插入调用runtime.deferreturn（）函数的代码，该函数会按照FILO的顺序调用当前函数注册的所有defer函数。如果当前goroutine发生了panic（宕机），或者调用了runtime.Goexit（）函数，runtime的panic处理逻辑会按照FILO的顺序遍历当前goroutine的整个defer链表，并逐一调用defer函数，直到某个defer函数执行了recover，或者所有defer函数执行完毕后程序结束运行。

runtime.deferproc（）函数的原型如下：

```
func deferproc(siz int32, fn * funcval)
```

参数fn指向一个runtime.funcval结构，该结构被runtime用来支持Function Value，其中只定义了一个uintptr类型的成员，存储的是目标函数的地址。通过3.3节对Function Value的探索，已知Go语言用两级指针结构统一了函数指针和闭包，这个funcval结构就是用来支持两级指针的。如图3-15所示，deferproc（）函数的参数fn是第一级指针，funcval中的uintptr成员是第二级指针。

参数siz表示defer函数的参数占用空间的大小，这部分参数也是通过栈传递的，虽然没有出现在deferproc（）函数的参数列表里，但实际上会被编译器追加到fn的后面，示例代码中df（）函数调用deferproc（）函数时的函数栈帧如图3-16所示。注意defer函数的参数在栈上的fn后面，而不是在

funcval结构的后面。这点不符合正常的Go语言函数调用约定，属于编译器的特殊处理。

基于第3章/code_3_28.go反编译得到的汇编代码，整理出等价的伪代码如下：

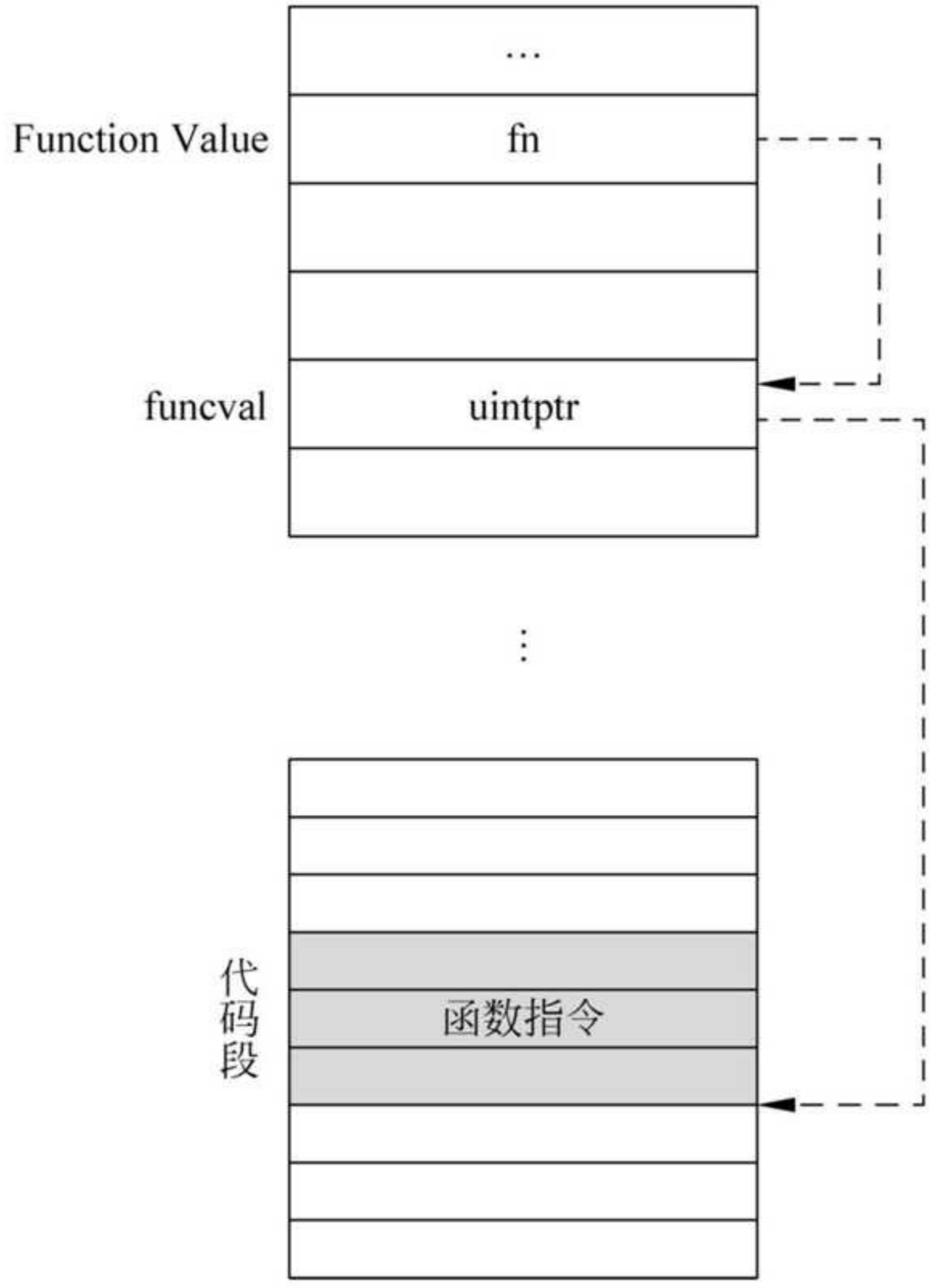

图3-15 funcval对Function Value两级指针的支持

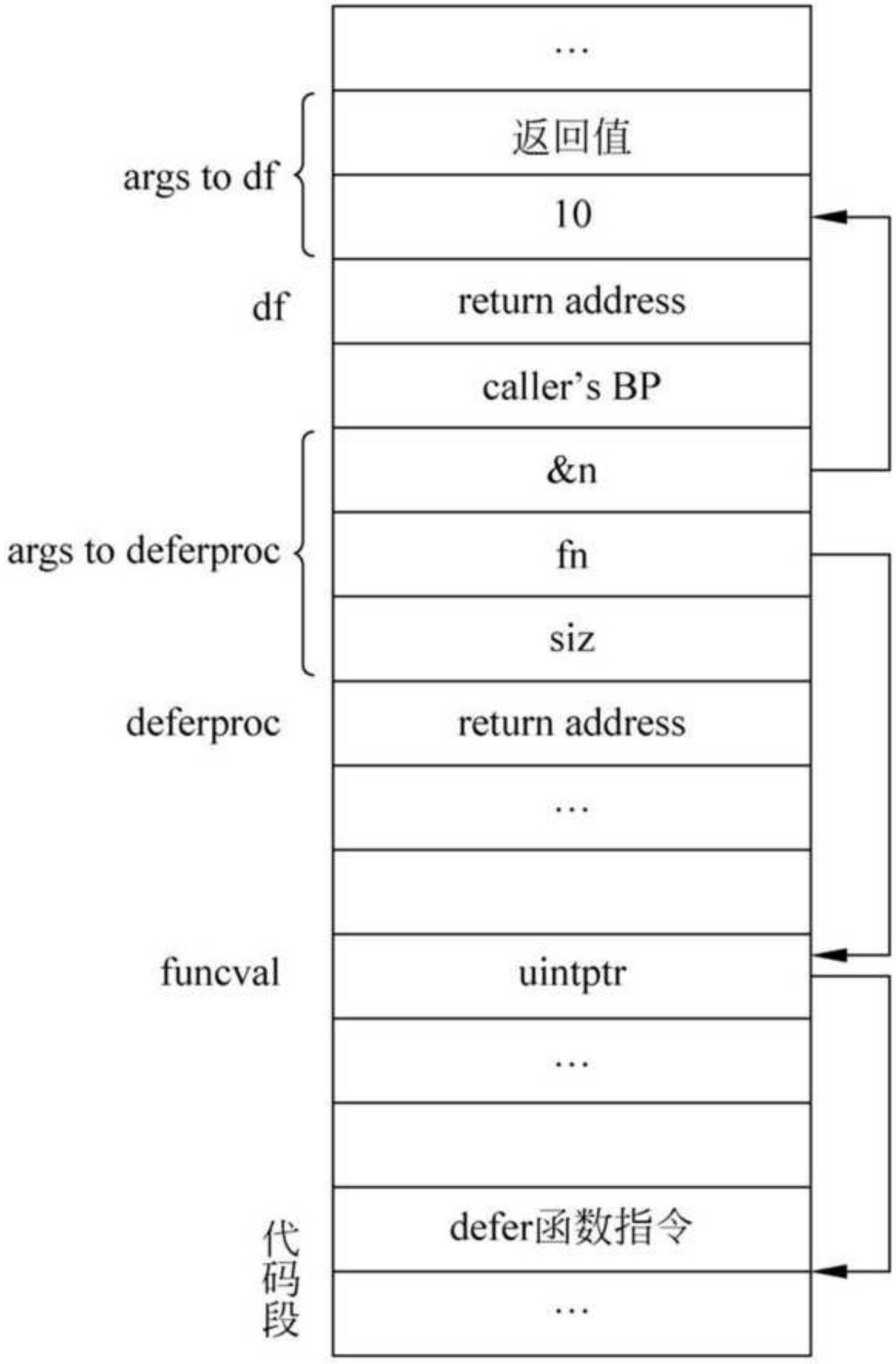

图3-16 df（）函数调用deferproc时的栈帧

```
func df(n int) (v int) {
    r := runtime.deferproc(8, df.func1, &n)
    if r > 0 {
        goto ret
    }
    v = n
    runtime.deferreturn()
    return
ret:
    runtime.deferreturn()
    return
}

func df.func1(i *int) {
    *i = 2
}
```

deferproc（）函数的返回值为0或非0时代表不同的含义，0代表正常流程，也就是已经把需要延迟执行的函数注册到了链表中，这种情况下程序可正常执行后续逻辑。返回值为1则表示发生了panic，并且当前defer函数执行了recover，这种情况会跳过当前函数后续的代码，直接执行返回逻辑。

还有一点需要特别注意一下，从函数原型来看，deferproc（）函数没有返回值，但实际上deferproc（）函数的返回值是通过AX寄存器返回的，这一点与一般的Go语言函数不同，却跟C语言的函数比较类似，等到3.5节讲解panic的时候再具体分析这么做的原因。

接下来看一下deferproc（）函数的具体实现，摘抄自runtime包的panic.go，代码如下：

```
//go:nosplit
func deferproc(siz int32, fn * funcval) { //arguments of fn follow fn
    if getg().m.curg != getg() {
        throw("defer on system stack")
    }

    sp := getcallersp()
    argp := uintptr(unsafe.Pointer(&fn)) + unsafe.Sizeof(fn)
    callerpc := getcallerpc()

    d := newdefer(siz)
    if d._panic != nil {
        throw("deferproc: d.panic != nil after newdefer")
    }
    d.fn = fn
    d.pc = callerpc
    d.sp = sp
    switch siz {
    case 0:
        //Do nothing.
    case sys.PtrSize:
        *(*uintptr)(deferArgs(d)) = *(*uintptr)(unsafe.Pointer(argp))
    default:
        memmove(deferArgs(d), unsafe.Pointer(argp), uintptr(siz))
    }

    return0()
}
```

通过getcallersp（）函数获取调用者的SP，也就是调用deferproc（）函数之前SP寄存器的值。这个值有两个用途，一是在deferreturn（）函数执行defer函数时用来判断该defer是不是被当前函数注册的，二是在执行recover的时候用来还原栈指针。

基于unsafe指针运算得到编译器追加在fn之后的参数列表的起始地址，存储在argp中。

通过getcallerpc（）函数获取调用者指令指针的位置，在amd64上实际就是deferproc（）函数的返回地址，从调用者df（）函数的视角来看就是CALL runtime.deferproc后面的那条指令的地址。这个地址主要用来在执行recover的时候还原指令指针。

调用newdefer（）函数分配一个runtime._defer结构，newdefer（）函数内部使用了两级缓冲池来避免频繁的堆分配，并且会自动把新分配的_defer结构添加到链表的头部。

创建好_defer结构，接下来就是赋值操作了，不过在那之前，我们先来看一下runtime._defer的定义，代码如下：

```
type _defer struct {
    siz     int32
    started bool
    sp      uintptr //sp at time of defer
    pc      uintptr
    fn      * funcval
    _panic  * _panic //panic that is running defer
    link    * _defer
}
```

（1）siz表示defer参数占用的空间大小，与deferproc（）函数的第1个参数一样。

（2）started表示有个panic或者runtime.Goexit（）函数已经开始执行该defer函数。

（3）sp、pc和fn已经解释过，此处不再赘述。

（4）_panic的值是在当前goroutine发生panic后，runtime在执行defer函数时，将该指针指向当前的_panic结构。

（5）link指针用来指向下一个_defer结构，从而形成链表。

现在的问题是_defer中没有发现用来存储defer函数参数的空间，参数应该被存储到哪里？

实际上runtime.newdefer（）函数用了和编译器一样的手段，在分配_defer结构的时候，后面额外追加了siz大小的空间，如图3-17所示，所以deferproc（）函数接下来会将fn、callerpc、sp都复制到_defer结构中相应的字段，然后根据siz大小来复制参数，最后通过return0（）函数来把返回值0写入AX寄存器中。

deferproc（）函数的大致逻辑就是这样，它把defer函数的相关数据存储在runtime._defer这个结构中并添加到了当前goroutine的defer链表头部。

通过deferproc（）函数注册完一个defer函数后，deferproc（）函数的返回值是0。后面如果发生了panic，又通过该defer函数成功recover，那么指令指针和栈指针就会恢复到这里设置的pc、sp处，看起来就像刚从runtime.deferproc（）函数返回，只不过返回值为1，编译器插入的if语句继而会跳过函数体，仅执行末尾的deferreturn（）函数。

2．deferreturn

在正常情况下，注册过的defer函数是由runtime.deferreturn（）函数负责执行的，正常情况指的就是没有panic或runtime.Goexit（）函数，即当前函数完成执行并正常返回时。deferreturn（）函数的代码如下：

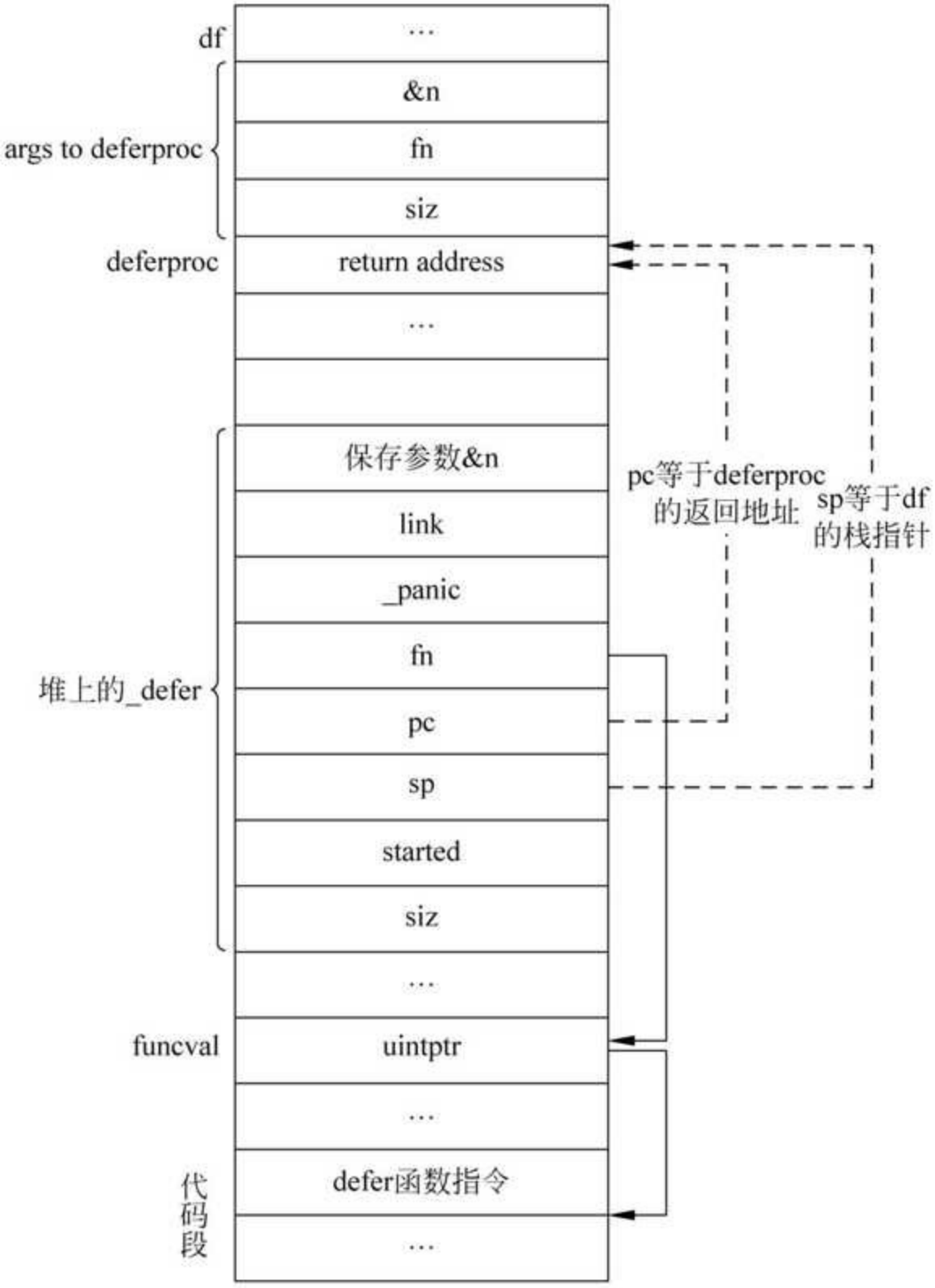

图3-17 deferproc执行中为_defer赋值

```
//go:nosplit
func deferreturn(arg0 uintptr) {
    gp := getg()
    d := gp._defer
    if d == nil {
        return
    }
    sp := getcallersp()
    if d.sp != sp {
        return
    }

    switch d.siz {
    case 0:
        //Do nothing.
    case sys.PtrSize:
        *(*uintptr)(unsafe.Pointer(&arg0)) = *(*uintptr)(deferArgs(d))
    default:
        memmove(unsafe.Pointer(&arg0), deferArgs(d), uintptr(d.siz))
    }
    fn := d.fn
    d.fn = nil
    gp._defer = d.link
    freedefer(d)
    jmpdefer(fn, uintptr(unsafe.Pointer(&arg0)))
}
```

值得注意的是参数arg0的值没有任何含义，实际上编译器并不会传递这个参数，deferreturn（）函数内部通过它获取调用者栈帧上args to callee区间的起始地址，从而可以将defer函数所需参数复制到该区间。defer函数的参数个数要比编译器传给deferproc（）函数的参数还少两个，所以调用者的args to callee区间大小肯定足够，不必担心复制参数会覆盖掉栈帧上的其他数据。

deferreturn（）函数的主要逻辑如下：

（1）若defer链表为空，则直接返回，否则获得第1个_defer的指针d，但并不从链表中移除。

（2）判断d.sp是否等于调用者的SP，即判断d是否由当前函数注册，如果不是，则直接返回。

（3）如果defer函数有参数，d.siz会大于0，就将参数复制到栈上&arg0处。

（4）将d从defer链表移除，链表头指向d.link，通过runtime.freedefer（）函数释放d。和runtime.newdefer（）函数对应，runtime.freedefer（）函数会把d放回缓冲池中，缓冲池内部按照defer函数参数占用空间的多少分成了5个列表，对于参数太多且占用空间太大的d，超出了缓冲池的处理范围则不会被缓存，后续会被GC回收。

（5）通过runtime.jmpdefer（）函数跳转到defer函数去执行。

runtime.jmpdefer（）函数是用汇编语言实现的，amd64平台下的实现代码如下：

```
TEXT runtime·jmpdefer(SB), NOSPLIT, $0-16
    MOVQ    fv+0(FP), DX        //fn
    MOVQ    argp+8(FP), BX      //caller sp
    LEAQ    -8(BX), SP          //caller sp after CALL
    MOVQ    -8(SP), BP          //restore BP as if deferreturn returned
    SUBQ    $5, (SP)            //return to CALL again
    MOVQ    0(DX), BX
    JMP     BX                  //but first run the deferred function
```

第2行把fn赋值给DX寄存器，3.3节中已经讲过Function Value调用时用DX寄存器传递闭包对象地址。接下来的3行代码通过设置SP和BP来还原deferreturn（）函数的栈帧，结合最后一条指令是跳转到defer函数而不是通过CALL指令来调用，这样从调用栈来看就像是deferreturn（）函数的调用者直接调用了defer函数。

还有一点需要特别注意，jmpdefer（）函数会调整返回地址，在amd64平台下会将返回地址减5，即一条CALL指令的大小，然后才会跳转到defer函数去执行。这样一来，等到defer函数执行完毕返回的时候，刚好会返回编译器插入的runtime.deferreturn（）函数调用之前，从而实现无循环、无递归地重复调用deferreturn（）函数。直到当前函数的所有defer都执行完毕，deferreturn（）函数会在第1、第2步判断时返回，不经过jmpdefer（）函数调整栈帧和返回地址，从而结束重复调用。

使用deferproc（）函数实现defer的好处是通用性比较强，能够适应各种不同的代码逻辑。例如if语句块中的defer和循环中的defer，示例代码如下：

```
//第3章/code_3_29.go
func fn(n int) (r int) {
    if n & 1 != 0 {
        defer func() {
            r <<= 1
        }()
    }
    for i := 0; i < n; i++{
        defer func() {
            r <<= 1
        }()
    }
    return n
}
```

因为defer函数的注册是运行阶段才进行的，可以跟代码逻辑很好地整合在一起，所以像if这种条件分支不用完成额外工作就能支持。由于每个runtime._defer结构都是基于缓冲池和堆动态分配的，所以即使不定次数的循环也不用额外处理，多次注册互不干扰。

但是链表与堆分配组合的最大缺点就是慢，即使用了两级缓冲池来优化runtime._defer结构的分配，性能方面依然不太乐观，所以在后续的版本中就开始了对defer的优化之旅。

3.4.2 栈上分配

在1.13版本中对defer做了一点小的优化，即把runtime._defer结构分配到当前函数的栈帧上。很明显这不适用于循环中的defer，循环中的defer仍然需要通过deferproc（）函数实现，这种优化只适用于只会执行一次的defer。

编译器通过runtime.deferprocStack（）函数来执行这类defer的注册，相比于runtime.deferproc（）函数，少了通过缓冲池或堆分配_defer结构的步骤，性能方面还是稍有提升的。deferprocStack（）函数的代码如下：

```
//go:nosplit
func deferprocStack(d * _defer) {
    gp := getg()
    if gp.m.curg != gp {
        //go code on the system stack can't defer
        throw("defer on system stack")
    }
    //siz and fn are already set.
    d.started = false
    d.heap = false
    d.sp = getcallersp()
    d.pc = getcallerpc()

    *(*uintptr)(unsafe.Pointer(&d._panic)) = 0
    *(*uintptr)(unsafe.Pointer(&d.link)) = uintptr(unsafe.Pointer(gp._defer))
    *(*uintptr)(unsafe.Pointer(&gp._defer)) = uintptr(unsafe.Pointer(d))

    return0()
}
```

runtime._defer结构中新增了一个bool型的字段heap来表示是否为堆上分配，对于这种栈上分配的_defer结构，deferreturn（）函数就不会用freedefer（）函数进行释放了。因为编译器在栈帧上已经把_defer结构的某些字段包括后面追加的fn的参数都准备好了，所以deferprocStack（）函数这里只需为剩余的几个字段赋值，与deferproc（）函数的逻辑基本一致。最后几行中通过unsafe.Pointer做类型转换再赋值，源码注释中解释为避免写屏障，暂时理解成为提升性能就行了，这个写屏障到第8章再详细介绍。

同样使用第3章/code_3_28.go，经过Go 1.13编译器转换后的伪代码如下：

```
func df(n int) (v int) {
    var d struct {
        runtime._defer
        n *int
    }
    d.siz = 8
    d.fn = df.func1
    d.n = &n
    r := runtime.deferprocStack(&d)
    if r > 0 {
        goto ret
    }
    v = n
    runtime.deferreturn()
    return
ret:
    runtime.deferreturn()
    return
}

func df.func1(i *int) {
    *i *= 2
}
```

值得注意的是，如图3-18所示，编译器需要根据defer函数的参数和返回值占用的空间，来为df（）函数栈帧的args to callee区间分配足够的大小，以使deferreturn（）函数向栈帧上复制defer函数参数时不会覆盖其他区间的数据。

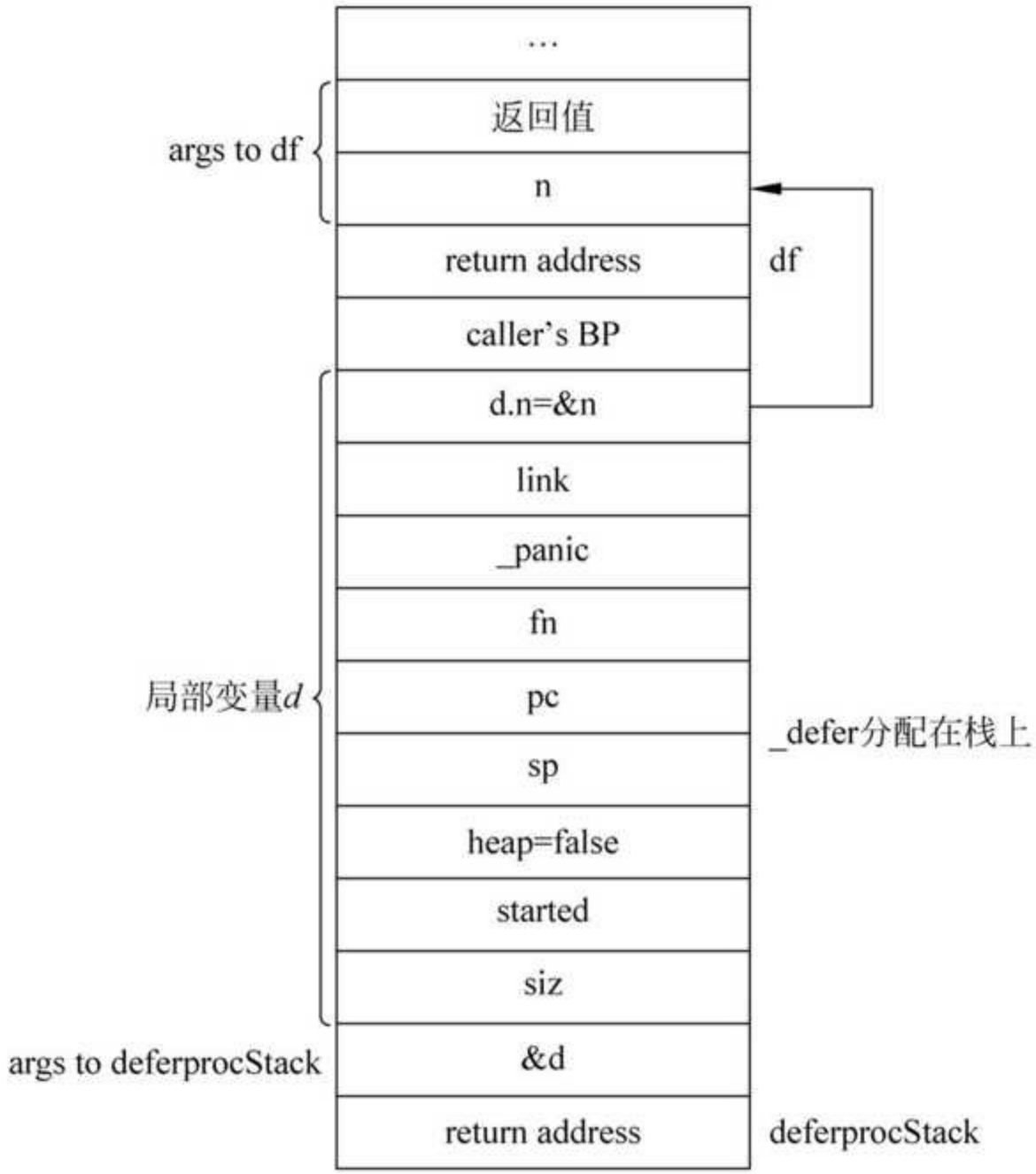

图3-18 df（）函数调用deferprocStack（）时的栈帧

栈上分配_defer这种优化只是节省了_defer结构的分配、释放时间，仍然需要将defer函数添加到链表中，在调用的时候也还要复制栈上的参数，整体提升比较有限。经过笔者的Benchmark测试，1.13版本比1.12版本大约有25%的性能提升。

3.4.3 高效的**open coded defer**

经过Go 1.13版本对defer的优化，虽然性能上得到了提升，但是远没有达到开发者的预期。因为在并发场景下，defer经常被用来释放资源，例如函数返回时解锁Mutex等，相比之下defer自身的开销就有些大了。

因此在Go 1.14版本中又进行了一次优化，这次优化也是针对那些只会执行一次的defer。编译器不再基于链表实现这类defer，而是将这类defer直接展开为代码中的函数调用，按照倒序放在函数返回前去执行，这就是所谓的open coded defer。

依然使用第3章/code_3_28.go，在1.14版本中经编译器转换后的伪代码如下：

```
func df(n int) (v int) {
    v = n
    func(i *int) {
        *i *= 2
    }(&n)
    return
}
```

这里会有两个问题：

（1）如何支持嵌套在if语句块中的defer？

（2）当发生panic时，如何保证这些defer得以执行呢？

第1个问题其实并不难解决，可以在栈帧上分配一个变量，用每个二进制位来记录一个对应的defer函数是否需要被调用。Go语言实际上用了一字节作为标志，可以最多支持8个defer，为什么不支持更多呢？笔者是这样理解的，open coded defer本来就是为了提高性能而设计的，一个函数中写太多defer，应该是不太在意这种层面上的性能了。

还需要考虑的一个问题是，deferproc（）函数在注册的时候会存储defer函数的参数副本，defer函数的参数经常是当前函数的局部变量，即使它们后来被修改了，deferproc（）函数存储的副本也是不会变的，副本是注册那一时刻的状态，所以在open coded defer中编译器需要在当前函数栈帧上分配额外的空间来存储defer函数的参数。

综上所述，一个示例代码如下：

```
//第 3 章/code_3_30.go
func fn(n int) (r int) {
    if n > 0 {
        defer func(i int) {
            r <<= i
        }(n)
    }
    n++
    return n
}
```

经编译器转换后的等价代码如下：

```
func fn(n int) (r int) {
    var f byte
    var i int
    if n > 0 {
        f |= 1
        i = n
    }
    n++
    r = n
    if f&1 > 0 {
        func(i int) {
            r <<= i
        }(i)
    }
    return
}
```

其中局部变量f就是专门用来支持if这类条件逻辑的标志位，局部变量i用作n在defer注册那一刻的副本，函数返回前根据标志位判断是否调用defer函数。示例中fn（）函数调用defer（）函数时栈帧如图3-19所示。

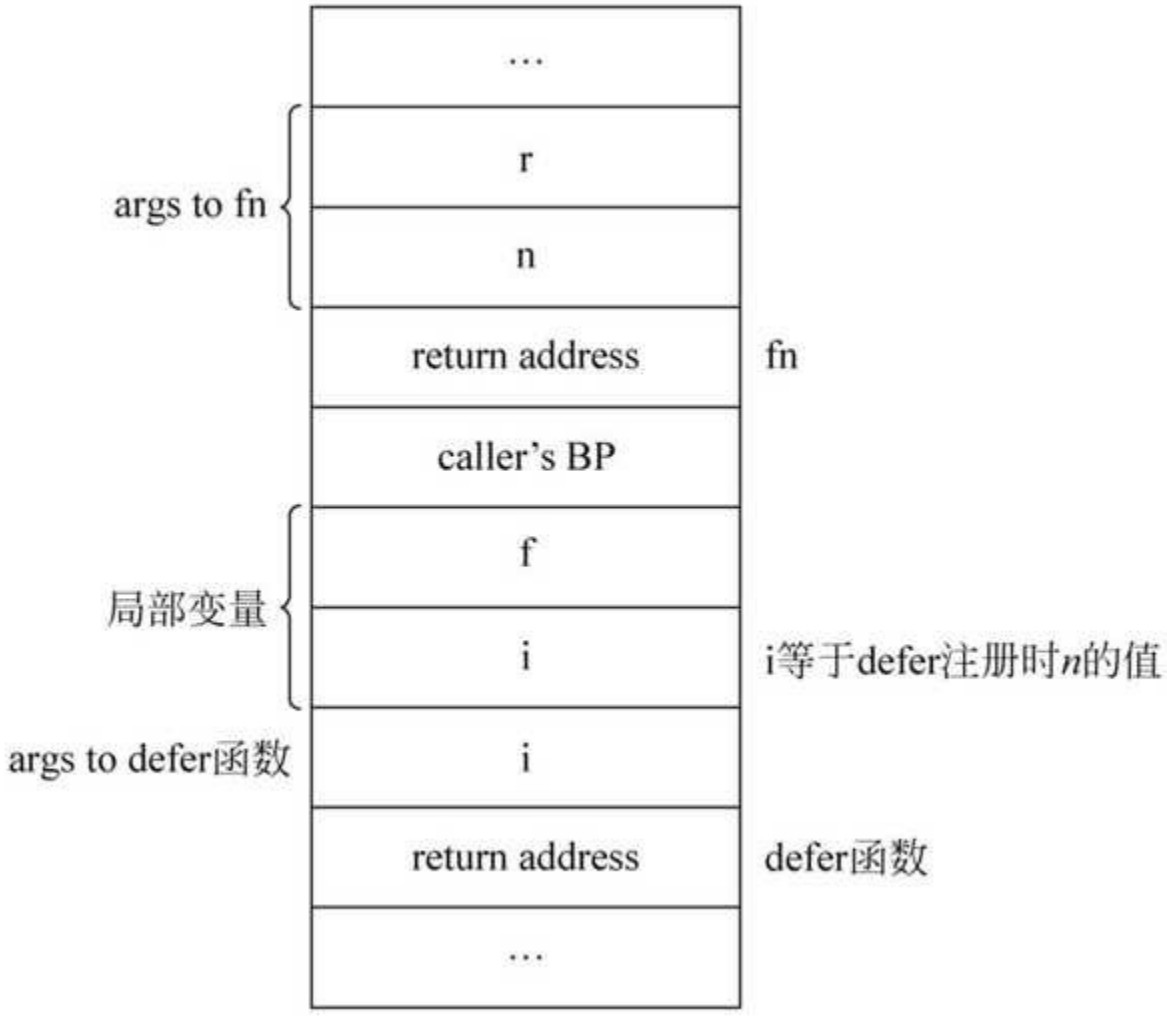

图3-19 fn（）函数通过open coded defer的方式调用defer函数

根据笔者的测试，open coded defer的性能比Go 1.12版本几乎提升了一个数量级，当然这是在代码没有发生panic的情况下。关于open coded defer如何保证在发生panic时能够被调用，也就是上面的第2个问题，将在3.5节中进行探索和介绍。

3.5 panic

panic（）和recover（）这对内置函数，实现了Go特有的异常处理流程。如果把panic（）函数视为其他语言中的throw语句，则带有recover（）函数的defer函数就起到了catch语句的作用。只有在defer函数中调用recover（）函数才有效，因为发生panic之后只有defer函数能够得到执行。Go语言在设计上保证所有的defer函数都能够得到调用，所以适合用defer来释放资源，即使发生panic也不会造成资源泄露。

本节结合Go语言runtime的部分源码，探索panic和recover的实现原理。

3.5.1 gopanic（）函数

内置panic（）函数是通过runtime中的gopanic（）函数实现的，代码中调用panic（）函数会被编译器转换为对gopanic（）函数的调用。在版本1.13和1.14中随着deferprocStack（）函数和open coded defer的引入，gopanic（）函数的实现也变得愈加复杂，但是核心逻辑并没有发生太大变化，所以本节还是从1.12版本的gopanic（）函数的源码开始进行讲解。

鉴于源码篇幅较长，本着先整体后局部的原则，把gopanic（）函数的源码按照逻辑划分成几部分，首先从宏观上看一下整个函数，代码如下：

```
func gopanic(e interface{}) {
    gp := getg()

    //一些校验

    var p _panic
    p.arg = e
    p.link = gp._panic
    gp._panic = (*_panic)(noescape(unsafe.Pointer(&p)))

    atomic.Xadd(&runningPanicDefers, 1)

    //for 循环

    preprintpanics(gp._panic)

    fatalpanic(gp._panic)
    *(*int)(nil) = 0
}
```

从函数原型来看，与内置函数panic（）完全一致，有一个interface{}类型的参数，这使gopanic（）函数可以接受任意类型的参数。函数首先通过getg（）函数得到当前goroutine的g对象指针gp，然后会进行一些校验工作，主要目的是确保处在系统栈、内存分配过程中、禁止抢占或持有锁的情况下不允许发生panic。接下来gopanic（）函数在栈上分配了一个_panic类型的对象p，把参数e赋值给p的arg字段，并把p安放到当前goroutine的_panic链表的头部，特意使用noescape（）函数来避免p逃逸，因为panic本身就是与栈的状态强相关的。

runtime._panic结构的定义代码如下：

```
//go:notinheap
type _panic struct {
    argp      unsafe.Pointer
    arg       interface{}
    link      *_panic
    recovered bool
    aborted   bool
}
```

（1）argp字段用来在defer函数执行阶段指向其args from caller区间的起始地址，到3.5.2节中再进一步分析argp字段更深层的意义。

（2）arg字段保存的就是传递给gopanic（）函数的参数。

（3）link字段用来指向链表中的下一个_panic结构。

（4）recovered字段表示当前panic已经被某个defer函数通过recover恢复。

（5）aborted字段表示发生了嵌套的panic，旧的panic被新的panic流程标记为aborted。

gopanic（）函数的源码中最关键的就是接下来的for循环了，在这个循环中逐个调用链表中的defer函数，并检测recover的状态。如果所有的defer函数都执行完后还是没有recover，则循环就会结束，最后的fatalpanic（）函数就会结束当前进程。for循环的主要代码如下：

```
for {
    d := gp._defer
    if d == nil {
        break
    }

    if d.started {
        if d._panic != nil {
            d._panic.aborted = true
        }
        d._panic = nil
        d.fn = nil
        gp._defer = d.link
        freedefer(d)
        continue
    }

    //1)调用 defer 函数
    //2)释放_defer 结构
    //3)检测 recover
}
```

每次循环开始都会从gp的_defer链表头部取一项赋值给d，直到链表为空时结束循环。接下来判断若d.started为真则表明当前是一个嵌套的panic，也就是在原有panic或Goexit（）函数执行defer函数的时候又触发了panic，因为触发panic的defer函数还没有执行完，所以还没有从链表中移除。这里会把d关联的旧的_panic设置为aborted，然后把d从链表中移除，并通过freedefer（）函数释放。

后续的3大块逻辑就是：调用defer函数、释放_defer结构和检测recover。

1．调用**defer**函数

调用defer函数的代码如下：

```
d.started = true
d._panic = (*_panic)(noescape(unsafe.Pointer(&p)))
p.argp = unsafe.Pointer(getargp(0))
reflectcall(nil, unsafe.Pointer(d.fn), deferArgs(d), uint32(d.siz), uint32(d.siz))
p.argp = nil
```

首先将d.started设置为true，这样如果defer函数又触发了panic，新的panic遍历defer链表时，就能通过started的值确定该defer函数已经被调用过了，避免重复调用。

然后为d._panic赋值，将d关联到当前panic对象p，并使用noescape（）函数避免p逃逸，这一步是为了后续嵌套的panic能够通过d._panic找到上一个panic。

接下来，p.argp被设置为当前gopanic（）函数栈帧上args to callee区间的起始地址，recover（）函数通过这个值来判断自身是否直接被defer函数调用，这个在3.5.2节中再详细讲解。

最关键的就是接下来的reflectcall（）函数调用了，它的函数声明代码如下：

```
func reflectcall(argtype *_type, fn, arg unsafe.Pointer, argsize uint32, retoffset uint32)
```

reflectcall（）函数的主要逻辑是根据argsize的大小在栈上分配足够的空间，然后把arg处的参数复制到栈上，复制的大小为argsize字节，然后调用fn（）函数，再把返回值复制回arg+retoffset处，复制的大小为argsize-retoffset字节，如果argtype不为nil，则根据argtype来应用写屏障。

在编译阶段，编译器无法知道gopanic（）函数在运行阶段会调用哪些defer函数，所以也无法预分配足够大的args to callee区间，只能通过reflectcall（）函数在运行阶段进行栈增长。defer函数的返回值虽然也会被复制回调用者的栈帧上，但是Go语言会将其忽略，所以这里不必应用写屏障。

2．释放**_defer**结构

释放_defer结构的代码如下：

```
if gp._defer != d {
    throw("bad defer entry in panic")
}
d._panic = nil
d.fn = nil
gp._defer = d.link

pc := d.pc
sp := unsafe.Pointer(d.sp)
freedefer(d)
```

调用完d.fn（）函数后，不应该出现gp._defer不等于d这种情况。假如在d.fn（）函数执行的过程中没有造成新的panic，那么所有新注册的defer都应该在d.fn（）函数返回的时候被deferreturn（）函数移出链表。假如d.fn（）函数执行过程中造成了新的panic，若没有recover，则不会再回到这里，若经recover之后再回到这里，则所有在d.fn（）函数执行过程中注册的defer也都应该在d.fn（）函数返回之前被移出链表。

其他几行代码就是把d的_panic和fn字段置为nil，然后从gp._defer链表中移除，把d的pc和sp字段保存在局部变量中，供接下来检测执行recover时使用，然后通过freedefer（）函数把d释放。此处的sp类型必须是指针，因为后续如果栈被移动，只有指针类型会得到更新。

3．检测**recover**

检测recover的代码如下：

```
if p.recovered {
    atomic.Xadd(&runningPanicDefers, -1)

    gp._panic = p.link
    for gp._panic != nil && gp._panic.aborted {
        gp._panic = gp._panic.link
    }
    if gp._panic == nil {
        gp.sig = 0
    }
    gp.sigcode0 = uintptr(sp)
    gp.sigcode1 = pc
    mcall(recovery)
    throw("recovery failed")
}
```

如果d.fn（）函数成功地执行了recover，则当前_panic对象p的recovered字段就会被设置为true，此处通过检测后就会执行recover逻辑。

首先把p从gp的_panic链表中移除，然后循环移除链表头部所有已经标为aborted的_panic对象。如果没有发生嵌套的panic，则此时gp._panic应该是nil，不为nil就表明发生了嵌套的panic，而且只是内层的panic被recover。代码的最后把局部变量sp和pc赋值给gp的sigcode0和sigcode1字段，然后通过mcall（）函数执行recovery（）函数。mcall（）函数会切换到系统栈，然后把gp作为参数来调用recovery（）函数。

recovery（）函数负责用存储在sigcode0和sigcode1中的sp和pc恢复gp的执行状态。recovery（）函数的主要逻辑代码如下：

```
func recovery(gp *g) {
    sp := gp.sigcode0
    pc := gp.sigcode1

    if sp != 0 && (sp < gp.stack.lo || gp.stack.hi < sp) {
        //省略打印错误信息的代码
        throw("bad recovery")
    }

    gp.sched.sp = sp
    gp.sched.pc = pc
    gp.sched.lr = 0
    gp.sched.ret = 1
    gogo(&gp.sched)
}
```

首先确保栈指针sp的值不能为0，并且还要在gp栈空间的上界与下界之间，然后把sp和pc赋值给gp.sched中对应的字段，并且把返回值设置为1。

调用gogo（）函数之后，gp的栈指针和指令指针就会被恢复到sp和pc的位置，而这个位置是deferproc（）函数通过getcallersp（）函数和getcallerpc（）函数获得的，即deferproc（）函数正常返回后的位置，所以经过某个defer函数执行recover（）函数后，当前goroutine的栈指针和指令指针会被恢复到deferproc（）函数刚刚注册完该defer函数后返回的位置，只不过返回值是1而不是0。编译器插入的代码会检测deferproc（）函数的返回值，这些在3.4.1节中已经介绍过了。

这里需要分析一下“为什么deferproc（）函数的返回值是通过AX寄存器而不是通过栈传递的”这个问题了。现在已经知道deferproc（）函数有两种可能的返回：第一种是正常执行，注册完defer函数后返回，这种情况下编译器是可以基于栈传递返回值的；第二种是panic后再经过recover返回，在gogo（）函数执行前，SP还没有恢复到调用deferproc（）函数时的位置，由于编译器会把defer函数的参数追加在deferproc（）函数的参数后面，所以返回值在栈上的位置还需要动态计算，实现起来有些复杂，所以还是通过寄存器传递返回值更加简单高效。

3.5.2 gorecover（）函数

3.5.1节中梳理了gopanic（）函数的主要逻辑，其中for循环每调用完一个defer函数都会检测p.recovered字段，如果值为true就执行recover逻辑。也就是说真正的recover逻辑是在gopanic（）函数中实现的，defer函数中调用了内置函数recover（），实际上只会设置_panic的一种状态。内置函数recover（）对应runtime中的gorecover（）函数，代码如下：

```
//go:nosplit
func gorecover(argp uintptr) interface{} {
    gp := getg()
    p := gp._panic
    if p != nil && !p.recovered && argp == uintptr(p.argp) {
        p.recovered = true
        return p.arg
    }
    return nil
}
```

内置函数recover（）是没有参数的，但是gorecover（）函数却有一个参数argp，这也是编译器做的手脚。编译器会把调用者的args from caller区间的起始地址作为参数传递给gorecover（）函数。示例代码如下：

```
//第3章/code_3_31.go
func fn() {
    defer func(a int) {
        recover()
        println(a)
    }(0)
}
```

经编译器转换后的等价代码如下：

```
func fn() {
    defer func(a int) {
        gorecover(uintptr(unsafe.Pointer(&a)))
        println(a)
    }(0)
}
```

为什么要传递这个argp参数呢？从代码逻辑来看，gorecover（）函数会把它跟当前_panic对象p的argp字段比较，只有相等时才会把p.recovered设置为true。如图3-20所示，p.argp的值是在gopanic（）函数的for循环中设置的，通过getargp（）函数获得的gopanic（）函数栈帧args to callee区间的起始地址。接下来才会通过reflectcall（）函数调用defer函数，所以在发生recover时，传递给gorecover（）函数的参数argp是defer函数栈帧上args from caller区间的起始地址，也就是reflectcall（）函数的args to callee区间的起始地址。

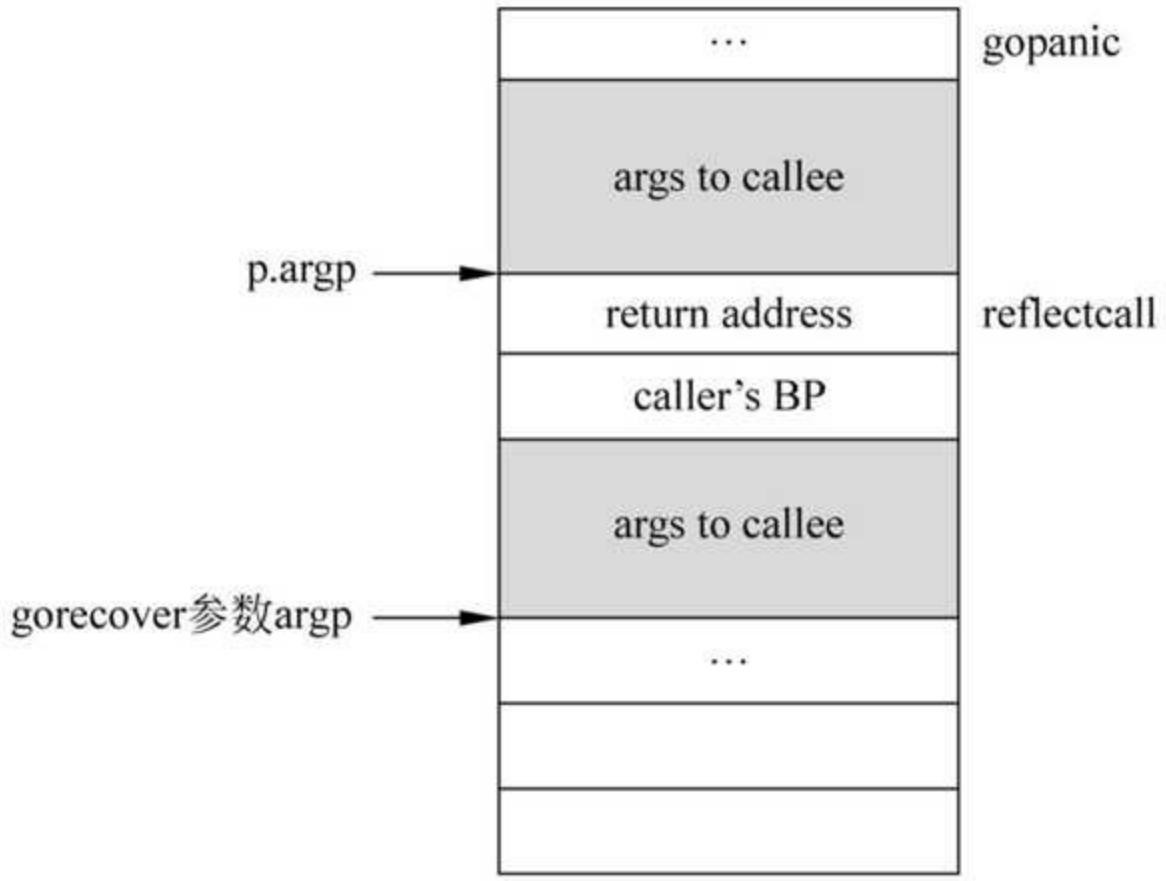

图3-20 p.argp和gorecover（）函数参数argp的关系

reflectcall（）函数是由gopanic（）函数调用的，那两者的args to callee区间的起始地址怎么可能相等呢？这个问题着实让笔者困惑不已。反复查看gopanic（）函数、gorecover（）函数的代码，以及reflectcall（）函数的汇编代码，加上反编译defer函数，都没有找到答案。最终还是忍不住反编译了reflectcall（）函数和它所依赖的一系列callXXX（）函数，这里XXX代表的就是函数栈帧上args to callee区间的大小。

reflectcall（）函数在源码中的代码如下：

```
TEXT ·reflectcall(SB), NOSPLIT, $0-32
    MOVLQZX argsize+24(FP), CX
    DISPATCH(runtime·call32, 32)
    DISPATCH(runtime·call64, 64)
    DISPATCH(runtime·call128, 128)
    //省略部分代码以节省篇幅
    DISPATCH(runtime·call268435456, 268435456)
    DISPATCH(runtime·call536870912, 536870912)
    DISPATCH(runtime·call1073741824, 1073741824)
    MOVQ    $runtime·badreflectcall(SB), AX
    JMP  AX
```

reflectcall（）函数会根据argsize的大小跳转到合适的callXXX（）函数去执行，看起来与p.argp的问题无关，通过反汇编来检验也没有发现什么特殊逻辑。再从源码中查看这组callXXX（）函数的实现，发现是通过宏定义实现的，宏定义的代码如下：

```
#define CALLFN(NAME,MAXSIZE) \
TEXT NAME(SB), WRAPPER, $MAXSIZE-32; \
    NO_LOCAL_POINTERS; \
    /* copy arguments to stack */ \
    MOVQ    argptr+16(FP), SI; \
    MOVLQZX argsize+24(FP), CX; \
    MOVQ    SP, DI; \
    REP;MOVSB; \
    /* call function */ \
    MOVQ    f+8(FP), DX; \
    PCDATA  $PCDATA_StackMapIndex, $0; \
    CALL    (DX); \
    /* copy return values back */ \
    MOVQ    argtype+0(FP), DX; \
    MOVQ    argptr+16(FP), DI; \
    MOVLQZX argsize+24(FP), CX; \
    MOVLQZX retoffset+28(FP), BX; \
    MOVQ    SP, SI; \
    ADDQ    BX, DI; \
    ADDQ    BX, SI; \
    SUBQ    BX, CX; \
    CALL    callRet<>(SB); \
    RET
```

从代码逻辑来看，这一系列callXXX（）函数才是实际完成reflectcall（）函数功能的地方。callXXX（）函数中完成了参数的复制、目标函数的调用及返回值的复制，但是看起来与p.argp也没有什么关系。为了避免编译器有什么背后的隐含逻辑，还是反编译一个call32（）函数看一下，代码如下：

```
$ go tool objdump -S -s '^runtime.call32$' gom.exe
TEXT runtime.call32(SB) C:/go/1.12.17/go/src/runtime/asm_amd64.s
0x4489a0        65488b0c2528000000      MOVQ GS:0x28, CX
0x4489a9        488b8900000000          MOVQ 0(CX), CX
0x4489b0        483b6110                CMPQ 0x10(CX), SP
0x4489b4        7659                    JBE 0x448a0f
0x4489b6        4883ec28                SUBQ $0x28, SP
0x4489ba        48896c2420              MOVQ BP, 0x20(SP)
0x4489bf        488d6c2420              LEAQ 0x20(SP), BP
0x4489c4        488b5920                MOVQ 0x20(CX), BX                       //10
0x4489c8        4885db                  TESTQ BX, BX                            //11
0x4489cb        7549                    JNE 0x448a16                            //12
0x4489cd        488b742440              MOVQ 0x40(SP), SI
0x4489d2        8b4c2448                MOVL 0x48(SP), CX
0x4489d6        4889e7                  MOVQ SP, DI
0x4489d9        f3a4                    REP; MOVSB DS:0(SI), ES:0(DI)
0x4489db        488b542438              MOVQ 0x38(SP), DX
0x4489e0        ff12                    CALL 0(DX)
0x4489e2        488b542430              MOVQ 0x30(SP), DX
0x4489e7        488b7c2440              MOVQ 0x40(SP), DI
0x4489ec        8b4c2448                MOVL 0x48(SP), CX
0x4489f0        8b5c244c                MOVL 0x4c(SP), BX
0x4489f4        4889e6                  MOVQ SP, SI
0x4489f7        4801df                  ADDQ BX, DI
0x4489fa        4801de                  ADDQ BX, SI
0x4489fd        4829d9                  SUBQ BX, CX
0x448a00        e86bffffff              CALL callRet(SB)
0x448a05        488b6c2420              MOVQ 0x20(SP), BP
0x448a0a        4883c428                ADDQ $0x28, SP
0x448a0e        c3                      RET
0x448a0f        e86cfdffff              CALL runtime.morestack_noctxt(SB)
0x448a14        eb8a                    JMP runtime.call32(SB)
0x448a16        488d7c2430              LEAQ 0x30(SP), DI                       //33
0x448a1b        48393b                  CMPQ DI, 0(BX)                          //34
0x448a1e        75ad                    JNE 0x4489cd                            //35
0x448a20        488923                  MOVQ SP, 0(BX)                          //36
0x448a23        eba8                    JMP 0x4489cd                            //37
```

除去prolog、epilog和与上述宏定义对应的代码，可以看到第10～12行和第33～37行是被编译器额外插入的。这几行代码的逻辑就是：如果gp._panic不为nil且gp._panic.argp的值等于当前函数栈帧args from caller区间的起始地址，就把它的值改成当前函数栈帧args to callee区间的起始地址。因为reflectcall（）函数没有移动栈指针，而且是通过JMP指令跳转到call32（）函数的，所以当前函数栈帧的args from caller区间就是reflectcall（）函数的args from caller区间。也就是说，通过在callXXX系列函数中对gp._panic.argp进行修正，使gorecover（）函数中的相等比较得以成立。与编译器插入的这些指令等价的Go代码如下：

```
gp := getg()
if gp._panic != nil {
    if gp._panic.argp == uintptr(unsafe.Pointer(&argtype)) {
        gp._panic.argp = getargp(0)
    }
}
```

费这么大的劲，gorecover（）函数中这个相等比较的意义是什么呢？其实，是为了实现Go语言对recover强加的一条限制：必须在defer函数中直接调用recover（）函数才有用，不可嵌套在其他函数中。recover（）函数调用有效的示例代码如下：

```
//第 3 章/code_3_32.go
func fn() {
    defer func() {
        recover()
    }()
}
```

recover（）函数调用无效的示例代码如下：

```
//第 3 章/code_3_33.go
func fn() {
    defer func() {
        r()
    }()
}

func r() {
    recover()
}
```

笔者认为这种限制是必要的，Go语言的recover与其他语言的try和catch有明显的不同，即不像catch语句那样能够限定异常的类型。如果没有对recover的这种限制，就会使代码行为变得不可控，panic可能经常会被某个深度嵌套的recover恢复，这并不是开发者想要的。

3.5.3 嵌套的panic

Go语言的panic是支持嵌套的，第1个panic在执行defer函数的时候可能会注册新的defer函数，也可能会触发新的panic。如果新的panic被新注册的defer函数中的recover恢复，则旧的panic就会继续执行，否则新的panic就会把旧的panic置为aborted。理解嵌套panic的关键就是关注defer链表和panic链表的变化，本节用两个简单的例子来加深一下理解。

先看一个简单的panic嵌套的例子，代码如下：

```
//第 3 章/code_3_34.go
func fn() {
    defer func() {
        panic("2")
    }()
    panic("1")
}
```

fn（）函数首先将一个defer函数注册到当前goroutine的defer链表头部，记为defer1，然后当panic（"1"）执行时，会在当前goroutine的_panic链表中新增一个_panic结构，记为panic1，panic1触发defer执行，defer1中started字段会被标记为true，_panic字段会指向panic1，如图3-21所示。

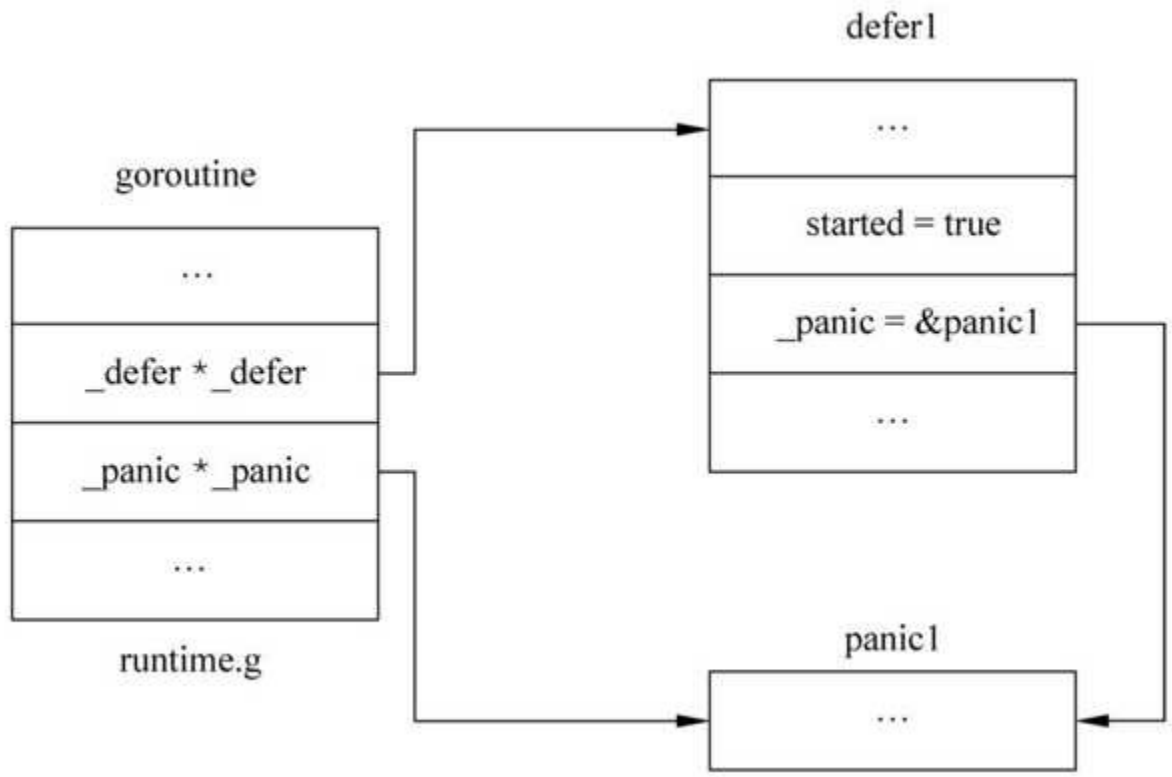

图3-21 panic2执行前的_defer链表和_panic链表

然后执行到panic（"2"）这里，也会在当前goroutine的_panic链表中新增一项，记为panic2。如图3-22所示，panic2同样会去执行defer链表，通过defer1记录的_panic字段找到panic1，并将其标记为aborted，然后移除defer1，处理defer链表中的后续节点。

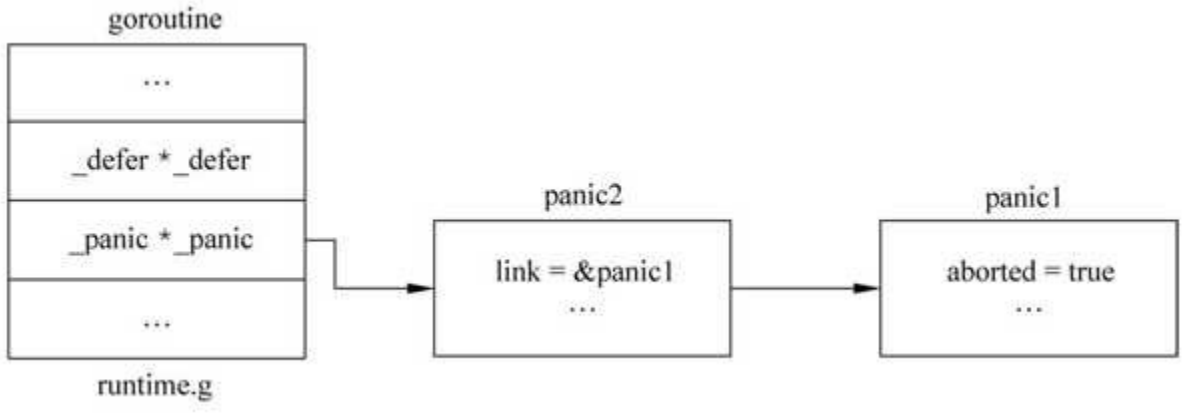

图3-22 panic2执行后的_defer链表和_panic链表

接下来，在第3章/code_3_34.go的defer函数中嵌套一个带有recover的defer函数，代码如下：

```
//第 3 章/code_3_35.go
func fn() {
    defer func() {
        defer func() {
            recover()
        }
        panic("2")
    }()
    panic("1")
}
```

依然把fn（）函数首先注册的defer函数记为defer1，把接下来执行的panic记为panic1，此时goroutine的_defer链表和_panic链表与图3-21中的链表并无不同。只不过当panic1触发defer1执行时，会再次注册一个defer函数，记为defer2，然后才会执行到panic（"2"），这里触发第二次panic，在_panic链表中新增一项，记为panic2。在panic2执行defer链表之前，_defer链表和_panic链表的情况如图3-23所示。

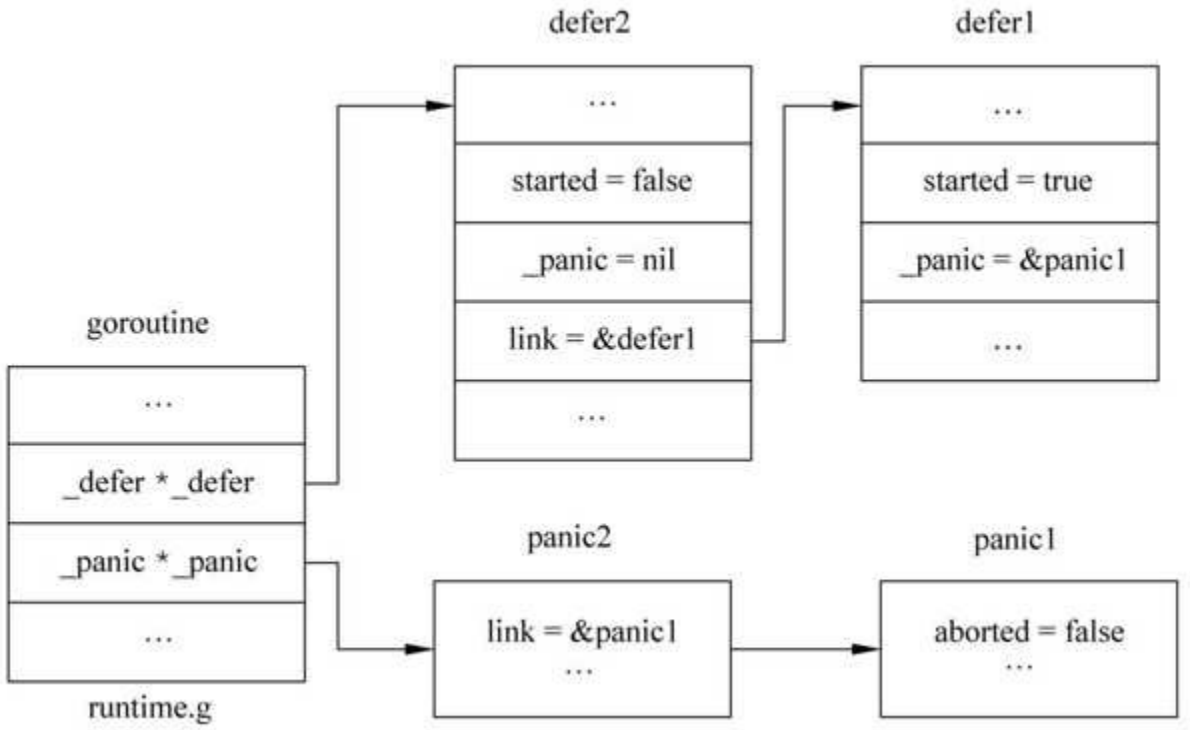

图3-23 defer2执行前的_defer链表和_panic链表

然后panic2去执行_defer链表，首先执行defer2，将其started字段置为true，_panic字段指向panic2。待到defer2执行recover（）函数时，只会把panic2的recovered字段置为true，defer2结束后，从_defer链表中移除，如图3-24所示。

接下来，panic处理逻辑检测到panic2已经被刚刚执行的defer2恢复了，所以会把panic2从_panic链表中移除，如图3-25所示，然后进入recovery（）函数的逻辑中。

结合3.5.1节中的recovery（）函数的介绍，panic2被recover后，当前协程会恢复到defer1中注册完defer2刚刚返回时的状态，只不过返回值被置为1，直接跳转到最后的deferreturn（）函数处，而此时defer链表中已经没有defer1注册的defer函数了，所以defer1结束返回，返回panic1执行defer链表的逻辑中继续执行。

从_panic链表和_defer链表的角度来看，位于_panic链表头部的始终是当前正在执行的panic，如果它在遍历_defer链表的过程中通过_defer结构的started字段和_panic字段发现了上一个panic，就会将

其设为aborted。如果在两次panic之间，_defer链表中加入了新的带有recover的defer函数，则这些defer函数就能够在上一个panic被发现前结束当前panic流程，上一个panic也就不会被aborted，继而恢复执行。

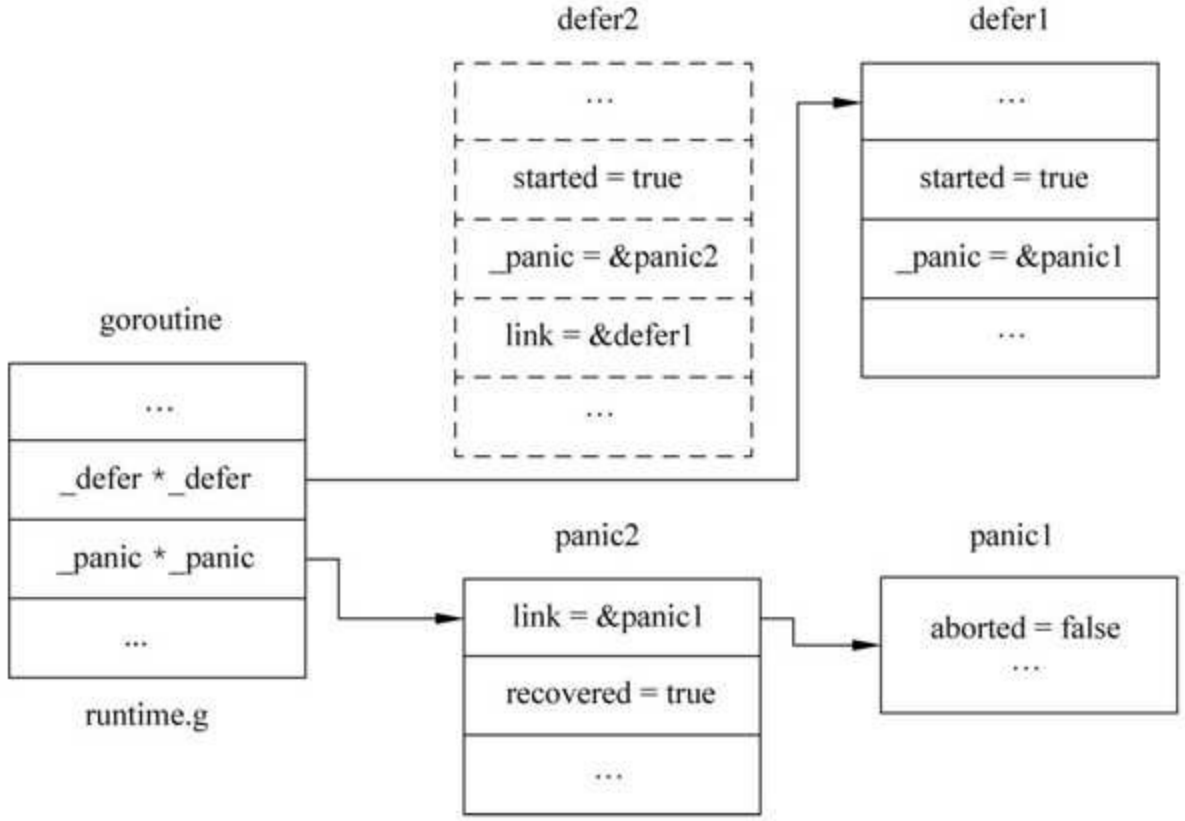

图3-24 defer2结束后的_defer链表和_panic链表

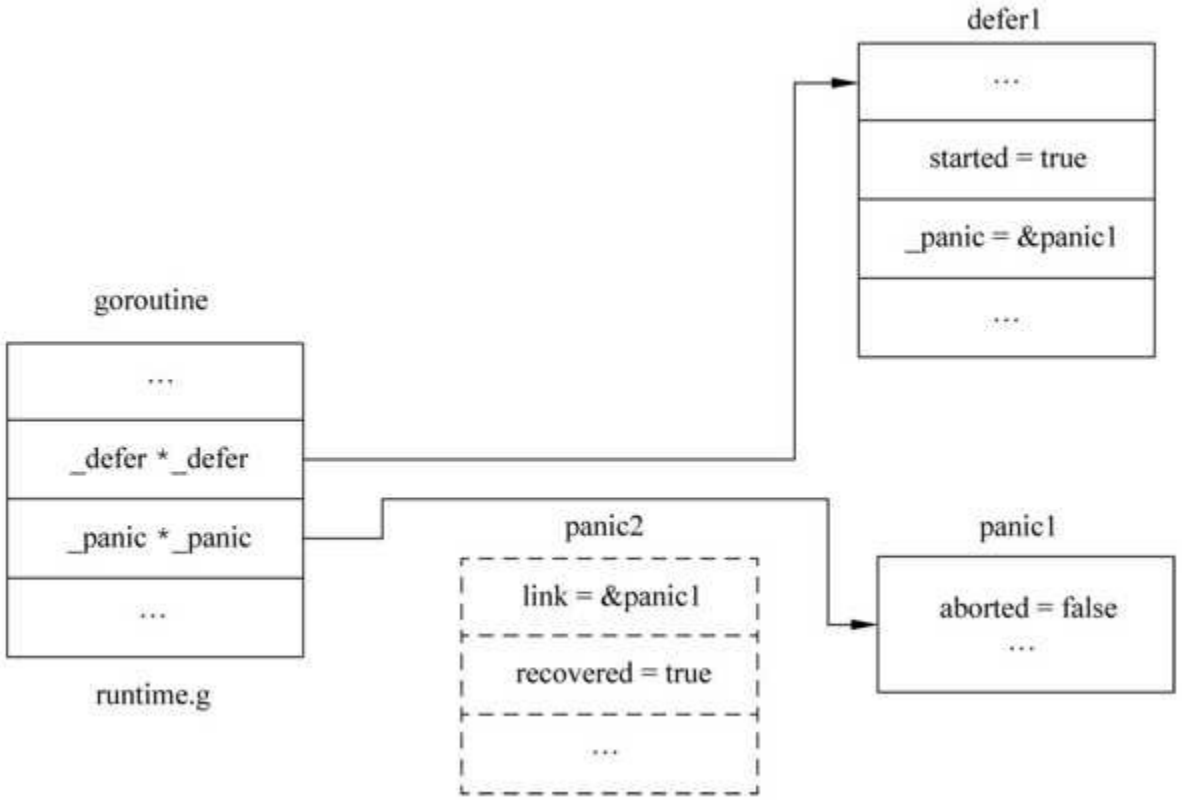

图3-25 panic2恢复后的_defer链表和_panic链表

3.5.4 支持**open coded defer**

3.4.3节讲到open coded defer是以直接调用的方式实现的，并不会被注册到当前goroutine的_defer链表中，那么在发生panic的时候如何找到这些open coded defer函数并执行呢？先来看一下1.14版本中runtime._defer结构的定义，代码如下：

```
type _defer struct {
    siz       int32
    started   bool
    heap      bool
    openDefer bool
    sp        uintptr
    pc        uintptr
    fn        *funcval
    _panic    *_panic
    link      *_defer
    fd        unsafe.Pointer
    varp      uintptr
    framepc   uintptr
}
```

其中的heap字段是在Go 1.13版本中随deferprocStack一起引入的，用来区分_defer结构是堆分配还是栈分配。openDefer、fd、varp和framepc字段都是Go 1.14版本中为了支持open coded defer而引入的，也就是说open coded defer还是可能会被添加到_defer链表中的。什么时候会被添加到链表中呢？就是在panic的时候。

Go 1.14版本中的panic为了支持open coded defer实现了两个重要的函数，即addOne-OpenDeferFrame（）函数和runOpenDeferFrame（）函数。前者从调用栈的栈顶开始做回溯扫描，直到找到一个带有open coded defer的栈帧，为该栈帧分配一个_defer结构，为各字段赋值后添加到_defer链表中合适的位置。不管目标栈帧上有几个open coded defer函数，只分配一个_defer结构，因为后续通过runOpenDeferFrame（）函数来执行的时候，会一并执行栈帧上所有的open coded defer函数。添加到_defer链表中的位置是根据目标栈帧在调用栈中的位置计算的，而不是添加到头部。runOpenDeferFrame（）函数循环执行指定栈帧上所有的open coded defer函数，返回值表示栈帧上所有的open coded defer函数是否都执行完毕，如果因为某个defer函数执行了recover而造成循环中止，则返回值为false。以上两个函数依赖于符号表中目标栈帧的OpenCodedDeferInfo。

gopanic（）函数中几个关键的步骤也都为open coded defer做了相应的修改：

（1）在for循环开始之前，先通过addOneOpenDeferFrame（）函数将最近的一个open coded defer栈帧添加到_defer链表中。

（2）在调用defer函数的时候，如果openDefer为true，则使用runOpenDeferFrame（）函数来执行，通过返回值来判断目标栈帧上的open coded defer已完全执行，并且没有recover，就再次调用addOneOpenDeferFrame（）函数把下一个open coded defer栈帧添加到_defer链表中。

（3）根据runOpenDeferFrame（）函数的返回值来判断，只有完全执行的节点才能从_defer链表中移除。事实上只有openDefer节点才有可能出现不完全执行的情况，因为一个栈帧上可能有多个open coded defer函数，假如其中某一个调用了recover（）函数，后续的就不会再被调用了，所以该节点不能从_defer链表中移除，recover之后的逻辑负责调用这些剩余的open coded defer。

（4）检测到当前panic的recovered为true后，需要把_defer链表中尚未开始执行的openDefer节点移除，因为recover之后这些open coded defer会被正常调用。

那么，包含多个open coded defer函数的栈帧出现不完全执行的情况时，也就是中间的某个defer函数

调用了recover（）函数时，剩余的defer函数是在哪里调用的呢？其实是被deferreturn（）函数调用的。编译器在每个包含open coded defer的函数的最后都会插入一条调用runtime.deferreturn（）函数的指令，这条指令处在一个特殊的分支上，正常流程不会执行到它，而addOneOpenDeferFrame（）函数在为_defer结构的pc字段赋值的时候，使用的就是这条指令的地址，也就是说当某个open coded defer调用recover之后，指令指针会恢复到这条指令处，进而调用runtime.deferreturn（）函数。1.14版的deferreturn（）函数中对于openDefer为true的节点会使用runOpenDeferFrame（）函数来处理，从而使栈帧上剩余的open coded defer得到执行。也不用担心重复调用问题，因为runOpenDeferFrame（）函数会把已经调用过的defer函数的相应标志位清0。

3.6 本章小结

在Go语言中，函数是非常基础也是非常重要的一个特性。3.1节探索了函数的栈帧布局及栈帧上的内存对齐，认识到返回值、参数和局部变量就像是3个struct。3.2节探索了编译器是如何判断变量是否逃逸的，了解了编译器总是会尽量尝试在栈上分配局部变量。3.3节通过反汇编和使用函数钩子等方法，分析了Function Value的实现原理，理解了函数指针和闭包在实现层面的统一。3.4节介绍了defer在最近几个版本中的演变，以及最新的open coded defer。3.5节梳理了panic和recover的实现逻辑。

本章的内容比较重要，希望各位读者能够结合实践深入理解，以便后续能更加高效地学习和探索。

第4章
方法

Go语言支持面向对象思想，提供了type关键字，可以用来自定义类型，并且可以为自定义类型实现方法。下面定义一个Point类型，代码如下：

```
//第 4 章/code_4_1.go
package gom

type Point struct {
    x float64
}

func (p Point) X() float64 {
    return p.x
}

func (p * Point) SetX(x float64) {
    p.x = x
}
```

Point表示一维坐标系内的一个点，并且按照Go语言的风格为其实现了一个Getter方法和一个Setter方法。本章后续内容将会以Point类型为研究对象，展开与方法相关的问题的探索。

从语法角度来看，Go语言的方法并不像C++、Java中class的方法那样包含在type定义的语句块内，而是像普通函数一样直接定义在package层，只不过多了一个接收者。以Point类型的两种方法为例，处在func关键字和方法名之间的，就是方法的接收者，它看起来就像一个额外的参数。

4.1 接收者类型

在第3章中已经探索过普通函数的调用约定，了解了参数和返回值是通过栈传递的，这里就会比较好奇方法接收者的传递方式：到底是像一般参数那样通过栈传递，还是像C++的thiscall那样使用某个指定的寄存器？

通过反编译很容易验证。为了排除编译器内联优化造成的干扰，下面采用只编译不链接的方式来得到OBJ文件，然后对编译得到的OBJ文件进行反编译分析，编译命令如下：

```
$ go tool compile -trimpath="`pwd`=>" -l -p gom point.go
```

上述命令禁用了内联优化，编译完成后会在当前工作目录生成一个point.o文件，这就是我们想要的OBJ文件。通过go tool nm可以查看该文件中实现了哪些函数，nm会输出OBJ文件中定义或使用到的符号信息，通过grep命令过滤代码段符号对应的T标识，即可查看文件中实现的函数，执行命令如下：

```
$ go tool nm point.o | grep T
   1562 T gom.(*Point).SetX
   1899 T gom.(*Point).X
   1555 T gom.Point.X
```

可以看到point.o中一共实现了3个方法，它们都定义在Point类型所在的gom包中。第1个是Point的SetX（）方法，它的接收者类型是*Point，第3个是Point的X（）方法，它的接收者类型是Point，这些都与源代码一致。比较奇怪的是第二个方法，这是一个接收者类型为*Point的X（）方法，源代码中并没有这个方法，它是怎么来的呢？只能是由编译器生成的。那么编译器为什么要生成它呢？这就需要循序渐进地进行探索了。

为了方便描述，我们将接收者类型为值类型的方法称为值接收者方法，将接收者类型为指针类型的方法称为指针接收者方法。接下来先通过反编译的方式看一下，这两种接收者参数都是如何传递的。

4.1.1 值类型

先来看一下Point类型中的值接收者方法X（），反编译后得到的汇编代码如下：

```
$ go tool objdump -S -s '^gom.Point.X$' point.o
TEXT gom.Point.X(SB) gofile..point.go
        return p.x
  0x1555        f20f10442408        MOVSD_XMM 0x8(SP), X0
  0x155b        f20f11442410        MOVSD_XMM X0, 0x10(SP)
  0x1561        c3                  RET
```

因为函数过于简单，对栈空间也没有太大消耗，所以编译器没有插入与栈增长相关的代码，也没有通过SUB指令移动SP来为方法X（）分配栈帧，所以SP指向的是CALL指令压入栈中的返回地址。

第4行代码用SP作为基址并加上8字节偏移，把该地址处的一个float64复制到X0寄存器中。

第5行代码用SP作为基址并加上16字节偏移，把X0中的float64复制到该地址处。

第6行代码就是普通的返回指令。

按照上述汇编代码逻辑，栈上的布局如图4-1所示。

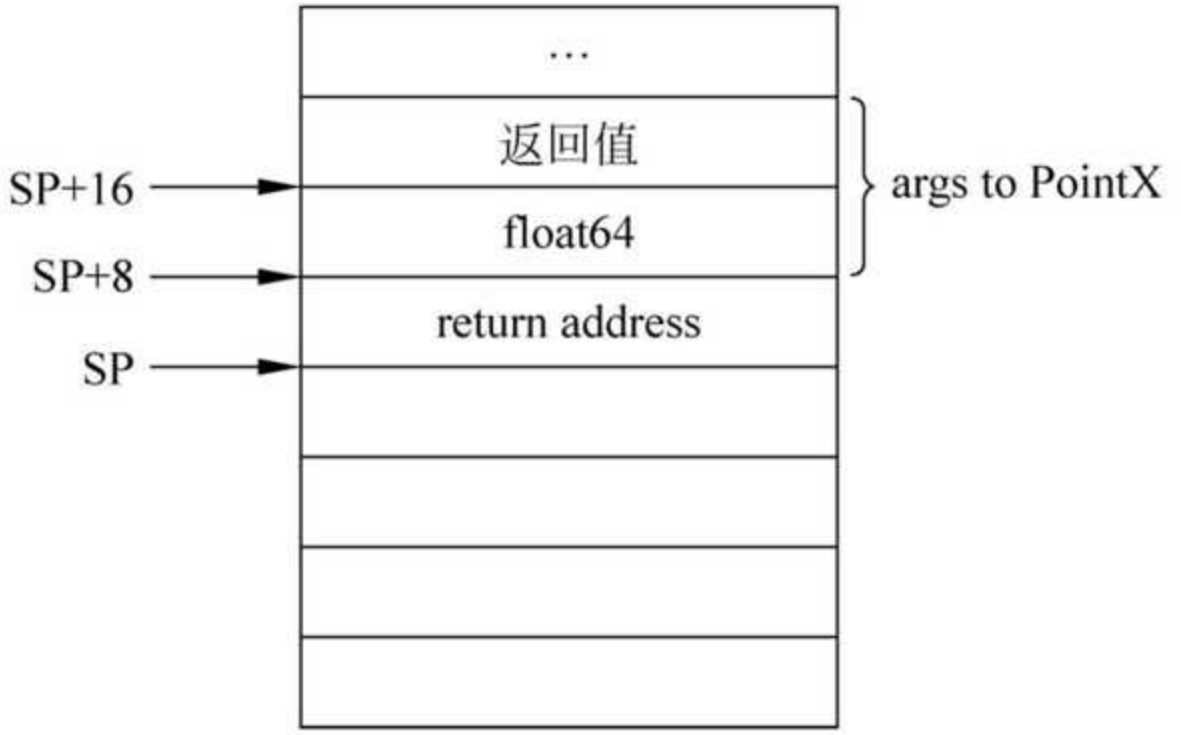

图4-1 调用Point.X（）方法后的栈帧布局

结合方法X（）的源代码，栈指针SP加16字节偏移处，应该就是函数的返回值。SP加8字节偏移处，应该就是函数第1个参数的位置。从代码逻辑来看，这个参数存储的就是p.x的值，而Point类型只有x这一个成员，所以第1个参数是p的值。这就说明值类型的接收者实际上是作为第1个参数通过栈来传递的，与普通的函数调用并没有什么不同。

Go语言允许通过方法的完全限定名称（Full Qualified Name）把方法当成一个普通函数那样调用，只不过需要把接收者作为第1个参数显式地传递，示例代码如下：

```
p := Point{x: 10}
Point.X(p)
```

可以认为p.X（）这种写法只是编译器提供的语法糖，本质上会被转换为Point.X（p）这种普通的函数调用，而接收者就是隐含的第1个参数。

4.1.2 指针类型

4.1.1节分析了值接收者参数的传递方式，本节再来看一下指针类型接收者的参数传递方式。还是通过反编译的方式，这次要反编译SetX（）方法，反编译后得到的汇编代码如下：

```
$ go tool objdump -S -s '^gom.\(\*Point\).SetX$' point.o
TEXT gom.(*Point).SetX(SB) gofile..point.go
        p.x = x
  0x1562        f20f10442410        MOVSD_XMM 0x10(SP), X0
  0x1568        488b442408          MOVQ 0x8(SP), AX
  0x156d        f20f1100            MOVSD_XMM X0, 0(AX)
}
  0x1571        c3                  RET
```

跟之前一样，因为函数很简单，所以既没有插入与栈增长相关的代码，也没有移动SP来分配栈

帧，SP指向栈上的返回地址。

第4行代码用SP作为基址加上16字节偏移，把该地址处的一个float64复制到X0寄存器中。

第5行代码用SP作为基址加上8字节偏移，把该地址处的一个64位数值复制到AX寄存器中。

第6行代码用AX作为基址，把X0寄存器中的float64复制到该地址处。

第8行是返回指令。

按照上述汇编代码逻辑，画出栈上的布局如图4-2所示。

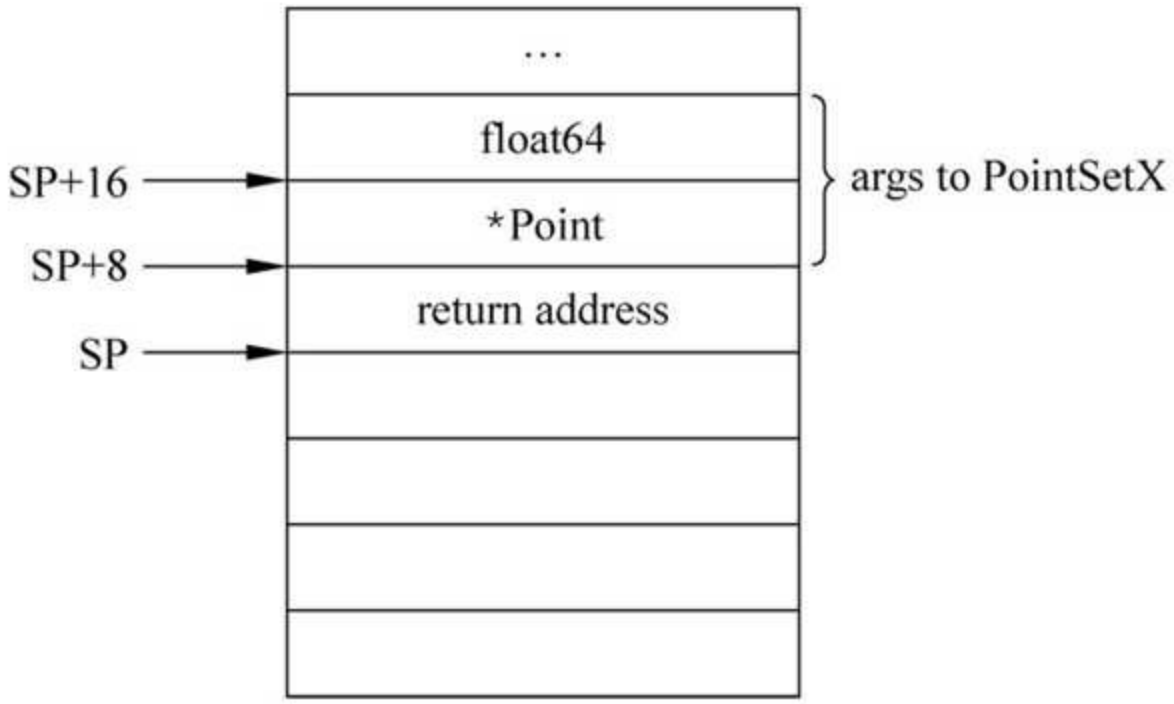

图4-2 调用Point.SetX（）方法后的栈帧布局

结合SetX（）方法的源码可以推断出，栈指针SP加16字节偏移处存储的浮点型数值，就是SetX（）方法的参数x。SP加8字节偏移处存储的64位数值就是接收者p的地址，所以跟值类型接收者类似，指针类型的接收者也是作为第1个参数来传递的，只不过传递的是一个地址。

同值接收者方法一样，也可以通过完全限定名称把指针接收者方法作为一个普通函数那样调用，只是语法上稍有不同，代码如下：

```
p := &Point{}
(*Point).SetX(p, 10)
```

4.1.3 包装方法

本节开头通过nm查看OBJ文件中的符号的时候，发现OBJ文件中多了一个源码中没有的方法。源码中X（）方法的接收者是值类型，而OBJ文件中多了一个拥有指针类型接收者的X（）方法。猜测这种方法应该是编译器根据源代码中原本的X（）方法自动生成的包装方法，通过接收者的地址可以得到接收者的值，所以反编译来看一下代码逻辑，反编译得到的汇编代码如下：

```
$ go tool objdump -S -s '^gom.\(\*Point\).X$' point.o
TEXT gom.(*Point).X(SB) gofile..<autogenerated>

  0x1b8e       65488b0c2528000000      MOVQ GS:0x28, CX
  0x1b97       488b8900000000          MOVQ 0(CX), CX     [3:7]R_TLS_LE
  0x1b9e       483b6110                CMPQ 0x10(CX), SP
  0x1ba2       7650                    JBE 0x1bf4
  0x1ba4       4883ec18                SUBQ $0x18, SP
  0x1ba8       48896c2410              MOVQ BP, 0x10(SP)
  0x1bad       488d6c2410              LEAQ 0x10(SP), BP
  0x1bb2       488b5920                MOVQ 0x20(CX), BX
  0x1bb6       4885db                  TESTQ BX, BX
  0x1bb9       7540                    JNE 0x1bfb
  0x1bbb       488b442420              MOVQ 0x20(SP), AX
  0x1bc0       4885c0                  TESTQ AX, AX
  0x1bc3       7429                    JE 0x1bee
  0x1bc5       f20f1000                MOVSD_XMM 0(AX), X0
  0x1bc9       f20f110424              MOVSD_XMM X0, 0(SP)
  0x1bce       e800000000              CALL 0x1bd3        [1:5]R_CALL:gom.Point.X
  0x1bd3       f20f10442408            MOVSD_XMM 0x8(SP), X0
  0x1bd9       f20f11442428            MOVSD_XMM X0, 0x28(SP)
  0x1bdf       488b6c2410              MOVQ 0x10(SP), BP
  0x1be4       4883c418                ADDQ $0x18, SP
  0x1be8       c3                      RET
  0x1be9       0f1f440000              NOPL 0(AX)(AX*1)
  0x1bee       e800000000              CALL 0x1bf3        [1:5]R_CALL:runtime.panicwrap
  0x1bf3       90                      NOPL
  0x1bf4       e800000000              CALL 0x1bf9        [1:5]R_CALL:runtime.morestack_noctxt
  0x1bf9       eb93                    JMP gom.(*Point).X(SB)
  0x1bfb       488d7c2420              LEAQ 0x20(SP), DI
  0x1c00       48393b                  CMPQ DI, 0(BX)
  0x1c03       75b6                    JNE 0x1bbb
  0x1c05       488923                  MOVQ SP, 0(BX)
  0x1c08       ebb1                    JMP 0x1bbb
```

通过gofile对应的autogenerated可以确定该方法确实是由编译器自动生成的。反编译得到的汇编代码还是有些复杂，为了便于理解，在保证逻辑一致的前提下，转换后的伪代码如下：

```
func (p * Point) X() float64 {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    if gp._panic != nil {
        if unsafe.Pointer(&p) == gp._panic.argp {
            gp._panic.argp = unsafe.Pointer(getargp(0))
        }
    }
    if p == nil {
        runtime.panicwrap()
    }
    return Point.X(*p)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

第1个if语句块通过比较栈指针SP和gp.stackguard0来判断是否需要栈增长。

第2个if用于检测是否正处于panic流程中，这种情况下当前方法应该是被某个defer直接或间接地调用了，要按需修改gp._panic.argp的值，因为当前方法是编译器自动包装的，通过修改argp来跳过包装方法的栈帧，使后面调用的原始方法中的recover能够生效。

第3个if用于检测p是否为nil，因为包装方法需要根据p的地址得到*p的值，如果地址为nil就调用runtime.panicwrap。

最后一步才是调用原始的Point.X（）方法并传递*p的值作为参数。

如果不禁用内联优化，则生成的代码会稍微有些不同，但是大致逻辑还是一样的。编译器会为代码中定义的值接收者方法生成指针接收者的包装方法，这在语义上是可行的，但反过来却不可以，因为通过传递的值是无法得到原始变量的地址的。

虽然知道了包装方法的大致逻辑，但还是没有搞清楚编译器生成包装方法的原因。如果是为了支持通过指针直接调用值接收者方法，则直接在调用端进行指针解引用就可以了，总不至于为此生成包装方法吧？为了验证这个问题，再次准备一个函数用来反编译，函数的代码如下：

```
func PointX(p *Point) float64 {
    return p.X()
}
```

大致思路就是通过指针来调用值接收者方法，再通过反编译看一下实际调用的是不是包装方法。反编译得到的汇编代码如下：

```
$ go tool objdump -S -s '^gom.PointX$' point.o
TEXT gom.PointX(SB) gofile..point.go
func PointX(p *Point) float64 {
  0x1a17      65488b0c2528000000      MOVQ GS:0x28, CX
  0x1a20      488b8900000000          MOVQ 0(CX), CX     [3:7]R_TLS_LE
  0x1a27      483b6110                CMPQ 0x10(CX), SP
  0x1a2b      7637                    JBE 0x1a64
  0x1a2d      4883ec18                SUBQ $0x18, SP
  0x1a31      48896c2410              MOVQ BP, 0x10(SP)
  0x1a36      488d6c2410              LEAQ 0x10(SP), BP
        return p.X()
  0x1a3b      488b442420              MOVQ 0x20(SP), AX
  0x1a40      f20f1000                MOVSD_XMM 0(AX), X0
  0x1a44      f20f110424              MOVSD_XMM X0, 0(SP)
  0x1a49      e800000000              CALL 0x1a4e     [1:5]R_CALL:gom.Point.X
  0x1a4e      f20f10442408            MOVSD_XMM 0x8(SP), X0
  0x1a54      f20f11442428            MOVSD_XMM X0, 0x28(SP)
  0x1a5a      488b6c2410              MOVQ 0x10(SP), BP
  0x1a5f      4883c418                ADDQ $0x18, SP
  0x1a63      c3                      RET
func PointX(p *Point) float64 {
  0x1a64      e800000000              CALL 0x1a69     [1:5]R_CALL:runtime.morestack_noctxt
  0x1a69      ebac                    JMP gom.PointX(SB)
```

可以看到p.X（）实际上会在调用端对指针解引用，然后调用值接收者方法，并没有调用编译器生成的包装方法。那这个包装方法有什么用途呢？现在还不能解释，到第5章介绍接口时再来回答这个问题。

4.2 Method Value

第3章中探索了Function Value底层的数据结构，实质上可能是个两级指针，也可能是个闭包对象，要结合具体的上下文才能确定。简单来讲，把一个函数存储在一个变量中，这个变量就是一个Function Value。相应地，把一个方法存储在一个变量中，这个变量就是个Method Value。那么Method Value又有着怎样的底层实现呢？与Function Value有什么异同？本节就围绕这些问题展开探索。

4.2.1 基于类型

可以通过方法的完全限定名称把自定义类型的某个方法赋值给一个变量，这样就会得到一个基于类型的Method Value，也就是所谓的Method Expression。还是以Point类型为例，定义一个基于类型的Method Value，示例代码如下：

```
x := Point.X
```

4.1节已经验证了方法其实就是个普通的函数，接收者是隐含的第1个参数，所以这里可以推断，基于类型的Method Value就是个普通的Function Value，本质上是个两级指针，而且第二级的指针是在编译阶段静态分配的。

通过示例代码很容易验证上述推断，代码如下：

```
func GetX() func(Point) float64 {
    return Point.X
}
```

上述代码可以成功编译，说明Point.X（）函数可以被赋值给func（Point）float64类型的Function Value。接下来反编译GetX（）函数，得到的汇编代码如下：

```
$ go tool objdump -S -s 'gom.GetX' point.o
TEXT gom.GetX(SB) gofile..point.go
        return Point.X
  0x17b4        488d0500000000        LEAQ 0(IP), AX        [3:7]R_PCREL:gom.Point.X·f
  0x17bb        4889442408            MOVQ AX, 0x8(SP)
  0x17c0        c3                    RET
```

第4行代码用IP作为基址加上一个偏移0来得到一个地址，这个0只作为预留的一个32位整数，等到链接阶段，链接器会填写上实际的偏移值。第4行代码得到的地址被用作返回值，也就是最终的Function Value，而该地址处就是第二级指针，从而验证了上述推断。

4.2.2 基于对象

可以把一个对象的某个方法赋值给一个变量，这样就会得到一个基于对象的Method Value，示例代码如下：

```
p := Point{x: 10}
x := p.X
```

从语义角度来看，与基于类型的Method Value不同，基于对象的Method Value隐式地包含了对象的数据，所以在上述代码中调用x时不需要再显式地传递接收者参数。第3章中已经了解了闭包的实现原理，所以这里推断x是个指向闭包对象的指针，通过闭包的捕获列表捕获了对象p。

为了验证这种推断，实现一个示例函数，代码如下：

```
func X(p Point) func() float64 {
    return p.X
}
```

反编译上面的函数，得到的汇编代码如下：

```
$ go tool objdump -S -s '^gom.X$' point.o
TEXT gom.X(SB) gofile..point.go
func X(p Point) func() float64 {
  0x213c      65488b0c2528000000    MOVQ GS:0x28, CX
  0x2145      488b8900000000        MOVQ 0(CX), CX     [3:7]R_TLS_LE
  0x214c      483b6110              CMPQ 0x10(CX), SP
  0x2150      764a                  JBE 0x219c
  0x2152      4883ec18              SUBQ $0x18, SP
  0x2156      48896c2410            MOVQ BP, 0x10(SP)
  0x215b      488d6c2410            LEAQ 0x10(SP), BP
        return p.X
  0x2160      488d0500000000        LEAQ 0(IP), AX
[3:7]R_PCREL:type.noalg.struct { F uintptr; R gom.Point }
  0x2167      48890424              MOVQ AX, 0(SP)
  0x216b      e800000000            CALL 0x2170       [1:5]R_CALL:runtime.newobject
  0x2170      488b442408            MOVQ 0x8(SP), AX
  0x2175      488d0d00000000        LEAQ 0(IP), CX     [3:7]R_PCREL:gom.Point.X-fm
  0x217c      488908                MOVQ CX, 0(AX)
  0x217f      f20f10442420          MOVSD_XMM 0x20(SP), X0
  0x2185      f20f114008            MOVSD_XMM X0, 0x8(AX)
  0x218a      4889442428            MOVQ AX, 0x28(SP)
  0x218f      488b6c2410            MOVQ 0x10(SP), BP
  0x2194      4883c418              ADDQ $0x18, SP
  0x2198      c3                    RET
func X(p Point) func() float64 {
  0x2199      0f1f00                NOPL 0(AX)
  0x219c      e800000000            CALL 0x21a1     [1:5]R_CALL:runtime.morestack_noctxt
  0x21a1      eb99                  JMP gom.X(SB)
```

为了便于理解，改写成逻辑等价的伪代码如下：

```
func X(p Point) func() float64 {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    o := new(struct { F uintptr; R gom.Point })
    o.F = gom.Point.X-fm
    o.R = p
    return o
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

编译器为返回值自动定义了一个struct，第1个成员是一个函数指针，第2个成员是一个Point对象。对应到闭包对象的结构，捕获列表中是Point类型的对象，闭包函数是gom.Point.X-fm（）函数，也是由编译器自动生成的。下面反编译一下这个闭包函数，得到的汇编代码如下：

```
$ go tool objdump -S -s '^gom.Point.X-fm$' point.o
TEXT gom.Point.X-fm(SB) gofile..point.go
func (p Point) X() float64 {
  0x2b1b      65488b0c2528000000      MOVQ GS:0x28, CX
  0x2b24      488b8900000000          MOVQ 0(CX), CX     [3:7]R_TLS_LE
  0x2b2b      483b6110                CMPQ 0x10(CX), SP
  0x2b2f      7633                    JBE 0x2b64
  0x2b31      4883ec18                SUBQ $0x18, SP
  0x2b35      48896c2410              MOVQ BP, 0x10(SP)
  0x2b3a      488d6c2410              LEAQ 0x10(SP), BP
  0x2b3f      f20f104208              MOVSD_XMM 0x8(DX), X0
  0x2b44      f20f110424              MOVSD_XMM X0, 0(SP)
  0x2b49      e800000000              CALL 0x2b4e     [1:5]R_CALL:gom.Point.X
  0x2b4e      f20f10442408            MOVSD_XMM 0x8(SP), X0
  0x2b54      f20f11442420            MOVSD_XMM X0, 0x20(SP)
  0x2b5a      488b6c2410              MOVQ 0x10(SP), BP
  0x2b5f      4883c418                ADDQ $0x18, SP
  0x2b63      c3                      RET
  0x2b64      e800000000              CALL 0x2b69     [1:5]R_CALL:runtime.morestack
  0x2b69      ebb0                    JMP gom.Point.X-fm(SB)
```

等价的伪代码如下：

```
func Point.X-fm() float64 {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    p := (*struct { F uintptr; R gom.Point })(unsafe.Pointer(DX))
    return Point.X(p.R)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

主要逻辑就是通过DX寄存器得到闭包对象的地址，再以捕获列表里的Point对象的值作为参数调用Point.X（）方法，并把Point.X（）方法的返回值作为自己的返回值。

进一步探索会发现，闭包是捕获对象的值还是捕获地址，跟Method Value对应的方法接收者类型一致。上述示例中Point.X（）方法的接收者为值类型，所以闭包捕获的也是值类型，如果换成接收者为指针类型的*Point.SetX（）方法，闭包捕获列表中就会相应地变成指针类型。

至此可以进行一下总结，基于类型的Method Value和基于对象的Method Value本质上都是Function Value，只不过前者是简单的两级指针，而后者通常是个闭包（考虑编译器优化）。

4.3 组合式继承

Go语言中提供了一种组合式的继承方式，在语法和思想上都与C++、Java等语言中的继承有些不同。本节要探索一下编译器是如何支持这种继承方式的。

继续使用Point类型，定义一个Point2d类型来表示二维坐标系内的一个点，采用组合式继承的方式继承Point类型，代码如下：

```
//第 4 章/code_4_2.go
type Point2d struct {
    Point
    y float64
}

func (p Point2d) Y() float64 {
    return p.y
}

func (p *Point2d) SetY(y float64) {
    p.y = y
}
```

接下来的探索将用到这两个类，为了叙述方便，后续内容将继续采用传统的面向对象术语，把Point称为基类，而Point2d就是Point的子类。

4.3.1 嵌入值

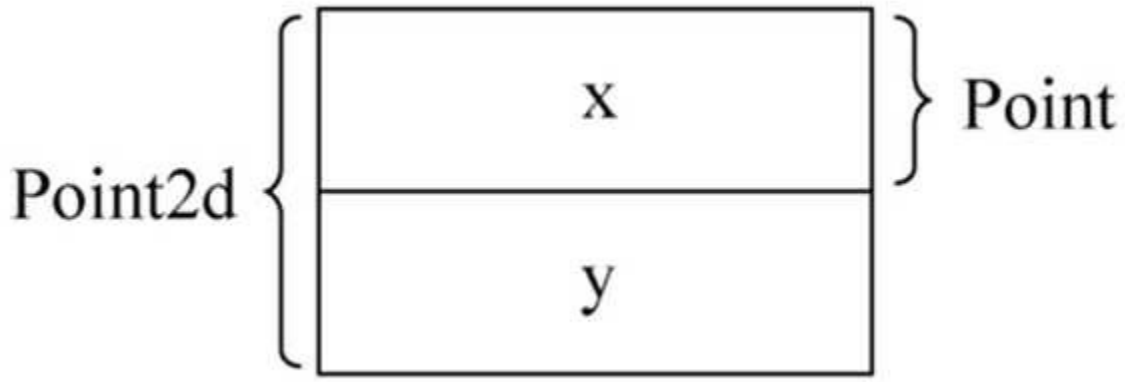

图4-3 Point2d内存布局示意图

在Point2d的类型定义中，Point类型以嵌入值的形式嵌入Point2d中，Point就是Point2d的一个字段，Point2d类型的内存布局如图4-3所示。

组合式继承也是继承，所以Point2d应该会继承Point的所有方法，可以再次用nm命令查看一下OBJ文件中为Point2d类型实现了哪些函数和方法，命令如下：

```
$ go tool nm point.o | grep 'T ' | grep Point2d
    5896 T gom.(*Point2d).SetX
    47d2 T gom.(*Point2d).SetY
    58a7 T gom.(*Point2d).X
    591d T gom.(*Point2d).Y
    599f T gom.Point2d.X
    47c5 T gom.Point2d.Y
    585d T type..eq.gom.Point2d
```

最后一个函数是由编译器自动生成的，用于判断两个Point2d对象是否相等，现阶段不用关心这个函数。剩下的就是Point2d类型的6个方法，其中有3个和X相关，另外3个和Y相关。和Y相关的这3个方法没有什么特殊的，即Point2d类型的方法。和X相关的这3个方法，应该就是从Point类型继承过来的，接下来逐个看一下这3个方法的逻辑。

首先反编译一下Point2d.X（）方法，为了节省篇幅，这里不再列出汇编代码，还是用笔者根据汇编代码整理的等价伪代码来代替，代码如下：

```
func (p Point2d) X() float64 {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    if gp._panic != nil {
        if unsafe.Pointer(&p) == gp._panic.argp {
            gp._panic.argp = unsafe.Pointer(getargp(0))
        }
    }
    return Point.X(p.Point)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

忽略其中编译器插入的栈增长和按需修改gp._panic.argp的代码，这样就只剩下以p.Point为参数来调用Point.X（）方法的代码，也就说明这是个包装方法，因此可以推测，编译器对于继承来的方法都是通过生成相应的包装方法来调用原始方法的方式实现的。接下来就通过分析Point2d继承的其他两个方法来验证。

反编译（*Point2d）.SetX（）方法得到的汇编代码如下：

```
$ go tool objdump -S -s '^gom.\(\*Point2d\).SetX$' point.o
TEXT gom.(*Point2d).SetX(SB) gofile..<autogenerated>

  0x7d27        488b442408        MOVQ 0x8(SP), AX             //第1条指令
  0x7d2c        8400              TESTB AL, 0(AX)              //第2条指令
  0x7d2e        4889442408        MOVQ AX, 0x8(SP)             //第3条指令
  0x7d33        e900000000        JMP gom.(*Point).SetX(SB)    //第4条指令
[1:5]R_CALL:gom.(*Point).SetX
```

编译器没有为这个SetX（）方法生成复杂的包装逻辑，只是实现了一个空指针校验和跳转指令。

第1条指令把接收者的值复制到AX寄存器中。

第2条指令尝试访问AX存储的地址处的数据，如果接收者为空指针就会触发空指针异常。

第3条指令把AX的值复制到栈上接收者参数的位置，这一行其实可以优化掉。

第4条指令用于跳转到（*Point）.SetX（）方法的起始地址。

为什么可以直接跳转呢？从传参的角度来看就比较好理解了。Point是Point2d的第1个字段，所以Point2d的地址也就等于内嵌的Point的地址，所以可以认为（*Point2d）.SetX（）方法和（*Point）.SetX（）方法的参数和返回值无论是内存布局还是逻辑含义都一样，所以直接跳转是没有问题的。可以认为这是编译器对指针接收者包装方法进行了优化，对于值接收者包装方法则不会进行这种优化，因为子类一般会对基类进行扩展，在作为值传递的时候内存布局无法保证一致。

最后反编译一下（*Point2d）.X（）方法，对照汇编整理出的伪代码如下：

```
func (p * Point2d) X() float64 {
entry:
    gp : = getg()
    if SP < = gp.stackguard0 {
        goto morestack
    }
    if gp._panic != nil {
        if unsafe.Pointer(&p) == gp._panic.argp {
            gp._panic.argp = unsafe.Pointer(getargp(0))
        }
    }
    return Point.X(p.Point)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

可以看到除了接收者为指针类型外，代码逻辑与Point2d.X（）方法基本一致，所以嵌入值实现的组合式继承并没有什么特别的地方，编译器会为继承的方法生成包装方法。实际上，Point和Point2d的这些方法都很简单，正常情况下都会被编译器内联优化掉。这里先记住编译器是如何生成这些包装方法的，在后续的章节中会逐渐发现它们的真正用途。

4.3.2 嵌入指针

将Point2d类型定义修改为嵌入*Point类型，即可实现嵌入指针的组合式继承，代码如下：

```
//第 4 章/code_4_3.go
type Point2d struct {
    * Point
    y float64
}
```

相应地，Point2d中不再直接包含Point的值，而是包含Point对象的地址。两者在内存中的布局关系也变得与之前不同，如图4-4所示。

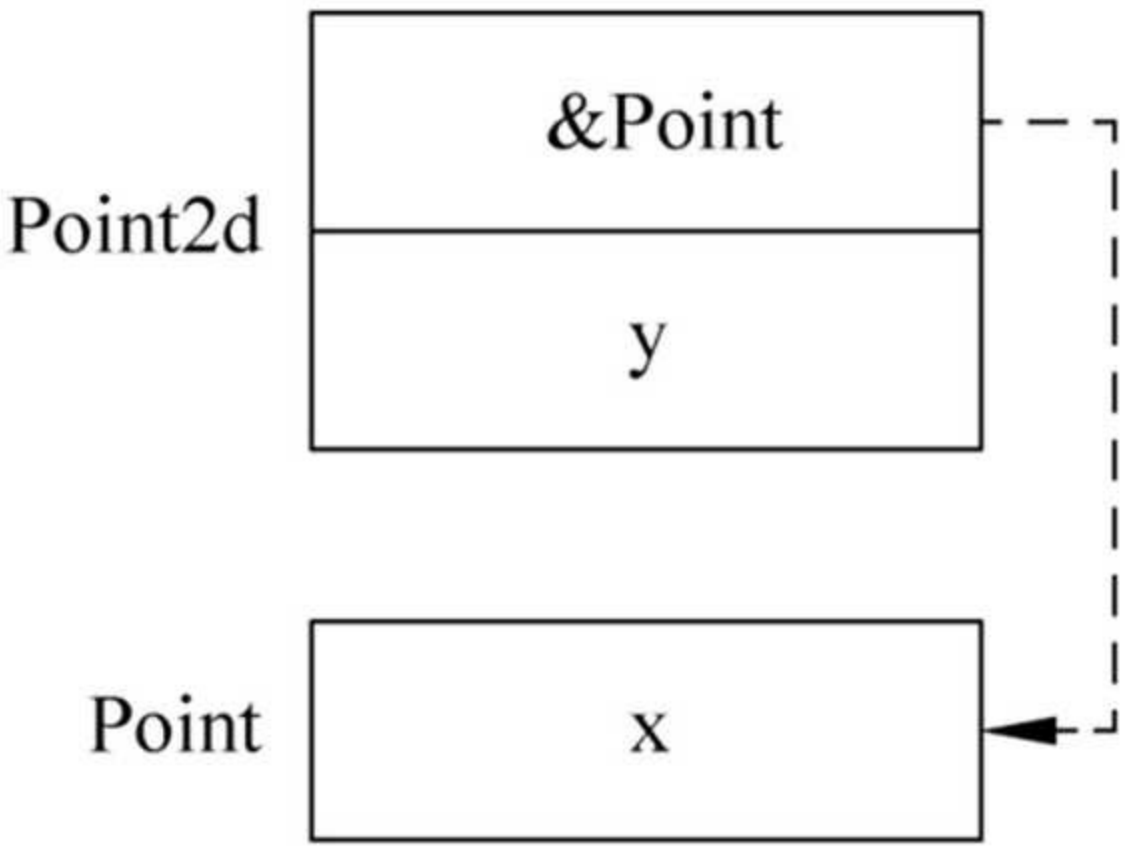

图4-4 Point2d与Point的内存布局关系

再用nm命令查看一下OBJ文件中为Point2d类型实现了哪些函数和方法，命令如下：

```
$ go tool nm point.o | grep 'T ' | grep Point2d
    94e2 T gom.(*Point2d).SetX
    77f0 T gom.(*Point2d).SetY
    94f4 T gom.(*Point2d).X
    956a T gom.(*Point2d).Y
    95ec T gom.Point2d.SetX
    9656 T gom.Point2d.X
    77e3 T gom.Point2d.Y
    94b1 T type..eq.gom.Point2d
```

这里值得注意的是Point2d.SetX（）方法，它的存在意味着虽然接收者Point2d是通过值的形式传递的，但是通过Point2d的值可以得到原始Point对象的地址，所以依然可以对原始Point对象进行修改。

至于其他几个继承的方法，这里就不再一一进行反编译了，只是看一下在嵌入指针的情况下（*Point2d）.SetX（）方法还会不会被优化处理，代码如下：

```
$ go tool objdump -S -s '^gom.\(\*Point2d\).SetX$' point.o
TEXT gom.(*Point2d).SetX(SB) gofile..<autogenerated>
  0x94e2      488b442408    MOVQ 0x8(SP), AX            //第1行指令
  0x94e7      488b00        MOVQ 0(AX), AX              //第2行指令
  0x94ea      4889442408    MOVQ AX, 0x8(SP)            //第3行指令
  0x94ef      e900000000    JMP gom.(*Point).SetX(SB)   //第4行指令
[1:5]R_CALL:gom.(*Point).SetX
```

编译器还进行了优化处理，第1行指令把栈上的接收者参数复制到AX寄存器中，其实也就是Point2d对象的地址。第2行指令把Point2d的第1个字段的值复制到AX寄存器中，也就是Point对象的地址。第3行指令把AX的值复制回栈上的接收者参数处。第4行指令用于跳转到（*Point）.SetX（）方法的起始地址。

至于其他3种方法，编译器都会生成相应的包装方法，这里不再赘述，直接列出对应的伪代码。为了使代码更加简洁，这里省略了栈增长和处理gp._panic.argp的相关逻辑，精简后的代码如下：

```
func (p Point2d) X() float64 {
    return Point.X(*p.Point)
}

func (p *Point2d) X() float64 {
    return Point.X(*(*p).Point)
}

func (p Point2d) SetX() {
    (*Point).SetX(p.Point)
}
```

通过本节的探索可以发现，因为在嵌入指针的情况下总是能够得到基类对象的地址，所以子类中的值接收者方法可以调用基类中的指针接收者方法，编译器会尽可能把符合逻辑的包装方法都生成出来。

4.3.3 多重继承

组合式继承之下的多重继承，实际上就是在子类的定义中嵌入多个基类。嵌入的多个基类可以按需嵌入值或嵌入指针，内存布局方面前两节已经分别给出相应的图示，这里不再赘述。本节主要探索一下多重继承对编译器生成包装方法会有哪些影响。

首先定义两种类型A和B，分别为它们实现一组相同的方法Value（）和Set（），代码如下：

```
//第 4 章/code_4_4.go
package gom
type A struct {
    a int
}

type B struct {
    b int
}

func (a A) Value() int {
    return a.a
}

func (a *A) Set(v int) {
    a.a = v
}

func (b B) Value() int {
    return b.b
}

func (b *B) Set(v int) {
    b.b = v
}
```

然后定义一种类型C，将A和B以值的形式嵌入，代码如下：

```
//第 4 章/code_4_5.go
type C struct {
    A
    B
}
```

通过nm命令查看编译生成的OBJ文件中都实现了哪些方法，命令如下：

```
$ go tool nm multi.o | grep 'T '
    24a6 T gom.(*A).Set
    2be3 T gom.(*A).Value
    24bf T gom.(*B).Set
    2c59 T gom.(*B).Value
    249b T gom.A.Value
    24b4 T gom.B.Value
```

发现只有A和B的方法，编译器没有为C生成任何方法。结合Go语言官方文档的说明，因为同时嵌入A和B而且嵌套的层次相同，所以编译器不知道应该让包装方法继承自谁，这种情况只能由程序员手工实现。

下面再来看一下嵌套层次不同的情况。定义一种类型D，把A以嵌入值的形式嵌入D中，然后把C中的A改成D，代码如下：

```
//第 4 章/code_4_6.go
type C struct {
    D
    B
}

type D struct {
    A
}
```

再次通过nm命令查看，命令如下：

```
$ go tool nm multi.o | grep 'T '
    3a7c T gom.(*A).Set
    4603 T gom.(*A).Value
    3a95 T gom.(*B).Set
    4679 T gom.(*B).Value
    47d4 T gom.(*C).Set
    47e9 T gom.(*C).Value
    46ef T gom.(*D).Set
    4700 T gom.(*D).Value
    3a71 T gom.A.Value
    3a8a T gom.B.Value
    4853 T gom.C.Value
    476a T gom.D.Value
```

这次类型C成功地继承了这一组方法，对这些方法进行反编译就能确定是继承自类型B，因为B的嵌套层次比A要浅，编译器优先选择短路径。

4.4 本章小结

本章首先探索了方法接收者的传递方式，发现接收者实际上就是编译器隐式传递的第1个参数，也是通过栈传递的，所以方法调用本质上与普通的函数调用是一样的。同时，还发现了编译器会为值接收者方法生成指针接收者的包装方法，暂时还没有弄清楚这样做的意义，然后又探索了Method Value的实现原理，发现其本质上依然是Function Value，不过编译器会自动为基于对象的Method Value生成闭包函数，闭包捕获的类型与方法接收者的类型一致。最后还探索了组合式继承之下编译器是如何为继承的方法生成包装方法的。

通过了解底层的具体实现，对方法有了更深入的理解，接下来的第5章我们将走进Go语言的动态语言特性。

第5章
接口

接口在Go语言中扮演着非常重要的角色，它是多态、反射和类型断言等一众动态语言特性的基础。本章将通过反编译、runtime源码分析等手段，逐步梳理清楚接口的底层实现。

5.1 空接口

这里所谓的空接口并不是nil，而是指不包含任何方法的接口，也就是interface{}。在面向对象编程中，接口用来对行为进行抽象，也就是定义对象需要支持的操作，这组操作对应的就是接口中列出的方法。不包含任何方法的接口可以认为不要求对象支持任何操作，因此能够接受任意类型的赋值，所以Go语言的interface{}什么都能装。

5.1.1 一个更好的void*

如果用unsafe.Sizeof（）函数获取一个interface{}类型变量的大小，在64位平台上是16字节，在32位平台上是8字节。interface{}类型本质上是个struct，由两个指针类型的成员组成，在runtime中可以找到对应的struct定义，代码如下：

```
type eface struct {
    _type * _type
    data unsafe.Pointer
}
```

还有一个专门的类型转换函数efaceOf（），该函数接受的参数是一个interface{}类型的指针，返回值是一个eface类型的指针，内部实际只进行了一下指针类型的转换，也就说明interface{}类型在内存布局层面与eface类型完全等价。efaceOf（）函数的代码如下：

```
func efaceOf(ep * interface{}) * eface {
    return ( * eface)(unsafe.Pointer(ep))
}
```

接下来看一下eface的两个指针成员，data字段比较好理解，它是一个unsafe.Pointer类型的指针，用来存储实际数据的地址。unsafe.Pointer在含义上和C语言中的void*有些类似，只用来表明这是一个指针，并不限定指向的目标数据的类型，可以接受任意类型的地址。至于_type类型，之前在探索变量逃逸和闭包的时候曾经见到过，当时是作为runtime.newobject（）函数的参数出现的，它在Go语言的runtime中被用来描述数据类型，笔者习惯称之为类型元数据。eface的这个_type字段用来描述data的类型元数据，也就是说它给出了data的数据类型。

例如，把一个int类型变量n的地址赋值给一个interface{}类型的变量e，代码如下：

```
//第 5 章/code_5_1.go
var n int
var e interface{} = &n
```

如图5-1所示，变量e的data字段存储的是变量n的地址，而变量e的_type字段存储的是*int类型的类型元数据的地址。

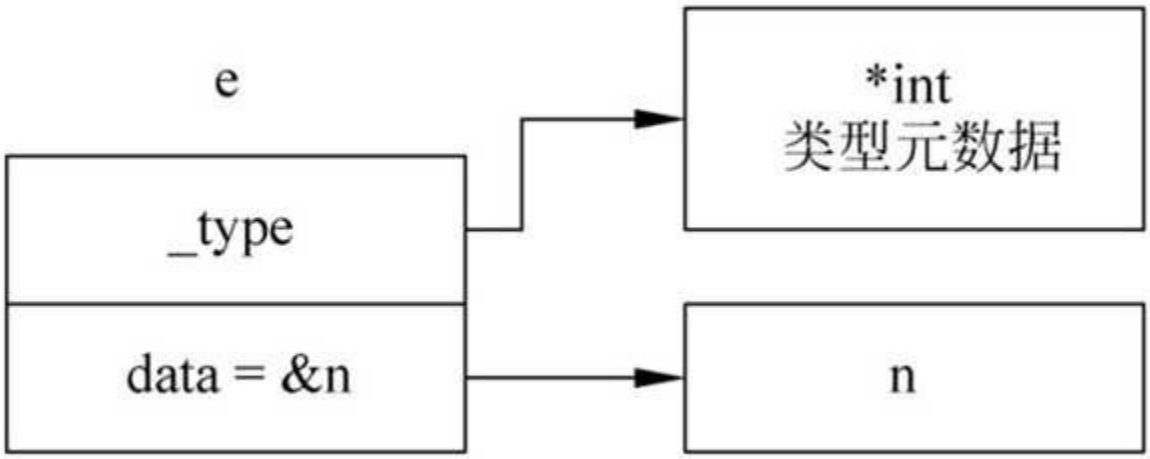

图5-1 空接口变量e与赋值变量n的关系

与void*相比，interface{}通过多出来的这个_type字段给出了数据的类型信息，程序在运行阶段可以基于这种类型信息对数据进行特定操作，因此interface{}就相当于一个增强版的void*。

就变量n本身而言，它的类型信息只会被编译器使用，编译阶段参考这种类型信息来分配存储空间、生成机器指令，但是并不会把这种类型信息写入最终生成的可执行文件中。从内存布局的角度来讲，变量n在64位和32位平台分别占用8字节和4字节，占用的这些空间全部用来存放整型的值，没有任何空间被用来存放整型类型信息。

把变量n的地址赋值给interface{}类型的变量e的这个操作，意味着编译器要把*int的类型元数据生成出来，并把其地址赋给变量e的_type字段，这些类型元数据会被写入最终的可执行文件，程序在运行阶段即取即用。这个简单的赋值操作实际上完成了类型信息的萃取。

为了能够方便地通过反编译进行验证，将第5章/code_5_1.go稍微修改一下，代码如下：

```
//第 5 章/code_5_2.go
func p2e(p * int) (e interface{}) {
    e = p
    return
}
```

然后进行编译和反编译，得到相应的汇编代码如下：

```
$ go tool compile -trimpath="`pwd`=>" -l -p gom eface.go
$ go tool objdump -S -s '^gom.p2e$' eface.o
TEXT gom.p2e(SB) gofile..eface.go
        return
  0x7b0         488d0500000000          LEAQ 0(IP), AX     [3:7]R_PCREL:type.*int
  0x7b7         4889442410              MOVQ AX, 0x10(SP)
  0x7bc         488b442408              MOVQ 0x8(SP), AX
  0x7c1         4889442418              MOVQ AX, 0x18(SP)
  0x7c6         c3                      RET
```

虽然看起来很简单，还是把它转换为等价的伪代码，这样更加直观，代码如下：

```
func p2e(n *int) (e eface) {
    e._type = &type.*int
    e.data = unsafe.Pointer(n)
    return
}
```

其中的type.*int就是需要的类型元数据，编译器在生成的指令中为它的地址预留了位置，等到链接阶段生成可执行文件时链接器会填写上实际的地址。

提到变量的类型，一般指的是声明类型，例如变量n的声明类型是int。在Go这种强类型语言中，变量的声明类型是不能改变的，即使通过类型转换得到一个新的变量，原变量的类型还是不会改变。对于interface{}类型的变量e，它的声明类型是interface{}，这一点也是不能改变的。变量e就像是一个容器，可以装载任意类型的数据，并通过_type字段记录数据的类型，无论装载什么类型的数据，容器本身的类型不会改变。因为_type会随着变量e装载不同类型的数据而发生改变，所以后文中将它称为变量e的动态类型，并相应地把变量e的声明类型称为静态类型。

5.1.2 类型元数据

在C语言中类型信息主要存在于编译阶段，编译器从源码中得到具体的类型定义，并记录到相应的内存数据结构中，然后根据这些类型信息进行语法检查、生成机器指令等。例如x86整数加法和浮点数加法采用完全不同的指令集，编译器根据数据的类型来选择。这些类型信息并不会被写入可执行文件，即使作为符号数据被写入，也是为了方便调试工具，并不会被语言本身所使用。Go与C语言不同的是，在设计之初就支持面向对象编程，还有其他一些动态语言特征，这些都要求运行阶段能够获得类型信息，所以语言的设计者就把类型信息用统一的数据结构来描述，并写入可执行文件中供运行阶段使用，这就是所谓的类型元数据。

既然已经不止一次遇到_type这种类型元数据类型，这里就来简单看一下它的具体定义。摘抄自Go 1.15版本的runtime源码，代码如下：

```
type _type struct {
    size       uintptr
    ptrdata    uintptr
    hash       uint32
    tflag      tflag
    align      uint8
    fieldAlign uint8
    kind       uint8
    equal func(unsafe.Pointer, unsafe.Pointer) bool
    gcdata     *Byte
    str        nameOff
    ptrToThis typeOff
}
```

各个字段的含义及主要用途如表5-1所示。

表5-1 _type各字段的含义及主要用途

字段	含义及主要用途
size	表示此类型的数据需要占用多少字节的存储空间，runtime 中很多地方会用到它，最典型的就是内存分配的时候，例如 newobject()、mallocgc()
ptrdata	ptrdata 表示数据的前多少字节包含指针，用来在应用写屏障时优化范围大小。例如某个 struct 类型在 64 位平台上占用 32 字节，但是只有第 1 个字段是指针类型，这个值就是 8，剩下的 24 字节就不需要写屏障了。GC 进行位图标记的时候，也会用到该字段
hash	当前类型的哈希值，runtime 基于这个值构建类型映射表，加速类型比较和查找
tflag	额外的类型标识，目前由 4 个独立的二进制位组合而成。tflagUncommon 表明这种类型元数据结构后面有个紧邻的 uncommontype 结构，uncommontype 主要在自定义类型定义方法集时用到。tflagExtraStar 表示类型的名称字符串有个前缀的 *，因为对于程序中的大多数类型 T 而言，*T 也同样存在，复用同一个名称字符串能够节省空间。tflagNamed 表示类型有名称。tflagRegularMemory 表示相等比较和哈希函数可以把该类型的数据当成内存中的单块区间来处理
align	表示当前类型变量的对齐边界
fieldAlign	表示当前类型的 struct 字段的对齐边界
kind	表示当前类型所属的分类，目前 Go 语言的 reflect 包中定义了 26 种有效分类
equal	用来比较两个当前类型的变量是否相等
gcdata	和垃圾回收相关，GC 扫描和写屏障用来追踪指针
str	偏移，通过 str 可以找到当前类型的名称等文本信息
ptrToThis	偏移，假设当前类型为 T，通过它可以找到类型 *T 的类型元数据

_type提供了适用于所有类型的最基本的描述，对于一些更复杂的类型，例如复合类型slice和map等，runtime中分别定义了maptype、slicetype等对应的结构。例如slicetype就是由一个用来描述类型本身的_type结构和一个指向元素类型的指针组成，代码如下：

```
type slicetype struct {
    typ _type
    elem * _type
}
```

Go语言允许为自定义类型实现方法，这些方法的相关信息也会被记录到自定义类型的元数据中，一般称为类型的方法集信息。在梳理_type结构的各个字段时，没有发现任何跟方法集有关的字段，那么runtime是如何以_type为起点来找到方法集信息的呢？

考虑到Go语言只允许为自定义类型实现方法，所以要找到元数据中的方法集信息，就要从自定义类型出发。自定义类型，也就是代码中使用type关键字定义的类型，示例代码如下：

```
type Integer int
```

图5-2 自定义类型Integer的类型元数据结构

在上述代码中，int本身是内置类型，不允许为其实现方法，而基于int定义的Integer是个自定义类型。还记得_type结构的tflag字段是几个标志位，当tflagUncommon这一位为1时，表示类型为自定义类型。从runtime的源码可以发现，_type类型有一个uncommon（）方法，对于自定义类型可以通过此方法得到一个指向uncommontype结构的指针，也就是说编译器会为自定义类型生成一个uncommontype结构，例如上述自定义类型Integer的类型元数据结构如图5-2所示。

uncommontype结构的定义代码如下：

```
type uncommontype struct {
    pkgpath nameOff
    mcount  uint16 //number of methods
    xcount  uint16 //number of exported methods
    moff    uint32 //offset from this uncommontype to [mcount]method
    _       uint32 //unused
}
```

通过pkgpath可以知道定义该类型的包名称，mcount表示该类型共有多少个方法，xcount表示有多少个方法被导出，也就是首字母大写使包外可访问。moff是个偏移值，那里就是方法集的元数据，也就是一组method结构构成的数组。例如，若为自定义类型Integer定义两个方法，它的类型元数据及其method数组的内存布局如图5-3所示。

method数组中每个method结构对应一个方法，代码如下：

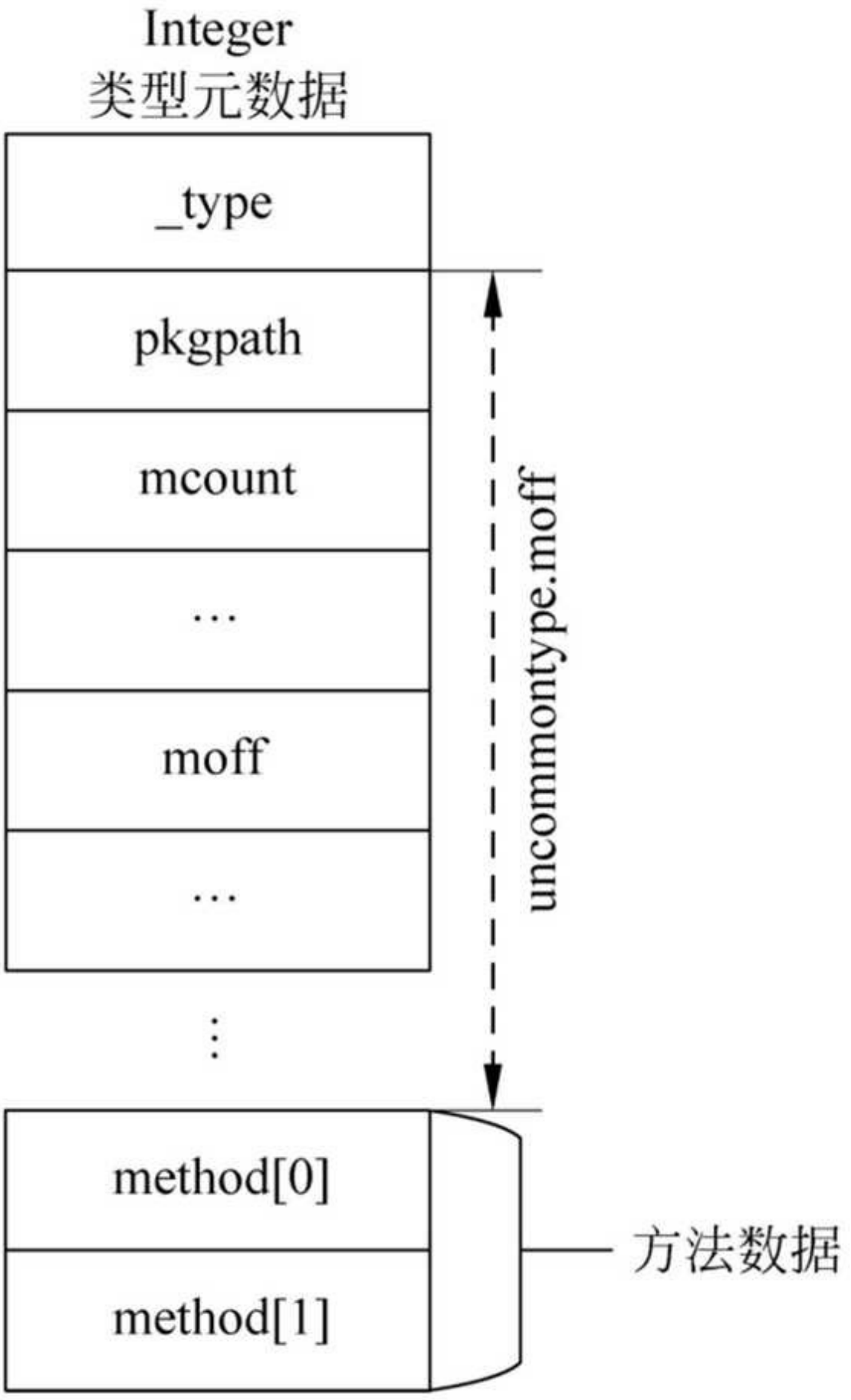
Integer
类型元数据
_type
pkgpath
mcount
…
moff
…
uncommontype.moff
method[0]
method[1]
方法数据

图5-3 Integer类型元数据及其method数组的内存布局

```
type method struct {
    name nameOff
    mtyp typeOff
    ifn textOff
    tfn textOff
}
```

通过name偏移能够找到方法的名称字符串，mtyp偏移处是方法的类型元数据，进一步可以找到参数和返回值相关的类型元数据。若自定义类型有A（）、B（）、C（）3个方法（如图5-4所示），则method数组会按照name升序排列，运行阶段可以高效地进行二分查找。

ifn是供接口调用的方法地址，tfn是正常的方法地址，这两个方法地址有什么不同呢？ifn的接收者类型一定是指针，而tfn的接收者类型跟源代码中的实现一致，这里先不进行过多的解释，在5.2.3节中会深入分析这两者的不同。

以上这些类型元数据都是在编译阶段生成的，经过链接器的处理后被写入可执行文件中，runtime中的类型断言、反射和内存管理等都依赖于这些元数据，本章的后续内容都与这些类型元数据有着密切的关系。

5.1.3 逃逸与装箱

由于interface{}的data字段是个指针，存储的是数据的地址，所以不可避免地也会跟变量逃逸扯上关系。在进行逃逸分析的时候，直接把interface{}当作原始数据类型的指针来看待即可，效果是等价的，此处不再赘述。

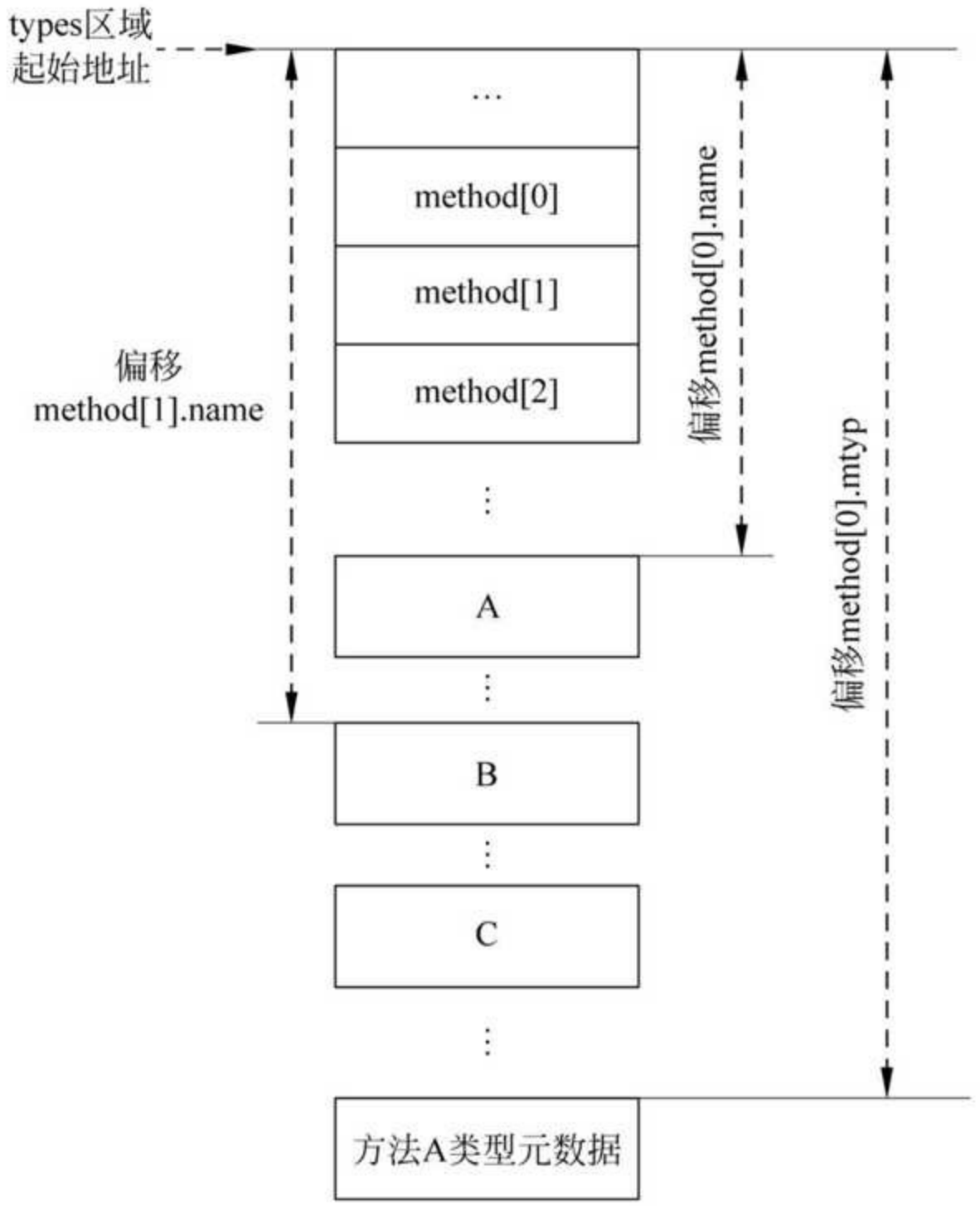

图5-4 method数组排序

接下来有一个需要仔细探索的问题：data字段是个指针，那么它是如何接收来自一个值类型的赋值的呢？示例代码如下：

```
//第 5 章/code_5_3.go
n := 10
var e interface{} = n
```

在上述代码中变量e的数据结构如图5-5所示。

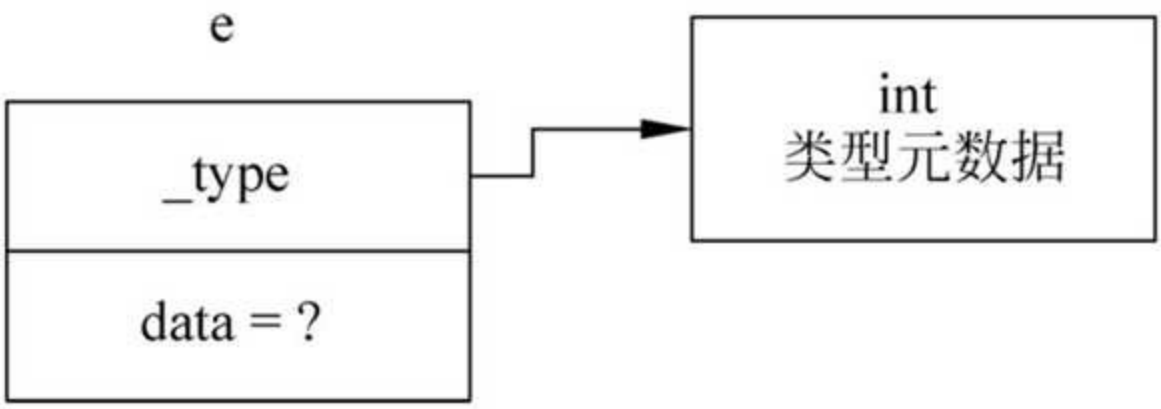

图5-5 interface{}类型的变量e的数据结构

e.data这里存储的是什么还真不太好猜测，还是直接反编译一下比较简单。依旧把第5章/code_5_3.go放到一个函数中，代码如下：

```
//第 5 章/code_5_4.go
func v2e(n int) (e interface{}) {
    e = n
    return
}
```

反编译后得到汇编代码如下：

```
$ go tool objdump -S -s '^gom.v2e$' eface.o
TEXT gom.v2e(SB) gofile..eface.go
func v2e(n int) (e interface{}) {
  0xb0a       65488b0c2528000000      MOVQ GS:0x28, CX
  0xb13       488b8900000000          MOVQ 0(CX), CX          [3:7]R_TLS_LE
  0xb1a       483b6110                CMPQ 0x10(CX), SP
  0xb1e       763c                    JBE 0xb5c
  0xb20       4883ec18                SUBQ $0x18, SP
  0xb24       48896c2410              MOVQ BP, 0x10(SP)
  0xb29       488d6c2410              LEAQ 0x10(SP), BP
        e = n
  0xb2e       488b442420              MOVQ 0x20(SP), AX
  0xb33       48890424                MOVQ AX, 0(SP)
  0xb37       e800000000              CALL 0xb3c              [1:5]R_CALL:runtime.convT64
  0xb3c       488b442408              MOVQ 0x8(SP), AX
        return
  0xb41       488d0d00000000          LEAQ 0(IP), CX          [3:7]R_PCREL:type.int
  0xb48       48894c2428              MOVQ CX, 0x28(SP)
  0xb4d       4889442430              MOVQ AX, 0x30(SP)
  0xb52       488b6c2410              MOVQ 0x10(SP), BP
  0xb57       4883c418                ADDQ $0x18, SP
  0xb5b       c3                      RET
func v2e(n int) (e interface{}) {
  0xb5c       e800000000              CALL 0xb61
[1:5]R_CALL:runtime.morestack_noctxt
  0xb61       eba7                    JMP gom.v2e(SB)
```

虽然代码篇幅不太长，但还是转换成等价的伪代码比较容易理解，代码如下：

```
func v2e(n int) (e eface) {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    e.data = runtime.convT64(n)
    e._type = &type.int
    return
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

忽略与栈增长相关的代码，真正感兴趣的就是为变量e的两个成员赋值的这两行代码。先把变量n的值作为参数调用runtime.convT64（）函数，并把返回值赋给了e.data。又把type.int的地址赋给了e._type。后者倒是比较容易理解，因为变量e的动态类型是变量n的类型，即int，但这个runtime.convT64（）函数的逻辑还需要再看一下，看一看它的返回值究竟是什么。
runtime.convT64（）函数的源代码如下：

```
func convT64(val uint64) (x unsafe.Pointer) {
    if val < uint64(len(staticuint64s)) {
        x = unsafe.Pointer(&staticuint64s[val])
    } else {
        x = mallocgc(8, uint64Type, false)
        *(*uint64)(x) = val
    }
    return
}
```

当val的值小于staticuint64s的长度时，直接返回staticuint64s中第val项的地址。否则就通过mallocgc（）函数分配一个uint64，把val的值赋给它并返回它的地址。这个staticuint64s如图5-6所示，是个长度为256的uint64数组，每个元素的值都跟下标一致，存储了0～255这256个值，主要用来避免常用数字频繁地进行堆分配。

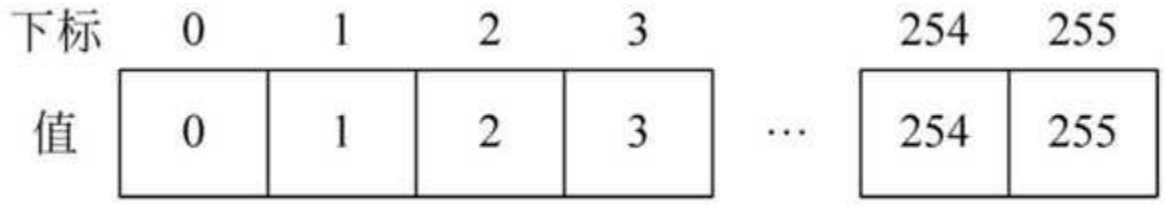

图5-6 staticuint64s数组

整体来看convT64（）函数的功能，实际上就是堆分配一个uint64，并且将val参数作为初始值赋给它。由于示例中变量n的值为10，在staticuint64s的长度范围内，所以变量e的data字段存储的就是staticuint64s中下标为10的存储空间的地址，如图5-7所示。

通过staticuint64s这种优化方式，能够反向推断出：被convT64分配的这个uint64，它的值在语义层面是不可修改的，是个类似const的常量，这样设计主要是为了跟interface{}配合来模拟装载值。interface{}被设计成一个容器，但它本质上是个指针，可以直接装载地址，用来实现装载值，实际的内存要分配在别的地方，并把内存地址存储在这里。convT64（）函数的作用就是分配这个存储值的内存空间，实际上runtime中有一系列这类函数，如convT32（）、convTstring（）和convTslice（）等。

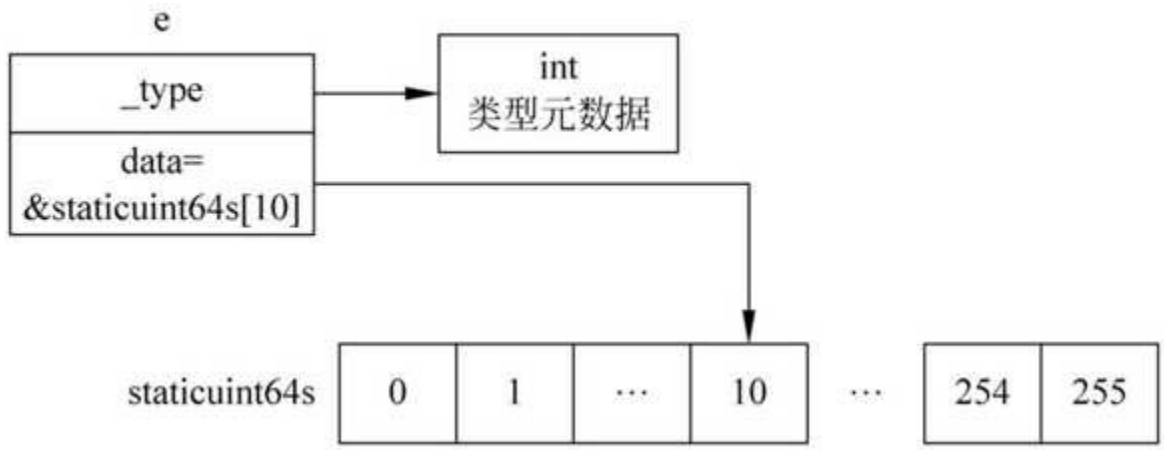

图5-7 变量e的数据结构

至于为什么这个值不可修改，因为interface{}只是一个容器，它支持把数据装入和取出，但是不支持直接在容器里修改。这有些类似于Java和C#中的自动装箱，只不过interface{}是个万能包装类。

那么值类型装箱就一定会进行堆分配吗？这个问题也需要验证。既然已经知道逃逸会造成堆分配，那就构造一个值类型装箱但不逃逸的场景，示例代码如下：

```
//第 5 章/code_5_5.go
func fn(n int) bool {
    return notNil(n)
}

func notNil(a interface{}) bool {
    return a != nil
}
```

编译时需要禁止内联优化，编译器还能够通过notNil（）函数的代码实现判定有没有发生逃逸，反编译fn（）函数得到的汇编代码如下：

```
$ go tool objdump -S -s '^gom.fn$' eface.o
TEXT gom.fn(SB) gofile..eface.go
func fn(n int) bool {
  0xfd6         65488b0c2528000000      MOVQ GS:0x28, CX
  0xfdf         488b8900000000          MOVQ 0(CX), CX      [3:7]R_TLS_LE
  0xfe6         483b6110                CMPQ 0x10(CX), SP
  0xfea         764a                    JBE 0x1036
  0xfec         4883ec28                SUBQ $0x28, SP
  0xff0         48896c2420              MOVQ BP, 0x20(SP)
  0xff5         488d6c2420              LEAQ 0x20(SP), BP
        return notNil(n)
  0xffa         488b442430              MOVQ 0x30(SP), AX
  0xfff         4889442418              MOVQ AX, 0x18(SP)
  0x1004        488d0500000000          LEAQ 0(IP), AX      [3:7]R_PCREL:type.int
  0x100b        48890424                MOVQ AX, 0(SP)
  0x100f        488d442418              LEAQ 0x18(SP), AX
  0x1014        4889442408              MOVQ AX, 0x8(SP)
  0x1019        e800000000              CALL 0x101e      [1:5]R_CALL:gom.notNil
  0x101e        0fb6442410              MOVZX 0x10(SP), AX
  0x1023        88442438                MOVB AL, 0x38(SP)
  0x1027        488b6c2420              MOVQ 0x20(SP), BP
  0x102c        4883c428                ADDQ $0x28, SP
  0x1030        c3                      RET
func fn(n int) bool {
  0x1031        0f1f440000              NOPL 0(AX)(AX*1)
  0x1036        e800000000              CALL 0x103b      [1:5]R_CALL:runtime.morestack_noctxt
  0x103b        eb99                    JMP gom.fn(SB)
```

转换为等价的伪代码如下：

```
func fn(n int) bool {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    v := n
    return notNil(eface{_type: &type.int, data: &v})
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

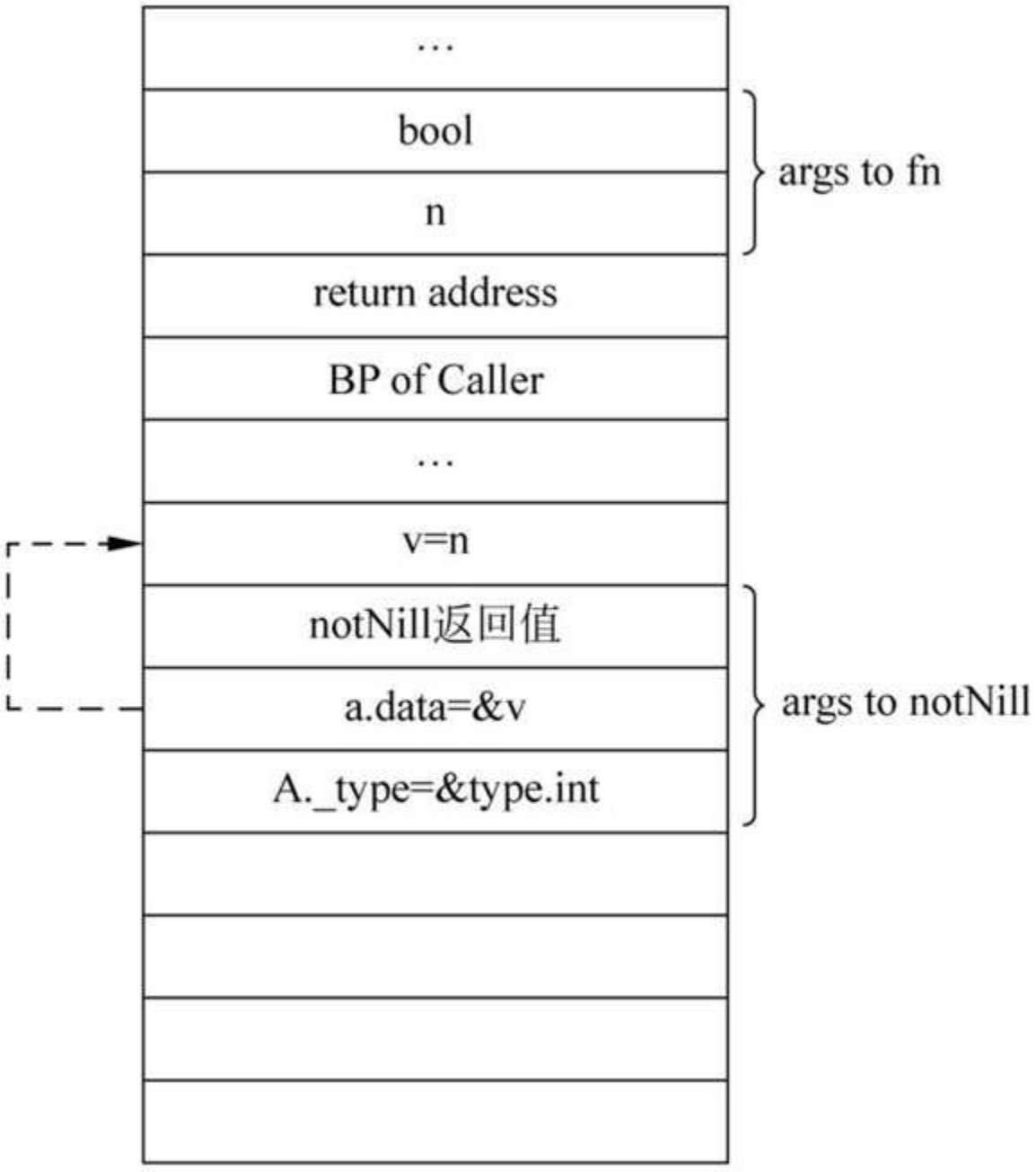

图5-8 fn（）函数的调用栈

伪代码中fn（）函数的调用栈如图5-8所示，注意局部变量v，它实际上是被编译器采用隐式方式分配的，被用作变量n的值的副本，却并没有分配到堆上。

interface{}在装载值的时候必须单独复制一份，而不能直接让data存储原始变量的地址，因为原始变量的值后续可能会发生改变，这就会造成逻辑错误。

上面的例子总算证明了装箱不一定进行堆分配，是否堆分配还是要经过逃逸分析。只有值类型装箱后又涉及逃逸的情况，才会用到runtime中的一系列convT（）函数。

关于不包含任何方法的空接口就先研究到这里，下面来看一下包含方法的非空接口。

5.2 非空接口

与空接口对应，非空接口指的是至少包含一种方法的接口，就像io.Reader和io.Writer。非空接口通过一组方法对行为进行抽象，从而隔离具体实现达到解耦的目的。Go的接口比Java等语言中的接口更加灵活，自定义类型不需要通过implements关键字显式地标明自己实现了某个接口，只要实现了接口中所有的方法就实现了该接口。也只有实现了接口中所有的方法，才算是实现了该接口。

本节探索一下Go语言中非空接口的底层实现，后文中提到的接口均指非空接口，为了能够加以区分，不再将interface{}称为空接口，而是直接称为interface{}类型。

5.2.1 动态派发

在面向对象编程中，接口的一个核心功能是支持多态，实际上就是方法的动态派发。调用接口的某个方法时，调用者不需要知道背后对象的具体类型就能调用对象的指定方法。例如类型A和B都实现了fmt.Stringer接口，示例代码如下：

```
//第 5 章/code_5_6.go
type A struct{}

type B struct{}

func (A) String() string {
    return "This is A"
}

func (B) String() string {
    return "This is B"
}

func toString(o fmt.Stringer) string {
    return o.String()
}

func main() {
    println(toString(A{})) //This is A
    println(toString(B{})) //This is B
}
```

其中toString（）函数的实现者并不需要知道参数o背后的具体类型，接口机制会在运行阶段自动完成方法调用的动态派发，所以toString（A{}）会调用类型A的String（）方法，进而返回字符串 " This is A " ，而toString（B{}）则返回字符串 " This is B " 。下面分析一下动态派发如何实现。

1. 方法地址静态绑定

要进行方法（函数）调用，有两点需要确定：一是方法的地址，也就是在代码段中的指令序列的起始地址；二是参数及调用约定，也就是要传递什么参数及如何传递的问题（通过栈或者寄存器），返回值的读取也包含在调用约定范畴内。调用约定及编译器如何根据调用约定来生成相关指令，在第3章已经探索过了，这里的重点是如何确定目标方法的地址。

首先来看一个不使用接口而直接通过自定义类型的对象实例调用其方法的例子，代码如下：

```
//go:noinline
func ReadFile(f *os.File, b []byte) (n int, err error) {
    return f.Read(b)
}
```

上述ReadFile（）函数实际上只调用了*os.File类型的Read（）方法，为了方便后续反编译，禁止编译器对该函数进行内联。对build得到的可执行文件进行反编译，得到对应的汇编代码如下：

```
$ go tool objdump -S -s '^main.ReadFile$' gom.exe
TEXT main.ReadFile(SB) C:/gopath/src/fengyoulin.com/gom/main.go
func ReadFile(f *os.File, b []byte) (n int, err error) {
  0x4b4240      65488b0c2528000000    MOVQ GS:0x28, CX
  0x4b4249      488b8900000000        MOVQ 0(CX), CX
  0x4b4250      483b6110              CMPQ 0x10(CX), SP
  0x4b4254      7662                  JBE 0x4b42b8
  0x4b4256      4883ec40              SUBQ $0x40, SP
  0x4b425a      48896c2438            MOVQ BP, 0x38(SP)
  0x4b425f      488d6c2438            LEAQ 0x38(SP), BP
        return f.Read(b)
  0x4b4264      488b442448            MOVQ 0x48(SP), AX
  0x4b4269      48890424              MOVQ AX, 0(SP)
  0x4b426d      488b442450            MOVQ 0x50(SP), AX
  0x4b4272      4889442408            MOVQ AX, 0x8(SP)
  0x4b4277      488b442458            MOVQ 0x58(SP), AX
  0x4b427c      4889442410            MOVQ AX, 0x10(SP)
  0x4b4281      488b442460            MOVQ 0x60(SP), AX
  0x4b4286      4889442418            MOVQ AX, 0x18(SP)
  0x4b428b      e87035ffff            CALL os.(*File).Read(SB)
  0x4b4290      488b442420            MOVQ 0x20(SP), AX
  0x4b4295      488b4c2428            MOVQ 0x28(SP), CX
  0x4b429a      488b542430            MOVQ 0x30(SP), DX
  0x4b429f      4889442468            MOVQ AX, 0x68(SP)
  0x4b42a4      48894c2470            MOVQ CX, 0x70(SP)
  0x4b42a9      4889542478            MOVQ DX, 0x78(SP)
  0x4b42ae      488b6c2438            MOVQ 0x38(SP), BP
  0x4b42b3      4883c440              ADDQ $0x40, SP
  0x4b42b7      c3                    RET
func ReadFile(f *os.File, b []byte) (n int, err error) {
  0x4b42b8      e8a3e8faff            CALL runtime.morestack_noctxt(SB)
  0x4b42bd      eb81                  JMP main.ReadFile(SB)
```

可以看到CALL指令直接调用了os.（*File）.Read（）方法，地址以Offset的形式编码在指令中。实际上这个地址是编译器在OBJ文件中预留了空间，然后由链接器填写实际的Offset，有兴趣的读者可以自己反编译OBJ文件查看，这里不再赘述。与汇编代码等价的Go风格的伪代码如下：

```
func ReadFile(f *os.File, b []byte) (n int, err error) {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    return os.(*File).Read(f, b)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

排除掉这些栈增长代码，就剩下一个再普通不过的函数（方法）调用了。从汇编语言的角度来看，上述方法的调用是通过CALL指令+相对地址实现的，方法地址在可执行文件构建阶段就确定了，一般将这种情况称为方法地址的静态绑定。

显而易见，这种地址静态绑定的方式无法支持方法调用的动态派发，因为编译阶段并不知道对象的具体类型，所以无法确定要绑定到何种方法。对于动态派发来讲，编译阶段能够确定的是要调用的方法的名字，以及方法的原型（参数与返回值列表）。以第5章/code_5_6.go中的toString（）函数为例，要调用的方法名字是String，没有入参，有一个string类型的返回值。实际上有这些信息就足够了，运行阶段根据这些信息就能完成动态派发。

2．动态查询类型元数据

至于动态派发的代码实现，可以有很多种不同版本。先不去管Go语言到底是如何实现的，如果让我们来设计，可以怎么做呢？

我们假设非空接口的数据结构与eface相同，同样包含一个类型元数据指针和一个数据指针。5.1节已经简单地分析了与类型元数据相关的数据结构，知道自定义类型的类型元数据中存有方法集信息，方法集信息是一组method结构构成的数组，通过它可以找到对应方法的方法名、参数和返回值的类型，以及代码的地址。method结构的代码如下：

```
type method struct {
    name nameOff
    mtyp typeOff
    ifn textOff
    tfn textOff
}
```

类型元数据中的method数组是按照方法名升序排列的，可以直接应用二分法查找。运行阶段利用这些信息就可以根据方法名和原型动态绑定方法地址了。假如现在有一个io.Reader类型的接口变量r，其背后动态类型是*os.File，代码如下：

```
var r io.Reader = f
n, err := r.Read(buf)
```

首先，可以通过变量r得到*os.File的类型元数据，如图5-9所示，然后根据方法名称Read以二分法查找匹配的method结构，找到后再根据method.mtyp得到方法本身的类型元数据，最后对比方法原型是否一致（参数和返回值的类型、顺序是否一致）。如果原型一致，就找到了目标方法，通过method.ifn字段得到方法的地址，然后就像调用普通函数一样调用就可以了。

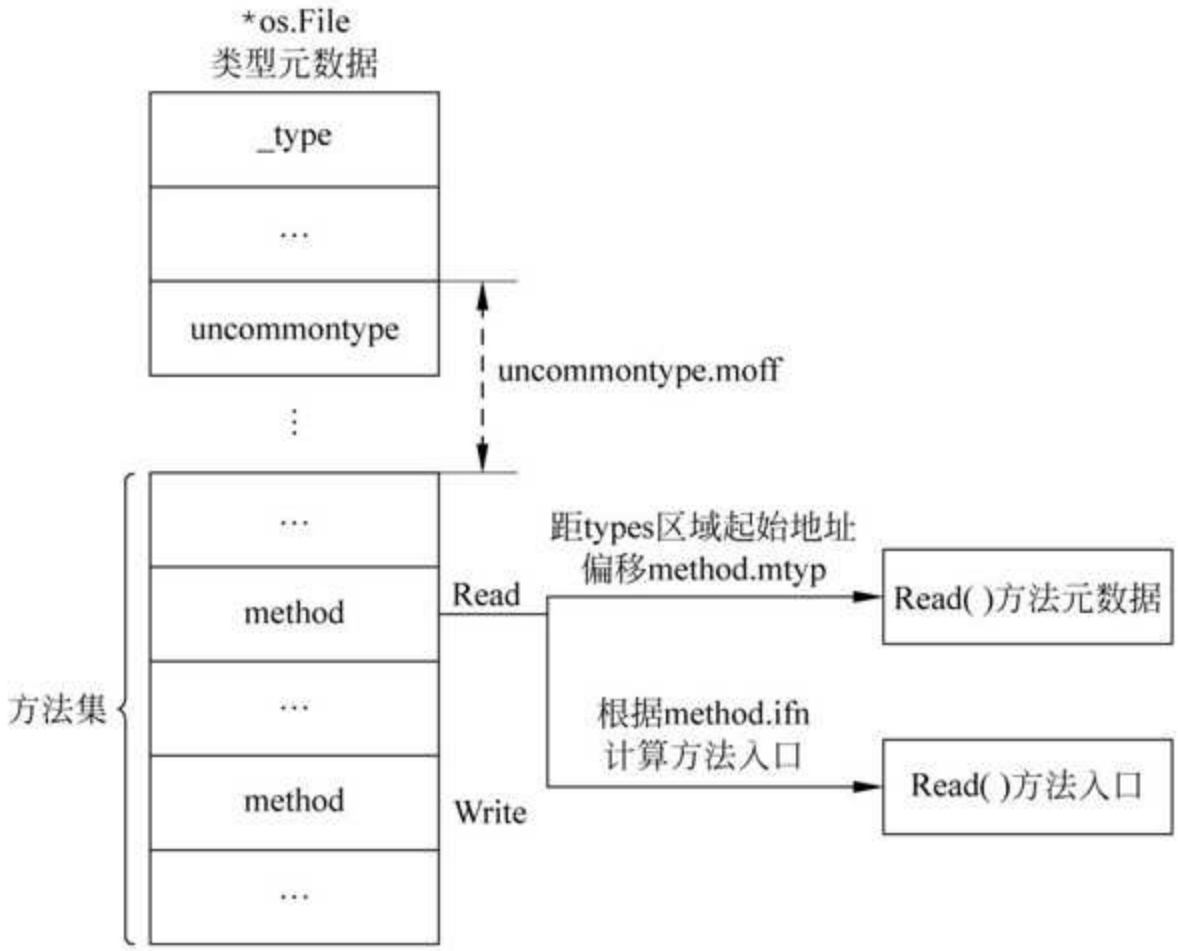

图5-9 *os.File的类型元数据

单就动态派发而言，这种方式确实可以实现，但是有一个明显的问题，那就是效率低，或者说性能差。跟地址静态绑定的方法调用比起来，原本一条CALL指令完成的事情，这里又多出了一次二分查找加方法原型匹配，增加的开销不容小觑，可能会造成动态派发的方法比静态绑定的方法多一倍开销甚至更多，所以必须进行优化。不能在每次方法调用前都到元数据中去查找，尽量做到一次查找、多次使用，这里可以一定程度上参考C++的虚函数表实现。

3．C++虚函数机制

C++中的虚函数机制跟接口的思想很相似，编程语言允许父类指针指向子类对象，当通过父类的指针来调用虚函数时，就能实现动态派发。具体实现原理就是，编译器为每个包含虚函数的类都生成一张虚函数表，实际上是个地址数组，按照虚函数声明的顺序存储了各个虚函数的地址。此外还会在类对象的头部安插一个虚指针（GCC安插在头部，其他编译器或有不同），指向类型对应的虚函数表。运行阶段通过类对象指针调用虚函数时，会先取得对象中的虚指针，进一步找到对象类型对应的虚函数表，然后基于虚函数声明的顺序，以数组下标的方式从表中取得对应函数的地址，这样整个动态派发过程就完成了。

人们经常在父类中只声明一组纯虚函数，也就是不实现函数体，这种只包含一组纯虚函数的类就更符合接口的设计思想了。例如，将父类Type用作接口，声明两个纯虚函数，两个子类A和B分别继承自父类Type，并且实现这两个虚函数，相当于实现了接口，示例代码如下：

```
//第 5 章/code_5_7.cpp
class Type {
public:
    virtual string Name() = 0;
    virtual size_t Size() = 0;
};

class A : public Type {
public:
    string Name() {
        return "A";
    }
    size_t Size() {
        return sizeof( * this);
    }
};

class B : public Type {
public:
    string Name() {
        return "B";
    }
    size_t Size() {
        return sizeof( * this);
    }
private:
    int somedata;
};
```

可以测试多态的效果，测试代码如下：

```
//第 5 章/code_5_8.cpp
Type * pts[2] = {new A, new B};
for(int i = 0; i < 2; ++i) {
    cout << pts[i] -> Name() << "," << pts[i] -> Size() << endl;
}
```

在笔者的64位计算机上，输出结果如下：

```
A,8
B,16
```

图5-10以在上述代码中的pts为起点，展示出A、B的对象实例、各自虚指针及虚函数表在内存中的关联关系，这样就能一目了然地看懂C++虚函数的动态派发原理了。

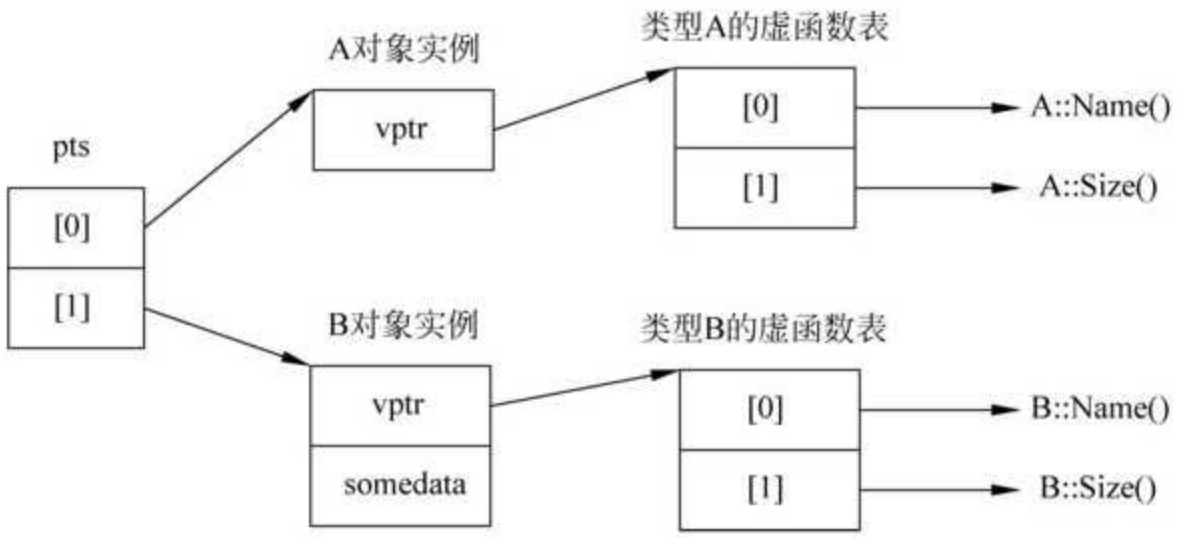

图5-10 C++虚函数动态派发示例

运行阶段通过父类指针调用虚函数时，并不需要关心指向的是哪个子类。数组pts的元素类型是Type*，运行阶段先通过pts[0]和pts[1]找到子类对象中的虚指针vptr，再通过vptr最终定位到子类类型的虚函数表，根据函数声明的顺序按下标取得函数的实际地址。两个指针加一个数组，就完成了整个动态派发的核心逻辑，效率还是非常高的。

参考C++的虚函数表思想，再回过头来看Go语言中接口的设计，如果把这种基于数组的函数地址表应用在接口的实现中，基本就能消除每次查询地址造成的性能开销。显然这里需要对eface结构进行扩展，加入函数地址表相关字段，经过扩展的eface姑且称作efacex，代码如下：

```
type efacex struct {
    tab  * struct {
        _type * _type
        fun  [1]uintptr //方法数
    }
    data unsafe.Pointer
}
```

把原本的类型元数据指针_type和新添加的方法地址数组fun打包到一个struct中，并用这个struct的地址替换掉eface中原本的_type字段，得到修改后的efacex，如图5-11所示。

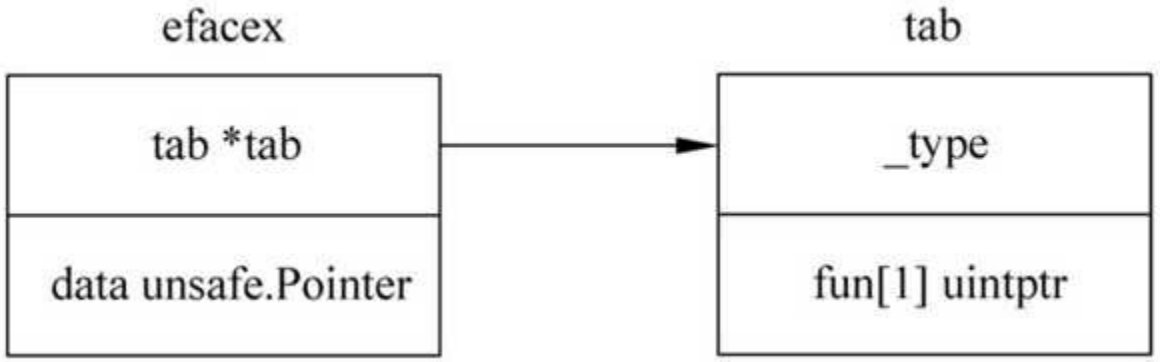

图5-11 参照C++虚函数机制修改后的非空接口数据结构

添加的fun数组相当于C++的虚函数表，这个数组的长度与接口中方法的个数一致，是动态分配

的。在struct的最后放置一个动态长度的数组，这是C语言中常用的技巧。什么时候为fun数组赋值呢？当然是在为整个efacex结构赋值的时候最合适，示例代码如下：

```
//第 5 章/code_5_9.go
f, _ := os.Open("gom.go")
var rw io.ReadWriter
rw = f
```

从f到rw这个看似简单的赋值，至少要展开成如下几步操作：①根据rw接口中方法的个数动态分配tab结构，这里有两个方法，fun数组的长度是2。②从*os.File的方法集中找到Read（）方法和Write（）方法，把地址写入fun数组对应下标。③把*os.File的元数据地址赋值给tab._type。④把f赋值给data，也就是数据指针。赋值后rw的数据结构如图5-12所示。

这样一来，只需要在为接口变量赋值的时候对方法集进行查找，后续调用接口方法的时候，就可以像C++的虚函数那样直接按数组下标读取地址了。

实际上，fun数组也不用每次都重新分配和初始化，从指定具体类型到指定接口类型变量的赋值，运行阶段无论发生多少次，每次生成的fun数组都是相同的。例如从*os.File到io.ReadWriter的赋值，每次都会生成一个长度为2的fun数组，数组的两个元素分别用于存储（*os.File）.Read和（*os.File）.Write的地址。也就是说通过一个确定的接口类型和一个确定的具体类型，就能够唯一确定一个fun数组，因此可以通过一个全局的map将fun数组进行缓存，这样就能进一步减少方法集的查询，从而优化性能。

本节结合C++的虚函数机制，简单地推演了一下动态派发的实现原理，跟Go语言的实现已经很接近了，接下来看一下Go语言中的具体实现。

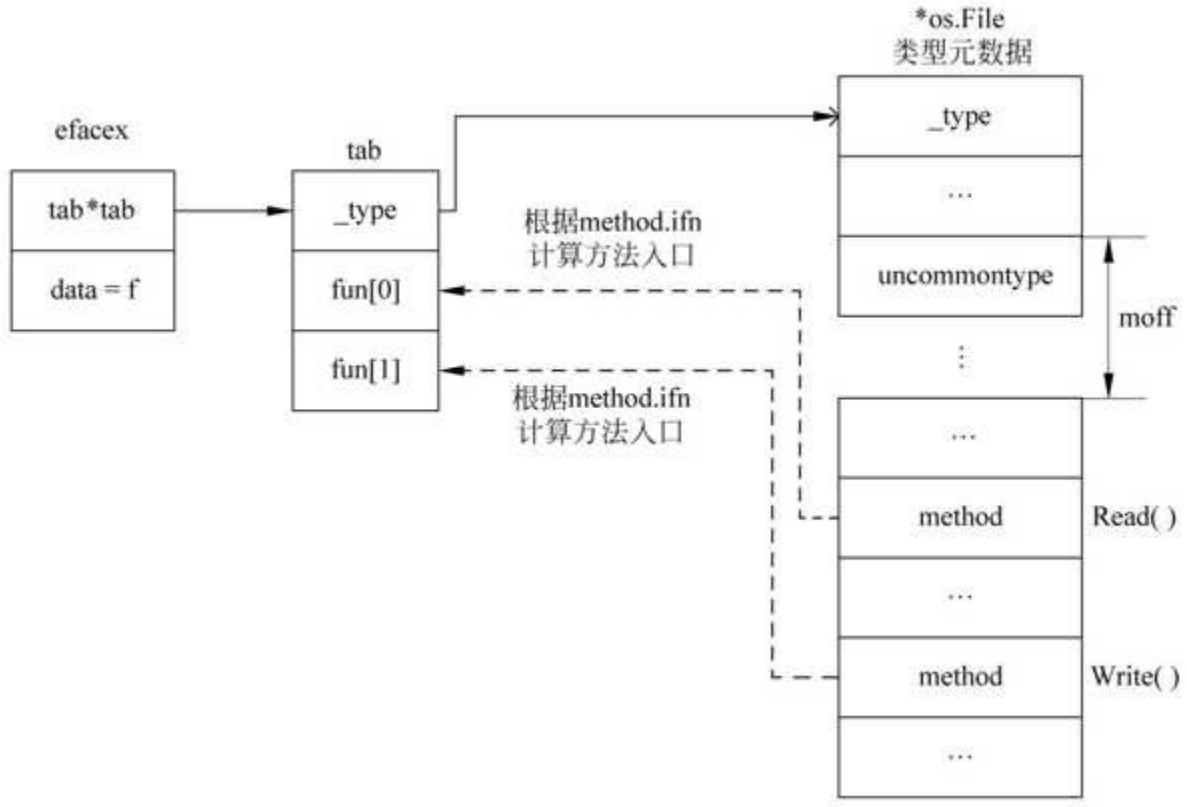

图5-12 基于efacex设计的非空接口变量rw赋值后的数据结构

5.2.2 具体实现

5.2.1节中为了加入地址数组fun，把原本用于interface{}的eface结构扩展成了efacex，实际上在Go语言的runtime中与非空接口对应的结构类型是iface，代码如下：

```
type iface struct {
    tab  * itab
    data unsafe.Pointer
}
```

因为也是通过数据指针data来装载数据的，所以也会有逃逸和装箱发生。其中的itab结构就包含了具体类型的元数据地址_type，以及等价于虚函数表的方法地址数组fun，除此之外还包含了接口本身的类型元数据地址inter，代码如下：

```
type itab struct {
    inter  * interfacetype
    _type  * _type
    hash   uint32
    _      [4]byte
    fun    [1]uintptr
}
```

根据5.1节对类型元数据的简单介绍，从_type到uncommontype，再到[mcount]method，已经找到了自定义类型的方法集。下面再来看一下运行时动态生成itab的相关逻辑。

1．接口类型元数据

首先看一下接口类型的元数据信息对应的数据结构，代码如下：

```
type interfacetype struct {
    typ     _type
    pkgpath name
    mhdr    []imethod
}
```

除去最基本的typ字段，pkgpath表示接口类型被定义在哪个包中，mhdr是接口声明的方法列表。imethod结构的代码如下：

```
type imethod struct {
    name nameOff
    ityp typeOff
}
```

比自定义类型的method结构少了方法地址，只包含方法名和类型元数据的偏移。这些偏移的实际类型为int32，与指针的作用一样，但是64位平台上比使用指针节省一半空间。以ityp为起点，可以找到方法的参数（包括返回值）列表，以及每个参数的类型信息，也就是说这个ityp是方法的原型信息。

第5章/code_5_9.go中非空接口类型的变量rw的数据结构如图5-13所示。

2．如何获得**itab**

运行阶段可通过runtime.getitab函数来获得相应的itab，该函数被定义在runtime包中的iface.go文件中，函数原型的代码如下：

```
func getitab(inter *interfacetype, typ *_type, canfail bool) *itab
```

前两个参数inter和typ分别是接口类型和具体类型的元数据，canfail表示是否允许失败。如果typ没有实现inter要求的所有方法，则canfail为true时函数返回nil，canfail为false时就会造成panic。对应到具体的语法就是comma ok风格的类型断言和普通的类型断言，代码如下：

```
r, ok := a.(io.Reader) //comma ok
r := a.(io.Reader) //有可能造成 panic
```

上述代码第一行就是comma ok风格的类型断言，如果a没有实现io.Reader接口，则ok为false。第二行就不同了，如果a没有实现io.Reader接口，就会造成panic。

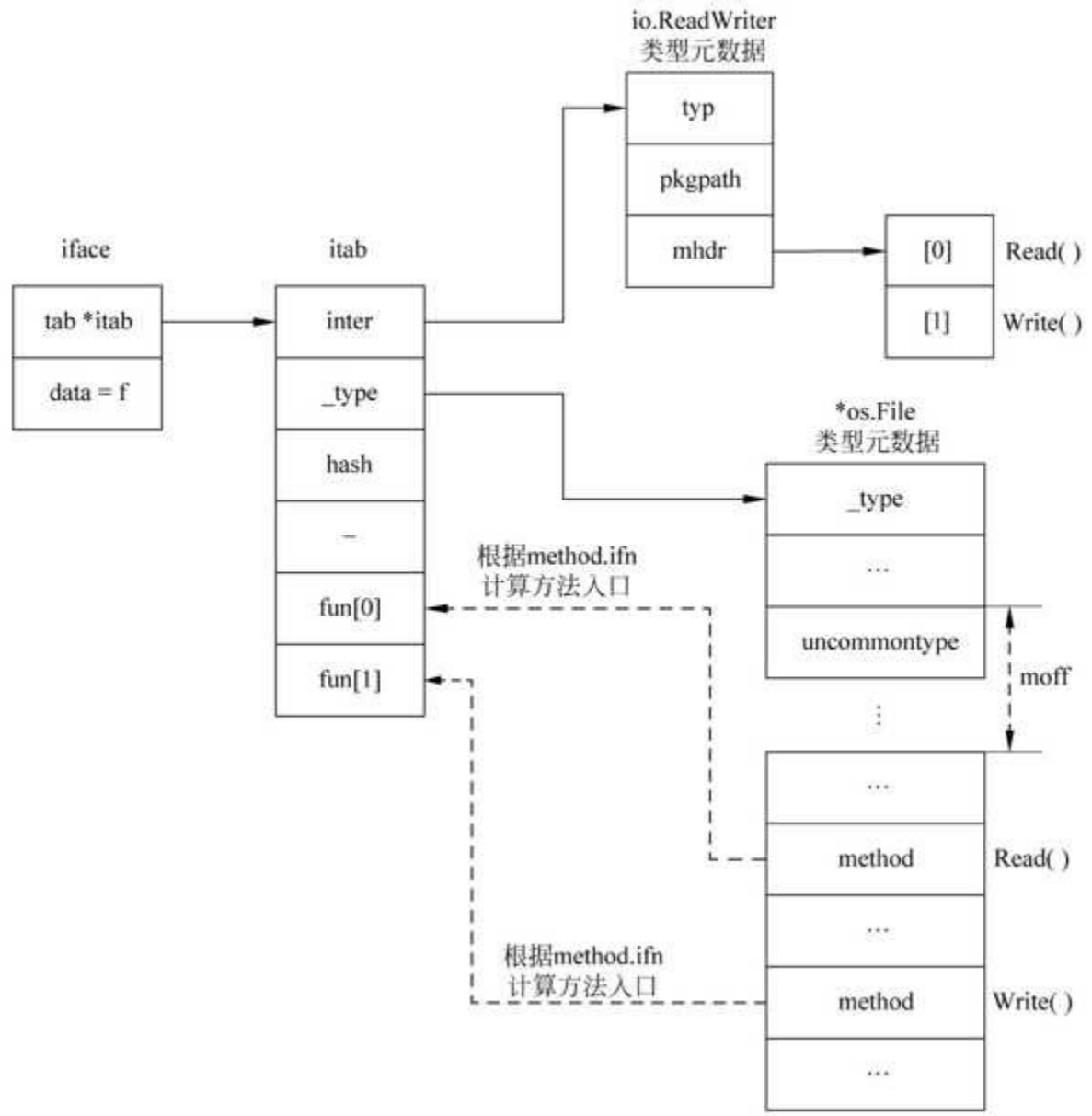

图5-13 io.ReadWriter类型的变量rw的数据结构

getitab（）函数的代码摘抄自Go语言runtime源码，代码如下：

```
func getitab(inter *interfacetype, typ *_type, canfail bool) *itab {
    if len(inter.mhdr) == 0 {
        throw("internal error - misuse of itab")
    }
    if typ.tflag&tflagUncommon == 0 {
        if canfail {
            return nil
        }
        name := inter.typ.nameOff(inter.mhdr[0].name)
        panic(&TypeAssertionError{nil, typ, &inter.typ, name.name()})
    }
    var m *itab

    t := (*itabTableType)(atomic.Loadp(unsafe.Pointer(&itabTable)))
    if m = t.find(inter, typ); m != nil {
        goto finish
    }
    lock(&itabLock)
    if m = itabTable.find(inter, typ); m != nil {
        unlock(&itabLock)
        goto finish
    }
    m = (*itab)(persistentalloc(unsafe.Sizeof(itab{}) + uintptr(len(inter.mhdr) - 1) *
sys.PtrSize, 0, &memstats.other_sys))
    m.inter = inter
    m._type = typ
    m.hash = 0
    m.init()
    itabAdd(m)
    unlock(&itabLock)
finish:
    if m.fun[0] != 0 {
        return m
    }
    if canfail {
        return nil
    }
    panic(&TypeAssertionError{concrete: typ, asserted: &inter.typ, missingMethod: m.init
()})
}
```

函数的主要逻辑如下：①校验inter的方法列表长度不为0，为没有方法的接口生成itab是没有意义的。②通过typ.tflag标志位来校验typ为自定义类型，因为只有自定义类型才能有方法集。③在不加锁的前提下，以inter和typ作为key查找itab缓存itabTable，找到后就跳转到⑤。④加锁后再次查找缓存，如果没有就通过persistentalloc（）函数进行持久化分配，然后初始化itab并调用itabAdd添加到缓存中，最后解锁。⑤通过itab的fun[0]是否为0来判断typ是否实现了inter接口，如果没实现，则根据canfail决定是否造成panic，若实现了，则返回itab地址。

判断itab.fun[0]是否为零，也就是判断第一个方法的地址是否有效，因为Go语言会把无效的itab也缓存起来，主要是为了避免缓存穿透。基于一个确定的接口类型和一个确定的具体类型，就能够唯一确定一个itab，如图5-14所示。按照一般的思路，只有具体类型实现了该接口，才能得到一个itab，进而缓存起来。这样会有个问题，假如具体类型没有实现该接口，但是运行阶段有大量这样的类型断言，缓存中查不到对应的itab，就会每次都查询元数据的方法列表，从而显著影响性能，所以Go语言会把有效、无效的itab都缓存起来，通过fun[0]加以区分。

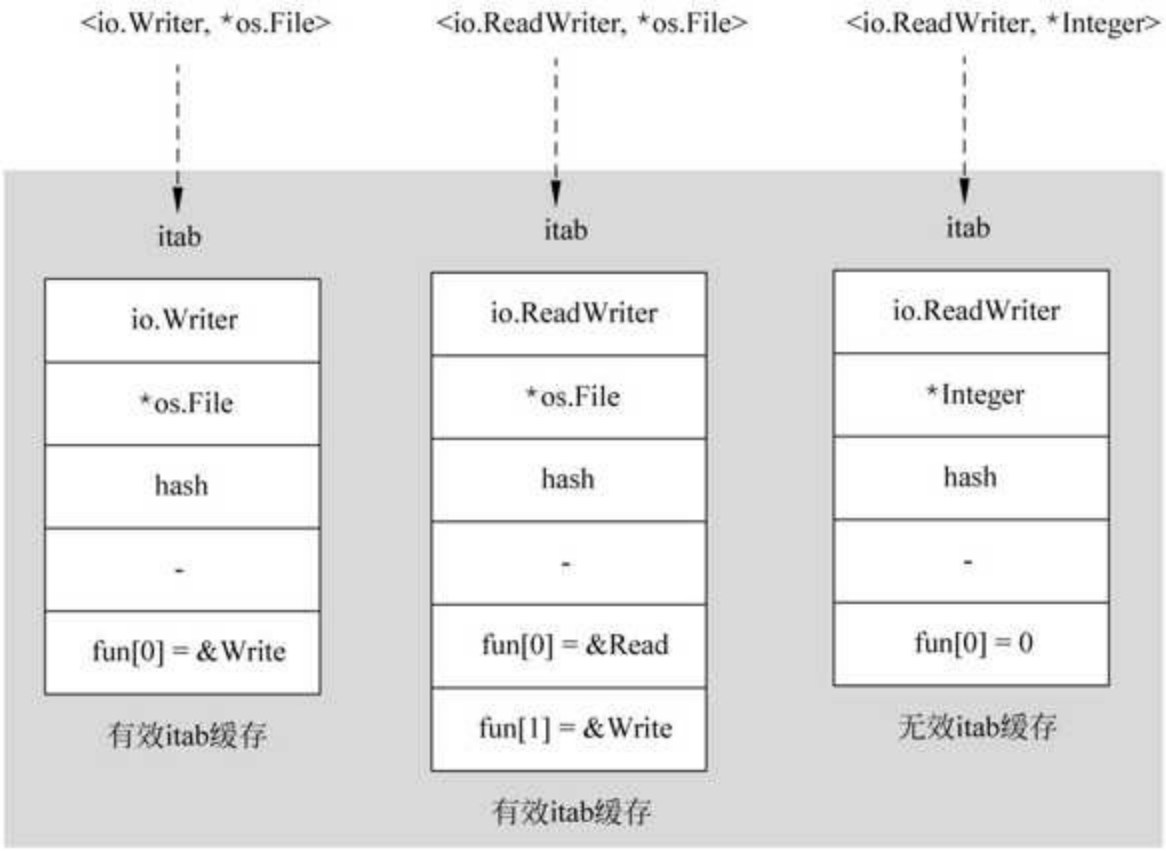

图5-14 interfacetype和_type与itab的对应关系

3．itab缓存

itabTable就是runtime中itab的全局缓存，它本身是个itabTableType类型的指针，itabTableType的代码如下：

```
type itabTableType struct {
    size    uintptr
    count   uintptr
    entries [itabInitSize] * itab
}
```

其中entries是实际的缓存空间，size字段表示缓存的容量，也就是entries数组的大小，count表示实际已经缓存了多少个itab。entries的初始大小是通过itabInitSize指定的，这个常量的值为512。当缓存存满以后，runtime会重新分配整个struct，entries数组是itabTableType的最后一个字段，可以无限增大它的下标来使用超出容量大小的内存，只要在struct之后分配足够的空间就够了，这也是C语言里常用的手法。

itabTableType被实现成一个散列表，如图5-15所示。查找和插入操作使用的key是由接口类型元数据与动态类型元数据组合而成的，哈希值计算方式为接口类型元数据哈希值inter.typ.hash与动态类型元数据哈希值typ.hash进行异或运算。

方法find（）和add（）分别负责实现itabTableType的查找和插入操作，方法add（）操作内部不会扩容存储空间，重新分配操作是在外层实现的，因此对于find（）方法而言，已经插入的内容不会再被修改，所以查找时不需要加锁。方法add（）操作需要在加锁的前提下进行，getitab（）函数是通过调用itabAdd（）函数来完成添加缓存的，itabAdd（）函数内部会按需对缓存进行扩容，然后调用add（）方法。因为缓存扩容需要重新分配itabTableType结构，为了并发安全，使用原子操

作更新itabTable指针。加锁后立刻再次查询也是出于并发的考虑，避免其他协程已经将同样的itab添加至缓存。

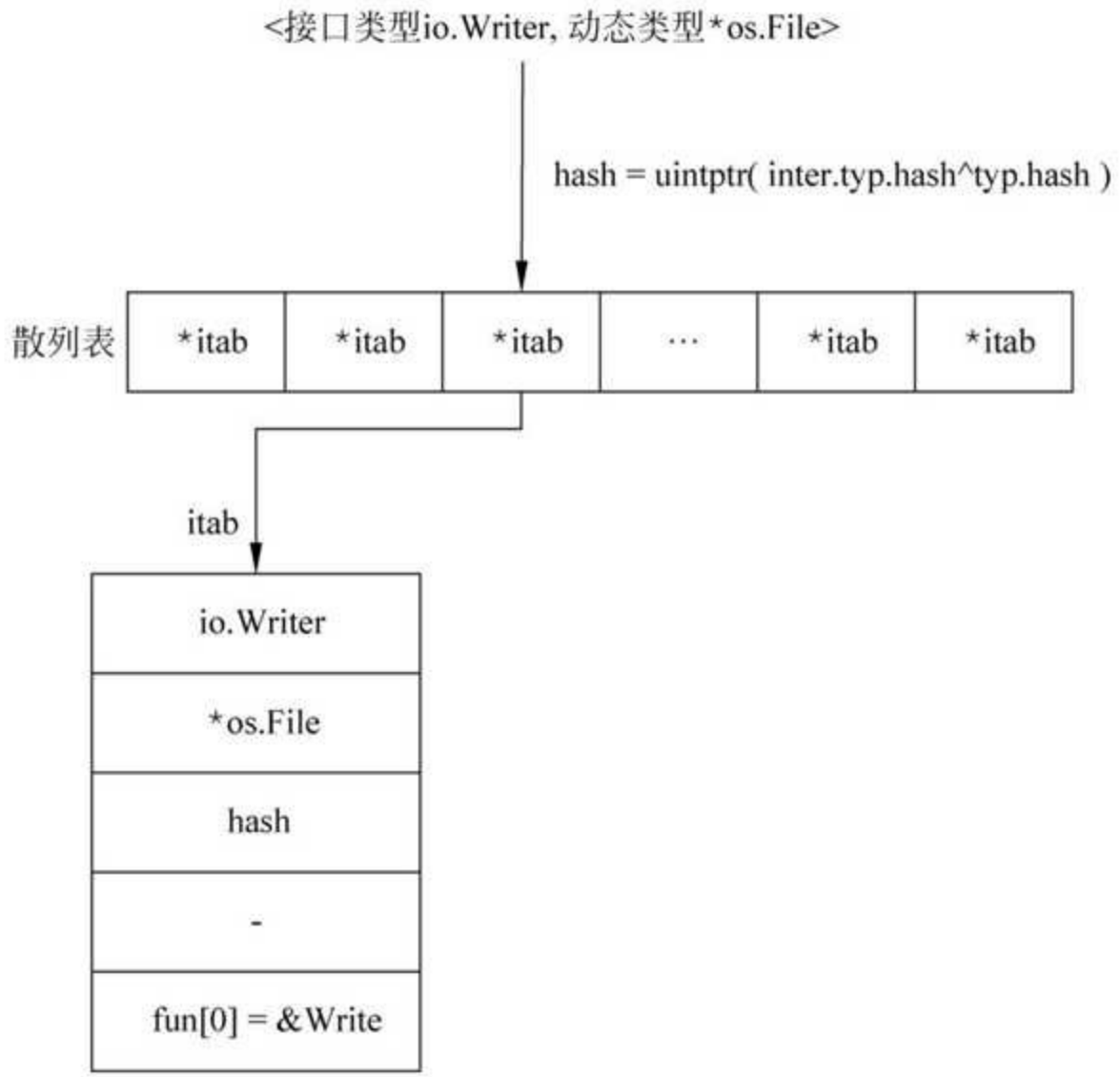

图5-15 itabTableType哈希表

通过persistentalloc（）函数分配的内存不会被回收，分配的大小为itab结构的大小加上接口方法数减一个指针的大小，因为itab中的fun数组声明的长度为1，已经包含了一个指针，分配空间时只需补齐剩下的即可。

还有一个值得一提，就是itab类型的init方法，这里为了节省篇幅，不再摘抄对应的源码。init（）函数内部就是遍历接口的方法列表和具体类型的方法集，来寻找匹配的方法的地址。虽然遍历操作使用了两层嵌套循环，但是方法列表和方法集都是有序的，两层循环实际上都只需执行一次。匹配方法时还会考虑方法是否导出，以及接口和具体类型所在的包。如果是导出的方法则直接匹配成功，如果方法未导出，则接口和具体类型需要定义在同一个包中，方可匹配成功。最后需要再次强调的是，对于匹配成功的方法，地址取的是method结构中的ifn字段，具体的细节在5.2.3节中会继续分析，关于方法集的探索就先到这里。

5.2.3 接收者类型

5.2.1节和5.2.2节中都提到了具体类型方法元数据中的ifn字段，该字段存储的是专门供接口使用的方法地址。所谓专门供接口使用的方法，实际上就是个接收者类型为指针的方法。还记不记得第4

章中分析OBJ文件时，发现编译器总是会为每个值接收者方法包装一个指针接收者方法？这也就说明，接口是不能直接使用值接收者方法的，这是为什么呢？

5.2.2节已经看过了接口的数据结构iface，它包含一个itab指针和一个data指针，data指针存储的就是数据的地址。对于接口来讲，在调用指针接收者方法时，传递地址是非常方便的，也不用关心数据的具体类型，地址的大小总是一致的。假如通过接口调用值接收者方法，就需要通过接口中的data指针把数据的值复制到栈上，由于编译阶段不能确定接口背后的具体类型，所以编译器不能生成相关的指令来完成复制，进而无法调用值接收者方法。

有些读者可能还记得3.4节讲到的runtime.reflectcall（）函数，它能够在运行阶段动态地复制参数并完成函数调用。如果基于reflectcall（）函数，能不能实现通过接口调用值接收者方法呢？

肯定是可以实现的，接口的itab中有具体类型的元数据，确实能够应用reflectcall（）函数，但是有个明显的问题，那就是性能太差。跟几条用于传参的MOV指令加一条普通的CALL指令相比，reflectcall（）函数的开销太大了，所以Go语言选择为值接收者方法生成包装方法。对于代码中的值接收者方法，类型元数据method结构中的ifn和tfn的值是不一样的，指针接收者方法的ifn和tfn是一样的。

比较有意思的是，从类型元数据来看，T和*T是不同的两种类型。接收者类型为T的所有方法，属于T的方法集。因为编译器自动包装指针接收者方法的关系，*T的方法集包含所有方法，也就是所有接收者类型为T的方法加上所有接收者类型为*T的方法。我们可以用一段代码来实际验证一下二者的关系，代码如下：

```
//第 5 章/code_5_10.go
package main
import (
    "fmt"
    "strconv"
    "unsafe"
)
type Integer int
func (i Integer) Value() float64 {
    return float64(i)
}
func (i Integer) String() string {
    return strconv.Itoa(int(i))
}

type Number interface {
    Value() float64
    String() string
}
```

```
func main() {
    i := Integer(0)
    fmt.Println(Methods(i))
    fmt.Println(Methods(&i))
    var n Number = i
    p := (*[5]unsafe.Pointer)((*face)(unsafe.Pointer(&n)).t)
    fmt.Println((*p)[3], (*p)[4])
}

func Methods(a interface{}) (r []Method) {
    e := (*face)(unsafe.Pointer(&a))
    u := uncommon(e.t)
    if u == nil {
        return nil
    }
    s := methods(u)
    r = make([]Method, len(s))
    for i := range s {
        r[i].Name = name(nameOff(e.t, s[i].name))
        r[i].Type = String(typeOff(e.t, s[i].mtyp))
        r[i].IFn = textOff(e.t, s[i].ifn)
        r[i].TFn = textOff(e.t, s[i].tfn)
    }
    return
}

type Method struct {
    Name string
    Type string
    IFn unsafe.Pointer
    TFn unsafe.Pointer
}

type face struct {
    t unsafe.Pointer
    d unsafe.Pointer
}

//go:linkname uncommon reflect.(*rtype).uncommon
func uncommon(t unsafe.Pointer) unsafe.Pointer

//go:linkname methods reflect.(*uncommonType).methods
func methods(u unsafe.Pointer) []method
```

```
type method struct {
    name int32
    mtyp int32
    ifn int32
    tfn int32
}

//go:linkname nameOff reflect.(*rtype).nameOff
func nameOff(t unsafe.Pointer, off int32) unsafe.Pointer

//go:linkname typeOff reflect.(*rtype).typeOff
func typeOff(t unsafe.Pointer, off int32) unsafe.Pointer

//go:linkname textOff reflect.(*rtype).textOff
func textOff(t unsafe.Pointer, off int32) unsafe.Pointer

//go:linkname String reflect.(*rtype).String
func String(t unsafe.Pointer) string

//go:linkname name reflect.name.name
func name(n unsafe.Pointer) string
```

其中Number接口声明了两个方法，即Value（）方法和String（）方法。自定义Integer实现了这两种方法，并且接收者类型都是值类型。为了直接解析类型元数据以获得ifn和tfn的值，示例中使用了linkname机制来调用reflect包中的私有函数，还用了unsafe包访问内存。在amd64平台上，用Go 1.15可以成功编译上述代码，运行结果如下：

```
$ ./code_5_10.exe
[{String func() string 0xcf5fe0 0xcf59a0} {Value func() float64 0xcf6080 0xcf5980}]
                                        //第 1 行输出
[{String func() string 0xcf5fe0 0xcf5fe0} {Value func() float64 0xcf6080 0xcf6080}]
                                        //第 2 行输出
0xcf5fe0 0xcf6080                       //第 3 行输出
```

第1行输出打印出了Integer类型的方法集，String（）和Value（）这两个方法各自的IFn和TFn都不相等，这是因为IFn指向接收者为指针类型的方法代码，而TFn指向接收者为值类型的方法代码。

第2行输出打印出了*Integer类型的方法集，这两个方法各自的IFn和TFn是相等的，都与第1条指令中同名方法的IFn的值相等。

第3行输出打印出了Number接口itab中fun数组中的两个方法地址，与第1行输出Integer方法集中对应方法的IFn的值一致。Integer和*Integer类型方法集的关系如图5-16所示。

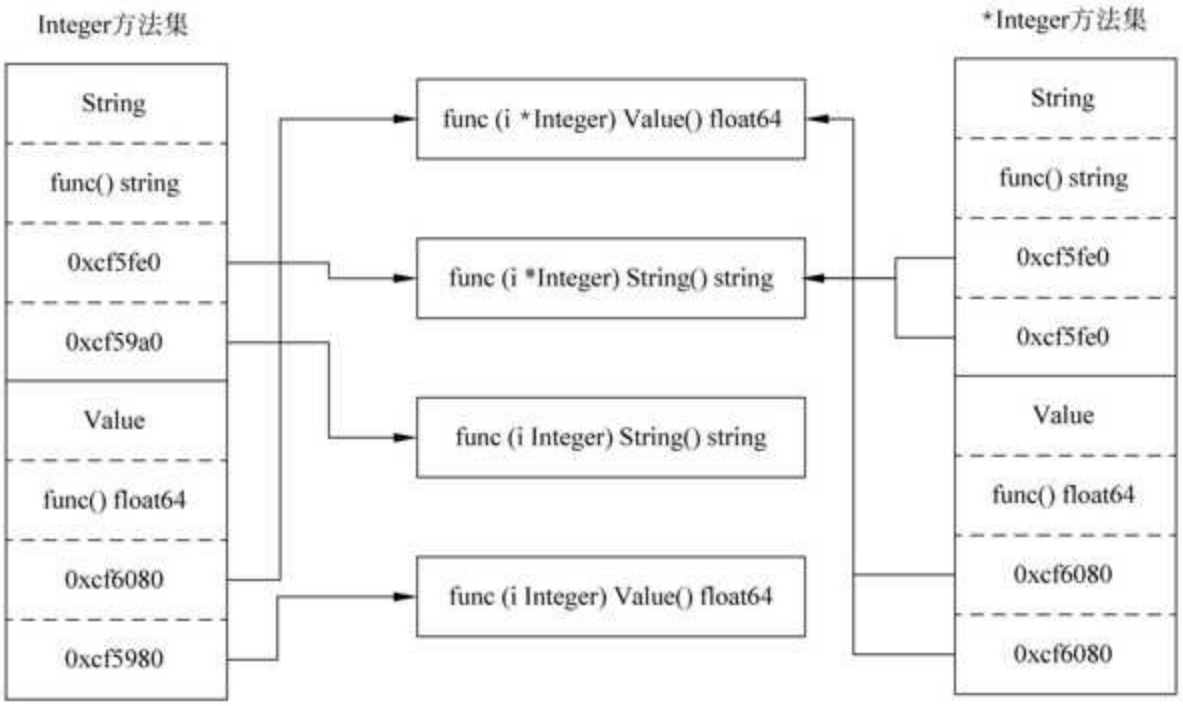

图5-16 Integer和*Integer类型的方法集

有一点需要格外注意，虽然把i赋值给Number类型的接口n后，n的itab最终使用的是这一对接收者为指针类型的方法，但这是通过查找Integer的方法集查到的，语义角度还是Integer类型实现了Number接口。如果把&i赋值给n，编译器和runtime才会从*Integer的方法集中查找。因为编译器会为代码中所有的值接收者方法包装生成对应的指针接收者方法，所以*Integer的方法集是Integer方法集的超集，也就是Integer类型实现的所有接口，即*Integer类型都实现了。反之不然，从语义角度无法为指针接收者方法包装生成对应的值接收者方法，因为原始的数据地址在值接收者方法中已经丢失。

通过以上示例，还能够证明一点，Integer和*Integer的方法集及Number接口的itab中的方法都是按名称升序排列的，与代码中声明和实现的顺序无关，这和5.1.2节讲方法集时从runtime源码中看到的逻辑是一致的。

5.2.4 组合式继承

在第4章讲解方法的时候，曾经探索过基于嵌入的组合式继承，当时发现编译器会对继承的方法进行包装。因为自定义类型继承来的方法会影响到实现了哪些接口，所以本节再来回顾一下组合式继承，从方法集的角度进行分析，示例代码如下：

```
//第 5 章/code_5_11.go
package inherit

type A int

func (a A) Value() int {
    return int(a)
}

func (a *A) Set(n int) {
    *a = A(n)
}

type B struct {
    A
    b int
}

type C struct {
    *A
    c int
}
```

类型A有一个值接收者方法Value（）和一个指针接收者方法Set（），将A以值嵌入的方式嵌入类型B中，以地址嵌入的方式嵌入类型C中，然后看一下B、C、*B和*C会继承哪些方法。先用go tool compile命令把上述源码编译为OBJ文件，然后就可以通过go tool nm工具确认了，命令如下：

```
$ go tool compile -p inherit -trimpath="`pwd`=>" gom.go
$ go tool nm gom.o | grep 'T '
    31ea T inherit.(*A).Set
    3b3f T inherit.(*A).Value
    3b95 T inherit.(*B).Set
    3ba6 T inherit.(*B).Value
    3c7a T inherit.(*C).Set
    3c8c T inherit.(*C).Value
    31df T inherit.A.Value
    3c10 T inherit.B.Value
    3cfd T inherit.C.Set
    3d67 T inherit.C.Value
```

通过这个列表就能知道各个自定义类型的方法集中有哪些方法，将以上结果整理为更加直观的表格形式，如表5-2所示，还是要注意T和*T是不同的类型。

值接收者方法始终能够被继承，但只有在能够获得嵌入对象的地址的情况下才能继承指针接收者方法，所以无论是值嵌入还是地址嵌入，*B和*C都能继承Set（）方法。由于嵌入地址的关系，C也能够继承Set（）方法。

如果一个接口要求实现Value（）和Set（）这两个方法，则上述几种自定义类型中*A、*B、*C和C都实现了该接口。A和B没有实现Set（）方法，也就是说A和B的方法集中没有Set（）方法。通过接口调用C的方法时，虽然实际上调用的是*C的方法，但语义层面还是C实现了这两个方法，只不过接口机制需要指针接收者。

表5-2 示例程序中各自定义类型包含的方法的情况

自定义类型	有 Value()方法	有 Set()方法
A	√	
*A	√	√
B	√	
*B	√	√
C	√	√
*C	√	√

所以回过头再来看，Go语言不允许为T和*T定义同名方法，实际上并不是因为不支持函数重载，前面已经看到了A.Value（）方法和（*A）.Value（）方法是可以区分的。其根本原因就是编译器要为值接收者方法生成指针接收者包装方法，要保证两者的逻辑一致，所以不允许用户同时实现，用户可能会实现成不同的逻辑。

5.3 类型断言

所谓类型断言，就是运行阶段根据元数据信息，来判断数据是否属于某种具体类型，或者是否实现了某个接口。既然要用到类型元数据，那么源操作数就必须是interface{}或某个接口类型的变量，也就是说底层是runtime.eface或runtime.iface类型。

类型断言在语法上有两种不同的形式，第一种就是直接断言为目标类型，这也是最正常的写法，示例代码如下：

```
dest := source.(dest_type)
```

这种形式存在一定风险，如果断言失败，就会造成panic。第二种形式比较安全，这种形式的代码常被称为comma ok风格，因为有个额外的bool变量来表明操作是否成功，人们习惯把这个bool变量命名为ok。这种形式的断言无论成败都不会造成panic，示例代码如下：

```
dest, ok := source.(dest_type)
```

本节就根据源操作数和目标类型的不同，把类型断言分成4种情况，结合反编译和runtime源码分析，分别探索几种情况的实现原理。

5.3.1 E To具体类型

E指的是runtime.eface，也就是interface{}类型，而具体类型是相对于抽象类型来讲的，抽象类型指的是接口，接口通过方法列表对行为进行抽象，所以具体类型指的是除接口以外的内置类型和自定义类型。E To具体类型的断言就是从容器中把数据取出来。

先来看一看第一种形式，也就是从interface{}直接断言为某个具体类型，下面把这部分逻辑放到一个单独的函数中，以便于后续分析，代码如下：

```
func normal(a interface{}) int {
    return a.(int)
}
```

用go tool compile命令编译包含上述函数的源码文件e2t.go，会得到一个OBJ文件e2t.o，再用go tool objdump命令反编译该文件中的normal（）函数，得到的汇编代码如下：

```
$ go tool compile -p gom -trimpath="`pwd`=>" e2t.go
$ go tool objdump -S -s '^gom.normal$' e2t.o
TEXT gom.normal(SB) gofile..e2t.go
func normal(a interface{}) int {
  0x7d2       65488b0c2528000000      MOVQ GS:0x28, CX
  0x7db       488b8900000000          MOVQ 0(CX), CX      [3:7]R_TLS_LE
  0x7e2       483b6110                CMPQ 0x10(CX), SP
  0x7e6       7651                    JBE 0x839
  0x7e8       4883ec20                SUBQ $0x20, SP
  0x7ec       48896c2418              MOVQ BP, 0x18(SP)
  0x7f1       488d6c2418              LEAQ 0x18(SP), BP
        return a.(int)
  0x7f6       488d0500000000          LEAQ 0(IP), AX      [3:7]R_PCREL:type.int
  0x7fd       488b4c2428              MOVQ 0x28(SP), CX
  0x802       4839c8                  CMPQ CX, AX
  0x805       7517                    JNE 0x81e
  0x807       488b442430              MOVQ 0x30(SP), AX
  0x80c       488b00                  MOVQ 0(AX), AX
  0x80f       4889442438              MOVQ AX, 0x38(SP)
  0x814       488b6c2418              MOVQ 0x18(SP), BP
  0x819       4883c420                ADDQ $0x20, SP
  0x81d       c3                      RET
  0x81e       48890c24                MOVQ CX, 0(SP)
  0x822       4889442408              MOVQ AX, 0x8(SP)
  0x827       488d0500000000          LEAQ 0(IP), AX      [3:7]R_PCREL:type.interface {}
  0x82e       4889442410              MOVQ AX, 0x10(SP)
  0x833       e800000000              CALL 0x838          [1:5]R_CALL:runtime.panicdottypeE
  0x838       90                      NOPL
func normal(a interface{}) int {
  0x839       e800000000              CALL 0x83e          [1:5]R_CALL:runtime.morestack_noctxt
  0x83e       eb92                    JMP gom.normal(SB)
```

汇编代码还是不太直观，转换成等价的伪代码如下：

```
func normal(a runtime.eface) int {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    if a._type != &type.int {
        runtime.panicdottypeE(a._type, &type.int, &type.interface{})
    }
    return *(*int)(a.data)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

编译器插入与栈增长相关的代码已经屡见不鲜了，真正与类型断言相关的代码只有4行。逻辑也很简单，就是判断a._type与int类型的元数据地址是否相等，如果不相等就调用panicdottypeE（）函数，如果相等就把a.data作为*int来提取int数值。

再来看一下comma ok风格的断言，代码如下：

```
func commaOk(a interface{}) (n int, ok bool) {
    n, ok = a.(int)
    return
}
```

用相同的命令进行编译和反编译，得到的汇编代码如下：

```
$ go tool compile -p gom -trimpath="`pwd`=>" e2t2.go
$ go tool objdump -S -s '^gom.commaOk$' e2t2.o
TEXT gom.commaOk(SB) gofile..e2t2.go
        n, ok = a.(int)
  0x810       488d0500000000    LEAQ 0(IP), AX          [3:7]R_PCREL:type.int
  0x817       488b4c2408        MOVQ 0x8(SP), CX
  0x81c       4839c8            CMPQ CX, AX
  0x81f       7515              JNE 0x836
  0x821       488b442410        MOVQ 0x10(SP), AX
  0x826       488b00            MOVQ 0(AX), AX
        return
  0x829       4889442418        MOVQ AX, 0x18(SP)
        n, ok = a.(int)
  0x82e       0f94c0            SETE AL
        return
  0x831       88442420          MOVB AL, 0x20(SP)
  0x835       c3                RET
  0x836       b800000000            MOVL $0x0, AX
        n, ok = a.(int)
  0x83b       ebec                  JMP 0x829
```

因为函数栈帧足够小，并且没有调用任何外部函数，所以编译器无须插入栈增长代码。转换成等价的伪代码也比较精简，代码如下：

```
func commaOk(a runtime.eface) (n int, ok bool) {
    if a._type != &type.int {
        return 0, false
    }
    return *(*int)(a.data), true
}
```

核心逻辑还是判断a._type与int类型的元数据地址是否相等，如果不相等就返回int类型零值和false，如果相等就把a.data作为*int来提取int数值，然后和true一起返回。

综上所述，从interface{}到具体类型的断言如图5-17所示，基本上就是一个指针比较操作加上一个具体类型相关的复制操作，执行时应该还是很高效的。

图5-17 从interface{}到具体类型的断言

5.3.2 E To I

E指的还是runtime.eface，I指的则是runtime.iface，E To I也就是从interface{}到某个自定义接口类型的断言。断言的目标为接口类型，E背后的具体类型需要实现接口要求的所有方法，所以涉及具体类型的方法集遍历、动态分配itab等操作。5.2.2节已经分析过runtime中用来完成此工作的getitab（）函数，本节继续探索类型断言是如何使用该函数的。

还是先按照一般的代码风格实现一个包含断言逻辑的函数，代码如下：

```
func normal(a interface{}) io.ReadWriter {
    return a.(io.ReadWriter)
}
```

用同样的命令进行编译和反编译，得到的汇编代码如下：

```
$ go tool compile -p gom -trimpath="`pwd`=>" e2i.go
$ go tool objdump -S -s '^gom.normal$' e2i.o
TEXT gom.normal(SB) gofile..e2i.go
func normal(a interface{}) io.ReadWriter {
  0x8b5       65488b0c2528000000    MOVQ GS:0x28, CX
  0x8be       488b8900000000        MOVQ 0(CX), CX  [3:7]R_TLS_LE
  0x8c5       483b6110              CMPQ 0x10(CX), SP
  0x8c9       7650                  JBE 0x91b
  0x8cb       4883ec30              SUBQ $0x30, SP
  0x8cf       48896c2428            MOVQ BP, 0x28(SP)
  0x8d4       488d6c2428            LEAQ 0x28(SP), BP
        return a.(io.ReadWriter)
  0x8d9       488d0500000000        LEAQ 0(IP), AX  [3:7]R_PCREL:type.io.ReadWriter
  0x8e0       48890424              MOVQ AX, 0(SP)
  0x8e4       488b442438            MOVQ 0x38(SP), AX
  0x8e9       4889442408            MOVQ AX, 0x8(SP)
  0x8ee       488b442440            MOVQ 0x40(SP), AX
  0x8f3       4889442410            MOVQ AX, 0x10(SP)
  0x8f8       e800000000            CALL 0x8fd    [1:5]R_CALL:runtime.assertE2I
  0x8fd       488b442418            MOVQ 0x18(SP), AX
  0x902       488b4c2420            MOVQ 0x20(SP), CX
  0x907       4889442448            MOVQ AX, 0x48(SP)
  0x90c       48894c2450            MOVQ CX, 0x50(SP)
  0x911       488b6c2428            MOVQ 0x28(SP), BP
  0x916       4883c430              ADDQ $0x30, SP
  0x91a       c3                    RET
func normal(a interface{}) io.ReadWriter {
  0x91b       e800000000            CALL 0x920    [1:5]R_CALL:runtime.morestack_noctxt
  0x920       eb93                  JMP gom.normal(SB)
```

在保证逻辑不变的前提下，写出等价的Go风格伪代码如下：

```
func normal(a runtime.eface) io.ReadWriter {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    return runtime.assertE2I(&type.io.ReadWriter, a)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

除去编译器插入的栈增长代码，核心逻辑就是调用了runtime.assertE2I（）函数，摘抄runtime包中的函数代码如下：

```
func assertE2I(inter *interfacetype, e eface) (r iface) {
    t := e._type
    if t == nil {
        panic(&TypeAssertionError{nil, nil, &inter.typ, ""})
    }
    r.tab = getitab(inter, t, false)
    r.data = e.data
    return
}
```

函数先校验了E的具体类型元数据指针不可为空，没有具体类型的元数据是无法进行断言的，然后通过调用getitab（）函数来得到对应的itab，data字段直接复制。注意调用getitab（）函数时最后一个参数为false，根据之前的源码分析已知这个参数是canfail。canfail为false时，如果t没有实现inter要求的所有方法，getitab（）函数就会造成panic。

接下来再看一下comma ok风格的断言，代码如下：

```
func commaOk(a interface{}) (i io.ReadWriter, ok bool) {
    i, ok = a.(io.ReadWriter)
    return
}
```

编译再反编译之后，得到的汇编代码如下：

```
$ go tool compile -p gom -trimpath="`pwd`=>" e2i2.go
$ go tool objdump -S -s '^gom.commaOk$' e2i2.o
TEXT gom.commaOk(SB) gofile..e2i2.go
func commaOk(a interface{}) (i io.ReadWriter, ok bool) {
  0x979       65488b0c2528000000    MOVQ GS:0x28, CX
  0x982       488b8900000000        MOVQ 0(CX), CX    [3:7]R_TLS_LE
  0x989       483b6110              CMPQ 0x10(CX), SP
  0x98d       7659                  JBE 0x9e8
  0x98f       4883ec38              SUBQ $0x38, SP
  0x993       48896c2430            MOVQ BP, 0x30(SP)
  0x998       488d6c2430            LEAQ 0x30(SP), BP
        i, ok = a.(io.ReadWriter)
  0x99d       488d0500000000        LEAQ 0(IP), AX    [3:7]R_PCREL:type.io.ReadWriter
  0x9a4       48890424              MOVQ AX, 0(SP)
  0x9a8       488b442440            MOVQ 0x40(SP), AX
  0x9ad       4889442408            MOVQ AX, 0x8(SP)
  0x9b2       488b442448            MOVQ 0x48(SP), AX
  0x9b7       4889442410            MOVQ AX, 0x10(SP)
  0x9bc       e800000000            CALL 0x9c1    [1:5]R_CALL:runtime.assertE2I2
  0x9c1       488b442418              MOVQ 0x18(SP), AX
  0x9c6       488b4c2420              MOVQ 0x20(SP), CX
  0x9cb       0fb6542428              MOVZX 0x28(SP), DX
        return
  0x9d0       4889442450              MOVQ AX, 0x50(SP)
  0x9d5       48894c2458              MOVQ CX, 0x58(SP)
  0x9da       88542460                MOVB DL, 0x60(SP)
  0x9de       488b6c2430              MOVQ 0x30(SP), BP
  0x9e3       4883c438                ADDQ $0x38, SP
  0x9e7       c3                      RET
func commaOk(a interface{}) (i io.ReadWriter, ok bool) {
  0x9e8       e800000000              CALL 0x9ed    [1:5]R_CALL:runtime.morestack_noctxt
  0x9ed       eb8a                    JMP gom.commaOk(SB)
```

写成等价的Go风格伪代码如下：

```
func commaOk(a runtime.eface) (i io.ReadWriter, ok bool) {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    return runtime.assertE2I2(&type.io.ReadWriter, a)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

可以看到这次主要通过runtime.assertE2I2（）函数来完成，从runtime包中找到该函数的源代码如下：

```
func assertE2I2(inter *interfacetype, e eface) (r iface, b bool) {
    t := e._type
    if t == nil {
        return
    }
    tab := getitab(inter, t, true)
    if tab == nil {
        return
    }
    r.tab = tab
    r.data = e.data
    b = true
    return
}
```

与之前不同的是，可以通过第2个返回值来表示操作的成功与否，所以不用再造成panic。如果E的具体类型指针为空，则直接返回false。调用getitab（）函数时也把canfail设置为true，并且需要检测返回的tab是否为nil，以此来判断是否成功。

综上所述，E To I形式的类型断言，主要通过runtime中的assertE2I（）和assertE2I2（）这两个函数实现，底层的主要任务如图5-18所示，都是通过getitab（）函数完成的方法集遍历及itab分配和初始化。因为getitab（）函数中用到了全局的itab缓存，所以性能方面应该也是很高效的。

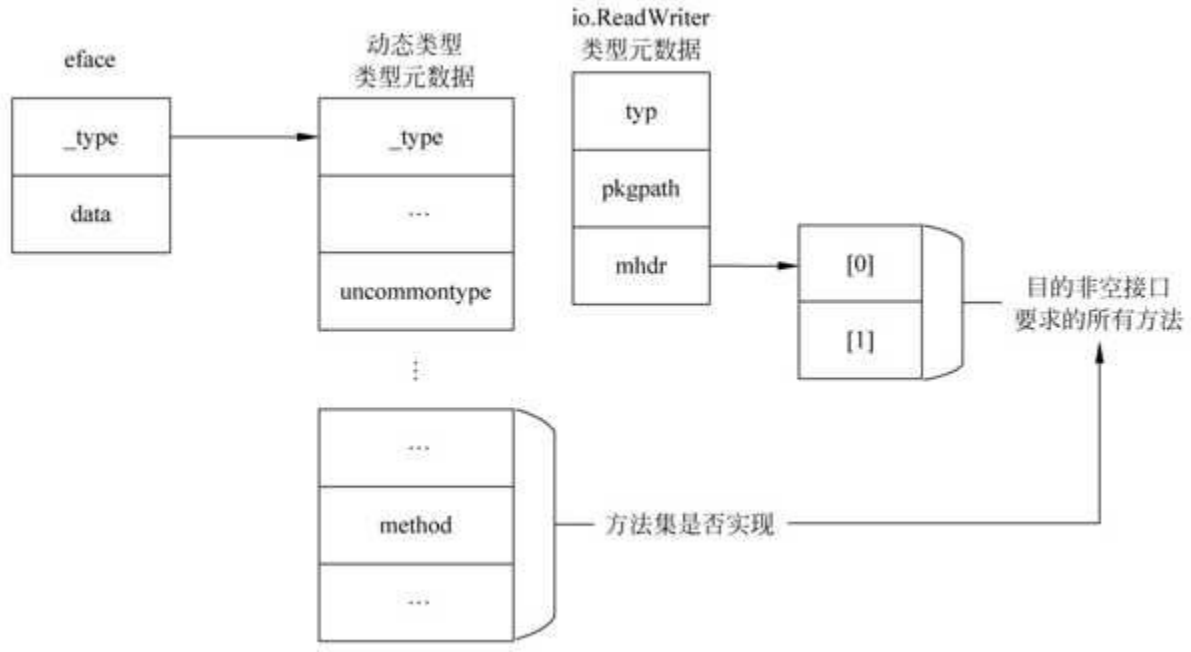

图5-18 从interface{}到非空接口的类型断言

5.3.3 I To具体类型

5.3.1节和5.3.2节主要探索了源类型为interface{}的类型断言，目标分为具体类型和接口类型两种情况。接下来看一下源类型为接口类型的类型断言，本节首先分析目标为具体类型的断言实现，也就是从runtime.iface转换为某种具体类型。

还是先按照一般的写法把类型断言逻辑放到一个单独的函数中，代码如下：

```
func normal(i io.ReadWriter) *os.File {
    return i.(*os.File)
}
```

然后使用go tool命令编译和反编译，得到的汇编代码如下：

```
$ go tool compile -p gom -trimpath="`pwd`=>" i2t.go
$ go tool objdump -S -s '^gom.normal$' i2t.o
TEXT gom.normal(SB) gofile..i2t.go
func normal(i io.ReadWriter) *os.File {
  0x10bd      65488b0c2528000000    MOVQ GS:0x28, CX
  0x10c6      488b8900000000        MOVQ 0(CX), CX    [3:7]R_TLS_LE
  0x10cd      483b6110              CMPQ 0x10(CX), SP
  0x10d1      7655                  JBE 0x1128
  0x10d3      4883ec20              SUBQ $0x20, SP
  0x10d7      48896c2418            MOVQ BP, 0x18(SP)
  0x10dc      488d6c2418            LEAQ 0x18(SP), BP
        return i.(*os.File)
  0x10e1      488d0500000000        LEAQ 0(IP), AX
[3:7]R_PCREL:go.itab.*os.File,io.ReadWriter
  0x10e8      488b4c2428            MOVQ 0x28(SP), CX
  0x10ed      4839c8                CMPQ CX, AX
  0x10f0      7514                  JNE 0x1106
  0x10f2      488b442430            MOVQ 0x30(SP), AX
  0x10f7      4889442438            MOVQ AX, 0x38(SP)
  0x10fc      488b6c2418            MOVQ 0x18(SP), BP
  0x1101      4883c420              ADDQ $0x20, SP
  0x1105      c3                    RET
  0x1106      48890c24              MOVQ CX, 0(SP)
  0x110a      488d0500000000        LEAQ 0(IP), AX  [3:7]R_PCREL:type.*os.File
  0x1111      4889442408            MOVQ AX, 0x8(SP)
  0x1116      488d0500000000        LEAQ 0(IP), AX  [3:7]R_PCREL:type.io.ReadWriter
  0x111d      4889442410            MOVQ AX, 0x10(SP)
  0x1122      e800000000            CALL 0x1127     [1:5]R_CALL:runtime.panicdottypeI
  0x1127      90                    NOPL
func normal(i io.ReadWriter) *os.File {
  0x1128      e800000000            CALL 0x112d     [1:5]R_CALL:runtime.morestack_noctxt
  0x112d      eb8e                  JMP gom.normal(SB)
```

与之前从interface{}断言有些不同，为了更加直观，写出等价的Go风格伪代码如下：

```
func normal(i runtime.iface) *os.File {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    if i.tab != &go.itab.*os.File,io.ReadWriter {
        runtime.panicdottypeI(i.tab, &type.*os.File, &type.io.ReadWriter)
    }
    return (*os.File)(i.data)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

其中的go.itab.*os.File，io.ReadWriter指的就是全局itab缓存中与*os.File和io.ReadWriter这一对类型对应的itab。这个itab是在编译阶段就被编译器生成的，所以代码中可以直接链接到它的地址。这个断言的核心逻辑就是比较iface中tab字段的地址是否与目标itab地址相等。如果不相等就调用panicdottypeI，如果相等就把iface的data字段返回。注意这里因为*os.File是指针类型，所以不涉及自动拆箱，也就没有与具体类型相关的复制操作，如果具体类型为值类型就不然了。

实际反编译之前，笔者曾经以为会比较i.tab._type和&type.*os.File，但是Go语言的实际实现更为直接高效，也省去了对i.tab的非空校验。

再来看一下comma ok风格的断言，代码如下：

```
func commaOk(i io.ReadWriter) (f *os.File, ok bool) {
    f, ok = i.(*os.File)
    return
}
```

先编译成OBJ，再反编译，得到的汇编代码如下：

```
$ go tool compile -p gom -trimpath="`pwd`=>" i2t2.go
$ go tool objdump -S -s '^gom.commaOk$' i2t2.o
TEXT gom.commaOk(SB) gofile..i2t2.go
        f, ok = i.(*os.File)
  0x10a8        488d0500000000      LEAQ 0(IP), AX      [3:7]R_PCREL:go.itab.*os.File,
io.ReadWriter
  0x10af        488b4c2408          MOVQ 0x8(SP), CX
  0x10b4        4839c8              CMPQ CX, AX
  0x10b7        7512                JNE 0x10cb
  0x10b9        488b442410          MOVQ 0x10(SP), AX
        return
  0x10be        4889442418          MOVQ AX, 0x18(SP)
        f, ok = i.(*os.File)
  0x10c3        0f94c0              SETE AL
        return
  0x10c6        88442420            MOVB AL, 0x20(SP)
  0x10ca        c3                  RET
  0x10cb        b800000000          MOVL $0x0, AX
        f, ok = i.(*os.File)
  0x10d0        ebec                JMP 0x10be
```

因为不需要调用panicdottypeI（）函数的关系，所以编译器可以省略掉与栈增长相关的代码。核心逻辑还是比较itab的地址，写出等价的Go风格伪代码如下：

```
func commaOk(i runtime.iface) (f *os.File, ok bool) {
    if i.tab != &go.itab.*os.File,io.ReadWriter {
        return nil, false
    }
    return (*os.File)(i.data), true
}
```

与一般风格的类型断言也没有太大的不同，不同点就是通过返回值为false表示断言失败，代替了调用panicdottypeI（）函数。

综上所述，I To具体类型的断言与E To具体类型的断言在实现上极其相似，核心逻辑如图5-19所示，都是一个指针的相等判断。

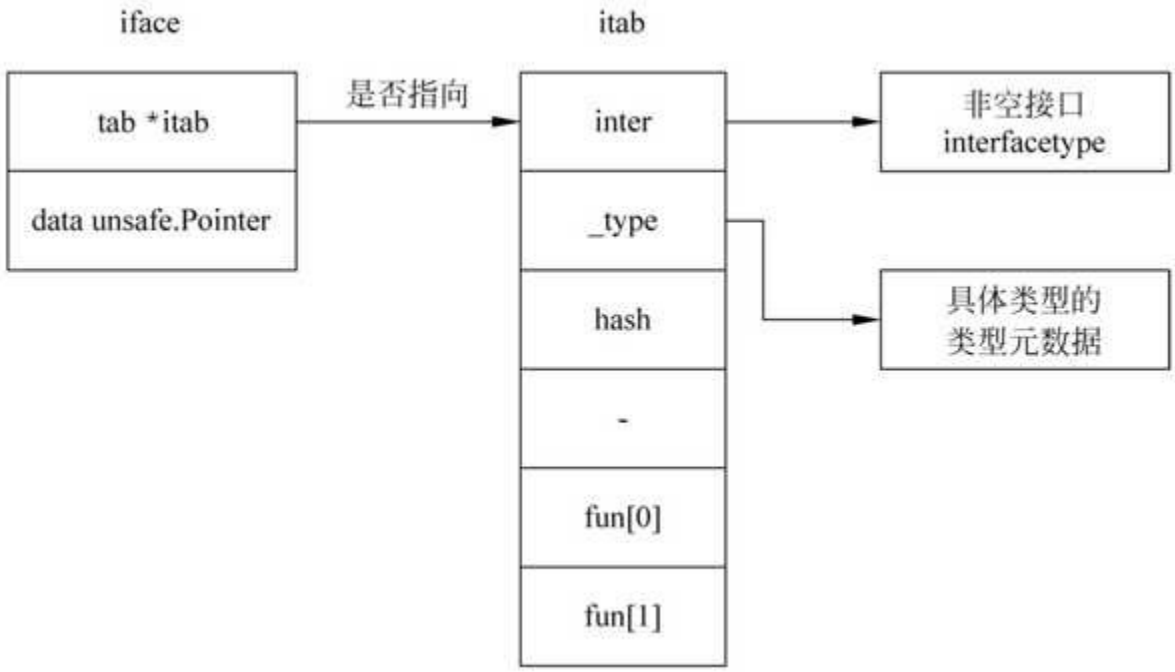

图5-19 从非空接口到具体类型的类型断言

是否涉及自动拆箱，要视具体类型为值类型还是指针类型而定。值类型要进行拆箱操作，也就是从data地址处把值复制出来，指针类型则无须拆箱，直接返回data即可，无论源类型为E或I，其实都是一样的。

5.3.4 I To I

本节探索类型断言的最后一种场景，从一种接口类型到另一种接口类型，因为接口类型对应着runtime.iface，所以简称为I To I。断言的源接口和目标接口应该有着不同的类型，而实际影响断言的就是目标接口有着怎样的方法列表，底层应该还是基于getitab（）函数。

按照一般的类型断言风格，准备一个示例函数，代码如下：

```
func normal(rw io.ReadWriter) io.Reader {
    return rw.(io.Reader)
}
```

还是经过编译和反编译，得到的汇编代码如下：

```
$ go tool compile -p gom -trimpath="`pwd`=>" i2i.go
$ go tool objdump -S -s '^gom.normal$' i2i.o
TEXT gom.normal(SB) gofile..i2i.go
func normal(rw io.ReadWriter) io.Reader {
  0x64c       65488b0c2528000000      MOVQ GS:0x28, CX
  0x655       488b8900000000          MOVQ 0(CX), CX     [3:7]R_TLS_LE
  0x65c       483b6110                CMPQ 0x10(CX), SP
  0x660       7650                    JBE 0x6b2
  0x662       4883ec30                SUBQ $0x30, SP
  0x666       48896c2428              MOVQ BP, 0x28(SP)
  0x66b       488d6c2428              LEAQ 0x28(SP), BP
        return rw.(io.Reader)
  0x670       488d0500000000          LEAQ 0(IP), AX     [3:7]R_PCREL:type.io.Reader
  0x677       48890424                MOVQ AX, 0(SP)
  0x67b       488b442438              MOVQ 0x38(SP), AX
  0x680       4889442408              MOVQ AX, 0x8(SP)
  0x685       488b442440              MOVQ 0x40(SP), AX
  0x68a       4889442410              MOVQ AX, 0x10(SP)
  0x68f       e800000000              CALL 0x694     [1:5]R_CALL:runtime.assertI2I
  0x694       488b442418              MOVQ 0x18(SP), AX
  0x699       488b4c2420              MOVQ 0x20(SP), CX
  0x69e       4889442448              MOVQ AX, 0x48(SP)
  0x6a3       48894c2450              MOVQ CX, 0x50(SP)
  0x6a8       488b6c2428              MOVQ 0x28(SP), BP
  0x6ad       4883c430                ADDQ $0x30, SP
  0x6b1       c3                      RET
func normal(rw io.ReadWriter) io.Reader {
  0x6b2       e800000000              CALL 0x6b7     [1:5]R_CALL:runtime.morestack_noctxt
  0x6b7       eb93                    JMP gom.normal(SB)
```

写出逻辑等价的Go风格伪代码如下：

```
func normal(i runtime.iface) io.Reader {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    return runtime.assertI2I(&type.io.Reader, i)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

实际上就是调用了runtime.assertI2I（）函数，该函数的源代码如下：

```
func assertI2I(inter * interfacetype, i iface) (r iface) {
    tab : = i.tab
    if tab == nil {
        panic(&TypeAssertionError{nil, nil, &inter.typ, ""})
    }
    if tab.inter == inter {
        r.tab = tab
        r.data = i.data
        return
    }
    r.tab = getitab(inter, tab._type, false)
    r.data = i.data
    return
}
```

先校验i.tab不为nil，否则就意味着没有类型元数据，类型断言也就无从谈起，然后检测i.tab.inter是否等于inter，相等就意味着源接口和目标接口类型相同，直接复制就可以了。最后才调用getitab（）函数，根据inter和i.tab._type获取对应的itab。canfail参数为false，所以如果getitab（）函数失败就会造成panic。

再来看一下comma ok风格的断言，准备的函数代码如下：

```
func commaOk(rw io.ReadWriter) (r io.Reader, ok bool) {
    r, ok = rw.(io.Reader)
    return
}
```

将上述代码先编译为OBJ，再进行反编译，得到的汇编代码如下：

```
$ go tool compile -p gom -trimpath="`pwd`=>" i2i2.go
$ go tool objdump -S -s '^gom.commaOk$' i2i2.o
TEXT gom.commaOk(SB) gofile..i2i2.go
func commaOk(rw io.ReadWriter) (r io.Reader, ok bool) {
  0x710       65488b0c2528000000    MOVQ GS:0x28, CX
  0x719       488b8900000000        MOVQ 0(CX), CX    [3:7]R_TLS_LE
  0x720       483b6110              CMPQ 0x10(CX), SP
  0x724       7659                  JBE 0x77f
  0x726       4883ec38              SUBQ $0x38, SP
  0x72a       48896c2430            MOVQ BP, 0x30(SP)
  0x72f       488d6c2430            LEAQ 0x30(SP), BP
        r, ok = rw.(io.Reader)
  0x734       488d0500000000        LEAQ 0(IP), AX    [3:7]R_PCREL:type.io.Reader
  0x73b       48890424              MOVQ AX, 0(SP)
  0x73f       488b442440              MOVQ 0x40(SP), AX
  0x744       4889442408              MOVQ AX, 0x8(SP)
  0x749       488b442448              MOVQ 0x48(SP), AX
  0x74e       4889442410              MOVQ AX, 0x10(SP)
  0x753       e800000000              CALL 0x758    [1:5]R_CALL:runtime.assertI2I2
  0x758       488b442418              MOVQ 0x18(SP), AX
  0x75d       488b4c2420              MOVQ 0x20(SP), CX
  0x762       0fb6542428              MOVZX 0x28(SP), DX
        return
  0x767       4889442450              MOVQ AX, 0x50(SP)
  0x76c       48894c2458              MOVQ CX, 0x58(SP)
  0x771       88542460                MOVB DL, 0x60(SP)
  0x775       488b6c2430              MOVQ 0x30(SP), BP
  0x77a       4883c438                ADDQ $0x38, SP
  0x77e       c3                      RET
func commaOk(rw io.ReadWriter) (r io.Reader, ok bool) {
  0x77f       e800000000              CALL 0x784    [1:5]R_CALL:runtime.morestack_noctxt
  0x784       eb8a                    JMP gom.commaOk(SB)
```

等价的Go风格伪代码如下：

```
func commaOk(rw io.ReadWriter) (r io.Reader, ok bool) {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    return runtime.assertI2I2(&type.io.Reader, i)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

这次是通过runtime.assertI2I2（）函数实现的，该函数的代码如下：

```
func assertI2I2(inter *interfacetype, i iface) (r iface, b bool) {
    tab := i.tab
    if tab == nil {
        return
    }
    if tab.inter != inter {
        tab = getitab(inter, tab._type, true)
        if tab == nil {
            return
        }
    }
    r.tab = tab
    r.data = i.data
    b = true
    return
}
```

如果i.tab为nil，则直接返回false。只有在i.tab.inter与inter不相等时才调用getitab（）函数，而且canfail为true，如果getitab（）函数失败，则不会造成panic，而是返回nil。

综上所述，I To I的类型断言，如图5-20所示，实际上是通过runtime.assertI2I（）函数和runtime.assertI2I2（）函数实现的，底层也都是基于getitab（）函数实现的。

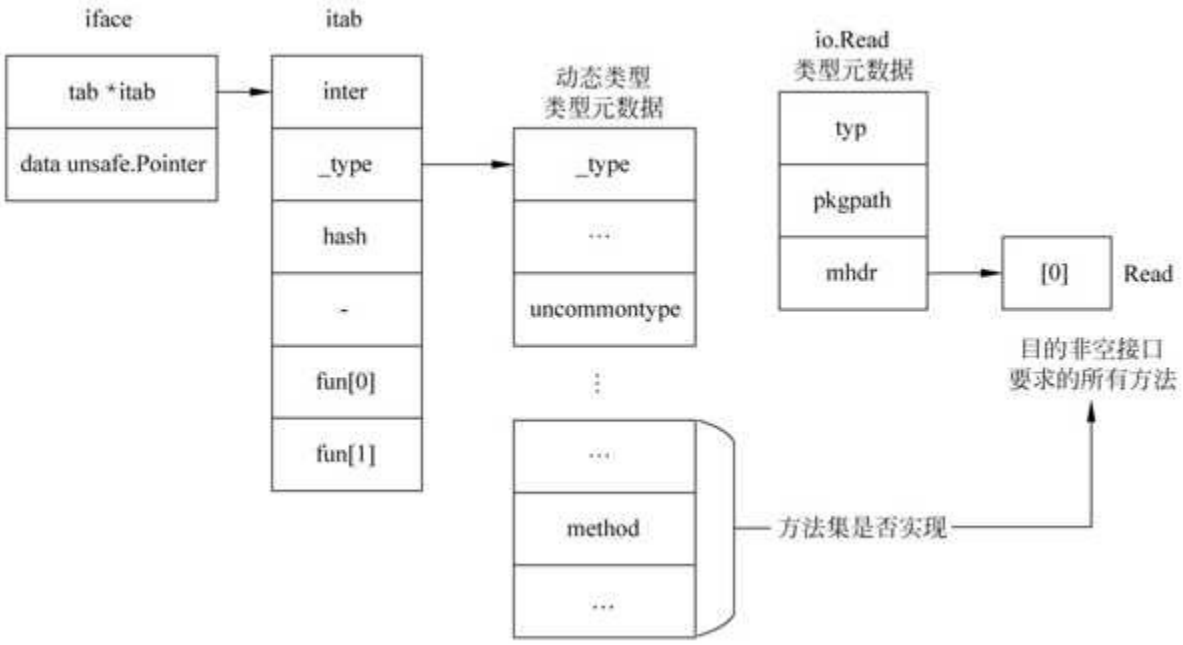

图5-20 从非空接口到非空接口的类型断言

5.4 反射

所谓反射，实际上就是围绕类型元数据展开的编程。程序的源码中包含最全面的类型信息，在C/C++一类的编程语言中，源码中的类型信息主要供编译阶段使用，这些类型信息定义了数据对象的内存布局、所支持的操作等，编译器依赖这些信息来生成相应的机器指令。经过编译之后，上层语言中那些直观、抽象的代码都被转换成了具体的机器指令，指令中操作的都是不同宽度的整型、浮点数这类很底层的数据类型，那些上层语言中的抽象数据类型也不复存在了。

而对于Go、Java这类支持反射的编程语言，经过编译阶段以后，代码中定义的各种类型信息会被保留下来。编译器会使用特定的数据结构来装载类型信息，并把它们写入生成的OBJ文件中，这些信息最终会被链接器存放到可执行文件相应的节区，供运行阶段检索使用。在Go语言中用来装载类型信息的数据结构就是5.1.2节介绍过的runtime._type，也就是我们俗称的类型元数据。在介绍动态派发和类型断言时，已经见识过类型元数据的重要性，本节就更系统地研究Go语言的类型系统，以及在此之上建立的强大的反射机制。

5.4.1 类型系统

Go语言一共提供了26种类型种类：一个布尔型，包含uintptr在内一共11种整型，两种浮点类型，两种复数类型，一个字符串类型，指针、数组、切片、map和struct共5种常用复合类型，以及chan、func、interface和unsafe.Pointer这4种特殊类型。这26种类型是Go语言整个类型系统的基础，任何更复杂的类型都由这些类型组合而来，即使用户自定义的类型有着各种各样的名称，它们的种类也不会超出这26种的范畴。

至此，我们已经知道类型元数据是用runtime._type结构表示的，那么这些数据是如何组织起来的，以及运行阶段又是如何解析的呢？带着这个问题，下面就深入runtime的源码中去找答案。

1. 类型信息的萃取

提到反射和类型，很自然地就会想起reflect包中用于获取类型信息的TypeOf（）函数，该函数有一个interface{}类型的参数，可以接受传入任意类型。函数的返回值类型是reflect.Type，这是个接口类型，提供了一系列方法来从类型元数据中提取信息。TypeOf（）函数所做的事情如图5-21所示，就是找到传入参数的类型元数据，并以reflect.Type形式返回。

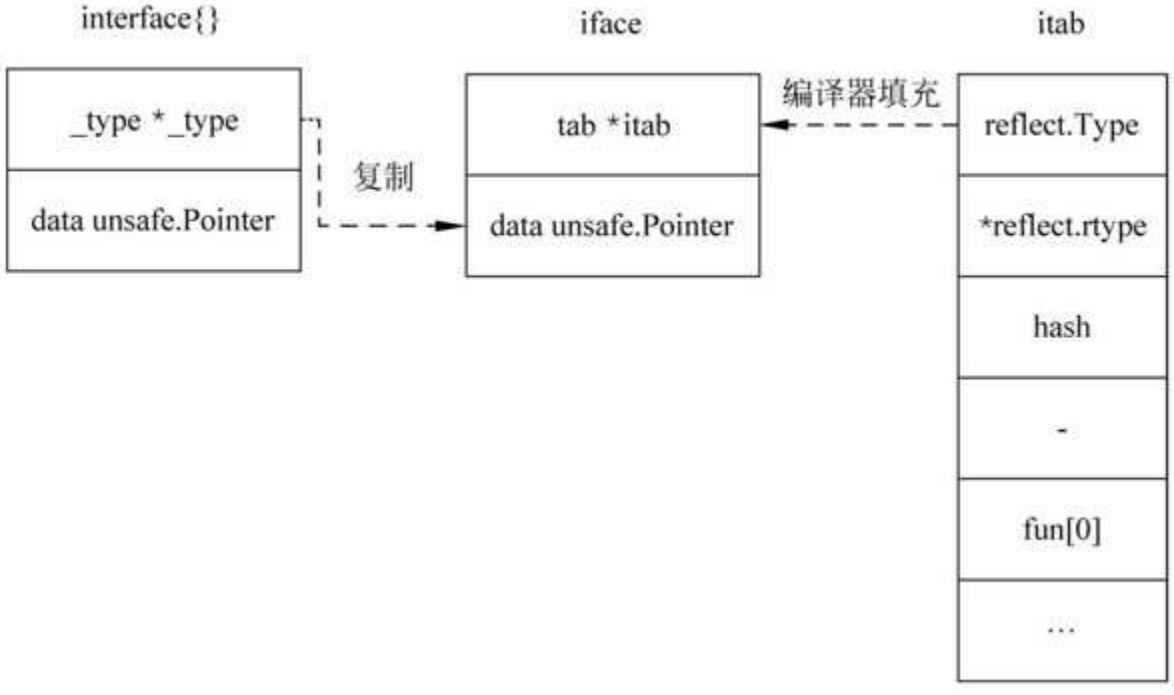

图5-21 由一个*_type和一个*itab组建一个iface

TypeOf（）函数的代码如下：

```
func TypeOf(i interface{}) Type {
    eface := *(*emptyInterface)(unsafe.Pointer(&i))
    return toType(eface.typ)
}
```

第2行代码相当于把传入的参数i强制转换成了emptyInterface类型，emptyInterface类型和5.1节介绍过的eface类型在内存布局上等价，emptyInterface类型定义的代码如下：

```
type emptyInterface struct {
    typ *rtype
    word unsafe.Pointer
}
```

其中的rtype类型与runtime._type类型在内存布局方面也是等价的，只不过因为无法使用其他包中未导出的类型定义，所以需要在reflect包中重新定义一下。代码中的eface.typ实际上就是从interface{}变量中提取出的类型元数据地址，再来看一下toType（）函数，代码如下：

```
func toType(t *rtype) Type {
    if t == nil {
        return nil
    }
    return t
}
```

先判断了一下传入的rtype指针是否为nil，如果不为nil就把它作为Type类型返回，否则返回nil。从这里可以知道*rtype类型肯定实现了Type接口，之所以要加上这个nil判断，需要考虑到Go的接口类型是个双指针结构，一个指向itab，另一个指向实际的数据对象。如图5-22所示，只有在两个指针都为nil的时候，接口变量才等于nil。

用一段更直观的代码加以说明，代码如下：

```
//第5章/code_5_12.go
var rw io.ReadWriter
if rw == nil {
    println(1)
}
var f *os.File
rw = f
if rw == nil {
    println(2)
}
```

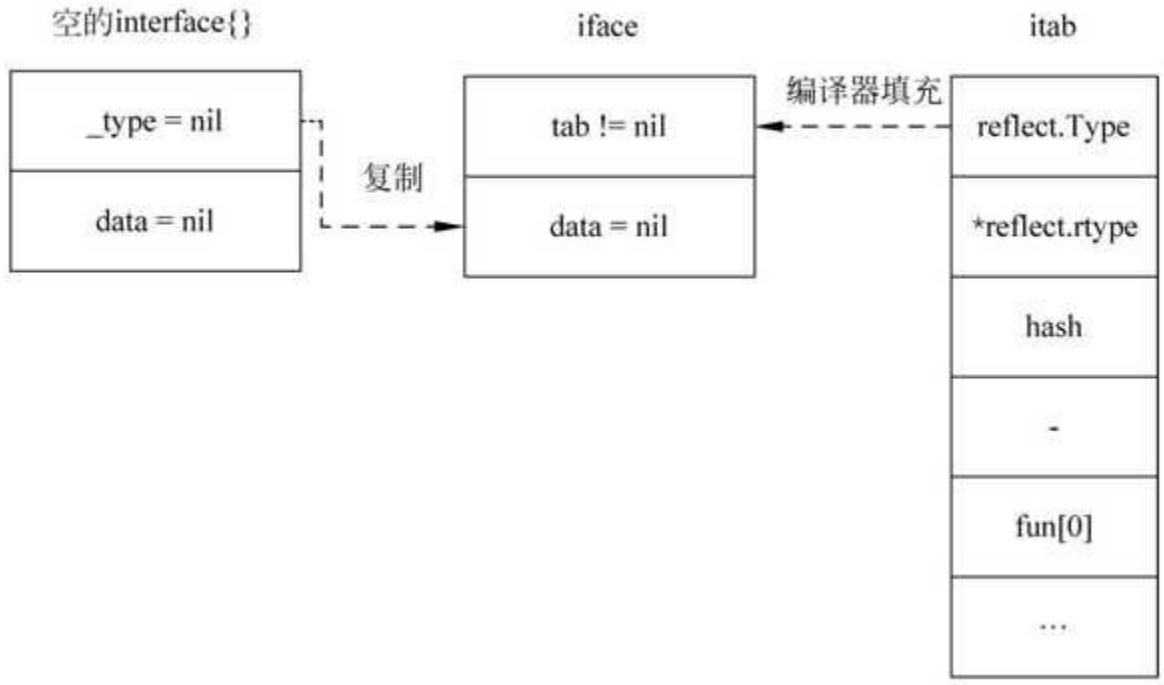

图5-22 萃取前判断非空

在上述代码中第1个if处判断结果为真，所以会打印出1。第2个if处rw不再为nil，所以不会打印2。这里需要注意一下，f本身为nil，赋值给rw之后却不再为nil，这是因为接口的双指针结构，其中数据指针为nil，itab指针不为空。也就是说nil指针也是有类型的，所以在赋值给interface{}和一般的非空接口变量时要格外注意。toType（）函数中前置的nil检测就是为了避免返回一个itab指针不为nil，而数据指针为nil的Type变量，使上层代码无法通过nil检测区分返回值是否有效，由此带来诸多不便和隐患。

综上所述，TypeOf（）函数所做的事情就是从interface{}中提取出类型元数据地址，然后在地址不为nil的时候将其作为Type类型返回。并没有太神奇的逻辑，而interface{}中的类型元数据地址是从哪里来的呢？当然是在编译阶段由编译器赋值的，实际的地址可能是由链接器填写的，也就是说源头还是要追溯到最初的源码中。

2. 类型系统的初始化

迄今为止，见过的所有基于类型元数据的特性都少不了interface的影子，通过反射实现类型信息的萃取也要依赖于interface参数，然而对于interface{}和非空接口，其中用到的类型元数据，论及源头都是在编译阶段由编译器赋值的。这样一来，整个类型系统给人的感觉就像是一个KV存储，只能在获得某个key的前提下去查询对应的value，有没有一个地方能够遍历所有的key呢？下面就带着这个问题去研究runtime的源码。

通过buildmode＝plugin可以把Go项目构建成一个动态链接库，后续以插件的形式被程序的主模块按需加载，这样一来运行阶段就需要加载多个二进制模块。由于每个模块中都有自己的一组类型元数据，所以就会出现类型信息不一致的问题，像类型断言这样的特性，底层通过比较元数据地址实现，也就无法正常工作了。保证类型系统中的类型唯一性至关重要，因此Go语言的runtime会在类型系统的初始化阶段进行去重操作，如图5-23所示。

下面从源码层面看一下具体的实现，用来初始化类型系统的就是runtime.typelinksinit（）函数，代码如下：

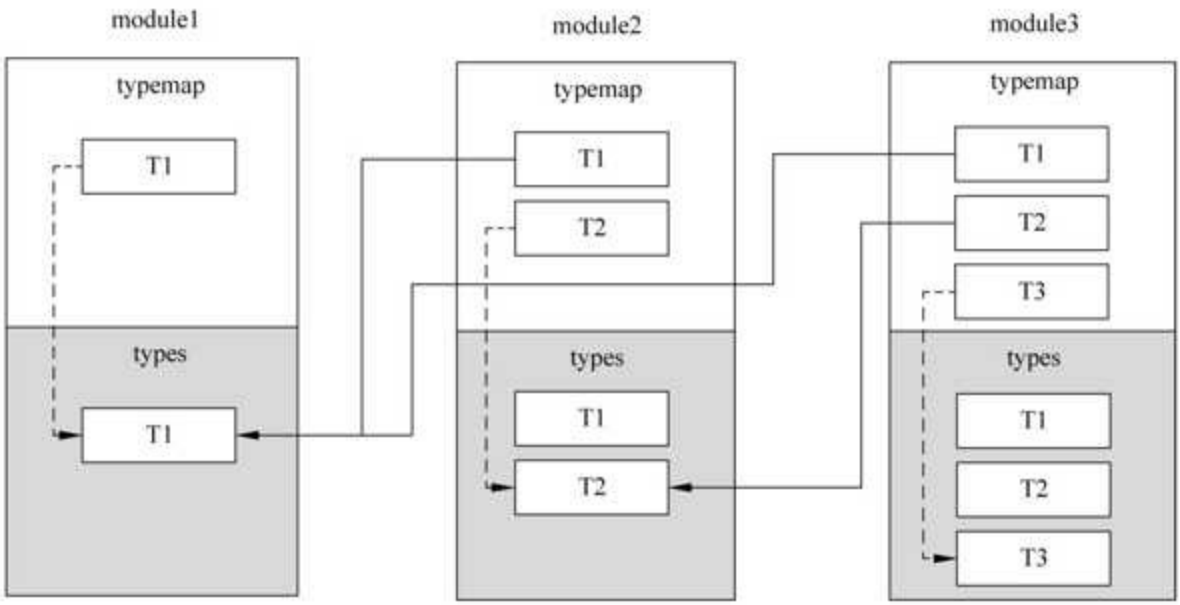

图5-23 类型系统初始化利用typemap去重

```
func typelinksinit() {
    if firstmoduledata.next == nil {
        return
    }
    typehash := make(map[uint32][]*_type, len(firstmoduledata.typelinks))

    modules := activeModules()
    prev := modules[0]
    for _, md := range modules[1:] {
        //把前一个模块中的各种类型收集到 typehash 中
    collect:
        for _, tl := range prev.typelinks {
            var t *_type
            if prev.typemap == nil {
                t = (*_type)(unsafe.Pointer(prev.types + uintptr(tl)))
            } else {
                t = prev.typemap[typeOff(tl)]
            }
            //已经有的就不重复添加了
            tlist := typehash[t.hash]
            for _, tcur := range tlist {
                if tcur == t {
                    continue collect
                }
            }
            typehash[t.hash] = append(tlist, t)
        }
```

```
        if md.typemap == nil {
            //如果当前模块 typelinks 中的某种类型与某个前驱模块中的某类型一致
            //通过当前模块的 typemap 将其映射到前驱模块中的对应类型
            tm := make(map[typeOff]*_type, len(md.typelinks))
            pinnedTypemaps = append(pinnedTypemaps, tm)
            md.typemap = tm
            for _, tl := range md.typelinks {
                t := (*_type)(unsafe.Pointer(md.types + uintptr(tl)))
                for _, candidate := range typehash[t.hash] {
                    seen := map[_typePair]struct{}{}
                    if typesEqual(t, candidate, seen) {
                        t = candidate
                        break
                    }
                }
                md.typemap[typeOff(tl)] = t
            }
        }

        prev = md
    }
}
```

在类型系统内部，元数据间通过typeOff互相引用，typeOff实际上就是个int32。类型元数据在二进制文件中是存放在一起的，单独占据了一段空间，moduledata结构的types字段和etypes字段就是这段空间的起始地址和结束地址。typeOff表示的就是目标类型的元数据距离起始地址types的偏移。梳理一下这个函数的大致逻辑：

（1）分配了一个map[uint32][]*_type类型的变量typehash，用来收集所有模块中的类型信息，用类型的hash作为map的key，收集的是类型元数据_type结构的地址，把hash相同的类型的地址放到同一个slice中。

（2）通过activeModules（）函数得到当前活动模块的列表，也就是所有能够正常使用的Go二进制模块，然后从第2个模块开始向后遍历。

（3）每次循环中通过前一个模块的typelinks字段，收集模块内的类型信息，将typehash中尚未包含的类型添加进去，注意是收集前一个模块的类型信息。这样一来，typehash中包含的类型信息都是该类型在整个模块列表中首次出现时的那个地址。假如按照A、B、C的顺序遍历模块列表，而类型T在B和C中都出现过，typehash中只会包含B模块中T的地址。

（4）如果当前模块的typemap为nil，就分配一个新的map并填充数据。遍历当前模块的typelinks，对于其中所有的类型，先去typehash中查找，优先使用typehash中的类型地址，typehash中没有的类型才使用当前模块自身包含的地址，把地址添加到typemap中。pinnedTypemaps主要是避免GC回收掉typemap，因为模块列表对于GC不可见。

这样当整个循环执行完成后，所有模块中的typemap中的任何一种类型都是该类型在整个模块列表中第一次出现时的地址，也就实现了类型信息的唯一化，而每个模块的typelinks字段就相当于遍历该模块所有类型的入口，虽然并不能从这里找到所有类型信息（有些闭包的类型信息就不会包含）。后续通过typeOff引用类型元数据时，会先从typemap中查找，如果找不到才会把当前模块的types加上typeOff作为结果返回，5.4.2节会更详细地分析讲解。

经过typelinksinit之后，用于反射的类型元数据实现了唯一化，跨多个模块的reflect就不会出现不一致现象了，但是回过头来继续看一看5.3.1节的类型断言的实现原理，底层直接比较类型元数据的地址，不会用到模块的typemap字段，所以上述唯一化操作应该无法解决这类问题。

类型断言所用到的元数据地址是由编译器直接编码在指令中的，下面先来研究一下编译器是如何确定类型元数据地址的，代码如下：

```
func IsBool(a interface{}) bool {
    _, ok := a.(bool)
    return ok
}
```

用compile命令将上述代码编译成OBJ文件，然后进行反编译，得到的汇编代码如下：

```
$ go tool compile -p gom -o assert.o assert.go
$ go tool objdump -S -s 'IsBool' assert.o
TEXT gom.IsBool(SB) gofile../home/kylin/go/src/fengyoulin.com/gom/assert.go
        _, ok := a.(bool)
  0x422       488b442408          MOVQ 0x8(SP), AX          //第 1 条指令
  0x427       488d0d00000000      LEAQ 0(IP), CX
                                  [3:7]R_PCREL:type.bool    //第 2 条指令
  0x42e       4839c8              CMPQ CX, AX
        return ok
  0x431       0f94442418          SETE 0x18(SP)
  0x436       c3                  RET
```

第2条汇编指令LEAQ用于获取bool类型元数据的地址，第1个操作数0（IP）中的0是个偏移量，编译阶段只预留了4字节的空间，所以在OBJ文件中是0，等到链接器填写了实际的偏移量后可执行文件中就会有值了。LEAQ offset（IP），CX的含义就是把当前指令指针IP的值加上offset，把结果存入CX寄存器中。这种计算方式是以当前指令位置为基址，然后加上32位的偏移来得到目标地址。32位偏移能够覆盖-2GB～2GB的偏移范围，多用于单个二进制文件内部的寻址，因为单个二进制文件的大小一般不会超过2GB。

对于模块间的地址引用，这种相对地址的计算方式就不能很好地支持了。因为64位地址空间中两个模块间的距离可能会超过2GB，所以需要直接使用64位宽度的地址。还是使用IsBool（）函数，这次编译的时候加上一个dynlink参数，实际上在以plugin方式构建项目时工具链会自动添加这个编译参数。再反编译得到的OBJ文件，汇编代码如下：

```
$ go tool compile -dynlink -p gom -o assert.o assert.go
$ go tool objdump -S -s 'IsBool' assert.o
TEXT gom.IsBool(SB) gofile../home/kylin/go/src/fengyoulin.com/gom/assert.go
        _, ok := a.(bool)
  0x44d       488b442408          MOVQ 0x8(SP), AX
  0x452       488b0d00000000      MOVQ 0(IP), CX    [3:7]R_GOTPCREL:type.bool
  0x459       4839c8              CMPQ CX, AX
        return ok
  0x45c       0f94442418          SETE 0x18(SP)
  0x461       c3                  RET
```

唯一的不同就是原来的LEAQ变成了MOVQ，含义也发生了很大变化，LEAQ与MOVQ的区别如图5-24所示。LEAQ直接把当前指令地址加上偏移用作元数据地址，而MOVQ从当前指令地址加上偏移处取出一个64位整型，用作类型元数据的地址。也就是MOVQ不直接计算元数据地址，而是又多了一层中转，也就是又多了一层灵活性。

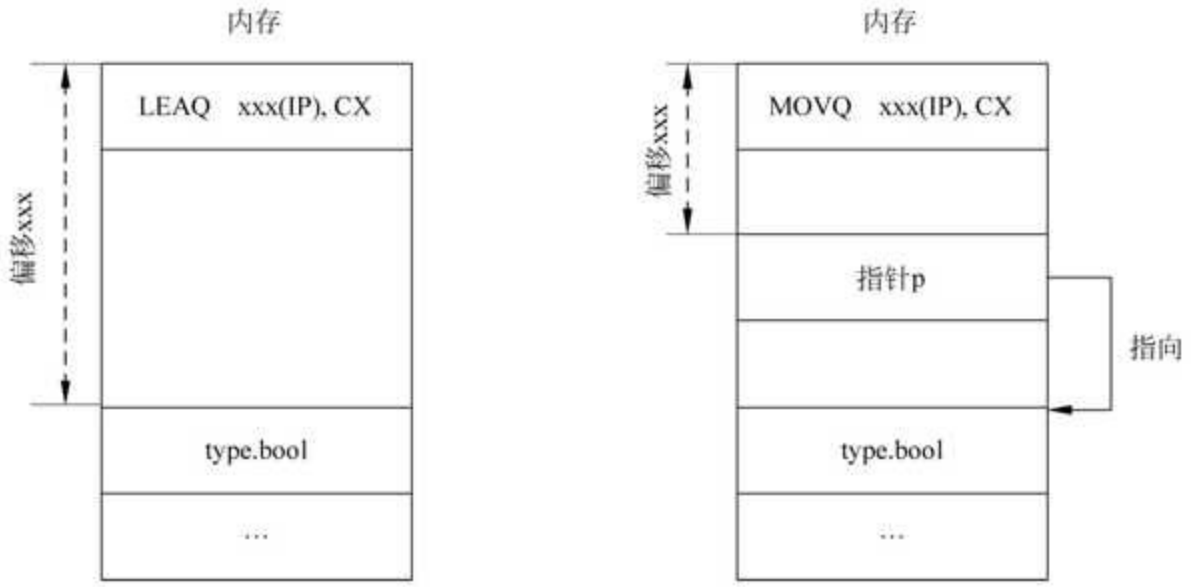

图5-24 LEAQ与MOVQ的区别

进一步分析会发现，MOVQ读取地址的地方是ELF文件中一个叫.got的节区，.got节中有一个全局偏移表（Global Offset Table），表中的一系列重定位项会在ELF文件被加载的时候由操作系统的动态链接器完成赋值。像类型断言这种，代码中直接使用元数据地址的场景，其中的类型唯一性问题在二进制模块加载的时候就被动态链接器处理掉了，如图5-25所示。

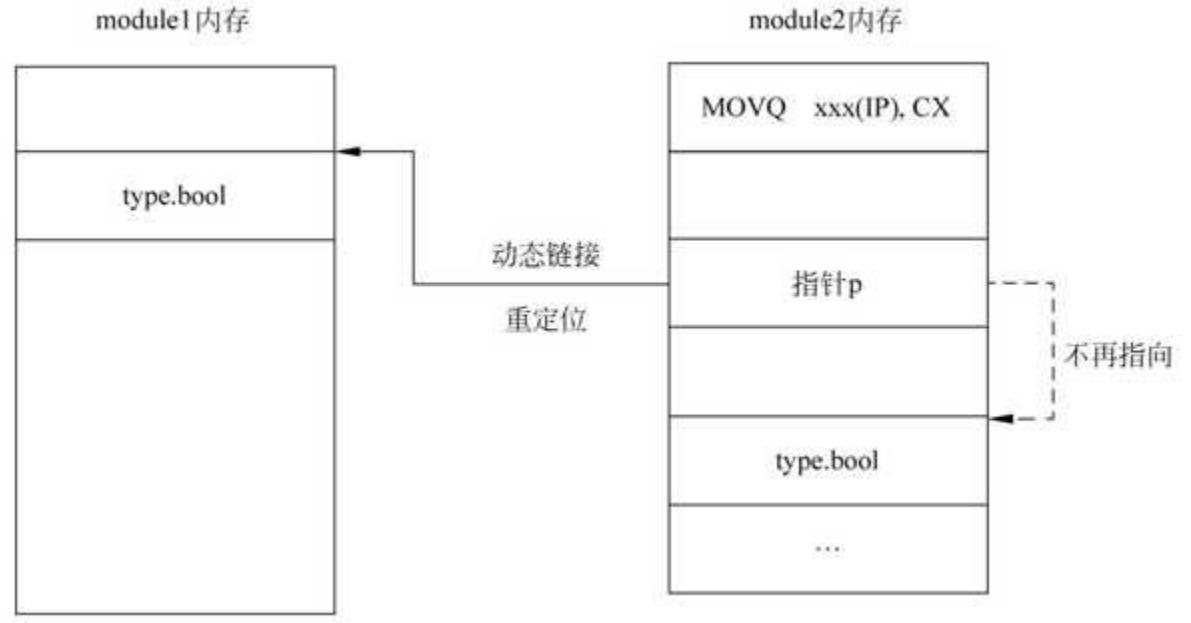

图5-25 地址被动态链接重定位直接使用类型元数据

讲解了这么多，都是通过读源码和反编译的方式来分析的，还是需要有个实例来运行验证一下。下面就基于Go的plugin机制来实践一下，实验环境是运行在amd64架构上的Linux系统。

首先创建第1个mod，这个模块只定义了一个User类型，下面来看各个文件的源码。

（1）go.mod文件的代码如下：

```
//第 5 章/mod1/go.mod
module fengyoulin.com/mod1

go 1.14
```

（2）user.go文件的代码如下：

```
//第 5 章/mod1/user.go
package mod1

type User struct {
    ID int
    Nick string
}
```

然后是第2个mod，这个模块按照plugin的形式，实现了一个UserFactory。

（1）go.mod文件的代码如下：

```
//第 5 章/mod2/go.mod
module fengyoulin.com/mod2

go 1.14

require fengyoulin.com/mod1 v0.0.0

replace fengyoulin.com/mod1 => /home/kylin/go/src/fengyoulin.com/mod1
```

（2）factory.go文件的代码如下：

```
//第 5 章/mod2/factory.go
package main

import "fengyoulin.com/mod1"

type uf struct{}

func (*uf) NewUser(id int, nick string) interface{} {
    return &mod1.User{
        ID: id,
        Nick: nick,
    }
}

var UserFactory uf
```

接下来是第3个mod，这个模块也是一个plugin，实现了一个UserChecker。（1）go.mod文件的代码如下：

```
//第 5 章/mod3/go.mod
module fengyoulin.com/mod3

go 1.14

require fengyoulin.com/mod1 v0.0.0

replace fengyoulin.com/mod1 => /home/kylin/go/src/fengyoulin.com/mod1
```

（2）checker.go文件的代码如下：

```
//第 5 章/mod3/checker.go
package main

import "fengyoulin.com/mod1"

type uc struct{}

func (*uc) IsUser(a interface{}) bool {
    _, ok := a.(*mod1.User)
    return ok
}

var UserChecker uc
```

第4个模块，也是最后一个模块，此模块是用来加载并调用前面两个plugin的主程序。

（1）go.mod文件的代码如下：

```
//第 5 章/mod4/go.mod
module fengyoulin.com/mod4

go 1.14
```

（2）main.go文件的代码如下：

```
//第 5 章/mod4/main.go
package main

import (
    "log"
    "plugin"
    "reflect"
)

type UserFactory interface {
    NewUser(id int, nick string) interface{}
}

type UserChecker interface {
    IsUser(a interface{}) bool
}

func factory() UserFactory {
    p, err := plugin.Open("./mod2.so")
    if err != nil {
        log.Fatalln(err)
    }
    a, err := p.Lookup("UserFactory")
```

```
    if err != nil {
        log.Fatalln(err)
    }
    uf, ok := a.(UserFactory)
    if !ok {
        log.Fatalln("not a UserFactory")
    }
    return uf
}

func checker() UserChecker {
    p, err := plugin.Open("./mod3.so")
    if err != nil {
        log.Fatalln(err)
    }
    a, err := p.Lookup("UserChecker")
    if err != nil {
        log.Fatalln(err)
    }
    uc, ok := a.(UserChecker)
    if !ok {
        log.Fatalln("not a UserChecker")
    }
    return uc
}

func main() {
    uf := factory()
    uc := checker()
    u := uf.NewUser(1, "Jack")
    if !uc.IsUser(u) {
        log.Println("not a User")
    }
    t := reflect.TypeOf(u)
    println(u, t.String())
    select{}
}
```

最后，以plugin模式构建mod2和mod3，会得到两个so库，命令如下：

```
$ go build -buildmode=plugin
```

主程序mod4直接使用go build命令以默认方式构建就可以了。构建完成后，将mod2.so及mod3.so复制到mod4所在目录下，然后运行mod4，命令如下：

```
$ ./mod4
(0x7f3039507ba0,0xc0000a0100) *mod1.User
```

其中第1个地址0x7f3039507ba0就是*mod1.User的类型元数据的地址，可以通过查看当前进程地址空间中的模块布局，来确定该地址位于哪个模块中。打开另一个终端，执行命令如下：

```
$ ps aux | grep mod4
kylin 16805 0.0 0.2 751788 5880 pts/0 Sl+ 21:18 0:00 ./mod4
...
$ cat /proc/16805/maps
...
7f3038efa000-7f3038fb9000 r-xp 00000000 fd:02 1057567 ./mod3.so
7f3038fb9000-7f30391b9000 ---p 000bf000 fd:02 1057567 ./mod3.so
7f30391b9000-7f3039212000 r--p 000bf000 fd:02 1057567 ./mod3.so
7f3039212000-7f3039216000 rw-p 00118000 fd:02 1057567 ./mod3.so
7f3039216000-7f3039241000 rw-p 00000000 00:00 0
7f3039241000-7f3039300000 r-xp 00000000 fd:02 1057436 ./mod2.so
7f3039300000-7f3039500000 ---p 000bf000 fd:02 1057436 ./mod2.so
7f3039500000-7f3039559000 r--p 000bf000 fd:02 1057436 ./mod2.so
7f3039559000-7f303955d000 rw-p 00118000 fd:02 1057436 ./mod2.so
7f303955d000-7f3039588000 rw-p 00000000 00:00 0
...
```

可以看到类型元数据的地址落在了mod2.so的第3个区间内，也就是说mod3.so的got中的地址项被动态链接器修改了。假如对换一下mod4的main（）函数的前两行代码的顺序，也就是先加载mod3.so，后加载mod2.so，就会发现程序使用的*mod1.User的元数据位于mod3.so中，也就是以先加载的模块为准，感兴趣的读者可以自己尝试，这里不再赘述。

综上所述，代码中typelinksinit构造了各模块的typemap（首个模块除外），这样就实现了类型元数据间引用关系的唯一化，而在二进制模块加载时动态链接器能够使代码中引用的类型元数据地址唯一化，前者作用于类型系统内部，后者作用于类型系统的入口，从而整体上解决了多个二进制模块的类型信息不一致问题。

5.4.2 类型元数据详细讲解

在5.1.2节已经介绍过用来表示类型元数据的runtime._type类型，以及其中各个字段的含义，reflect包中的rtype类型与runtime._type类型是等价的。本节深入研究各种类型的元数据细节，重点分析array、slice、map、struct及指针等几种复合数据类型的元数据结构。再结合反射提供的方法，探索类型系统是如何解析元数据的。

下面先看一下布尔、整型、浮点、复数、字符串和unsafe.Pointer这些基本类型，元数据中关键字段的取值如表5-3所示。

表5-3 基本类型元数据中关键字段的取值

type	kind	size	ptrdata	tflag	align	fieldAlign	equal
bool	1	1	0	15	1	1	runtime.memequal8
int	2	8	0	15	8	8	runtime.memequal64
int8	3	1	0	15	1	1	runtime.memequal8
int16	4	2	0	15	2	2	runtime.memequal16
int32	5	4	0	15	4	4	runtime.memequal32
int64	6	8	0	15	8	8	runtime.memequal64
uint	7	8	0	15	8	8	runtime.memequal64
uint8	8	1	0	15	1	1	runtime.memequal8
uint16	9	2	0	15	2	2	runtime.memequal16
uint32	10	4	0	15	4	4	runtime.memequal32
uint64	11	8	0	15	8	8	runtime.memequal64
uintptr	12	8	0	15	8	8	runtime.memequal64
float32	13	4	0	7	4	4	runtime.f32equal
float64	14	8	0	7	8	8	runtime.f64equal
complex32	15	8	0	7	4	4	runtime.c64equal
complex64	16	16	0	7	8	8	runtime.c128equal
string	24	16	8	7	8	8	runtime.strequal
unsafe.Pointer	58	8	8	15	8	8	runtime.memequal64

其中有几个地方需要解释一下：

（1）unsafe.Pointer类型的kind值是58，实际上kind字段只有低5位用来表示数据类型所属的种类，第6位在源码中定义为kindDirectIface，其含义是该类数据可以直接存储在interface中。通过5.1节和5.2节已经知道interface的结构实际上是个双指针，所以能够直接存储在其中的类型，本质上来讲应该都是个地址。除了地址之外，其他的值类型需要经过装箱操作。unsafe.Pointer类型可以直接存储在interface中，所以其kind值就是原本的类型编号26|32＝58。

（2）ptrdata一列表示数据类型的前多少字节内包含地址，string类型本质上是一个指针和一个整型组成的结构，在amd64平台上指针大小为8字节。unsafe.Pointer本身是一个指针。

（3）对于浮点、复数和string类型，tflag中的tflagRegularMemory位没有被设置。浮点数不能直接像整型那样直接比较内存，string包含指针，实际上数据存储在别的地方。这一点通过最后一列的equal函数也可以看出来。

对于复合类型而言，单个rtype结构就不够用了，所以会在此基础之上进行扩展，利用struct嵌入可以很方便地实现。用来描述array类型的arrayType定义的代码如下：

```
type arrayType struct {
    rtype
    elem * rtype //数组元素类型
    slice * rtype //切片类型
    len uintptr
}
```

其中的rtype嵌入arrayType结构中，相当于arrayType继承自rtype。elem指向数组元素的类型元数据，len表示数组的长度，通过元素类型和长度就确定了数组的类型。slice字段指向相同元素类型的切片对应的元数据，因为反射提供的与切片相关的函数在操作数组时需要根据array的元数据找到slice的元数据，这样直接持有一个地址更加高效。

切片类型元数据的结构比数组要简单一些，除了rtype和元素类型外，没有了长度字段，也不用指向其他类型，因为切片运算的结果还是切片类型，代码如下：

```
type sliceType struct {
    rtype
    elem * rtype //切片元素类型
}
```

指针类型的元数据结构和切片类型一样，除了嵌入的rtype之外，还包含了一个元素类型，也就是指针所指向的数据的类型，代码如下：

```
type ptrType struct {
    rtype
    elem * rtype //指向的元素类型
}
```

struct类型的元数据结构就稍微复杂一些了，有一个pkgPath字段记录着该struct被定义在哪个包里，还有一个切片记录着一组structField，也就是struct的所有字段，代码如下：

```
type structType struct {
    rtype
    pkgPath name
    fields []structField //按照在 struct 内的 offset 排列
}
```

每个structField用于描述struct的一个字段，字段必须有名字，所以name字段不能为空。typ指向字段类型对应的元数据，offsetEmbed字段是由两个值组合而成的，先把字段的偏移量的值左移一位，然后最低位用来表示是否为嵌入字段，代码如下：

```
type structField struct {
    name        name     //始终非空
    typ         * rtype  //字段的类型
    offsetEmbed uintptr  //字段偏移量,是否为嵌入字段
}
```

map的元数据结构就更复杂了，需要记录key、elem及bucket对应的类型元数据地址，还有用来对key进行哈希运算的hasher（）函数，还要记录key slot、value slot及bucket的大小，flags字段用来记录一些标志位，代码如下：

```
type mapType struct {
    rtype
    key         * rtype   //key 类型
    elem        * rtype   //元素类型
    bucket      * rtype   //内部 bucket 的类型
    hasher      func(unsafe.Pointer, uintptr) uintptr
    keysize     uint8     //key slot 大小
    valuesize   uint8     //value slot 大小
    bucketsize  uint16    //bucket 大小
    flags       uint32
}
```

其中flags字段的几个标志位的含义如表5-4所示。

表5-4 flags字段的几个标志位的含义

标志位	含　义
最低位	表示 key 是以间接方式存储的，因为当 key 的数据类型大小超过 128 后，就会存储地址而不是直接存储值
第二位	表示 value 是以间接方式存储的，与 key 一样，value 类型大小超过 128 后就会存储地址
第三位	表示 key 的数据类型是 reflexive 的，也就是可以使用==运算符来比较相等性
第四位	表示 map 在覆盖时 key 是否需要被再复制一次（覆盖），否则在 key 已经存在的情况下不会对 key 进行赋值
第五位	表示 hash 函数可能会触发 panic

下面再来看一下channel的类型元数据结构，elem字段指向元素类型，dir字段存储了通道的方向，也就是send、recv，或者既send又recv，代码如下：

```
type chanType struct {
    rtype
    elem * rtype //channel 元素类型
    dir uintptr //channel 方向(send,recv)
}
```

关于dir字段，虽然在结构体中的类型是uintptr，但是reflect包在操作该字段的时候会把它转换为reflect.ChanDir类型。ChanDir类型本质上是个int，表示的是channel的方向，定义了3个常量值：RecvDir的值是1，表示可以recv；SendDir的值是2，表示可以send；BothDir是前两者的组合，值是3，表示既能recv又能send。

接下来是函数类型的元数据结构，inCount表示输入参数的个数，outCount表示返回值的个数。这两个count都是uint16类型，所以理论上可以有65535个入参，由于outCount的最高位被用来表示最后一个入参是否为变参（…），所以理论上的返回值最多有32767个，代码如下：

```
type funcType struct {
    rtype
    inCount uint16
    outCount uint16 //最高位表示是否为变参函数
}
```

最后就是接口类型的元数据结构，与runtime.interfacetype是等价的，在5.2节已经分析过了，此处不再赘述。在reflect包中的定义代码如下：

```
type interfaceType struct {
    rtype
    pkgPath name
    methods []imethod
}
```

至此，总共26种类型都介绍完了，Go语言中所有的内置类型、标准库类型，以及用户自定义类型都不会超出这26种类型。

下面来看一下，运行阶段如何根据typeOff定位元数据的地址，以及在存在多个模块时是如何利用各模块的typemap实现唯一化的，主要逻辑在runtime.resolveTypeOff（）函数中，代码如下：

```
func resolveTypeOff(ptrInModule unsafe.Pointer, off typeOff) *_type {
    if off == 0 {
        return nil
    }
    base := uintptr(ptrInModule)
    var md *moduledata
    for next := &firstmoduledata; next != nil; next = next.next {
        if base >= next.types && base < next.etypes {
            md = next
            break
        }
    }
    if md == nil {
        reflectOffsLock()
        res := reflectOffs.m[int32(off)]
        reflectOffsUnlock()
        if res == nil {
            //省略少量代码
            throw("runtime: type offset base pointer out of range")
        }
        return (*_type)(res)
    }
    if t := md.typemap[off]; t != nil {
        return t
    }
    res := md.types + uintptr(off)
    if res > md.etypes {
        //省略少量代码
        throw("runtime: type offset out of range")
    }
    return (*_type)(unsafe.Pointer(res))
}
```

因为typeOff这个偏移量是相对于模块的types起始地址而言的，所以要通过ptrInModule来确定是在哪个模块中查找。该函数的逻辑大致分为以下几步：

（1）遍历所有模块，查找ptrInModule这个地址落在哪个模块的types区间内，后续就在这个模块中查找。

（2）如果上一步没能找到对应的模块，就到reflectOffs中去查找，这里面都是运行阶段通过反射机制动态创建的类型，如果找到，则直接返回。

（3）尝试在模块的typemap中通过off查找对应的类型，如果找到，则直接返回。因为typemap中已经是typelinksinit处理好的数据，这一步实现了类型信息的唯一化。

（4）最后才会尝试用types直接加上off作为元数据地址，只要该地址没有超出当前模块的类型数据区间就行。因为首个模块没有typemap，所以这一步是必要的。

最后来看一下反射是如何在运行阶段创建类型的。构造对应的类型元数据并没有什么难点，关键是如何与编译阶段生成的大量类型信息整合起来。因为是运行阶段创建的类型，所以不会有重定位之类的问题，只需考虑如何根据typeOff来检索就好了。reflect包中addReflectOff（）函数用来为动态生成的类型分配typeOff，具体逻辑是在runtime.reflect_addReflectOff（）函数中实现的，reflect.addReflectOff（）函数又是通过linkname机制链接过去的，函数的代码如下：

```
func reflect_addReflectOff(ptr unsafe.Pointer) int32 {
    reflectOffsLock()
    if reflectOffs.m == nil {
        reflectOffs.m = make(map[int32]unsafe.Pointer)
        reflectOffs.minv = make(map[unsafe.Pointer]int32)
        reflectOffs.next = -1
    }
    id, found := reflectOffs.minv[ptr]
    if !found {
        id = reflectOffs.next
        reflectOffs.next--
        reflectOffs.m[id] = ptr
        reflectOffs.minv[ptr] = id
    }
    reflectOffsUnlock()
    return id
}
```

在梳理该函数的逻辑之前，有必要先弄清楚reflectOffs的类型，代码如下：

```
var reflectOffs struct {
    lock mutex
    next int32
    m map[int32]unsafe.Pointer
    minv map[unsafe.Pointer]int32
}
```

其中，lock用来保护整个struct中的其他字段，next表示下一个可分配的typeOff值，m是从typeOff值到类型元数据地址的映射，minv是m的逆映射，也就是从类型元数据地址到typeOff的映射。理清这些之后，再来梳理上面函数的逻辑：

（1）先加锁。

（2）通过检查m是否为nil来判断是否已经初始化了，注意next的初始值是-1。

（3）先通过元数据的地址ptr在minv里面查找，如果已经有了就不再添加了。

（4）把next的值作为typeOff分配给ptr，分别添加到m和minv中，然后递减next。

（5）解锁，返回查找到的或新分配的typeOff。

所以运行阶段动态分配的typeOff都是负值，只是用作唯一ID，并不是真正地偏移了，而编译阶段生成的typeOff是真正的偏移，是与本模块types区间起始地址的差，都是正值。返回去再看前面的resolveTypeOff（）函数，只有在通过ptrInModule找不到对应的二进制模块时才会查找reflectOffs，因为编译时期生成的那些类型元数据是不可能依赖动态生成的类型元数据的，只有动态生成的类型元数据才有可能依赖动态生成的类型元数据，而动态分配的内存是不会匹配上任何一个模块的types区间的。

关于类型元数据的分析就到这里，笔者只是选了自己认为还算重要的几部分内容着重分析了一下，感兴趣的读者可以从reflect的源码中发现更多有趣的细节，这里就不占用更多篇幅了。

5.4.3 对数据的操作

至此，对于反射如何解析类型元数据已经有了大致的了解，而大多数场景下使用反射的最终目的是

操作数据。为了便于对数据进行操作，reflect包提供了Value类型，通过该类型的一系列方法来动态操作各种数据类型。Value类型本身是个struct，代码如下：

```
type Value struct {
    typ  *rtype
    ptr unsafe.Pointer
    flag
}
```

Value的作用就像它的名字那样，用来装载一个值，其中的typ字段指向值的类型对应的元数据。ptr字段可能是值本身（对于本质上是个地址的值，即kindDirectIface），也可能是一段内存的起始地址，实际的值存放在那里。flag字段存储了一系列标志位，各个标志位的含义如表5-5所示。

表5-5 flag字段各个标志位的含义

标 志 位	含 义
flagStickyRO：1<<5	未导出且非嵌入的字段，是只读的
flagEmbedRO：1<<6	未导出且嵌入的字段，是只读的
flagIndir：1<<7	ptr字段中存储的是值的地址，而非值本身
flagAddr：1<<8	值是可寻址的(addressable)
flagMethod：1<<9	值是个Method Value

其中前两个只读标志位主要是针对struct的字段而言的，如果目标字段也是个struct，这些只读标志会被更内层的字段继承。flag本质上是个uintptr，所以至少有32位，最低5位一般与typ.kind的低5位一致，只有在值是个Method时例外，此时flag的低5位为reflect.Func，高22位存储了Method在方法集中的序号，方法的接收者是通过typ和ptr来描述的，如图5-26所示。

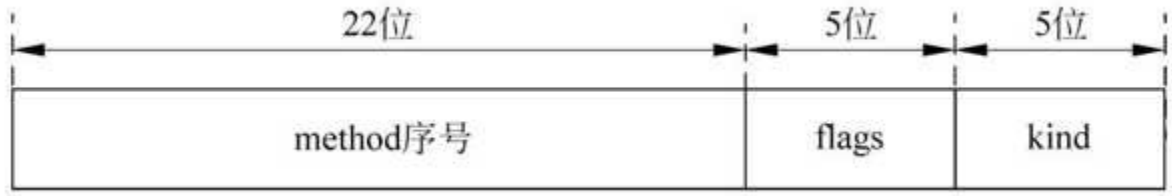

图5-26 flag字段的结构

再来看一下reflect.ValueOf（）函数，该函数会返回一个Value对象。类似于reflect.TypeOf（）函数，可以认为是反射操作数据的起点，代码如下：

```
func ValueOf(i interface{}) Value {
    if i == nil {
        return Value{}
    }
    escapes(i)
    return unpackEface(i)
}
```

一个入参，类型也是interface{}，如果为nil，就会返回一个零值的Value，零值的Value是Invalid的。escapes的作用是确保i.data指向的数据会逃逸，因为反射相关的代码涉及较多unsafe操作，编译器的逃逸分析极有可能无法追踪某些实质上逃逸了的变量，而误把它们分配到栈上，从而造成问题。后续的版本可能会允许Value指向栈上的值，现阶段先忽略此问题。最后的unpackEface（）函数才是关键，代码如下：

```
func unpackEface(i interface{}) Value {
    e := (*emptyInterface)(unsafe.Pointer(&i))
    t := e.typ
    if t == nil {
        return Value{}
    }
    f := flag(t.Kind())
    if ifaceIndir(t) {
        f |= flagIndir
    }
    return Value{t, e.word, f}
}
```

如果e.typ为nil，也就是得不到类型元数据，就返回一个无效的Value对象。用t.Kind（）的返回值对flag进行初始化，也就是复制了t.kind的低5位。如果值本身不是个地址，还要设置flagIndir标志位。ptr字段也是直接复制自e.word，也就是interface{}中的数据指针。

用一段实际的代码看一下typ和flag的取值，代码如下：

```
//第 5 章/code_5_13.go
type Value struct {
    typ unsafe.Pointer
    ptr unsafe.Pointer
    flag uintptr
}

func toType(p unsafe.Pointer) (t reflect.Type) {
    t = reflect.TypeOf(0)
    (*[2]unsafe.Pointer)(unsafe.Pointer(&t))[1] = p
    return
}

func main() {
    n := 6789
    s := []interface{}{
        n,
        &n,
    }
    for i, v := range s {
        r := reflect.ValueOf(v)
        p := (*Value)(unsafe.Pointer(&r))
        println(i, p.typ, p.ptr, p.flag, toType(p.typ).String())
    }
}
```

这段代码的作用是分别基于int和*int两种类型的输入，用reflect.ValueOf（）函数得到两个Value，然后打印出Value的各个字段。在笔者的计算机上得到的输出如下：

```
$ ./code_5_13.exe
0 0x24c060 0xc00000c078 130 int
1 0x2487c0 0xc00000c070 22 *int
```

其中int类型对应的flag是130＝128+2，也就是flagIndir加上kindInt。*int类型对应的flag是22，等于kindPtr。事实上unpackEface（）函数只是简单地从interface{}中复制了类型指针和数据指针，在把int类型赋值给interface{}时发生了装箱操作，所以设置了flagIndir。

由此看来，Value和interface{}非常相似，都有一个类型指针和一个数据指针，不同的是Value多了一个flag字段，基于flag中提供的信息可以实现很多很灵活的操作，比较典型的有如Elem（）方法和Addr（）方法。先来看一下Elem（）方法，代码如下：

```
func (v Value) Elem() Value {
    k := v.kind()
    switch k {
    case Interface:
        var eface interface{}
        if v.typ.NumMethod() == 0 {
            eface = *(*interface{})(v.ptr)
        } else {
            eface = (interface{})(*(*interface {
                M()
            })(v.ptr))
        }
        x := unpackEface(eface)
        if x.flag != 0 {
            x.flag |= v.flag.ro()
        }
        return x
    case Ptr:
        ptr := v.ptr
        if v.flag&flagIndir != 0 {
            ptr = *(*unsafe.Pointer)(ptr)
        }
        if ptr == nil {
            return Value{}
        }
        tt := (*ptrType)(unsafe.Pointer(v.typ))
        typ := tt.elem
        fl := v.flag&flagRO | flagIndir | flagAddr
        fl |= flag(typ.Kind())
        return Value{typ, ptr, fl}
    }
    panic(&ValueError{"reflect.Value.Elem", v.kind()})
}
```

Elem（）方法的功能是根据地址返回地址处存储的对象，要求v的kind必须是Interface或Ptr，否则就会造成panic。已经分析过interface的双指针结构，可以把它等价于一个带有类型的指针。下面先来梳理一下处理Interface的逻辑：

（1）通过接口方法数判断是否为eface，如果方法数为0就可以直接把ptr强制转换为*interface{}类型，然后通过指针解引用操作得到eface的值。

（2）对于方法数不为0的接口类型就是iface，先把ptr强制转换为*interface{M（）}类型，然后通过指针解引用操作得到iface的值，再强制转换为interface{}类型，也就是eface。

（3）调用unpackEface（）函数，从eface中提取类型指针和数据指针的值，并设置flag字段，返回一个新的Value。这一步几乎等价于ValueOf（）函数。

（4）通过设置flag来继承v的只读相关标志位。

前两步是从ptr地址处提取出interface{}类型的值，第二步需要解释一下，有关不同接口类型间的强制类型转换。假如有A、B两个接口类型，其中A的方法列表是B方法列表的子集，那么编译器允许通过强制类型转换把B类型的实例转换成A类型。例如从io.ReadWriter到io.Reader，也可以从io.Writer到interface{}，因为空集是任意集合的子集，所以第二步接口中的M方法没有实际意义，只是告诉编译器这是个有方法的接口，双指针是itab指针和数据指针。

对于不同iface之间的强制类型转换，编译器会调用runtime.convI2I（）函数。从iface到eface的强制类型转换，编译器直接生成代码复制类型元数据指针和数据指针。

再来梳理一下处理Ptr时的逻辑：

（1）检查flag中的flagIndir标志，如果是间接存储的，就进行一次指针解引用操作。

（2）如果ptr为nil，就返回一个无效的Value。

（3）将typ修改为指针元素类型对应的元数据地址。

（4）根据typ.Kind（）函数设置新的flag，设置flagIndir和flagAddr标志，并继承只读标志。

（5）基于新的typ、ptr和flag构造Value并返回结果。

其中值得注意的是flagAddr标志，通俗来讲该标志位表示能够获得原始变量的地址，而不只是值的副本，Set系列方法会检查该标志位，只有在设置了该标志位的情况下才允许修改，否则是没有意义的，会触发panic。

Addr（）方法可以认为是Elem（）方法的逆操作，功能上等价于取地址操作，要求目标必须是可定址的，也就是有flagAddr标志，代码如下：

```
func (v Value) Addr() Value {
    if v.flag&flagAddr == 0 {
        panic("reflect.Value.Addr of unaddressable value")
    }
    fl := v.flag & flagRO
    return Value{v.typ.ptrTo(), v.ptr, fl | flag(Ptr)}
}
```

typ.ptrTo（）根据当前类型T得到了*T的元数据地址，新的flag就是kindPtr加上继承的只读标志位。ptr的值没有改变，这一点很重要，Value的相关方法会根据typ和flag来确定如何解释ptr。修改一下本节最开始的示例，看一下Elem（）方法和Addr（）方法逆操作的效果，代码如下：

```
//第5章/code_5_14.go
func main() {
    n := 6789
    v := reflect.ValueOf(&n)
    p := (*Value)(unsafe.Pointer(&v))
    println(p.typ, p.ptr, p.flag, toType(p.typ).String())
    e := v.Elem()
    p = (*Value)(unsafe.Pointer(&e))
    println(p.typ, p.ptr, p.flag, toType(p.typ).String())
    f := e.Addr()
    p = (*Value)(unsafe.Pointer(&f))
    println(p.typ, p.ptr, p.flag, toType(p.typ).String())
}
```

在笔者的计算机上得到的输出如下：

```
$ ./code_5_14.exe
0xc187c0 0xc00000c070 22 * int
0xc1c060 0xc00000c070 386 int
0xc187c0 0xc00000c070 22 * int
```

第2行输出的flag值是386，也就是kindInt、flagIndir、flagAddr组合的结果，再加上typ为int，与*int是等价的，可以互相转换，所以在调用json.Unmarshal（）之类的函数时，需要把struct实例的地址传进去，这样struct才是可定址的，函数内部才能为struct的字段赋值。

通过反射来操作数据，实际上也是围绕着类型元数据展开的，本节主要分析了Value各个字段的作用，以及比较重要的flagIndir和flagAddr这两个标志位。以此为起点，各位有兴趣的读者可以自行阅读reflect源码，以此来了解更多底层实现细节，本节就讲解到这里。

5.4.4 对链接器裁剪的影响

第4章在讲解方法的时候，我们发现了编译器会为接收者为值类型的方法生成接收者为指针类型的包装方法，经过本章的探索，我们知道这些包装方法主要是为了支持接口，但是如果反编译或者用nm命令来分析可执行文件，就会发现不只是这些包装方法，就连代码中的原始方法也不一定会存在于可执行文件中。这是怎么回事呢？

道理其实很简单，链接器在生成可执行文件的时候，会对所有OBJ文件中的函数、方法及类型元数据等进行统计分析，对于那些确定没有用到的数据，链接器会直接将其裁剪掉，以优化最终可执行文件的大小。看起来一切顺理成章，但是又有一个问题，反射是在运行阶段工作的，通过反射还可以调用方法，那么链接器是如何保证不把反射要用的方法给裁剪掉呢？

于是笔者就做了一个小小的实验，编译一个示例，代码如下：

```
//第 5 章/code_5_15.go
type Number float64

func (n Number) IntValue() int {
    return int(n)
}

func main() {
    n := Number(9)
    v := reflect.ValueOf(n)
    _ = v
}
```

然后用nm命令分析得到的可执行文件，命令如下：

```
$ go tool nm code_5_15.exe | grep Number
```

结果发现IntValue（）方法被裁剪掉了，对main（）函数稍做修改，代码如下：

```
//第 5 章/code_5_16.go
func main() {
    n := Number(9)
    v := reflect.ValueOf(n)
    v.MethodByName("")
    _ = v
}
```

再次编译并用nm命令检查，命令如下：

```
$ go tool nm code_5_16.exe | grep Number
  48f0c0 T main.(*Number).IntValue
  48efa0 T main.Number.IntValue
```

这次IntValue的两个方法都被保留了下来，如果换成v.Method（0）也能达到同样的效果。也就是说链接器裁剪的时候会检查用户代码是否会通过反射来调用方法，如果会就把该类型的方法保留下来，只有在明确确认这些方法在运行阶段不会被用到时，才可以安全地裁剪。

再次修改main（）函数的代码来进一步尝试，代码如下：

```
//第 5 章/code_5_17.go
func main() {
    n := Number(9)
    var a interface{} = n
    println(a)
    v := reflect.ValueOf("")
    v.MethodByName("")
}
```

发现这种情况下Number的两个方法依旧被保留了下来，从代码逻辑来看，运行阶段是不可能用到Number的方法的。再把main（）函数修改一下，代码如下：

```
//第 5 章/code_5_18.go
func main() {
    n := Number(9)
    println(n)
    v := reflect.ValueOf("")
    v.MethodByName("")
}
```

这次有所不同，Number的两个方法被裁剪掉了。由此可以总结出反射影响方法裁剪的两个必要条件：一是代码中存在从目标类型到接口类型的赋值操作，因为运行阶段类型信息萃取始于接口。二是代码中调用了MethodByName（）方法或Method（）方法。因为代码中有太多灵活的逻辑，编译阶段的分析无法做到尽如人意。

5.5 本章小结

本章以空接口interface{}为起点，初步介绍了Go语言的类型元数据，并且分析了数据指针带来的逃逸和装箱问题。非空接口部分，深入分析了实现方法动态派发的底层原理，还找到了编译器生成指针接收者包装方法的原因，即为了让接口方法调用更简单高效。还分析了组合式继承对方法集的影响，也是对非空接口的支持。类型断言分为4种场景共8种情况，分别通过反编译确认了汇编代码层面的实现原理。最后的反射部分，对类型系统进行了更深入的分析，并对反射如何操作数据进行了简单的探索。

接口，尤其是其背后的类型系统，有很多细节，本章无法全面地进行介绍。笔者只是把自己认为比较典型的问题拿出来分析一下，鼓励各位读者去源码中发现更多乐趣。

第6章
goroutine

本章的研究对象是Go语言最广为人知、最亮眼的特性，即goroutine，也就是我们俗称的协程。从本质上来讲，协程更像是一个用户态的线程，主要就是独立的用户栈加上几个关键寄存器的状态。事实上，这种技术早在多年以前就已经存在了，例如Windows NT的纤程（Fiber），起码已经存在了二十多年，但是一直不怎么受关注，几乎也没什么人使用。为什么到了Go语言中，协程就成了这么了不起的技术了呢？一方面是乘了互联网时代高并发场景的东风，另一方面（也是更关键的），就是和IO多路复用技术的巧妙结合。

因为在语言层面原生支持协程，让开发人员可以很轻松地应对高并发场景，使Go语言非常适合作为互联网时代的服务器端开发语言。那么协程到底是一种什么技术呢？为什么能够在目前的服务器端开发中大放异彩呢？让我们带着这些问题，展开本章的探索之旅。

6.1 进程、线程与协程

想要了解协程，还要从最早的进程说起，再到线程，最后是协程。对比之下才能更容易地理解这些技术是如何演进的。

6.1.1 进程

对于现代操作系统来讲，进程是一个非常基础的概念。进程包含了一组资源，其中有进程的唯一ID、虚拟地址空间、打开文件描述符表（或句柄表）等，还有至少一个线程，也就是主线程。最值得一提的就是虚拟地址空间，本书第1章在介绍汇编基础时，也简单地介绍了x86处理器的页表映射机制。现代操作系统利用硬件提供的页表机制，通过为不同进程分配独立的页表，实现进程间地址空间的隔离。如图6-1所示，不同进程的地址空间中相同的线性地址addr1，经过页表映射以后，最终会落到不同的物理页面，对应不同的物理地址。有了进程间地址空间的隔离，一些含有Bug或者恶意的程序就不能非法访问其他进程的内存了，这样才有安全性可言。

如果要创建一个新的进程，则操作系统需要进行哪些操作呢？以Linux为例，Linux通过clone系统调用来创建新的进程。clone会为新的进程分配对应的内核数据结构和内核栈，以及分配新的进程ID，然后复制打开文件描述符表、文件系统信息、信号处理器（Signal Handlers）、进程地址空间和命名空间。因为复制了父进程的打开文件描述符表，所以子进程可以很方便地继承父进程已经打开的文件、socket等资源，使像nginx、php-fpm这种多进程的工作模式能够比较方便地实现。

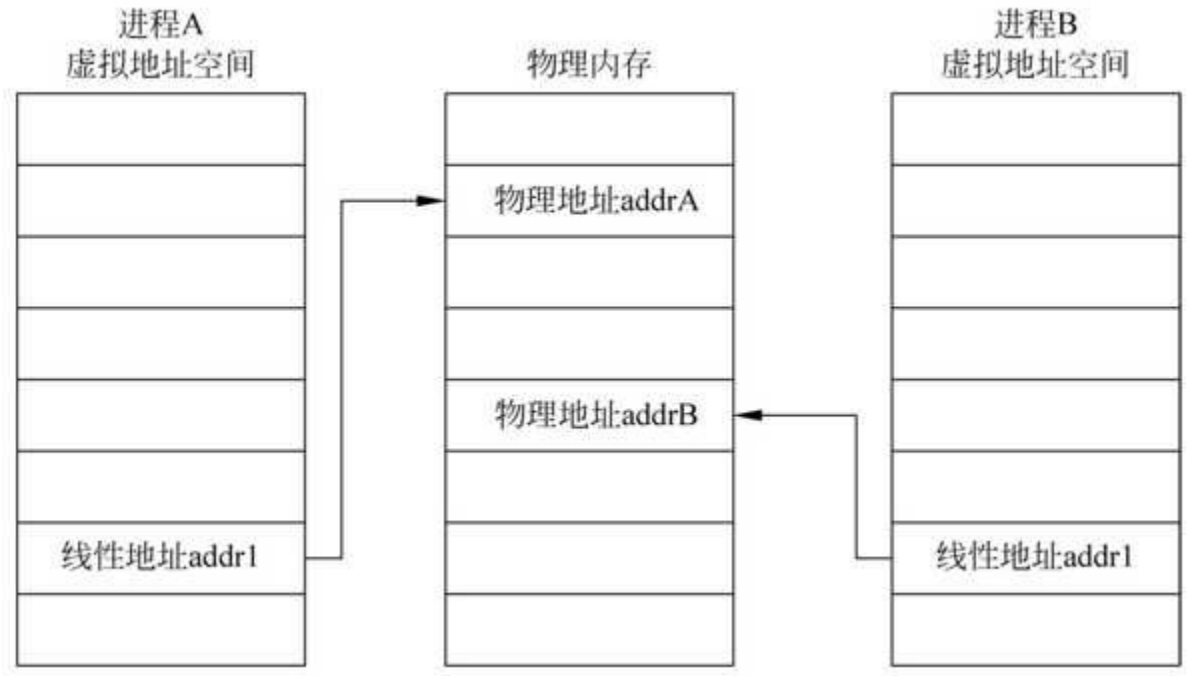

图6-1 进程间地址空间的隔离

操作系统在复制进程的地址空间时，基于Copy on Write技术，避免了不必要的内存复制，但是新进程还是需要有独立的页表，因此创建大量进程时首先会造成内存方面的显著开销，而后操作系统在进行调度的时候，切换进程需要同步切换页表，页目录寄存器一经修改，TLB缓存也随即失效，造成地址转换效率降低，进一步影响性能，所以在技术演进迭代的过程中，多进程模式很快就遇到了瓶颈，无法充分发挥CPU的计算能力，然而多任务的大趋势是不可阻挡的，于是多线程技术应运而生。

6.1.2 线程

如果理解了进程的组成，再来看多线程就很容易理解了。原本单线程的进程中只有一个主线程，主线程再通过线程API创建出其他的线程，这就是所谓的多线程模式了。

在多线程模式下，进程的打开文件描述符表、文件系统信息、虚拟地址空间和命名空间是被进程内的所有线程共享的，但是每个线程拥有自己的内核数据结构、内核栈和用户栈，以及信号处理器。

线程是进程中的执行体，如图6-2所示，为什么要有一个用户栈和一个内核栈呢？因为我们的线程在执行过程中经常需要在用户态和内核态之间切换，通过系统调用进入内核态使用系统资源。

对于内核来讲，任何的用户代码都被视为不安全的，可能有Bug或者带有恶意的代码，所以操作系统不允许用户态的代码访问内核数据。线程进入内核态之后执行的是内核提供的代码，也就是安全的受信任的代码，但是如果跟用户态代码共用一个栈就会留下安全漏洞，栈上的数据可能会被用户程序非法读取和篡改，所以要给内核态分配单独的栈，用户态的程序无法访问内核栈。

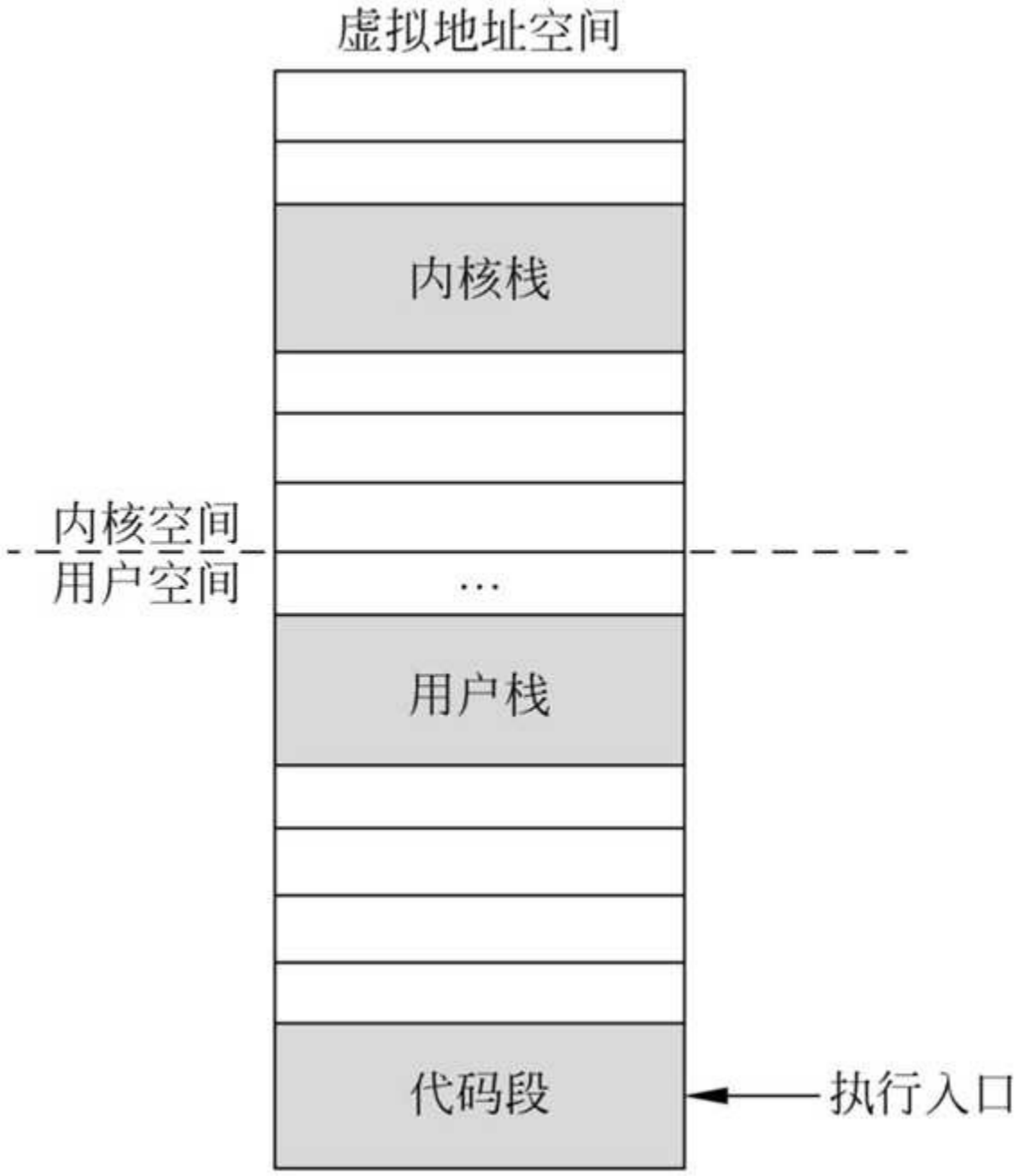

图6-2 线程的用户栈和内核栈

调度系统切换线程时，如果两个线程属于同一个进程，开销要比属于不同进程时小得多，如图6-3所示。因为不需要切换页表，相应地，TLB缓存也就不会失效。同一个进程中的多个线程，因为共

享同一个虚拟地址空间，所以线程间数据共享变得十分简单高效，只要做好同步就不会有太大问题，因此，与多进程模式相比，多线程模式大幅优化了性能，系统的吞吐量也随之显著提升。

但是随着并发量的不断增大，应用程序需要创建越来越多的线程，当系统的线程数量达到十万或百万级别时，系统又将遭遇性能瓶颈。一方面，线程的内核数据结构、内核栈和用户栈会占用大量的内存，在线程数量庞大时尤其显著。另一方面，操作系统基于时间片策略来调度所有的线程，在如此庞大的线程数量下，为了尽量降低延迟，线程每次得以运行的时间片会被压缩，从而造成线程切换频率增高。如果是IO密集型的应用，就会有更多的切换发生，多数时候还没有用完时间片，就因为IO等待而挂起了。调度系统切换线程的上下文，本身是有一定开销的，在线程数量适中、时间片足够大时，切换的频率相对较低，这部分开销可以忽略不计。在线程切换频繁时，调度本身的开销会占用大量CPU资源，造成系统吞吐量严重下降。

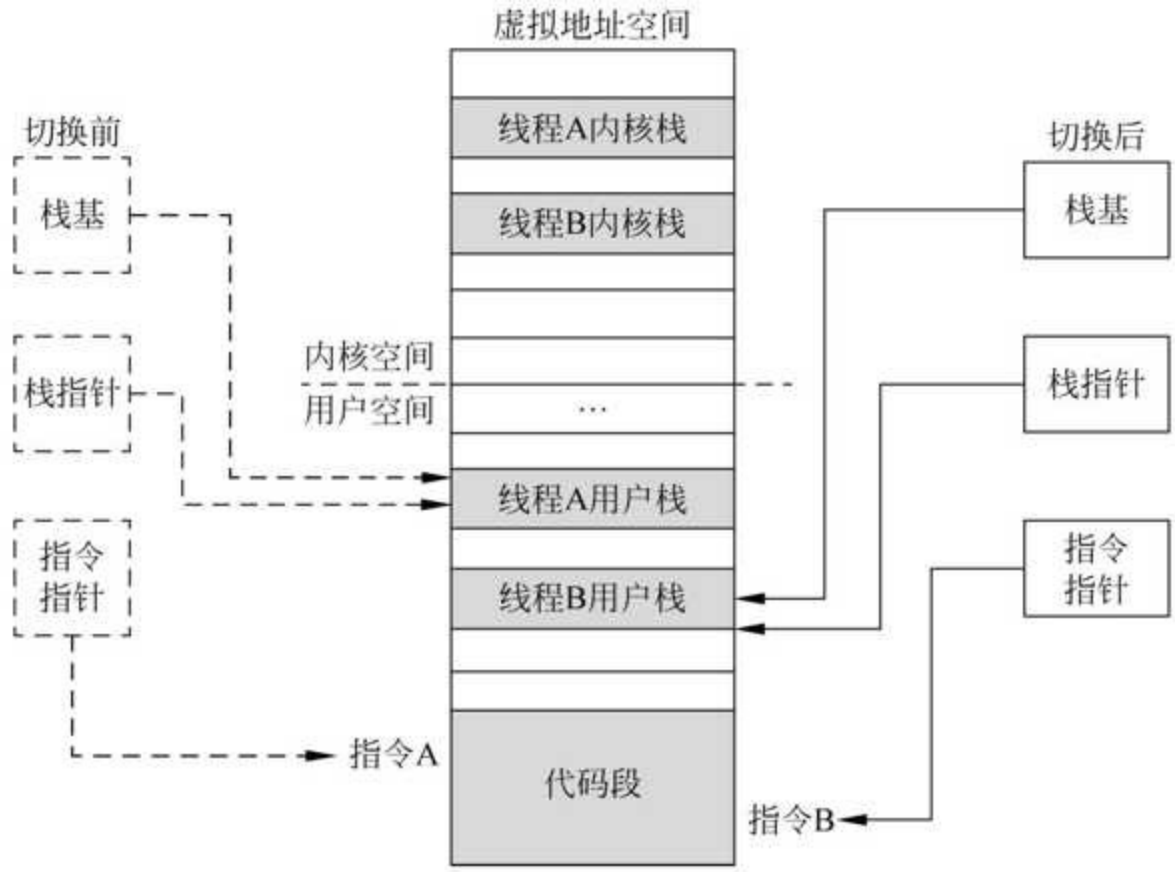

图6-3 同进程间线程切换

问题越来越明了了，看起来我们需要一种更轻量的线程，在设计上需要满足两方面的要求：一是节省内存空间，让主流服务器的内存大小能够轻松装载十万或百万级这种轻量线程。二是调度代价低，也就是切换起来更轻快。因为高并发的场景就摆在那里，我们就是需要创建大量线程，并且要求频繁地切换。有了如此明确的需求，协程就被创造出来了。

6.1.3 协程

笔者最初接触到的协程实现是Windows NT提供的纤程，从开发者文档中发现了关于Fiber的那组API。Fiber设计得非常有意思，它是完全在用户态实现的，所以不需要对系统内核作任何改动，内核层面并不知道有纤程这种东西的存在。就这一点来讲，goroutine也是一样的，不与系统内核耦合。6.1.2节简单介绍了线程的组成，对比着来看纤程，可以认为它就是基于线程的用户态部分做了一些改造。原本的线程有一个用户栈和一个内核栈，一个单线的执行逻辑在用户态和内核态之间跳跃。对比来看，纤程只有用户栈（没有内核栈），并且调度相关的数据结构也存储在用户空间中。线程是被操作系统调度的，主要基于时间片策略进行抢占式调度，而纤程是完全在用户空间实现的，要靠主动让出的方式来切换。

从具体实现来看，纤程就是一个由入口函数地址、参数和独立的用户栈组成的任务，相当于让线程可以有多个用户栈，如图6-4所示，在每个用户栈上执行不同的任务。线程能够修改自己的栈指针寄存器，不仅可以上下移动，还可以直接切换到新的栈，所以实现起来并不困难。有一点需要注意的是，线程的用户栈是由操作系统负责管理的，一般会预留较大的空间，然后按照实际使用情况逐渐分配、映射，而纤程（协程）的栈，需要由用户程序自己来管理。

与纤程相关的API已经存在了二十多年，但是很少见到相关的应用案例。为什么一直不温不火呢？可以从两个方面简单地思考一下。一是新技术本身的易用性与可用性，二是应用新技术后能带来的效益提升。

从易用性来看，系统提供的纤程API实现了创建、销毁、切换等基本功能，而实际的调度策略需要开发者自己实现，还是有一定的复杂性的。从效益方面来看，也没有太大诱惑力。我们把计算机执行的任务分成CPU密集型和IO密集型，CPU密集型任务一般更看重吞吐量，所以要尽量减少上下文切换，每次直接用完时间片就好了，似乎没有纤程的用武之地，而IO密集型任务，可能会更看重响应延迟，例如互联网应用的网关，但是当时主要的网络IO模型还是阻塞式IO，动不动直接就让线程挂起了，也没给纤程留下什么发挥的空间，所以在很长一段时间里，像纤程这种协程技术，更像是一个实验性质的模型，没有得到太广泛的应用，直到协程遇到了IO多路复用。

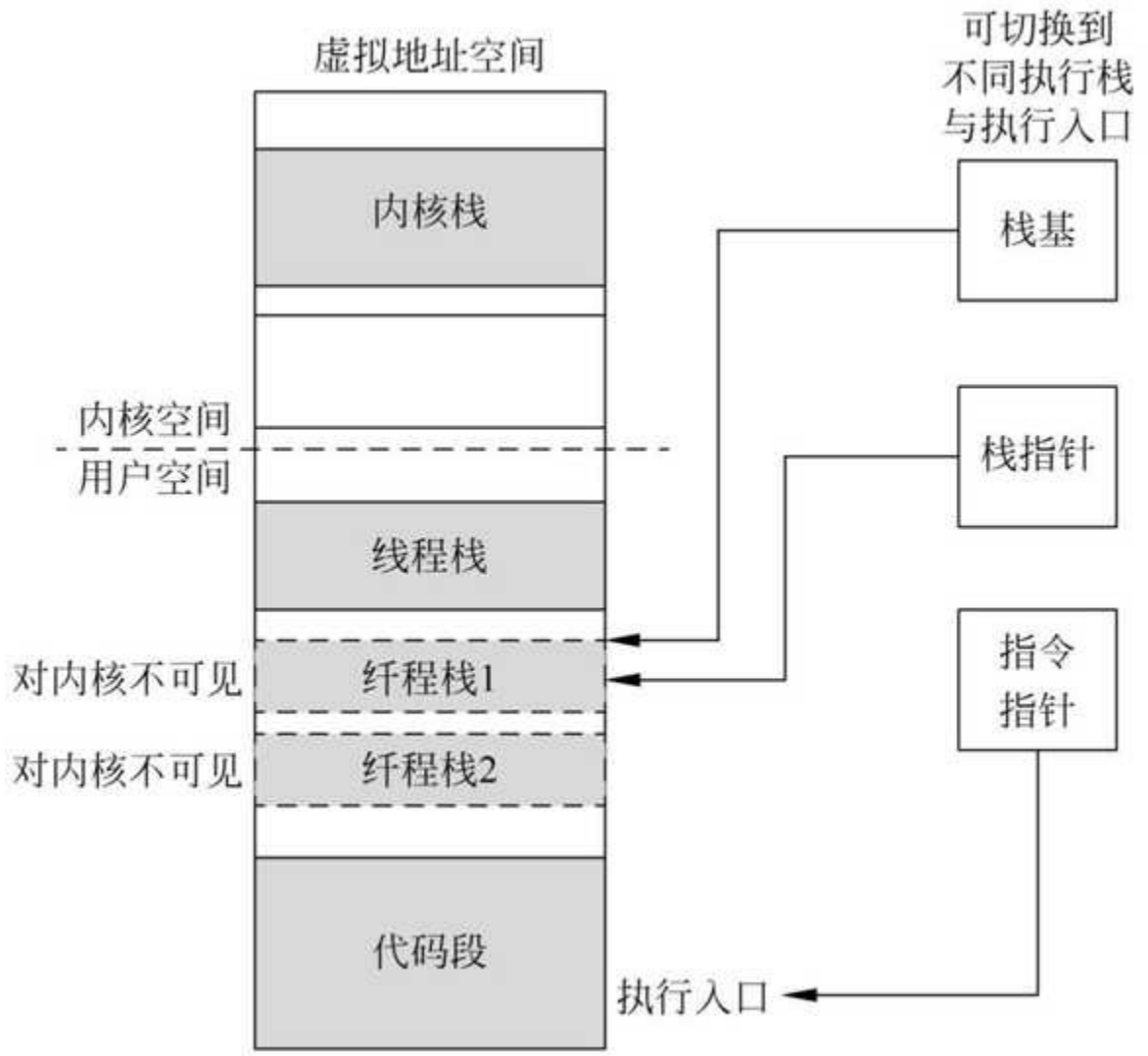

图6-4 纤程概念示意图

6.2 IO多路复用

提到IO多路复用技术，现在已经是老生常谈的技术了。从早期的select、poll，到后来的epoll、kqueue、event port，这门技术已经发展得非常成熟。应该有很多人是从nginx开始了解IO多路复用技术的，当然也可能是redis、nio等。本章主要研究goroutine，为什么要把IO多路复用拿出来讲解呢？因为Go语言是集协程思想和IO多路复用技术之大成者，复杂烦琐的事情都由runtime去处理了，极大地方便了开发者。那么IO多路复用到底是一种什么样的技术呢？它又解决了什么问题呢？接下来就带着这两个疑问，概括地了解IO多路复用技术。

早年的服务器程序都是以阻塞式IO来处理网络请求的，造成的最大问题就是会让线程挂起，直至IO完成才会恢复运行。在这种技术背景下，开发者需要为每个请求创建一个线程，线程数会随着并发等级直线增加，进而造成系统不堪重负。一个解决思路就是把请求和线程解耦，不要让请求绑定到一个线程或占用一个线程，然后用线程池之类的技术控制线程的数量。阻塞式IO显然不能满足这种需求，可以考虑使用非阻塞式IO或IO多路复用。下面就来对比一下这三者的不同。

6.2.1 3种网络IO模型

参考《UNIX网络编程》一书，我们把一个常见的TCP socket的recv请求分成两个阶段：一是等待数据阶段，等待网络数据就绪；二是数据复制阶段，把数据从内核空间复制到用户空间。对于阻塞式IO来讲，整个IO过程是一直阻塞的，直至这两个阶段都完成。UNIX系统上的socket默认工作在阻塞模式下，经典的阻塞式网络IO模型如图6-5所示。

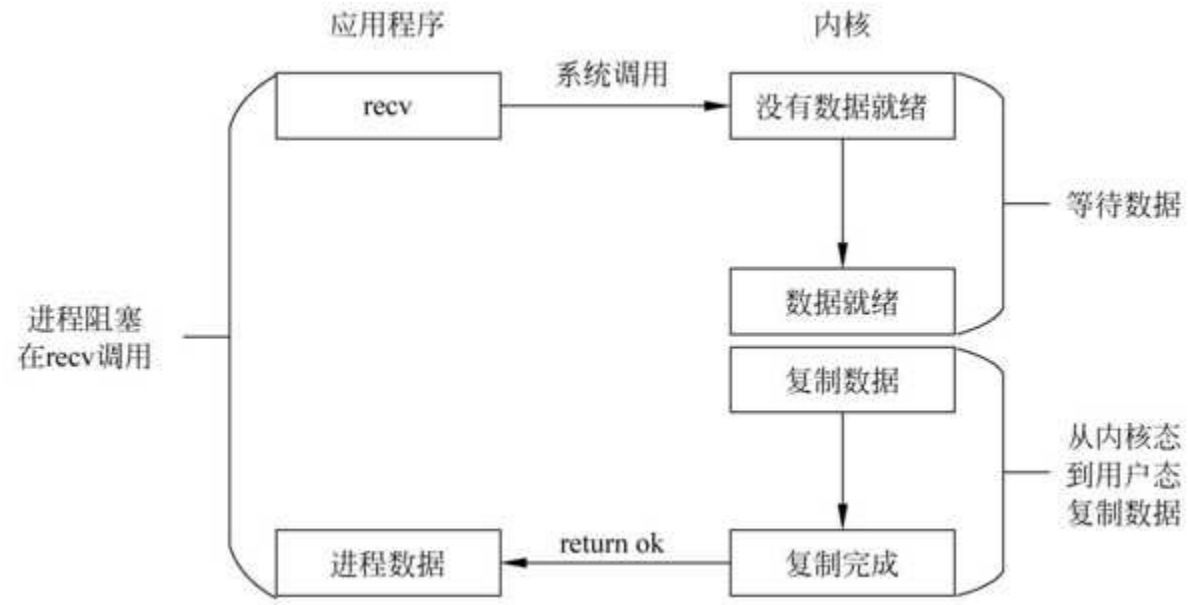

图6-5 经典的阻塞式网络IO模型

如果想要启用非阻塞式IO，需要在代码中使用fcntl（）函数将对应socket的描述符设置成O_NONBLOCK模式。非阻塞式网络IO模型如图6-6所示，与阻塞模式的不同之处主要体现在第一阶段，即等待数据阶段。在非阻塞模式下，线程等待数据的时候不会阻塞，从编程角度来看就是recv（）函数会立即返回，并返回错误代码EWOULDBLOCK（某些平台的SDK也可能是EAGAIN），表明此时数据尚未就绪，可以先去执行别的任务。程序一般会以合适的频率重复调用recv（）函数，也就是进行轮询操作。在数据就绪之前，recv（）函数会一直返回错误代码EWOULDBLOCK。等到数据就绪后，再进入复制数据阶段，从内核空间到用户空间。因为非阻塞模式下的数据复制也是同步进行的，所以可以认为第二阶段也是阻塞的。总之，与阻塞式IO相比，这里只有第二阶段是阻塞的。

非阻塞式IO看起来比阻塞式要强多了，因为网络的延迟相对比较高，与计算机执行一两个函数花费的时间根本不在一个数量级，因此在整个IO操作过程中，第二阶段的耗时跟第一阶段相比几乎

是无足轻重的。那么有了非阻塞式IO是不是就万事大吉了呢？实则不然，从图6-6就可以看出来，虽然第一阶段不会阻塞，但是需要频繁地进行轮询。一次轮询就是一次系统调用，如果轮询的频率过高就会空耗CPU，造成大量的额外开销，如果轮询频率过低，就会造成数据处理不及时，进而使任务的整体耗时增加。

IO多路复用技术就是为解决上述问题而诞生的，如图6-7所示，IO多路复用集阻塞式与非阻塞式之所长。与非阻塞式IO相似，从socket读写数据不会造成线程挂起。在此基础之上把针对单个socket的轮询改造成了批量的poll操作，可以通过设置超时时间选择是否阻塞等待。只要批量socket中有一个就绪了，阻塞挂起的线程就会被唤醒，进而去执行后续的数据复制操作。

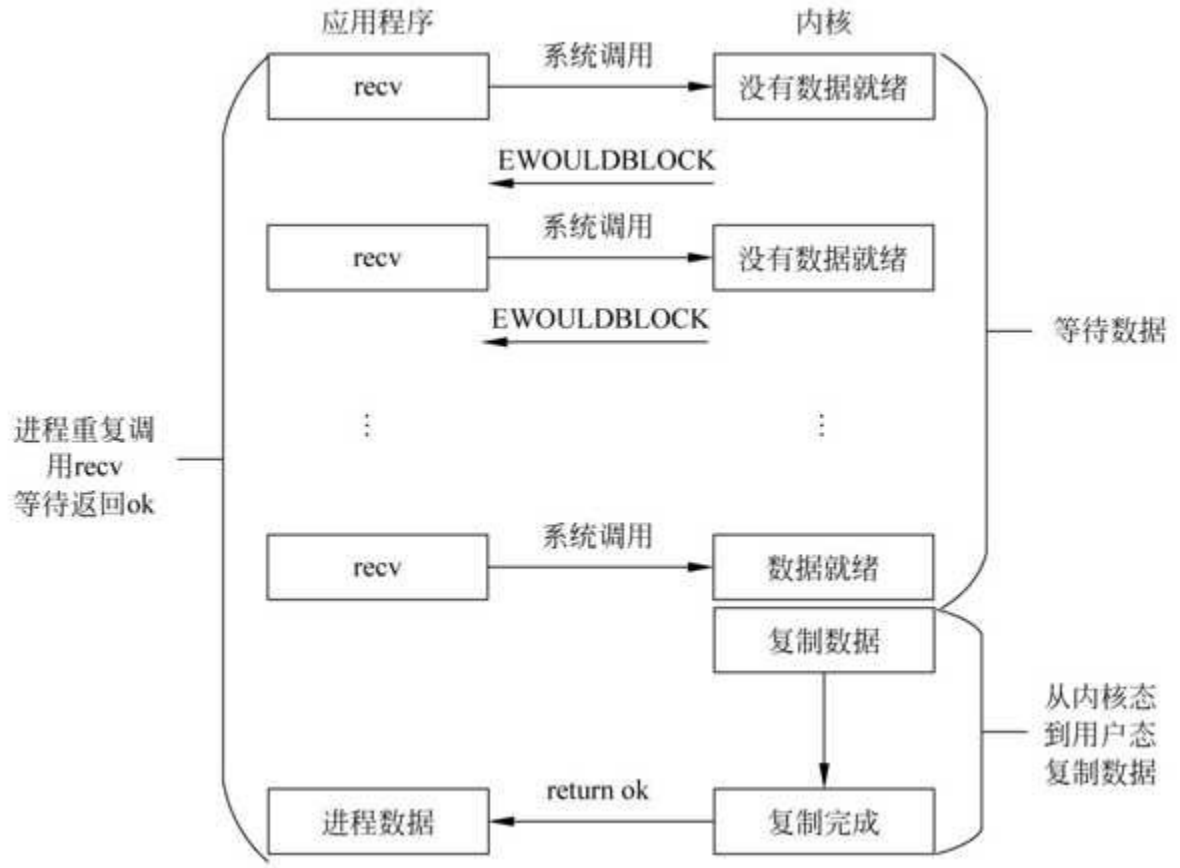

图6-6 非阻塞式网络IO模型

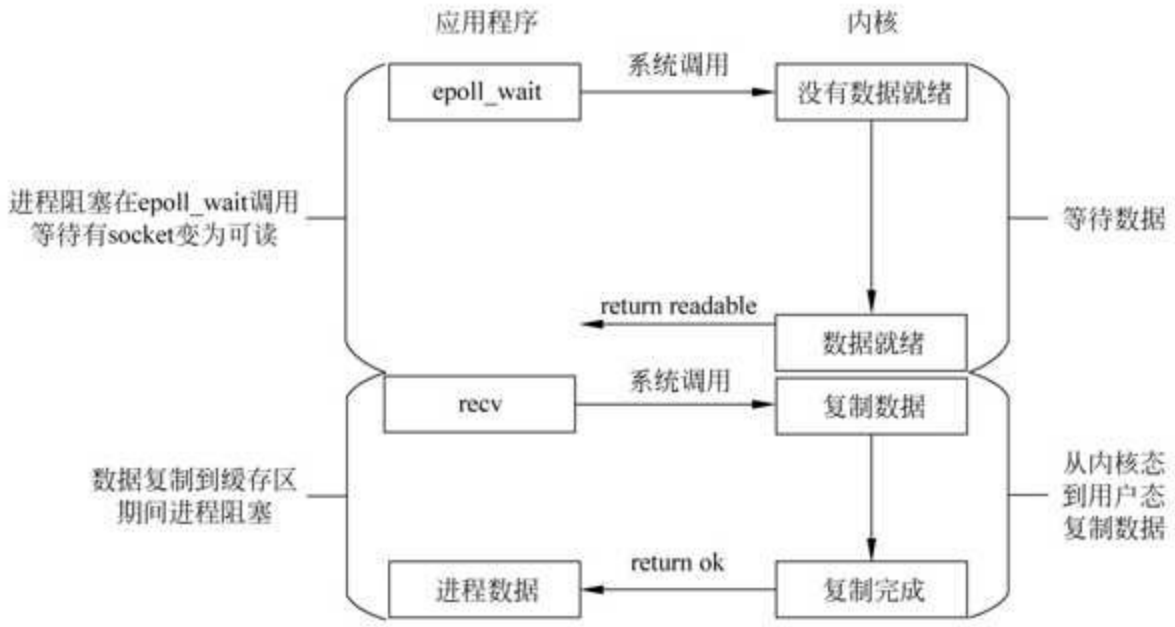

图6-7 IO多路复用

就拿Linux上的epoll来讲，在实际编程时，对指定的socket进行读或写操作之前，会先通过epoll_ctl（）函数把socket的描述符添加到epoll中，然后通过epoll_wait（）函数进行监听等待，等到其中的socket变成可读、可写时，epoll_wait（）函数就会返回。因为epoll是批量监听的，所以要比阻塞式IO单个等待高效很多。至于是监听socket可读还是可写，要看epoll_ctl（）函数添加描述符时指定的事件参数，示例代码如下：

```
struct epoll_event evt = {0};
evt.events = EPOLLIN;
evt.data.fd = fd;
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &evt);
```

这里就是把描述符fd添加到epfd这个epoll实例中，其中的EPOLLIN表明要监听的是可读事件。把EPOLLIN换成EPOLLOUT就可以监听可写事件了。epoll_event结构的代码如下：

```
typedef union epoll_data {
    void        *ptr;
    int         fd;
    uint32_t    u32;
    uint64_t    u64;
} epoll_data_t;

struct epoll_event {
    uint32_t    events;      /* Epoll events */
    epoll_data_t data;       /* User data variable */
};
```

其中epoll_data_t类型的data字段是给开发者用的，用来存放开发者自定义的数据。等到对应的socket有IO事件触发时，这些数据会被epoll_wait（）函数返回。epoll_wait（）函数的原型如下：

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

events参数指向一段可以容纳maxevents个epoll_event结构的内存，这段内存是由开发者来分配的，epoll_wait（）函数会利用这段内存返回一组epoll_event结构。返回的epoll_event结构的events字段代表具体发生的IO事件，data字段是由开发者自定义的数据，开发者需要通过它来找到与IO事件关联的socket。更多具体细节可参阅Linux开发者手册Section 2。

这里需要注意的是，如何理解一个socket的可读、可写状态呢？就TCP通信来讲，每个socket都有自己配套的收、发缓冲区，发送数据的时候调用send（）函数，实际上先把数据写到了socket的发送缓冲区中，系统会在合适的时机把数据发送给远程的对端，然后清空socket的发送缓冲区。同理，对端发送过来的数据会被系统自动存放到socket的接收缓冲区，等待应用程序通过recv（）函数来读取。通俗地讲，当发送缓冲区被写满的时候，自然不能继续写入数据，此时的socket是不可写的。等到系统把数据发送出去并在发送缓冲区中腾出空间时，socket就变成了可写的了。同理，当接收缓冲区中没有任何数据时，socket是不可读的，等到系统收到了远程对端发送的数据并把数据存放到socket的接收缓冲区后，socket就变成可读的了。通过epoll高效地监听批量socket的状态，避免了非阻塞式IO频繁轮询地空耗CPU，又不会像阻塞式IO那样每个socket挂起一个线程，从而大大提高了服务器程序的运行效率。

下面就用一个简单的HTTP GET请求，来实际对比阻塞式IO和基于epoll的IO多路复用有什么差异。

6.2.2 示例对比

我们站在客户端的视角，去除掉不太相关的细节，从TCP连接的建立开始梳理。先来梳理阻塞式IO。

1．阻塞式**IO**下的**GET**请求

阻塞式网络IO的主要流程如下：

（1）客户端通过connect（）函数发起连接，此时线程会被挂起等待，直到三次握手完成、连接成功建立（或者出现错误）以后，connect（）函数才会返回，线程继续执行。

（2）客户端通过send（）函数发送HTTP请求报文，因为GET请求报文一般很小，socket的发送缓冲区足以装载这些数据，所以线程一般不会阻塞。

（3）通过recv（）函数读取服务器端返回的数据，因为网络通信的延迟与程序指令执行耗时根本不在一个数量级，所以在这时接收缓冲区内数据尚未就绪，线程一般会阻塞。

（4）等到数据从服务器端到达客户端后，recv（）函数完成数据的复制（从内核空间到用户空间），并返回，然后上层的HTTP协议处理数据，判断传输是否完成，如果未完成，则重复执行第（3）步，直至传输完成，然后连接可能会被关闭或复用，这个我们就不关心了。

整体流程如图6-8所示，可以发现，阻塞式IO的逻辑非常清晰，只有单一的一条线，是平铺式的、顺序执行的。代码写起来很简单，后续也便于维护，只是执行效率不是很高，无法充分发挥服务器硬件的能力。

2．应用**epoll**的**GET**请求

接下来再看一下运用epoll时，一个GET请求是如何执行的。整体流程如图6-9所示。

（1）先通过fcntl（）函数把要用来发起连接的socket设置成O_NONBLOCK模式，然后使用connect（）函数发起连接。因为socket是非阻塞的，所以connect（）函数会立即返回EINPROGRESS，表示连接正在建立中。

（2）因为我们接下来要发送请求报文，要保证socket是可写的，所以就用epoll_ctl（）函数指定EPOLLOUT事件把socket描述符添加到epoll中，然后调用epoll_wait（）函数等待连接就绪。

（3）连接建立完成后，socket的发送缓冲区是空的，也就是可写的，所以epoll_wait（）函数会成功返回，上层的HTTP协议负责完成请求报文的发送，假设报文较小，只需一次send（）函数调用。

（4）接下来需要接收服务器端返回的结果，要保证socket是可读的，将epoll中socket对应的描述符改为监听EPOLLIN事件。

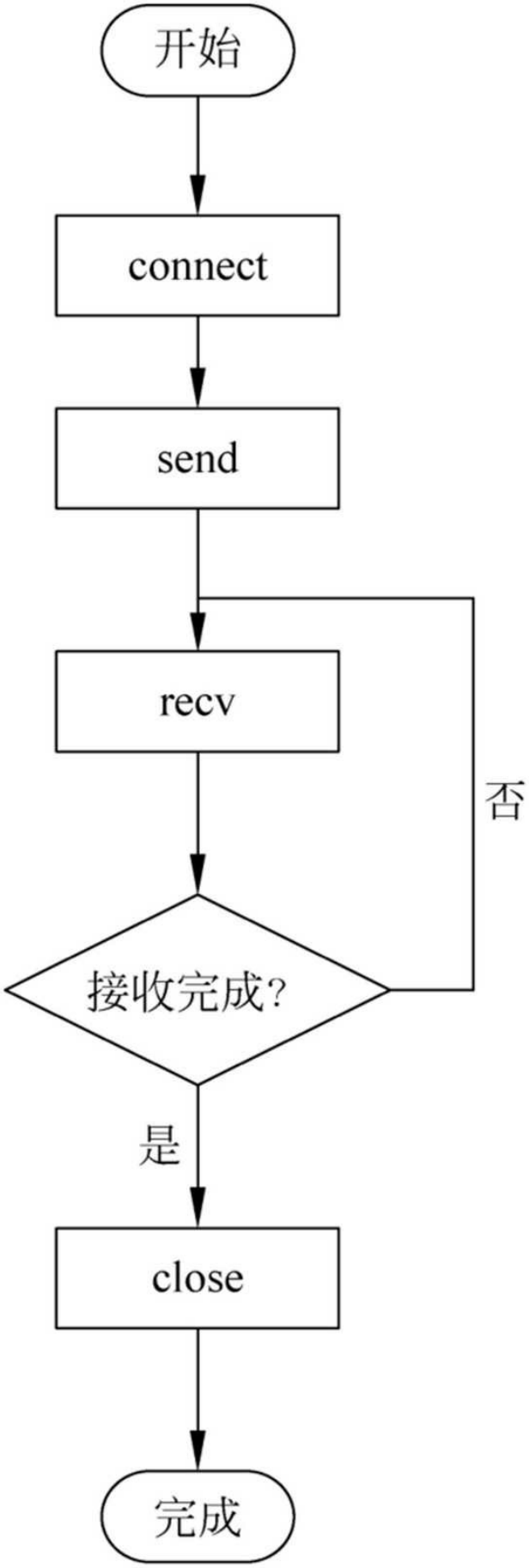
开始
connect
send
recv
否
接收完成?
是
close
完成

图6-8 阻塞式IO下一个HTTP GET请求的处理流程

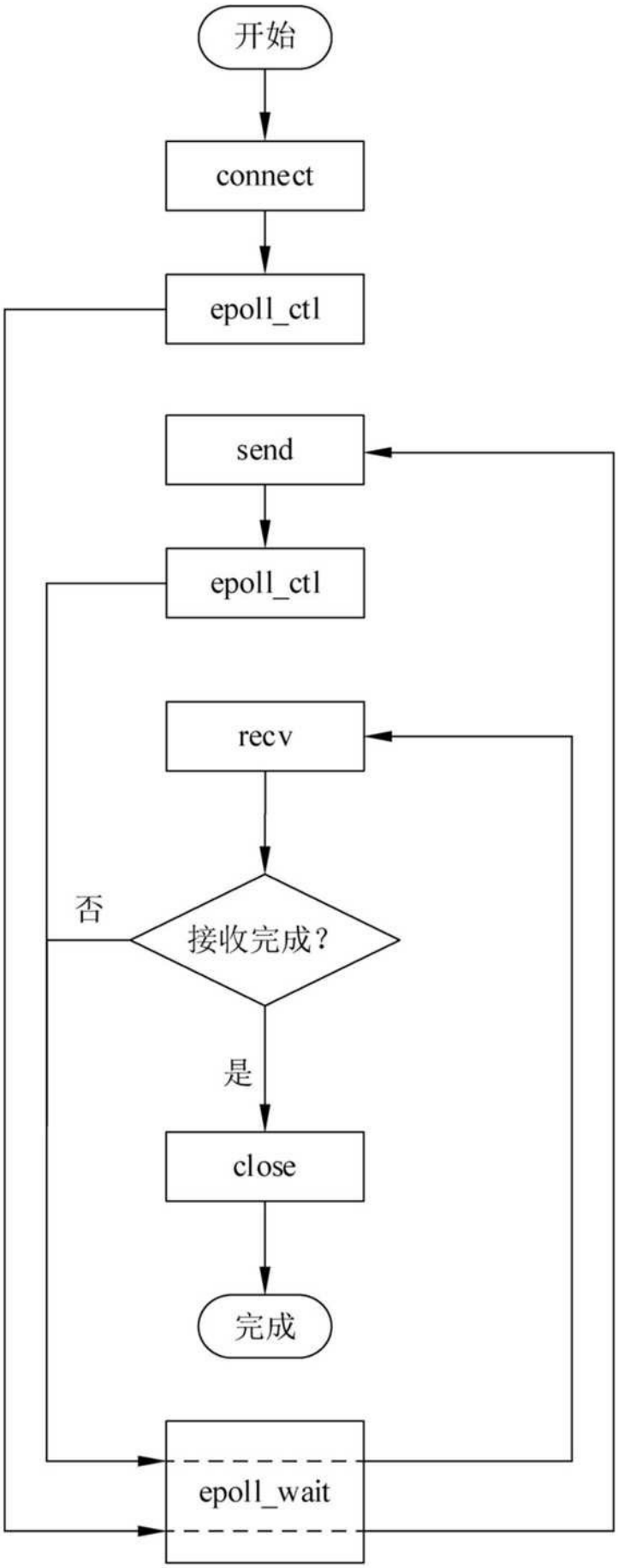
开始
connect
epoll_ctl
send
epoll_ctl
recv
否
接收完成？
是
close
完成
epoll_wait

图6-9 应用epoll的GET请求

（5）调用epoll_wait（）函数等待数据就绪，当服务器端的数据包到达客户端后，epoll_wait（）函数会成功返回，程序通过recv（）函数读取接收缓冲区中的数据。HTTP协议会判断传输是否完成，未完成则重复执行本步骤。

基于epoll的处理逻辑就不像阻塞式IO那样简明了，人们一般称之为IO事件循环。整个事件循环重复地执行epoll_wait（）函数，每次epoll_wait（）函数会返回一组已触发的epoll_event事件，其中有些是可读事件，有些是可写事件（还可能有一些错误事件，这里暂且忽略）。上层协议需要遍历每个epoll_event事件来处理与之关联的socket，因此还要记录与socket关联的请求的处理状态，例如有的socket处于连接建立状态，接下来要发送请求报文，还有的socket读取服务器端返回的数据读到一半，等下次可读时还要继续读取。epoll_event结构中的data字段用来存储这些信息，我们在通过epoll_ctl（）函数向epoll中添加socket描述符的时候，会把该socket相关的状态数据都存储在一个结构体中，并把该结构体的地址赋值给data.ptr，然后与socket的描述符一起添加到epoll中。于是，程序代码的逻辑就像状态机，状态的转移由IO事件与上层协议逻辑共同决定。

综上所述，IO多路复用的出现确实大大提升了应用程序的网络IO效率，对于高并发量的服务器端程序来讲，改善尤为明显。带来的问题就是显著提升了编程的难度，按照事件循环的方式实现复杂的应用逻辑非常烦琐。虽然后来催生了一些事件库来方便开发者进行开发，形成了一种基于回调函数的编程风格，但还是不够直观和方便。能否有一种技术，让我们既能够像阻塞式IO那样平铺直叙地书写代码逻辑，又能兼得IO多路复用这样的高性能呢？

6.3 巧妙结合

6.2节中不止一次地提到了IO事件循环，循环也就意味着每次执行之后都会回到原点，从程序执行的底层来看，也就是指令指针和栈指针的还原。因为一个socket（请求）的生命周期往往要跨多轮循环，所以循环内部不能在栈上存储socket的状态信息。这其实很好理解，因为IO事件的触发是随机的，因此每次可读、写的一组socket也是随机的，而栈帧的分配与释放是有严格顺序的，所以无法把socket的状态存储到栈帧上。

如果为每个socket（请求）分配一个独立的栈是不是就可以了呢？此时应该已经很自然地想到协程，把每个网络请求放到一个单独的协程中去处理，底层的IO事件循环在处理不同的socket时直接切换到与之关联的协程栈，如图6-10所示。

这样一来，就把IO事件循环隐藏到了runtime内部，开发者可以像阻塞式IO那样平铺直叙地书写代码逻辑，尽情地把数据存放在栈帧上的局部变量中，代码执行网络IO时直接触发协程切换，切换到下一个网络数据已经就绪的协程。当底层的IO事件循环完成本轮所有协程的处理后，再次执行netpoll，如此循环往复，开发者不会有任何感知，程序却得以高效执行。

关于协程的分析就到这里，Go语言中的协程调度并不只是基于IO事件的，只是笔者认为协程与IO多路复用这两种技术的结合确实非常巧妙，对于目前的服务器端编程语言、编程框架来讲，应该可以称得上是最关键的技术了。当然，读者也可以有不同的观点。本章接下来的内容，我们将会围绕goroutine的调度模型进行更加深入的探索。

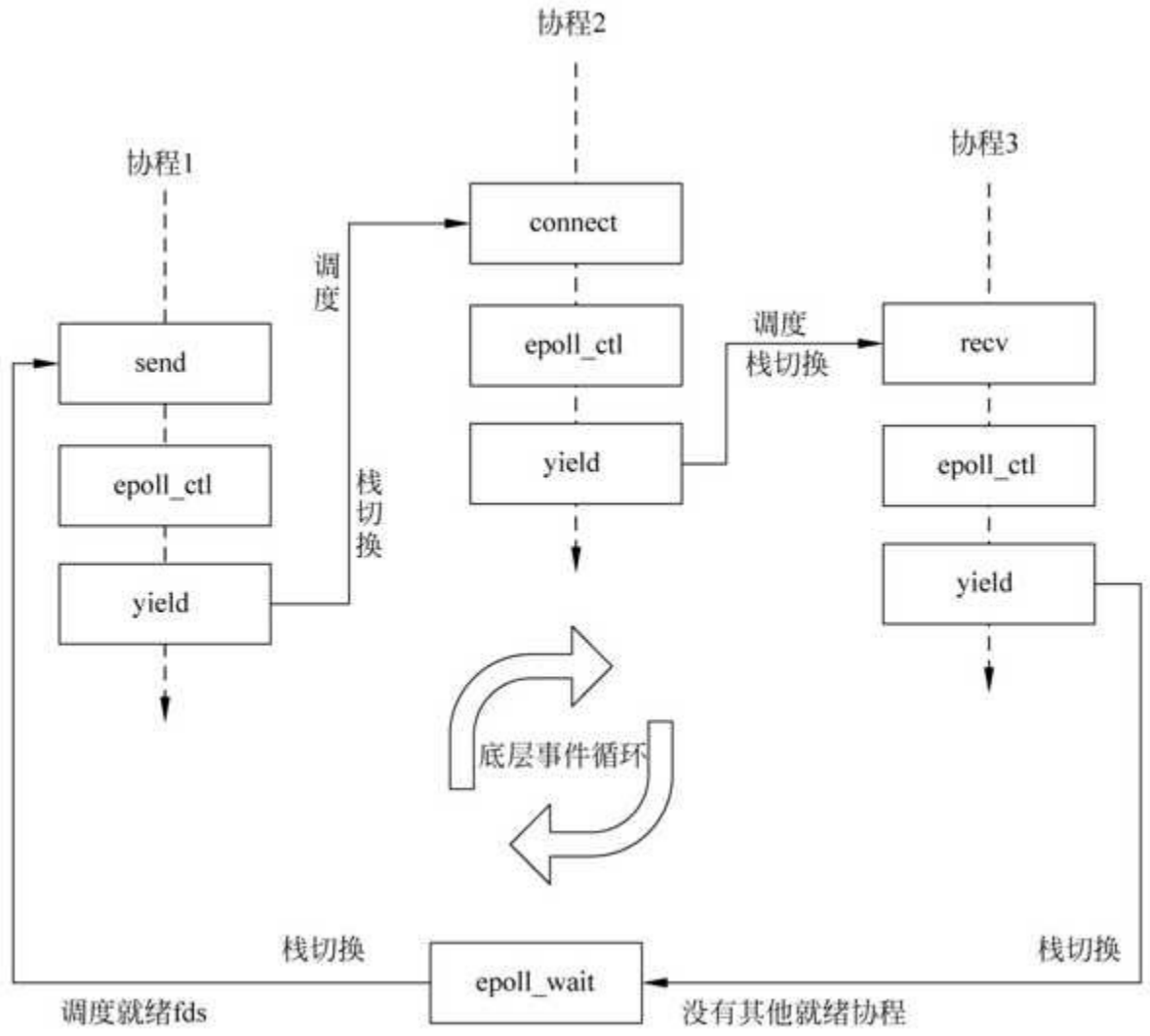

图6-10 协程与IO多路复用的结合

6.4 GMP模型

6.4.1 基本概念

说到Go语言的调度系统，GMP调度模型经常被提起。其中的G指的就是goroutine；M是Machine的缩写，指的是工作线程；P则是指处理器Processor，代表了一组资源，M要想执行G的代码，必须持有一个P才行。

简单来讲GMP就是Task、Worker和Resource的关系，G和P都是Go语言实现的抽象度更高的组件，而对于工作线程而言，Machine一词表明了它与具体的操作系统、平台密切相关，对具体平台的适配、特殊处理等大多在这一层实现。

6.4.2 从GM到GMP

在早期版本的Go实现中（1.1版本之前），是没有P的，只有G和M，GM模型如图6-11所示。

后来为什么要引入一个P呢？主要因为GM调度模型有几个明显的问题：

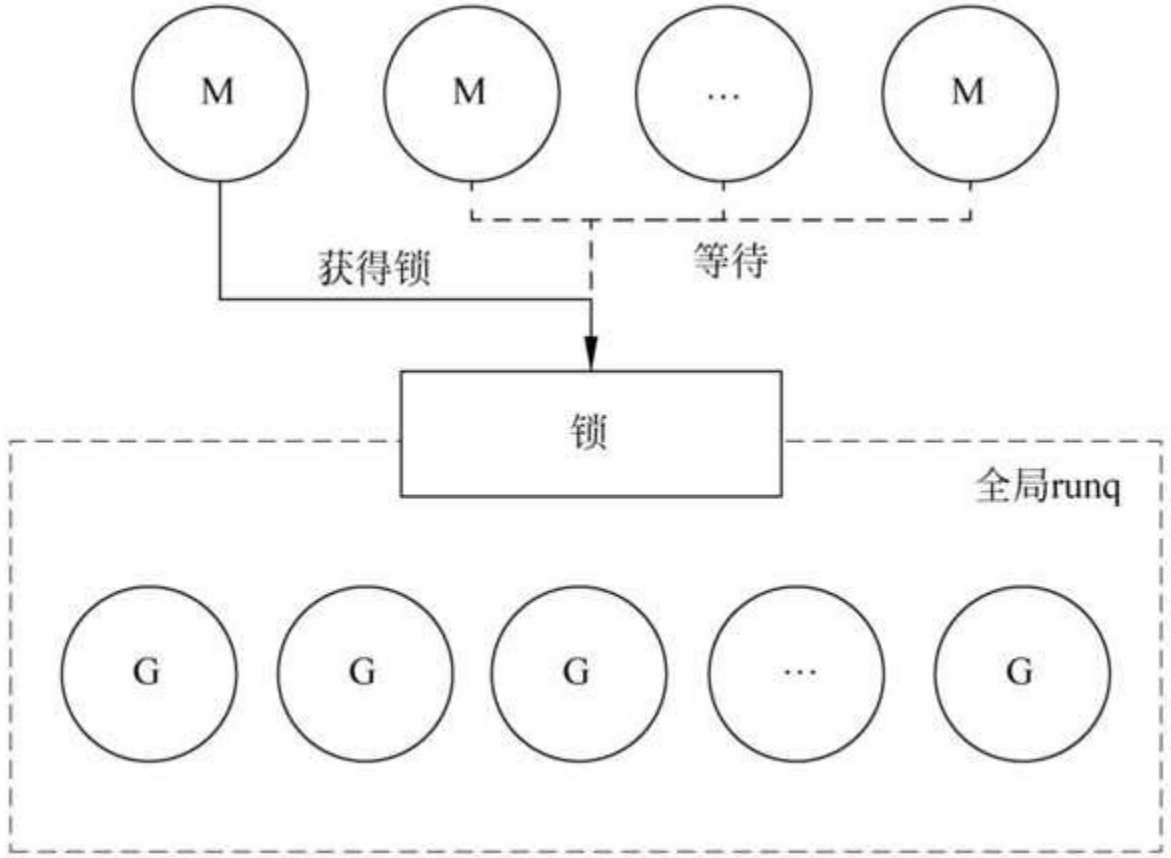

图6-11 GM调度模型

（1）用一个全局的mutex保护着一个全局的runq（就绪队列），所有goroutine的创建、结束，以及调度等操作都要先获得锁，造成对锁的争用异常严重。根据Go官方的测试，在一台CPU使用率约为70%的8核心服务器上，锁的消耗占比约为14%。

（2）G的每次执行都会被分发到随机的M上，造成在不同M之间频繁切换，破坏了程序的局部性，主要原因也是因为只有一个全局的runq。例如在一个chan上互相唤醒的两个goroutine就会面临这种问题。还有一点就是新创建的G会被创建它的M放入全局runq中，但是会被另一个M调度执行，也会造成不必要的开销。

（3）每个M都会关联一个内存分配缓存mcache，造成了大量的内存开销，进一步使数据的局部性变差。实际上只有执行Go代码的M才真地需要mcache，那些阻塞在系统调用中的M根本不需要，而实际执行Go代码的M可能仅占M总数的1%。

（4）在存在系统调用的情况下，工作线程经常被阻塞和解除阻塞，从而增加了很多开销。

为了解决上述这些问题，新的调度器被设计出来。总体的优化思路就是将处理器P的概念引入runtime，并在P之上实现工作窃取调度程序。M仍旧是工作线程，P表示执行Go代码所需的资源。当一个M在执行Go代码时，它需要有一个关联的P，当M执行系统调用或者空闲时，则不需要P。GMP调度模型如图6-12所示。

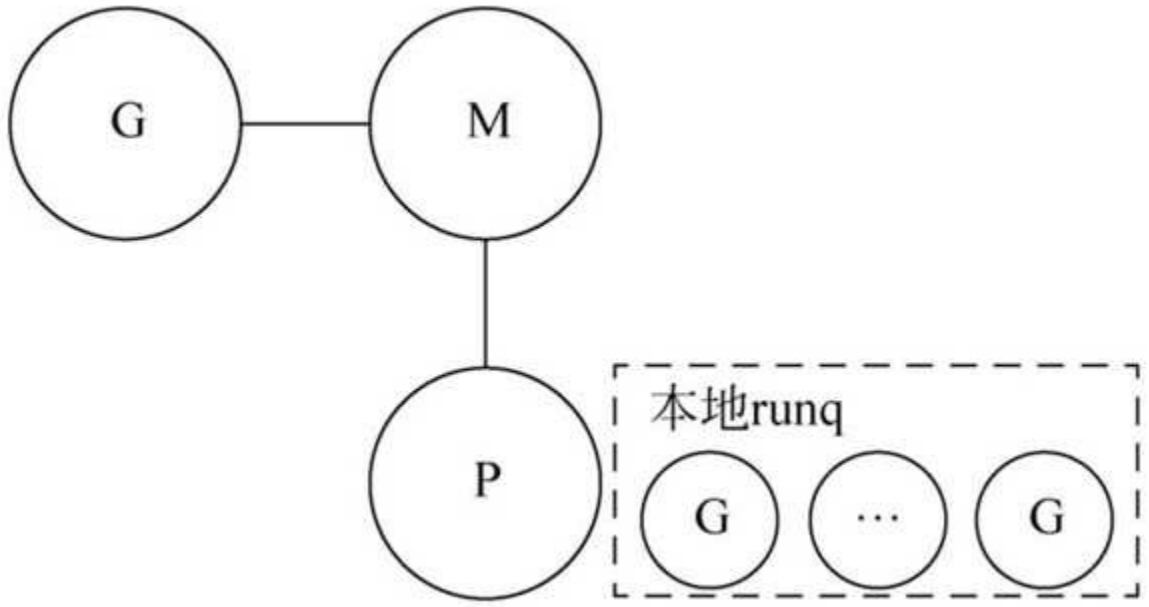

图6-12 GMP调度模型

通过GOMAXPROCS可以精确地控制P的个数，为了支持工作窃取机制，所有的P被放在同一个数组中，GOMAXPROCS的变动需要Stop/Start The World来调整P数组的大小。原本在sched中的一些变量被移动到了P中以实现去中心化，例如gfree list、runq，这样可以大幅减少全局锁争用。M中的一些和Go代码执行相关的变量也被移动到了P中，例如mcache、stackalloc，如此一来减小了不必要的资源浪费，也优化了局部性。

1. 本地**runq**和全局**runq**

本地runq和全局runq的使用如图6-13和图6-14所示，当一个G从等待状态变成就绪状态后，或者新创建了一个G的时候，这个G会被添加到当前P的本地runq。当M执行完一个G后，它会先尝试从关联的P的本地runq中取下一个，如果本地runq为空，则到全局runq中去取，如图6-13所示，如果全局runq也为空，如图6-14所示，就会去其他的P那里窃取一半的G过来。

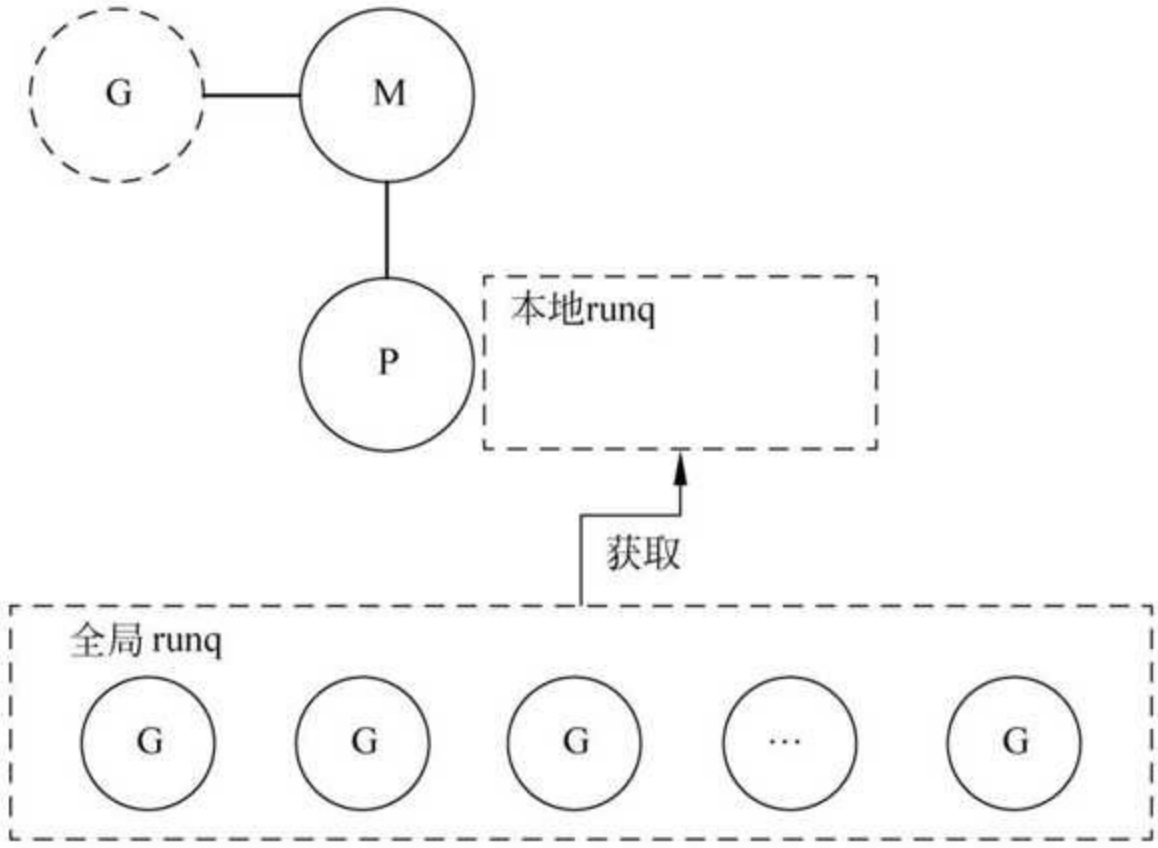

图6-13 本地runq为空到全局runq获取G

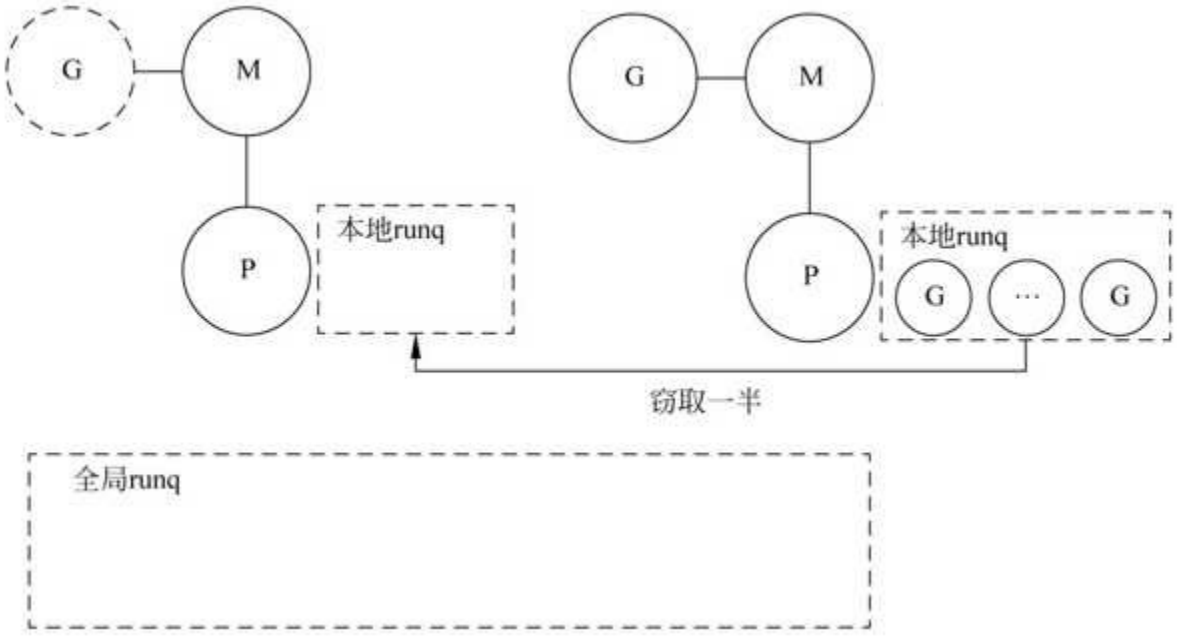

图6-14 全局runq也为空窃取其他P的G

2．M的自旋

当一个M进入系统调用时，它必须确保有其他的M来执行Go代码。新的调度器设计引入了一定程度的自旋，就不用再像之前那样过于频繁地挂起和恢复M了，这会多消耗一些CPU周期，但是对整体性能的影响是正向的。

自旋分两种：第一种是一个有关联P的M，自旋寻找可执行的G；第二种是一个没有P的M，自旋寻找可用的P。这两种自旋的M的个数之和不超过GOMAXPROCS，当存在第二种自旋的M时，第一种自旋的M不会被挂起。

当一个新的G被创建出来或者M即将进行系统调用，或者M从空闲状态变成忙碌状态时，它会确保至少有一个处于自旋状态的M（除非所有的P都忙碌），这样保证了处于可执行状态的G都可以得到调度，同时还不会频繁地挂起、恢复M。

这些理论主要节选自Go 1.1调度器的设计文档，添加了一些笔者自己的理解。所谓调度，简单来讲就是工作线程M如何执行G的问题，具体的实现则包含很多细节，接下来就基于源代码来梳理一下主要逻辑，先从相关的数据结构开始。

6.5 GMP主要数据结构

6.5.1 runtime.g

基于Go 1.14版以后的源码，首先来看一下G，也就是goroutine对应的数据结构runtime.g，完整的结构定义字段较多，这里只从中摘选与调度实现较为密切的部分字段，代码如下：

```
type g struct {
    stack       stack
    stackguard0 uintptr
    stackguard1 uintptr
    m           *m
    sched       gobuf
    atomicstatus uint32
    goid        int64
    schedlink   guintptr
    preempt     bool
    lockedm     muintptr
    waiting     *sudog
    timer       *timer
}
```

部分字段的用途如表6-1所示。

表6-1 runtime.g部分字段的用途

字　段	用　途
stack	描述了 goroutine 的栈空间
stackguard0	被正常的 goroutine 使用，编译器安插在函数头部的栈增长代码，用它来和 SP 比较，按需进行栈增长。它的值一般是 stack.lo＋StackGuard，也可能被设置成 StackPreempt，以触发一次抢占
stackguard1	原理和 stackguard0 差不多，只不过是被 g0 和 gsignal 中的 C 代码使用
m	关联到正在执行当前 G 的工作线程 M
sched	被调度器用来保存 goroutine 的执行上下文
atomicstatus	用来表示当前 G 的状态
goid	当前 goroutine 的全局唯一 ID
schedlink	被调度器用于实现内部链表、队列，对应的 guintptr 类型从逻辑上讲等价于 *g，而底层类型却是个 uintptr，这样是为了避免写障碍
preempt	为 true 时，调度器会在合适的时机触发一次抢占
lockedm	关联到与当前 G 绑定的 M，可以参考一下 LockOSThread
waiting	主要用于实现 channel 中的等待队列，这个留到第 7 章再深入了解
timer	runtime 内部实现的计时器类型，主要用来支持 time.Sleep

其中很多字段被笔者精简了，例如之前讲过的_defer、_panic链表，感兴趣的读者可以去看一看完整的源代码。以上有几个字段需要重点解释一下。

（1）stack是个结构体类型，它的定义代码如下：

```
type stack struct {
    lo uintptr
    hi uintptr
}
```

它是用来描述goroutine的栈空间的，对应的内存区间是一个左闭右开区间[lo，hi）。

（2）用来存储goroutine执行上下文的sched字段需要格外注意，它与goroutine协程切换的底层实现直接相关，其对应的gobuf结构代码如下：

```
type gobuf struct {
    sp uintptr
    pc uintptr
    g guintptr
    ctxt unsafe.Pointer
    ret sys.Uintreg
    lr uintptr
    bp uintptr
}
```

sp字段存储的是栈指针，pc字段存储的是指令指针，g用来反向关联到对应的G。ctxt指向闭包对象，也就是说用go关键字创建协程的时候传递的是一个闭包，这里会存储闭包对象的地址。ret用来存储返回值，实际上是利用AX寄存器实现类似C函数的返回值，目前只发现panic-recover机制用到了该字段。lr在arm等架构上用来存储返回地址，x86没有用到该字段。bp用来存储栈帧基址。

（3）atomicstatus描述了当前G的状态，它主要有如表6-2所示几种取值（省略部分过时无用的状态）。

表6-2 atomicstatus的取值及其含义

取　值	含　义
_Gidle	goroutine 刚刚被分配，还没有被初始化
_Grunnable	goroutine 应该在某个 runq 中，当前并没有在运行用户代码，它的栈不归自己所有
_Grunning	goroutine 可能正在运行用户代码，它的栈归自己所有。运行中的 groutine 不在任何一个 runq 中，并且有关联的 M 和 P
_Gsyscall	goroutine 正在执行一个系统调用，而没有在执行用户代码。它的栈归自己所有，不在任何一个 runq 中，并且有关联的 M
_Gwaiting	goroutine 阻塞在 runtime 中，没有在执行用户代码。它不在任何 runq 中，但是应该被记录在其他地方，例如一个 channel 的等待队列中。它的栈不归自己所有，除非 channel 操作将会在 channel lock 的保护下读写栈上的部分数据。否则，当一个 goroutine 进入_Gwaiting 状态后，再去访问它的栈是不安全的
_Gdead	goroutine 当前没有被用到，它可能刚刚退出运行，在一个空闲链表中，或者刚刚完成初始化。它没有在执行用户代码，可能分配了栈，也可能没有。G 和它的栈（如果有）由退出 G 或从空闲列表中获得 G 的 M 所有
_Gcopystack	goroutine 的栈正在被移动。它没有在执行用户代码，也不在 runq 中。栈的所有权归把当前 goroutine 置为_Gcopystack 状态的 goroutine 所有
_Gpreempted	goroutine 因为 suspendG 抢占而停止。该状态和_Gwaiting 很像，但是没有谁负责将 goroutine 置为就绪状态。一些 suspendG 必须原子性地把该状态转换为_Gwaiting 状态，并且负责重新将该 goroutine 置为就绪状态

标志位_Gscan与上述的一些状态组合，可以得到_Gscanrunnable、_Gscanrunning、_Gscansyscall、

_Gscanwaiting和_Gscanpreempted这些组合状态。除_Gscanrunning外，其他的组合状态都表示GC正在扫描goroutine的栈，goroutine没有在执行用户代码，栈的所有权归设置了_GScan标志位的goroutine所有。_Gscanrunning有些特殊，在GC通知G扫描栈的时候，它被用来短暂地阻止状态变换，其他方面和_Grunning一样。栈扫描完成后，goroutine将会切换回原来的状态，移除_GScan标志位。

（4）waiting对应的sudog结构，留到第7章讲同步时再进行深入分析。

6.5.2 runtime.m

接下来讲解GMP中的M，也就是工作线程Machine。对应的数据结构是runtime.m，仍然只摘选部分与调度相关的字段，代码如下：

```
type m struct {
    g0          *g
    gsignal     *g
    curg        *g
    p           puintptr
    nextp       puintptr
    oldp        puintptr
    id          int64
    preemptoff  string
    locks       int32
    spinning    bool
    park        note
    alllink     *m
    schedlink   muintptr
    lockedg     guintptr
    freelink    *m
}
```

部分字段的用途如表6-3所示。

表6-3 runtime.m部分字段的用途

字　　段	用　　途
g0	并不是一个真正的 goroutine，它的栈是由操作系统分配的，初始大小比普通 goroutine 的栈要大，被用作调度器执行的栈
gsignal	本质上是用来处理信号的栈，因为一些 UNIX 系统支持为信号处理器配置独立的栈
curg	指向的是 M 当前正在执行的 G
p	GMP 中的 P，即关联到当前 M 上的处理器
nextp	用来将 P 传递给 M，调度器一般是在 M 阻塞时为 m.nextp 赋值，等到 M 开始运行后会尝试从 nextp 处获取 P 进行关联
oldp	用来暂存执行系统调用之前关联的 P
id	M 的唯一 ID
preemptoff	不为空时表示要关闭对 curg 的抢占，字符串内容给出了相关的原因
locks	记录了当前 M 持有锁的数量，不为 0 时能够阻止抢占发生
spinning	表示当前 M 正处于自旋状态
park	用来支持 M 的挂起与唤醒，可以很方便地实现每个 M 单独挂起和唤醒
alllink	把所有的 M 连起来，构成 allm 链表
schedlink	被调度器用于实现链表，如空闲 M 链表
lockedg	关联到与当前 M 绑定的 G，可参考 LockOSThread
freelink	用来把已经退出运行的 M 连起来，构成 sched.freem 链表，方便下次分配时复用

6.5.3 runtime.p

再来看一下GMP中的P，也就是Processor对应的数据结构runtime.p，这里只摘选我们感兴趣的部分字段，代码如下：

```
type p struct {
    id          int32
    status      uint32
    link        puintptr
    schedtick   uint32
    syscalltick uint32
    sysmontick  sysmontick
    m           muintptr
    goidcache   uint64
    goidcacheend uint64
    runqhead    uint32
    runqtail    uint32
    runq        [256]guintptr
    runnext     guintptr
    gFree struct {
        gList
        n int32
    }
    preempt     bool
}
```

各个字段的主要用途如表6-4所示。

表6-4 runtime.p各个字段的主要用途

字　　段	主 要 用 途
id	P 的唯一 ID，等于当前 P 在 allp 数组中的下标
status	表示 P 的状态，具体的取值和含义稍后再讲解
link	是一个没有写屏障的指针，被调度器用来构造链表
schedtick	记录了调度发生的次数，实际上在每发生一次 goroutine 切换且不继承时间片的情况下，该字段会加一
syscalltick	每发生一次系统调用就会加一
sysmontick	被监控线程用来存储上一次检查时的调度器时钟嘀嗒，用以实现时间片算法
m	本质上是个指针，反向关联到当前 P 绑定的 M
goidcache	用来从全局 sched.goidgen 处申请 goid 分配区间，批量申请以减少全局范围的锁争用
goidcacheend	
runqhead	当前 P 的就绪队列，用一个数组和一头一尾两个下标实现了一个环形队列
runqtail	
runq	
runnext	如果不为 nil，则指向一个被当前 G 准备好（就绪）的 G，接下来将会继承当前 G 的时间片开始运行。该字段存在的意义在于，假如有一组 goroutine 中有生产者和消费者，它们在一个 channel 上频繁地等待、唤醒，那么调度器会把它们作为一个单元来调度。每次使用 runnext 比添加到本地 runq 尾部能大幅减少延迟
gFree	用来缓存已经退出运行的 G，方便再次分配时进行复用
preempt	在 Go 1.14 版本被引入，以支持新的异步抢占机制

status字段有5种不同的取值，分别表示P所处的不同状态，如表6-5所示。

表6-5 P的不同状态

状　　态	含　　义
_Pidle	空闲状态。此时的 P 没有被用来执行用户代码或调度器代码，通常位于空闲链表中，能够被调度器获取，它的状态可能正在由空闲转变成其他状态。P 的所有权归空闲链表或某个正在改变它状态的线程所有，本地 runq 为空
_Prunning	运行中状态。当前 P 正被某个 M 持有，并且用于执行用户代码或调度器代码。只有持有 P 所有权的 M，才被允许将 P 的状态从_Prunning 转变为其他状态。在任务都执行完以后，M 会把 P 设置为_Pidle 状态。在进入系统调用时，M 会把 P 设置为_Psyscall 状态。挂起以执行 GC 时，会设置为_Pgcstop 状态。某些情况下，M 还可能会直接把 P 的所有权交给另一个 M
_Psyscall	系统调用状态。此时的 P 没有执行用户代码，它和一个处于 syscall 中的 M 间存在弱关联关系，可能会被另一个 M 窃取走
_Pgcstop	GC 停止状态。P 被 STW 挂起以执行 GC，所有权归执行 STW 的 M 所有，执行 STW 的 M 会继续使用处于_Pgcstop 状态的 P。当 P 的状态从_Prunning 转变成_Pgcstop 时，会造成关联的 M 释放 P 的所有权，然后进入阻塞状态。P 会保留它的本地 runq，然后 Start The World 会重新启动这些本地 runq 不为空的 P
_Pdead	停用状态。因为 GOMAXPROCS 收缩，会造成多余的 P 被停用，当 GOMAXPROCS 再次增大时还会被复用。一个停用的 P，大部分资源被剥夺，只有很少量保留

6.5.4 schedt

还有最后一个数据结构需要关注，也就是用来保存调度器全局数据的sched变量对应的schedt类型。就像这个结构的类型名字一样，其中的字段大多数和调度相关，所以就不再进行删减了。摘取Go

1.16版源代码中的schedt结构定义，代码如下：

```
type schedt struct {
    goidgen         uint64
    lastpoll        uint64
    pollUntil       uint64
    lock mutex
    midle           muintptr
    nmidle          int32
    nmidlelocked    int32
    mnext           int64
    maxmcount       int32
    nmsys           int32
    nmfreed         int64
    ngsys           uint32
    pidle           puintptr
    npidle          uint32
```

```
    nmspinning uint32
    runq     gQueue
    runqsize int32
    disable struct {
        user     bool
        runnable gQueue
        n        int32
    }
    gFree struct {
        lock     mutex
        stack    gList
        noStack gList
        n        int32
    }

    sudoglock  mutex
    sudogcache * sudog

    deferlock mutex
    deferpool [5] * _defer

    freem * m

    gcwaiting  uint32
    stopwait   int32
    stopnote   note
    sysmonwait uint32
    sysmonnote note

    sysmonStarting uint32

    safePointFn   func( * p)
    safePointWait int32
    safePointNote note

    profilehz int32

    procresizetime int64
    totaltime      int64
    sysmonlock mutex
}
```

其中部分字段的主要用途如表6-6所示。

表6-6 schedt部分字段的主要用途

字　段	主 要 用 途
goidgen	用作全局的 goid 分配器，以保证 goid 的唯一性。P 中的 goidcache 就是从这里批量获取 goid 的
lastpoll	记录的是上次执行 netpoll 的时间，如果等于 0，则表示某个线程正在阻塞式地执行 netpoll
pollUntil	表示阻塞式的 netpoll 将在何时被唤醒。Go 1.14 版重构了 Timer，引入该字段，唤醒 netpoller 以处理 Timer
lock	全局范围的调度器锁，访问 sched 中的很多字段需要提前获得该锁
midle	空闲 M 链表的链表头，nmidle 记录的是空闲 M 的数量，即链表的长度
nmidlelocked	统计的是与 G 绑定(LockOSThread)且处于空闲状态的 M，绑定的 G 没有在运行，相应的 M 不能用来运行其他 G，只能挂起，以便进入空闲状态
mnext	记录了共创建了多少个 M，同时也被用作下一个 M 的 ID
maxmcount	限制了最多允许的 M 的个数，除去那些已经释放的
nmsys	统计的是系统 M 的个数，这些 M 不在检查死锁的范围内
nmfreed	统计的是累计已经释放了多少 M
ngsys	记录的是系统 goroutine 的数量，会被原子性地更新
pidle	空闲 P 链表的表头，npidle 记录了空闲 P 的个数，也就是链表的长度
nmspinning	记录的是处于自旋状态的 M 的数量
runq	全局就绪队列
runqsize	记录的是全局就绪队列的长度
disable	用来禁止调度用户 goroutine，其中的 user 变量被置为 true 后，调度器将不再调度执行用户 goroutine，系统 goroutine 不受影响。期间就绪的用户 goroutine 会被临时存放到 disable.runnable 队列中，变量 n 记录了队列的长度
gFree	用来缓存已退出运行的 G，lock 是本结构单独的锁，避免争用 sched.lock。stack 和 noStack 这两个列表分别用来存储有栈和没有栈的 G，因为在 G 结束运行被回收的时候，如果栈大小超过了标准大小，就会被释放，所以有一部分 G 是没有栈的。变量 n 是两个列表长度之和，也就是总共缓存了多少个 G
sudoglock	构成了 sudog 结构的中央缓存，供各个 P 存取
sudogcache	
deferlock	构成了_defer 结构的中央缓存，关于 defer 的详情可阅读 3.4 节的相关内容
deferpool	
freem	一组已经结束运行的 M 构成的链表的表头，通过 m.freelink 链接到下一项，链表中的内容在分配新的 M 时会被复用
gcwaiting	表示 GC 正在等待运行，和 stopwait、stopnote 一同被用于实现 STW。stopwait 记录了 STW 需要停止的 P 的数量，发起 STW 的线程会先把 GOMAXPROCS 赋值给 stopwait，也就是需要停止所有的 P。再把 gcwaiting 置为 1，然后在 stopnote 上睡眠等待被唤醒。其他正在运行的 M 检测到 gcwaiting 后会释放关联 P 的所有权，并把 P 的状态置为_Pgcstop，再把 stopwait 的值减 1，然后 M 把自己挂起。M 在自我挂起之前如果检测到 stopwait=0，也就是所有 P 都已经停止了，就会通过 stopnote 唤醒发起 STW 的线程
sysmonwait	不为 0 时表示监控线程 sysmon 正在 sysmonnote 上睡眠，其他的 M 会在适当的时机将 sysmonwait 置为 0，并通过 sysmonnote 唤醒监控线程
sysmonStarting	表示主线程已经创建了监控线程 sysmon，但是后者尚未开始运行，某些操作需要等到 sysmon 启动之后才能进行
safePointFn	是个 Function Value，safePointWait 和 safePointNote 的作用有些类似于 stopwait 和 stopnote，被 runtime. forEachP 用来确保每个 P 都在下一个 GC 安全点执行了 safePointFn
profilehz	用来设置性能分析的采样频率
procresizetime	统计了改变 GOMAXPROCS 所花费的时间
totaltime	
sysmonlock	监控线程 sysmon 访问 runtime 数据时会加上的锁，其他线程可以通过它和监控线程进行同步

与调度相关的数据结构的介绍就到这里，从其中一些字段的用途就能够大致感受到调度器实现的思路。接下来，尝试按照不同的阶段或不同的功能模块逐步了解整个调度器。

6.6 调度器初始化

6.6.1 调度器初始化过程

Go程序代码经过build之后，生成的是系统原生的可执行文件。可执行文件一般会有个执行入口，也就是被加载到内存后指令开始执行的地址。如图6-15所示，这个执行入口在不同平台上不尽相同。在amd64+linux平台上，使用-buildmode＝exe模式构建出来的可执行文件，其对外暴露的执行入口是_rt0_amd64_linux，对应runtime源码中的汇编函数_rt0_amd64_linux（），该函数只有一条JMP指令，用于跳转到汇编函数_rt0_amd64（）。汇编函数_rt0_amd64（）只有3条指令，用于立刻调用runtime.rt0_go（）函数。rt0_go（）函数也是个汇编函数，该函数包含了Go程序启动的大致流程。

接下来我们就从可执行文件的执行入口开始讲解，一直讲解到程序中的main（）函数，看一看Go程序是如何开始执行的。以下是rt0_go（）函数的主要逻辑：

（1）初始化g0的栈区间，检测CPU厂商及型号，按需调用_cgo_init（）函数，设置和检测TLS，将m0和g0相互关联，并将g0设置到TLS中，如图6-16所示。

（2）调用runtime.args（）函数来暂存命令行参数以待后续解析。部分系统会在这里获取与硬件相关的一些参数，例如物理页面大小。

（3）调用runtime.osinit（）函数，所有的系统都会在这里获取CPU核心数，如果上一步没有成功获取物理页面大小，则部分系统会再次获取。Linux系统会在这里获取Huge物理页面的大小。

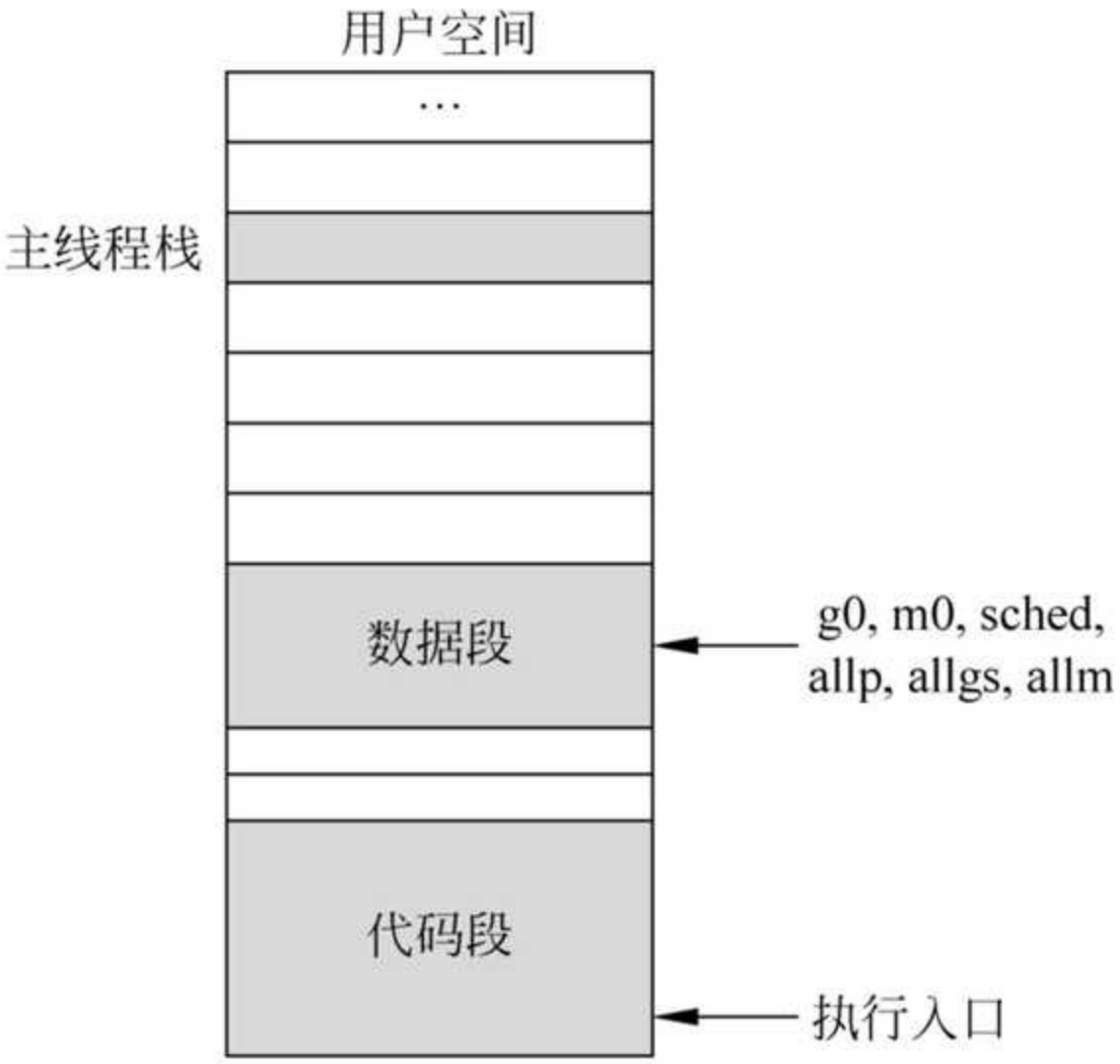

图6-15 可执行文件的内存布局

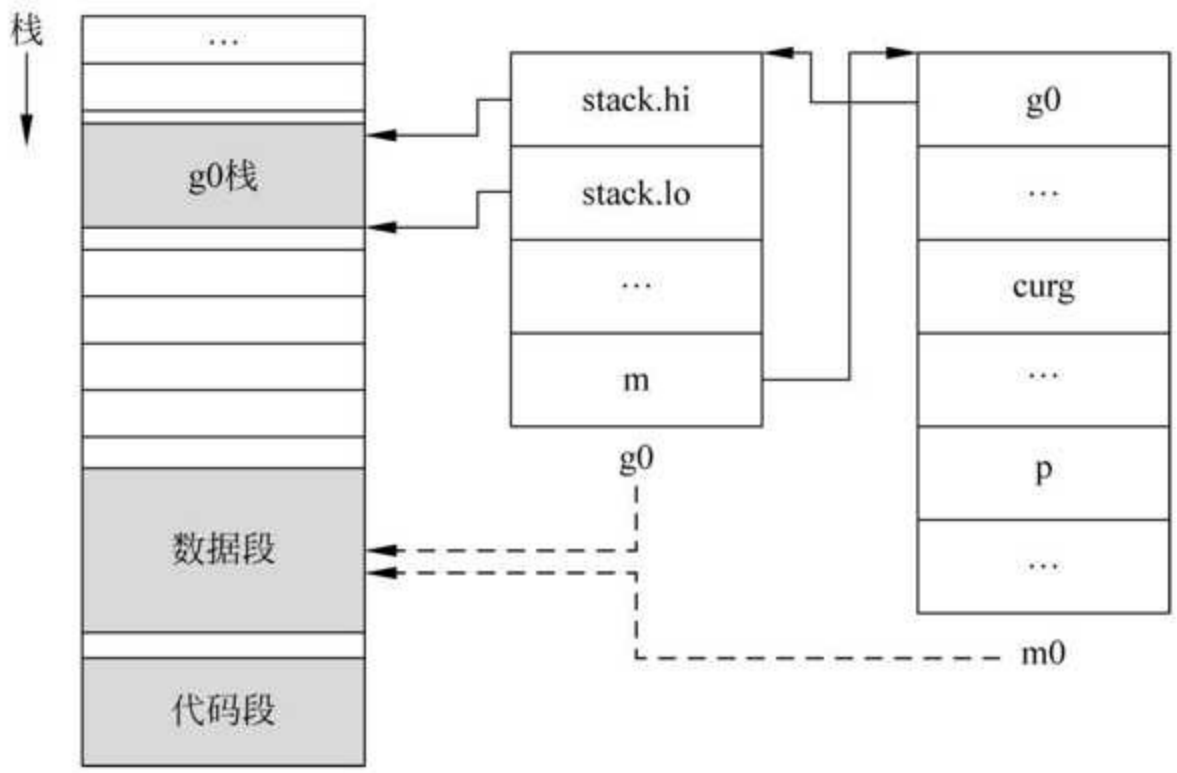

图6-16 初始化g0栈并关联m0

（4）调用runtime.schedinit（）函数，就像它的名字那样，这个函数会初始化调度系统，函数的逻辑较为复杂，相关细节稍后再展开介绍。

（5）调用runtime.newproc（）函数，创建主goroutine，指定的入口函数是runtime.main（）函数，这是程序启动后第1个真正的goroutine，如图6-17所示。

（6）调用runtime.mstart（）函数，当前线程进入调度循环。一般情况下线程调用mstart（）函数进入调度循环后不会再返回。进入调度循环的线程会去执行上一步创建的goroutine，如图6-18所示。

主goroutine得到执行后，runtime.main（）函数会设置最大栈大小、启动监控线程sysmon、初始化runtime包、开启GC，最后初始化main包并调用main.main（）函数。main.main（）函数是用户代码的主函数，整个初始化过程至此彻底结束。

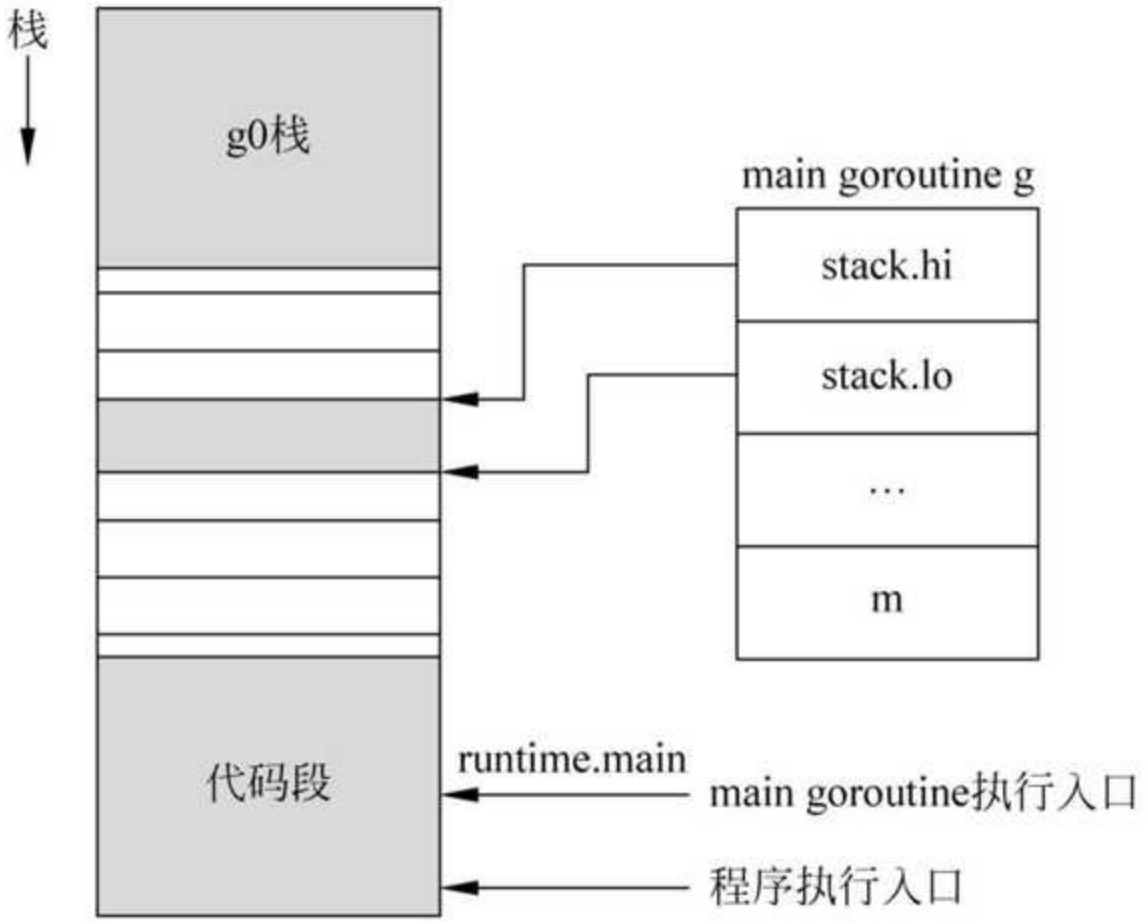

图6-17 创建主goroutine

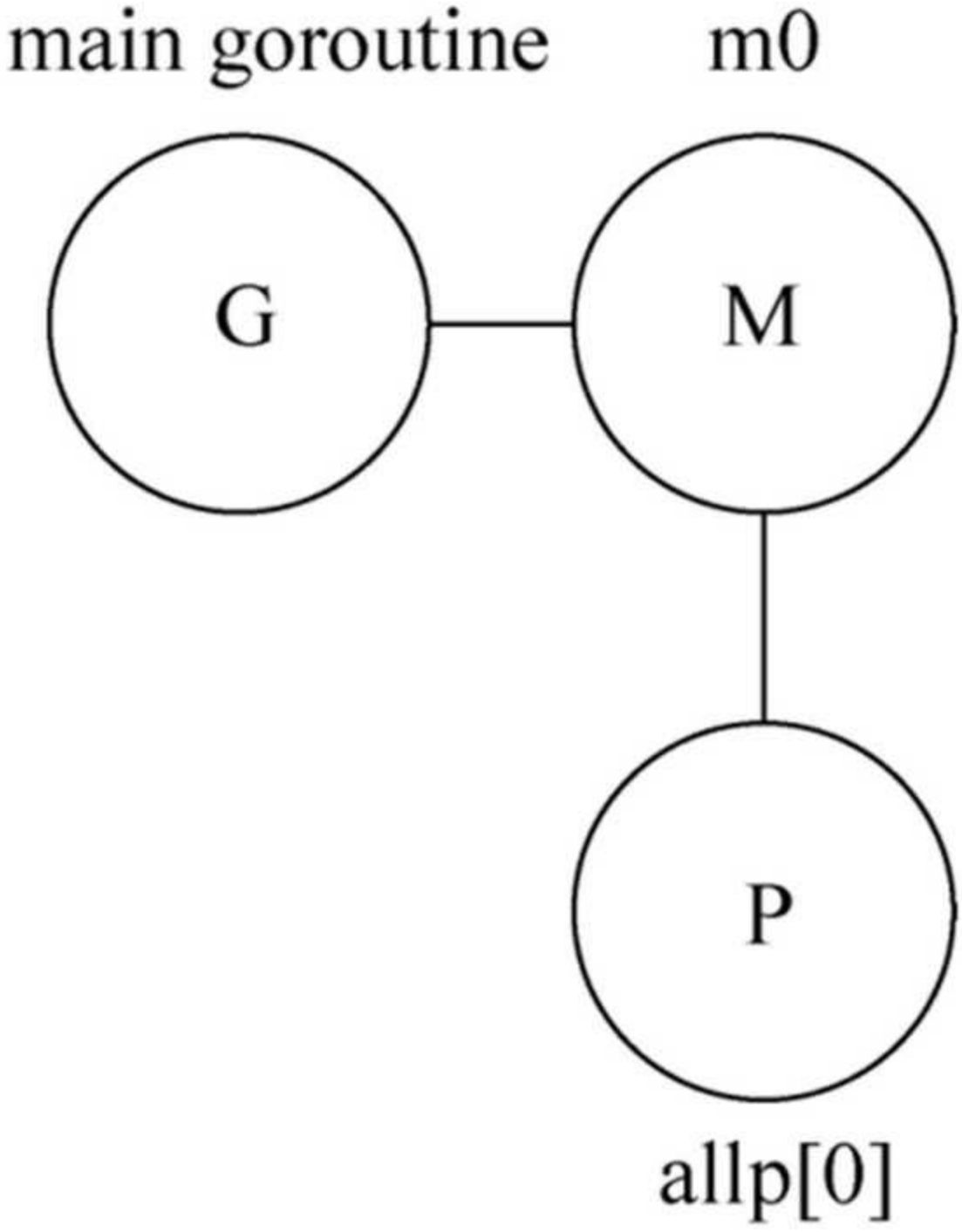

图6-18 主goroutine执行

6.6.2 runtime.schedinit（）函数

在上述整个流程中，调用runtime.schedinit（）函数实际上做了很多事情，需要把这个函数的逻辑梳理一下，该函数也是通过调用多个其他函数完成操作的，调用的函数及其用途如表6-7所示。

表6-7 runtime.schedinit（）函数调用的函数及其用途

调用函数	用　途
moduledataverify()	校验程序的各个模块，因为 golang 支持 shared、plugin 等 build 模式，可能会有多个二进制模块，这里会校验各个模块的符号、ABI 等，确保模块间一致
stackinit()	初始化栈分配。goroutine 的栈是动态分配、动态增长的，这一步会初始化用于栈分配的全局缓冲池，以及相关的锁
mallocinit()	初始化堆分配
fastrandinit()	初始化 fastrandseed，后者会被接下来的 mcommoninit()函数用到
mcommoninit()	为当前工作线程 M 分配 ID、初始化 gsignal，并把 M 添加到 allm 链表中
cpuinit()	进行与 CPU 相关的初始化工作，检测 CPU 是否支持某些指令集，以及根据 GODEBUG 环境变量来启用或禁用某些硬件特性
alginit()	根据 CPU 对 AES 相关指令的支持情况，选择不同的 Hash 算法
modulesinit()	基于所有的已加载模块，构造一个活跃模块切片 modulesSlice，并初始化 GC 需要的 Mask 数据
typelinksinit()	基于活跃模块列表构建模块级的 typemap，实现全局范围内对类型元数据进行去重，第 5 章中进行过详细介绍
itabsinit()	遍历活跃模块列表，将编译阶段生成的所有 itab 添加到 itabTable 中
goargs()	解析命令行参数，程序中通过 os.Args 得到的参数是在这里初始化的（Windows 除外）
goenvs()	解析环境变量，程序中通过 os.Getenv 获取的环境变量是在这里初始化的（Windows 除外）
parsedebugvars()	解析环境变量 GODEBUG，为 runtime 各个调试参数赋值
gcinit()	初始化与 GC 相关的参数，根据环境变量 GOGC 设置 gcpercent
procresize()	根据 CPU 的核数或环境变量 GOMAXPROCS 确定 P 的数量，调用 procresize 进行调整

直至runtime.schedinit（）函数执行完，P都已经初始化完毕，此时还没有创建任何goroutine，所有P的runq都是空的。根据procresize（）函数的逻辑，函数返回后当前线程会和第1个P关联，也就是allp[0]。接下来的runtime.newproc（）函数会创建第1个goroutine，并把它放到P的本地runq中，如图6-19所示。

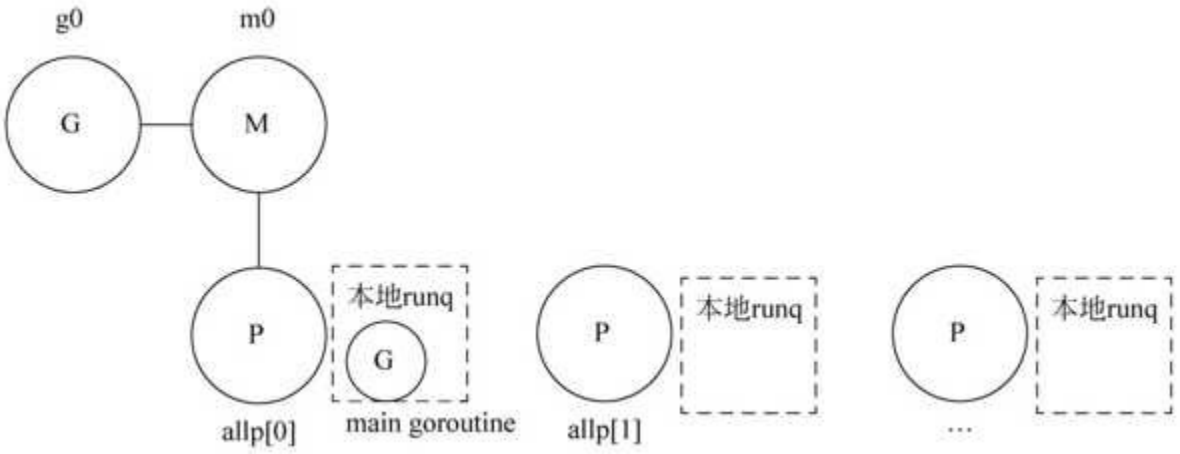

图6-19 第1个goroutine创建后的GMP模型

6.7 G的创建与退出

跟线程类似，goroutine的创建与退出是两个比较关键的操作。在分析用来创建goroutine的runtime.newproc（）函数之前，需要先了解几个重要的底层汇编函数。

6.7.1 相关汇编函数

1．runtime.systemstack（）函数

首先是runtime.systemstack（）函数，该函数被设计用来临时性地切换至当前M的g0栈，完成某些操作后再切换回原来goroutine的栈。该函数主要用于执行runtime中一些会触发栈增长的函数，因为goroutine的栈是被runtime管理的，所以runtime中这些逻辑就不能在普通的goroutine上执行，以免陷入递归。g0的栈是由操作系统分配的，可以认为空间足够大，被runtime用来执行自身逻辑非常安全。runtime.systemstack（）函数的代码如下：

```
//func systemstack(fn func())
TEXT runtime·systemstack(SB), NOSPLIT, $0-8
    MOVQ    fn+0(FP), DI          //把 fn 存入 DI 寄存器
    get_tls(CX)
    MOVQ    g(CX), AX             //从 TLS 获取当前 g,存入 AX 中
    MOVQ    g_m(AX), BX           //将 g.m 存入 BX 中

    CMPQ    AX, m_gsignal(BX)     //比较当前 g 是否是 m.gsignal
    JEQ     noswitch              //gsignal 也是系统栈,不切换

    MOVQ    m_g0(BX), DX          //将 m.g0 存入 DX 中
    CMPQ    AX, DX                //比较当前 g 是不是 g0
    JEQ     noswitch              //已经在 g0 上,不需要切换

    CMPQ    AX, m_curg(BX)
    JNE  bad                      //当前 g 与 m.curg 不一致

    //将当前 g 的状态保存至 g->sched
    MOVQ    $runtime·systemstack_switch(SB), SI
    MOVQ    SI, (g_sched+gobuf_pc)(AX)
    MOVQ    SP, (g_sched+gobuf_sp)(AX)
    MOVQ    AX, (g_sched+gobuf_g)(AX)
    MOVQ    BP, (g_sched+gobuf_bp)(AX)

    //将 g0 设置到 TLS
    MOVQ    DX, g(CX)
    MOVQ    (g_sched+gobuf_sp)(DX), BX //g0.sched.sp =>BX
    //使 g0 的调用栈看起来像 mstart 调用的 systemstack,以停止 traceback
    SUBQ    $8, BX
    MOVQ    $runtime·mstart(SB), DX
    MOVQ    DX, 0(BX) //构造一个 mstart 的栈帧
    MOVQ    BX, SP

    //调用目标函数,用 DX 传递闭包上下文
    MOVQ    DI, DX
    MOVQ    0(DI), DI
    CALL    DI

    //切换回原来的 g
    get_tls(CX)
    MOVQ    g(CX), AX
    MOVQ    g_m(AX), BX
    MOVQ    m_curg(BX), AX
    MOVQ    AX, g(CX)
    MOVQ    (g_sched+gobuf_sp)(AX), SP
```

```
    MOVQ    $0, (g_sched+gobuf_sp)(AX)
    RET

noswitch:
    //已经在系统栈了,直接跳转到目标函数,省略掉 systemstack 的栈帧
    MOVQ    DI, DX
    MOVQ    0(DI), DI
    JMP     DI

bad:
    //出错:g 不是 gsignal,不是 g0,也不是 curg
    MOVQ    $runtime·badsystemstack(SB), AX
    CALL    AX
    INT     $3
```

函数接收一个没有参数和返回值的Function Value作为参数，静态的函数和闭包都能支持。如果当前已经处于gsignal或g0的栈上，则systemstack（）函数没有任何作用，就像调用者不使用systemstack（）函数而直接调用fn（）函数一样，所以是可以嵌套使用的。需要注意的是，当从g0切换回g的时候，并没有将g0的状态保存到g0.sched中，也就是说每次从g0切换至其他的goroutine后，g0栈上的内容就被抛弃了，下次切换至g0还是从头开始。

2．runtime.mcall（）函数

runtime.mcall（）函数和systemstack（）函数很像，也是切换到系统栈去执行某个Function Value，但是也有些不同，mcall（）函数不能在g0栈上调用，而且也不会再切换回来，函数的代码如下：

```
//func mcall(fn func( * g))
TEXT runtime·mcall(SB), NOSPLIT, $ 0 - 8
    MOVQ    fn + 0(FP), DI

    get_tls(CX)
    MOVQ    g(CX), AX                   //将当前 g 的状态保存到 g - > sched
    MOVQ    0(SP), BX                   //调用者的指令指针位置,PC
    MOVQ    BX, (g_sched + gobuf_pc)(AX)
    LEAQ    fn + 0(FP), BX              //调用者的栈指针,SP
    MOVQ    BX, (g_sched + gobuf_sp)(AX)
    MOVQ    AX, (g_sched + gobuf_g)(AX)
    MOVQ    BP, (g_sched + gobuf_bp)(AX)

    //切换至 m - > g0 的栈,然后调用 fn
    MOVQ    g(CX), BX
    MOVQ    g_m(BX), BX
    MOVQ    m_g0(BX), SI
    CMPQ    SI, AX                      //如果当前 g 就是 m - > g0,则调用 badmcall
    JNE     3(PC)
    MOVQ    $ runtime·badmcall(SB), AX
    JMP     AX
    MOVQ    SI, g(CX)                   //将 m - > g0 设置到 TLS
    MOVQ    (g_sched + gobuf_sp)(SI), SP //切换至 g0 的栈
    PUSHQ   AX
    MOVQ    DI, DX                      //传递闭包上下文
    MOVQ    0(DI), DI
    CALL    DI
    POPQ    AX
    MOVQ    $ runtime·badmcall2(SB), AX
    JMP     AX
    RET
```

函数会先把当前g的状态保存到g.sched，然后切换至g0栈，用当前g的指针作为参数调用fn（）函数。这个流程非常适合goroutine将自己挂起，fn（）函数中执行调度逻辑对g进行后续处理。需要注意该函数预期fn（）函数不会返回，也就是说fn（）函数中的调度逻辑需要选择下一个可执行的g，并完成切换。如何切换到新的g去执行呢？这就是接下来要介绍的runtime.gogo（）函数。

3．runtime.gogo（）函数

runtime.gogo（）函数的代码如下：

```
//func gogo(buf *gobuf)
//从 gobuf 恢复 goroutine 的状态,类似 C 语言的 longjmp
TEXT runtime·gogo(SB), NOSPLIT, $16-8
    MOVQ    buf+0(FP), BX       //gobuf
    MOVQ    gobuf_g(BX), DX
    MOVQ    0(DX), CX           //确保 buf.g 不等于 nil
    get_tls(CX)
    MOVQ    DX, g(CX)
    MOVQ    gobuf_sp(BX), SP    //恢复栈指针 SP
    MOVQ    gobuf_ret(BX), AX
    MOVQ    gobuf_ctxt(BX), DX
    MOVQ    gobuf_bp(BX), BP
    MOVQ    $0, gobuf_sp(BX)    //清零以优化 GC
    MOVQ    $0, gobuf_ret(BX)
    MOVQ    $0, gobuf_ctxt(BX)
    MOVQ    $0, gobuf_bp(BX)
    MOVQ    gobuf_pc(BX), BX
    JMP     BX
```

函数有一个*gobuf类型的参数，buf.g是要恢复运行的goroutine，gogo（）函数利用gobuf中保存的状态来还原对应的寄存器，再跳转到buf.pc地址处去执行指令。

既然有longjmp，自然也有与之对应的setjmp，也就是runtime.gosave（）函数。

4．runtime.gosave（）函数

runtime.gosave（）函数用来把当前goroutine的执行状态保存到gobuf中，代码如下：

```
//func gosave(buf *gobuf)
//将执行状态保存到 gobuf 中,类似 C 语言的 setjmp
TEXT runtime·gosave(SB), NOSPLIT, $0-8
    MOVQ    buf+0(FP), AX       //gobuf 地址 => AX
    LEAQ    buf+0(FP), BX       //调用者 SP => BX
    MOVQ    BX, gobuf_sp(AX)
    MOVQ    0(SP), BX           //调用者 PC => BX
    MOVQ    BX, gobuf_pc(AX)
    MOVQ    $0, gobuf_ret(AX)
    MOVQ    BP, gobuf_bp(AX)
    //断言 ctxt 是 0,只有初创尚未运行时不为 0
    MOVQ    gobuf_ctxt(AX), BX
    TESTQ   BX, BX
    JZ      2(PC)
    CALL    runtime·badctxt(SB)
    get_tls(CX)
    MOVQ    g(CX), BX
    MOVQ    BX, gobuf_g(AX)
    RET
```

函数取的SP和PC的值就像是刚从gosave（）函数返回，后续如果使用gogo（）函数进行longjmp，程序会从调用者调用gosave（）函数的下一条指令继续执行。关于gobuf.ctxt，因为创建goroutine时go关键字后面的Function Value可能是个闭包，所以要依靠ctxt来传递闭包对象。一旦使用gogo（）函数来恢复执行，gobuf.ctxt就会被清零。

了解了上述几个底层函数之后，阅读与调度相关的源码就会比较方便了。下面就来看一下负责创建新goroutine的runtime.newproc（）函数。

6.7.2 runtime.newproc（）函数

先来看一个Hello World示例，代码如下：

```
//第 6 章/code_6_1.go
package main
func hello(name string){
    println("Hello ", name)
}
func main(){
    name := "Goroutine"
    go hello(name)
}
```

通过6.6节关于初始化的介绍，我们已经了解了从程序执行入口开始，到main.main（）函数执行的大致过程。main.main（）函数执行时会通过go关键字创建一个协程，我们姑且把它记为hello goroutine，这里的go关键字实际上会被编译器转换成对runtime.newproc（）函数的调用。函数的代码如下：

```
//go:nosplit
func newproc(siz int32, fn *funcval) {
    argp := add(unsafe.Pointer(&fn), sys.PtrSize)
    gp := getg()
    pc := getcallerpc()
    systemstack(func() {
        newg := newproc1(fn, argp, siz, gp, pc)
        _p_ := getg().m.p.ptr()
        runqput(_p_, newg, true)

        if mainStarted {
            wakep()
        }
    })
}
```

我们先绘制创建hello goroutine的newproc（）函数调用栈，如图6-20所示，其中有几个要点需要分别进行说明：

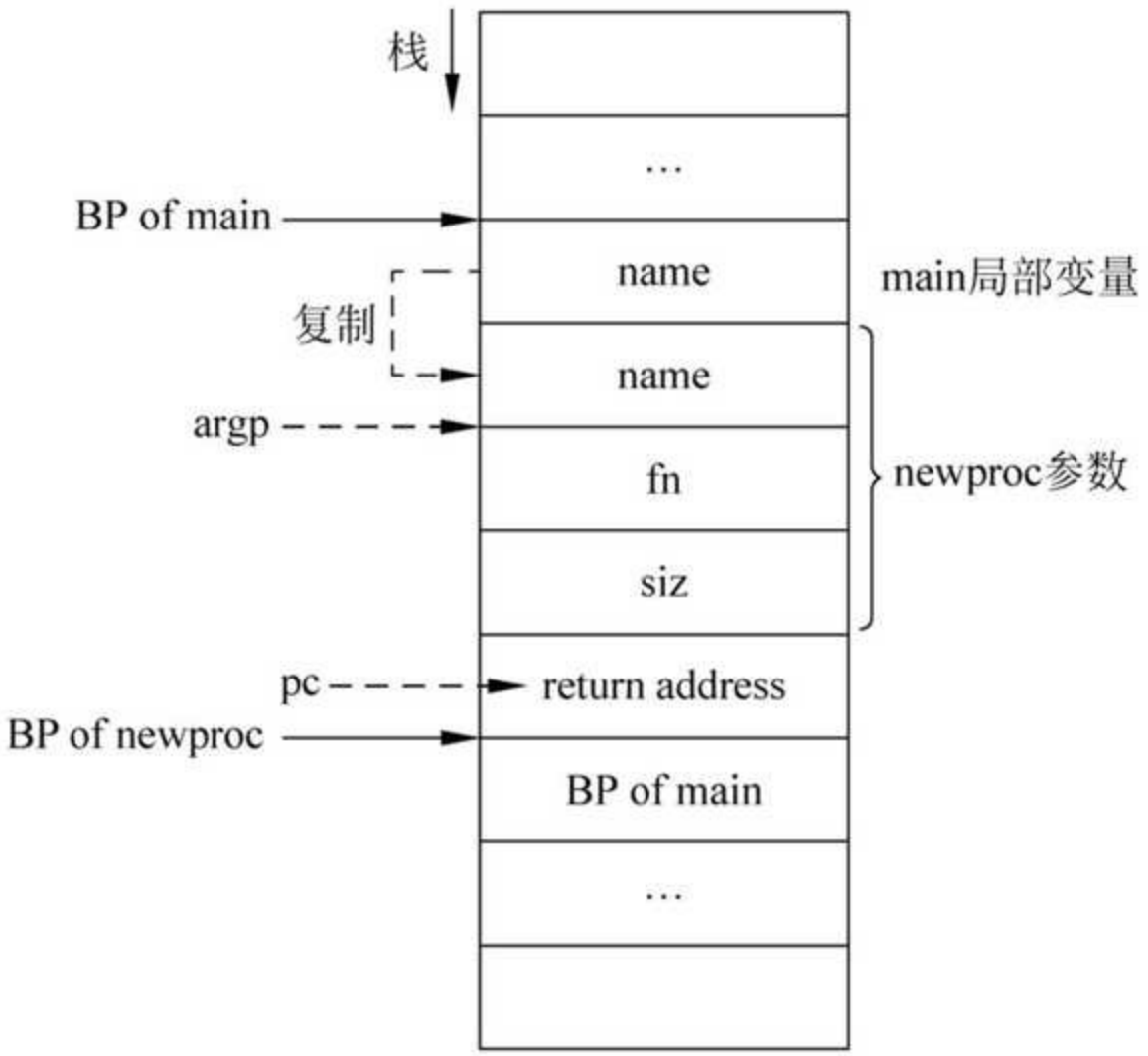

图6-20 创建hello goroutine的newproc（）函数调用栈

（1）argp指针所指向的位置在栈上位于参数fn之后，就像是newproc（）函数的第3个参数。从argp开始siz字节的数据实际上是fn（）函数的参数，被编译器追加在了栈上参数fn的后面，这一点与defer机制的runtime.deferproc（）函数一致。从newproc（）函数的原型来看，这些被追加的参数是不可见的，所以newproc（）函数必须是nosplit，以免移动栈时丢失这些参数。

（2）通过getcallerpc（）函数获取的是创建者的指令指针，主要被新的goroutine用于记录自己是在哪里被创建的。

（3）实际上真正的创建工作是在runtime.newproc1（）函数中完成的，该函数有些复杂，可能会造成栈增长，同时又有nosplit的限制，所以要通过systemstack（）函数切换至系统栈执行。

（4）新创建的newg通过runqput（）函数被放置在当前P的本地runq中。mainStarted表示runtime.main（）函数，即主goroutine已经开始执行，此后才会通过wakep（）函数启动新的工作线程，以保证main（）函数总会被主线程调度执行。

对于新goroutine的分配及初始化工作，都是在runtime.newproc1（）函数中完成的，该函数的代码篇幅较大，此处就不将整段代码贴出来了，只在必要的地方节选一些。函数的主要逻辑包含以下几部分：

（1）分配新的g，先尝试gfget（）函数从空闲队列中获取，如果没有，再用malg（）函数分配新的g。

（2）计算栈上所需空间的大小，用参数的大小加额外预留的空间，还要经过对齐。

（3）根据上一步的计算确定SP的位置，把参数复制到新g的栈上，需要用到写屏障。

（4）初始化执行上下文，这里用到gostartcallfn（）函数，稍后会进一步展开介绍。

（5）将G的状态设置为_Grunnable，并根据当前P的goidcache为g分配ID。

关于新goroutine执行上下文的初始化比较关键，因为初始化过的g会先被放入P的本地runq中，等到接下来的调度循环中才会被执行。切换到新的goroutine执行会用到runtime.gogo（）函数，也就是基于g.sched的gobuf来恢复执行现场，所以初始化的时候要在g.sched中模拟出一个执行现场，关键代码如下：

```
newg.sched.sp = sp
newg.sched.pc = funcPC(goexit) + sys.PCQuantum
newg.sched.g = guintptr(unsafe.Pointer(newg))
gostartcallfn(&newg.sched, fn)
```

创建hello goroutine时newproc1（）函数模拟的执行现场如图6-21所示。其中的sp就是从栈底留出参数及额外空间后的位置，pc的位置比较有意思，是runtime.goexit（）函数的起始地址加上1字节（sys.PCQuantum在amd64上是1字节）。这样初始化pc是为了让调用栈看起来像是起始于goexit（）函数，然后goexit（）函数调用了fn（）函数，也就是hello（）函数。如此一来，当fn（）函数执行完毕后，会返回goexit（）函数中，goexit（）函数中实现了goroutine结束后退出的标准逻辑。pc的值之所以需要是goexit（）函数的地址加1，是因为这样才像是goexit（）函数调用了fn（）函数，如果指向goexit（）函数的起始地址就不合适了，那样goexit（）函数看起来还没有执行。

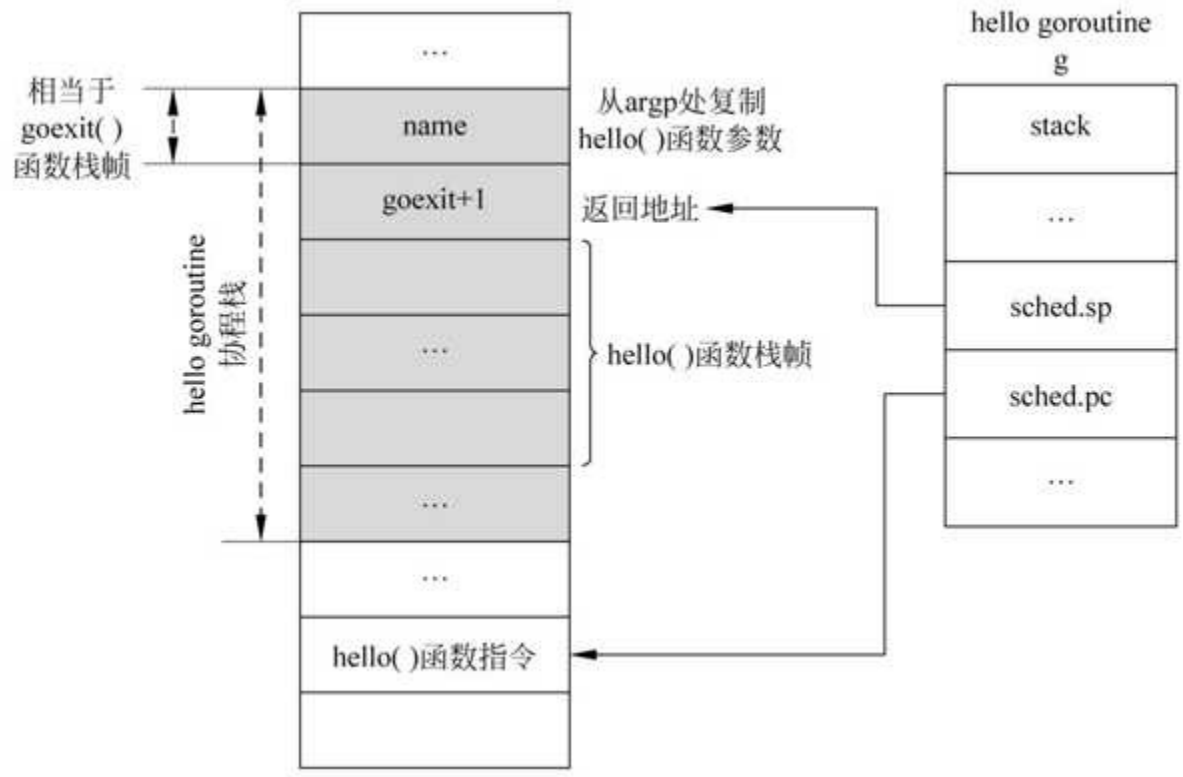

图6-21 创建hello goroutine时newproc1（）函数模拟的执行现场

读者可能会担心goexit（）函数地址加1会造成指令错乱，实际不会有问题，因为goexit（）函数的代码已经考虑到这一层了，代码如下：

```
TEXT runtime·goexit<ABIInternal>(SB),NOSPLIT,$0-0
    BYTE    $0x90       //NOP
    CALL    runtime·goexit1(SB)
    BYTE    $0x90       //NOP
```

首尾各有一条NOP指令占位，所以入口地址加1后不会有什么影响，正好对齐到了接下来的CALL指令，而且还可以发现goexit（）函数真正的逻辑是在goexit1（）函数中实现的，这个暂不展开介绍。接下来继续看goroutine执行上下文的初始化，gostartcallfn（）函数内部调用了gostartcall（）函数实现了主要功能，x86对应的gostartcall（）函数的源代码如下：

```
func gostartcall(buf *gobuf, fn, ctxt unsafe.Pointer) {
    sp := buf.sp
    if sys.RegSize > sys.PtrSize {
        sp -= sys.PtrSize
        *(*uintptr)(unsafe.Pointer(sp)) = 0
    }
    sp -= sys.PtrSize
    *(*uintptr)(unsafe.Pointer(sp)) = buf.pc
    buf.sp = sp
    buf.pc = uintptr(fn)
    buf.ctxt = ctxt
}
```

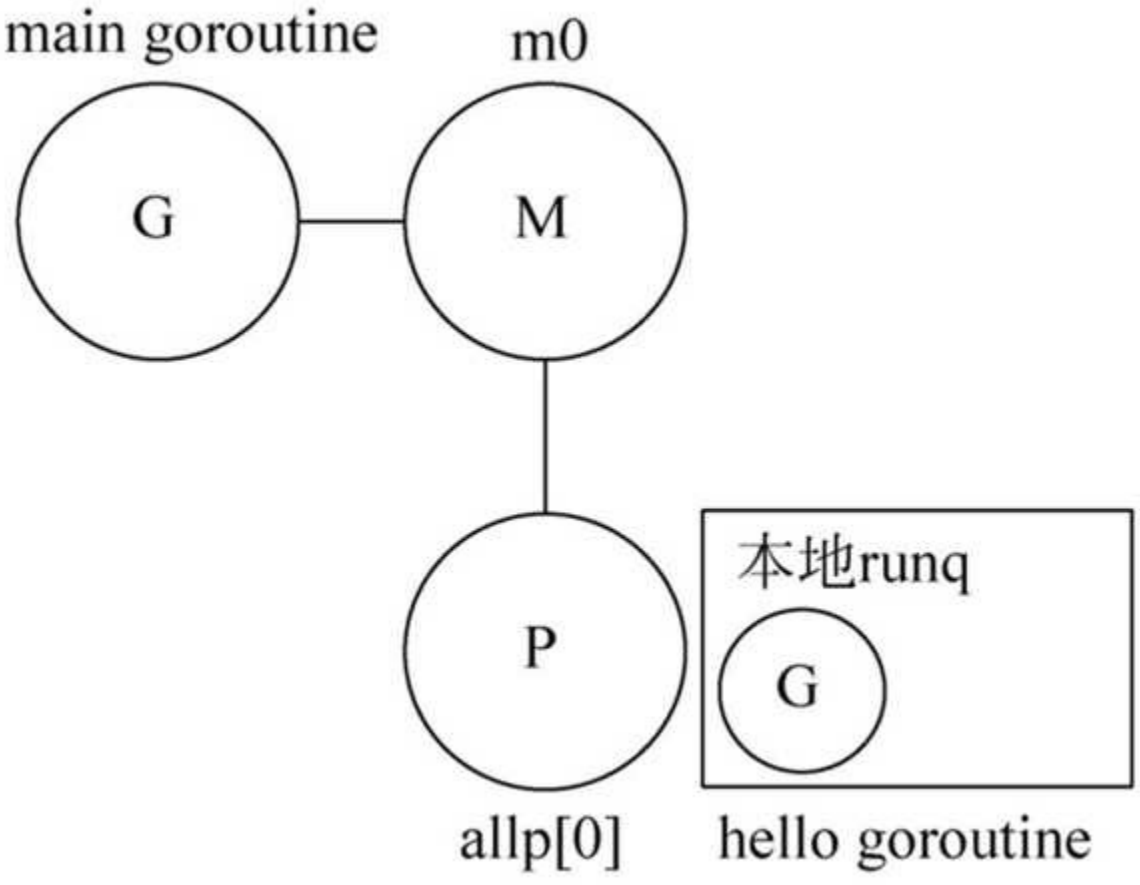

图6-22 hello goroutine创建后的GMP

寄存器大小不等于指针大小的情况可直接忽略，函数的主要逻辑是：先把SP向下移动一个指针大小，然后把PC的值写入SP指向的内存，这相当于在栈上压入了一个新的栈帧，原PC成为返回地址。最后更新gobuf的sp和pc字段，新的pc是fn，最终构造的执行现场就像是goexit（）函数刚刚调用了fn（）函数，刚刚完成跳转还没来得及执行fn（）函数的指令。

至此，runtime.newproc（）函数创建新goroutine的流程大致梳理完了，新goroutine已经被放置到了P的本地runq中，会在后续的调度循环中得到执行，hello goroutine创建完成后GMP模型如图6-22所示。

main goroutine在main.main（）函数返回后就会调用exit（）函数结束进程，所以示例代码中的hello goroutine还没来得及得到调度执行，整个进程就结束了。可以通过等待timer或channel的方式拖延main.main（）函数的返回时间，这样就可以等到hello goroutine退出后再结束进程了。

至于goroutine的退出，相对而言就比较简单了，runtime.goexit1（）函数实际上也不是主要逻辑实现的地方，该函数只不过通过mcall（）函数调用了goexit0（）函数。为什么要通过mcall（）函数调用呢？因为当前goroutine即将退出了，不能继续执行，必须切换至系统栈来完成收尾处理。goexit0（）函数中才是真正进行收尾的地方，该函数的逻辑比较简单，主要包括以下几个步骤：

（1）将g的状态置为_Gdead。

（2）g的一些字段需要做清零处理。

（3）通过dropg（）函数将g与当前M解绑。

（4）调用gfput（）函数将g放入空闲队列，以便于复用。

（5）调用schedule（）函数，调度执行其他已经就绪的goroutine。

其中最后一步调用的runtime.schedule（）函数就是我们通常所讲的调度循环，确切地说应该是调度循环中的一次循环，工作线程通过不断地调用schedule（）函数来调度执行下一个goroutine。6.8节将从schedule（）函数入手，梳理一下调度的主要逻辑。

6.8 调度循环

6.8.1 runtime.schedule（）函数

工作线程通过调用runtime.schedule（）函数进行一次调度，该函数就是调度逻辑的主要实现。源代码稍微有点多，下面分成几部分进行梳理。

第一部分代码如下：

```
_g_ := getg()
if _g_.m.locks != 0 {
    throw("schedule: holding locks")
}
if _g_.m.lockedg != 0 {
    stoplockedm()
    execute(_g_.m.lockedg.ptr(), false)
}
if _g_.m.incgo {
    throw("schedule: in cgo")
}
```

函数开始处先通过getg（）函数获得了当前正在运行的g，执行schedule（）函数时一般都是系统栈g0。接下来的第1个if语句校验当前线程没持有锁，不允许在持有锁的情况下进行调度，以免造成runtime内部错误，这里的锁是runtime底层的锁，与sync包中的Mutex等不是一个级别。第2个if判断当前M有没有和G绑定，如果有，这个M就不能用来执行其他的G了，只能挂起等待绑定的G得到调度。第3个if判断线程是不是正在进行cgo函数调用，这种情况下g0栈正在被cgo使用，所以也不允许调度。

第二部分代码如下：

```
top:
    pp := _g_.m.p.ptr()
    pp.preempt = false

    if sched.gcwaiting != 0 {
        gcstopm()
        goto top
    }
    if pp.runSafePointFn != 0 {
        runSafePointFn()
    }
    if _g_.m.spinning && (pp.runnext != 0 || pp.runqhead != pp.runqtail){
        throw("schedule: spinning with local work")
    }
```

从top标签开始就是真正的调度逻辑了，设置这个标签的目的，是为了后面某些情况下需要goto这里重来一遍。通过把preempt设置为false，来禁止对P的抢占。检测sched.gcwaiting，挂起自己，以便及时响应STW，调度逻辑中多个地方都有对gcwaiting的检测。runSafePointFn（）函数被GC用来在安全点执行清空工作队列之类的操作。最后对spinning的判断属于一致性校验，在P本地runq有任务的情况下，M不应该处于spinning状态。

第三部分代码如下：

```
checkTimers(pp, 0)

var gp *g
var inheritTime bool

tryWakeP := false
if trace.enabled || trace.shutdown {
    gp = traceReader()
    if gp != nil {
        casgstatus(gp, _Gwaiting, _Grunnable)
        traceGoUnpark(gp, 0)
        tryWakeP = true
    }
}
if gp == nil && gcBlackenEnabled != 0 {
    gp = gcController.findRunnableGCWorker(_g_.m.p.ptr())
    tryWakeP = tryWakeP || gp != nil
}
```

通过checkTimers（）函数处理当前P上的定时器，关于定时器会在6.10节中详细讲解。接下来的两个if语句块尝试获得待运行的Trace Reader和GC Worker，一般的goroutine切换至就绪状态时会通过wakep（）函数按需启动新的线程，但是这两者不会，所以通过tryWakeP记录是否需要wakep（）函数。

第四部分代码如下：

```
if gp == nil {
    if _g_.m.p.ptr().schedtick%61 == 0 && sched.runqsize > 0 {
        lock(&sched.lock)
        gp = globrunqget(_g_.m.p.ptr(), 1)
        unlock(&sched.lock)
    }
}
if gp == nil {
    gp, inheritTime = runqget(_g_.m.p.ptr())
}
if gp == nil {
    gp, inheritTime = findrunnable()
}
if _g_.m.spinning {
    resetspinning()
}
```

在schedtick能够被61整除的时候，优先尝试从全局runq中获取任务，其他情况则只从本地runq中获取。大致相当于每调度60次本地runq，就会调度一次全局runq。这样做是为了在保证效率的基础上兼顾公平性，否则本地队列上的两个持续唤醒的goroutine会造成全局队列一直得不到调度。如果前面所有的步骤都没有找到一个待运行的goroutine，就会调用findrunnable（）函数来找任务执行，该函数会一直阻塞，直到找到可运行的goroutine，而且findrunnable（）函数是个十足的质量级函数，稍后再进行介绍。代码执行到这里，gp肯定已经不是nil了，如果M处于spinning状态，就要调用resetspinning（）函数来脱离spinning状态，resetspinning（）函数会调用wakep（）函数按需启动新的线程。

第五部分代码如下：

```
if sched.disable.user && !schedEnabled(gp) {
    lock(&sched.lock)
    if schedEnabled(gp) {
        unlock(&sched.lock)
    } else {
        sched.disable.runnable.pushBack(gp)
        sched.disable.n++
        unlock(&sched.lock)
        goto top
    }
}
```

至此，虽然已经找到了待运行的g，还要确定目前是否处于禁止调度用户协程的状态。在禁止调度用户协程的状态下，gp如果是系统协程就可以正常执行，用户协程需要先通过disable队列暂存起来，调度逻辑跳转到top重新寻找可执行的g。等到允许调度用户协程时，disable队列中的g会被重新加入runq中。

最后一部分代码如下：

```
if tryWakeP {
    wakep()
}
if gp.lockedm != 0 {
    startlockedm(gp)
    goto top
}
execute(gp, inheritTime)
```

第1个if根据tryWakeP来尝试唤醒新的线程，以保证有足够的线程来调度Trace Reader和GC Worker。第2个if判断gp是否有绑定的线程，如果有就必须唤醒绑定的线程来执行gp，而且当前线程也要回到top再来一遍。若gp没有绑定的M，就通过execute（）函数来执行gp。executre（）函数会关联gp和当前的M，将gp的状态设置为_Grunning，并通过gogo（）函数恢复执行上下文，这里不再展开介绍，感兴趣的读者可自行阅读源码。

至此，schedule（）函数就梳理完了，主要逻辑如图6-23所示，整体还算简单明了。我们并没有看到传说中的任务窃取等逻辑，这些逻辑在哪里呢？那就是接下来要梳理的findrunnable（）函数了。

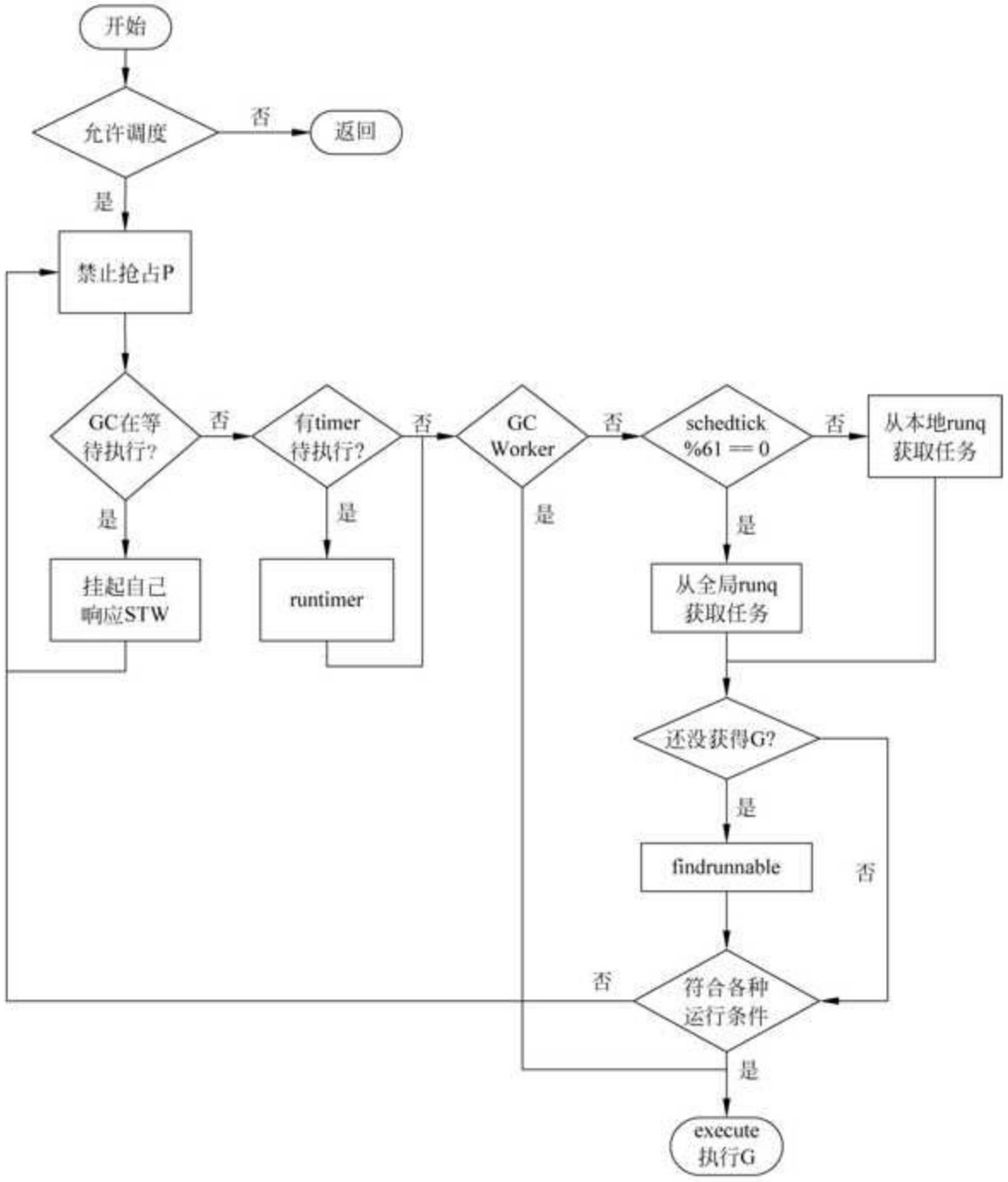

图6-23 schedule调度循环主要逻辑

6.8.2 runtime.findrunnable（）函数

findrunnable（）函数的逻辑可以分成前后两部分，前半部分完成了timer触发、netpoll和任务窃取，后半部分针对的是没有找到任务的情况，会处理GC后台标记任务、按需执行netpoll，实在没有任务就会挂起等待。findrunnable（）函数的主要逻辑如图6-24所示。

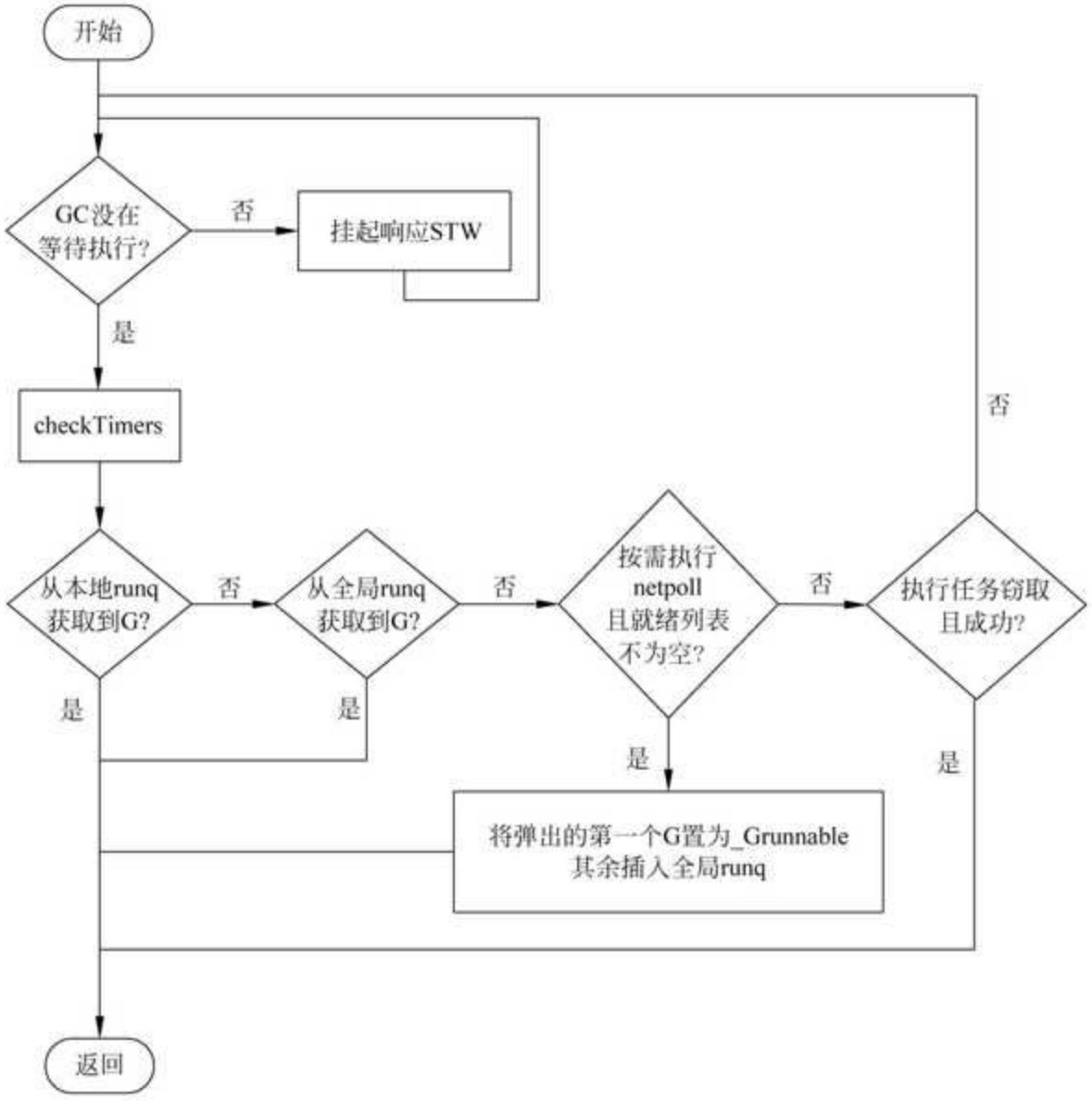

图6-24 findrunnable（）函数的主要逻辑

由于findrunnable（）函数的整体代码量比较大，在这里我们就不全部贴出来了，只把前一半的代码分几部分进行分析。

第一部分代码如下：

```
    _g_ := getg()

top:
    _p_ := _g_.m.p.ptr()
    if sched.gcwaiting != 0 {
        gcstopm()
        goto top
    }
    if _p_.runSafePointFn != 0 {
        runSafePointFn()
    }
```

函数的开头处也是要先检测gcwaiting及runSafePointFn，后续逻辑有可能会阻塞，为了避免GC等待太长时间，检测逻辑被放在了top标签的内部，每次跳转回来都会进行检测。

第二部分代码如下：

```
now, pollUntil, _ := checkTimers(_p_, 0)

if fingwait && fingwake {
    if gp := wakefing(); gp != nil {
        ready(gp, 0, true)
    }
}
if *cgo_yield != nil {
    asmcgocall(*cgo_yield, nil)
}

if gp, inheritTime := runqget(_p_); gp != nil {
    return gp, inheritTime
}

if sched.runqsize != 0 {
    lock(&sched.lock)
    gp := globrunqget(_p_, 0)
    unlock(&sched.lock)
    if gp != nil {
        return gp, false
    }
}
```

调用checkTimers（）函数会运行当前P上所有已经达到触发时间的计时器，这可能会使一些goroutine从_Gwaiting变成_Grunnable状态。接下来按需唤醒finalizer goroutine，然后检查本地runq和全局runq中是否有可运行的任务，找到任务就可以直接返回了。

第三部分代码如下：

```
if netpollinited() && atomic.Load(&netpollWaiters) > 0 &&
    atomic.Load64(&sched.lastpoll) != 0 {
    if list := netpoll(0); !list.empty() {
        gp := list.pop()
        injectglist(&list)
        casgstatus(gp, _Gwaiting, _Grunnable)
        if trace.enabled {
            traceGoUnpark(gp, 0)
        }
        return gp, false
    }
}
```

按需执行一次非阻塞的netpoll，如果返回的列表非空，就把第1个g从列表中pop出来，将剩余的插入全局runq，把这个g的状态置为_Grunnable，然后返回。

第四部分代码如下：

```
procs := uint32(gomaxprocs)
ranTimer := false
if !_g_.m.spinning && 2 * atomic.Load(&sched.nmspinning) >= procs -
    atomic.Load(&sched.npidle) {
    goto stop
}
if !_g_.m.spinning {
    _g_.m.spinning = true
    atomic.Xadd(&sched.nmspinning, 1)
}
const stealTries = 4
for i := 0; i < stealTries; i++{
    stealTimersOrRunNextG := i == stealTries - 1

    for enum := stealOrder.start(fastrand()); !enum.done(); enum.next() {
        if sched.gcwaiting != 0 {
            goto top
        }
        p2 := allp[enum.position()]
        if _p_ == p2 {
            continue
        }

        if stealTimersOrRunNextG && timerpMask.read(enum.position()) {
            tnow, w, ran := checkTimers(p2, now)
            now = tnow

            if w != 0 && (pollUntil == 0 || w < pollUntil) {
                pollUntil = w
            }
            if ran {
                if gp, inheritTime := runqget(_p_); gp != nil {
                    return gp, inheritTime
                }
                ranTimer = true
            }
        }

        if !idlepMask.read(enum.position()) {
            if gp := runqsteal(_p_, p2, stealTimersOrRunNextG); gp != nil {
                return gp, false
            }
        }
    }
}
if ranTimer {
    goto top
}
```

这一大段代码实现了核心的任务窃取逻辑，第1个if判断的含义是，如果当前处于spinning状态的M的数量大于忙碌的P的数量的一半，就让当前M阻塞。目的是避免在GOMAXPROCS较大而程序实际的并发性很低的情况下，造成不必要的CPU消耗。

任务窃取逻辑会循环尝试4次，最后一次才会窃取runnext和timer，也就是说前3次只会从其他P的本地runq中窃取。stealOrder用来实现一个公平的随机窃取顺序，timerpMask和idlepMask用来快速判断指定位置的P是否有timer或者是否空闲。如果ran为true，表示checkTimers（）执行了其他P的timer，可能会使某些goroutine变成_Grunnable状态，所以先检查当前P的本地runq，如果没有找到就跳转回top重来一次。

调度相关的逻辑中会频繁地对runq进行操作，runtime为此专门提供了一组函数，常见的函数例如runqget（）函数、runqput（）函数等，还有上面的runqsteal（）函数也是其中的一个。这些函数的逻辑都比较简明，这里只把runqget（）函数的源码分析一下，目的是看一看P的本地队列如何支持继承时间片，代码如下：

```
func runqget(_p_ *p) (gp *g, inheritTime bool) {
    for {
        next := _p_.runnext
        if next == 0 {
            break
        }
        if _p_.runnext.cas(next, 0) {
            return next.ptr(), true
        }
    }

    for {
        h := atomic.LoadAcq(&_p_.runqhead)
        t := _p_.runqtail
        if t == h {
            return nil, false
        }
        gp := _p_.runq[h%uint32(len(_p_.runq))].ptr()
        if atomic.CasRel(&_p_.runqhead, h, h+1) {
            return gp, false
        }
    }
}
```

原来是通过runnext字段实现的，只有取自runnext的g对应的inheritTime才是true，其他本地runq中的g的返回值都为false，也就是不会继承时间片。相应地，如果某个g需要继承时间片，runqput（）函数就会把它设置到runnext，感兴趣的读者可以自行查看源代码。

本节就讲解到这里，主要对schedule（）函数和findrunnable（）函数进行了简要梳理，基本了解了每一轮调度循环都会做些什么。应该说schedule（）函数就是调度循环的实现，但是当goroutine开始执行用户代码后，执行流是如何再回到runtime中去调用schedule（）函数的呢？这就是6.9节中要探索的抢占式调度。

6.9 抢占式调度

就像操作系统要负责线程的调度一样，Go的runtime要负责goroutine的调度。现代操作系统调度线程都是抢占式的，我们不能依赖用户代码主动让出CPU，或者因为IO、锁等待而让出，这样会造成调度的不公平。基于经典的时间片算法，当线程的时间片用完之后，会被时钟中断给打断，调度器会将当前线程的执行上下文进行保存，然后恢复下一个线程的上下文，分配新的时间片，令其开始执行。这种抢占对于线程本身是无感知的，由系统底层支持，不需要开发人员特殊处理。

基于时间片的抢占式调度有个明显的优点，能够避免CPU资源持续被少数线程占用，从而使其他线程长时间处于饥饿状态。goroutine的调度器也用到了时间片算法，但是和操作系统的线程调度还是有些区别的，因为整个Go程序都运行在用户态，所以不能像操作系统那样利用时钟中断来打断运行中的goroutine。也得益于完全在用户态实现，goroutine的调度切换更加轻量。

本节就来实际研究一下，runtime到底是如何抢占运行中的goroutine的。为了避免过于枯燥乏味，先不直接解读源码，而是先做个实验，准备的示例代码如下：

```
//第 6 章/code_6_2.go
package main

import "fmt"
func main() {
    go func(n int) {
        for {
            n++
            fmt.Println(n)
        }
    }(0)
    for {}
}
```

6.9.1 Go 1.13的抢占式调度

笔者使用的是Go 1.13.15版，build完成后运行得到的是可执行文件。程序会如你所料地运行起来，飞快地打印出一行行递增的数字。不要着急，让程序多运行一会儿，用不了太长时间你就会发现程序突然停了，不再继续打印了。在笔者测试的64位Linux系统上，最大数字没有超过500000，程序似乎就停住了。是真的停住了吗？如果用top命令查看，就会发现CPU占用达到100%。也就是说程序还在运行中，并且占满了一个CPU核心。

为了弄清楚程序到底在做什么，我们使用调试器delve查看一下当前所有的goroutine的状态，执行的命令如下：

```
(dlv) grs
* Goroutine 1 - User: ./main.go:12 main.main (0x48cf9e) (thread 17835)
  Goroutine 2 - User: /root/go1.13/src/runtime/proc.go:305 runtime.gopark (0x42b4e0) [force
gc (idle)]
  Goroutine 3 - User: /root/go1.13/src/runtime/proc.go:305 runtime.gopark (0x42b4e0) [GC
sweep wait]
  Goroutine 4 - User: /root/go1.13/src/runtime/proc.go:305 runtime.gopark (0x42b4e0) [GC
scavenge wait]
  Goroutine 5 - User: /root/go1.13/src/runtime/proc.go:305 runtime.gopark (0x42b4e0) [GC
worker (idle)]
  Goroutine 6 - User: /root/go1.13/src/runtime/proc.go:305 runtime.gopark (0x42b4e0) [GC
worker (idle)]
  Goroutine 17 - User: /root/go1.13/src/runtime/proc.go:305 runtime.gopark (0x42b4e0)
[finalizer wait]
  Goroutine 18 - User: ./main.go:9 main.main.func1 (0x48cfe7) (thread 17837)
[8 goroutines]
```

可以看到一共有8个goroutine，除了1号和18号是在执行用户代码外，其他都与GC相关且都处于空闲或等待状态。1号goroutine正在执行main（）函数，main.go的第12行就是main（）函数最后空的for循环，说明它一直在这里循环，占满一个CPU核心的应该就是它。18号goroutine执行的位置在func1（）函数中，对照源码行号来看就是协程中的fmt.Println（）函数。我们通过调试器切换到18号goroutine，然后查看它的调用栈，执行的命令如下：

```
(dlv) gr 18
Switched from 1 to 18 (thread 17837)
(dlv) bt
0  0x0000000000455553 in runtime.futex
   at /root/go1.13/src/runtime/sys_linux_amd64.s:536
1  0x0000000000451700 in runtime.systemstack_switch
   at /root/go1.13/src/runtime/asm_amd64.s:330
2  0x0000000000417457 in runtime.gcStart
   at /root/go1.13/src/runtime/mgc.go:1287
3  0x000000000040b026 in runtime.mallocgc
   at /root/go1.13/src/runtime/malloc.go:1115
4  0x0000000000408f8b in runtime.convT64
   at /root/go1.13/src/runtime/iface.go:352
5  0x000000000048cfe7 in main.main.func1
   at ./main.go:9
6  0x0000000000453651 in runtime.goexit
   at /root/go1.13/src/runtime/asm_amd64.s:1357
```

按照这个调用栈，结合我们看到的现象进行分析：协程中要调用fmt.Println（）函数，该函数的参数类型是interface{}，所以要先调用runtime.convT64（）函数来把一个int64（amd64平台上的int本质上是int64）转换为interface{}类型，而convT64（）函数内部需要分配内存，经过多次循环之后达到了GC阈值，要先进行GC才能继续执行，所以mallocgc（）函数调用gcStart（）函数开始执行GC。后续能够看出gcStart（）函数内部切换至了系统栈，然后发生了等待阻塞。

我们通过源码看一下mgc.go的1287行到底在干什么，代码如下：

```
systemstack(stopTheWorldWithSema)
```

原来是通过systemstack（）函数切换至系统栈，然后调用stopTheWorldWithSema（）函数，看来是

要STW，但为什么会阻塞呢？这就要讲讲STW的实现原理了。6.5.4节在解释schedt的gcwaiting字段时有过简单介绍，这里摘选了该函数的核心代码来看一下，代码如下：

```
lock(&sched.lock)
sched.stopwait = gomaxprocs
atomic.Store(&sched.gcwaiting, 1)
preemptall()

_g_.m.p.ptr().status = _Pgcstop
sched.stopwait--

for _, p := range allp {
    s := p.status
    if s == _Psyscall && atomic.Cas(&p.status, s, _Pgcstop) {
        if trace.enabled {
            traceGoSysBlock(p)
            traceProcStop(p)
        }
        p.syscalltick++
        sched.stopwait--
    }
}

for {
    p := pidleget()
    if p == nil {
        break
    }
    p.status = _Pgcstop
    sched.stopwait--
}
wait := sched.stopwait > 0
unlock(&sched.lock)

if wait {
    for {
        if notetsleep(&sched.stopnote, 100 * 1000) {
            noteclear(&sched.stopnote)
            break
        }
        preemptall()
    }
}
```

先根据gomaxprocs的值设置stopwait，实际上就是P的个数，然后把gcwaiting置为1，并通过preemptall（）函数去抢占所有运行中的P。preemptall（）函数会遍历allp这个切片，调用preemptone（）函数逐个抢占处于_Prunning状态的P。接下来把当前M持有的P置为_Pgcstop状态，并把stopwait减去1，表示当前P已经被抢占了，然后遍历allp，把所有处于_Psyscall状态的P置为_Pgcstop状态，并把stopwait减去对应的数量。再循环通过pidleget（）函数取得所有空闲的P，都置为_Pgcstop状态，从stopwait减去相应的数量。最后通过判断stopwait是否大于0，也就是是否还有没被抢占的P，来确定是否需要等待。如果需要等待，就以100μm为超时时间，在sched.stopnote上等待，超时后再次通过preemptall（）函数抢占所有P。因为preemptall（）函数不能保证一次就成功，所以需要循环。最后一个响应gcwaiting的工作线程在自我挂起之前，会通过stopnote唤醒当前

线程，STW也就完成了。

实际用来执行抢占的preemptone（）函数的代码如下：

```
func preemptone(_p_ *p) bool {
    mp := _p_.m.ptr()
    if mp == nil || mp == getg().m {
        return false
    }
    gp := mp.curg
    if gp == nil || gp == mp.g0 {
        return false
    }

    gp.preempt = true

    gp.stackguard0 = stackPreempt
    return true
}
```

第1个if判断是为了避开当前M，不能抢占自己。第2个if用于避开处于系统栈的M，不能打断调度器自身，而所谓的抢占，就是把g的preempt字段设置成true，并把stackguard0这个栈增长检测的下界设置成stackPreempt。这样就能实现抢占了吗？

还记不记得之前反编译很多函数的时候，都会看到编译器安插在函数头部的栈增长代码？例如对于一个递归式的斐波那契函数，代码如下：

```
//第 6 章/code_6_3.go
func fibonacci(n int) int {
    if n < 2 {
        return 1
    }
    return fibonacci(n - 1) + fibonacci(n - 2)
}
```

经过反编译后，可以看到最终生成的汇编指令如下：

```
TEXT main.fibonacci(SB) /root/work/sched/main.go
func fibonacci(n int) int {
  0x4526e0      64488b0c25f8ffffff    MOVQ FS:0xfffffff8, CX
  0x4526e9      483b6110              CMPQ 0x10(CX), SP
  0x4526ed      766e                  JBE 0x45275d
  0x4526ef      4883ec20              SUBQ $0x20, SP
  0x4526f3      48896c2418            MOVQ BP, 0x18(SP)
  0x4526f8      488d6c2418            LEAQ 0x18(SP), BP
        if n < 2 {
  0x4526fd      488b442428            MOVQ 0x28(SP), AX
  0x452702      4883f802              CMPQ $0x2, AX
  0x452706      7d13                  JGE 0x45271b
            return 1
  0x452708      48c744243001000000    MOVQ $0x1, 0x30(SP)
  0x452711      488b6c2418            MOVQ 0x18(SP), BP
  0x452716      4883c420              ADDQ $0x20, SP
  0x45271a      c3                    RET
        return fibonacci(n-1) + fibonacci(n-2)
  0x45271b      488d48ff              LEAQ -0x1(AX), CX
  0x45271f      48890c24              MOVQ CX, 0(SP)
  0x452723      e8b8ffffff            CALL main.fibonacci(SB)
  0x452728      488b442408            MOVQ 0x8(SP), AX
  0x45272d      4889442410            MOVQ AX, 0x10(SP)
  0x452732      488b4c2428            MOVQ 0x28(SP), CX
  0x452737      4883c1fe              ADDQ $-0x2, CX
  0x45273b      48890c24              MOVQ CX, 0(SP)
  0x45273f      e89cffffff            CALL main.fibonacci(SB)
  0x452744      488b442410            MOVQ 0x10(SP), AX
  0x452749      4803442408            ADDQ 0x8(SP), AX
  0x45274e      4889442430            MOVQ AX, 0x30(SP)
  0x452753      488b6c2418            MOVQ 0x18(SP), BP
  0x452758      4883c420              ADDQ $0x20, SP
  0x45275c      c3                    RET
func fibonacci(n int) int {
  0x45275d      e85e7affff            CALL runtime.morestack_noctxt(SB)
  0x452762      e979ffffff            JMP main.fibonacci(SB)
```

还是转换成等价的Go风格的伪代码更容易理解，也更直观，伪代码如下：

```
func fibonacci(n int) int {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    return fibonacci(n-1) + fibonacci(n-2)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

实际上，编译器安插在函数开头的检测代码会有几种不同的形式，具体用哪种形式是根据函数栈帧

的大小来定的。不管怎样检测，最终目的都是一样的，就是避免当前函数的栈帧超过已分配栈空间的下界，也就是通过提前分配空间来避免栈溢出。

执行抢占的时候，preemptone（）函数设置的那个stackPreempt是个常量，将其赋值给stackguard0之后，就会得到一个很大的无符号整数，在64位系统上是0xfffffffffffffade，在32位系统上是0xfffffade。实际的栈不可能位于这个地方，也就是说SP寄存器始终会小于这个值，因此，只要代码执行到这里，肯定就会去执行runtime.morestack_noctxt（）函数，而morestack_noctxt（）函数只是直接跳转到runtime.morestack（）函数，而后者又会调用runtime.newstack（）函数。newstack（）函数内部检测到如果stackguard0＝stackPreempt这个常量，就不会真正进行栈增长操作，而是去调用gopreempt_m，后者又会调用goschedImpl（）函数。最终goschedImpl（）函数会调用schedule（）函数，还记得schedule（）函数开头检测gcwaiting的if语句吗？工作线程就是在那些地方响应STW的。执行流能够一路走到schedule（）函数，这就是通过栈增长检测代码实现goroutine抢占的原理。

现在就比较容易理解我们实验程序停住的原因了，执行fmt.Println（）函数的goroutine需要执行GC，进而发起了STW，而main（）函数中的空for循环因为没有调用任何函数，所以没有机会执行栈增长检测代码，也就不能被抢占了。

如图6-25所示，Go 1.13版本及之前的抢占依赖于goroutine检测到stackPreempt标识而自动让出，并不算是真正意义上的抢占。一个空的for循环就让程序挂起了，这可真是个隐患。虽然我们不会在生产环境写出这种代码，但是对于调度器来讲，毕竟是个缺陷，所以在Go 1.14版本中，这个问题被解决了。

6.9.2 Go 1.14的抢占式调度

Go 1.14实现了真正的抢占式调度，从现象来看，还是采用第6章/code_6_2.go那个实验代码，用Go 1.14版生成可执行文件，再运行就不会阻塞了。从Go 1.14版开始，空的for循环这类代码也能被抢占了，就像操作系统通过中断打断运行中的线程一样。

这种真正的抢占是如何实现的呢？在UNIX系操作系统上是基于信号实现的，所以也称为异步抢占。接下来就以Linux系统为例，实际研究一下。这次需要先从源码开始，对比一下Go 1.14版与Go 1.13版有哪些不同，了解了具体的细节之后再通过调试等手段进行相关实践。

下面就是Go 1.14版runtime.preemptone（）函数的源码，可以看到比之前的Go 1.13版多出来了最后的那个if语句块，代码如下：

```
func preemptone(_p_ *p) bool {
    mp := _p_.m.ptr()
    if mp == nil || mp == getg().m {
        return false
    }
    gp := mp.curg
    if gp == nil || gp == mp.g0 {
        return false
    }
    gp.preempt = true
    gp.stackguard0 = stackPreempt
    if preemptMSupported && debug.asyncpreemptoff == 0 {
        _p_.preempt = true
        preemptM(mp)
    }
    return true
}
```

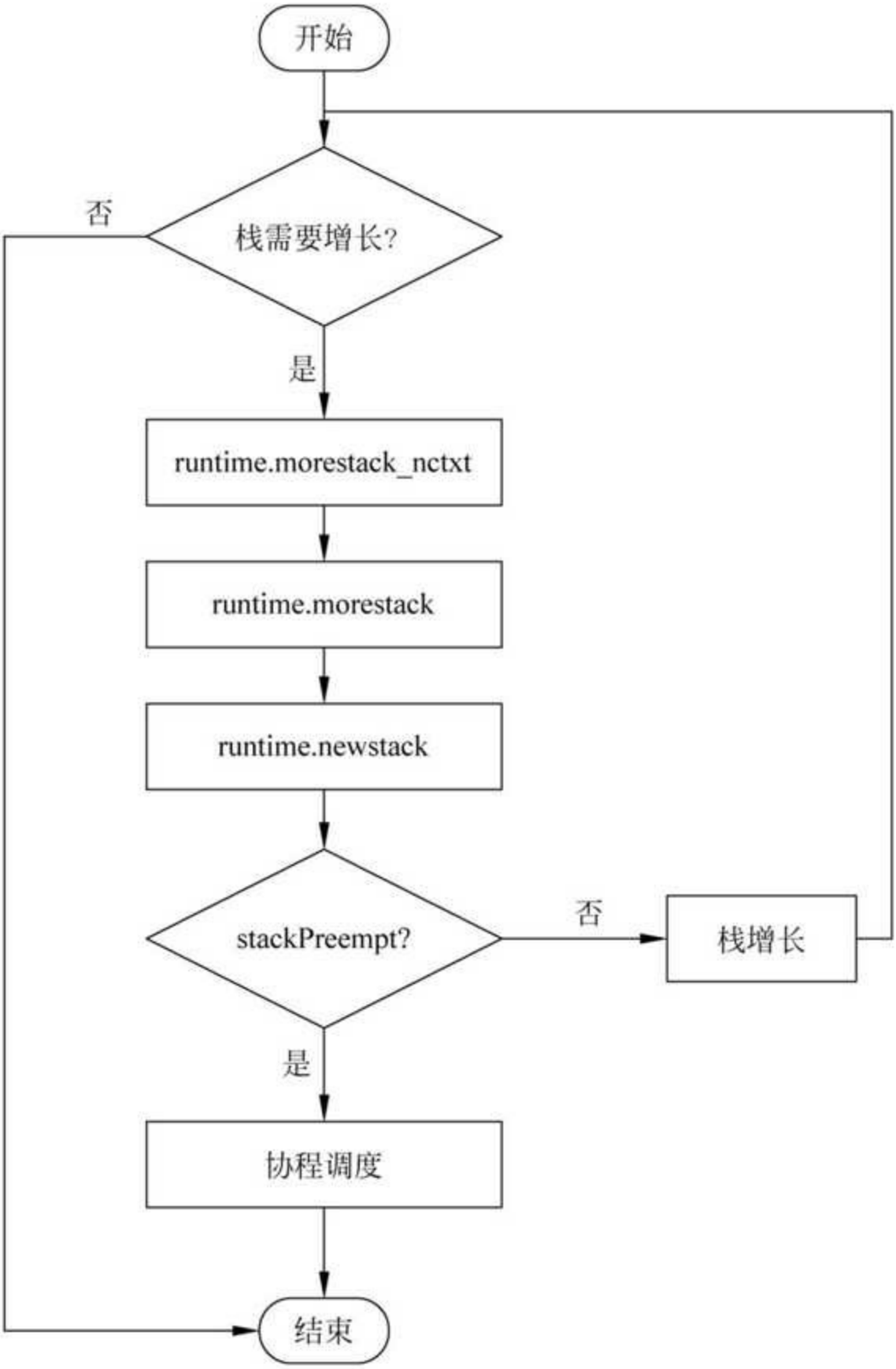

图6-25 Go 1.13版本中的抢占式调度流程

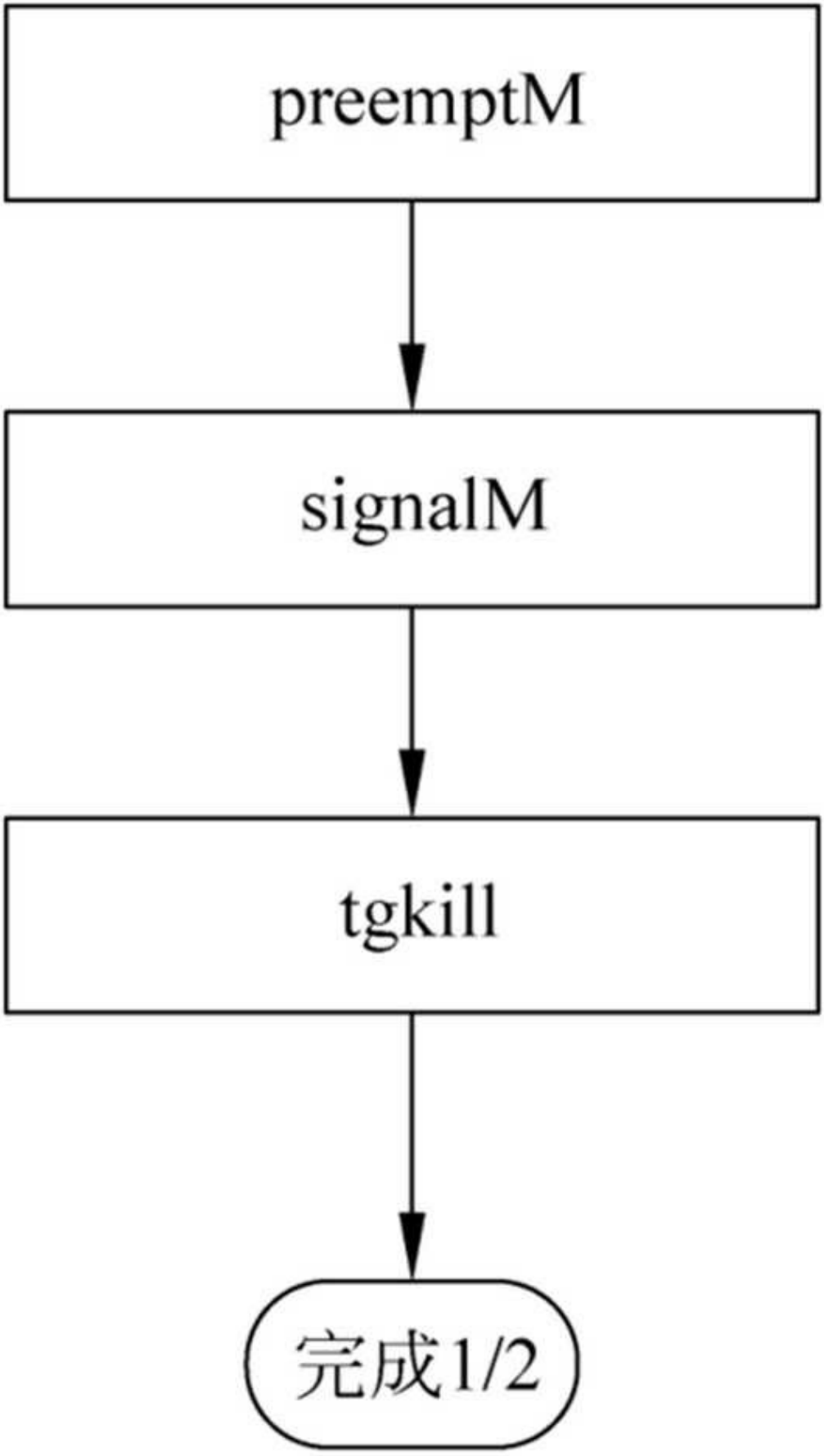
preemptM
signalM
tgkill
完成1/2

图6-26 Linux系统中异步抢占的前一半工作

其中的preemptMSupported是个常量，因为受硬件特性的限制，在某些平台上是无法支持这种抢占的。debug.asyncpreemptoff则是让用户可以通过GODEBUG环境变量来禁用异步抢占，默认情况下是被启用的。在P的数据结构中也新增了一个preempt字段，这里会把它设置为true。实际的抢占操作是由preemptM（）函数完成的。

preemptM（）函数的主要逻辑就是通过runtime.signalM（）函数向指定M发送sigPreempt信号。至于signalM（）函数，就是调用操作系统的信号相关系统调用，将指定信号发送给目标线程。至此，异步抢占逻辑的主要工作就算完成了前一半，如图6-26所示，信号已经发出去了。

异步抢占工作的后一半就要由接收到信号的工作线程来完成了。还是先定位到相应的源码，runtime.sighandler（）函数就是负责处理接收的信号的，其中有这样一个if语句，代码如下：

```
if sig == sigPreempt {
    doSigPreempt(gp, c)
}
```

如果收到的信号是sigPreempt，就调用doSigPreempt（）函数。doSigPreempt（）函数的代码如下：

```
func doSigPreempt(gp * g, ctxt * sigctxt) {
    if wantAsyncPreempt(gp) && isAsyncSafePoint(gp, ctxt.sigpc(), ctxt.sigsp(), ctxt.siglr())
    {
        ctxt.pushCall(funcPC(asyncPreempt))
    }

    atomic.Xadd(&gp.m.preemptGen, 1)
    atomic.Store(&gp.m.signalPending, 0)

    if GOOS == "darwin" {
        atomic.Xadd(&pendingPreemptSignals, -1)
    }
}
```

重点就在于第1个if语句块，它先通过wantAsyncPreempt（）函数确认runtime确实想要对指定的G实施异步抢占，再通过isAsyncSafePoint（）函数确认G当前执行上下文是能够安全地进行异步抢占的。实际看一下wantAsyncPreempt（）函数的源码，代码如下：

```
func wantAsyncPreempt(gp * g) bool {
    return (gp.preempt || gp.m.p != 0 && gp.m.p.ptr().preempt) && readgstatus (gp) & ^_Gscan
== _Grunning
}
```

它会同时检查G和P的preempt字段，并且G当前需要处于_Grunning状态。在每轮调度循环中，P和G的preempt字段都会被置为false，所以这个检测能够避免刚刚切换至一个新的G后马上又被抢占。isAsyncSafePoint（）函数的代码比较复杂且涉及较多其他细节，这里就不展示源码了。它从以下几个方面来保证在当前位置进行异步抢占是安全的：

（1）可以挂起G并安全地扫描它的栈和寄存器，没有潜在的隐藏指针，而且当前并没有打断一个写屏障。

（2）G还有足够的栈空间来注入一个对asyncPreempt（）函数的调用。

（3）可以安全地和runtime进行交互，例如未持有runtime相关的锁，因此在尝试获得锁时不会造成

死锁。

以上两个函数都确认无误后，才通过pushCall向G的执行上下文中注入一个函数调用，要调用的目标函数是runtime.asyncPreempt（）函数。这是一个汇编函数，它会先把各个寄存器的值保存在栈上，也就是先将现场保存到栈上，然后调用runtime.asyncPreempt2（）函数。asyncPreempt2（）函数的代码如下：

```
func asyncPreempt2() {
    gp := getg()
    gp.asyncSafePoint = true
    if gp.preemptStop {
        mcall(preemptPark)
    } else {
        mcall(gopreempt_m)
    }
    gp.asyncSafePoint = false
}
```

其中preemptStop主要在GC标记时被用来挂起运行中的goroutine，preemptPark（）函数会把当前g切换至_Gpreempted状态，然后调用schedule（）函数，而通过preemptone（）函数发起的异步抢占会调用gopreempt_m（）函数，它最终也会调用schedule（）函数。至此，整个抢占过程就完整地实现了。

关于如何在执行上下文中注入一个函数调用，我们在这里结合AMD64架构做一下更细致的说明。runtime源码中与AMD64架构对应的pushCall（）函数的代码如下：

```
func (c *sigctxt) pushCall(targetPC uintptr) {
    pc := uintptr(c.rip())
    sp := uintptr(c.rsp())
    sp -= sys.PtrSize
    *(*uintptr)(unsafe.Pointer(sp)) = pc
    c.set_rsp(uint64(sp))
    c.set_rip(uint64(targetPC))
}
```

先把SP向下移动一个指针大小的位置，把PC的值存入栈上SP指向的位置，然后将PC的值更新为targetPC。这样就模拟了一条CALL指令的效果，如图6-27所示，栈上存入的PC的旧值就相当于返回地址。此时整个执行上下文的状态就像是goroutine在被信号打断的位置额外执行了一条CALL targetPC指令，由于执行流刚刚跳转到targetPC地址处，所以还没来得及执行目标地址处的指令。

当sighandler（）函数处理完信号并返回之后，被打断的goroutine得以继续执行，会立即调用被注入的asyncPreempt（）函数。经过一连串的函数调用，最终执行到schedule（）函数。异步抢占的后一半工作流程如图6-28所示。

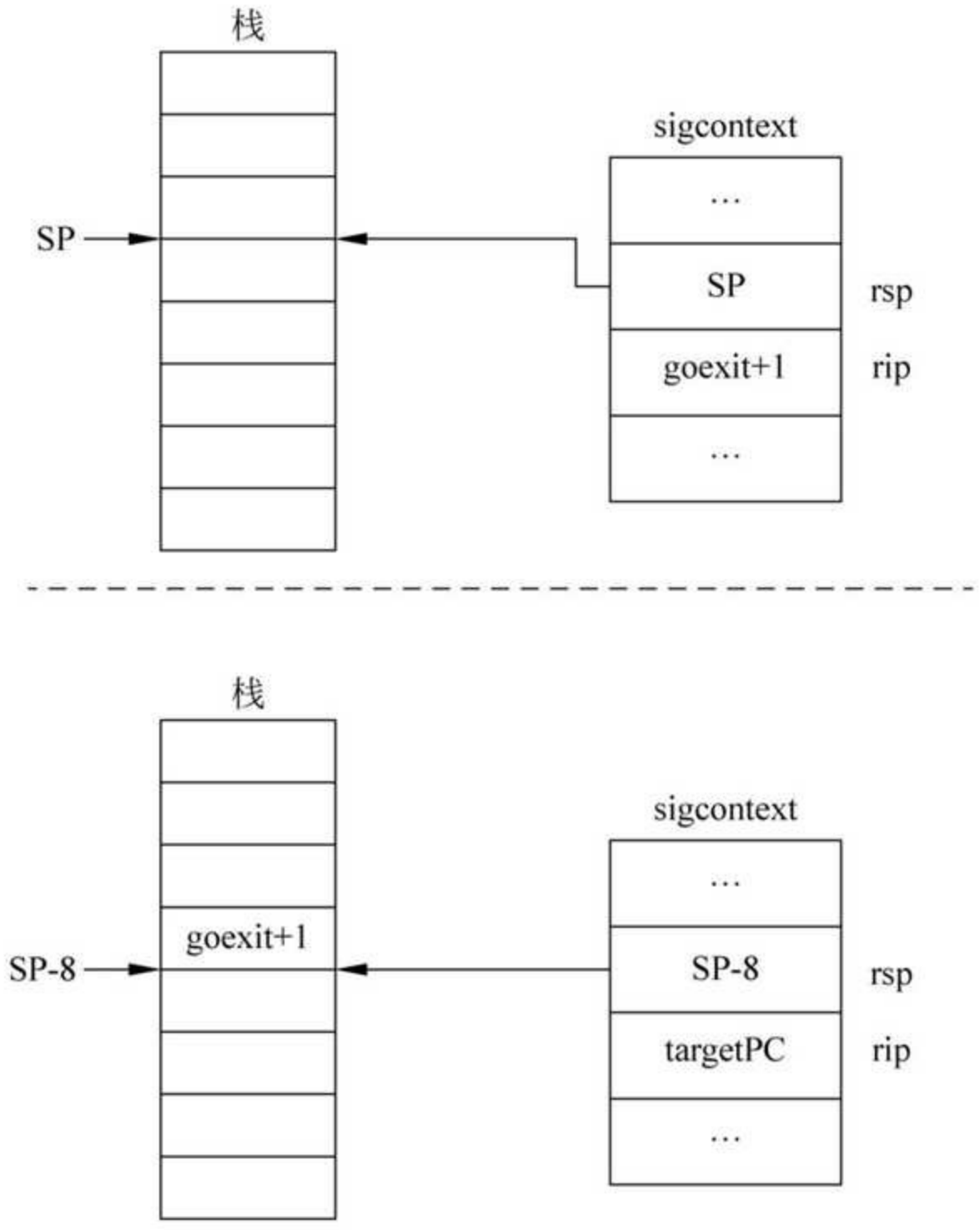

图6-27 AMD64架构下注入一个函数调用

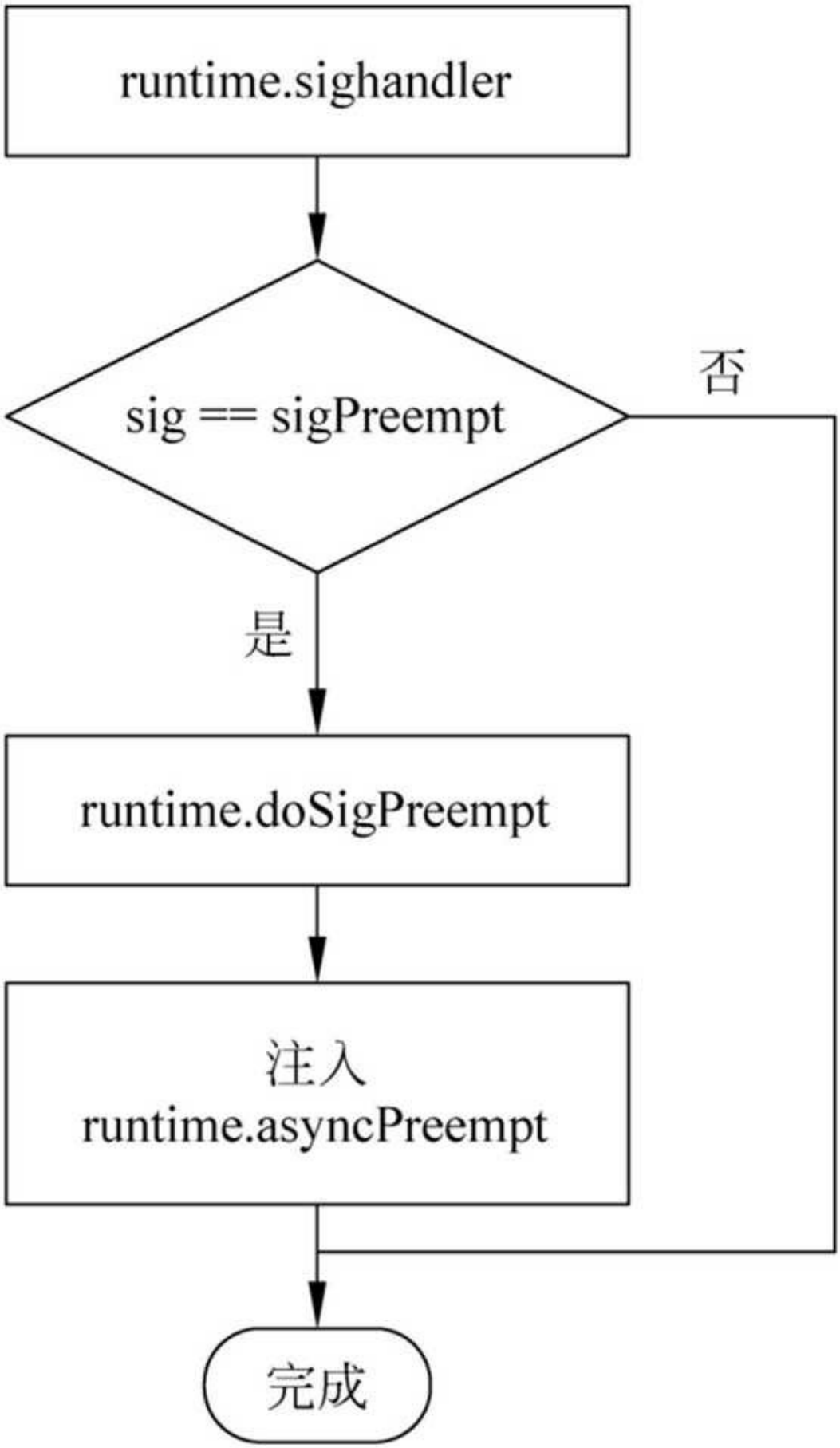
runtime.sighandler
sig == sigPreempt
否
是
runtime.doSigPreempt
注入
runtime.asyncPreempt
完成

图6-28 Linux系统中异步抢占的后一半工作

了解了整个流程之后，我们再来做一个很简单的实验。还是采用第6章/code_6_2.go文件中的代码，用Go 1.14版编译之后再运行，可以发现程序会一直输出，不再阻塞。这时，用dlv调试器附加到目标进程，并且在runtime.asyncPreempt2（）函数中设置断点，然后让程序继续运行。等到命中断点后，查看调用栈的回溯，命令如下：

```
(dlv) bt
0  0x00000000004302f0 in runtime.asyncPreempt2
   at /root/go1.14/src/runtime/preempt.go:302
1  0x000000000045d91b in runtime.asyncPreempt
   at /root/go1.14/src/runtime/preempt_amd64.s:50
2  0x0000000000491daf in main.main
   at ./main.go:12
3  0x00000000004318ea in runtime.main
   at /root/go1.14/src/runtime/proc.go:203
4  0x000000000045bff1 in runtime.goexit
   at /root/go1.14/src/runtime/asm_amd64.s:1373
```

从栈回溯来看是main（）函数调用了asyncPreempt（）函数，而main.go的12行正是那个空的for循环，它没有调用任何函数，这个调用就是被pushCall（）函数注入的。

还有一种方式，可以通过GODEBUG环境变量来禁用异步抢占，此时会发现Go 1.14版编译的程序运行一段时间后也会阻塞，命令如下：

```
$ GODEBUG = 'asyncpreemptoff = 1' ./code_6_2
```

另外还有一点，如果把协程中用来打印的fmt.Println（）函数换成println（）函数，则会发现运行很久都不会阻塞，即使是Go 1.13版编译的程序也是如此。这是因为println（）函数不需要额外分配内存，感兴趣的读者可以自行尝试。本节关于抢占式调度的探索就讲解到这里。

6.10 timer

6.10.1 一个示例

在6.7.2节介绍协程创建时我们使用了一个hello goroutine的例子，其中main goroutine创建的hello goroutine还没执行，main.main（）函数就返回了，然后exit（）函数就结束了进程。下面我们让main goroutine在timer中等待一下，让hello goroutine有时间得以运行，代码如下：

```
//第 6 章/code_6_4.go
package main
import "time"
func hello(name string){
    println("Hello ", name)
}
func main(){
    name := "Goroutine"
    go hello(name)
    time.Sleep(time.Second)
}
```

当main goroutine执行到time.Sleep（）函数时，会创建一个timer对象，timer对象会记录timer的触发时间和时间到达时需要执行的回调函数，以及是哪个协程在等待timer等信息。

设置好timer对象后，就会调用gopark（）函数，使当前goroutine挂起，让出CPU。main goroutine的状态会从_Grunning改为_Gwaiting，不会进入当前P的本地runq，而是进到刚刚创建的那个timer中等待，随后hello goroutine有机会得到调度执行，如图6-29所示。

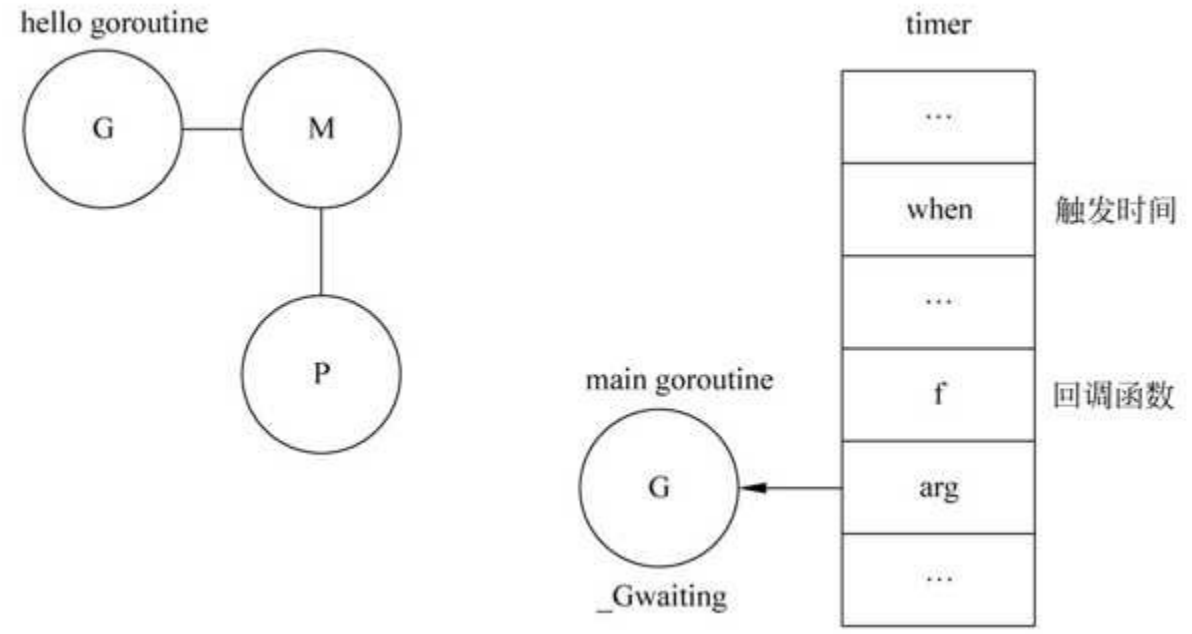

图6-29 main goroutine等待timer时hello goroutine得到调度

等到timer触发时间到达后，回调函数timer.f（）得以执行，对于time.Sleep（）函数而言，timer.f被设置为goroutineReady（）函数，函数的代码如下：

```
func goroutineReady(arg interface{}, seq uintptr) {
    goready(arg.(*g), 0)
}
```

goroutineReady（）函数直接调用goready（）函数，它会切换到g0栈，并执行runtime.ready（）函数。待ready的协程自然是main goroutine，此时它的状态是_Gwaiting，接下来会被修改为_Grunnable，表示它又可以被调度执行了，然后，它会被放到当前P的本地runq中，所以，timer等待的时间到达后，main goroutine又可以得到调度执行了。接下来，在main goroutine恢复执行后，main.main（）函数执行后返回，进程退出。

通过这个修改后的例子，我们初步了解了协程等待timer时让出与恢复的大致过程，接下来我们展开一些细节。

6.10.2 数据结构

在runtime中，每个计时器都用一个timer对象表示。在Go 1.14版本及后续的版本中，timer的结构定义代码如下：

```
type timer struct {
    pp       puintptr
    when     int64
    period   int64
    f        func(interface{}, uintptr)
    arg      interface{}
    seq      uintptr
    nextwhen int64
    status   uint32
}
```

其中各个字段的用途如表6-8所示。

表6-8 timer数据结构各个字段的用途

字　段	用　途
pp	从 Go 1.14 版本开始，runtime 使用堆结构来管理 timer，而每个 P 中都有一个最小堆。这个 pp 字段本质上是个指针，表示当前 timer 被放置在哪个 P 的堆中
when	时间戳，精确到纳秒级别，表示 timer 何时会触发
period	表示周期性 timer 的触发周期，单位也是纳秒
f	回调函数，当 timer 触发的时候会调用它，它就是要定时完成的任务
arg	调用 f 时传给它的第 1 个参数
seq	起到一个序号的作用，主要被 netpoller 使用
nextwhen	时间戳，在修改 timer 时用来记录修改后的触发时间。修改 timer 时不会直接修改 when 字段，这样会打乱堆的有序状态，所以先更新 nextwhen 并将 status 设置为已修改状态，等到后续调整 timer 在堆中位置时再更新 when 字段
status	表示 timer 当前的状态，取值自 runtime 中一组预定义的常量

针对status字段，runtime源码中定义了10种状态，如表6-9所示。

表6-9 status不同状态的含义

状　态	含　义
timerNoStatus	表示没有状态，一般是刚分配还没添加到堆中的 timer
timerWaiting	表示处于等待状态的 timer，已经被添加到某个 P 的堆中等待触发
timerRunning	表示运行中的 timer，是一个短暂的状态，也就是触发后 timer.f 执行期间
timerDeleted	表示已删除但还未被移除的 timer，仍位于某个 P 的堆中，但是不会触发
timerRemoving	表示正在被从堆中移除，也是一个短暂的状态
timerRemoved	表示已经被从堆中移除了，一般是从 Deleted 到 Removing 最终到 Removed
timerModifying	表示 timer 当前正在被修改，也是一个短暂的状态
timerModifiedEarlier	表示 timer 被修改到一个更早的时间，新时间存在于 nextwhen 中
timerModifiedLater	表示 timer 被修改到一个更晚的时间，新时间存在于 nextwhen 中
timerMoving	表示一个被修改过的 timer 正在被移动，也是一个短暂的状态

在runtime.p中也有一组专门用来支持timer的字段，节选的相关代码如下：

```
type p struct {
    timer0When            uint64
    timerModifiedEarliest uint64
    timersLock            mutex
    timers               []*timer
    numTimers             uint32
    adjustTimers          uint32
    deletedTimers         uint32
}
```

其中各个字段的用途如表6-10所示，timerModifiedEarliest是在Go 1.16版本中新增的，其余字段都是在Go 1.14版本中重构timer时引入的。

表6-10 runtime.p中支持timer的字段及其用途

字　段	用　途
timer0When	表示位于最小堆堆顶的 timer 的触发时间，也就是复制其 when 字段
timerModifiedEarliest	表示所有已知的处于 timerModifiedEarlier 状态的 timer 中，nextwhen 值最小的那个 timer 的 nextwhen 值。和 timer0When 相同，都是通过 atomic 函数来操作的，两者之间更小的那个被认为是接下来第 1 个要触发的 timer 的触发时间
timersLock	主要用来保护 P 中的 timer 堆，也就是 timers 切片
timers	timers 切片就是用来存放 timer 的最小堆，这是一个 4 叉堆，与传统 2 叉堆相比有着更少的层数，也能更好地利用缓存的局部性原理
numTimers	记录的是堆中 timer 的总数，应该与 timers 切片的长度一致
adjustTimers	记录的是堆中处于 timerModifiedEarlier 状态的 timer 的总数
deletedTimers	记录的是堆中已删除但还未被移除的 timer 的总数

6.10.3 操作函数

runtime中有一组与timer相关的函数，其中有最底层的siftupTimer（）函数和siftdownTimer（）函数，它们用来在最小堆中根据when字段的值上下移动timer，以维持堆的有序性。doaddtimer（）函数、dodeltimer（）函数及dodeltimer0（）函数用来将指定的timer添加到堆中，或者将其从堆中移除，这也是偏底层一些的函数，不会被runtime中除timer以外的其他模块直接调用。像

addtimer（）、deltimer（）、modtimer（）和resettimer（）这几个函数就属于timer模块提供的接口了，runtime中的其他模块可以直接调用这些函数，例如6.11节要讲解的netpoller就会用到这组函数。至于startTimer（）、stopTimer（）和resetTimer（）这些函数，只是对这组接口函数进行了简单包装，并通过linkname机制链接到time包，提供给标准库使用。最后，还有被调度器调用的adjusttimers（）函数、runtimer（）函数和clearDeletedTimers（）函数，它们会对timer堆进行维护，以及运行那些到达触发时间的timer。

在上述函数中，像addtimer（）、deltimer（）和modtimer（）这些函数还是比较简单的，接下来我们就逐一看一下Go 1.14版本中它们的源码。

1. 添加

addtimer（）函数的源码如下：

```
func addtimer(t *timer) {
    if t.when < 0 {
        t.when = maxWhen
    }
    if t.status != timerNoStatus {
        throw("addtimer called with initialized timer")
    }
    t.status = timerWaiting

    when := t.when

    pp := getg().m.p.ptr()
    lock(&pp.timersLock)
    cleantimers(pp)
    doaddtimer(pp, t)
    unlock(&pp.timersLock)

    wakeNetPoller(when)
}
```

先对t的when和status字段进行校验及修正，然后对pp的timersLock加锁，在锁的保护下调用cleantimers（）函数清理堆顶，可能存在已被删除的timer，再调用doaddtimer（）函数把t添加到堆中。添加操作至此就完成了，然后进行解锁操作。最后调用的wakeNetPoller（）函数会根据when的值按需唤醒阻塞中的netpoller，目的是让调度线程能够及时处理timer。

2. 删除

deltimer（）函数用来删除一个timer，这里的删除操作主要是修改timer的状态，并不是从堆中移除，函数的代码如下：

```
func deltimer(t *timer) bool {
    for {
        switch s := atomic.Load(&t.status); s {
        case timerWaiting, timerModifiedLater:
            mp := acquirem()
            if atomic.Cas(&t.status, s, timerModifying) {
                tpp := t.pp.ptr()
                if !atomic.Cas(&t.status, timerModifying, timerDeleted) {
                    badTimer()
                }
                releasem(mp)
                atomic.Xadd(&tpp.deletedTimers, 1)
                return true
            } else {
                releasem(mp)
            }
        case timerModifiedEarlier:
            mp := acquirem()
            if atomic.Cas(&t.status, s, timerModifying) {
                tpp := t.pp.ptr()
                atomic.Xadd(&tpp.adjustTimers, -1)
                if !atomic.Cas(&t.status, timerModifying, timerDeleted) {
                    badTimer()
                }
                releasem(mp)
                atomic.Xadd(&tpp.deletedTimers, 1)
                return true
            } else {
                releasem(mp)
            }
        case timerDeleted, timerRemoving, timerRemoved:
            return false
        case timerRunning, timerMoving:
            osyield()
        case timerNoStatus:
            return false
        case timerModifying:
            osyield()
        default:
            badTimer()
        }
    }
}
```

该函数不会改动timer堆，所以不需要对timersLock加锁。正常的处理流程是对处在timerWaiting、timerModifiedLater和timerModifiedEarlier这3种状态的timer用原子操作函数Cas（）先把status改为timerModifying，再进一步改为timerDeleted，同时要原子性地将相关计数减1。调用acquirem（）函数是为了避免操作过程中被抢占，可能会造成死锁问题。对于timerDeleted、timerRemoving、timerRemoved和timerNoStatus这几种状态的timer，要么不会再触发，要么根本不在堆中，所以不需要进行处理。至于timerRunning、timerMoving和timerModifying，分别表示timer正在运行、正在被移动，以及正在被修改，这种时候不能对timer进行删除操作，必须等到timer脱离当前状态以后再进一步操作，这3种状态都是比较短暂的，所以使用osyield（）函数暂时让出CPU即可。

3. 修改

modtimer（）函数的代码稍微多一些，我们将代码分成上下两部分来分析。第一部分代码如下：

```
func modtimer(t * timer, when, period int64, f func(interface{}, uintptr), arg interface{},
seq uintptr) {
    if when < 0 {
        when = maxWhen
    }

    status : = uint32(timerNoStatus)
    wasRemoved : = false
    var mp * m
loop:
    for {
    switch status = atomic.Load(&t.status); status {
        case timerWaiting, timerModifiedEarlier, timerModifiedLater:
            mp = acquirem()
            if atomic.Cas(&t.status, status, timerModifying) {
                break loop
            }
            releasem(mp)
        case timerNoStatus, timerRemoved:
            mp = acquirem()
            if atomic.Cas(&t.status, status, timerModifying) {
                wasRemoved = true
                break loop
            }
            releasem(mp)
        case timerDeleted:
            mp = acquirem()
            if atomic.Cas(&t.status, status, timerModifying) {
                atomic.Xadd(&t.pp.ptr().deletedTimers, - 1)
                break loop
            }
            releasem(mp)
        case timerRunning, timerRemoving, timerMoving:
            osyield()
        case timerModifying:
            osyield()
        default:
            badTimer()
        }
    }
```

这个for加switch语句结构跟deltimer（）函数有些类似，凡是以ing结尾的状态都表示需要等待。值得注意的是wasRemoved用来表示指定的timer已经不在堆中，后面需要与还在堆中的timer分别进行处理。

第二部分代码如下：

```
        t.period = period
        t.f = f
        t.arg = arg
        t.seq = seq

        if wasRemoved {
            t.when = when
            pp := getg().m.p.ptr()
            lock(&pp.timersLock)
            doaddtimer(pp, t)
            unlock(&pp.timersLock)
            if !atomic.Cas(&t.status, timerModifying, timerWaiting) {
                badTimer()
            }
            releasem(mp)
            wakeNetPoller(when)
        } else {
            t.nextwhen = when

            newStatus := uint32(timerModifiedLater)
            if when < t.when {
                newStatus = timerModifiedEarlier
            }

            adjust := int32(0)
            if status == timerModifiedEarlier {
                adjust--
            }
            if newStatus == timerModifiedEarlier {
                adjust++
            }
            if adjust != 0 {
                atomic.Xadd(&t.pp.ptr().adjustTimers, adjust)
            }

            if !atomic.Cas(&t.status, timerModifying, newStatus) {
                badTimer()
            }
            releasem(mp)

            if newStatus == timerModifiedEarlier {
                wakeNetPoller(when)
            }
        }
    }
```

可以看到，对于wasRemoved的timer，需要被添加到堆中，所以对应分支的逻辑与addtimer（）函数很相似，改动timer堆需要对timersLock加锁。对于原本就在堆中的timer，需要把新的触发时间when赋值给它的nextwhen字段，而不能直接改动它的when字段，因为在这里不打算改动它在堆里的位置。新的触发时间如果比原来更早，就把状态设为timerModifiedEarlier，否则状态为timerModifiedLater。经过这次改动，堆中处于timerModifiedEarlier状态的timer可能增加了一个，也可能减少了一个，还可能不增不减，如果有变化，就把adjustTimers相应地增加或减少1。最后，如

果新的触发时间比原来更早，还要调用wakeNetPoller（）函数，为的是更早地唤醒调度线程，以便处理timer。

在Go 1.16版中，modtimer（）函数会返回一个bool值，表示被修改的timer是否在运行前完成了修改，并且会更新p中的timerModifiedEarliest字段。在Go 1.14版中，p中的timer0When用来存储最小堆堆顶的timer的when字段，表示最早要触发的timer，但是对于修改过的timer，触发时间可能会被修改成一个更早的时间，却没有相应的字段来记录这个修改后的最早时间。直到Go 1.16版才新增了这个timerModifiedEarliest字段，用来存储修改后的最早时间。这样一来，在调度器处理timer时，通过这两个时间中更小的那个就能直接确定最早的触发时间，而不需要对堆进行重排序。

至此，我们已经知道timer是存储在最小堆中的，以及是如何被添加、删除和修改的。接下来看一下调度器是如何运行timer的。

4．运行

先来看两个会被调度器用到的函数，首先是用来调整timer堆的adjusttimers（）函数，代码如下：

```
func adjusttimers(pp *p) {
    if len(pp.timers) == 0 {
        return
    }
    if atomic.Load(&pp.adjustTimers) == 0 {
        if verifyTimers {
            verifyTimerHeap(pp)
        }
        return
    }
    var moved []*timer
loop:
    for i := 0; i < len(pp.timers); i++{
        t := pp.timers[i]
        if t.pp.ptr() != pp {
            throw("adjusttimers: bad p")
        }
        switch s := atomic.Load(&t.status); s {
        case timerDeleted:
            if atomic.Cas(&t.status, s, timerRemoving) {
                dodeltimer(pp, i)
                if !atomic.Cas(&t.status, timerRemoving, timerRemoved) {
                    badTimer()
                }
                atomic.Xadd(&pp.deletedTimers, -1)
                i--
            }
        case timerModifiedEarlier, timerModifiedLater:
            if atomic.Cas(&t.status, s, timerMoving) {
                t.when = t.nextwhen
                dodeltimer(pp, i)
                moved = append(moved, t)
                if s == timerModifiedEarlier {
                    if n := atomic.Xadd(&pp.adjustTimers, -1); int32(n) <= 0 {
                        break loop
                    }
                }
                i--
            }
        case timerNoStatus, timerRunning, timerRemoving, timerRemoved, timerMoving:
            badTimer()
```

```
            case timerWaiting:
                //nothing to do
            case timerModifying:
                osyield()
                i--
            default:
                badTimer()
            }
        }
        if len(moved) > 0 {
            addAdjustedTimers(pp, moved)
        }
        if verifyTimers {
            verifyTimerHeap(pp)
        }
    }
```

如果p.adjustTimers等于0，也就说明没有触发时间比p.timer0When更早的timer，该函数就会直接返回。因为p.adjustTimers记录的是堆中状态为timerModifiedEarlier的timer的数量，也就是修改后触发时间被提前的timer的数量。

在接下来的for循环中，会顺便清理掉已删除的timer，因为最小堆的结构特点，删除下标i位置的元素不会影响前面元素的顺序，所以每次删除后只需将i减1，再继续遍历就不会漏掉内容了。同理，timerModifiedEarlier和timerModifiedLater两种状态的timer也是先从堆中移除，然后追加到moved切片中，遍历完成后再由addAdjustedTimers（）函数统一添加回去，这样就可避免中途对整个堆重新排序，所以只需遍历一次就可以了。addAdjustedTimers（）函数的逻辑很简单，代码如下：

```
func addAdjustedTimers(pp *p, moved []*timer) {
    for _, t := range moved {
        doaddtimer(pp, t)
        if !atomic.Cas(&t.status, timerMoving, timerWaiting) {
            badTimer()
        }
    }
}
```

接下来讲解runtimer（）函数，调度器就是通过它来运行timer的，函数的代码如下：

```
//go:systemstack
func runtimer(pp *p, now int64) int64 {
    for {
        t := pp.timers[0]
        if t.pp.ptr() != pp {
            throw("runtimer: bad p")
        }
        switch s := atomic.Load(&t.status); s {
        case timerWaiting:
            if t.when > now {
                return t.when
            }
            if !atomic.Cas(&t.status, s, timerRunning) {
                continue
            }
            runOneTimer(pp, t, now)
            return 0
        case timerDeleted:
            if !atomic.Cas(&t.status, s, timerRemoving) {
                continue
            }
            dodeltimer0(pp)
            if !atomic.Cas(&t.status, timerRemoving, timerRemoved) {
                badTimer()
            }
            atomic.Xadd(&pp.deletedTimers, -1)
            if len(pp.timers) == 0 {
                return -1
            }
        case timerModifiedEarlier, timerModifiedLater:
            if !atomic.Cas(&t.status, s, timerMoving) {
                continue
            }
            t.when = t.nextwhen
            dodeltimer0(pp)
            doaddtimer(pp, t)
            if s == timerModifiedEarlier {
                atomic.Xadd(&pp.adjustTimers, -1)
            }
            if !atomic.Cas(&t.status, timerMoving, timerWaiting) {
                badTimer()
            }
        case timerModifying:
            osyield()
        case timerNoStatus, timerRemoved:
            badTimer()
        case timerRunning, timerRemoving, timerMoving:
            badTimer()
        default:
            badTimer()
        }
    }
}
```

该函数必须在系统栈上运行，for循环中始终取堆顶的那个timer。如果t处于timerWaiting状态，则进一步比较t.when和当前时间，如果时间还没到就返回t.when，否则就通过runOneTimer（）函数来运行t，并返回0。如果t处于timerDeleted状态，就会通过dodeltimer0（）函数把它从堆中移除，如果堆的大小变成0就返回-1，否则继续循环。如果t处于timerModifiedEarlier或timerModifiedLater状态，则先把它从堆中移除，然后重新添加进去。整体来看，只要函数的返回值不为0，就表示暂时没有timer可以运行。

再来看一下runOneTimer（）函数的逻辑，简单起见省略了部分代码，只保留了主要逻辑，代码如下：

```
//go:systemstack
func runOneTimer(pp *p, t *timer, now int64) {
    f := t.f
    arg := t.arg
    seq := t.seq

    if t.period > 0 {
        delta := t.when - now
        t.when += t.period * (1 + -delta/t.period)
        siftdownTimer(pp.timers, 0)
        if !atomic.Cas(&t.status, timerRunning, timerWaiting) {
            badTimer()
        }
        updateTimer0When(pp)
    } else {
        dodeltimer0(pp)
        if !atomic.Cas(&t.status, timerRunning, timerNoStatus) {
            badTimer()
        }
    }

    unlock(&pp.timersLock)
    f(arg, seq)
    lock(&pp.timersLock)
}
```

如果t.period字段大于0，也就说明t是个周期性的timer，此时需要把t.when设置为下次触发的时间，并调整t在堆中的位置，还要按需更新p的timer0When字段。如果是一次性的timer，就将其从堆中移除。最后，在解锁的情况下调用回调函数f（），完成后重新加锁，这样能够避免因f（）函数中调用timer相关函数造成死锁的情况。

至此，timer模块的主要函数就梳理得差不多了，接下来看一看调度器是如何处理timer的。还记得schedule（）函数和findrunnable（）函数都会调用的那个checkTimers（）函数吗？它就是联接调度循环与timer模块的纽带，函数的代码如下：

```
func checkTimers(pp *p, now int64) (rnow, pollUntil int64, ran bool) {
    if atomic.Load(&pp.adjustTimers) == 0 {
        next := int64(atomic.Load64(&pp.timer0When))
        if next == 0 {
            return now, 0, false
        }
        if now == 0 {
            now = nanotime()
        }
        if now < next {
            if pp != getg().m.p.ptr() || int(atomic.Load(&pp.deletedTimers)) <= int
(atomic.Load(&pp.numTimers)/4) {
                return now, next, false
            }
        }
    }
    lock(&pp.timersLock)
    adjusttimers(pp)
    rnow = now
    if len(pp.timers) > 0 {
        if rnow == 0 {
            rnow = nanotime()
        }
        for len(pp.timers) > 0 {
            if tw := runtimer(pp, rnow); tw != 0 {
                if tw > 0 {
                    pollUntil = tw
                }
                break
            }
            ran = true
        }
    }
    if pp == getg().m.p.ptr() && int(atomic.Load(&pp.deletedTimers)) > len(pp.timers)/4 {
        clearDeletedTimers(pp)
    }
    unlock(&pp.timersLock)
    return rnow, pollUntil, ran
}
```

函数会先处理p.adjustTimers为0的情况，这意味着堆中不存在触发时间被提前的timer，所以p.timer0When就是最早的触发时间了。p.timer0When＝0，表示堆是空的，所以不需要进一步处理了。如果p.timer0When大于当前时间，就表示还没有到达任何timer的触发时间，这时候如果堆中处于timerDeleted状态的timer数量没有达到总数的1/4，就直接返回。

接下来先对p.timersLock加锁，再通过adjusttimers调整timer堆，这样就能把那些被修改过的timer放到正确的位置。后续的for循环会一直调用runtimer（）函数，直到timer堆为空或者runtimer（）函数的返回值不等于0。如果runtimer（）函数的返回值大于0，此返回值就是下个timer的触发时间，作为pollUntil返回，让阻塞式的netpoll能够在适当的时间超时返回。最后的clearDeletedTimers（）函数保证timer堆能够得到清理，因为adjusttimers（）函数在p.adjustTimers为0时不会进行任何操作，所以这个清理操作是必要的，避免太多已删除的timer影响堆性能。

6.11 netpoller

在Go语言的runtime中，netpoller是负责把IO多路复用和协程调度结合起来的模块。如果goroutine执行网络IO时需要等待，则netpoller就会自动将其挂起，等到数据就绪以后再将其唤醒，用户代码对这一切都是无感知的，所以对于开发者来讲非常方便。本节还是从源码入手，分析并探索netpoller实现的原理。Go语言的源码包含对多平台架构的支持，我们主要研究Linux系统上的netpoller实现，并且假设大家对epoll已经有了最基本的了解。

6.11.1 跨平台的netpoller

为了支持多个平台，Go的开发者对netpoller的源码进行了抽象，各个平台共用的逻辑被放置在netpoll.go文件中，分别适配各个平台的代码都有自己单独的文件，例如netpoll_epoll.go是针对Linux系统的，netpoll_kqueue.go是针对macOS和BSD系统的。这些适配不同平台的代码被抽象成一组标准函数，这样一来netpoller的绝大部分代码就不用考虑具体的平台了。在Go 1.14版本中，这组函数一共有7个，函数的原型如下：

```
func netpollinit()
func netpollIsPollDescriptor(fd uintptr) bool
func netpollopen(fd uintptr, pd * pollDesc) int32
func netpollclose(fd uintptr) int32
func netpollarm(pd * pollDesc, mode int)
func netpollBreak()
func netpoll(delay int64) gList
```

接下来就结合netpoll_epoll.go中与Linux系统对应的一组实现，逐个梳理各个函数的用途，源代码摘选自Go 1.14版本的runtime。

1. netpollinit（）函数

netpollinit（）函数用来初始化poller，只会被调用一次。在Linux系统上主要用来创建epoll实例，还会创建一个非阻塞式pipe，用来唤醒阻塞中的netpoller，代码如下：

```
func netpollinit() {
    epfd = epollcreate1(_EPOLL_CLOEXEC)
    if epfd < 0 {
        epfd = epollcreate(1024)
        if epfd < 0 {
        println("runtime: epollcreate failed with", -epfd)
            throw("runtime: netpollinit failed")
        }
        closeonexec(epfd)
    }
    r, w, errno := nonblockingPipe()
    if errno != 0 {
        println("runtime: pipe failed with", -errno)
        throw("runtime: pipe failed")
    }
    ev := epollevent{
        events: _EPOLLIN,
    }
    *(**uintptr)(unsafe.Pointer(&ev.data)) = &netpollBreakRd
    errno = epollctl(epfd, _EPOLL_CTL_ADD, r, &ev)
    if errno != 0 {
        println("runtime: epollctl failed with", -errno)
        throw("runtime: epollctl failed")
    }
    netpollBreakRd = uintptr(r)
    netpollBreakWr = uintptr(w)
}
```

其中，epfd、netpollBreakRd和netpollBreakWr都是包级别的变量。efpd是epoll实例的文件描述符，netpollBreakRd和netpollBreakWr是非阻塞管道两端的文件描述符，分别被用作读取端和写入端。读取端netpollBreakRd被添加到epoll中监听EPOLLIN事件，后续从写入端netpollBreakWr写入数据就能唤醒阻塞中的poller。

2．netpollIsPollDescriptor（）函数

netpollIsPollDescriptor（）函数用来判断文件描述符fd是否被poller使用，在Linux对应的实现中，只有epfd、netpollBreakRd和netpollBreakWr属于被poller使用的描述符，函数的代码如下：

```
func netpollIsPollDescriptor(fd uintptr) bool {
    return fd == uintptr(epfd) || fd == netpollBreakRd || fd == netpollBreakWr
}
```

3．netpollopen（）函数

netpollopen（）函数用来把要监听的文件描述符fd和与之关联的pollDesc结构添加到poller实例中，在Linux上就是添加到epoll中，代码如下：

```
func netpollopen(fd uintptr, pd * pollDesc) int32 {
    var ev epollevent
    ev.events = _EPOLLIN | _EPOLLOUT | _EPOLLRDHUP | _EPOLLET
    * ( ** pollDesc)(unsafe.Pointer(&ev.data)) = pd
    return - epollctl(epfd, _EPOLL_CTL_ADD, int32(fd), &ev)
}
```

文件描述符是以EPOLLET（监听边缘触发模式）被添加到epoll中的，同时监听读、写事件。pollDesc类型的数据结构pd作为与fd关联的自定义数据会被一同添加到epoll中。

4．netpollclose（）函数

netpollclose（）函数用来把文件描述符fd从poller实例中移除，也就是从epoll中删除，代码如下：

```
func netpollclose(fd uintptr) int32 {
    var ev epollevent
    return - epollctl(epfd, _EPOLL_CTL_DEL, int32(fd), &ev)
}
```

5．netpollarm（）函数

netpollarm（）函数只有在应用水平触发的系统上才会被用到，Linux不会用到该函数，只是为了通过编译而用来凑数的，代码如下：

```
func netpollarm(pd * pollDesc, mode int) {
    throw("runtime: unused")
}
```

6．netpollBreak（）函数

netpollBreak（）函数用来唤醒阻塞中的netpoll，它实际上就是向netpollBreakWr描述符中写入数据，这样一来epoll就会监听到netpollBreakRd的EPOLLIN事件，代码如下：

```
func netpollBreak() {
    for {
        var b byte
        n : = write(netpollBreakWr, unsafe.Pointer(&b), 1)
        if n == 1 {
            break
        }
        if n == - _EINTR {
            continue
        }
        if n == - _EAGAIN {
            return
        }
        println("runtime: netpollBreak write failed with", - n)
        throw("runtime: netpollBreak write failed")
    }
}
```

因为write调用可能会被打断，所以在遇到EINTR错误的时候，netpollBreak（）函数会通过for循环持续尝试向netpollBreakWr中写入一字节数据。

7．netpoll（）函数

还剩最后一个函数，也是最为关键的，那就是netpoll（）函数。在6.8节分析调度循环的时候，我们知道该函数会返回一个gList，里面是因为IO数据就绪而能够恢复运行的一组g。我们把函数的源码分成3部分分别进行梳理。

第一部分代码如下：

```
if epfd == -1 {
    return gList{}
}
var waitms int32
if delay < 0 {
    waitms = -1
} else if delay == 0 {
    waitms = 0
} else if delay < 1e6 {
    waitms = 1
} else if delay < 1e15 {
    waitms = int32(delay / 1e6)
} else {
    waitms = 1e9
}
```

epfd的初始值是-1，而有效的文件描述符不会小于0。epfd仍旧等于-1，表明epoll尚未初始化，此时netpoll（）函数就会返回一个空的gList。接下来的if语句块把纳秒级的delay转换成了毫秒级的waitms。

第二部分代码如下：

```
    var events [128]epollevent
retry:
    n := epollwait(epfd, &events[0], int32(len(events)), waitms)
    if n < 0 {
        if n != -_EINTR {
            println("runtime: epollwait on fd", epfd, "failed with", -n)
            throw("runtime: netpoll failed")
        }
        if waitms > 0 {
            return gList{}
        }
        goto retry
    }
```

通过epollwait（）函数等待IO事件，缓冲区大小为128个epollevent，超时时间是waitms。如果epollwait（）函数被中断打断，就通过goto来重试。waitms＞0时不会重试，因为需要返回调用者中去重新计算超时时间。

第三部分代码如下：

```
var toRun gList
for i := int32(0); i < n; i++{
    ev := &events[i]
    if ev.events == 0 {
        continue
    }

    if *(**uintptr)(unsafe.Pointer(&ev.data)) == &netpollBreakRd {
        if ev.events != _EPOLLIN {
            println("runtime: netpoll: break fd ready for", ev.events)
            throw("runtime: netpoll: break fd ready for something unexpected")
        }
        if delay != 0 {
            var tmp [16]Byte
            read(int32(netpollBreakRd), noescape(unsafe.Pointer(&tmp[0])),
int32(len(tmp)))
        }
        continue
    }

    var mode int32
    if ev.events&(_EPOLLIN|_EPOLLRDHUP|_EPOLLHUP|_EPOLLERR) != 0 {
        mode += 'r'
    }
    if ev.events&(_EPOLLOUT|_EPOLLHUP|_EPOLLERR) != 0 {
        mode += 'w'
    }
    if mode != 0 {
        pd := *(**pollDesc)(unsafe.Pointer(&ev.data))
        pd.everr = false
        if ev.events == _EPOLLERR {
            pd.everr = true
        }
        netpollready(&toRun, pd, mode)
    }
}
return toRun
```

通过for循环遍历所有IO事件。对于文件描述符netpollBreakRd而言，只有EPOLLIN事件是正常的，其他都会被视为异常。只有在delay不为0，也就是阻塞式netpoll时，才读取netpollBreakRd中的数据。根据epoll返回的IO事件标志位为mode赋值：r表示可读，w表示可写，r+w表示既可读又可写。mode不为0，表示有IO事件，需要从ev.data字段得到与IO事件关联的pollDesc，检测IO事件中的错误标志位，并相应地为pd.everr赋值，最后调用netpollready（）函数。netpollready（）函数的代码如下：

```
func netpollready(toRun *gList, pd *pollDesc, mode int32) {
    var rg, wg *g
    if mode == 'r' || mode == 'r'+'w' {
        rg = netpollunblock(pd, 'r', true)
    }
    if mode == 'w' || mode == 'r'+'w' {
        wg = netpollunblock(pd, 'w', true)
    }
    if rg != nil {
        toRun.push(rg)
    }
    if wg != nil {
        toRun.push(wg)
    }
}
```

该函数的作用是，根据mode的值从pollDesc中取出IO需求被满足的goroutine，然后添加到toRun列表中。例如mode的值是可读或可读可写，而pollDesc中也有等待读事件的goroutine，那么这个goroutine就该被唤醒继续运行了，所以就会把这个goroutine添加到toRun中。从pollDesc中获得对应G指针的操作是由netpollunblock（）函数完成的。

在进一步探索之前，需要先弄清楚pollDesc结构中各个字段的含义，每个文件描述符被添加到netpoller中之后，都由一个pollDesc来表示，该结构的定义代码如下：

```
//go:notinheap
type pollDesc struct {
    link     *pollDesc
    lock     mutex
    fd       uintptr
    closing bool
    everr    bool
    user     uint32
    rseq     uintptr
    rg       uintptr
    rt       timer
    rd       int64
    wseq     uintptr
    wg       uintptr
    wt       timer
    wd       int64
}
```

通过notinheap注释可以知道，该数据结构不允许被分配在堆上，runtime会使用持久化分配器来为该结构分配内存，并且实现了专用的pollCache进行缓存。pollDesc各字段的用途如表6-11所示。

表6-11 pollDesc各字段的用途

字　　段	用　　途
link	用于实现 pollCache 缓存，将空闲的 pollDesc 串成一个链表
lock	用来保护 pollDesc 结构中 seq 和 tiwer 相关字段
fd	要监听的文件描述符
closing	表示文件描述符 fd 正在被从 poller 中移除
everr	表示 poller 返回的 IO 事件中包含错误标志位
user	在 Linux 下没有被用到，aix、solaris 等会利用它存储一些扩展信息
rseq	一直自增的序列号，因为 pollDesc 结构会被复用，通过增加 rseq 的值，能够避免复用后的 pollDesc 被旧的读超时 timer 干扰
rg	有 4 种可能的取值，常量 pdReady、pdWait，一个 G 的指针，以及 nil。pdReady 表示 fd 的数据已经就绪，可供读取，某个 goroutine 消费掉这些数据后会把 rg 置为 nil。pdWait 表示某个 goroutine 即将挂起并等待 fd 的可读事件，goroutine 挂起后 rg 会被改成该 g 的指针，或者一个并发的可读事件会把 rg 置为 pdReady，抑或一个并发的读取超时或 close 操作会把 rg 置为 nil
rt	用于实现读超时的 timer，它会在超时时间到达时唤醒等待读的 goroutine
rd	设置的读超时时间
wseq	与 rseq、rg、rt 及 rd 类似，只不过针对的是写操作
wg	
wt	
wd	

在了解了pollDesc的结构后，继续看netpollunblock（）函数的代码，代码如下：

```
func netpollunblock(pd *pollDesc, mode int32, ioready bool) *g {
    gpp := &pd.rg
    if mode == 'w' {
        gpp = &pd.wg
    }

    for {
        old := *gpp
        if old == pdReady {
            return nil
        }
        if old == 0 && !ioready {
            return nil
        }
        var new uintptr
        if ioready {
            new = pdReady
        }
        if atomic.Casuintptr(gpp, old, new) {
            if old == pdWait {
                old = 0
            }
            return (*g)(unsafe.Pointer(old))
        }
    }
}
```

首先要讲解的是函数的参数，mode可以是字符r或w，分别表示要取得pd中等待读或等待写的g，ioready表示与mode相对应的IO事件是否已触发，也就是fd是否可读或可写。

变量gpp，也就是g指针的指针，默认获取的是pd.rg的地址。如果mode是w，则是pd.wg的地址。在接下来的for循环中，先处理的是old值为pdReady的情况，也就是说IO已经就绪，却没有等待IO的协程，那么无论本次ioready的值如何，都不需要更新*gpp的值，于是直接返回nil。如果old值为0，并且ioready为false，表示既没有协程在等待，也没有已就绪的IO事件，所以不需要做任何处理，直接返回nil。接下来声明了变量new，其默认值为0，对应指针类型的nil。如果ioready为真，则new会被赋值为pdReady。接下来的CAS函数会把新的状态new赋给*gpp，并修正old的值。因为old最终会被强转换为*g类型，所以必须是一个有效的指针或nil。

综上所述，netpollunblock（）函数不会阻塞，它会根据mode和ioready的值从pd中取出等待IO的g，如果没有，则返回nil。该函数还可能会更新rg或wg的值，新的值为0或pdReady。

回过头来看，从netpoll（）函数到netpollready（）函数，再到这里的netpollunblock（）函数，就是一步步把epollwait（）函数返回的IO事件存储到了对应的pollDesc中。如果有正在等待该事件的协程，就会被添加到gList中返回，继而被添加到runq中。

至此，我们已经了解了等待IO的协程是如何被netpoller唤醒的，但是协程又是如何因IO等待而挂起的呢？这可以从标准库中与网络IO相关的函数和方法入手，接下来就以TCP连接的Read方法为入口，逐层深入分析源码。

6.11.2 TCP连接的Read（）方法

net.TCPConn通过嵌入net.conn类型而继承了后者的Read（）方法，而net.（*conn）.Read（）方法会调用net.（*netFD）.Read（）方法，后者又会调用internal/poll.（*FD）.Read（）方法，后者又会调用internal/poll.（*pollDesc）.waitRead（）方法，waitRead（）方法会调用internal/poll.

（*pollDesc）.wait（）方法。wait（）方法通过调用internal/poll.runtime_pollWait（）函数实现功能，而后者则是通过linkname机制链接到runtime.poll_runtime_pollWait（）函数，该函数的代码如下：

```
func poll_runtime_pollWait(pd * pollDesc, mode int) int {
    err : = netpollcheckerr(pd, int32(mode))
    if err != 0 {
        return err
    }
    if GOOS == "solaris" || GOOS == "illumos" || GOOS == "aix" {
        netpollarm(pd, mode)
    }
    for !netpollblock(pd, int32(mode), false) {
        err = netpollcheckerr(pd, int32(mode))
        if err != 0 {
            return err
        }
    }
    return 0
}
```

该函数最主要的逻辑就是通过netpollblock（）函数实现的，与它的名字一样，netpollblock（）函数可能会造成调用它的goroutine阻塞而挂起，函数的代码如下：

```
func netpollblock(pd *pollDesc, mode int32, waitio bool) bool {
    gpp := &pd.rg
    if mode == 'w' {
        gpp = &pd.wg
    }

    for {
        old := *gpp
        if old == pdReady {
            *gpp = 0
            return true
        }
        if old != 0 {
            throw("runtime: double wait")
        }
        if atomic.Casuintptr(gpp, 0, pdWait) {
            break
        }
    }

    if waitio || netpollcheckerr(pd, mode) == 0 {
        gopark(netpollblockcommit, unsafe.Pointer(gpp), waitReasonIOWait,
traceEvGoBlockNet, 5)
    }
    old := atomic.Xchguintptr(gpp, 0)
    if old > pdWait {
        throw("runtime: corrupted polldesc")
    }
    return old == pdReady
}
```

该函数与netpollunblock（）函数有些相似，不同的是waitio表示是否要挂起以等待IO就绪，返回值为true，表示IO就绪，false则可能是超时或fd被移除。如果old值为pdReady，就表示当前IO已经处于就绪状态，所以直接返回true。如果old为0，就先通过CAS把它置为pdWait，表示当前协程即将挂起等待IO就绪，然后当前协程会调用gopark（）函数来挂起自己，netpollblockcommit（）函数会把当前g的地址赋值给*gpp。等到挂起的协程被netpoller唤醒后，就会从gopark返回，从gpp中取得新的IO状态，继续执行后续逻辑。

至此，我们就梳理完了goroutine是如何因为网络IO的原因而被挂起，以及又是如何在IO就绪之后被netpoller唤醒的。本节关于netpoller的探索就到这里，更多有趣的细节各位读者可自行阅读、分析源码。

6.12 监控线程

通过6.6节的介绍，我们已经知道监控线程是由main goroutine创建的。监控线程与GMP中的工作线程不同，并不需要依赖P，也不由GMP模型调度。它会重复执行一系列任务，只不过会视情况调整自己的休眠时间，接下来我们就简单介绍一下监控线程的主要任务。

6.12.1 按需执行timer和netpoll

在6.10节介绍timer时已经了解到每个P都持有一个最小堆，存储在p.timers中，用于管理自己的timer，而堆顶的timer就是接下来要触发的那一个，而timer中持有一个回调函数timer.f（），在指定时间到达后就会调用这个回调函数，但是谁负责在时间到达时调用回调函数呢？

在6.8节介绍调度程序的主要逻辑时，我们知道每次调度时都会调用checkTimers（）函数，检查并执行已经到时间的那些timer。不过这还不够稳妥，万一所有M都在忙，不能及时触发调度，可能会导致timer执行时间发生较大的偏差，所以还会通过监控线程来增加一层保障。

当监控线程检测到接下来有timer要执行时，不仅会按需调整休眠时间，还会在没有空闲M时创建新的工作线程，以保障timer可以顺利执行。

timer有明确的触发时间，但是IO事件的就绪就没那么确定了，所以为了降低IO延迟，需要时不时地主动轮询，以及时获得就绪的IO事件，也就是执行netpoll。

全局变量sched中会记录上次netpoll执行的时间（sched.lastpoll），如果监控线程检测到距离上次轮询已超过了10ms，就会再执行一次netpoll。实际上，不只是监控线程，第6章介绍过的调度器，以及第8章要介绍的GC在工作过程中都会按需执行netpoll。

6.12.2 抢占G和P

本着公平调度的原则，监控线程会对运行时间过长的G实行抢占操作，也就是告诉那些运行时间超过特定阈值（10ms）的G，该让出了。

如何确定哪些G运行时间过长了呢？runtime.p中有一个schedtick字段，每当调度执行一个新的G并且不继承上个G的时间片时，都会让它自增一，相关字段的代码如下：

```
type p struct {
    //……略去部分代码
    schedtick   uint32
    sysmontick  sysmontick
}
type sysmontick struct {
    schedtick   uint32
    schedwhen   int64
    syscalltick uint32
    syscallwhen int64
}
```

而p.sysmontick.schedwhen记录的是上一次调度的时间。监控线程如果检测到p.sysmontick.schedtick与p.schedtick不相等，说明这个P又发生了新的调度，就会同步这里的调度次数，并更新这个调度时间，相关代码如下：

```
pd := &_p_.sysmontick
//……略去部分代码
t := int64(_p_.schedtick)
//……略去部分代码
pd.schedtick = uint32(t)
pd.schedwhen = now
```

但是若p.sysmontick.schedtick与p.schedtick相等，就说明自p.sysmontick.schedwhen这个时间点之后，这个P并未发生新的调度，或者即使发生了新的调度，也继承了之前G的时间片，所以可以通过当前时间与schedwhen的差值，来判断当前P上的G是否运行时间过长了，代码如下：

```
pd.schedwhen + forcePreemptNS <= now
```

如果G真的运行时间过长了，要怎么通知它让出呢？这自然要使用6.9节介绍过的两种抢占方式了，通过设置stackPreempt标识，或者进行异步抢占。

为了充分利用CPU，监控线程还会抢占处在系统调用中的P。因为一个协程要执行系统调用，就要切换到g0栈，在系统调用没执行完之前，这个M和这个G不能被分开，但是用不到P，所以在陷入系统调用之前，当前M会让出P，解除与当前P的强关联，只在m.oldp中记录这个P。P的数目毕竟有限，如果有其他协程在等待执行，则放任P如此闲置就着实浪费了。还是把它关联到其他M，继续工作比较划算。

等到当前M从系统调用中恢复后，会先检测之前的P是否被占用，如果没有被占用就继续使用。否则再去申请一个，如果没申请到，就把当前G放到全局runq中去，然后当前线程就睡眠了。

6.12.3 强制执行GC

在runtime包的proc.go中有一个init（）函数，它会以forcegchelper（）函数为执行入口创建一个协程，代码如下：

```
func init() {
    go forcegchelper()
}
```

也就是说在程序初始化时就会创建一个辅助执行GC的协程，只不过它在做完必要的初始化工作后便会主动让出。等到它恢复执行时，就可以通过gcStart（）函数发起新一轮的GC了，代码如下：

```
var forcegc     forcegcstate
type forcegcstate struct {
    lock mutex
    g     *g
    idle uint32
}
func forcegchelper() {
    forcegc.g = getg()
    for {
        lock(&forcegc.lock)
        if forcegc.idle != 0 {
            throw("forcegc: phase error")
        }
        atomic.Store(&forcegc.idle, 1)
        goparkunlock(&forcegc.lock, waitReasonForceGGIdle, traceEvGoBlock, 1)
        //this goroutine is explicitly resumed by sysmon
        if debug.gctrace > 0 {
            println("GC forced")
        }
        //Time-triggered, fully concurrent.
        gcStart(gcTrigger{kind: gcTriggerTime, now: nanotime()})
    }
}
```

而监控线程会创建gcTriggerTime类型的gcTrigger，这种类型的GC触发器会检测距离上次执行GC的时间是否已经超过runtime.forcegcperiod，默认为两分钟，代码如下：

```
var forcegcperiod int64 = 2 * 60 * 1e9
```

如果超过指定时间，同时forcegc还没有被开启，就需修改forcegc的状态信息，并把forcegc.g记录的协程（程序初始化时创建的那个辅助执行GC的协程）添加到全局runq中。这样等到它得到调度执行时，就会开启新一轮的GC工作了。

监控线程的主要任务就介绍到这里，保障计时器正常执行，执行网络轮询，抢占长时间运行的或处在系统调用的P，以及强制执行GC，监控线程的这些工作任务无不是为了保障程序健康高效地执行。

6.13 本章小结

本章内容较多，稍微有些复杂。开篇先简单分析了进程、线程和协程的不同，实际上就是越来越轻量。接下来又对比了传统的阻塞式IO、非阻塞式IO，还有近年来流行的IO多路复用，更重要的是协程和IO多路复用这两项技术的巧妙结合。有了这些铺垫之后，就可以开始深入Go语言的协程调度了。首先就是GMP模型，从基本概念到主要的数据结构，然后结合源码分析，逐步梳理了调度器的初始化、协程的创建与退出，还有最核心的调度循环。之后用一个实例，通过调试加源码分析的方式，深入对比了Go 1.13版本和Go 1.14版本中抢占式调度的不同实现，笔者认为Go 1.14版本以后才是真正的抢占。最后几节主要基于源码分析，梳理了timer、netpoller的实现细节，以及监控线程的主要工作。虽然整体有些繁杂，但是对于想要深入了解goroutine的读者，还是有一定的参考价值的。

第7章
同步

在一开始接触多线程编程的时候，我们就被告知同步有多么重要，那个经典的银行取款的例子也已经听过了很多遍。之所以称为同步，就是因为存在并发，不过大多数对于并发同步的讲解都太上层了。本章通过对编译、执行及一些硬件特性的探索，进一步加深大家对同步的理解，希望能够帮助大家写出更健壮的程序。

7.1 Happens Before

在多线程的环境中，多个线程或协程同时操作内存中的共享变量，如果不加限制，就会出现出乎意料的结果。想保证结果正确，就需要在时序上让来自不同线程的访问串行化，彼此之间不出现重叠。线程对变量的操作一般只有Load和Store两种，就是我们俗称的读和写。Happens Before也可以认为是一种串行化描述或要求，目的是保证某个线程对变量的写操作，能够被其他的线程正确地读到。

按照字面含义，你可能会认为，如果事件e2在时序上于事件e1结束后发生，就可以说事件e1 happens before e2了。按照一般常识应该是这样的，在我们介绍内存乱序之前暂时可以这样理解，事实上这对于多核环境下的内存读写操作来讲是不够的。

如果e1 happens before e2，则可以说成e2 happens after e1。若要保证对变量v的某个读操作r，能够读取到某个写操作w写入v的值，必须同时满足以下条件：

（1）w happens before r。

（2）没有其他针对v的写操作happens after w且before r。

如果e1既不满足happens before e2，又不满足happens after e2，就认为e1与e2之间存在并发，如图7-1所示。单个线程或协程内部访问某个变量是不存在并发的，默认能满足happens before条件，因此某个读操作总是能读到之前最近一次写操作写入的值，但是在多个线程或协程之间就不一样了，因为存在并发的原因，必须通过一些同步机制实现串行化，以确立happens before条件。

7.1.1 并发

我们知道现代操作系统是基于时间片算法来调度线程的，goroutine也实现了基于时间片的抢占式调度。当线程的时间片用完时，可能会在任意两条机器指令间被打断。假设线程t1即将执行一个针对变量v的写操作w，而线程t2即将执行一个针对变量v的读操作r，我们想要让r读取w写入的值，也就是要让w happens before r。暂定我们的执行环境只有一个CPU内核，所以任一时刻t1和t2只能有一个在执行。即使这样也依然有问题，t1可能在执行w操作之前就被打断了，然后t2执行了r操作。如果不使用一些同步机制，我们无法保证t2的r操作执行时，t1的w操作已经执行完了。最常用的同步工具就是锁，但是针对某些特定场景，我们不用锁也可以让程序得到正确的结果。

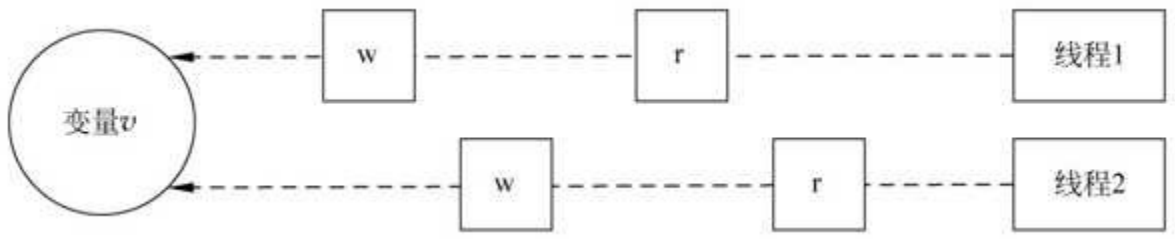

图7-1 多线程并发事件示意图

例如经典的生产者、消费者场景，有两个线程分别是生产者和消费者，两者之间通过共享变量来传递数据。为了让程序能够像预期那样运行，消费者线程必须在生产者线程完成共享变量的写操作之后才去读，生产者线程也必须在消费者线程完成读取之后才能再次将新的值写入共享变量，两者需要一直交替地执行。

可以通过引入另外一个变量实现这个目的，接下来就尝试用Go语言实现，例如原本的共享变量是int类型的变量data，我们再引入一个bool型变量ok，用来表示data的所有权，代码如下：

```
var data int
var ok int
```

当变量ok为false时，data的所有权归生产者所有，生产者首先为data赋值，完成之后再把ok设置为true，从而把data的所有权传递给了消费者，代码如下：

```
//第 7 章/code_7_1.go
go func() {
    for {
        if !ok {
            data = someValue
            ok = true
        }
    }
}()
```

当变量ok为true时，data的所有权归消费者所有，消费者读取完data的值后，再把ok的值设置为false，也就是把data的所有权又回传给了生产者，代码如下：

```
//第 7 章/code_7_2.go
go func() {
    for {
        if ok {
            sum += data
            ok = false
        }
    }
}()
```

如果编译器生成的指令与源码中语句的顺序严格一致，上述生产者协程和消费者协程在单核CPU上并发执行是可以保证结果正确的。一旦编译器对生成指令的顺序进行优化调整，或者程序在多核CPU上执行，就不能保证结果正确了，具体原因接下来会逐步分析。

在单核CPU上分时交替运行的多个线程，可以认为是最经典的并发场景。宏观上看起来同时在运行的多个线程，微观上是以极短时间片交替运行在同一个CPU上的。在多核CPU出现以前，并发指的就是这种情况，但是在多核CPU出现以后，并发就不像以前那么简单了，不仅是微观上的分时运行，还包含了并行的情况。

7.1.2 并行

抽象地解释并发，指的是多个事件在宏观上是同时发生的，但是并不一定要在同一时刻发生，而并行就不一样了，从微观角度来看，并行的两个事件至少有某一时刻是同时发生的，所以在单核CPU上的多线程只存在并发，不存在并行。只有在多核CPU上，线程才有可能并行执行。

针对7.1.1节中的生产消费示例，我们说过，如果编译器不调整指令顺序，并且在单核CPU上执行程序，就可以保证结果的正确性。如果在多核CPU上运行，就不能保证结果正确了，这里不能简单地认为是受并行的影响，根本原因是运行在不同CPU核心上的线程间可能会存在内存乱序，从现象来看就像是CPU在运行阶段调整了某些指令的顺序一样。我们将在7.2节中对内存乱序展开更深入和全面的探索。

7.2 内存乱序

一般来讲，我们认为代码会按照编写的顺序来执行，也就是逐语句、逐行地按顺序执行，然而事实并非如此，编译器有可能对指令顺序进行调整，处理器普遍具有乱序执行的特性，目的都是为了更优的性能。操作内存的指令可以分成Load和Store两类，也就是按读和写划分。编译器和CPU都会考虑指令间的依赖关系，在不会改变当前线程行为的前提下进行顺序调整，因此在单个线程内依然是逻辑有序的，语句间原本满足的happens before条件不会被破坏，但这种有序性只是在单个线程内，并不会保证线程间的有序性。

程序中的共享变量都位于内存中，指令顺序的变化会让多个线程同时读写内存中的共享变量时产生意想不到的结果。这种因指令乱序造成的内存操作顺序与预期不一致的问题，就是所谓的内存乱序。

7.2.1 编译期乱序

所谓的编译期乱序，指的是编译器对最终生成的机器指令进行了顺序调整，一般是出于性能优化的目的。造成的影响就是，机器指令的顺序与源代码中语句的顺序并不严格一致。这种乱序在C++中比较常见，尤其是在编译器的优化级别比较高的时候。

还是以生产者消费者为例，这次改成用C++实现。源码中有个整型共享变量data，它会被一对生产者、消费者线程操作，为了协调这两个线程，我们又加了一个bool型变量ok，代码如下：

```
int data;
bool ok = false;
```

生产者和消费者分别运行在两个线程中，都循环执行处理逻辑。生产者每次循环开始时会先检查ok的值，一直等到ok为false，也就表示data中没有数据，此时生产者就先为data赋值，再把ok设为true，表示data中的数据已经就绪了，代码如下：

```
void producer() {
    while(true) {
        if(!ok) {
            data = someValue; //produce
            ok = true;
        }
    }
}
```

消费者每次循环开始时也会先检查ok的值，一直等到ok为true后才去消费data中的数据，完成后再把ok的值设为false，这样生产者就可以生产新的数据了，代码如下：

```
void consumer() {
    while(true) {
        if(ok) {
            sum += data; //consume
            ok = false;
        }
    }
}
```

按照预期，这个程序应该能够正常运行，但是有时候结果可能会出乎意料，原因就是刚刚讲过的编译乱序问题。按照之前的设计，用ok来表示data当前的状态，生产者和消费者相互传递data的所有

权，这非常依赖data和ok的内存访问顺序。生产者和消费者都要先检查ok的值，在条件允许，也就是获取到所有权的情况下，先操作data，后为ok赋值。这个顺序是不能颠倒的，一旦改变了ok的值，就把data的所有权交给了对方。

编译器并不知道这些，它只要保证单个线程的行为不被改变就可以了。经过编译优化之后，生产者可能变成先把ok设置为true，再为data赋值，消费者也可能先把ok设置为false，再读取data的值，所以运行结果就会出现错误。

那么如何解决这种编译阶段的乱序问题呢？最常用的方法就是使用compiler barrier，俗称编译屏障。编译屏障会阻止编译器跨屏障移动指令，但是仍然可以在屏障的两侧分别移动。在GCC中，常用的编译屏障就是在两条语句之间嵌入一个空的汇编语句块，代码如下：

```
data = someValue;
asm volatile("" ::: "memory"); //compiler barrier
ok = true;
```

上面的示例加上编译屏障后，应该能够在x86平台上正常运行了，但是依然无法保证能够在其他平台上如预期地运行，原因就是CPU在执行期间也可能会对指令的顺序进行调整，也就是我们接下来要探索的执行期乱序。

7.2.2 执行期乱序

笔者已经不止一次地提到过，CPU可能在执行期间对指令顺序进行调整，也就是这里所谓的执行期乱序。在进行枯燥的分析之前，先用一段代码来让大家亲自见证执行期乱序，这样更有助于后续内容的理解。示例代码使用Go语言实现，平台是amd64，代码如下：

```
//第 7 章/code_7_3.go
func main() {
    s := [2]chan struct{}{
        make(chan struct{}, 1),
        make(chan struct{}, 1),
    }
    f := make(chan struct{}, 2)
    var x, y, a, b int64
    go func() {
        for i := 0; i < 1000000; i++{
            <-s[0]
            x = 1
            b = y
            f <- struct{}{}
        }
    }()
    go func() {
        for i := 0; i < 1000000; i++{
            <-s[1]
            y = 1
            a = x
            f <- struct{}{}
        }
    }()
    for i := 0; i < 1000000; i++{
        x = 0
        y = 0
        s[i%2] <- struct{}{}
        s[(i+1)%2] <- struct{}{}
        <-f
        <-f
        if a == 0 && b == 0 {
            println(i)
        }
    }
}
```

代码中一共有3个协程，4个int类型的共享变量，3个协程都会循环100万次，3个channel用于同步每次循环。循环开始时先由主协程将x、y清零，然后通过切片s中的两个channel让其他两个协程开始运行。协程一在每轮循环中先把1赋值给x，再把y赋值给b。协程二在每轮循环中先把1赋值给y，再把x赋值给a。f用来保证在每轮循环中都等到两个协程完成赋值操作后，主协程才去检测a和b的值，当两者同时为0时会打印出当前循环的次数。

从源码角度来看，无论如何a和b都不应该同时等于0。如果协程一完成赋值后协程二才开始执行，结果就是a＝1而b＝0，反过来就是a等于0而b等于1。如果两个协程的赋值语句并行执行，则结果就是a和b都等于1，然而实际运行时会发现大量打印输出，根本原因就是出现了执行期乱序。注意，执行期乱序要在并行环境下才能体现出来，单个CPU核心自己是不会体现出乱序的。Go程序可以使用GOMAXPROCS环境变量来控制P的数量，针对上述示例代码，将GOMAXPROCS设置为1即使在多核心CPU上也不会出现乱序。

协程一和协程二中的两条赋值语句形式相似，对应到x86汇编就是三条内存操作指令，按照顺序及分类分别是Store、Load、Store，如图7-2所示。

出现的乱序问题是由前两条指令造成的，称为Store-Load乱序，这也是当前x86架构CPU上能够观察到的唯一一种乱序。Store和Load分别操作的是不同的内存地址，从现象来看就像是先执行了Load而后执行了Store。

为什么会出现Store-Load乱序呢？我们知道现在的CPU普遍带有多级指令和数据缓存，指令执行系统也是流水线式的，可以让多条指令同时在流水线上执行。一般的内存属于write-back cacheable内存，简称WB内存。对于WB内存而言，Store和Load指令并不是直接操作内存中的数据的，而是先把指定的内存单元填充到高速缓存中，然后读写高速缓存中的数据。

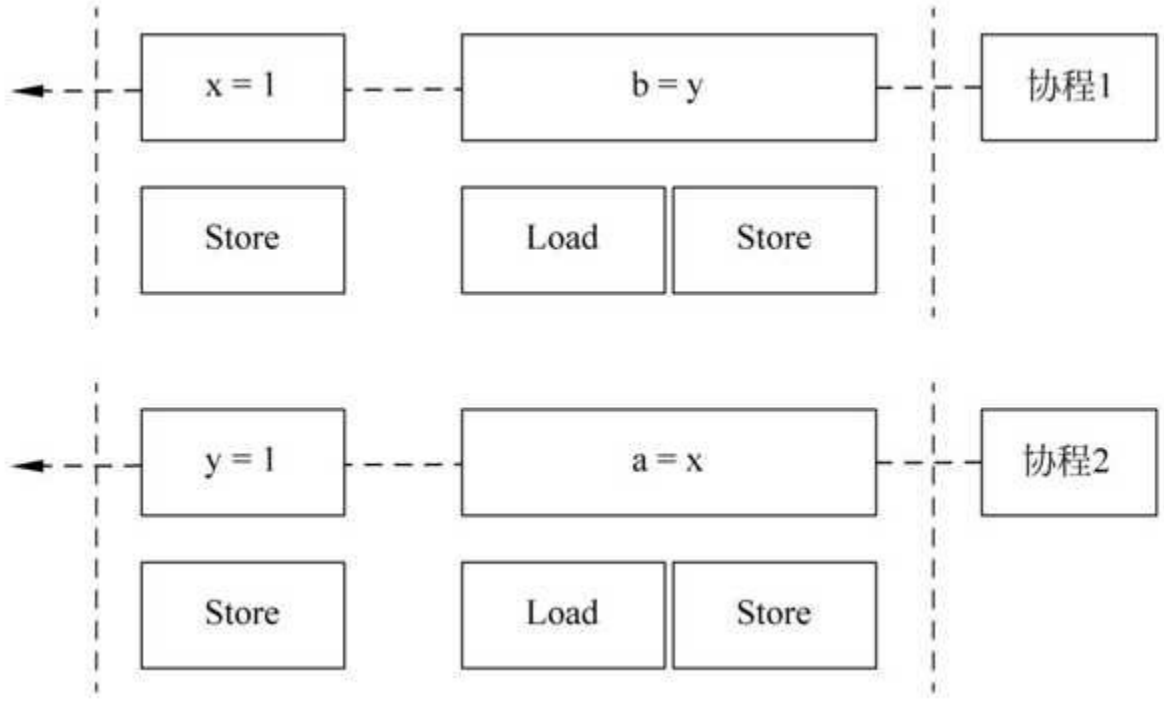

图7-2 协程一和协程二的赋值语句对应的汇编指令

Load指令的大致流程是，先尝试从高速缓存中读取，如果缓存命中，则读操作就完成了，如图7-3（a）所示。如果缓存未命中，则先填充对应的Cache Line，然后从Cache Line中读取，如图7-3（b）所示。

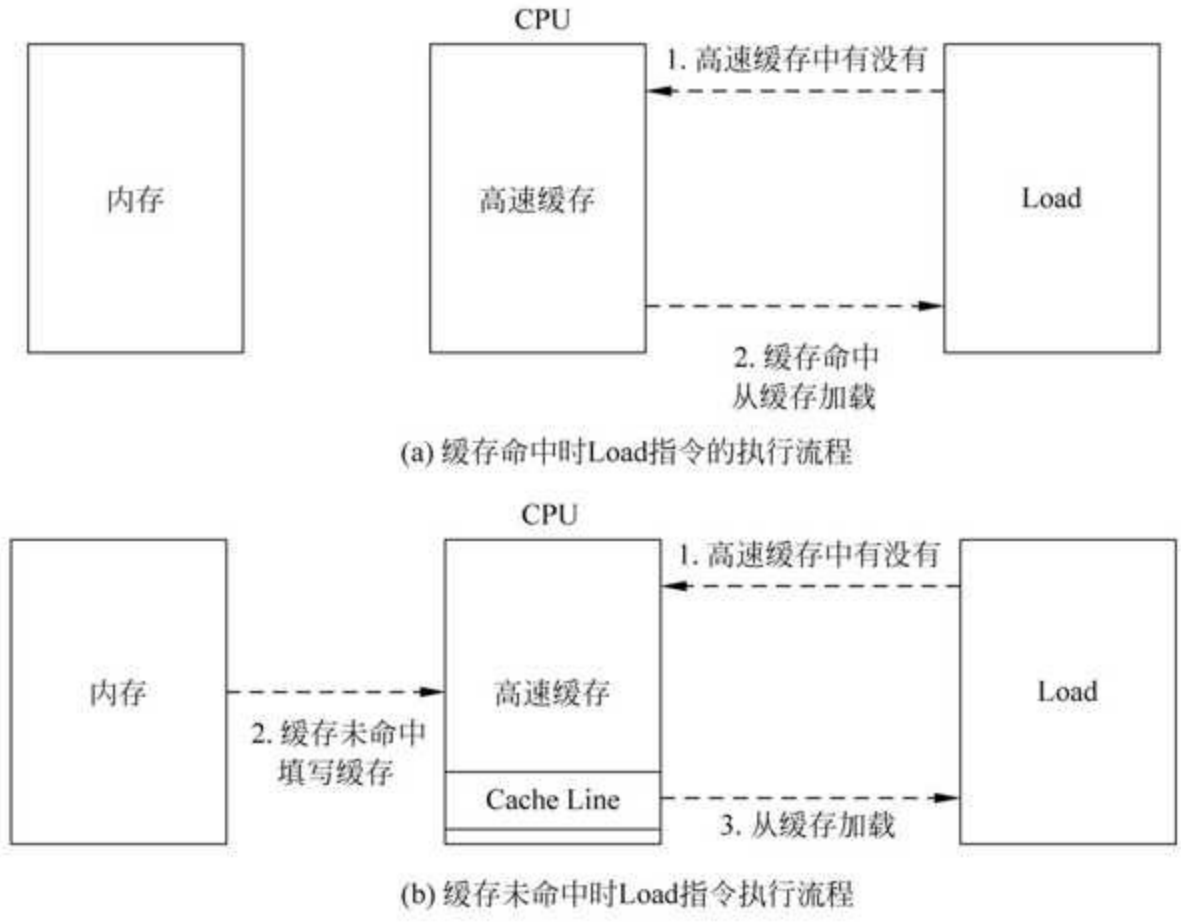

图7-3 Load指令的执行流程

Store指令的大致流程类似，先尝试写高速缓存，如果缓存命中，则写操作就完成了。如果缓存未命中，则先填充对应的Cache Line，然后写到Cache Line中，如图7-4所示。

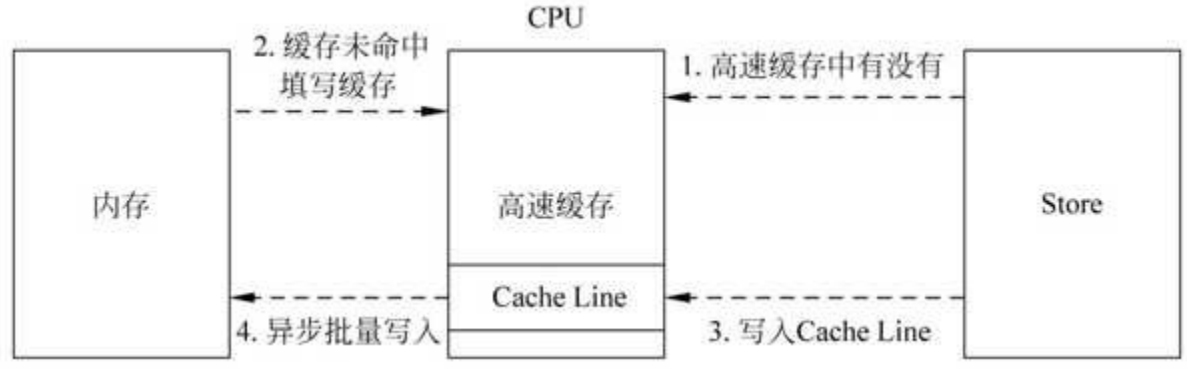

图7-4 Store指令执行流程

可能有些读者会对Store操作写之前要先填充Cache Line感到疑惑，这是因为高速缓存和内存之间的数据传输不是以字节为单位的，最小单位就是一个Cache Line。Cache Line大小因处理器的架构而异，常见的大小有32、64及128字节等。

在多核心的CPU上，Store操作会变得更复杂一些。每个CPU核心都拥有自己的高速缓存，例如x86的L1 Cache。写操作会修改当前核心的高速缓存，被修改的数据可能存在于多个核心的高速缓存中，CPU需要保证各个核心间的缓存一致性。目前主流的缓存一致性协议是MESI协议，MESI这个名字取自缓存单元可能的4种状态，分别是已修改的Modified，独占的Exclusive，共享的Shared和无效的Invalid。

如图7-5所示，当一个CPU核心要对自身高速缓存的某个单元进行修改时，它需要先通知其他CPU核心把各自高速缓存中对应的单元置为Invalid，再把自己的这个单元置为Exclusive，然后就可以进行修改了。

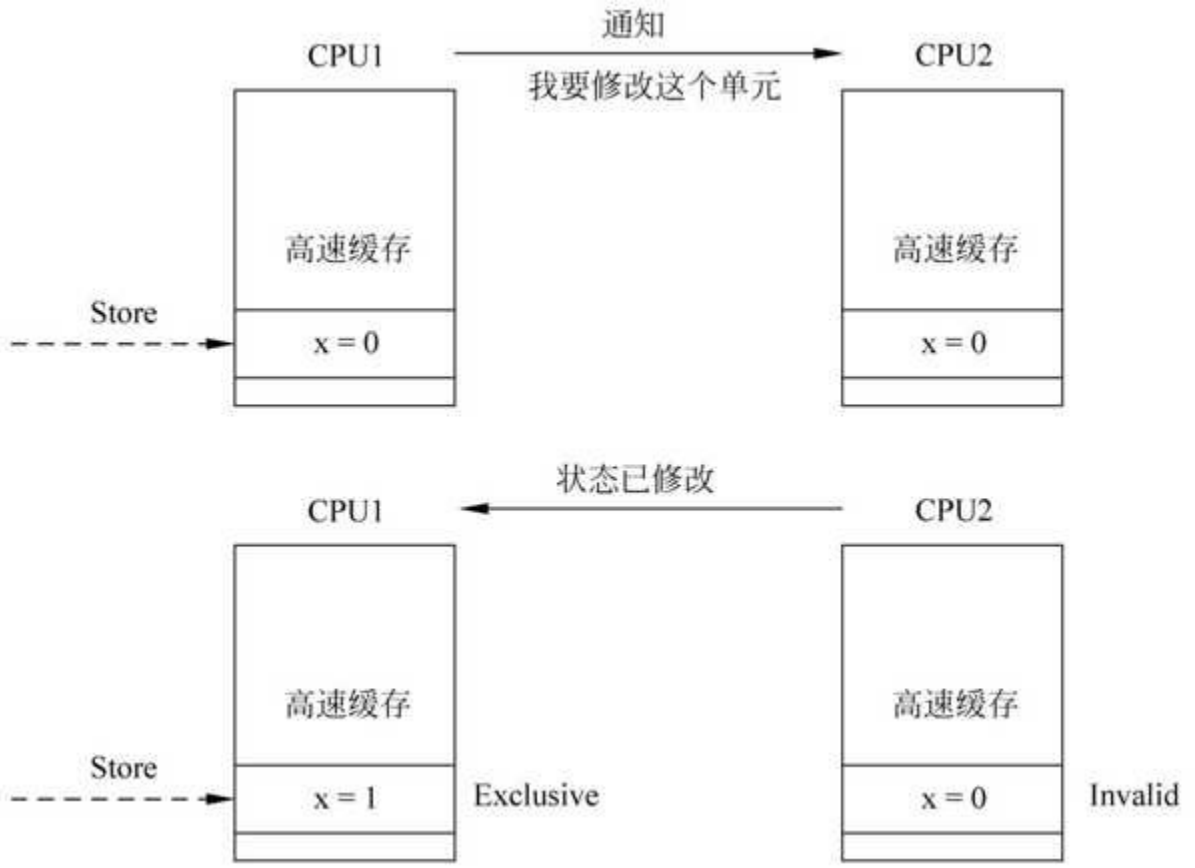

图7-5 一个CPU核心修改高速缓存数据单元的过程

这个过程涉及多核间的内部通信，是一个相对较慢的过程，为了避免当前核心因为等待而阻塞，CPU在设计上又引入了Store Buffer。当前核心向其他核心发出通知以后，可以先把要写的值放在Store Buffer中，然后继续执行后面的指令，等到其他核心完成响应以后，当前核心再把Store Buffer中的值合并到高速缓存中，如图7-6所示。

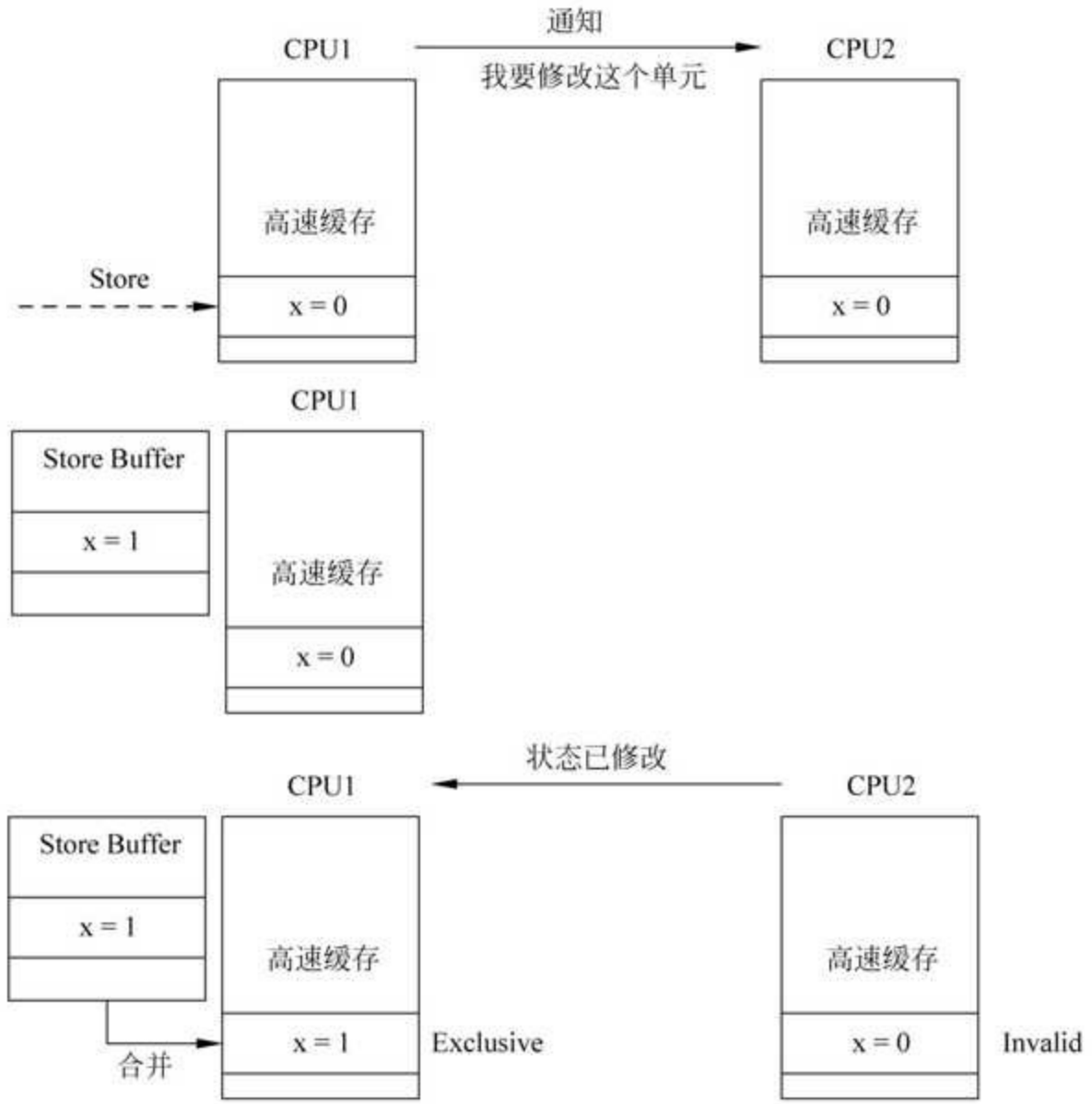

图7-6 引入Store Buffer后CPU修改高速缓存数据单元的过程

虽然高速缓存会保证多核一致性，但是Store Buffer却是各个核心私有的，因此对其他核心不可见。在Store-Load乱序中，从微观时序上，Load指令可能是在另一个线程的Store之后执行，但此时多核间通信尚未完成，对应的缓存单元还没有被置为Invalid，Store Buffer也没有被合并到高速缓存中，所以Load读到的是修改前的值。

如图7-7所示，如果协程一执行了Store命令，x的新值只是写入CPU1的Store Buffer，尚未合并到高速缓存，则此时协程二执行Load指令获得的x就是修改前的旧值0，而不是1。同样地，协程二修改y的值也可能只写入了CPU2的Store Buffer，所以协程一执行Load指令加载的y的值就是旧值0。

而当协程一执行最后一条Store指令时，b就被赋值为0。同样地，协程二会将a赋值为0。即使Store Buffer合并到高速缓存，x和y都被修改为新值，也已经晚了，如图7-8所示。

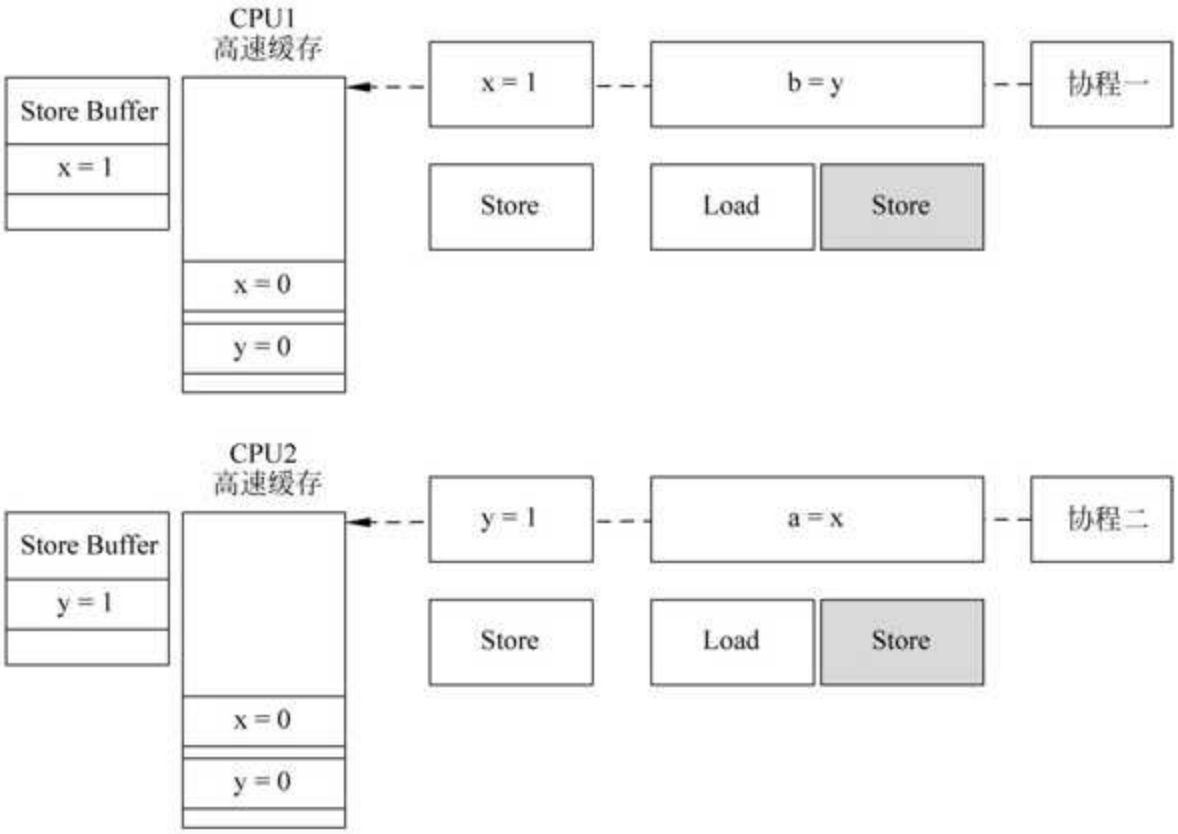

图7-7 写入Store Buffer后合并到高速缓存前Load数据

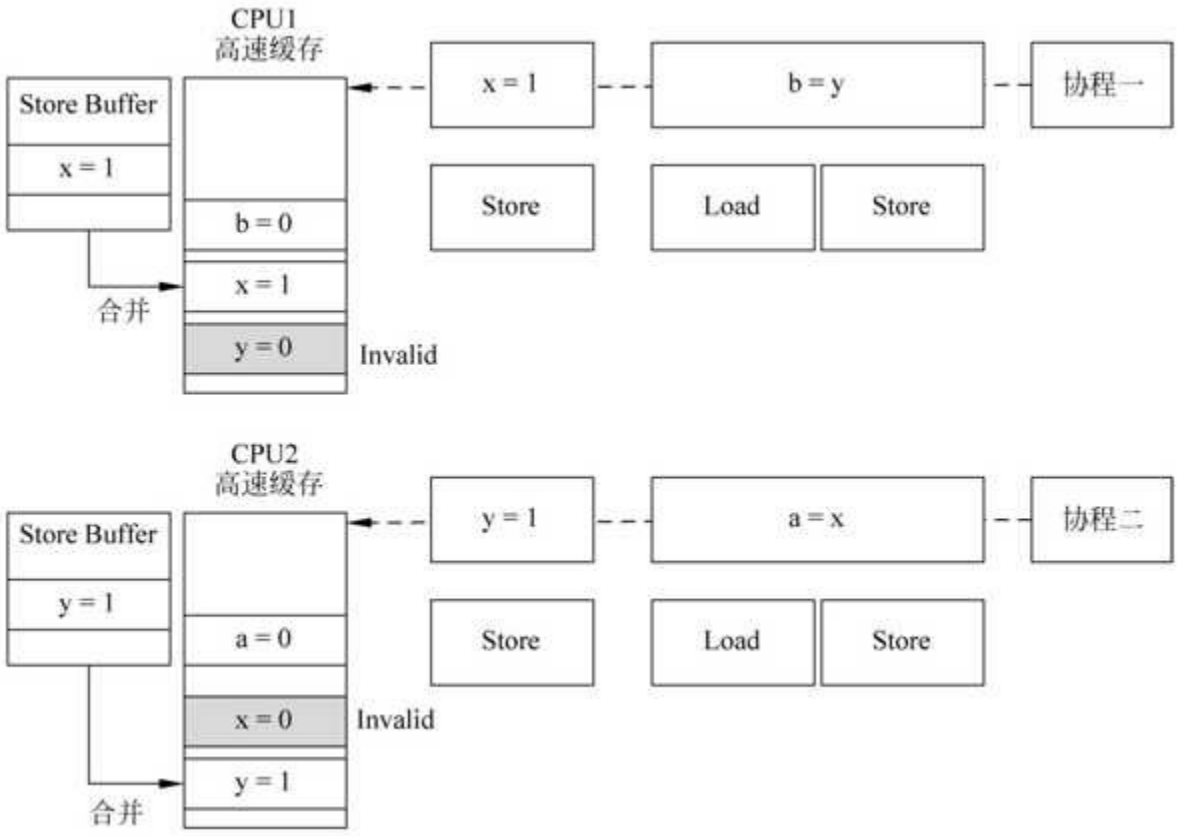

图7-8 合并到高速缓存后的数据状态

我们通过代码示例见证了x86的Store-Load乱序，Intel开发者手册上说x86只会出现这一种乱序。抛开固定的平台架构，理论上可能出现的乱序有4种：

（1）Load-Load，相邻的两条Load指令，后面的比前面的先读到数据。

（2）Load-Store，Load指令在前，Store指令在后，但是Store操作先变成全局可见，Load指令在此之后才读到数据。

（3）Store-Load，Store指令在前，Load指令在后，但是Load指令先读到了数据，Store操作在此之后才变成全局可见。这个我们已经在x86平台见证过了。

（4）Store-Store，相邻的两条Store指令，后面的比前面的先变成全局可见。

所谓的全局可见，指的是在多核CPU上对所有核心可见。因为笔者手边只有amd64架构的计算机，暂时无法验证其他几种乱序，有条件的读者可以在其他的架构上尝试一下。例如通过以下示例应该可以发现Store-Store乱序，代码如下：

```
//第 7 章/code_7_4.go
func main() {
    var wg sync.WaitGroup
    var x, y int64
    wg.Add(2)
    go func() {
        defer wg.Done()
        for i := 0; i < 1000000000; i++{
            if x == 0 {
                if y != 0 {
                    println("1:", i)
                }
                x = 1
                y = 1
            }
        }
    }()
    go func() {
        defer wg.Done()
        for i := 0; i < 1000000000; i++{
            if y == 1 {
                if x != 1 {
                    println("2:", i)
                }
                y = 0
                x = 0
            }
        }
    }()
    wg.Wait()
}
```

7.2.3 内存排序指令

执行期乱序会给结果带来很大的不确定性，这对于应用程序来讲是不能接受的，完全按照指令顺序执行又会使性能变差。为了解决这一问题，CPU提供了内存排序指令，应用程序在必要的时候能够通过这些指令来避免发生乱序。以目前的Intel x86处理器为例，提供了LFENCE、SFENCE和

MFENCE这3条内存排序指令，接下来我们就逐一分析它们的作用。

LFENCE是Load Fence的缩写，Fence翻译成中文是栅栏，可以认为起到分隔的作用，它会对当前核心上LFENCE之前的所有Load类指令进行序列化操作。具体来讲，针对当前CPU核心，LFENCE会在之前的所有指令都执行完后才开始执行，并且在LFENCE执行完之前，不会有后续的指令开始执行。特别是LFENCE之前的Load指令，一定会在LFENCE执行完成之前从内存接收到数据。LFENCE不会针对Store指令，Store指令之后的LFENCE可能会在Store写入的数据变成全局可见前执行完成。LFENCE之后的指令可以提前被从内存中加载，但是在LFENCE执行完之前它们不会被执行，即使是推测性的。

以上主要是Intel开发者手册对LFENCE的解释，它原本被设计用来阻止Load-Load乱序。让所有后续的指令在之前的指令执行完后才开始执行，这是Intel对功能的一个扩展，因此理论上它应该也能阻止Load-Store乱序。考虑到目前的x86 CPU不会出现这两种乱序，所以编程语言中暂时没有用到LFENCE指令进行多核同步，未来也许会用到。Go的runtime中用到了LFENCE的扩展功能来对RDTSC进行序列化，但是这并不属于同步的范畴。

SFENCE是Store Fence的缩写，它能够分隔两侧的Store指令，保证之前的Store操作一定会在之后的Store操作变成全局可见前先变成全局可见。结合7.2.2节的高速缓存和Store Buffer，笔者猜测SFENCE会影响到Store Buffer合并到高速缓存的顺序。

根据上述解释，SFENCE应该主要用来应对Store-Store乱序，由于现阶段的x86 CPU也不会出现这种乱序，所以编程语言暂时也未用到它进行多核同步。

MFENCE是Memory Fence的缩写，它会对之前所有的Load和Store指令进行序列化操作，这个序列化会保证MFENCE之前的所有Load和Store操作会在之后的任何Load和Store操作前先变成全局可见，所以上述3条指令中，只有MFENCE能够阻止Store-Load乱序。

我们对之前的示例代码稍做修改，尝试使用MFENCE指令来阻止Store-Load乱序，新的示例中用到了汇编语言，所以需要两个源码文件。首先是汇编代码文件fence_amd64.s，代码如下：

```
//第7章/fence_amd64.s
#include "textflag.h"
//func mfence()
TEXT ·mfence(SB), NOSPLIT, $0-0
    MFENCE
    RET
```

接下来是修改过的Go代码，被放置在fence.go文件中，跟之前会发生乱序的代码只有一点不同，就是在Store和Load之间插入了MFENCE指令，代码如下：

```
//第 7 章/fence.go
package main

func main() {
    s := [2]chan struct{}{
        make(chan struct{}, 1),
        make(chan struct{}, 1),
    }
    f := make(chan struct{}, 2)
    var x, y, a, b int64
    go func() {
        for i := 0; i < 1000000; i++{
            <-s[0]
            x = 1
            mfence()
            b = y
            f <- struct{}{}
        }
    }()
    go func() {
        for i := 0; i < 1000000; i++{
            <-s[1]
            y = 1
            mfence()
            a = x
            f <- struct{}{}
        }
    }()
    for i := 0; i < 1000000; i++{
        x = 0
        y = 0
        s[i%2] <- struct{}{}
        s[(i+1)%2] <- struct{}{}
        <-f
        <-f
        if a == 0 && b == 0 {
            println(i)
        }
    }
}

func mfence()
```

编译执行上述代码，会发现之前的Store-Load乱序不见了，程序不会有任何打印输出。如果将MFENCE指令换成LFENCE或SFENCE，就无法达到同样的目的了，感兴趣的读者可以自己尝试一下。

通过内存排序指令解决了执行期乱序造成的问题，但是这并不足以解决并发场景下的同步问题。要想结合代码逻辑轻松地实现多线程同步，就要用到专门的工具，这就是7.3节要介绍的锁。

7.3 常见的锁

本书的测试代码都比较简单，实际编程时的业务逻辑往往要复杂得多，需要同步保护的临界区中通常会有数十数百条指令，甚至更多。锁需要将所有线程（或协程）对临界区的访问进行串行化处理，需要同时保证两点要求：

（1）同时只能有一个线程获得锁，持有锁才能进入临界区。

（2）当线程离开临界区释放锁后，线程在临界区内做的所有操作都要全局可见。

本节会介绍几种在编程中常见的锁，并简单分析它们各自的实现原理，在此过程中需留意各种锁是如何保证以上两点要求的。

7.3.1 原子指令

软件层面的锁通常被实现为内存中的一个共享变量，加锁的过程至少需要3个步骤，按顺序依次是Load、Compare和Store。Load操作从内存中读取锁的最新状态，Compare操作用于检测是否处于未加锁状态，如果未加锁就通过Store操作进行修改，以便实现加锁。如果Compare发现已经处于加锁状态了，就不能执行后续的Store操作了。

如果用一般的x86汇编指令实现Load-Compare-Store操作，至少需要三条指令，例如CMP、JNE和MOV。CMP可以接收一个内存地址操作数，所以实质上包含了Load和Compare两步，JNE作为Compare的一部分用于实现条件跳转，MOV指令用来向指定内存地址写入数据，也就是Store操作，但是这样实现会有一个问题，我们知道线程用完时间片之后会被打断，假如线程a执行完CMP指令后发现未加锁，但是在执行MOV之前被打断了，然后线程b开始执行并获得了锁，接下来线程b在临界区中被打断，线程a恢复执行后也获得了锁，这样一来就会出现错误，如图7-9所示。

所以我们需要在一条指令中完成整个Load-Compare-Store操作，必须从硬件层面提供支持，例如x86就提供了CMPXCHG指令。

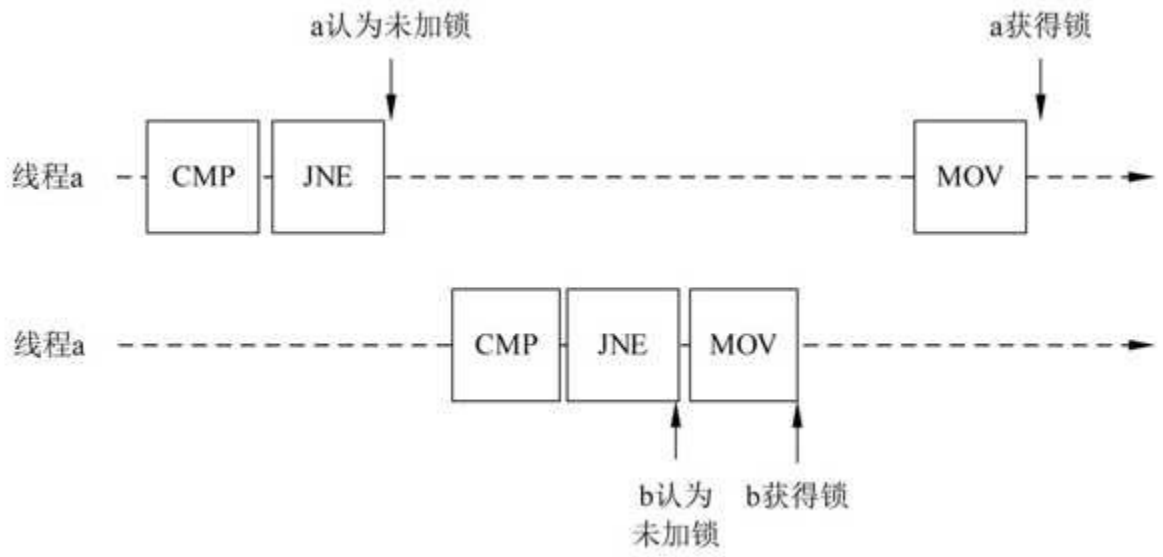

图7-9 同步问题

CMPXCHG是Compare and Exchange的缩写，该指令有两个操作数，用于实现锁的时候，第一操作数通常是个内存地址，也称为目的操作数，第二操作数是个通用寄存器。CMPXCHG会将AX寄存器和第一操作数进行比较，如果相等就把第二操作数复制到目的操作数中，若不相等就把目的操作数复制到AX寄存器中。基于这个指令实现锁，一条指令是不会在中间被打断的，所以就解决了之前的问题。

在单核环境下，任何能够通过一条指令完成的操作都可以称为原子操作，但是这也只适用于单核场景，在多核环境下，运行在不同CPU核心上的线程可能会并行加锁，不同核心同时执行CMPXCHG又会造成多个线程同时获得锁。如何解决这个问题呢？一种思路是，在当前核心执行CMPXCHG时，阻止其他核心执行CMPXCHG，x86汇编中的LOCK前缀用于实现这一目的。

LOCK前缀能够应用于部分内存操作指令，最简单的解释就是LOCK前缀会让当前CPU核心在当前指令执行期间独占总线，这样其他的CPU核心就不能同时操作内存了。事实上，只有对于不在高速缓存中的数据才会这样，对于高速缓存中的数据，LOCK前缀会通过MESI协议处理多核间缓存一致性。不管怎么说，加上LOCK前缀的CMPXCHG就无懈可击了。在多核环境下，这种带有LOCK前缀的指令也被称为原子指令。

至此，针对锁的两点要求，其中第1个可以通过原子指令实现了。那么如何做到第二点要求呢？就是释放锁之后，临界区内所有的操作要全局可见。事实上，锁本身的状态变化就必须是全局可见的，而且必须很及时，以保证高性能，因此，在x86 CPU上，LOCK前缀同时具有内存排序的作用，相当于在应用LOCK前缀的指令之后紧接着执行了一条MFENCE指令。综上所述，原子指令既能保证只允许一个线程进入临界区，又具有内存排序的作用，能够保证在锁的状态发生变化时，临界区中所有的修改随锁的状态一起变成全局可见。

7.3.2 自旋锁

自旋锁得以实现的基础是原子性的CAS操作，CAS即Compare And Swap，在x86平台上对应带有LOCK前缀的CMPXCHG指令。之所以称作自旋锁，是因为它会一直循环尝试CAS操作直到成功，看起来就像是一直在自旋等待。

接下来我们就尝试一下用汇编语言基于CMPXCHG指令实现一把自旋锁，首先在Go语言中基于int32创建一个自定义类型Spin，并为它实现Lock（）方法和Unlock（）方法，代码如下：

```
//第 7 章/code_7_5.go
type Spin int32

func (l *Spin) Lock() {
    lock((*int32)(l), 0, 1)
}

func (l *Spin) Unlock() {
    unlock((*int32)(l), 0)
}

func lock(ptr *int32, o, n int32)
func unlock(ptr *int32, n int32)
```

实际的加锁和解锁操作在lock（）和unlock（）这两个函数中实现，Go代码中只包含了这两个函数的原型声明，这两个函数是用汇编语言实现的，具体代码在spin_amd64.s文件中，代码如下：

```
//第 7 章/spin_amd64.s
#include "textflag.h"

//func lock(ptr * int32, old, new int32)
TEXT ·lock(SB), NOSPLIT, $0-16
    MOVQ  ptr+0(FP), BX
    MOVL  old+8(FP), DX
    MOVL  new+12(FP), CX
again:
    MOVL DX,AX
    LOCK
    CMPXCHGL  CX, 0(BX)
    JE    ok
    JMP   again
ok:
    RET

//func unlock(ptr * int32, val int32)
TEXT ·unlock(SB), NOSPLIT, $0-12
    MOVQ  ptr+0(FP), BX
    MOVL  val+8(FP), AX
    XCHGL AX, 0(BX)
    RET
```

lock（）函数把锁的地址放在了BX寄存器中，把用来比较的旧值old放到了DX寄存器中，把要写入的新值new放到了CX寄存器中。从标签again处开始是一个循环，每次循环开始前，把DX寄存器的值复制给AX寄存器，因为CMPXCHG隐含使用AX寄存器中的值作为比较用的旧值，并且可能会修改AX寄存器，所以每次循环需要重新赋值，这个循环不断尝试通过CMPXCHG进行加锁，成功后会通过JE指令跳出循环。因为Go的汇编风格有点类似于AT&T汇编，操作数书写顺序与Intel汇编相反，所以CMPXCHG的两个操作数中BX出现在CX右边。能够通过JE跳出循环，这是因为CMP操作会影响标志寄存器。

unlock（）函数通过XCHG指令将锁清零，实现了解锁操作。细心的读者可能会注意到这里没有LOCK前缀，根据Intel开发者手册所讲，XCHG指令隐含了LOCK前缀，所以代码中不用写，依然能够起到独占总线和内存排序的作用。

事实上，atomic包中的CompareAndSwapInt32（）函数和StoreInt32（）函数是基于CMPXCHG和XCHG这两条汇编指令实现的，所以上述的自旋锁可以改成完全用Go实现，代码如下：

```
//第 7 章/code_7_6.go
import "sync/atomic"

type Spin int32

func (l * Spin) Lock() {
    for !atomic.CompareAndSwapInt32((*int32)(l), 0, 1) {}
}

func (l * Spin) Unlock() {
    atomic.StoreInt32((*int32)(l), 0)
}
```

这样一来，我们确实实现了自旋锁，但是这跟生产环境中实际使用的自旋锁比起来还是有些差距。在锁竞争比较激烈的场景下，这种自旋会造成CPU使用率很高，所以还要进行优化。x86专门为此提供了PAUSE指令，它一方面能够提示处理器当前正处于自旋循环中，从而在退出循环的时候避免因检测到内存乱序而造成性能损失。另一方面，PAUSE能够大幅度减小自旋造成的CPU功率消

耗，从而达到节能和减少发热的效果。

可以把PAUSE指令加入我们汇编版本的lock（）函数实现中，修改后的代码如下：

```
//第 7 章/lock_amd64.s
//func lock(ptr * int32, old, new int32)
TEXT ·lock(SB), NOSPLIT, $0-16
    MOVQ  ptr+0(FP), BX
    MOVL  old+8(FP), DX
   MOVL  new+12(FP), CX
again:
   MOVL  DX,AX
    LOCK
    CMPXCHGL  CX, 0(BX)
    JE    ok
   PAUSE
    JMP   again
ok:
    RET
```

也可以把PAUSE指令单独放在一个函数中，这样就能够跟atomic包中的函数结合使用了，代码如下：

```
//第 7 章/pause_amd64.s
#include "textflag.h"

//func pause()
TEXT pause(SB), NOSPLIT, $0-0
    PAUSE
    RET
```

然后就能对Go代码实现的自旋锁进行优化了，代码如下：

```
//第 7 章/code_7_7.go
func (l *Spin) Lock() {
    for !atomic.CompareAndSwapInt32((*int32)(l), 0, 1) {
        pause()
    }
}
```

自旋锁的优点是比较轻量，不过它对适用的场景也是有要求的。首先，在单核心的环境下不适合使用自旋锁，因为单核系统上任一时刻只能有一个线程在运行，当前线程一直在自旋等待，而持有锁的线程得不到运行，锁就不可能被释放，等也是白等，纯属浪费CPU资源。这种情况下及时切换到其他可运行的线程会更高效一些，因此在单核环境下更适合用调度器对象。其次，即使是在多核环境下，也要考虑平均持有锁的时间，以及程序的并发程度等因素。在持有锁的时间占比很小，并且活跃线程数接近CPU核心数量时，自旋锁比较高效，也就是自旋的代价小于线程切换的代价。其他情况就不一定了，要结合实际场景分析再加上充分的测试。

7.3.3 调度器对象

笔者使用调度器对象这个名字，主要是受Windows NT内核的影响。更通俗地讲，应该说是操作系统提供的线程间同步原语，一般以一组系统调用的形式存在。例如Win32的Event，以及Linux的futex等。基于这些同步原语，可以实现锁及更复杂的同步工具。

这些调度器对象与自旋锁的不同主要是有一个等待队列。当线程获取锁失败时不会一直在那里自旋，而是挂起后进入等待队列中等待，然后系统调度器会切换到下一个可运行的线程。等到持有锁的线程释放锁的时候，会按照一定的算法从等待队列中取出一个线程并唤醒它，被唤醒的线程会获得所有权，然后继续执行。这些同步原语是由内核提供的，直接与系统的调度器交互，能够挂起和唤醒线程，这一点是自旋锁做不到的。等待队列可以实现支持FIFO、FILO，甚至支持某种优先级策略，但是也正是由于是在内核中实现的，所以应用程序需要以系统调用的方式来使用它，这就造成了一定的开销。在获取锁失败的情况下还会发生线程切换，进一步增大开销。调度器对象和自旋锁各自有适用的场景，具体如何选用还要结合具体场景来分析。

7.3.4 优化的锁

通过7.3.2节和7.3.3节，我们大致了解了自旋锁与调度器对象。前者主要适用于多核环境，并且持有锁的时间占比较小的情况。这种情况下，往往在几次自旋之后就能获得锁，比起发生一次线程切换的代价要小得多。后者主要适用于加锁失败就要挂起线程的场景，例如单核环境，或者持有锁的时间占比较大的情况，而在实际的业务逻辑中，持有锁的时间往往不是很确定，有可能较短也有可能较长，我们不好一概用一种策略进行处理，如果将两者结合，或许会有不错的效果。

将自旋锁和调度器对象结合，理论上就可以得到一把优化的锁了。加锁时首先经过自旋锁，但是需限制最大自旋次数，如果在有限次数内加锁成功也就成功了，否则就进一步通过调度器对象将当前线程挂起。等到持有锁的线程释放锁的时候，会通过调度器对象将挂起的线程唤醒。这样就结合了二者的优点，既避免了加锁失败立即挂起线程造成过多的上下文切换，又避免了无限制地自旋而空耗CPU，这也是如今主流的锁实现思路。

7.4 Go语言的同步

7.1～7.3节用了很大的篇幅讲解了与同步相关的一些理论基础，本节就回归到Go语言上来，结合runtime源码，分析一下与同步相关的组件的实现原理。

7.4.1 runtime.mutex

在Go 1.14版本的runtime中，mutex的定义代码如下：

```
type mutex struct {
    key uintptr
}
```

在Go 1.15及以后的版本中为了支持静态的Lock Rank而添加了lockRankStruct，这里暂时不需要关心。

runtime.mutex被runtime自身的代码使用，它是针对线程而设计的，不适用于协程。它本质上就是一个结合了自旋锁和调度器对象的优化过的锁，自旋锁部分没有什么特殊的，调度器对象部分在不同平台上需要使用不同的系统调用。在Linux上是基于futex实现的，该实现中把mutex.key作为一个uint32来使用，并且为其定义了3种状态，对应的3个常量的定义代码如下：

```
mutex_unlocked = 0
mutex_locked   = 1
mutex_sleeping = 2
```

unlocked表示当前处于未加锁状态，locked则表示已加锁状态，sleeping比较特殊一点，表示当前有线程因未能获得锁而通过futex睡眠等待。加锁函数的源代码如下：

```
func lock2(l *mutex) {
    gp := getg()

    if gp.m.locks < 0 {
        throw("runtime·lock: lock count")
    }
    gp.m.locks++

    v := atomic.Xchg(key32(&l.key), mutex_locked)
    if v == mutex_unlocked {
        return
    }

    wait := v

    spin := 0
    if ncpu > 1 {
        spin = active_spin
    }
    for {
        for i := 0; i < spin; i++{
            for l.key == mutex_unlocked {
                if atomic.Cas(key32(&l.key), mutex_unlocked, wait) {
                    return
                }
            }
            procyield(active_spin_cnt)
        }

        for i := 0; i < passive_spin; i++{
            for l.key == mutex_unlocked {
                if atomic.Cas(key32(&l.key), mutex_unlocked, wait) {
                    return
                }

            }
            osyield()
        }

        v = atomic.Xchg(key32(&l.key), mutex_sleeping)
        if v == mutex_unlocked {
            return
        }
        wait = mutex_sleeping
        futexsleep(key32(&l.key), mutex_sleeping, -1)
    }
}
```

首先通过atomic.Xchg（）函数将l.key替换成mutex_locked，然后判断原始值v，如果等于mutex_unlocked，就说明原本处于未加锁状态，而我们现在已经通过原子操作加了锁，这样就可以返回了。

既然v不等于mutex_unlocked，那就只能是mutex_locked和mutex_sleeping二者之一了，先把它的值

暂存在wait中。接下来根据处理器核心数ncpu是否大于1来决定是否需要自旋，因为在单核心系统上自旋是没有意义的。active_spin是个值为4的常量，表示主动自旋4次。

接下来就是尝试加锁的大循环了，大循环内部先经过两个小循环。第1个小循环是主动自旋的循环，它会循环spin次，也就是单核环境下循环0次，多核环境下循环4次。每次尝试之后都会通过procyield（）函数来稍微拖延一下时间，procyield（）函数是汇编语言实现的函数，代码如下：

```
//func procyield(cycles uint32)
TEXT runtime·procyield(SB),NOSPLIT,$0-0
    MOVL    cycles+0(FP), AX
again:
    PAUSE
    SUBL    $1, AX
    JNZ    again
    RET
```

实际上就是循环执行PAUSE指令。active_spin_count是个值为30的常量，所以就是循环执行30次PAUSE。

第2个小循环是个被动自旋循环。passive_spin是个值为1的常量，所以只会循环一次。之所以称为被动自旋，是因为它调用了osyield（）函数来等待，这也是它与主动自旋的唯一一点不同。osyield（）函数也是个用汇编语言实现的函数，它通过执行系统调用来切换至其他线程，代码如下：

```
TEXT runtime·osyield(SB),NOSPLIT,$0
    MOVL    $SYS_sched_yield, AX
    SYSCALL
    RET
```

上述两个循环的主要工作都是检测锁是否已经被释放了，假如有一个锁l，线程b尝试加锁，进入加锁的大循环，经过主动自旋和被动自旋两个小循环，如果自旋过程中发现锁被释放了，并且锁的原始状态为mutex_locked，则表示在b加锁之前有其他线程持有锁，却没有线程在等待它，所以就将l置为mutex_locked。若是锁的原始状态为mutex_sleeping，则表示已经有其他线程在等待这个锁了，那么现在即使线程b获得了锁，也应该将锁置为mutex_sleeping。

总而言之，只要自旋过程中加锁成功，就得将锁置为其原始值，也就是源码中保存到wait中的状态，如图7-10所示。

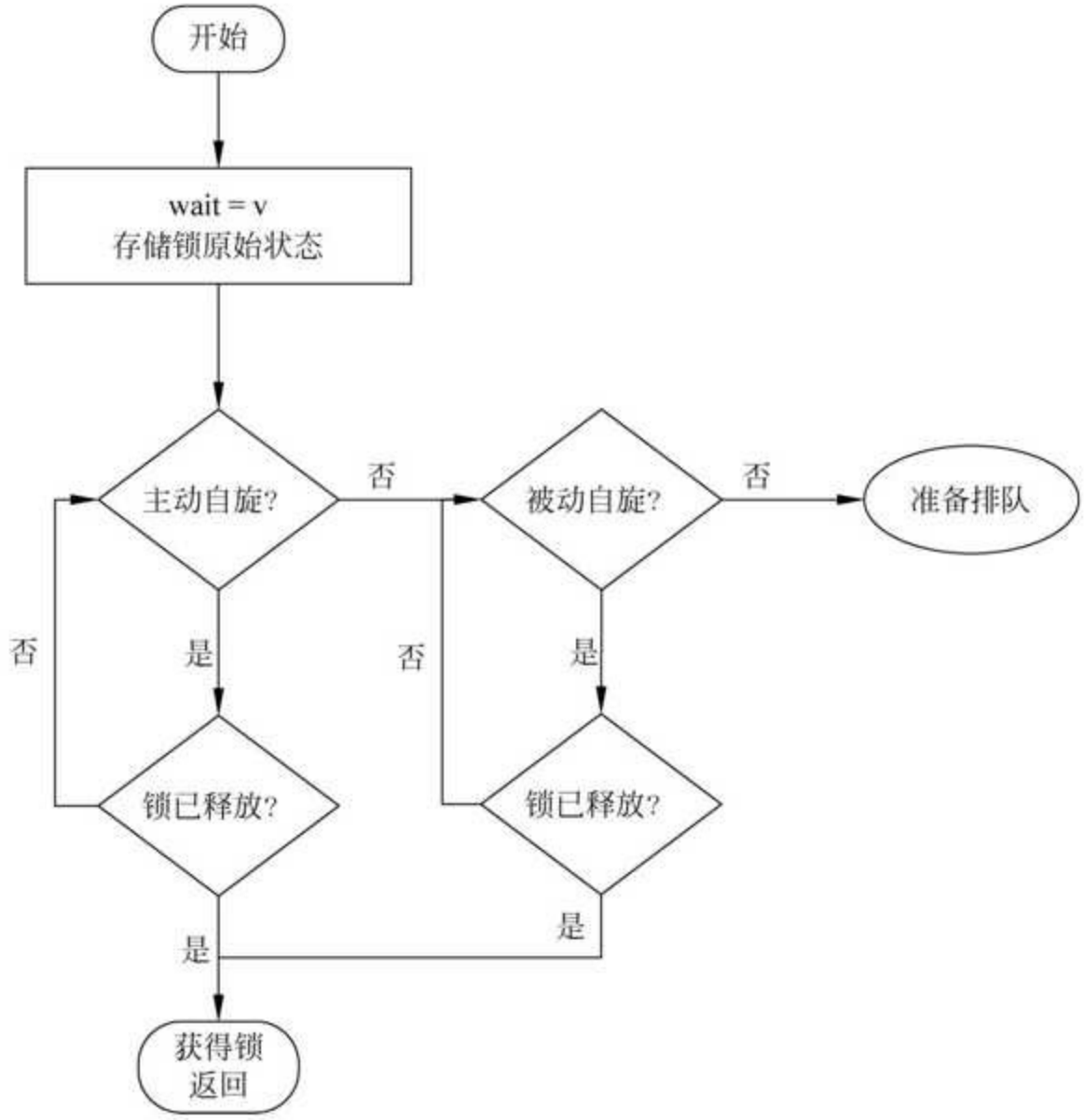

图7-10 自旋过程中获得锁

这里需要注意一下，如果持有锁的线程在释放的时候发现锁的状态为mutex_sleeping，就会通过futex唤醒睡眠等待的线程。假如线程a持有锁l，线程b在睡眠等待这个锁，接下来线程c尝试加锁，它首先通过atomic.Xchg（）函数把锁的状态替换为mutex_locked，然后进入自旋。恰巧，在线程c自旋过程中线程a要释放锁，但此时锁的状态为mutex_locked，释放锁时不会去唤醒等待的线程，而线程c却会获得锁，不过会将锁恢复为mutex_sleeping状态。这一过程中锁l的状态变化如图7-11所示。

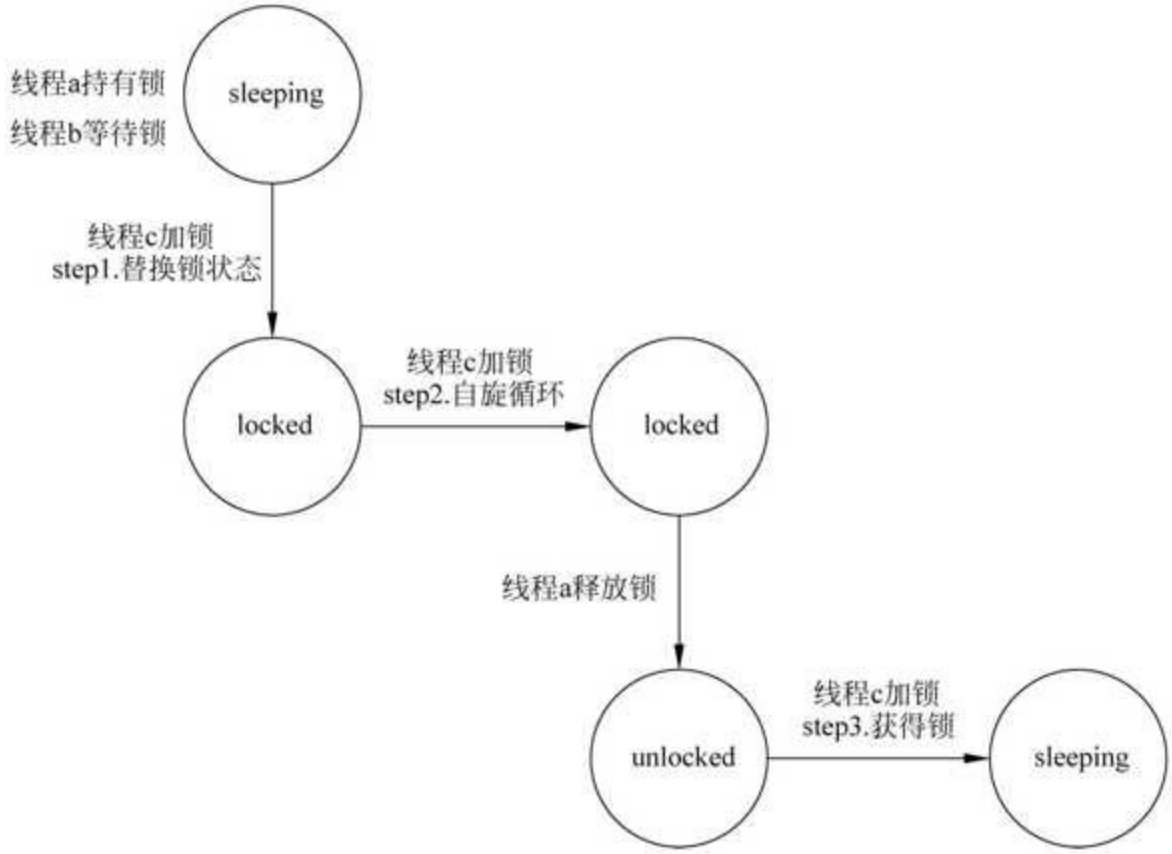

图7-11 一次插队过程中锁的状态变化

整个过程下来，相当于线程c跳过了futex排队，直接从锁的上一个持有者线程a那里接收了所有权，通过futex唤醒睡眠线程的操作被推后了，但是并没有被忘记。这样相当于发生了一次插队，但是避免了一次线程切换，从整体上来看会提升性能。

然而，若是经过上述两个自旋循环都没能获得锁，就可以通过atomic.Xchg（）函数把l.key替换为mutex_sleeping，因为当前线程准备要去睡眠等待了，但是仍要在真正去睡眠之前，检查一下锁是否被释放了，若已经释放，则当前线程仍可以加锁成功，然后就可以直接返回了。不过直接将mutex_sleeping状态保留在锁中，可能会有点小问题。

因为，可能锁的原始状态为mutex_locked，并没有线程在futex上睡眠等待这个锁，因而在释放锁的时候可能会有多余的唤醒操作。不过没有关系，这样的小问题会被忽略，只要保证不丢失应有的唤醒就可以了。

在大循环的最后，如果来到这里就表示之前所有尝试都没能获得锁，所以就调用futexsleep让当前线程挂起，超时时间-1表示会一直睡眠直到被唤醒。

解锁函数的逻辑比较简单，主要通过atomic.Xchg（）函数将l.key替换成mutex_unlocked，然后检查替换前的旧值，如果等于mutex_sleeping，就通过futexwakeup唤醒一个线程。感兴趣的读者可自行查看源码，这里不再赘述。

7.4.2 semaphore

runtime中的semaphore是可供协程使用的信号量实现，预期用它来提供一组sleep和wakeup原语，目标与Linux的futex相同。也就是说，不要把它视为信号量，而是应把它当成实现睡眠和唤醒的一种方式。每个sleep都与一次wakeup对应，即使因为竞争的关系，wakeup发生在sleep之前。

semaphore的核心逻辑是通过semacquire1（）函数和semrelease1（）函数实现的，semacquire1（）函数用来执行获取操作，函数的原型如下：

```
func semacquire1(addr * uint32, lifo bool, profile semaProfileFlags, skipframes int)
```

参数addr是用作信号量的uint32型变量的地址，lifo表示是否采用LIFO的排队策略。profile与性能分析相关，表示要进行哪些种类的采样，目前有semaBlockProfile和semaMutexProfile两种。skipframes用来指示栈回溯跳过runtime自身的栈帧。

semrelease1（）函数用来执行释放操作，函数的原型如下：

```
func semrelease1(addr * uint32, handoff bool, skipframes int)
```

handoff参数表示是否立即切换到被唤醒的协程。被唤醒的协程会设置到当前P的runnext，如果handoff为true，则当前协程会通过goyield（）让出CPU，被唤醒的协程会立刻得到调度。

runtime内部会通过一个大小为251的semtable来管理所有的semaphore，semtable的定义代码如下：

```
const semTabSize = 251

var semtable [semTabSize]struct {
    root semaRoot
    pad  [cpu.CacheLinePadSize - unsafe.Sizeof(semaRoot{})]Byte
}
```

如果只是一个大小固定的table，则肯定无法管理运行阶段数量不定的semaphore。事实上，runtime会把semaphore放到平衡树中，而semtable存储的是251棵平衡树的根，对应数据结构为semaRoot。semaRoot的定义代码如下：

```
type semaRoot struct {
    lock  mutex
    treap * sudog
    nwait uint32
}
```

lock用来保护这棵平衡树，treap字段是真正的平衡树数据结构的根，nwait字段表明了树中结点的数量，实际上平衡树的每个节点都是个sudog类型的对象，代码如下：

```
type sudog struct {
    g * g
    is Select bool
    next          * sudog
    prev          * sudog
    elem          unsafe.Pointer //data element (may point to stack)
    acquiretime   int64
    releasetime   int64
    ticket        uint32
    parent        * sudog //semaRoot binary tree
    waitlink      * sudog //g.waiting list or semaRoot
    waittail      * sudog //semaRoot
    c             * hchan //channel
}
```

sudog.g用于记录当前排队的协程，sudog.elem用于存储对应信号量的地址。当要使用一个信号量时，需要提供一个记录信号量数值的变量，根据它的地址addr进行计算并映射到semtable中的一棵

平衡树上，semroot（）函数专门用来把addr映射到对应平衡树的根，代码如下：

```
func semroot(addr *uint32) *semaRoot {
    return &semtable[(uintptr(unsafe.Pointer(addr))>>3)%semTabSize].root
}
```

它先把addr转换成uintptr，然后对齐到8字节，再对表的大小取模，结果用作数组下标。定位到某棵平衡树之后，再根据sudog.elem存储的地址与信号量变量的地址是否相等，进一步定位到某个节点，这样就能找到该信号量对应的等待队列了。

如图7-12所示，semtable中序号为0的平衡树包括5个节点，代表有5个不同的信号量通过地址计算并映射到这棵平衡树，而sudog节点d、e、f属于同一个信号量的等待队列，通过sudog.waitlink和sudog.waittail连接起来。

semacquire1（）函数会先通过调用cansemacquire（）函数来判断能否在不等待的情况下获取信号量，该函数的源码如下：

```
func cansemacquire(addr *uint32) bool {
    for {
        v := atomic.Load(addr)
        if v == 0 {
            return false
        }
        if atomic.Cas(addr, v, v-1) {
            return true
        }
    }
}
```

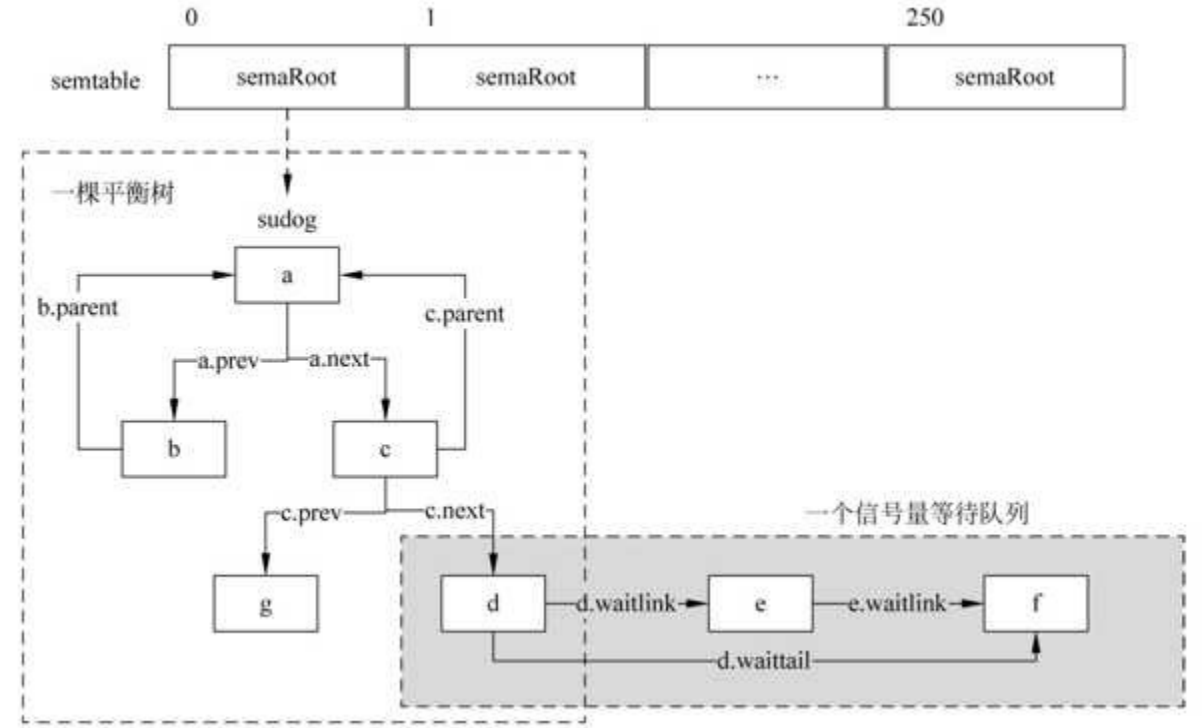

图7-12 semtable示例结构

其实很简单，在信号量的值大于0的前提下，循环尝试将信号量的值原子性地减1。如果成功了就返

回值true，上一层的semacquire1（）函数也就可以直接返回了。如果在减1之前发现信号量的值已经是0了，就返回值false，上一层的semacquire1（）函数就需要执行后续的排队逻辑了。排队逻辑是在一个for循环中实现的，因为有可能需要多次尝试，代码如下：

```
for {
    lockWithRank(&root.lock, lockRankRoot)
    atomic.Xadd(&root.nwait, 1)
    if cansemacquire(addr) {
        atomic.Xadd(&root.nwait, -1)
        unlock(&root.lock)
        break
    }
    root.queue(addr, s, lifo)
    goparkunlock(&root.lock, waitReasonSemacquire, traceEvGoBlockSync, 4+skipframes)
    if s.ticket != 0 || cansemacquire(addr) {
        break
    }
}
```

首先对root.lock加锁，然后把root.nwait加1，因为当前协程即将到平衡树中去等待了。再次尝试cansemacquire（）函数，这个尝试是必要的，因为这期间可能有其他协程释放了信号量，而且要注意操作nwait和addr的顺序，这里是先把nwait加1，后检测addr中的值，semrelease1中会先把addr中的值加1，后检测nwait，这样能够避免漏掉应有的唤醒。继续回到调用cansemacquire（）函数这里，如果返回值为true，也就表明获取了信号量，不需要进入平衡树等待了，因此再把nwait减去1，释放锁，然后跳出循环。若cansemacquire（）函数的返回值为false，就要继续排队的流程。通过调用root.queue（）方法，把与当前协程关联的sudog节点添加到平衡树中，然后调用gopark（）函数挂起当前协程。

semacquire1（）函数的核心逻辑基本上就是这些，再来看一下semrelease1（）函数，摘选部分关键代码如下：

```
root := semroot(addr)
atomic.Xadd(addr, 1)

if atomic.Load(&root.nwait) == 0 {
    return
}

lockWithRank(&root.lock, lockRankRoot)
if atomic.Load(&root.nwait) == 0 {
    unlock(&root.lock)
    return
}
s, t0 := root.dequeue(addr)
if s != nil {
    atomic.Xadd(&root.nwait, -1)
}
unlock(&root.lock)
```

它会先把信号量的值加1，然后判断nwait是否为0，如果没有协程在等待就直接返回了。否则就要对root.lock加锁，再次判断nwait是否为0，若不为0就通过root.dequeue（）方法从队列中取出一个协程，然后把nwait减去1并解锁。后面的代码通过goready（）函数唤醒协程，并按需调用goyield（）函数，以便让出CPU，这里就不把代码全贴出来了。

关于semaphore的探索就讲解到这里，它是为协程而设计的，也是7.4.3节中要介绍的sync.Mutex的基础。

7.4.3 sync.Mutex

Mutex这个名称的由来，应该是Mutual Exclusion的前缀组合，俗称互斥体或互斥锁。它是一把结合了自旋锁与信号量的优化过的锁，先来看一下Go语言sync包中Mutex的数据结构，代码如下：

```
type Mutex struct {
    state int32
    sema  uint32
}
```

因为足够简单，所以不需要额外的初始化，此结构的零值就是一个有效的互斥锁，处于Unlocked状态。state存储的是互斥锁的状态，加锁和解锁方法都是通过atomic包提供的函数原子性地操作该字段。那么，加锁失败时该如何排队等待这个Mutex呢？答案就是7.4.2节介绍的信号量。这里的sema字段用作信号量，为Mutex提供等待队列。

1. Mutex工作模式

Mutex有两种模式：正常模式和饥饿模式。正常模式下，一个尝试加锁的goroutine会先自旋几次，尝试通过原子操作获得锁，若几次自旋之后仍不能获得锁，则通过信号量排队等待。所有的等待者会按照先入先出（FIFO）的顺序排队，但是当一个等待者被唤醒后并不会直接拥有锁，而是需要和后来者（处于自旋阶段，尚未排队等待的协程）竞争。

这种情况下后来者更有优势，一方面原因是后来者正在CPU上运行，自然比刚被唤醒的goroutine更有优势，另一方面处于自旋状态的goroutine可以有很多，而被唤醒的goroutine每次只有一个，所以被唤醒的goroutine有很大概率获得不到锁，这种情况下它会被重新插入队列的头部，而不是尾部。当一个goroutine本次加锁等待的时间超过了1ms后，它会把当前Mutex切换至饥饿模式。

在饥饿模式下，Mutex的所有权从执行Unlock的goroutine直接传递给等待队列头部的goroutine。后来者不会自旋，也不会尝试获得锁，它们会直接从队列的尾部排队等待，即使Mutex处于Unlocked状态。

当一个等待者获得了锁之后，它会在以下两种情况时将Mutex由饥饿模式切换回正常模式：

（1）它是最后一个等待者，即等待队列空了。

（2）它的等待时间小于1ms，也就是它刚来不久，后面自然更没有饥饿的goroutine了。

正常模式下Mutex有更好的性能，但是饥饿模式对于防止尾端延迟（队列尾端的goroutine迟迟抢不到锁）来讲特别重要。

综上所述，在正常模式下自旋和排队是同时存在的，执行Lock的goroutine会先一边自旋一边通过原子操作尝试获得锁，尝试过几次后如果还没获得锁，就需要去排队等待了。这种在排队之前，先让大家来抢的模式，能够有更高的吞吐量，因为频繁地挂起、唤醒goroutine会带来较多的开销，但是又不能无限制地自旋，要把自旋的开销控制在较小的范围内，而饥饿模式下不再自旋尝试，所有goroutine都要排队，严格地按先来后到执行。

2. Mutex的状态

与Mutex的state字段相关的几个常量定义如下：

```
mutexLocked = 1 << iota //1
mutexWoken              //2
mutexStarving           //4
mutexWaiterShift = iota //3
```

mutexLocked表示互斥锁处于Locked状态。mutexWoken表示已经有goroutine被唤醒了，当该标志位被设置时，Unlock操作不会唤醒排队的goroutine。mutexStarving表示饥饿模式，该标志位被设置时Mutex工作在饥饿模式，清零时Mutex工作在正常模式。mutexWaiterShift表示除了最低3位以外，state的其他位用来记录有多少个等待者在排队。Mutex.state标志位如图7-13所示。

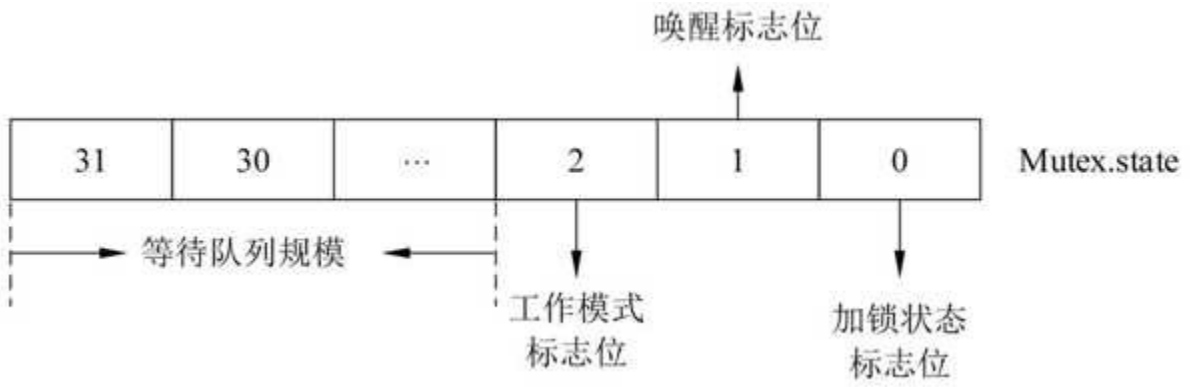

图7-13 Mutex.state标志位

3．Lock（）和Unlock（）方法

精简了注释和部分与race检测相关的代码，两个方法的代码如下：

```
func (m *Mutex) Lock() {
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        return
    }
    m.lockSlow()
}

func (m *Mutex) Unlock() {
    new := atomic.AddInt32(&m.state, -mutexLocked)
    if new != 0 {
        m.unlockSlow(new)
    }
}
```

这两个方法主要通过atomic函数实现了Fast path，相应的Slow path被单独放在了lockSlow（）方法和unlockSlow（）方法中。根据源码注释的说法，这样是为了便于编译器对Fast path进行内联优化。

1）Fash path

Lock（）方法的Fast path期望Mutex处于Unlocked状态，没有goroutine在排队，更不会饥饿。理想状况下，一个CAS操作就可以获得锁了。如果CAS操作没能获得锁，就需要进入Slow path了，也就是lockSlow（）方法。

Unlock（）方法同理，首先通过原子操作从state中减去mutexLocked，也就是释放锁，然后根据state的新值来判断是否需要执行Slow path。如果新值为0，也就意味着没有其他goroutine在排队，

所以不需要执行额外操作；如果新值不为0，则可能需要唤醒某个goroutine。

2）Slow path

lockSlow（）方法的逻辑比较复杂，需要整体上来理解，笔者通过注释对关键代码进行解释，代码如下：

```
func (m *Mutex) lockSlow() {
    var waitStartTime int64
    starving := false
    awoke := false
    iter := 0
    old := m.state
    for {
        //饥饿模式下不要自旋,因为所有权按顺序传递,自旋没有意义
        if old&(mutexLocked|mutexStarving) == mutexLocked
            && runtime_canSpin(iter) {
            //当前处于"主动自旋",尝试设置 mutexWoken 标识
            //以避免 Unlock 方法唤醒更多 goroutine
            if !awoke && old&mutexWoken == 0
            && old>>mutexWaiterShift != 0
            && atomic.CompareAndSwapInt32(&m.state, old, old|mutexWoken) {
                awoke = true
            }
            runtime_doSpin()
            iter++
            old = m.state
            continue
        }
        new := old
        //不要尝试获得处于饥饿模式的 mutex,后来者必须排队
        if old&mutexStarving == 0 {
```

```
            new | = mutexLocked
        }
        if old&(mutexLocked|mutexStarving) != 0 {
            new += 1 << mutexWaiterShift
        }
            //当前 goroutine 将 mutex 切换至饥饿模式
            //如果 mutex 已经处于 unlocked 状态,就不要切换了
        //因为 Unlock()方法认为处于饥饿模式的 mutex 等待队列不为空
            if starving && old&mutexLocked != 0 {
                new | = mutexStarving
        }
        if awoke {
                //当前 goroutine 是被唤醒的,检查并清除 mutexWoken 标志位
                if new&mutexWoken == 0 {
                    throw("sync: inconsistent mutex state")
            }
            new & ^ = mutexWoken
            }
            if atomic.CompareAndSwapInt32(&m.state, old, new) {
                if old&(mutexLocked|mutexStarving) == 0 {
                    break //通过 CAS 操作获得了锁
            }
                //被唤醒之后没有抢到锁,需要插入队列头部,而不是尾部
            queueLifo : = waitStartTime != 0
            if waitStartTime == 0 {
                    waitStartTime = runtime_nanotime()
            }
            runtime_SemacquireMutex(&m.sema, queueLifo, 1)
            starving = starving || runtime_nanotime() - waitStartTime > starvationThresholdNs
            old = m.state
            if old&mutexStarving != 0 {
                    //当前代码位置 goroutine 肯定是被唤醒的,而且 mutex 处于饥饿模式
                    //所有权被直接交给当前 goroutine
                    //但是这种情况下 mutex 的 state 会与实际情况不一致
                    //mutexLocked 标志位没有设置
                    //而且等待者计数中也没有减去当前 goroutine.需要修复 state
                    //注意,饥饿模式下传递 mutex 所有权不会设置 mutexWoken 标志
                    //只有正常模式下唤醒才会
                    if old&(mutexLocked|mutexWoken) != 0
                    || old >> mutexWaiterShift == 0 {
                        throw("sync: inconsistent mutex state")
            }
            delta : = int32(mutexLocked - 1 << mutexWaiterShift)
            if !starving || old >> mutexWaiterShift == 1 {
            //退出饥饿模式,至关重要
```

```
            delta -= mutexStarving
            }
            atomic.AddInt32(&m.state, delta)
            break
        }
        awoke = true
        iter = 0
        } else {
            old = m.state
        }
    }
}
```

然后是与之对应的unlockSlow（）函数的代码如下：

```
func (m * Mutex) unlockSlow(new int32) {
    if (new + mutexLocked)&mutexLocked == 0 {
        throw("sync: unlock of unlocked mutex")
    }
    if new&mutexStarving == 0 {
        old := new
        for {
            //如果等待队列为空或者已经有一个 goroutine 被唤醒或获得了锁
            //就不需要再去唤醒某个 goroutine 了
            //在饥饿模式下,所有权是直接从执行 Unlock 的 goroutine
            //传递给队列中首个等待者的,也不需要再唤醒
        if old>>mutexWaiterShift == 0 || old&(mutexLocked|mutexWoken|mutexStarving) != 0 {
                return
            }
            //尝试设置 mutexWoken 标志,以获得唤醒一个 goroutine 的权力
            new = (old - 1<<mutexWaiterShift) | mutexWoken
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            runtime_Semrelease(&m.sema, false, 1)
            return
        }
        old = m.state
        }
    } else {
        //饥饿模式: 将 mutex 的所有权传递给下一个等待者
        //该等待者会继承当前 goroutine 的时间片并立刻开始运行
        //注意:mutexLocked 标志位没有设置,被唤醒的 goroutine 会设置它
        //因为饥饿模式下的 mutex 会被认为处于 Locked 状态
    //所以后来者不会尝试获取它
        runtime_Semrelease(&m.sema, true, 1)
    }
}
```

3）自旋

再来看一下与自旋相关的函数，首先是判断能否自旋的sync.runtime_canSpin（）函数，它实际上是个名字链接，真正调用的是runtime.sync_runtime_canSpin（）函数，代码如下：

```
func sync_runtime_canSpin(i int) bool {
    if i >= active_spin || ncpu <= 1 || gomaxprocs <= int32(sched.npidle + sched.
nmspinning) + 1 {
        return false
    }
    if p := getg().m.p.ptr(); !runqempty(p) {
        return false
    }
    return true
}
```

sync.Mutex是协作式的，在自旋方面比较保守。自旋的次数比较少，并且需要同时满足以下条件：在一个多核机器上运行并且GOMAXPROCS＞1，并且至少有一个其他的P正在运行，此外，当前P的本地runq是空的。

不像runtime.mutex那样，这里不会进行被动（消极）自旋，因为全局runq或者其他P上或许还有可运行的任务。

sync.runtime_doSpin（）函数也是通过linkname机制链接到runtime.sync_runtime_doSpin（）函数的，真正的逻辑是通过procyield（）函数实现30次自旋。

4）信号量相关操作

7.4.2节已经介绍过semaphore，这里只简单看一下调用关系。sync.runtime_Semacquire-Mutex（）函数是个名字链接，实际上调用的是runtime.sync_runtime_SemacquireMutex（）函数，后者又会调用runtime.semacquire1（）函数。semacquire1（）函数在7.4.2节已经分析过了，它实现了排队入列逻辑，通过lifo参数可以实现FIFO和LIFO，实际上就是插入队列尾部还是头部。

sync.runtime_Semrelease（）函数也是个名字链接，实际上调用的是runtime.sync_runtime_Semrelease（）函数，后者又会调用runtime.semrelease1（）函数。semrelease1（）函数实现了排队出列逻辑，通过handoff参数可以让被唤醒的goroutine继承当前时间片并立刻开始运行。

7.4.4 channel

channel被设计用于实现goroutine间的通信，按照golang的设计思想：以通信的方式共享内存。因为channel在设计上就已经解决了同步问题，所以程序逻辑只要保证数据的所有权随通信传递就可以了。本节就来分析一下channel实现的原理，先从内存布局开始。

1．channel内存布局

make（）函数会在堆上分配一个runtime.hchan类型的数据结构，示例代码如下：

```
ch := make(chan int)
```

ch是存在于函数栈帧上的一个指针，指向堆上的hchan数据结构。为什么是堆上的一个结构体？首先，要实现channel这样的复杂功能，肯定不是几字节可以实现的，所以需要一个struct实现；其次，这种被设计用于实现协程间通信的组件，其作用域和生命周期不可能仅限于某个函数内部，所以golang直接将其分配在堆上。

接下来就结合在channel中的作用，解读一下hchan中都有哪些字段。协程间通信肯定涉及并发访问，所以要有锁来保护整个数据结构，代码如下：

```
lock mutex
```

channel分为无缓冲和有缓冲两种，对于有缓冲channel来讲，需要有相应的内存来存储数据，实际上就是一个数组，需要知道数组的地址、容量、元素的大小，以及数组的长度，也就是已有元素的个数，这几个字段的代码如下：

```
qcount   uint           //数组长度,即已有元素的个数
dataqsiz uint           //数组容量,即可容纳元素的个数
buf      unsafe.Pointer //数组地址
elemsize uint16         //元素大小
```

因为runtime中内存复制、垃圾回收等机制依赖数据的类型信息，所以hchan中还要有一个指针，指向元素类型的类型元数据，代码如下：

```
elemtype * _type  //元素类型
```

channel支持交替地读写（比起发送和接收，笔者更喜欢称send为写，称recv为读），有缓冲channel内的缓冲数组会被作为一个环形缓冲区使用，当下标超过数组容量后会回到第1个位置，所以需要有两个字段记录当前读和写的下标位置，代码如下：

```
sendx    uint    //下一次写下标位置
recvx    uint    //下一次读下标位置
```

当读和写操作不能立即完成时，需要能够让当前协程在channel上等待，当条件满足时，要能够立即唤醒等待的协程，所以要有两个等待队列，分别针对读和写，代码如下：

```
recvq    waitq  //读等待队列
sendq    waitq  //写等待队列
```

channel是能够被关闭的，所以要有一个字段记录是否已经关闭了，代码如下：

```
closed  uint32
```

最后整合起来，runtime.hchan结构的代码如下：

```
type hchan struct {
    qcount   uint           //数组长度,即已有元素的个数
    dataqsiz uint           //数组容量,即可容纳元素的个数
    buf      unsafe.Pointer //数组地址
    elemsize uint16         //元素大小
    closed   uint32
    elemtype * _type        //元素类型
    sendx    uint           //下一次写下标位置
    recvx    uint           //下一次读下标位置
    recvq    waitq          //读等待队列
    sendq    waitq          //写等待队列
    lock     mutex
}
```

至此，我们已经了解了channel的主要数据结构，从各个字段的作用基本就能了解到channel内部大致是如何运作的。接下来还是结合源码，分析一下send、recv和select都是如何实现的。

2．channel的send操作

1）阻塞式send操作

首先来看一下channel的常规send操作。假如有一个元素类型为int的channel，变量名为ch，常规的send操作的代码如下：

```
ch <- 10
```

其中ch可能有缓冲，也可能无缓冲，甚至可能为nil。按照上面的写法，有两种情况能使send操作不会阻塞：

（1）通道ch的recvq里已有goroutine在等待。

（2）通道ch有缓冲，并且缓冲区没有用尽。

在第一种情况中，只要ch的recvq中有协程在排队，当前协程就直接把数据交给recvq队首的那个协程就好了，然后两个协程都可以继续执行，无关ch有没有缓冲。在第二种情况中，ch有缓冲，并且缓冲区没有用尽，也就是底层数组没有存满，此时当前协程直接把数据追加到缓冲数组中，就可以继续执行。

同样是上面的写法，有3种情况会使send操作阻塞：

（1）通道ch为nil。

（2）通道ch无缓冲且recvq为空。

（3）通道ch有缓冲且缓冲区已用尽。

在第一种情况中，参照目前的实现，允许对nil通道执行send操作，但是会使当前协程永久性地阻塞在这个nil通道上，因死锁抛出异常的示例代码如下：

```
func main() {
    var ch chan int
    ch <- 10
}
```

在第二种情况中，ch为无缓冲通道，recvq中没有协程在等待，所以当前协程需要到通道的sendq中排队。第三种情况中，ch有缓冲且已用尽，隐含的信息就是recvq为空，不会出现缓冲区不为空且recvq也不为空的情况，所以当前协程只能到sendq中排队。

2）非阻塞式send

接下来再看一看channel的非阻塞式send操作。熟悉并发编程的读者应该知道，有些锁支持tryLock操作，也就是我想获得这把锁，但是万一已经被别人获得了，我不阻塞等待，可以去做其他事情。对于channel的非阻塞send就是：我想通过channel发送数据，但是如果当前没有接收者在排队等待，并且缓冲区没有剩余空间（包含无缓冲的情况），我就需要阻塞等待，但是我不想等待，所以立刻返回并告诉我“现在不能发送”就可以了。

对于单个通道的非阻塞send操作可以用如下代码实现，注意是一个select、一个case和一个default，哪个都不能少，代码如下：

```
select {
case ch <- 10:
    ...
default:
    ...
}
```

如果检测到ch发送数据不会阻塞，就会执行case分支，如果会阻塞，就会执行default分支。

3）环形缓冲区

我们通过一个简单例子介绍一下channel的数据缓冲区是如何使用的，为什么称它为环形缓冲区。

假如有一个元素类型为int的channel，缓冲区大小为5，目前sendq和recvq为空，缓冲区还有一个元素的空闲位置，此时，读下标recvx及写下标sendx的位置如图7-14所示。

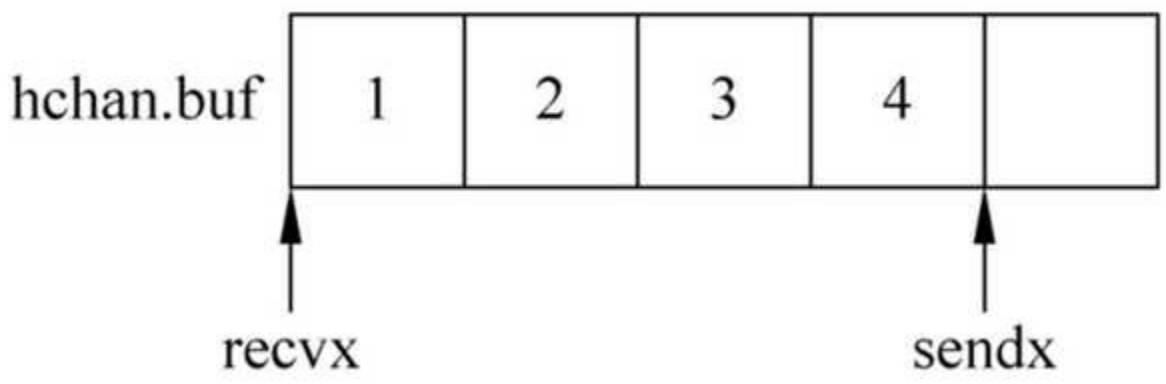

图7-14 示例channel读、写下标位置

接下来，有一个goroutine接收了一个元素，被读取的元素就是读下标所指向的第0个元素1，此时，channel缓冲区还有3个元素与两个空闲空间，读写下标位置如图7-15所示。

接下来，又有一个goroutine向这个channel发送了一个元素5，此时缓冲区的读写下标位置如图7-16所示。可以看到新的元素5被添加到最后一个空位处，但由于这是缓冲区最后一个位置，所以sendx回到了缓冲区头部，指向第0个位置。此时缓冲区还有4个元素与一个空闲位置。

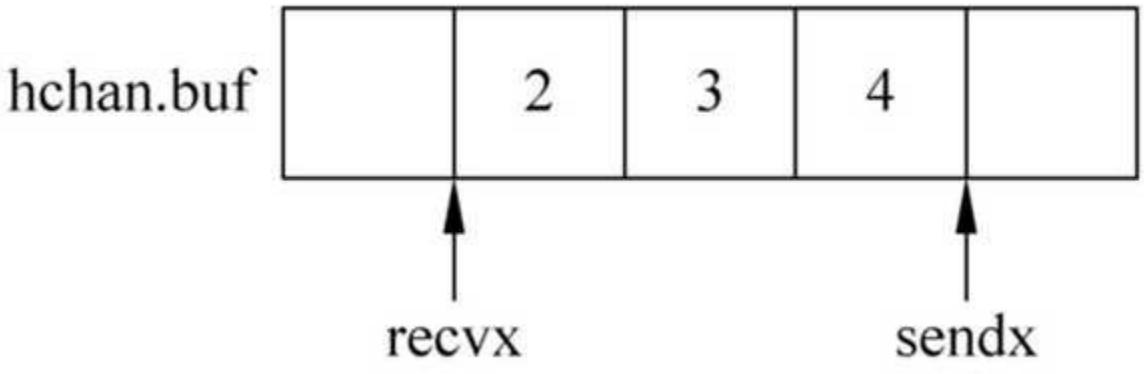

图7-15 读取一个元素后读、写下标位置

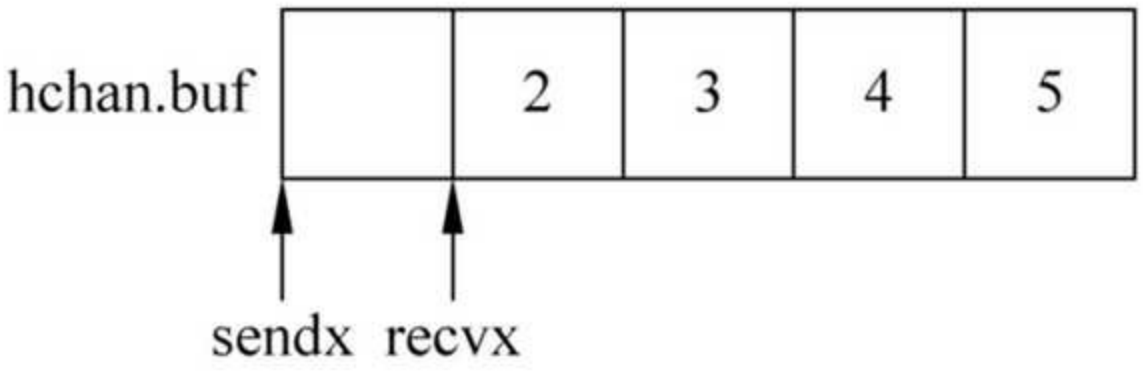

图7-16 发送一个元素后读、写下标位置

下面又有两个元素6和7发送到这个channel，元素6会占用此时sendx指向的第0个位置，此时，读、写下标相等，没有空闲位置了，表明缓冲区已满，发送元素7的goroutine只能进到sendq中排队等待，如图7-17所示。

此时排队等待要发送元素7的goroutine，只有等到有goroutine从这个channel读取数据后腾出空闲缓冲区位置，才能完成数据发送。例如接下来读取一个元素，recvx向后移动一个位置，元素7被存到空出的位置，sendq再次为空，缓冲区依然是满的，如图7-18所示。

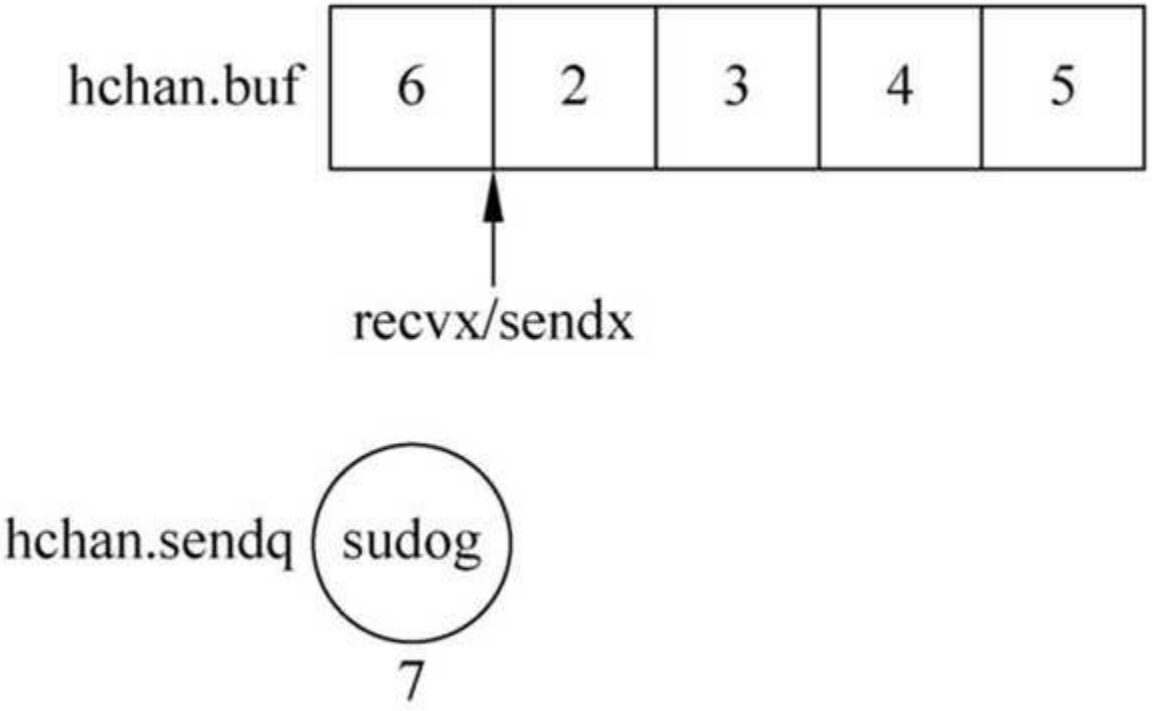

图7-17 又发送两个元素后缓冲区与sendq的状态

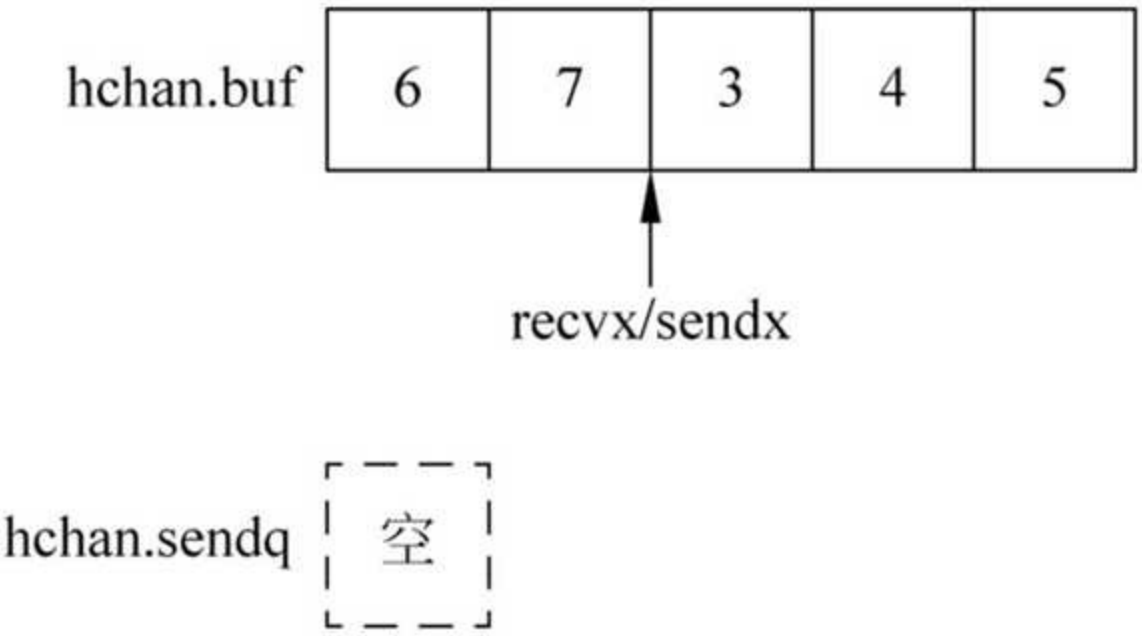

图7-18 读取一个元素后读、写下标和sendq的状态

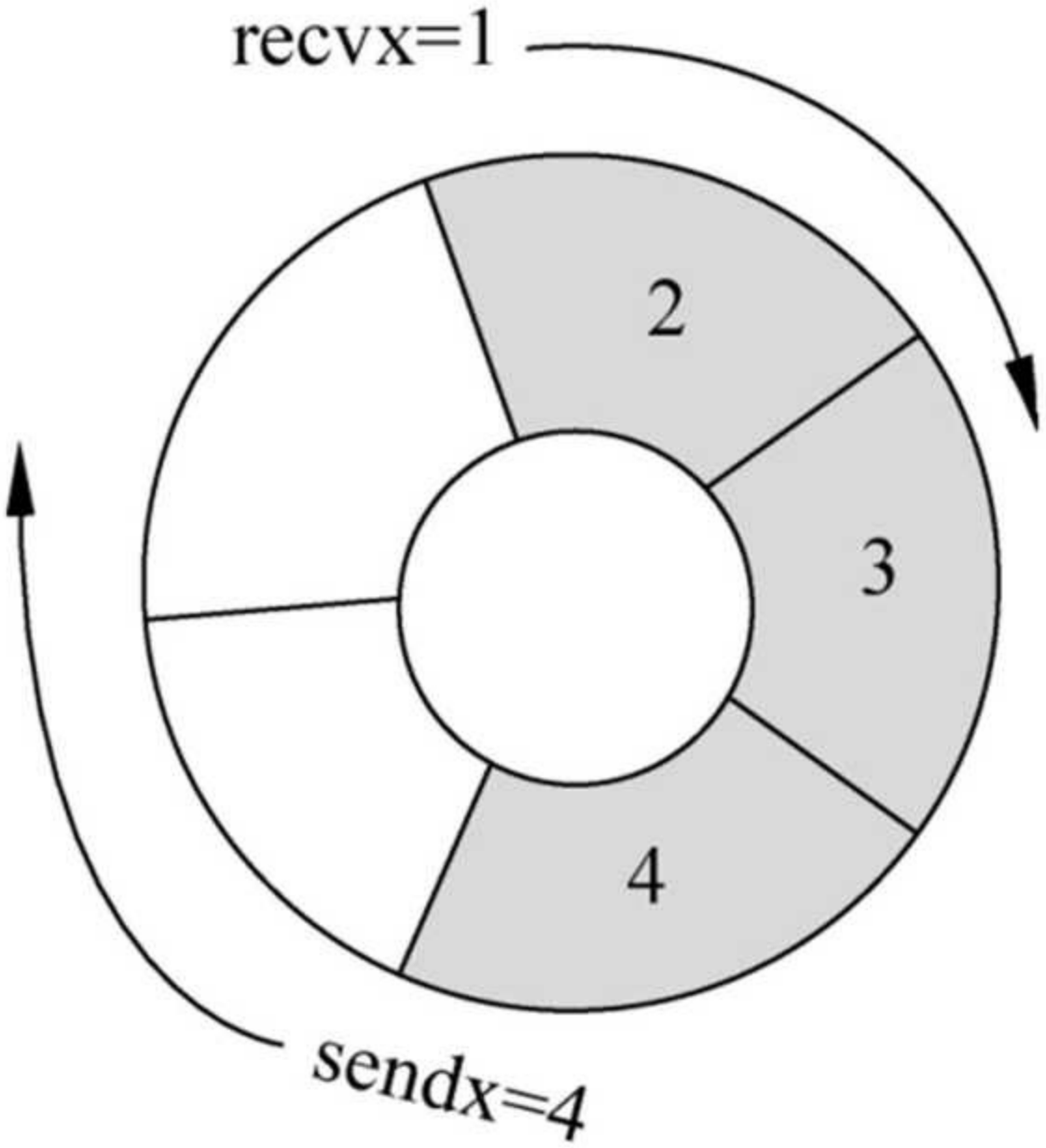

图7-19 环形缓冲区示意图

可以看到，这个channel缓冲区的读写下标都是从0到4再到0这样循环变化的，这就好像在使用一个环形缓冲区一样，例如图7-15所示的缓冲区对应图7-19所示的环形缓冲区，灰色区域代表已使用缓冲区，空白区域代表未使用缓冲区。

3．**send**操作的源码分析

channel的常规send操作会被编译器转换为对runtime.chansend1（）函数的调用，后者内部只是调用了runtime.chansend（）函数。非阻塞式的send操作会被编译器转换为对runtime.selectnbsend（）函数的调用，后者也仅仅调用了runtime.chansend（）函数，所以send操作主要通过chansend（）函数实现，接下来我们就来分析一下这个函数的源码。chansend（）函数的原型如下：

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool
```

其中，c是一个hchan指针，指向要用来send数据的channel。ep是一个指针，指向要被送入通道c的数据，数据类型要和c的元素类型一致。block表示如果send操作不能立即完成，是否想要阻塞等待。callerpc用以进行race相关检测，暂时不需要关心。返回值为true表示数据send完成，false表示目前不能发送，但因为不想阻塞（block为false）而返回。

这个函数的逻辑还算比较直观，接下来就分块梳理一下。以下省略掉了部分不太重要的代码，摘选主要逻辑，第一部分代码如下：

```
if c == nil {
    if !block {
        return false
    }
    gopark(nil, nil, waitReasonChanSendNilChan, traceEvGoStop, 2)
    throw("unreachable")
}
```

如果c为nil，进一步判断block：如果block为false，则直接返回false，表示未发送数据。如果block为true，就让当前协程永久地阻塞在这个nil通道上。

第二部分代码如下：

```
if !block && c.closed == 0 && full(c) {
    return false
}
```

如果block为false且closed为0，也就是在不想阻塞且通道未关闭的前提下，如果通道满了（无缓冲且recvq为空，或者有缓冲且缓冲已用尽），则直接返回false。本步判断是在不加锁的情况下进行的，目的是让非阻塞send在无法立即完成时能真正不阻塞（加锁操作可能阻塞）。

第三部分代码如下：

```
lock(&c.lock)
if c.closed != 0 {
    unlock(&c.lock)
    panic(plainError("send on closed channel"))
}
```

对hchan加锁，如果closed不为0，即通道已经关闭，则先解锁，然后panic。因为不允许用已关闭的通道进行send。

第四部分代码如下：

```
if sg := c.recvq.dequeue(); sg != nil {
    send(c, sg, ep, func() { unlock(&c.lock) }, 3)
    return true
}
```

如果recvq不为空，隐含了缓冲区为空，就从中取出第1个排队的协程，将数据传递给这个协程，并将该协程置为ready状态（放入run queue，进而得到调度），然后解锁，返回值为true。

第五部分代码如下：

```
if c.qcount < c.dataqsiz {
    qp := chanbuf(c, c.sendx)
    typedmemmove(c.elemtype, qp, ep)
    c.sendx++
    if c.sendx == c.dataqsiz {
        c.sendx = 0
    }
    c.qcount++
    unlock(&c.lock)
    return true
}
```

通过比较qcount和dataqsiz判断缓冲区是否还有剩余空间，在这里无缓冲的通道被视为没有剩余空间。如果有剩余空间，就将数据追加到缓冲区中，相应地移动sendx，增加qcount，然后解锁，返回值为true。

第六部分代码如下：

```
if !block {
    unlock(&c.lock)
    return false
}
```

运行到这里表明通道已满，如果block为false，即不想阻塞，则解锁，返回值为false。

第七部分代码如下：

```
gp := getg()
mysg := acquireSudog()
mysg.elem = ep
mysg.g = gp
mysg.isSelect = false
mysg.c = c
gp.waiting = mysg
c.sendq.enqueue(mysg)
atomic.Store8(&gp.parkingOnChan, 1)
gopark(chanparkcommit, unsafe.Pointer(&c.lock), waitReasonChanSend, traceEvGoBlockSend, 2)
```

当前协程把自己追加到通道的sendq中阻塞排队，gopark（）函数挂起协程后会调用chanparkcommit（）函数对通道解锁，等到有接收者接收数据后，阻塞的协程会被唤醒。chansend（）函数在向recvq中的协程发送数据时，调用了send（）函数，send（）函数的主要代码如下：

```
func send(c * hchan, sg * sudog, ep unsafe.Pointer, unlockf func(), skip int) {
    if sg.elem != nil {
        sendDirect(c.elemtype, sg, ep)
        sg.elem = nil
    }
    gp := sg.g
    unlockf()
    gp.param = unsafe.Pointer(sg)
    sg.success = true
    goready(gp, skip + 1)
}
```

其中，数据传递工作是通过sendDirect（）函数完成的，然后调用unlockf（）函数会把hchan解锁，最后通过goready（）函数唤醒接收者协程。因为发送数据会访问接收者协程的栈，所以sendDirect（）函数用到了写屏障，函数的代码如下：

```
func sendDirect(t * _type, sg * sudog, src unsafe.Pointer) {
    dst := sg.elem
    typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
    memmove(dst, src, t.size)
}
```

至此，channel的send操作就基本告一段落了，接下来我们再来看一看recv操作。

4．channel的recv操作

1）阻塞式recv

先来看一下channel的常规recv操作。假如有一个元素类型为int的channel，变量名为ch，常规的recv操作的代码如下：

```
//将结果丢弃
<-ch
//将结果赋值给变量 v
v := <-ch
//comma ok style,ok 为 false 表示 ch 已关闭且 v 是零值
v, ok := <-ch
```

其中ch可能有缓冲，也可能无缓冲，甚至可能为nil。按照上面的写法，有两种情况能使recv操作不会阻塞：

（1）通道ch的sendq里已有goroutine在等待。

（2）通道ch的sendq是空的，但是通道有缓冲且缓冲区中有数据。

在第一种情况中，只要ch的sendq中有协程在排队，就需要进一步判断通道是否有缓冲：如果无缓冲，当前协程就直接从sendq队首的那个协程获取数据，然后两者都可以继续执行。如果有缓冲，隐含信息就是缓冲区已满，否则sendq中不会有协程排队，这时当前协程从缓冲区取出第1个数据（缓冲区有了一个空闲位置），然后从sendq中取出第1个协程，把它的数据追加到缓冲区中，并把它置成ready状态，最终两个协程都能继续执行了。

在第二种情况中，ch的sendq中没有协程在排队，所以不需要关心。如果ch有缓冲，并且缓冲区有数据，则当前协程直接从缓冲区取出第1个数据，然后就可以继续执行了。

同样是上面的写法，有3种情况会使recv操作阻塞：

（1）通道ch为nil。

（2）通道ch无缓冲且sendq为空。

（3）通道ch有缓冲且缓冲区无数据。

在第一种情况中，参照目前的实现，允许对nil通道执行recv操作，但是会使当前协程永久性地阻塞在这个nil通道上，因死锁抛出异常的示例代码如下：

```
func main() {
    var ch chan int
    <- ch
}
```

在第二种情况中，ch为无缓冲通道，sendq中没有协程在等待，所以当前协程需要到通道的recvq中排队。在第三种情况中，ch有缓冲但是没有数据，隐含的信息是sendq为空，否则缓冲区不可能没有数据，所以当前协程只能到recvq中排队。

2）非阻塞式recv

再来看一下channel的非阻塞recv操作。还是类似于tryLock操作，我想获得这把锁，但是万一已经被别人获得了，我不阻塞等待，可以去做其他事情。对于通道的非阻塞recv就是：我想从通道接收数据，但是当前没有发送者在排队等待，并且缓冲区内无数据（包含无缓冲），我需要阻塞等待，但是我不想等待，所以立刻返回并告诉我“现在无数据”就可以了。

对于单个通道的非阻塞recv操作可以用如下代码实现，注意是一个select、一个case和一个default，哪个都不能少，代码如下：

```
select {
case <- ch: //此处可以带有赋值操作,或者 comma ok style
    ...
default:
    ...
}
```

如果检测到ch recv不会阻塞，就会执行case分支，如果会阻塞，就会执行default分支。

事实上，channel的常规recv操作会被编译器转换为对runtime.chanrecv1（）函数的调用，后者内部只是调用了runtime.chanrecv（）函数。comma ok写法会被编译器转换为对runtime.chanrecv2（）函数的调用，内部也是调用chanrecv（）函数，只不过比chanrecv1（）函数多了一个返回值。非阻塞式的recv操作会被编译器转换为对runtime.selectnbrecv（）函数或selectnbrecv2（）函数的调用（根据是否comma ok），后两者也仅仅调用了runtime.chanrecv（）函数，所以recv操作主要通过chanrecv（）函数实现，接下来我们就来分析一下这个函数的源码。

5．recv操作的源码分析

上面简单地分析了channel的常规recv操作和非阻塞recv操作，虽然两者在形式上看起来稍微有些差异，但是主要逻辑都是通过runtime.chanrecv（）函数实现的，下面简单地进行一下解读。
chanrecv（）函数的原型如下：

```
func chanrecv(c * hchan, ep unsafe.Pointer, block bool) (selected, received bool)
```

其中，c是一个hchan指针，指向要从中recv数据的channel。ep是一个指针，指向用来接收数据的内存，数据类型要和c的元素类型一致。block表示如果recv操作不能立即完成，是否想要阻塞等待。

selected为true表示操作完成（可能因为通道已关闭），false表示目前不能立刻完成recv，但因为不想阻塞（block为false）而返回。received为true表示数据确实是从通道中接收的，不是因为通道关闭而得到的零值，为false的情况需要结合selected来解释，可能是因为通道关闭而得到零值（selected为true），或者因为不想阻塞而返回（selected为false）。

chanrecv（）函数的大致逻辑与chansend（）函数的大致逻辑很相似，接下来还是省略不太重要的代码，对函数的主要逻辑分段进行梳理。

第一部分代码如下：

```
if c == nil {
    if !block {
        return
    }
    gopark(nil, nil, waitReasonChanReceiveNilChan, traceEvGoStop, 2)

    throw("unreachable")
}
```

如果c为nil，进一步判断block：如果block为false，就直接返回两个false，表示未recv数据。如果block为true，就让当前协程永久地阻塞在这个nil通道上。

第二部分代码如下：

```
if !block && empty(c) {
    if atomic.Load(&c.closed) == 0 {
        return
    }
    if empty(c) {
        if ep != nil {
            typedmemclr(c.elemtype, ep)
        }
        return true, false
    }
}
```

如果block为false，也就是在不想阻塞的前提下，并且通道是空的（无缓冲且sendq为空，或者通道有缓冲且缓冲区为空），就再判断通道是否已关闭。如果未关闭，则直接返回两个false，表示因不想阻塞而返回。已关闭就先把ep清空，然后返回true和false，表明因通道关闭而得到零值。本步判断是在不加锁的情况下进行的，目的是让非阻塞recv在无法立即完成时能真正不阻塞（加锁可能阻塞）。是否为空和是否已关闭这两个判断顺序不能打乱，要在后面判断通道是否关闭。因为关闭后的通道不能再被打开，这样保证了并发条件下的一致性。如果把判断closed前置，则在检查缓冲区和sendq时通道可能已关闭，这样会出现错误。

第三部分代码如下：

```
//加锁
lock(&c.lock)

if c.closed != 0 && c.qcount == 0 {
    unlock(&c.lock)
    if ep != nil {
        typedmemclr(c.elemtype, ep)
    }
    return true, false
}
```

如果closed不为0，即通道已经关闭，则解锁，然后给ep赋零值，返回值为true和false。

第四部分代码如下：

```
if sg := c.sendq.dequeue(); sg != nil {
    recv(c, sg, ep, func() { unlock(&c.lock) }, 3)
    return true, true
}
```

如果sendq不为空，就从中取出第1个排队的协程sg。如果有缓冲，则还需要滚动缓冲区，完成数据读取，并将协程sg置为ready状态（放入run queue，进而得到调度），然后解锁，这些工作都由recv（）函数完成。最后返回两个true。

第五部分代码如下：

```
if c.qcount > 0 {
    qp := chanbuf(c, c.recvx)
    if ep != nil {
        typedmemmove(c.elemtype, ep, qp)
    }
    typedmemclr(c.elemtype, qp)
    c.recvx++
    if c.recvx == c.dataqsiz {
        c.recvx = 0
    }
    c.qcount--
    unlock(&c.lock)
    return true, true
}
```

通过qcount判断缓冲区是否有数据，在这里无缓冲的通道被视为没有数据，因为到达这一步sendq一定为空。如果缓冲区有数据，将第1个数据取出并赋给ep，移动recvx，递减qcount，解锁，返回两个true。

第六部分代码如下：

```
if !block {
    unlock(&c.lock)
    return false, false
}
```

运行到这里就说明sendq和缓冲区都为空，如果block为false，也就是不想阻塞，则解锁，返回两个false。

第七部分代码如下：

```
gp := getg()
mysg := acquireSudog()
mysg.elem = ep
gp.waiting = mysg
mysg.g = gp
mysg.isSelect = false
mysg.c = c
c.recvq.enqueue(mysg)
atomic.Store8(&gp.parkingOnChan, 1)
gopark(chanparkcommit, unsafe.Pointer(&c.lock), waitReasonChanReceive, traceEvGoBlockRecv,
2)
```

最后，运行到这里就要阻塞了，当前协程把自己追加到通道的recvq中阻塞排队，gopark（）函数会在挂起当前协程后调用chanparkcommit（）函数解锁，等到后续recv操作完成时协程会被唤醒。

第八部分代码如下：

```
success := mysg.success
releaseSudog(mysg)
return true, success
```

被唤醒有可能是因为通道被关闭，所以最后的返回值received需要根据被唤醒的原因来判断，若是因为等到真实数据，则为true，若是因为通道关闭，则为false。chanrecv（）函数在从sendq中的协程接收数据时，调用了recv（）函数，recv（）函数的主要代码如下：

```
func recv(c * hchan, sg * sudog, ep unsafe.Pointer, unlockf func(), skip int) {
    if c.dataqsiz == 0 {
        if ep != nil {
            recvDirect(c.elemtype, sg, ep)
        }
    } else {
        qp := chanbuf(c, c.recvx)
        if ep != nil {
            typedmemmove(c.elemtype, ep, qp)
        }
        typedmemmove(c.elemtype, qp, sg.elem)
        c.recvx++
    if c.recvx == c.dataqsiz {
        c.recvx = 0
        }
        c.sendx = c.recvx //c.sendx = (c.sendx + 1) % c.dataqsiz
    }
    sg.elem = nil
    gp := sg.g
    unlockf()
    gp.param = unsafe.Pointer(sg)
    sg.success = true
    goready(gp, skip + 1)
}
```

如果是无缓冲通道，则直接通过recvDirect（）函数进行数据复制。若有缓冲，则同时隐含了缓冲区已满，这样sendq才会不为空。此时需要对缓冲区进行滚动，把缓冲区头部的数据取出来并接收，然后把sendq头部协程要发送的数据追加到缓冲区尾部。最后，通过goready（）函数唤醒发送者协程就可以了。

recvDirect（）函数和sendDirect（）函数类似，因为要访问其他协程的栈，所以在应用写屏障后进行数据复制，代码如下：

```
func recvDirect(t * _type, sg * sudog, dst unsafe.Pointer) {
    src := sg.elem
    typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
    memmove(dst, src, t.size)
}
```

关于channel的recv操作就先探索到这里，建议有兴趣的读者好好阅读一下源码。

6．channel之多路select

本节第2部分和第4部分在介绍channel的非阻塞式send和非阻塞式recv时提到过select，但是那只是针对单个通道的操作。不同的写法对应着不同的底层实现，接下来我们就简单地介绍一下多路select的用法，以及其底层的实现原理。

多路select指的是存在两个及以上的case分支，每个分支可以是一个channel的send或recv操作。例如ch1和ch2是两个元素类型为int的channel，示例代码如下：

```
//第 7 章/code_7_8.go
select {
case v := <- ch1:
    println(v)
case ch2 <- 10:
default:
}
```

其中default分支是可选的，上述代码会被编译器转换成对runtime.selectgo（）函数的调用，该函数的原型如下：

```
func selectgo(cas0 *scase, order0 *uint16, pc0 *uintptr, nsends, nrecvs int, block bool)
(int, bool)
```

cas0指向一个数组，数组里装的是select中所有的case分支，按照send在前recv在后的顺序。

order0指向一个大小等于case分支数量两倍的uint16数组，实际上是作为两个大小相等的数组来用的。前一个用来对所有case中channel的轮询操作进行乱序，后一个用来对所有case中channel的加锁操作进行排序。轮询操作需要是乱序的，避免每次select都按照case的顺序响应，对后面的case来讲是不公平的，而加锁顺序需要按照固定算法排序，按顺序加锁才能避免死锁。

pc0和race检测相关，这里暂时不用关心。nsends和nrecvs分别表示在cas0数组中执行send操作和recv操作的case分支的个数。

block表示是否想要阻塞等待，对应到代码中就是，有default分支的不阻塞，反之则会阻塞。

下面来看两个返回值，int型的第1个返回值表示最终哪个case分支被执行了，对应cas0数组的下标。如果因为不想阻塞而返回，则这个值是-1。bool类型的第2个返回值在对应的case分支执行的是recv操作时，用来表示实际接收到了一个值，而不是因为通道关闭得到的零值。

selectgo（）函数的逻辑比之chansend（）函数和chanrecv（）函数的逻辑要复杂一些，但是原理上是相通的。例如第7章/code_7_8.go中，一个协程通过多路select等待ch1和ch2，我们暂且把这个协程记为g1。

g1执行这个多路select时，会先按照有序的加锁顺序对所有channel加锁，然后按照乱序的轮询顺序检查所有channel的sendq或recvq，以及缓冲区。当检查到ch1或ch2时，如果发现它的等待队列或缓冲区不为空，就直接复制数据，进入对应分支。

假如所有channel的操作都不能立即完成，就把当前协程g1添加到所有channel的sendq或recvq中，所以g1被添加到ch1的recvq，也被添加到ch2的sendq中，如图7-20所示，然后就会调用gopark（）函数把自己挂起，工作线程挂起当前协程后会调用selparkcommit（）函数解锁所有channel。

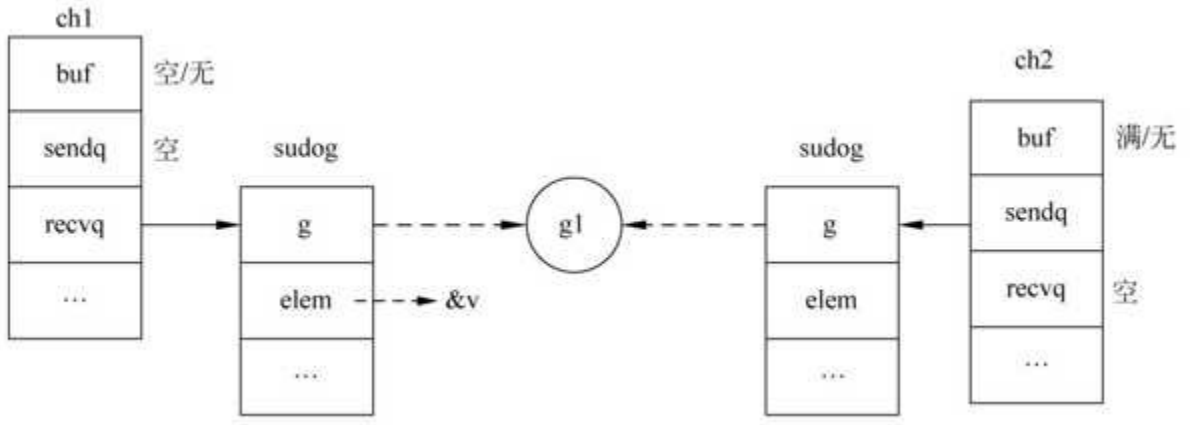

图7-20 g1在等待队列中等待ch1和ch2

假如接下来ch1有数据可读了，g1被唤醒，完成从ch1中recv数据后，会再次按照加锁顺序对所有channel加锁，然后从所有sendq或recvq中将自己移除，最后全部解锁后返回。

selectgo（）函数的代码占用篇幅较大，但主要逻辑还算比较清晰，这里不再逐段进行分析，感兴趣的读者可自行阅读源码。

7.5 本章小结

本章中，我们从单核环境的并发到多核环境的并行，以及编译阶段与执行阶段的内存乱序，逐个讲解了同步面临的问题。后面又介绍了编译屏障、内存排序指令等解决方案。重点讲解了原子指令及自旋锁的实现，因为它们是其他各种锁的基础。最后一节中，从源码层面分析了Go语言中几个关键的同步组件：runtime.mutex、semaphore、sync.Mutex及channel，希望各位读者有所收获。

第8章
堆

进程的堆内存通常指的是地址空间中区别于代码区和全局数据区的另一个内存区，允许程序在运行阶段动态地申请所需的内存空间。很多编程语言的runtime实现了自己的堆，例如C语言中为大家所熟知的malloc（）函数和free（）函数，一方面包揽了向操作系统申请内存页面及内存空间的管理等工作，另一方面为开发者提供了简单易用的API，使开发者不需要关心底层的细节，只是按需调用API分配和释放内存就可以了。Go语言的堆内存管理和C语言有一点明显的不同，就是当一段内存不再使用的时候，不需要开发者手动进行释放，而是由垃圾回收器GC来自动完成。本章从内存分配和垃圾回收两方面来看一下Go语言的堆内存管理。

8.1 内存分配

在使用其他编程语言的时候，堆内存分配通常是显式的，例如C语言中的malloc（）函数，以及C++中的new关键字等，基本上在它们出现的地方就意味着堆分配。在Go语言中，我们通常可能会认为出现new（）函数和make（）函数这两个内置函数的地方就是堆分配，实则不然。编译器会基于逃逸分析对内存的分配进行优化，有些没有逃逸的变量，即使源代码层面是通过new（）函数或make（）函数分配的，也不会在堆上分配，那些被认为逃逸的变量，即使没有用到new（）函数和make（）函数，也会在堆上分配。

在Go的runtime中，有一系列函数被用来分配内存。例如与new语义相对应的有newobject（）函数和newarray（）函数，分别负责单个对象的分配和数组的分配。与make语义相对应的有makeslice（）函数、makemap（）函数及makechan（）函数及一些变种，分别负责分配和初始化切片、map和channel。无论是new系列还是make系列，这些函数的内部无一例外都会调用runtime.mallocgc（）函数，它就是Go语言堆分配的关键函数。在开始分析mallocgc（）函数之前，我们需要先了解一些铺垫知识，例如一些关键的常量、数据结构和底层函数之类的。下面我们就先来了解这些基础内容，本节的最后再回过头来分析mallocgc（）函数。

8.1.1 sizeclasses

Go的堆分配采用了与tcmalloc内存分配器类似的算法，tcmalloc是谷歌公司开发的一款针对C/C++的内存分配器，在对抗内存碎片化和多核性能方面非常优秀，因此有着很广泛的应用。其他一些编程语言中也有类似tcmalloc的实现，例如PHP 7参考了tcmalloc的思想对堆分配进行了优化，得到了显著的性能提升。

参考tcmalloc实现的内存分配器，内部针对小块内存的分配进行了优化。这类分配器会按照一组预置的大小规格把内存页划分成块，然后把不同规格的内存块放入对应的空闲链表中，如图8-1所示。这些内存块通常有8字节、16字节、24字节、32字节、48字节，直到数十或数百KB，总共几十种大小规格。为了提高内存的利用率，这些规格大小并不都是2的整数次幂。程序申请内存的时候分配器会先根据要申请的空间大小找到最匹配的规格，然后从对应的空闲链表中分配一个内存块。

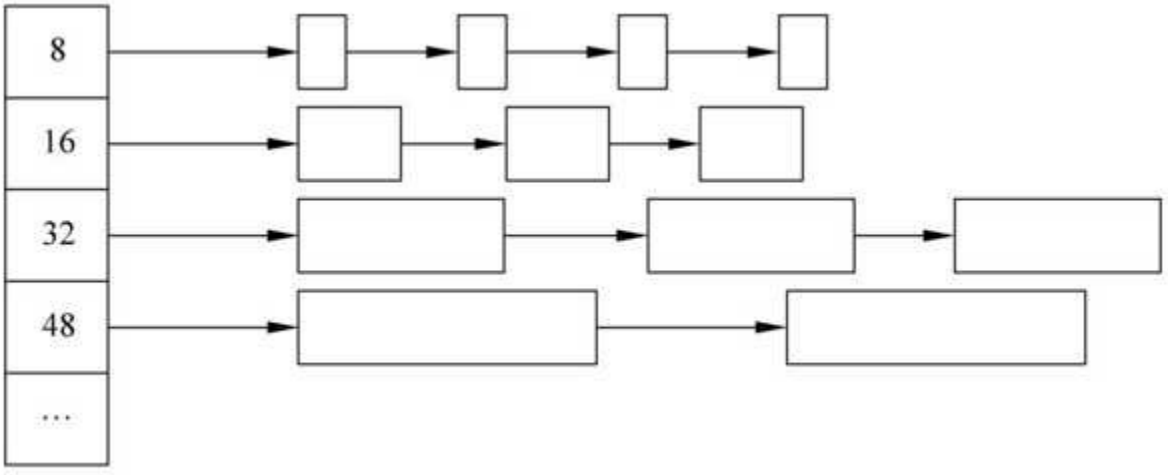

图8-1 tcmalloc内存分配器预置不同规格的链表

假如想要分配一段20字节大小的内存，分配器会认为所有预置的规格中24字节这个大小最为匹配，因此最终会实际分配一个大小为24字节的内存块。虽然不可避免地存在一定的空间浪费，但是解决了内存碎片化问题，还带来了一定程度上的性能提升。这些预置规格大小的选择，结合编程语言自身的特点，能够进一步提高内存空间的利用率。

在Go源代码runtime包的sizeclasses.go文件中，给出了一组预置的大小规格。在runtime版本1.8～1.15期间，一直是66种规格，其中最小的是8字节，最大的是32KB。Go 1.16版本新增了24字节大小这个规格，总共达到67种，如表8-1所示。

表8-1 sizeclasses预置的大小规格

class	Bytes/obj	B/span	objects	tail waste	max waste/%
1	8	8192	1024	0	87.50
2	16	8192	512	0	43.75
3	24	8192	341	8	29.24
4	32	8192	256	0	21.88
5	48	8192	170	32	31.52
6	64	8192	128	0	23.44
7	80	8192	102	32	19.07
8	96	8192	85	32	15.95
9	112	8192	73	16	13.56
10	128	8192	64	0	11.72
11	144	8192	56	128	11.82
12	160	8192	51	32	9.73
13	176	8192	46	96	9.59
14	192	8192	42	128	9.25
15	208	8192	39	80	8.12
16	224	8192	36	128	8.15
17	240	8192	34	32	6.62
18	256	8192	32	0	5.86
19	288	8192	28	128	12.16
20	320	8192	25	192	11.80
21	352	8192	23	96	9.88
22	384	8192	21	128	9.51
23	416	8192	19	288	10.71
24	448	8192	18	128	8.37
25	480	8192	17	32	6.82
26	512	8192	16	0	6.05
27	576	8192	14	128	12.33
28	640	8192	12	512	15.48
29	704	8192	11	448	13.93
30	768	8192	10	512	13.94
31	896	8192	9	128	15.52
32	1024	8192	8	0	12.40
33	1152	8192	7	128	12.41
34	1280	8192	6	512	15.55
35	1408	16384	11	896	14.00
36	1536	8192	5	512	14.00
37	1792	16384	9	256	15.57
38	2048	8192	4	0	12.45
39	2304	16384	7	256	12.46
40	2688	8192	3	128	15.59
41	3072	24576	8	0	12.47
42	3200	16384	5	384	6.22
43	3456	24576	7	384	8.83
44	4096	8192	2	0	15.60

续表

class	Bytes/obj	B/span	objects	tail waste	max waste/%
45	4864	24576	5	256	16.65
46	5376	16384	3	256	10.92
47	6144	24576	4	0	12.48
48	6528	32768	5	128	6.23
49	6784	40960	6	256	4.36
50	6912	49152	7	768	3.37
51	8192	8192	1	0	15.61
52	9472	57344	6	512	14.28
53	9728	49152	5	512	3.64
54	10240	40960	4	0	4.99
55	10880	32768	3	128	6.24
56	12288	24576	2	0	11.45
57	13568	40960	3	256	9.99
58	14336	57344	4	0	5.35
59	16384	16384	1	0	12.49
60	18432	73728	4	0	11.11
61	19072	57344	3	128	3.57
62	20480	40960	2	0	6.87
63	21760	65536	3	256	6.25
64	24576	24576	1	0	11.45
65	27264	81920	3	128	10.00
66	28672	57344	2	0	4.91
67	32768	32768	1	0	12.50

第一列是所谓的sizeclass，实际上就是所有规格按空间大小升序排列的序号。第二列是规格的空间大小，单位是字节。第三列表示需要申请多少字节的连续内存，目的是保证划分成目标大小的内存块以后，尾端因不能整除而剩余的空间要小于12.5%。Go使用8192字节作为页面大小，底层内存分配的时候都是以整页面为单位的，所以第三列都是8192的整数倍。第四列是第三列与第二列做整数除法得到的商，第五列则是余数，分别表示申请的连续内存能划分成多少个目标大小的内存块，以及尾端因不能整除而剩余的空间，也就是在内存块划分的过程中浪费掉的空间。最后一列就有点意思了，表示的是最大浪费百分比，结合了内存块划分时造成的尾端浪费和内存分配时向上对齐到最接近的块大小造成的块内浪费。

对于最大浪费百分比这一列，我们举两个例子计算并验证一下。先以大小为8字节的内存块为例，申请一个页也就是8192字节内存，可以划分成1024个块，因为没有余数，所以不存在尾端浪费。等到分配内存时，浪费最严重的情况是想要分配1字节时，向上对齐到8字节会浪费掉7/8，也就是87.5%。再来看一个块划分时不能整除的情况，例如大小为1408的内存块，申请两个页面也就是16384字节的内存，划分成11个内存块后剩余896字节。分配某个大小的内存时，1281～1408字节这个范围会被向上对齐到1408字节这个内存块大小，其中1281字节是浪费最严重的情况。假如划分的11个内存块实际上都用作1281字节大小的分配，加上尾端的896字节，最大浪费百分比＝（（1408-1281）×11+896）/16384，约等于14%。

关于sizeclasses就先介绍到这里，事实上，Go语言runtime中的sizeclasses.go文件是被程序生成出来的，源码就在mksizeclasses.go文件中，感兴趣的读者可以从源码中了解更多细节。

8.1.2 heapArena

Go语言的runtime将堆地址空间划分成多个arena，在amd64架构的Linux环境下，每个arena的大小是64MB，起始地址也是对齐到64MB的。每个arena都有一个与之对应的heapArena结构，用来存储

arena的元数据，如图8-2所示。

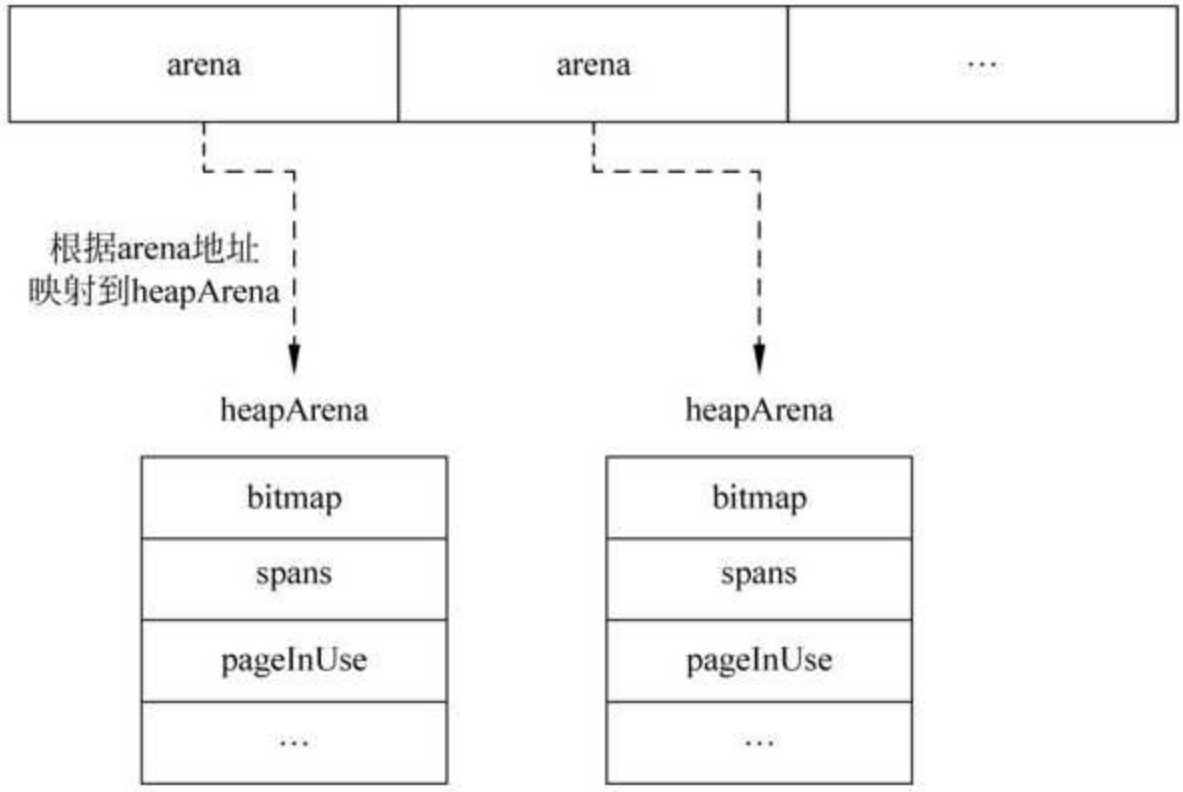

图8-2 area与heapArena的关系

heapArena是在Go的堆之外分配和管理的，其结构定义的代码如下：

```
type heapArena struct {
    bitmap       [heapArenaBitmapBytes]byte
    spans        [pagesPerArena] * mspan
    pageInUse    [pagesPerArena / 8]uint8
    pageMarks    [pagesPerArena / 8]uint8
    pageSpecials [pagesPerArena / 8]uint8
    checkmarks   * checkmarksMap
    zeroedBase   uintptr
}
```

bitmap字段是个位图，它用两个二进制位来对应arena中一个指针大小的内存单元，所以对于64MB大小的arena来讲，heapArenaBitmapBytes的值是64MB/8/8×2＝2MB，这个位图在GC扫描阶段会被用到。bitmap第一字节中的8个二进制位，对应的就是arena起始地址往后32字节的内存空间。用来描述一个内存单元的两个二进制位当中，低位用来区分内存单元中存储的是指针还是标量，1表示指针，0表示标量，所以也被称为指针/标量位。高位用来表示当前分配的这块内存空间的后续单元中是否包含指针，例如在堆上分配了一个结构体，可以知道后续字段中是否包含指针，如果没有指针就不需要继续扫描了，所以也被称为扫描/终止位。为了便于操作，一个位图字节中的指针/标量位和扫描/终止位被分开存储，高4位存储4个扫描/终止位，低4位存储4个指针/标量位。

例如在arena起始处分配一个slice，slice结构包括一个元素指针、一个长度及一个容量，对应的bitmap标记如图8-3所示。bitmap位图第一字节第0～2位标记slice 3个字段是指针还是标量，第4～6位标记3个字段是否需要继续扫描。

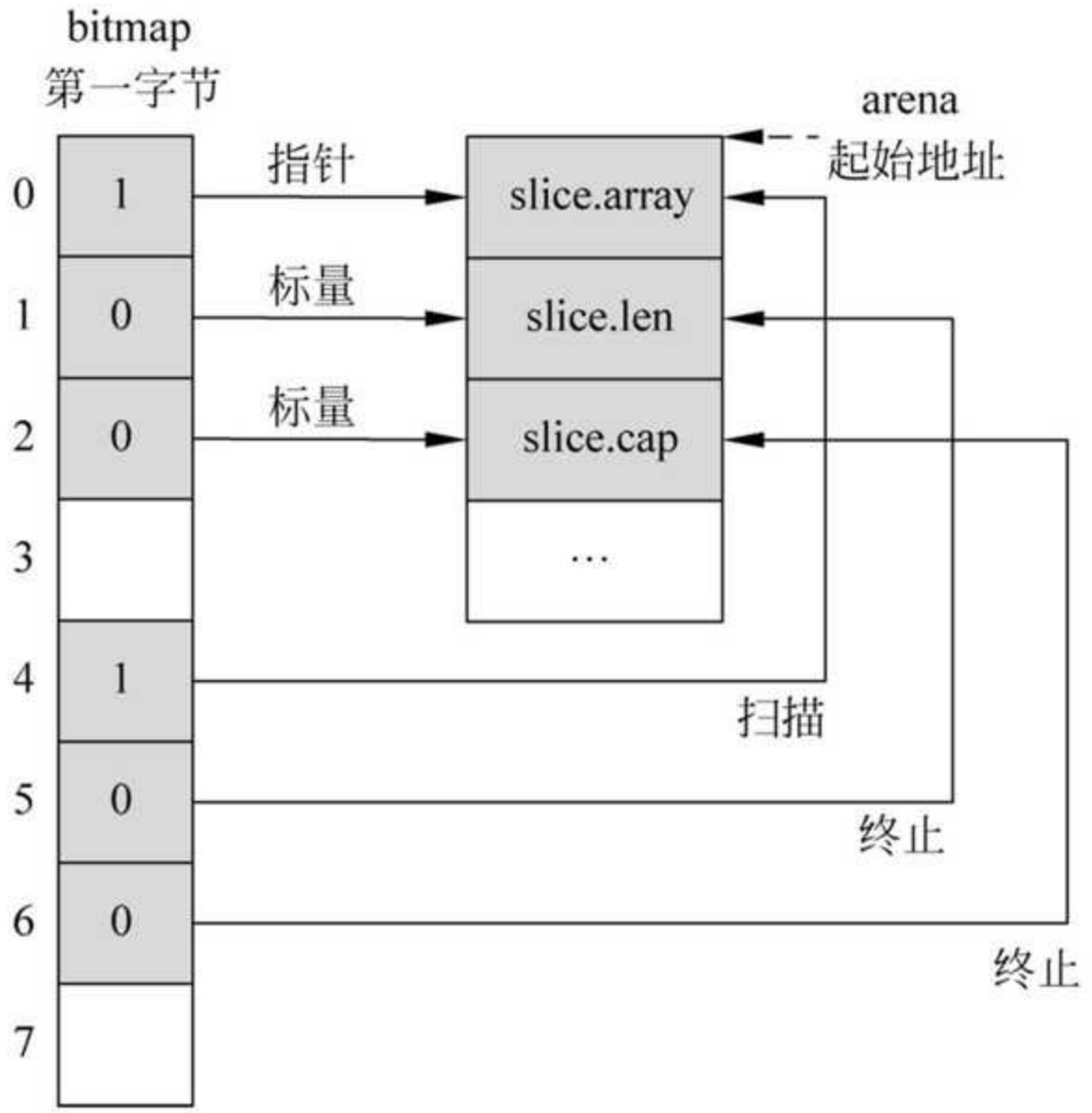

图8-3 arena起始处分配一个slice对应的bitmap标记

spans数组用来把当前arena中的页面映射到对应的mspan，暂时先认为一个mspan管理一组连续的内存页面，8.1.3节中会详细介绍mspan。pagesPerArena表示arena中共有多少个页面，用arena大小（64MB）除以页面大小（8KB）得到的结果是8192，也就是每个arena中有8192个页面。如图8-4所示，用给定地址相对arena起始地址的偏移除以页面大小，就可以得到对应页面在arena中的序号，将该序号用作spans数组的下标，就可以得到对应的mspan了。

pageInUse是个长度为1024的uint8数组，实际上被用作一个8192位的位图，通过它和spans可以快速地找到那些处于mSpanInUse状态的mspan。虽然pageInUse位图为arena中的每个页面都提供了一个二进制位，但是对于那些包含多个页面的mspan，只有第1个页面对应的二进制位会被用到，标记的是整个span。如图8-5所示，arena起始第一页对应的mspan只包含了一个页面，对应pageInUse位图第0位为1。第二页对应的mspan包含了连续的两个页面，对应pageInUse第1位被使用，记为1。接下来第四页至第六页对应一个mspan，在pageInUse位图中只有第四页对应的位被标记为1。

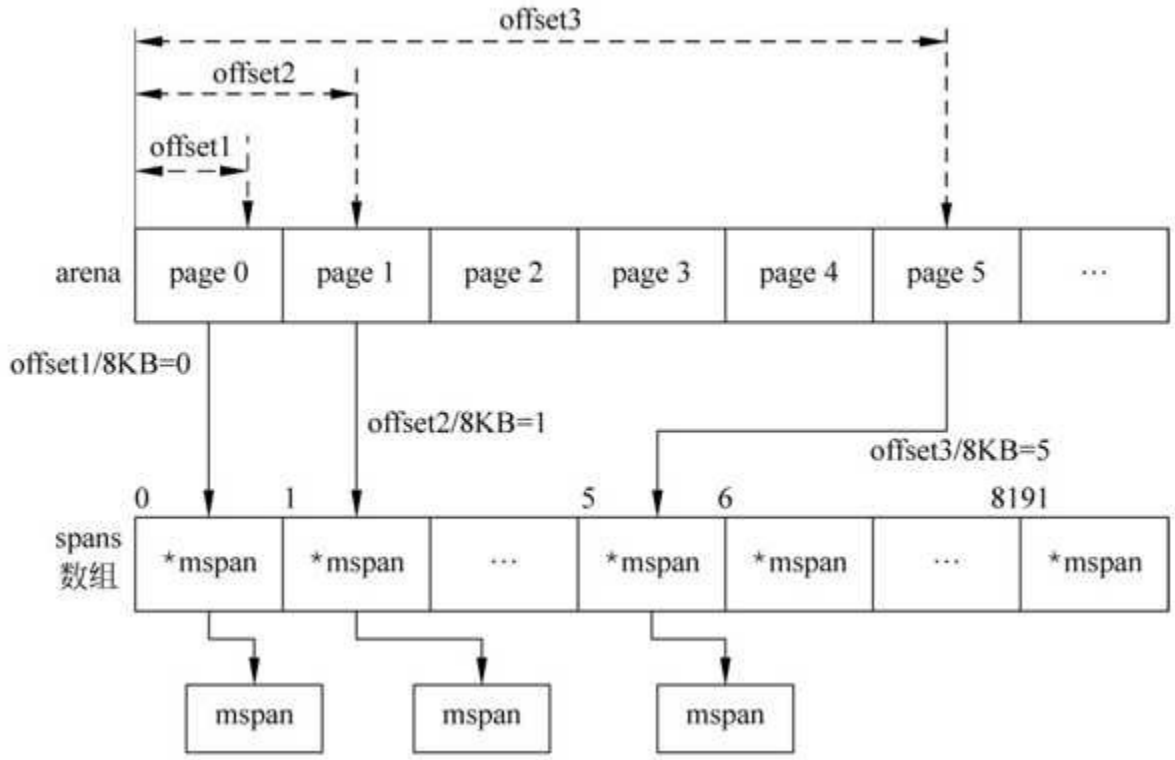

图8-4 arena中的页面到mspan的映射

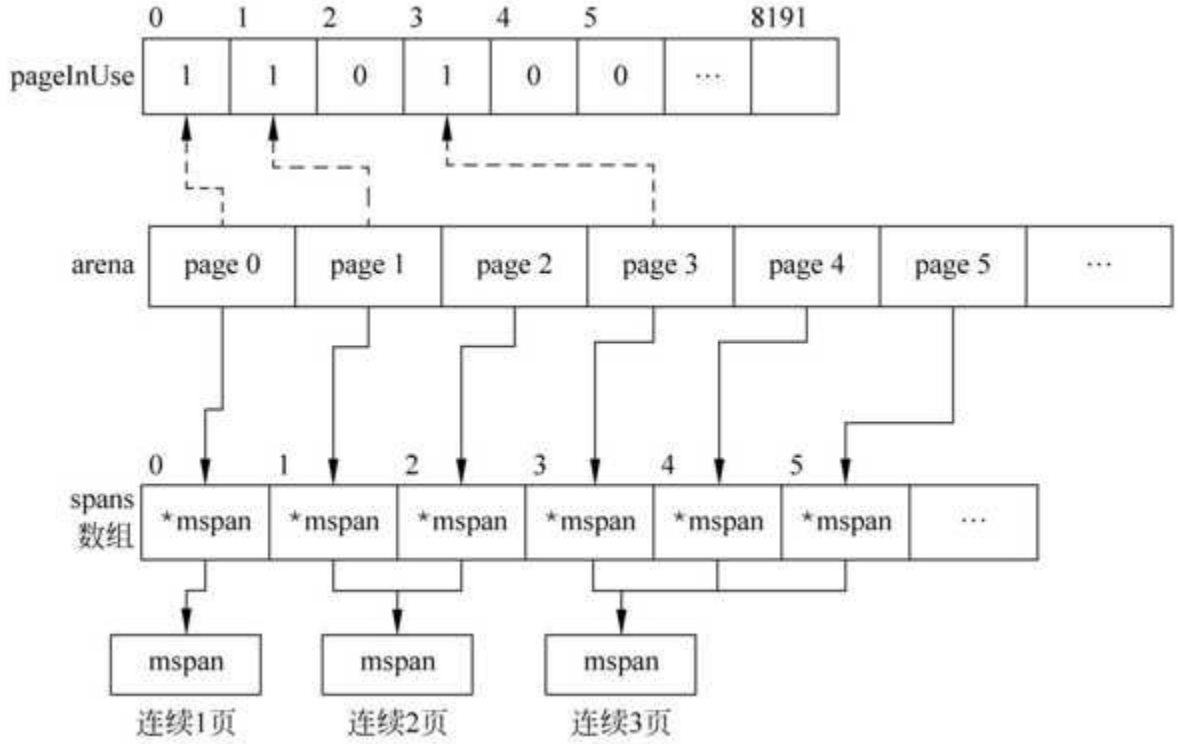

图8-5 pageInUse位图标记使用中的span

pageMarks表示哪些span中存在被标记的对象，与pageInUse一样用与起始页面对应的一个二进制位来标记整个span。在GC的标记阶段会原子性地修改这个位图，标记结束之后就不会再进行改动了。清扫阶段如果发现某个span中不存在任何被标记的对象，就可以释放整个span了。

pageSpecials又是一个与pageInUse类似的位图，只不过标记的是哪些span包含特殊设置，目前主要指的是包含finalizers，或者runtime内部用来存储heap profile数据的bucket。

checkmarks是一个大小为1MB的位图，其中每个二进制位对应arena中一个指针大小的内存单元。当开启调试debug.gccheckmark的时候，checkmarks位图用来存储GC标记的数据。该调试模式会在STW的状态下遍历对象图，用来校验并发回收器能够正确地标记所有存活的对象。

zeroedBase记录的是当前arena中下个还未被使用的页面的位置，相对于arena起始地址的偏移量。页面分配器会按照地址顺序分配页面，所以zeroedBase之后的页面都还没有被用到，因此还都保持着清零的状态。通过它可以快速判断分配的内存是否还需要进行清零。

1．arenaHint

Go的堆是动态按需增长的，初始化的时候并不会向操作系统预先申请一些内存备用，而是等到实际用到的时候才去分配。为避免随机地申请内存造成进程的虚拟地址空间混乱不堪，我们要让堆区从一个起始地址连续地增长，而arenaHint结构就是用来做这件事情的，它提示分配器从哪里分配内存来扩展堆，尽量使堆按照预期的方式增长，该结构的定义代码如下：

```
type arenaHint struct {
    addr uintptr
    down bool
    next * arenaHint
}
```

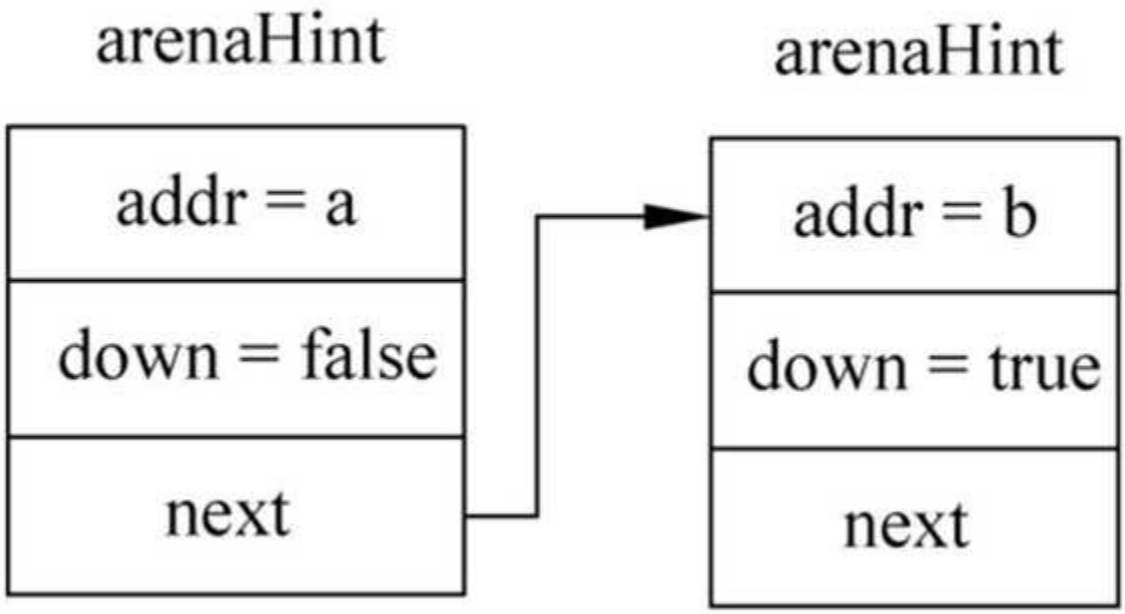

图8-6 两段可用区间通过arenaHint链表表示

addr是可用区间的起始地址，down表示向下增长。当down为false时，addr表示可用区间的低地址，类似数学上的左闭区间。当down为true时，addr表示可用区间的高地址，类似数学上的右开区间。arenaHint只给出了起始地址和增长方向，但没有给出可用空间的结束地址。next用来指向链表中的下一个arenaHint，sysAlloc（）函数根据当前arenaHint的指示来扩展堆空间，当申请内存遇到错误时会自动切换至下一个arenaHint。图8-6给出了两个向不同方向增长的arenaHint构成的链表。

2．arenaIdx

在amd64架构的Linux环境下，arena的大小和对齐边界都是64MB，所以整个虚拟地址空间都可以看

作由一系列arena组成的。如图8-7所示，arena区域的起始地址被定义为常量arenaBaseOffset。用一个给定的地址p减去arenaBaseOffset，然后除以arena的大小heapArenaBytes，就可以得到p所在arena的编号。反之，给定arena的编号，也能由此计算出arena的地址。相关计算的代码如下：

```
func arenaIndex(p uintptr) arenaIdx {
    return arenaIdx((p - arenaBaseOffset) / heapArenaBytes)
}

func arenaBase(i arenaIdx) uintptr {
    return uintptr(i) * heapArenaBytes + arenaBaseOffset
}
```

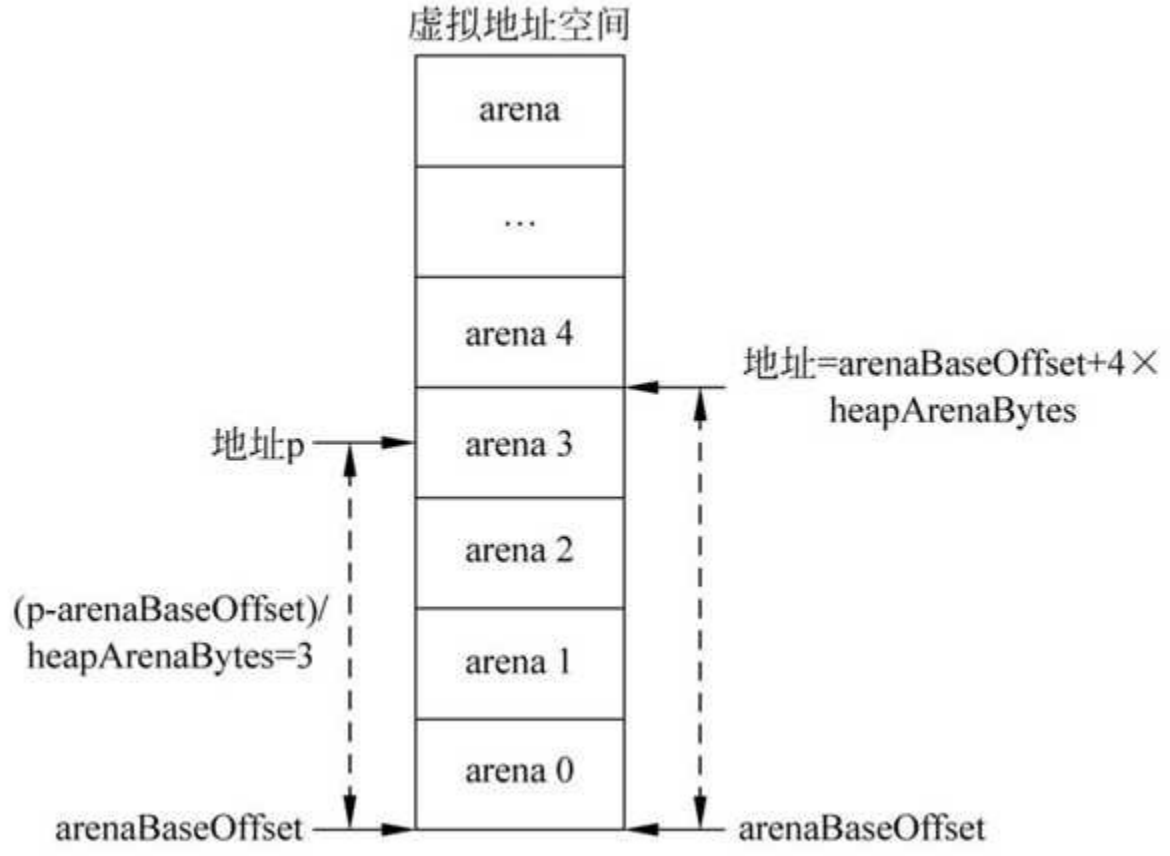

图8-7 地址与arena编号之间的换算

其中，arenaIdx类型底层是个uint，它的主要作用是用来寻址对应的heapArena。在amd64架构上虚拟地址的有效位数是48位，arena的大小是64MB，即26位，两者相差22位，也就是说整个地址空间对应4MB个arena。我们已经知道每个arena都有一个对应的heapArena结构，如果用arena的编号作为下标，把所有heapArena的地址放到一个数组中，则这个数组将占用32MB空间。32MB还可以接受，但是在某些系统上就不止32MB了，在amd64架构的Windows上，受系统原因影响，arena的大小是4MB，缩小了16倍，用来寻址heapArena的数组就会相应地变大16倍，那就无法接受了，所以Go的开发者把arenaIdx分成了两段，把用来寻址heapArena的数组也做成了两级，有点类似于两级页表，代码如下：

```
type arenaIdx uint

func (i arenaIdx) l1() uint {
    if arenaL1Bits == 0 {
        return 0
    } else {
        return uint(i) >> arenaL1Shift
    }
}

func (i arenaIdx) l2() uint {
    if arenaL1Bits == 0 {
        return uint(i)
    } else {
        return uint(i) & (1 << arenaL2 位 s - 1)
    }
}
```

在Linux系统上，arenaL1Bits被定义为0，而在amd64架构的Windows系统上被定义为6。第二级的位数等于虚拟地址有效位数48减去arena大小对应的位数和第一级的位数，在amd64架构下，arenaL2Bits在Linux系统上是22，在Windows系统上是20。再来看一下用来寻址heapArena的数组，它就是mheap结构的arenas字段，代码如下：

```
arenas [1 << arenaL1Bits] * [1 << arenaL2Bits] * heapArena
```

在Linux系统上，第一维数组的大小为1，相当于没有用到，只用到了第二维这个大小为4M的数组，arenaIdx全部的22位都用作第二维下标来寻址，如图8-8所示。

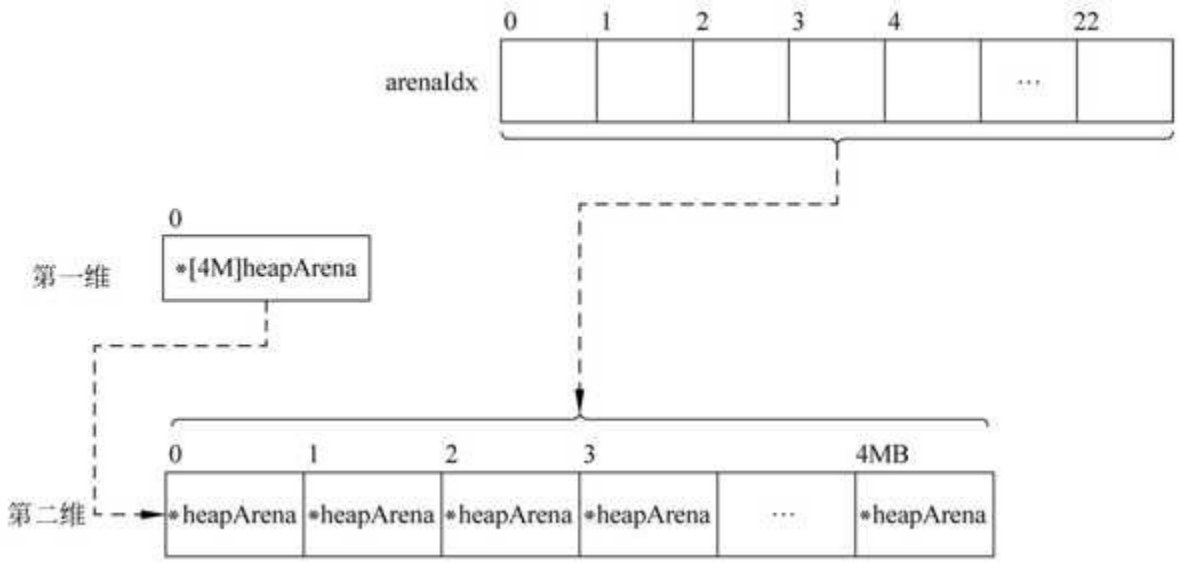

图8-8 Linux系统上用来寻址heapArena的二维数组

在Windows系统上，第一维数组的大小为64M，第二维大小为1M，因为两级都存储了指针，利用稀疏数组按需分配的特性，可以大幅节省内存。arenaIdx被分成两段，高6位用作第一维下标，低20位用作第二维下标，如图8-9所示。

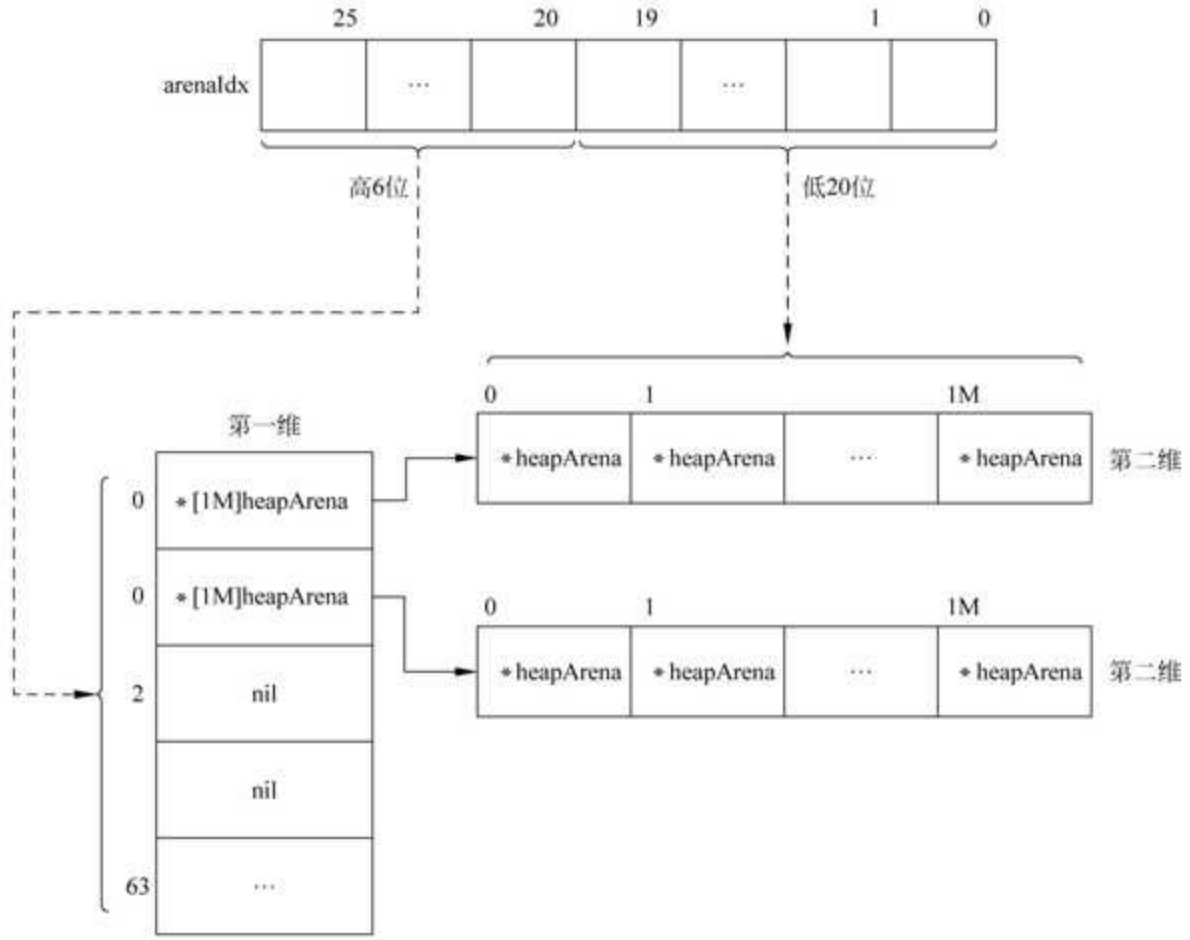

图8-9 Windows系统上用来寻址heapArena的二维数组

3．spanOf

至此，我们知道了如何根据一个给定的地址找到它所在的mspan。假设给定地址p，先用p减去堆区的起始地址，再除以arena的大小，就可以得到对应的arenaIdx，如图8-7所示。进一步如图8-8与图8-9所示，通过二维数组arenas得到heapArena的地址。再用p对arena的大小取模得到p在arena中的偏移量，然后除以页面大小，就可以得到对应页面的序号，将该序号用作spans数组的下标，就可以得到mspan的地址了，如图8-4所示。

在runtime中提供了一些函数，专门用来根据给定的地址查找对应的mspan，其中最常用的就是spanOf（）函数。该函数在进行映射的同时，还会校验给定的地址是不是一个有效的堆地址，如果有效就会返回对应的mspan指针，如果无效则返回nil，函数的代码如下：

```
func spanOf(p uintptr) *mspan {
    ri := arenaIndex(p)
    if arenaL1Bits == 0 {
        if ri.l2() >= uint(len(mheap_.arenas[0])) {
            return nil
        }
    } else {
        if ri.l1() >= uint(len(mheap_.arenas)) {
            return nil
        }
    }
    l2 := mheap_.arenas[ri.l1()]
    if arenaL1Bits != 0 && l2 == nil {
        return nil
    }
    ha := l2[ri.l2()]
    if ha == nil {
        return nil
    }
    return ha.spans[(p/pageSize) % pagesPerArena]
}
```

第1个最外层的if负责校验arenaIdx有没有越界，例如在amd64架构上，arenas数组是按照48位有效地址位来分配的，而程序代码中的地址被扩展到了64位，所以要经过校验才能保证安全。第2个最外层的if用来判断稀疏数组第二维的某个数组是否被分配，避免遇到空指针。最后一个if检测的是对应的arena是否已经分配，对于未分配的arena，与之对应的heapArena也不会被分配，所以指针为空。runtime中还有一个spanOfUnchecked（）函数，与spanOf（）函数功能类似，只不过移除了与安全校验相关的代码，需要调用者来保证提供的是一个有效的堆地址，函数的代码如下：

```
func spanOfUnchecked(p uintptr) *mspan {
    ai := arenaIndex(p)
    return mheap_.arenas[ai.l1()][ai.l2()].spans[(p/pageSize) % pagesPerArena]
}
```

本节关于arena相关的分析就到这里，期间我们多次提到了mspan，8.1.3节中将围绕mspan进行一些分析探索。

8.1.3 mspan

mspan用来记录和管理一组连续的内存页，这段连续的内存通常会被按照某个sizeclass划分成等大的内存块，内存块的分配及GC的标记和清扫都是在mspan层面完成的。除了自动管理模式之外，mspan也支持手动管理模式。和heapArena一样，mspan也是在堆之外单独分配的。在进一步分析探索之前，我们还是先来看一下mspan的数据结构，代码如下：

```
type mspan struct {
    next            * mspan
    prev            * mspan
    list            * mSpanList
    startAddr       uintptr
    npages          uintptr
    manualFreeList gclinkptr
    freeindex       uintptr
    nelems          uintptr
    allocCache      uint64
    allocBits       * gcBits
    gcmarkBits      * gcBits
    sweepgen        uint32
    divMul          uint16
    baseMask        uint16
    allocCount      uint16
    spanclass       spanClass
    state           mSpanStateBox
    needzero        uint8
    divShift        uint8
    divShift2       uint8
    elemsize        uintptr
    limit           uintptr
    speciallock     mutex
    specials        * special
}
```

next和prev用来构建mspan双链表，list指向双链表的链表头。startAddr指向当前span的起始地址，因为span都是按整页面分配的，所以指向的是首个页面的地址。npages记录的是当前span中有几个页面，乘以页面大小就可以得到span空间的大小。manualFreeList是个单链表，在mSpanManual类型的span中，用来串联所有空闲的对象。类型gclinkptr底层是个uintptr，它把每个空闲对象头部的一个uintptr用作指向下一个对象的指针，如图8-10所示。

nelems记录的是当前span被划分成了多少个内存块。freeindex是预期的下个空闲对象的索引，取值范围在0和nelems之间，下次分配时会从这个索引开始向后扫描，假如发现第N个对象是空闲的，就将其用于分配，并会把freeindex更新成N+1。allocBits和gcmarkBits分别指向当前span的分配位图和标记位图，其中每个二进制位对应span中的一个内存块，如图8-11所示。给定当前span中一个内存块的索引n，如果n>=freeindex并且allocBits[n/8]&（1<<（n%8））=0，则该内存块就是空闲的。

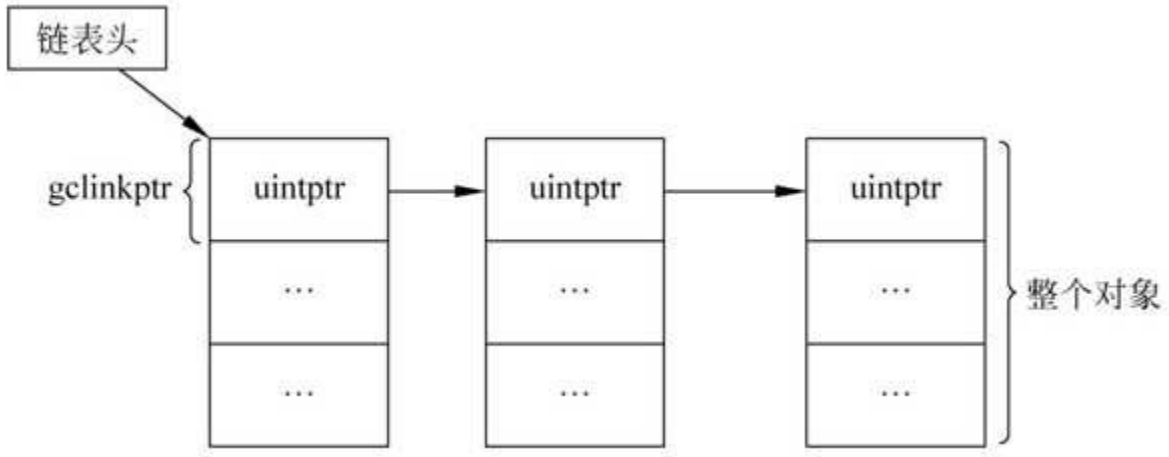

图8-10 gclinkptr串联的对象

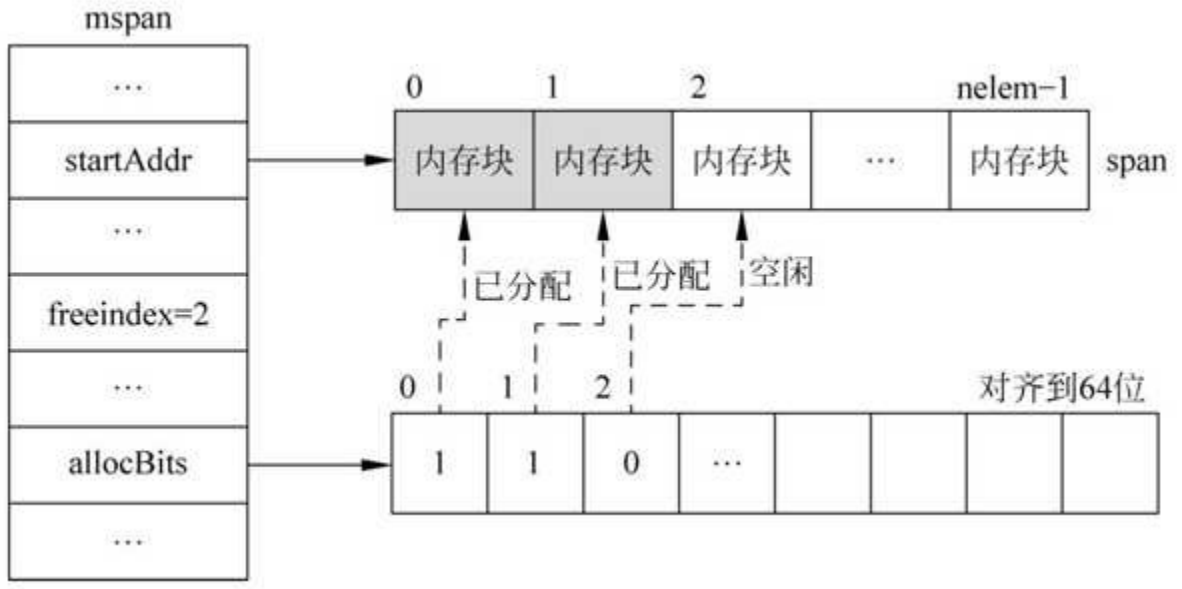

图8-11 allocBits位图对应span中已分配和未分配的内存块

清扫阶段会释放旧的allocBits，然后把gcmarkBits用作allocBits，并为gcmarkBits重新分配一段清零的内存。allocCache缓存了allocBits中从freeindex开始的64个二进制位，这样一来在实际分配时更高效。sweepgen与mheap.sweepgen相比较，能够得知当前span处于待清扫、清扫中、已清扫等哪种状态。divMul、baseMask、divShift、divShift2都是用来优化整数除法运算的，转换成乘法运算和位运算后更高效。allocCount用于记录当前span中有多少内存块被分配了。spanclass类似于sizeclass，实际上它把sizeclass左移了一位，用最低位记录是否不需要扫描，称为noscan，如图8-12所示。Go为同一种sizeclass提供了两种span，一种用来分配包含指针的对象，另一种用来分配不包含指针的对象。这样一来不包含指针的span就不用进一步扫描了，noscan位就是这个意思。

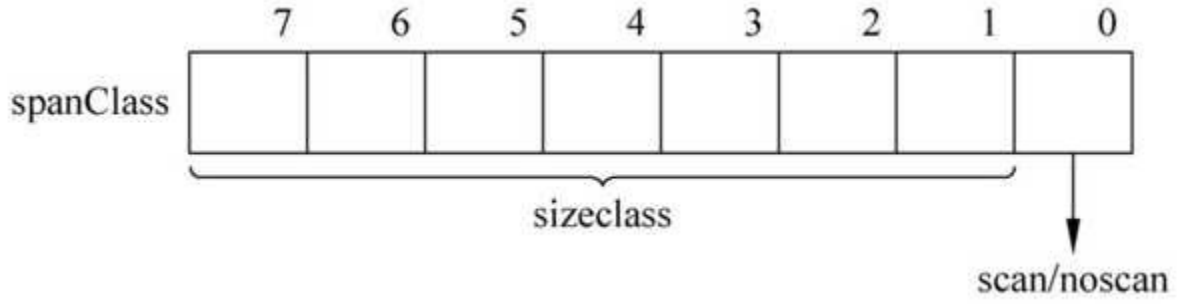

图8-12 spanclass中sizeclass和noscan位的位置

state记录的是当前span的状态，有mSpanDead、mSpanInUse和mSpanManual这3种取值，分别表示无效的mspan、被GC自动管理的span和手动管理的span。goroutine的栈分配用的就是mSpanManual状态的span。needzero表明分配之前需要对内存块进行清零。elemsize是内存块的大小，可以通过spanclass计算得到。limit记录的是span区间的结束地址，为右开区间。specials是个链表，用来记录添加的finalizer等，speciallock用来保护这个链表。

1．nextFreeIndex（）方法

在了解了mspan各个字段的大致作用之后，我们再来分析一个重要的方法nextFreeIndex（），这个方法的作用是寻找下一个空闲的索引，在实际分配内存的时候会用到它，该方法的源代码如下：

```
func (s *mspan) nextFreeIndex() uintptr {
    sfreeindex := s.freeindex
    snelems := s.nelems
    if sfreeindex == snelems {
        return sfreeindex
    }
    if sfreeindex > snelems {
        throw("s.freeindex > s.nelems")
    }

    aCache := s.allocCache
    bitIndex := sys.Ctz64(aCache)
    for bitIndex == 64 {
        sfreeindex = (sfreeindex + 64) &^(64 - 1)
        if sfreeindex >= snelems {
            s.freeindex = snelems
            return snelems
        }
        whichByte := sfreeindex / 8
        s.refillAllocCache(whichByte)
        aCache = s.allocCache
        bitIndex = sys.Ctz64(aCache)
    }
    result := sfreeindex + uintptr(bitIndex)
    if result >= snelems {
        s.freeindex = snelems
        return snelems
    }

    s.allocCache >>= uint(bitIndex + 1)
    sfreeindex = result + 1
    if sfreeindex%64 == 0 && sfreeindex != snelems {
        whichByte := sfreeindex / 8
        s.refillAllocCache(whichByte)
    }
    s.freeindex = sfreeindex
    return result
}
```

按照代码中的空行，可以把整个函数的逻辑分成三部分。第一部分只是做了些简单的校验，当freeindex等于nelems时，表明当前span中已经没有空闲的空间了。freeindex是不能大于nelems的，如果free index大于nelems，就意味着堆已经被破坏了，遇到这种不可恢复的错误，程序需要尽快崩溃。

第二部分的逻辑是寻找下个空闲索引，利用allocCache批量缓存allocBits能够提升效率。需要注意，allocCache中用0表示已分配，用1表示未分配，这点与allocBits是相反的，主要是为了方便通过Ctz64（）函数统计尾端为0的二进制位数量。如果Ctz64（）函数的返回值是64，说明allocCache中缓存的索引都被分配了，那就向后移动freeindex，并通过refillAllocCache（）方法重新填充allocCache，这个填充是按照64位对齐的。通过freeindex和对应的二进制位在allocCache中的偏移相加，就可以得到下个空闲索引，但是一定要判断有没有越界。allocBits因为要和allocCache配合，所以是按照64位的整倍数来分配的，但是nelems并不一定能被64整除，allocBits的位数是在nelems的基础上基于64做的向上对齐，所以尾部可能有一部分二进制位是无效的。

第三部分是分配后的调整工作，需要把freeindex指向分配后的下一个位置，allocCache也要相应地进行移位处理。如果此时freeindex能够被64整除，就说明allocCache缓存的二进制位都已经用完了，如果freeindex不等于nelems，也就是说当前span还有剩余空间，此时需要重新填充allocCache。最后，返回找到的空闲索引，函数返回后该索引也就被分配了。

现在回过头来看arena和span，heapArena层面实现了从堆地址到mspan的快速映射，并且为每个指针大小的内存单元提供了位图，这个位图能够用来区分指针和标量，以及确定要继续扫描还是应该终止，便于GC标记的时候高效地读取。span层面实现了细粒度内存单元的管理，与arena层面提供的位图不同，mspan中的分配位图和GC标记位图都是针对内存单元的，内存单元依据sizeclass指定的大小划分而成，而不是按照指针大小来提供的。heapArena和mspan中的位图，因为用处不一样，所以分别适合放在不同的地方。

2．setSpans

具体的span分配逻辑在mheap的allocSpan（）方法中，只不过代码篇幅有些长，就不进行详细分析了，这里只简单分析一下主要逻辑。allocSpan（）方法最主要的工作可以分成3步，第一步分配一组内存页面，第二步分配mspan结构，第三步设置heapArena中的spans映射。内存页面和mspan都有特定的分配器，这里不再进一步展开，重点关注一下第三步。mheap有个setSpans（）方法，专门用来把一个给定的span映射到相关的heapArena中，该方法的源代码如下：

```
func (h *mheap) setSpans(base, npage uintptr, s *mspan) {
    p := base / pageSize
    ai := arenaIndex(base)
    ha := h.arenas[ai.l1()][ai.l2()]
    for n := uintptr(0); n < npage; n++{
        i := (p + n) % pagesPerArena
        if i == 0 {
            ai = arenaIndex(base + n*pageSize)
            ha = h.arenas[ai.l1()][ai.l2()]
        }
        ha.spans[i] = s
    }
}
```

参数base给出了span这段内存的起始地址，npage给出了页面跨度，s是用来管理这个span的mspan结构。setSpans先根据base地址找到第1个heapArena，然后以页面为单位循环设置spans映射。当检测到达arena边界时，就会切换到下一个arena，说明span可以跨arena，如图8-13所示。

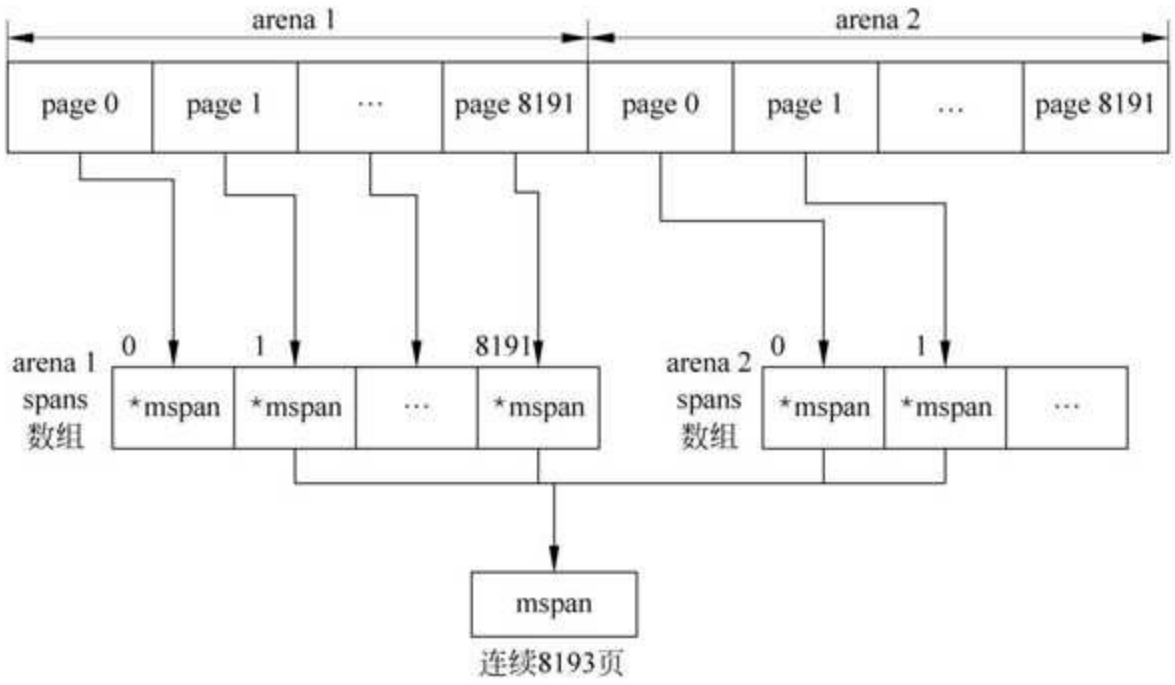

图8-13 跨arena的span示例

8.1.4 mcentral

在8.1.2节和8.1.3节中，我们了解了arena和span，arena中可以有多个不同sizeclass的span，将给定的地址经过heapArena.spans的映射，可以得到所属的mspan，这在GC标记的时候非常有用。只不过我们在分配内存的时候，需要根据sizeclass来找到对应的mspan，由于arena做不到这一点，因此，堆中引入了mcentral，可以先简单地把它理解成对应各种sizeclass的一组mspan空闲链表。在mheap中定义了一个mcentral的数组，代码如下：

```
central [numSpanClasses]struct {
    mcentral mcentral
    pad      [cpu.CacheLinePadSize - unsafe.Sizeof(mcentral{})
% cpu.CacheLinePadSize]byte
}
```

图8-14 mheap中的mcentral数组结构示意图

其中numSpanClasses是个值为136的常量，它是由（67+1）×2得来的。67种sizeclass再加上一个大小为0的sizeclass，然后乘以二是因为一份包含指针，另一份不包含指针，也就是noscan位，如图8-14所示。之所以不是直接基于mcentral的数组，而要再包一层struct，是为了使用pad来对齐到cache line大小，这样一来每个mcentral中的锁都在自己的cache line中。

一个mcentral类型的对象，对应一种spanClass，管理着一组属于该spanClass的mspan。mcentral的结构定义代码如下：

```
type mcentral struct {
    spanclass spanClass
    partial   [2]spanSet
    full      [2]spanSet
}
```

spanclass字段记录了当前mcentral管理着哪种类型的mspan。partial和full是两个spanSet数组，spanSet有自己的锁，是个并发安全地支持push和pop的*mspan集合。partial中都是还没有分配完的span，每个span至少包含一个空闲单元，full中都是没有空闲空间的span。这两个字段为什么都是数组呢？数组中的两个spanSet，有一个包含的是已清扫的span，另一个包含的是未清扫的span，并且它们在每轮GC中会互换角色。

mheap中的mcentral数组实现了全局范围的、基于spanClass的mspan管理。因为是全局的，所以需要加锁。为了进一步减少锁竞争，Go把mspan缓存到了每个P中，这就是8.1.5节中我们要了解的mcache。mcentral提供了两个方法用来支持mcache，一个是cacheSpan（）方法，它会分配一个span供mcache使用，另一个是uncacheSpan（）方法，mcache可以通过它把一个span归还给mcentral。这里不再分析这两个方法的代码，感兴趣的读者可自行查看。

8.1.5 mcache

在Go的GMP模型中，mcache是一个per-P的小对象缓存。因为每个P都有自己的一个本地mcache，所以不需要再加锁。mcache结构也是在堆之外由专门的分配器分配的，所以不会被GC扫描。mcache的结构定义代码如下：

```
type mcache struct {
    nextSample uintptr
    scanAlloc  uintptr
    tiny       uintptr
    tinyoffset uintptr
    tinyAllocs uintptr
    alloc      [numSpanClasses] * mspan
    stackcache [_NumStackOrders]stackfreelist
    flushGen   uint32
}
```

nextSample是配合memory profile来使用的，当开启memory profile的时候，每分配nextSample这么多内存后，就会触发一次堆采样。scanAlloc记录的是总共分配了多少字节scannable类型的内存，也就是noscan位为0、可以包含指针的span。tiny和tinyoffset用来实现针对noscan型小对象的tiny allocator，tiny用来指向一个16字节大小的内存单元，tinyoffset记录的是这个内存单元中空闲空间的偏移量。tiny allocator能够将一些小对象合并分配，极大地提高了空间利用率。tinyAllocs记录的是共进行了多少次tiny分配。alloc是根据spanClass缓存的一组mspan，因为不需要加锁，所以不用像mcentral那样对齐到cache line。stackcache是用来为goroutine分配栈的缓存，我们在第9章中将具体介绍栈内存的管理。flushGen记录的是上次执行flush时的sweepgen，如果不等于当前的sweepgen，就说明需要再次flush以进行清扫。

8.1.6 mallocgc

我们在8.1.1～8.1.5节中了解了与堆内存管理相关的一系列数据结构，还有几个比较关键的底层函数，了解这些都是为了能够更好地理解本节要介绍的mallocgc（）函数。本节之初已经讲过，mallocgc（）函数是堆分配的关键函数，runtime中的new系列函数和make系列函数都依赖于它。该函数的代码稍微有点长，完全贴出来既占用篇幅又不好分析，但是要完全不看代码又有种脱离实际的感觉，所以我们先大致讲一下函数的主要逻辑，再把关键代码分段进行细化分解。

mallocgc（）函数的主要逻辑按照代码的先后顺序可以分成如下几部分：

（1）检查当前goroutine的gcAssistBytes值，如果减去本次要分配的内存大小后结果为负值，就需要先调用gcAssistAlloc（）函数辅助GC完成一些标记任务。

（2）根据此次要分配的空间大小，以及是否要分配noscan类型空间，选用不同的分配策略。目前有3种分配策略，即tiny、sizeclass和large。

（3）如果分配的不是noscan类型空间，就需要调用heapBitsSetType（）函数，该函数会根据传入的类型元数据对heapArena中的位图进行标记。

（4）调用publicationBarrier、GC标记新分配的对象、memory profile采样、更新gcAssistBytes的值，按需发起GC等一系列收尾操作。

1. 辅助GC

辅助GC也就是mallocgc（）函数的第一部分，对应的源代码如下：

```
var assistG *g
if gcBlackenEnabled != 0 {
    assistG = getg()
    if assistG.m.curg != nil {
        assistG = assistG.m.curg
    }
    assistG.gcAssistBytes -= int64(size)
    if assistG.gcAssistBytes < 0 {
        gcAssistAlloc(assistG)
    }
}
```

其中gcBlackenEnabled就像是一个开关，它在GC标记开始的时候被设置为1，在标记结束的时候被清零，也就是只有在GC标记阶段才能执行辅助GC。每个goroutine都有自己的gcAssistBytes，在这个值用光之前不用执行辅助GC。辅助GC机制能够有效地避免程序过快地分配内存，从而造成GC工作线程来不及标记的问题。

2. 空间分配

空间分配指的是上述mallocgc（）函数的4个阶段中的第二阶段，这里会根据要分配的目标大小及是否为noscan型空间，来选用不同的分配策略。这里先来看一下是如何选择策略的，然后针对每种策略展开分析。选择分配策略的代码如下：

```
if size <= maxSmallSize {
    if noscan && size < maxTinySize {
        //使用 tiny allocator 分配
    } else {
        //使用 mcache.alloc 中对应的 mspan 分配
    }
} else {
    //直接根据需要的页面数,分配大的 mspan
}
```

maxSmallSize是个值为32768的常量，也就是说对于32KB以上的内存分配会直接根据需要的页面数分配一个新的span。maxTinySize是个值为16的常量，对于小于16字节且是noscan类型的内存分配请求会使用tiny分配器。对于[16，32768]这个范围内的noscan分配请求，以及不超过32768的所有scannable型分配请求都会使用预置的各种sizeclass来分配。

接下来我们先来看一下tiny allocator是如何分配空间的，相关代码如下：

```
off := c.tinyoffset
if size&7 == 0 {
    off = alignUp(off, 8)
} else if sys.PtrSize == 4 && size == 12 {
    off = alignUp(off, 8)
} else if size&3 == 0 {
    off = alignUp(off, 4)
} else if size&1 == 0 {
    off = alignUp(off, 2)
}
if off+size <= maxTinySize && c.tiny != 0 {
    x = unsafe.Pointer(c.tiny + off)
    c.tinyoffset = off + size
    c.tinyAllocs++
    mp.mallocing = 0
    releasem(mp)
    return x
}
span = c.alloc[tinySpanClass]
v := nextFreeFast(span)
if v == 0 {
    v, span, shouldhelpgc = c.nextFree(tinySpanClass)
}
x = unsafe.Pointer(v)
(*[2]uint64)(x)[0] = 0
(*[2]uint64)(x)[1] = 0
if size < c.tinyoffset || c.tiny == 0 {
    c.tiny = uintptr(x)
    c.tinyoffset = size
}
size = maxTinySize
```

先取出mcache中的tinyoffset，然后根据分配目标大小size进行对齐，如果对齐后的off加上size没有超过maxTinySize，就可以使用现有的tiny内存块直接分配。maxTinySize是常量16，也就是tiny allocator内部内存块的大小。如果当前内存块中剩余的空间不足以满足本次分配，就从mcache的alloc数组中找到对应tinySpanClass的mspan，并通过nextFreeFast（）函数重新分配一个16字节的内存块。如果对应的mspan中也没有空间了，nextFree（）方法会从mcentral中取一个新的mspan过来，并且返回值shouldhelpgc是true。最后，把新分配的内存块清零，如果本次分配之后新内存块的剩余空间大于旧内存块的剩余空间，就用新的把旧的替换掉。

tiny分配器被设计成能够将几个小块的内存分配请求合并到一个16字节的内存块中，这样能够提高内存空间的利用率。例如，通过tiny分配器分配16个1字节的内存，合并分配后利用率为100%，如图8-15所示。

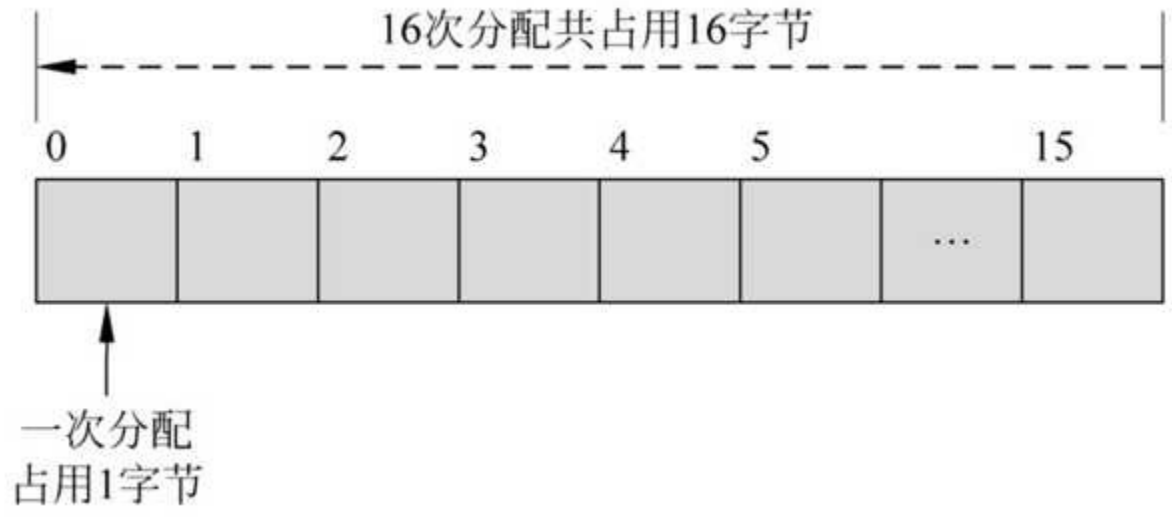

图8-15 使用tiny分配器连续分配16次1字节内存

如果没有tiny分配器，则每次分配1字节就需要适配sizeClass中最小的规格，即8字节，而且每次都会浪费7字节，内存实际利用率仅为12.5%，如图8-16所示。

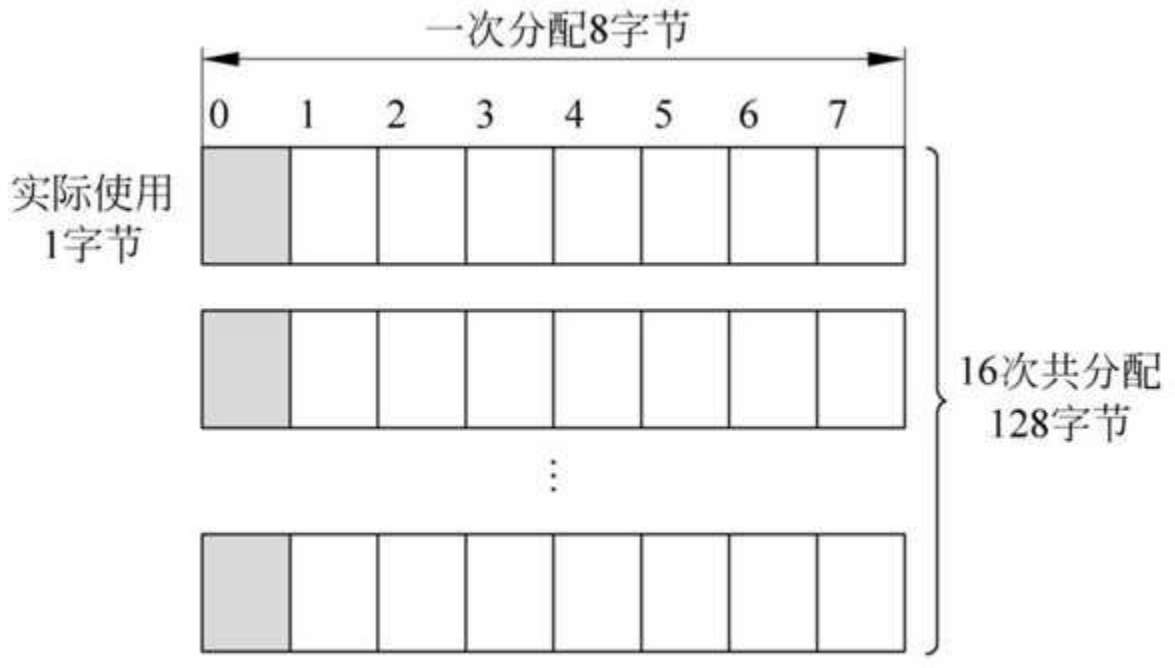

图8-16 适配sizeClass连续分配16次1字节内存

再来看一下使用预置的sizeclass来分配内存的情况，相关源代码如下：

```
var sizeclass uint8
if size <= smallSizeMax - 8 {
    sizeclass = size_to_class8[divRoundUp(size, smallSizeDiv)]
} else {
    sizeclass = size_to_class128[divRoundUp(size - smallSizeMax, largeSizeDiv)]
}
size = uintptr(class_to_size[sizeclass])
spc := makeSpanClass(sizeclass, noscan)
span = c.alloc[spc]
v := nextFreeFast(span)
if v == 0 {
    v, span, shouldhelpgc = c.nextFree(spc)
}
x = unsafe.Pointer(v)
if needzero && span.needzero != 0 {
    memclrNoHeapPointers(unsafe.Pointer(v), size)
}
```

smallSizeMax是个常量，值是1024。1024减去8是1016，也就是当size不超过1016时使用size_to_class8，否则使用size_to_class128将size映射到对应的sizeclass，然后结合noscan合成spc，并通过spc找到alloc数组中对应的mspan，再通过nextFreeFast（）函数分配内存块。如果mspan中也没有剩余空间，就调用nextFree（）方法去mcentral中取一个新的mspan。最后按需清空内存块。

当要分配的内存空间超过32KB时，就要直接分配内存页面了，具体代码如下：

```
shouldhelpgc = true
span = c.allocLarge(size, needzero, noscan)
span.freeindex = 1
span.allocCount = 1
x = unsafe.Pointer(span.base())
size = span.elemsize
```

allocLarge（）方法会把size向上对齐到整页面大小，然后分配一个大的span。最后，整个span被用作一个内存块返回给请求者。至此，空间分配逻辑也就梳理完了。

3. 位图标记

分配完空间之后，需要对heapArena中的位图进行标记，这个工作是由heapBitsSetType（）函数完成的。除此之外，还会把分配了多少需要扫描的空间累加到scanAlloc字段，具体代码如下：

```
var scanSize uintptr
if !noscan {
    if typ == deferType {
        dataSize = unsafe.Sizeof(_defer{})
    }
    heapBitsSetType(uintptr(x), size, dataSize, typ)
    if dataSize > typ.size {
        if typ.ptrdata != 0 {
            scanSize = dataSize - typ.size + typ.ptrdata
        }
    } else {
        scanSize = typ.ptrdata
    }
    c.scanAlloc += scanSize
}
```

如果分配的是noscan类型的空间，就可以跳过这一步了。计算scanSize的时候，用到了类型元数据中的ptrdata字段，它表示该类型的数据前多少字节中包含指针，后续还有数据也属于标量数据，只扫描前ptrdata字节就可以了。

4. 收尾工作

最后的收尾工作也包含多个操作，首先调用publicationBarrier（）函数，该函数相当于一个Store-Store屏障，在x86上根本用不着，所以被实现成一个空的函数，但在其他一些平台上有实际效果。在此之后要让GC标记新分配的对象，具体代码如下：

```
if gcphase != _GCoff {
    gcmarknewobject(span, uintptr(x), size, scanSize)
}
```

上述代码先进行了判断，只有在GC的标记阶段才能标记新分配的对象。在此之后Memory Profile的采样代码如下：

```
if rate := MemProfileRate; rate > 0 {
    if rate != 1 && size < c.nextSample {
        c.nextSample -= size
    } else {
        mp := acquirem()
        profilealloc(mp, x, size)
        releasem(mp)
    }
}
```

在Memory Profile开启的情况下，每分配nextSample字节内存以后，就进行一次采样。之后还剩最后一步GC相关操作，代码如下：

```
if assistG != nil {
    assistG.gcAssistBytes -= int64(size - dataSize)
}
if shouldhelpgc {
    if t := (gcTrigger{kind: gcTriggerHeap}); t.test() {
        gcStart(t)
    }
}
```

在分配的过程中，size可能已向上对齐过，所以可能会变大，而dataSize保存了原来真实的size值，最后要从分配内存的goroutine的gcAssistBytes中减去因size对齐而额外多分配的大小。最后执行检测操作，如果达到了GC的触发条件，就发起GC。

至此，关于堆内存分配的探索就先告一段落。本节中，我们了解了内置的sizeclasses和用来管理堆空间的arena，以及负责小块内存管理的span，分析了将堆地址映射到对应mspan的过程，以及mspan如何寻找下一个空闲的内存块。还有集中管理mspan的mcentral，和per-P缓存mspan的mcache。在本节的最后，分析了mallocgc（）函数的主要逻辑，了解了tiny、sizeclasses和large这3种内存分配策略，应该可以让各位读者理解堆内存分配的大致框架。

8.2 垃圾回收

垃圾回收器也就是我们通常所讲的GC，Go语言的垃圾收集器是基于精确类型的，并且被设计成可以与普通的线程并发运行，同时允许多个GC线程并行运行。它是一个使用写屏障的并发标记清除算法，是不分代的、不压缩的。一轮垃圾回收包含以下几个步骤：

（1）Sweep Termination，清扫终止，在本轮标记开始之前，先把上一轮剩余的清扫工作完成。具体来讲，首先Stop the World，从而使所有的P都达到一个GC安全点，然后清扫所有还未清扫的span，通常情况下不会存在还未清扫的span，除非GC提前触发。

（2）Concurrent Mark，并发标记，可以认为是本轮GC的主要工作。具体来讲，将gcphase从_GCoff改成_GCmark，启用写屏障和辅助GC，把GC root送入工作队列中。

直到所有的P都开启了写屏障后，GC才会开始标记对象，写屏障的开启也是在STW期间完成的，然后Start the World，GC工作线程和辅助GC会一起完成标记工作，写屏障会将指针赋值过程中被覆盖掉的旧指针和新指针同时着色，新分配的对象会被立即标为黑色。GC root包含所有协程的栈、可执行文件数据段和BSS段等全局数据区，以及来自runtime中一些堆外数据结构里的堆指针。扫描一个goroutine时会先将其挂起，对其栈上发现的指针进行着色，最后恢复它的运行。GC标记时，从工作队列中取出灰色对象，扫描该对象使其变成黑色，并对发现的指针进行着色，这可能又会向工作队列中添加更多指针。因为GC涉及多处本地缓存，所以它使用一种分布式算法来判断所有的GC root和灰色对象都已经处理完，之后会切换至Mark Termination，即标记终止状态。

（3）Mark Termination，标记终止。Stop the World，将gcphase设置成_GCmarktermination，关闭GC工作线程和辅助GC。冲刷所有的mcache，以将mspan还回mcentral中。

（4）Concurrent Sweep，并发清扫。将gcphase设置成_GCoff，重置清扫相关状态并关闭写屏障。Start the World，此后分配的对象就都是白色的了，必要时，分配之前会先对span进行清扫。除了分配时清扫之外，GC还会进行后台清扫。等到分配的内存达到一定的阈值后，又会触发下一轮GC。

如图8-17所示，在整个GC的过程中一共需要两次Stop the World，分别是在清扫终止和标记终止这两个阶段，程序需要STW来达到一致状态，好在这两个阶段都比较短暂。在并发标记和并发清扫阶段都允许普通goroutine和GC worker并发运行，整体上STW的占比是非常小的。

图8-17 一轮GC的几个阶段

8.2.1 GC root

所谓GC root，其实就是标记的起点。程序运行阶段内存中所有的变量、对象之间是有关联性的，例如通过一个struct的地址可以找到这个对象，它的所有字段中如果包含指针，又可以进一步寻址其他的变量或对象。通过指针，所有的变量和对象组成了一张大图，而GC root就是这张图的一组起点，从这组起点出发能够遍历整张图。没错，是一组起点而不是一个，因为这张图的结构非常复杂，有多个起点，只通过其中一两个起点不足以遍历整张图。下面我们就来看一看Go的几种GC root，以及GC是如何扫描的。

1. 全局数据区

全局数据区这种叫法是针对进程的内存布局来讲的。一般情况下，变量的分配位置就是全局数据区、栈区和堆区这3处。堆区主要用于运行阶段的动态分配，而栈区以栈帧为单位来分配，装载的是函数的参数、返回值和局部变量，全局数据区主要用于全局变量的分配。对应到Go语言，就是包级别的变量，此处全局的含义应该理解为与进程生命周期相同，而不是语法层面的作用域。在可执行文件中有两个节区可以被认为是全局数据区，一个是data段，里面都是有初始值的变量，另一个是bss段，里面是未初始化的变量。事实上，bss段在可执行文件中不会被实际分配空间，只是有个对应的header来描述它。在可执行文件加载的时候，加载器会为bss段分配空间，如图8-18所示。

全局数据区中的变量分配在构建阶段就已经确定了，在运行阶段有可能包含指向堆区的指针，所以需要把它作为GC root进行扫描。如何进行扫描呢？如果再逐个分析每个变量的类型元数据，效率就太低了，毕竟我们只关心其中是否包含指针而已。好在构建工具已经把供GC使用的位图写入了可执行文件中，通过moduledata的gcdatamask和gcbssmask字段就可以直接使用了，其中每一位对应全局数据区中的一个指针大小的内存块，实际上就是个指针位图，如图8-19所示。

2. 栈

这里指的是所有goroutine的栈，以及与栈密切相关的_defer、_panic这些对象。因为goroutine是活动的，会分配内存、进行指针赋值等操作，所以栈上往往会有大量指向堆内存的指针，_defer和_panic对象里也有一些指针字段，GC在对栈进行扫描时会一并处理。与全局数据区类似，栈扫描时也需要知道哪些是指针，所以需要获得与每个栈帧对应的指针位图，如图8-20所示，runtime.getStackMap（）函数能够返回目标栈帧的局部变量和参数的指针位图，以及栈帧上的对象列表。

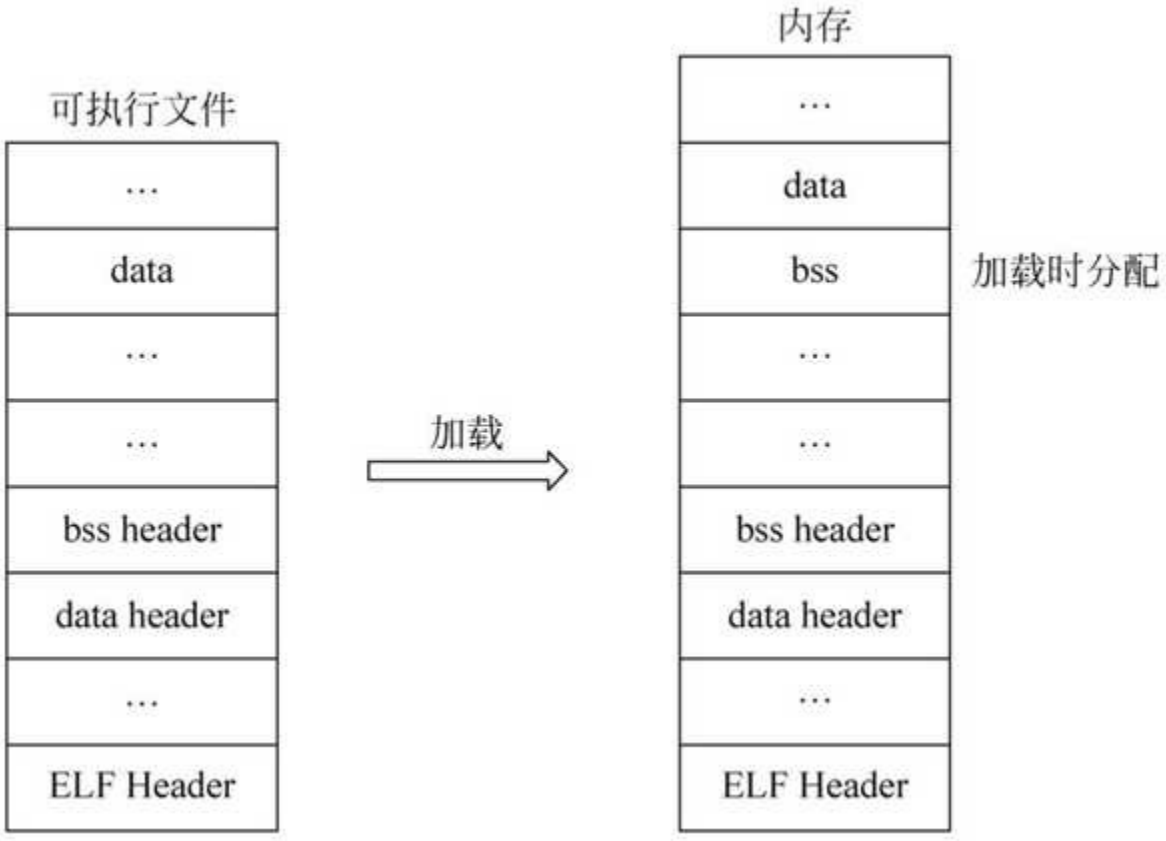

图8-18 可执行文件加载时为bss段分配空间

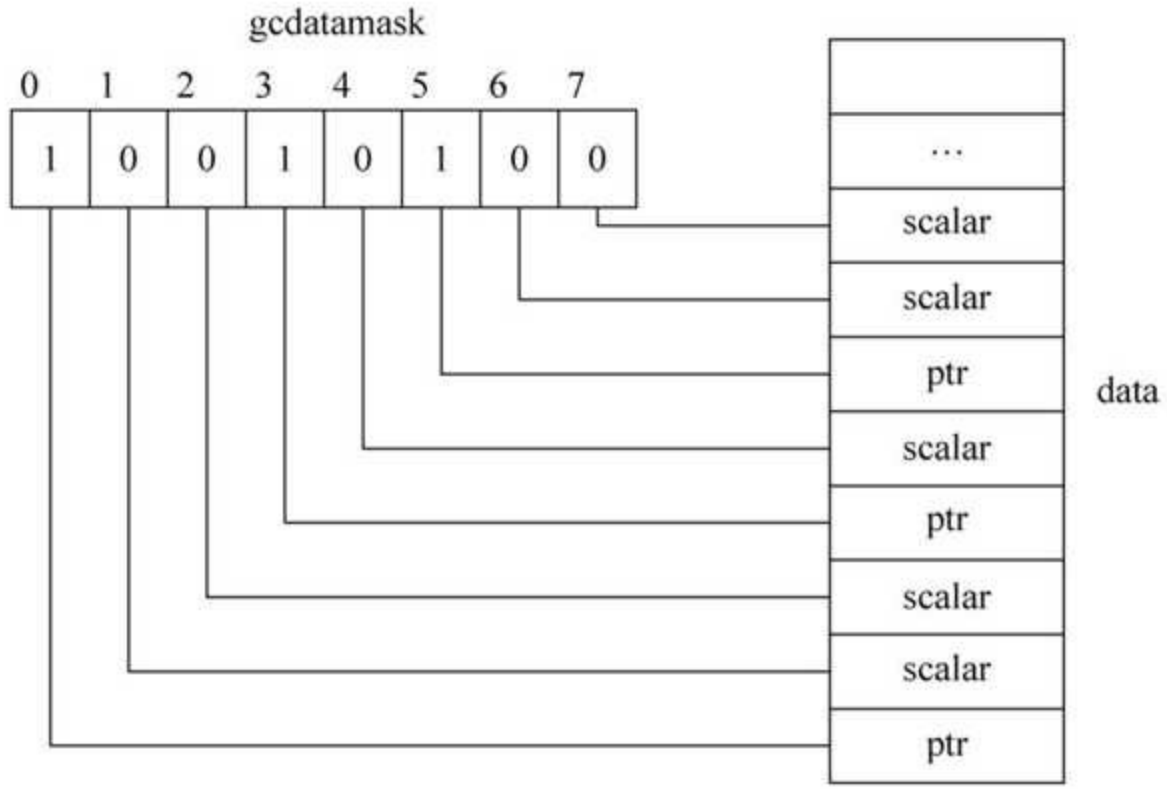

图8-19 gcdatamask是标记全局数据区的指针位图

3．Finalizer

通过runtime.SetFinalizer（）函数能够为堆上分配的对象关联一个finalizer（）函数，当GC发现了一个关联了finalizer（）函数的不可达对象时，它就会取消它们之间的关联，并在一个特有的协程中用该对象的地址作为参数来调用finalizer（）函数。这样一来就会使该对象再次变成可达的，只是不再有与之关联的finalizer（）函数了，这样下一轮GC就会发现它不可达，进而把它清理掉。

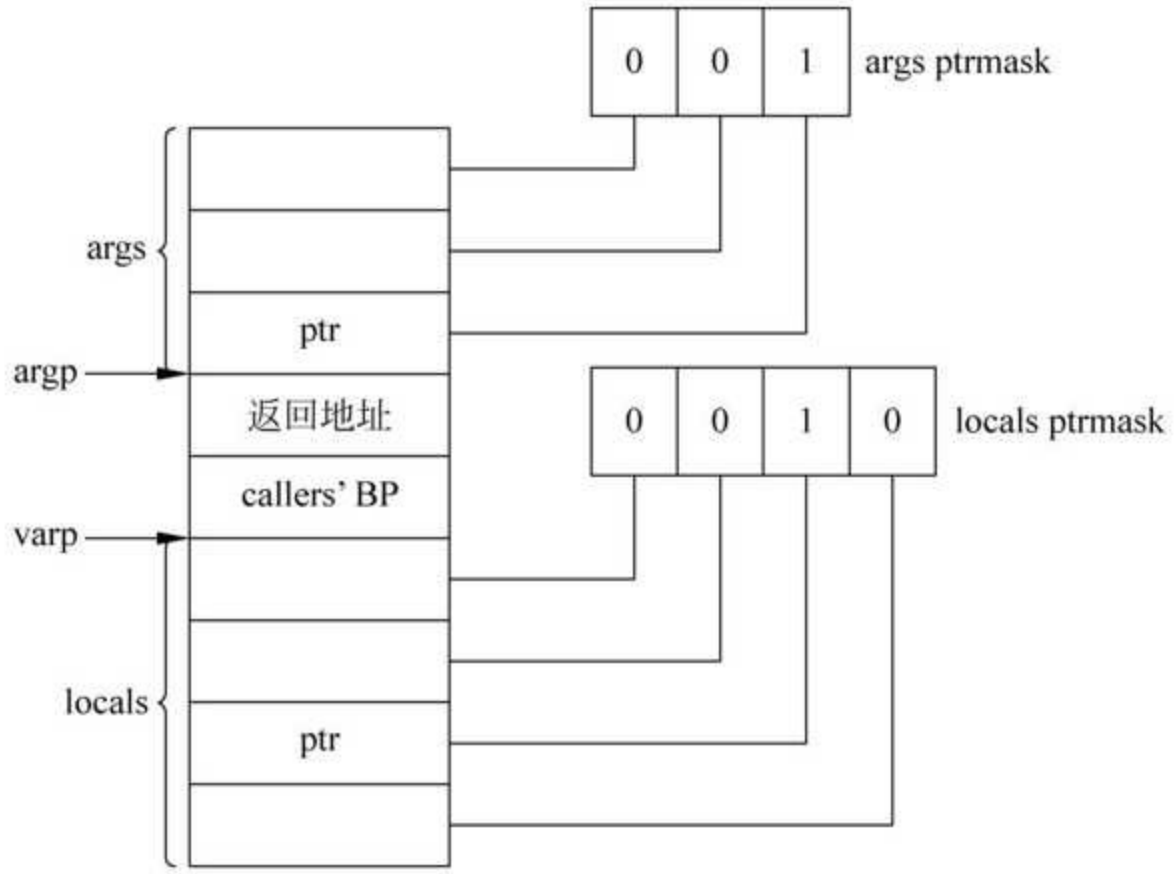

图8-20 每个栈帧都有对应的指针位图

Finalizer为什么也属于GC root呢？事实上，从该对象可到达的所有内容都必须被标记，对象自身却不需要。因为我们需要保证，在把对象的地址作为参数调用与它关联的finalizer（）函数时，通过该对象可到达的所有内容都被保留了下来，否则就会发生意料之外的错误。与对象关联的finalizer（）函数使用一个specialfinalizer结构来存储，该结构的定义代码如下：

```
//go:notinheap
type specialfinalizer struct {
    special special
    fn      *funcval
    nret    uintptr
    fint    *_type
    ot      *ptrtype
}
```

specialfinalizer对象不是在堆上分配的，因此其中的一些指针字段也需要扫描，主要是指向funcval的指针，因为这个Function Value本身可能是一个在堆上分配的闭包对象，如图8-21所示。

8.2.2 三色抽象

Go语言的GC使用了三色抽象标记堆中的对象，使用的3种不同颜色及其含义如表8-2所示。

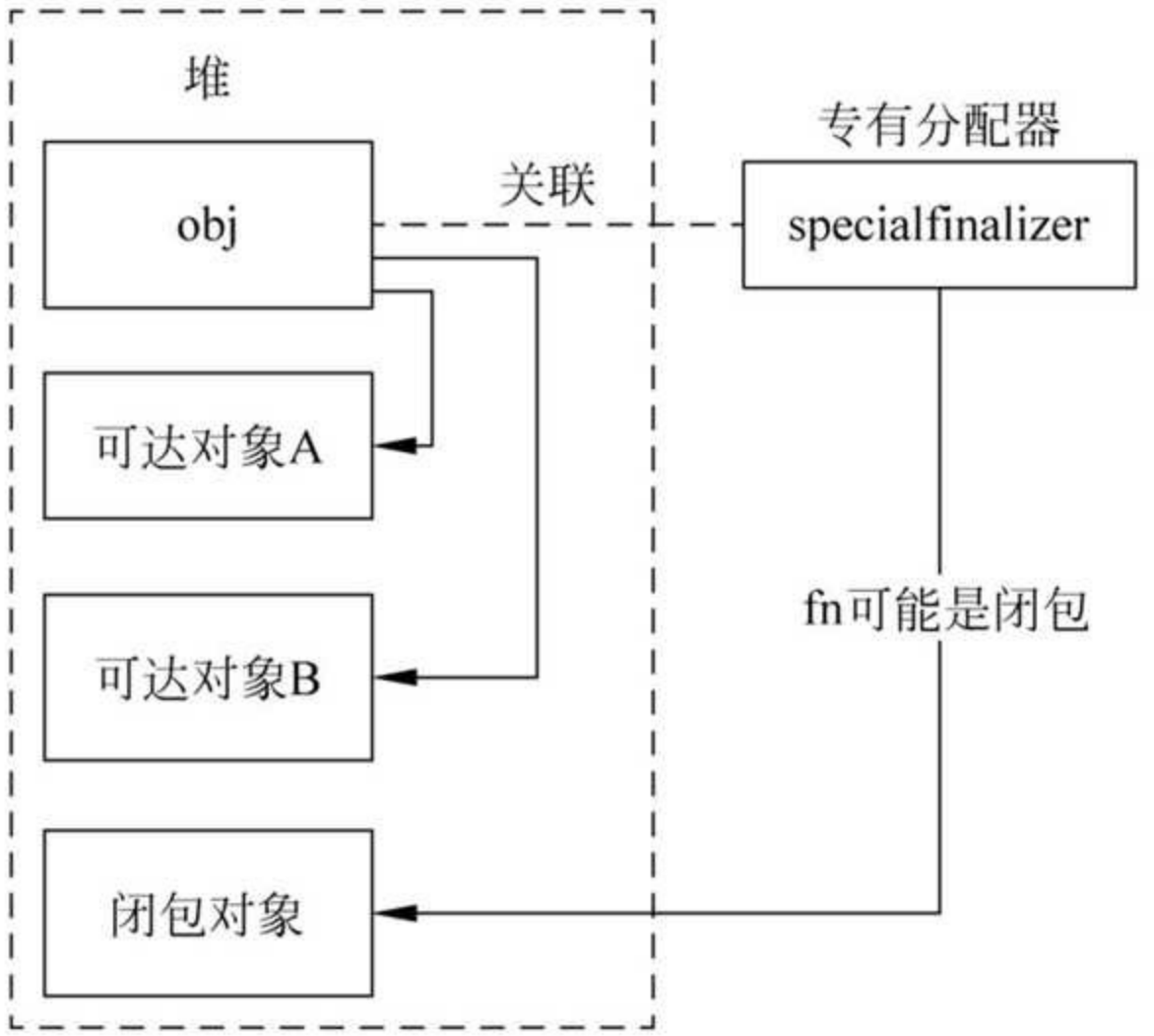

图8-21 标记关联有finalizer的对象

表8-2 三色抽象的颜色及其含义

颜　色	含　义
白色	表示未被标记也未扫描的对象
灰色	表示已经被标记但还未进行扫描的对象
黑色	表示已经被标记并且已经扫描完成的对象

每轮GC标记开始时，所有对象都是白色的，标记过程中3种颜色的对象都会存在，标记结束时只剩下黑色和白色的对象。在并发清理阶段会清理掉那些白色对象，因为新分配的对象也是白色的，所以要先将整个span清理后才能用于新的分配。

GC的工作队列实现了灰色对象指针的生产者——消费者模型，灰色对象实际上是一个被标记并且被添加到工作队列中等待扫描的对象，黑色对象同样也被标记过，但是不在工作队列中。如图8-22所示，写屏障、GC root扫描、栈扫描和对象扫描都会向工作队列中添加更多指针，扫描工作会消费工作队列中的灰色指针，使它们变成黑色并扫描它们，可能又会产生更多灰色指针。

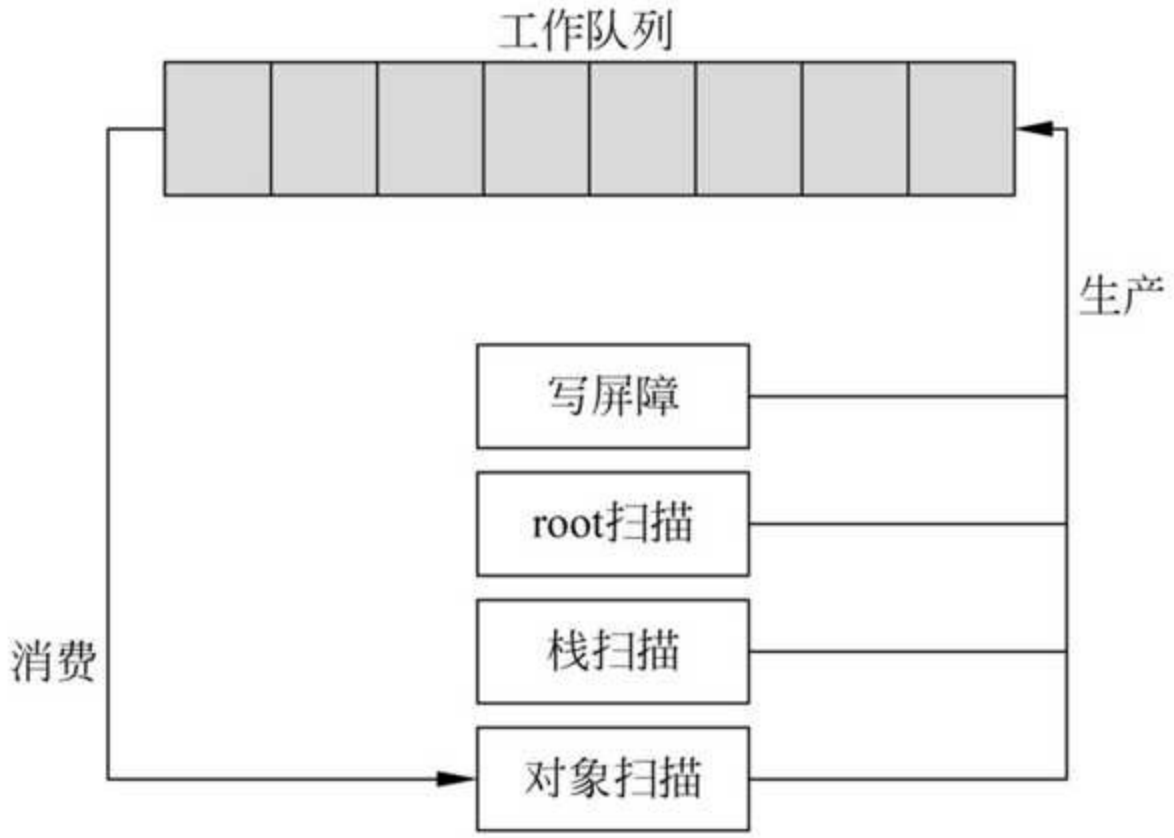

图8-22 工作队列的生产者——消费者模型

在runtime中GC工作队列的具体实现就是gcWork这个结构体类型，结构的定义代码如下：

```
type gcWork struct {
    wbuf1, wbuf2 *workbuf
    bytesMarked  uint64
    scanWork     int64
    flushedWork  bool
}
```

其中wbuf1和wbuf2分别是主要和次要工作缓冲区。bytesMarked记录了通过当前工作队列标记了多少内存空间，最终会被聚合到全局的work.bytesMarked中。scanWork记录了当前工作队列执行了多少扫描工作，也是以字节为单位的。flushedWork表示从上次gcMarkDone检测之后，有非空的工作缓冲区被冲刷到了全局的工作列表中，与标记终止的判定有关。

工作缓冲区对应的workbuf结构，以及workbuf内嵌的workbufhdr结构的定义代码如下：

```
//go:notinheap
type workbuf struct {
    workbufhdr
    obj [(_WorkbufSize - unsafe.Sizeof(workbufhdr{})) / sys.PtrSize]uintptr
}

type workbufhdr struct {
    node lfnode //must be first
    nobj int
}
```

其中的obj是个uintptr数组，用来存储扫描过程中发现的指针。nobj用于记录obj数组使用了多少，实际上是个递增的下标，为0时表示缓冲区是空的，等于obj的长度时表示缓冲区已满。lfnode的类型是个结构体类型，包含一个int64和一个uintptr，可以简单地认为它用来构建链表，包含指向下一个节点的指针。_WorkbufSize的值是2048，所以数组obj的容量应该是253。

实际为堆对象着色的工作是runtime中的greyobject（）函数实现的，以下就是笔者精简过的函数源码，去掉了与调试相关的部分代码，保留了最主要的逻辑，代码如下：

```
//go:nowritebarrierrec
func greyobject(obj, base, off uintptr, span *mspan, gcw *gcWork, objIndex uintptr) {
    if obj&(sys.PtrSize-1) != 0 {
        throw("greyobject: obj not pointer-aligned")
    }

    mbits := span.markBitsForIndex(objIndex)
    if mbits.isMarked() {
        return
    }
    mbits.setMarked()

    arena, pageIdx, pageMask := pageIndexOf(span.base())
    if arena.pageMarks[pageIdx]&pageMask == 0 {
        atomic.Or8(&arena.pageMarks[pageIdx], pageMask)
    }

    if span.spanclass.noscan() {
        gcw.BytesMarked += uint64(span.elemsize)
        return
    }

    if !gcw.putFast(obj) {
        gcw.put(obj)
    }
}
```

第1个if语句用于校验对象地址是不是按照指针对齐的，8.1节讲内存分配的时候已经知道各种sizeclass都是8的整数倍。接下来调用markBitsForIndex（）方法，可以通过对象内存块在span中的索引objIndex定位到gcmarkBits对应的二进制位，如果已经标记过了就直接返回，如果未标记就进行标记。heapArena结构中的pageMarks记录的是哪些span中存在被标记过的对象，所以要通过它把对象所在的span也标记一下。图8-23展示了greyobject（）方法标记对象的效果。

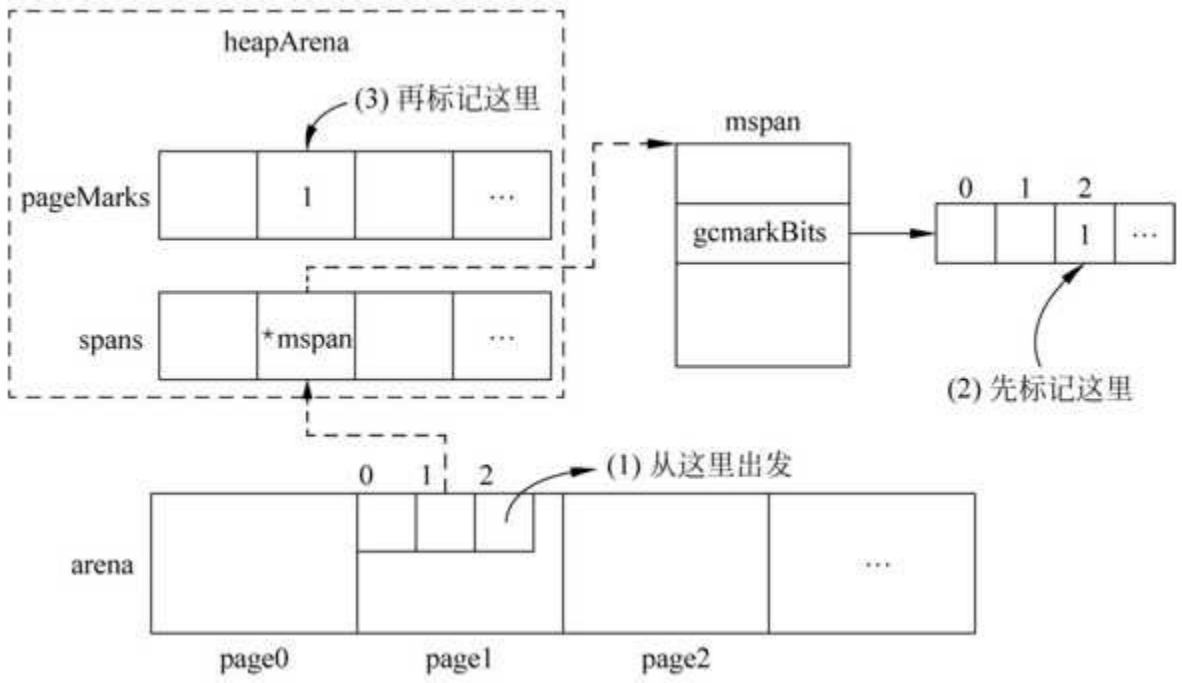

图8-23 greyobject标记对象

接下来的if语句用于判断对象所在的span是否为noscan型，也就是不包含指针，那样就不用进一步对对象进行扫描了，记录bytesMarked后直接返回，对象obj就是黑色的了。如果span不是noscan型的，就把对象指针添加到工作队列中，等待后续进一步对对象展开扫描。

8.2.3 写屏障

GC使用写屏障来追踪指针的赋值操作，Go使用的是一种组合了删除写屏障和插入写屏障的混合写屏障。删除写屏障负责对地址被覆盖掉的对象进行着色，插入写屏障负责对新地址指向的对象进行着色。在goroutine的栈是灰色的时候，才有必要执行插入写屏障。按照这种设计思想，混合写屏障的伪代码如下：

```
writePointer(slot, ptr):
    shade( * slot)
    if current stack is grey:
        shade(ptr)
    * slot = ptr
```

其中slot是个指向指针的指针，也就是指针赋值运算中目的操作数的地址，ptr是用来赋值的新值。shade（）函数会根据传入的地址标记堆上的对象，还会把该地址添加到GC工作队列中，前提是传入的是一个堆地址。混合写屏障能够防止goroutine对GC隐藏某个对象：

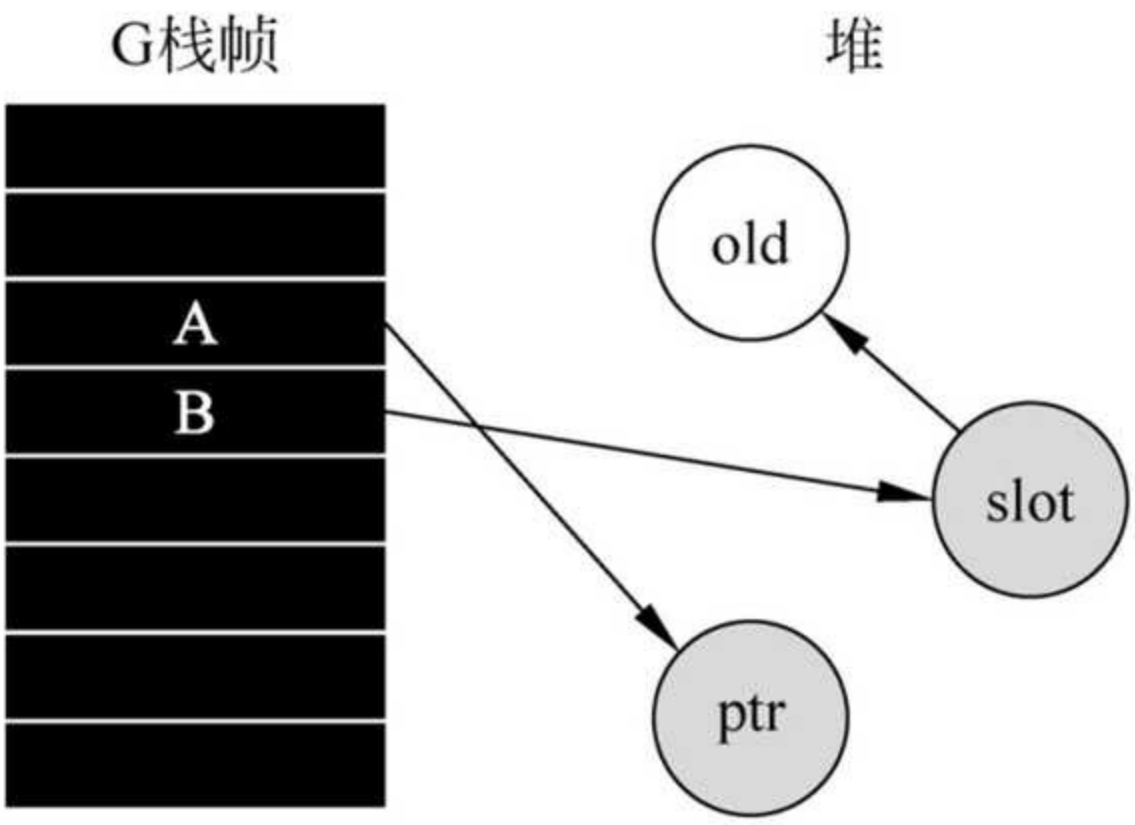

图8-24 示例初始状态G栈帧已完成扫描

（1）shade（*slot）能够防止goroutine把指向对象的唯一指针从堆或全局数据段移动到栈上，从而造成对象被隐藏，当goroutine尝试删除一个堆上的指针时，删除写屏障负责为该指针指向的对象着色。

例如，当前协程G的栈已经完成扫描，A和B是栈上的两个指针，如图8-24所示。

接下来G会执行如下操作：

①把old的地址写入栈上的本地变量A。

②把ptr的地址写入slot。

上述第一步操作因栈上没有插入写屏障，不会标记old指针，如果堆上没有删除写屏障，指向old的唯一路径被切断，old就不能被GC发现了，如图8-25所示。

所以在删除堆上的指针时应用删除写屏障对old进行标记，如图8-26所示。

（2）shade（ptr）能够防止goroutine把指向对象的唯一指针从它的栈上移动到堆或全局数据段的某个黑色对象里而造成的隐藏问题，当goroutine尝试向一个黑色对象里写入指针时，插入写屏障负责为该指针指向的对象着色。

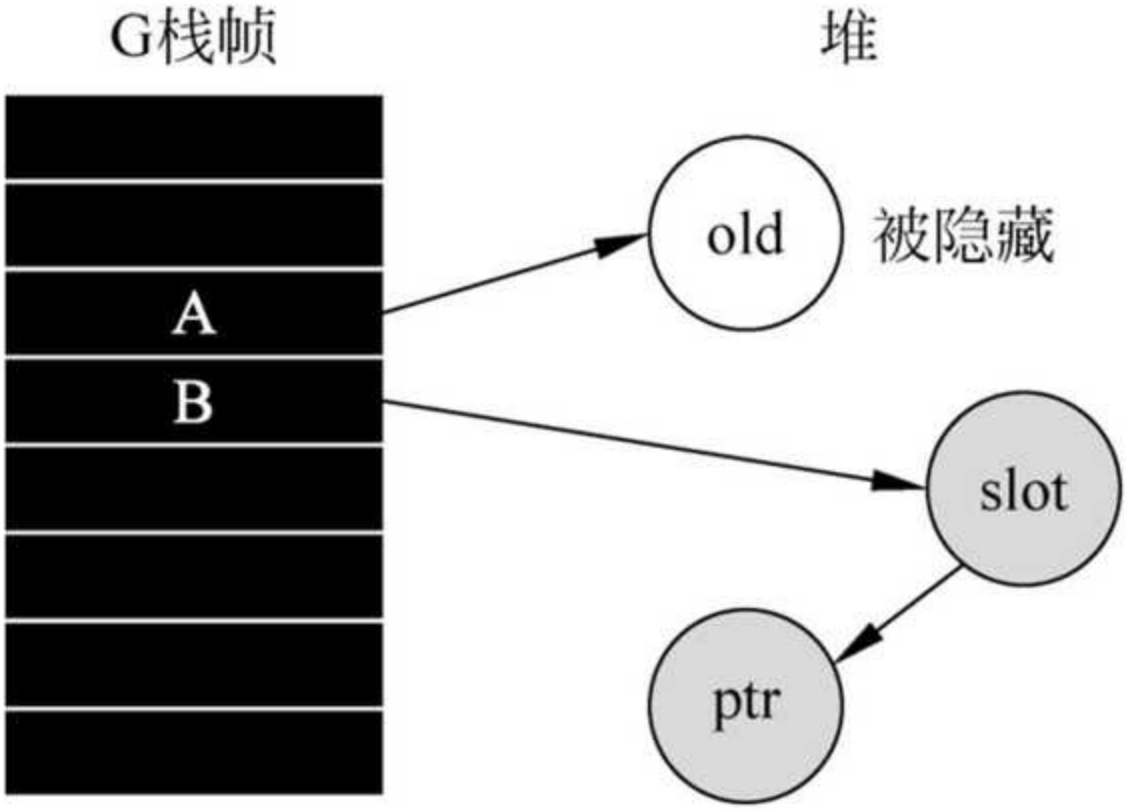

图8-25 堆上没应用删除写屏障时old被隐藏

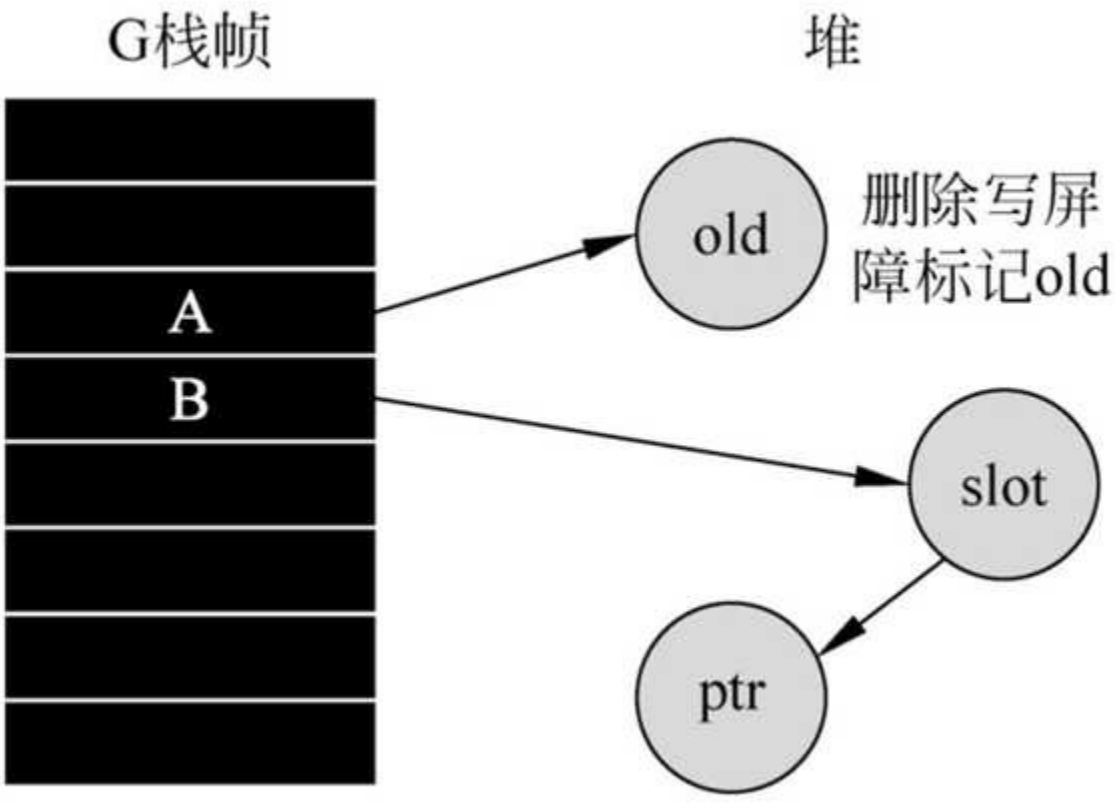

图8-26 堆上应用删除写屏障对old进行标记

例如，当前协程G的栈还未扫描，A和B是栈上的两个指针，堆上的slot和old是在标记期间分配的，所以都已被标记为黑色，之前分配的ptr还是白色，如图8-27所示。

接下来G会执行如下操作：

①把ptr的地址写入slot。

②把old的地址写入B。

上面第二步操作在栈上没有删除写屏障，不会标记ptr。如果没有插入写屏障，就会将白色对象ptr写入堆上的黑色对象slot，此时ptr就不能被GC发现了，如图8-28所示。

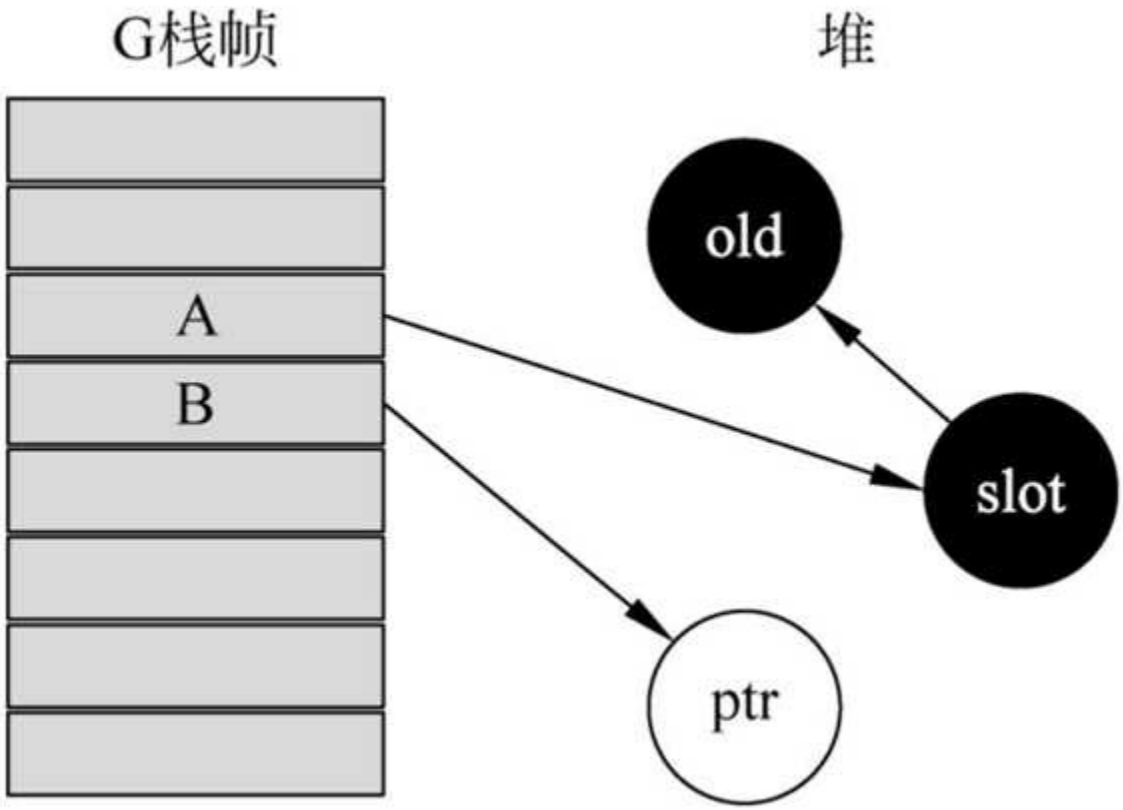

图8-27 示例初始状态G栈帧还未扫描

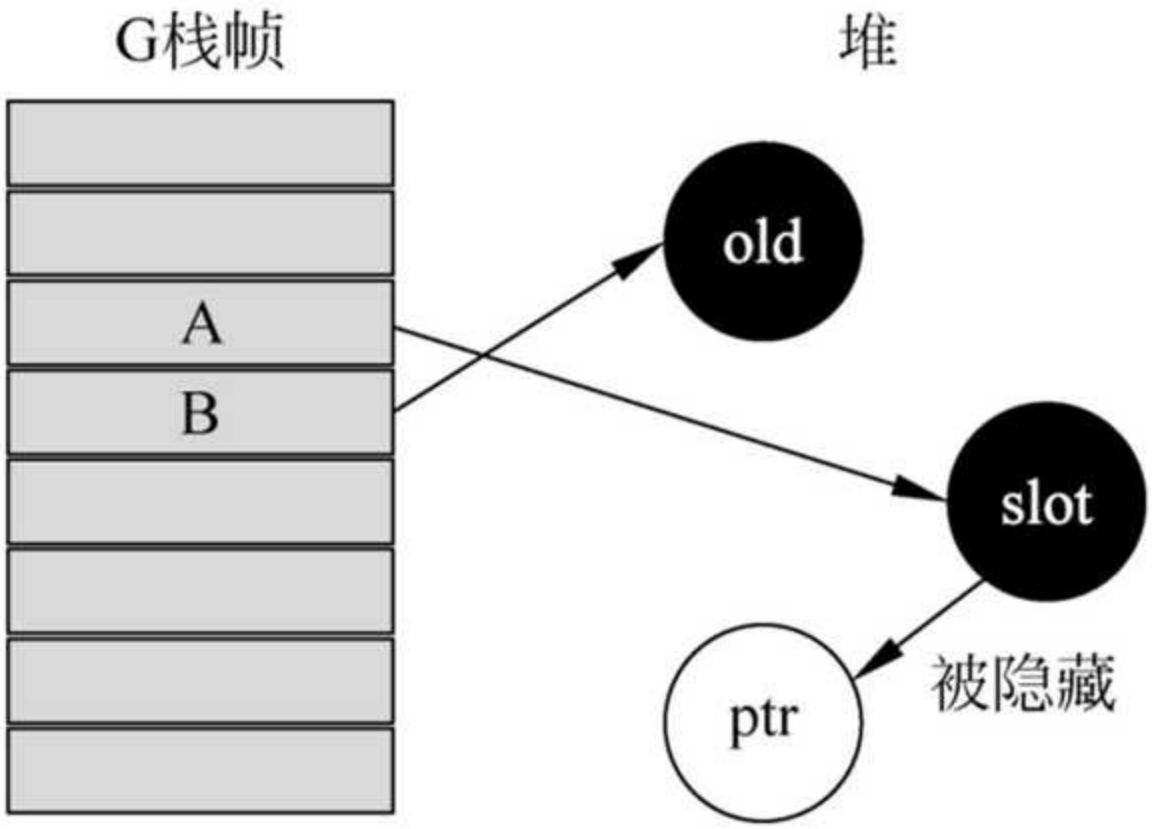

图8-28 没应用插入写屏障时ptr被隐藏

为了避免将白色对象写入堆上的黑色对象，就要靠插入写屏障，在写入slot时标记新指针ptr，如图8-29所示。

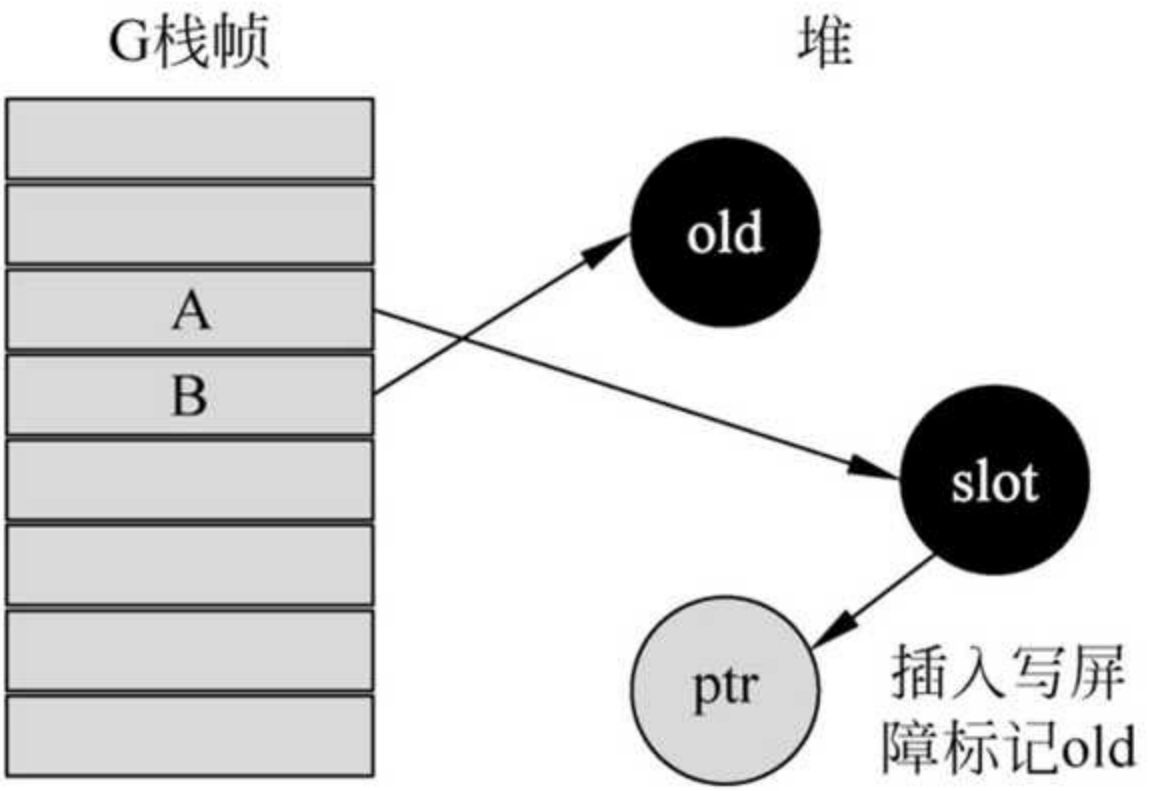

图8-29 堆上应用插入写屏障对ptr进行标记

（3）如果goroutine的栈是黑色的，则shade（ptr）就没有必要了。因为把对象指针从栈上移动到堆或全局数据段进而造成其隐藏的前提是，该指针在栈上时是未被标记的。栈刚刚被扫描完时，它指向的对象都是被标记过的，所以不会有隐藏的对象指针。shade（*slot）会防止后续有指针在栈上被隐藏。

还是直白点讲更好理解，当进入并发标记阶段以后，程序会从所有GC root开始扫描遍历整个对象图。并发标记在设计上允许普通goroutine和GC一起运行，只是在扫描goroutine栈的时候会暂时将其挂起，扫描完成后又会恢复运行。goroutine恢复运行之后可能又会对整个对象图进行一系列修改，主要是新分配内存和移动现有指针（指针赋值），标记阶段分配的内存都会被mallocgc（）函数直接标黑，所以不会有遗漏，但是指针赋值会有较多变数。我们不能重新扫描整个对象图，只想处理后来的增量变动，而写屏障就能很好地追踪到goroutine恢复运行后造成的增量变动。

因为变量一般就是存在于全局数据段、堆区和栈区，所以goroutine造成的增量变动也就脱离不了这几处位置。如果对全局数据段、堆和栈都应用插入写屏障，则可以跟踪到所有增量修改，但是这就要求goroutine在向全局数据段、堆及栈上写入指针时，都要经过插入写屏障。全局数据段和堆还可以接受，如果操作当前函数栈帧中的指针都要经过插入写屏障，无论是对程序的性能还是可执行文件的体积，都会造成很大的影响，因为编译器需要额外生成大量代码，所以我们还要寻求一种新的方案，能够使goroutine不必对当前函数栈帧上的指针应用写屏障，这也就是Go为什么要引入混合写屏障的原因。

首先，当为栈上的指针赋值时，新的地址值大致有4种来源，即全新分配、栈上的指针、堆上的指针，以及全局数据段的指针。标记阶段全新分配的对象会被mallocgc（）函数自动标黑，所以不需要额外处理。函数栈帧里两个指针间的赋值不必应用写屏障，因为已扫描过的栈不存在隐藏指针，未扫描过的栈不需要追踪增量变动，后续扫描时会完整处理。至于堆和全局数据段，如果仅仅从它们那里把指针复制到栈上，也不会有问题，我们虽然不再重新扫描栈，但是对象可以通过堆或全局数据段里旧有的指针来保证其可达性。怕的就是我们把指针复制到栈上以后，堆或全局数据段里旧有的指针被擦除了，而且不再有其他的指针指向该对象。栈上复制过来的指针就变成了唯一指针，然而我们不再重新扫描栈，所以对象就被隐藏了。

至此，关键的问题就变成了，当堆或全局数据段中的指针被擦除之前，需要灰化它指向的对象。这样一来，把堆和全局数据段里的指针复制到栈上也不用经过插入写屏障了，旧有指针不被擦除，由旧有指针来保证可达性，旧有指针被擦除前，删除写屏障负责灰化其指向的对象。这就是Go引入删除写屏障的原因，缘于我们不希望对栈应用插入写屏障。

插入写屏障就比较好理解了，前提还是因为我们不会重新扫描堆和全局数据段。如果当前的栈还未被扫描，栈上就有可能存在白色的指针。如果goroutine把一个白色指针赋值给堆或全局数据段里一个黑色对象的某个字段，并且栈上的旧有指针在栈扫描之前被覆盖，则该对象就被成功地隐藏了。插入写屏障就是用来跟踪写入堆和全局数据段的指针，从而防止对象隐藏。

删除写屏障应对就像是一种极端条件，只有在即将被擦除的指针本身位于堆或全局数据段上，并且是指向某个位于堆上的对象的唯一指针，并且该指针曾经被复制到某个黑色的协程栈上时，删除写屏障才会真正发挥作用。实际实现的时候不必去跟踪或检测这些条件，只是宽泛地对堆和全局数据段里的指针擦除进行跟踪就行了。插入写屏障也不必真地去检测goroutine的栈是否为黑色，这样会造成一定程度的性能损失，宽泛地跟踪写入堆和全局数据段的所有指针就行了。

下面我们就用一段简单的示例代码，通过反编译的方法，来看一看写屏障是如何被应用的，示例代码如下：

```
//第 8 章/code_8_1.go
package main

var p * int

func main() {
    toStack(&p)
    toGlobal(&p)
    toUnknown(&p, &p)
}

//go:noinline
func toStack(i ** int) (o * int) {
    o = * i
    return
}

//go:noinline
func toGlobal(i ** int) {
    p = * i
}

//go:noinline
func toUnknown(a ** int, i ** int) {
    * a = * i
}
```

代码中的3个函数对应3种不同的场景，toStack（）函数把一个未知来源的指针复制到栈上，函数的参数i是个int类型的指针，未知来源指的是它所指向的int类型的指针可以在栈、堆及全局数据段等任何位置。相应地，toGlobal（）函数会把一个未知来源的指针赋值给全局数据段里的变量p，而toUnknown（）函数则是把一个未知来源的指针复制到未知的目的地。

我们先来反编译一下toStack（）函数，按照之前的分析，它应该不会应用写屏障。反编译得到的汇编代码如下：

```
$ go tool objdump -S -s '^main.toStack$' barrier.exe
TEXT main.toStack(SB) C:/gopath/src/fengyoulin.com/barrier/main.go
        o = *i
  0x493f20        488b442408        MOVQ 0x8(SP), AX
  0x493f25        488b00            MOVQ 0(AX), AX
        return
  0x493f28        4889442410        MOVQ AX, 0x10(SP)
  0x493f2d        c3                RET
```

只有简单的4条汇编指令，完成指针复制后就返回了，确实没有应用写屏障的痕迹。别着急，接下来反编译toGlobal（）函数，按道理它应该用到写屏障，反编译得到的汇编代码如下：

```
$ go tool objdump -S -s '^main.toGlobal$' barrier.exe
TEXT main.toGlobal(SB) C:/gopath/src/fengyoulin.com/barrier/main.go
func toGlobal(i **int) {
  0x493f40        65488b0c2528000000    MOVQ GS:0x28, CX
  0x493f49        488b8900000000        MOVQ 0(CX), CX
  0x493f50        483b6110              CMPQ 0x10(CX), SP
  0x493f54        763b                  JBE 0x493f91
  0x493f56        4883ec08              SUBQ $0x8, SP
  0x493f5a        48892c24              MOVQ BP, 0(SP)
  0x493f5e        488d2c24              LEAQ 0(SP), BP
        p = *i
  0x493f62        488b442410            MOVQ 0x10(SP), AX
  0x493f67        488b00                MOVQ 0(AX), AX
  0x493f6a        833d0f690f0000        CMPL $0x0, runtime.writeBarrier(SB)
  0x493f71        7510                  JNE 0x493f83
  0x493f73        4889059e300b00        MOVQ AX, main.p(SB)
}
  0x493f7a        488b2c24              MOVQ 0(SP), BP
  0x493f7e        4883c408              ADDQ $0x8, SP
  0x493f82        c3                    RET
        p = *i
  0x493f83        488d3d8e300b00        LEAQ main.p(SB), DI
  0x493f8a        e85118fdff            CALL runtime.gcWriteBarrier(SB)
  0x493f8f        ebe9                  JMP 0x493f7a
func toGlobal(i **int) {
  0x493f91        e82afbfcff            CALL runtime.morestack_noctxt(SB)
  0x493f96        eba8                  JMP main.toGlobal(SB)
```

汇编代码的开头和结尾是我们熟悉的栈增长代码，CMPL$0x0，runtime.writeBarrier（SB）这条指令是在检测写屏障有没有开启，后面的CALL runtime.gcWriteBarrier（SB）指令是在调用写屏障的处理函数。我们稍后会梳理写屏障处理函数的逻辑，toUnknown（）函数反编译后得到的汇编代码如下：

```
$ go tool objdump -S -s '^main.toUnknown$' barrier.exe
TEXT main.toUnknown(SB) C:/gopath/src/fengyoulin.com/barrier/main.go
func toUnknown(a ** int, i ** int) {
  0x493fa0        65488b0c2528000000    MOVQ GS:0x28, CX
  0x493fa9        488b8900000000        MOVQ 0(CX), CX
  0x493fb0        483b6110              CMPQ 0x10(CX), SP
  0x493fb4        7637                  JBE 0x493fed
  0x493fb6        4883ec08              SUBQ $0x8, SP
  0x493fba        48892c24              MOVQ BP, 0(SP)
  0x493fbe        488d2c24              LEAQ 0(SP), BP
        *a = *i
  0x493fc2        488b7c2410            MOVQ 0x10(SP), DI
  0x493fc7        8407                  TESTB AL, 0(DI)
  0x493fc9        488b442418            MOVQ 0x18(SP), AX
  0x493fce        488b00                MOVQ 0(AX), AX
  0x493fd1        833da8680f0000        CMPL $0x0, runtime.writeBarrier(SB)
  0x493fd8        750c                  JNE 0x493fe6
  0x493fda        488907                MOVQ AX, 0(DI)
}
  0x493fdd        488b2c24              MOVQ 0(SP), BP
  0x493fe1        4883c408              ADDQ $0x8, SP
  0x493fe5        c3                    RET
        *a = *i
  0x493fe6        e8f517fdff            CALL runtime.gcWriteBarrier(SB)
  0x493feb        ebf0                  JMP 0x493fdd
func toUnknown(a ** int, i ** int) {
  0x493fed        e8cefafcff            CALL runtime.morestack_noctxt(SB)
  0x493ff2        ebac                  JMP main.toUnknown(SB)
```

与toGlobal（）函数的代码结构基本相同，我们又看到了CALL runtime.gcWriteBarrier（SB）指令。总结一下反编译这3个函数得到的结论：把指针复制到栈上不需要写屏障，把指针赋值给全局数据段中的变量需要写屏障。把指针复制到一个未知的位置（可能是栈、堆或全局数据段），也需要写屏障，因为对栈上的指针应用写屏障并不会出错，不应用是为了提高性能，而堆和全局数据段则必须用写屏障才行，否则可能造成对象隐藏而被错误地释放。

最后，我们来看一下runtime.gcWriteBarrier（）函数的逻辑，这个函数是用汇编语言实现的。在梳理汇编代码之前，先来看一个数据结构，也就是被写屏障用作缓冲区的wbBuf结构，代码如下：

```
type wbBuf struct {
    next uintptr

    end  uintptr
    buf  [wbBufEntryPointers * wbBufEntries]uintptr
}
```

其中buf是个指针数组，wbBufEntryPointers和wbBufEntries这两个常量的值分别是2和256。因为是删除加插入的混合写屏障，所以每次会向缓冲区中写入两个指针，而缓冲区总共可供这样写入256次，写入256次后便会写满。next指向缓冲区可用的位置，每次写入时向后移动两个指针大小。end刚好指向缓冲区之后，可以理解为右开区间的坐标值，当next等于end时表示缓冲区满了。

接下来可以梳理gcWriteBarrier（）函数的汇编源码了，笔者在代码中添加了注释以便于理解，代码如下：

```
TEXT runtime·gcWriteBarrier<ABIInternal>(SB),NOSPLIT,$120
    //函数里会用到 R14 和 R13 这两个寄存器,先在栈上备份一下
    MOVQ    R14, 104(SP)
    MOVQ    R13, 112(SP)
    get_tls(R13)
    MOVQ    g(R13), R13
    MOVQ    g_m(R13), R13
    MOVQ    m_p(R13), R13
    MOVQ    (p_wbBuf+wbBuf_next)(R13), R14
    //R14 里存储的是 wbBuf.next 的值,将其增加 16 字节分配空间
    LEAQ    16(R14), R14
    MOVQ    R14, (p_wbBuf+wbBuf_next)(R13)
    CMPQ    R14, (p_wbBuf+wbBuf_end)(R13)
    //AX 里存储的是指针赋值等号右边的新值,写入 wbBuf.buf 中
    MOVQ    AX, -16(R14)
    //DI 里是赋值等号左边变量的地址,取出旧值并存入 R13 中
    MOVQ    (DI), R13
    //将 R13 里的旧值写入 wbBuf.buf
    MOVQ    R13, -8(R14)
    //前面的 CMPQ 指令用于判断 wbBuf.buf 是否已满,按需 flush
    JEQ     flush
ret:
    MOVQ    104(SP), R14
    MOVQ    112(SP), R13
    //把 AX 中的新值写入 DI 指向的位置,完成了指针赋值操作
    MOVQ    AX, (DI)
    RET

flush:
    //备份除 R14 和 R13 外的其他寄存器,wbBufFlush 函数可能会用到它们
    MOVQ    DI, 0(SP)
```

```
    MOVQ    AX, 8(SP)
    MOVQ    BX, 16(SP)
    MOVQ    CX, 24(SP)
    MOVQ    DX, 32(SP)
    MOVQ    SI, 40(SP)
    MOVQ    BP, 48(SP)
    MOVQ    R8, 56(SP)
    MOVQ    R9, 64(SP)
    MOVQ    R10, 72(SP)
    MOVQ    R11, 80(SP)
    MOVQ    R12, 88(SP)
    MOVQ    R15, 96(SP)

    //将 wbBuf.buf 中的指针冲刷到 GC 的工作队列中
    CALL    runtime·wbBufFlush(SB)
    //还原 CALL 之前备份的这些寄存器
    MOVQ    0(SP), DI
    MOVQ    8(SP), AX
    MOVQ    16(SP), BX
    MOVQ    24(SP), CX
    MOVQ    32(SP), DX
    MOVQ    40(SP), SI
    MOVQ    48(SP), BP
    MOVQ    56(SP), R8
    MOVQ    64(SP), R9
    MOVQ    72(SP), R10
    MOVQ    80(SP), R11
    MOVQ    88(SP), R12
    MOVQ    96(SP), R15
    JMP     ret
```

有兴趣的读者可以返回前面反编译3个函数的地方，看一看编译器是如何通过DI和AX这两个寄存器向gcWriteBarrier（）函数传递参数的，关于写屏障的探索就先到这里。

8.2.4 触发方式

Go的GC共有3种触发方式，第1种是被runtime初始化阶段创建的sysmon线程和forcegchelper协程发起，属于基于时间的周期性触发。第2种是被我们刚刚分析过的mallocgc（）函数发起的，触发条件是堆大小达到或超过了临界值。第3种是被开发者通过runtime.GC（）函数强制触发。通过查看这三处源码，发现内部会调用同一个GC启动函数，这就是runtime.gcStart（）函数，函数的原型如下：

```
func gcStart(trigger gcTrigger)
```

我们不打算展开分析这个函数的源码，而是来研究一下它的参数，也就是这个gcTrigger类型，它是一个结构体，具体的定义代码如下：

```
type gcTrigger struct {
    kind gcTriggerKind
    now  int64
    n    uint32
}
```

其中gcTriggerKind底层是个int类型，runtime定义了3个gcTriggerKind类型的常量，该常量的取值及其含义如表8-3所示。

表8-3 gcTriggerKind的取值及其含义

取　值	含　义
gcTriggerHeap	表示触发原因是因为堆的大小达到或超过了临界值，这个临界值是由 GC 控制器计算出来的
gcTriggerTime	表示触发原因是因为距上次 GC 运行已经超过了 forcegcperiod 这么多纳秒的时间，目前这个时间周期被定义为两分钟
gcTriggerCycle	主要用于强制执行 GC

gcTrigger类型提供了一个test（）方法，用于检测当前有没有达到GC触发条件，源代码如下：

```
func (t gcTrigger) test() bool {
    if !memstats.enablegc || panicking != 0 || gcphase != _GCoff {
        return false
    }
    switch t.kind {
    case gcTriggerHeap:
        return memstats.heap_live >= memstats.gc_trigger
    case gcTriggerTime:
        if gcpercent < 0 {
            return false
        }
        lastgc := int64(atomic.Load64(&memstats.last_gc_nanotime))
        return lastgc != 0 && t.now - lastgc > forcegcperiod
    case gcTriggerCycle:
        return int32(t.n - work.cycles) > 0
    }
    return true
}
```

1．gcTriggerHeap

在处理gcTriggerHeap这种类型时，memstats.heap_live是当前的堆大小，memstats.gc_trigger是控制器计算得到的临界值。临界值来源于上次标记的堆大小和gcpercent的值，后者可以通过环境变量GOGC进行设置，表示当堆增长超过百分之多少后触发GC，参考的堆起始大小就是上次标记终止时标记的大小，控制器会在每次标记终止时更新临界值。那么第一次触发参考哪个值呢？因为没有上一次可供参考，所以第一次触发的临界值被预置为4MB，在gcinit（）函数里进行初始化。

mallocgc（）函数中会发起GC的相关代码在8.1.6节已经讲解过了，如果mallocgc（）函数分配了较大空间，则shouldhelpgc的值就是true，然后就会创建一个gcTriggerHeap类型的gcTrigger，通过test检测当前堆大小是否达到或超过临界值，按需调用gcStart（）函数发起GC。

2．gcTriggerTime

当处理gcTriggerTime类型时，memstats.last_gc_nanotime以纳秒为单位记录了上次GC执行的时刻，gcTrigger的now字段存储的是想要发起GC时的时间戳，两者之差如果超过forcegcperiod就会触发GC。至于gcTriggerCycle这种类型，首先要说明一下work.cycles，它会随着每轮GC自增，也就等于记录了当前执行到第几轮。runtime.GC（）函数会先读取work.cycles的值，然后把这个值加一作为n来构造一个gcTriggerCycle类型的gcTrigger，把它作为参数来调用gcStart（）函数。如果在这个过程中，下一轮GC已经在别处被触发，则work.cycles的值就会等于甚至大于t.n的值，t.test（）函数

就会返回false，gcStart（）函数也就随之返回，不会再重复执行了。

实际发起周期性GC的是forcegchelper协程，它是在runtime的init（）函数中被创建的，入口函数的代码如下：

```
func forcegchelper() {
    forcegc.g = getg()
    lockInit(&forcegc.lock, lockRankForcegc)
    for {
        lock(&forcegc.lock)
        if forcegc.idle != 0 {
            throw("forcegc: phase error")
        }
        atomic.Store(&forcegc.idle, 1)
        goparkunlock(&forcegc.lock, waitReasonForceGCIdle, traceEvGoBlock, 1)
        if debug.gctrace > 0 {
            println("GC forced")
        }
        gcStart(gcTrigger{kind: gcTriggerTime, now: nanotime()})
    }
}
```

它开始运行之后做的第一件事就是获取自身的g指针并赋值给forcegc.g，这样一来sysmon线程就可以通过forcegc.g来调度当前goroutine了。接下来它初始化了互斥锁forcegc.lock，然后就进入了一个无限循环。

每轮循环中先获得forcegc.lock锁，然后将forcegc.idle置为1，这样sysmon就能知道forcegchelper协程当前并没有在运行。设置完idle之后，通过goparkunlock（）函数来挂起自己，同时解锁forcegc.lock，此后便等待sysmon调度。得到调度执行后，调用gcStart（）函数发起一轮GC，触发类型为gcTriggerTime。

相应地，sysmon线程中调度forcegchelper的代码如下：

```
if t := (gcTrigger{kind: gcTriggerTime, now: now}); t.test() && atomic.Load(&forcegc.idle) !
= 0 {
    lock(&forcegc.lock)
    forcegc.idle = 0
    var list gList
    list.push(forcegc.g)
    injectglist(&list)
    unlock(&forcegc.lock)
}
```

先用当前时间创建一个类型为gcTriggerTime的gcTrigger，然后调用test方法来判断当前时间是否已经满足GC触发条件。如果达到GC触发条件且forcegchelper处于idle状态，就把forcegc.g添加到runq中。

3．gcTriggerCycle

至于用户可以强制执行GC的runtime.GC（）函数，关键代码如下：

```
n := atomic.Load(&work.cycles)
gcWaitOnMark(n)
gcStart(gcTrigger{kind: gcTriggerCycle, n: n + 1})
gcWaitOnMark(n + 1)
for atomic.Load(&work.cycles) == n+1 && sweepone() != ^uintptr(0) {
    sweep.nbgsweep++
    Gosched()
}
```

笔者略去了少量不太重要的逻辑，首先从work.cycles获取当前GC的周期数并存于局部变量n中，然后通过gcWaitOnMark等待第n轮标记结束，然后调用gcStart（）函数发起第n+1轮GC，并等待其标记结束，最后用一个for循环来完成第n+1轮的清扫工作。

关于GC触发方式的分析就到这里，感兴趣的读者可以深入研究一下gcStart（）等函数的源码。

8.2.5 GC Worker

GC的标记阶段会创建一组后台工作协程，还会启用assist机制让一般的协程在分配内存时辅助完成一部分标记工作。GC与一般的业务协程是并发运行的，为了避免GC过多地占用CPU，runtime中的常量gcGoalUtilization将最大使用率限制为30%，常量gcBackgroundUtilization将后台工作协程的最大CPU使用率限制为25%，两者之差（5%）是留给辅助GC的。

我们先来看一看后台工作协程的这个25%是如何实现的。这组协程是在哪里被创建的呢？是由gcBgMarkStartWorkers（）函数创建的，该函数的代码如下：

```
func gcBgMarkStartWorkers() {
    for gcBgMarkWorkerCount < gomaxprocs {
        go gcBgMarkWorker()
        notetsleepg(&work.bgMarkReady, -1)
        noteclear(&work.bgMarkReady)
        gcBgMarkWorkerCount++
    }
}
```

该函数通过一个for循环，创建gomaxprocs个工作协程，也就是保证每个P都能分配到一个。因为gomaxprocs可能会变化，所以用变量gcBgMarkWorkerCount记录了工作协程的数量，后续如果gomaxprocs被增大，下次调用该函数时就能把工作协程补齐。gomaxprocs减小，不需要销毁对应的工作协程，可以留待后续gomaxprocs再次被增大时复用。gcStart（）函数每次都会调用该函数，也就是每轮GC开始时都会检测后台协程数量，并按需补齐到gomaxprocs个。

感兴趣的读者可以阅读一下gcBgMarkWorker的源码，主要逻辑就是在一个for循环中执行标记逻辑并检测分布式标记是否已完成。在每轮循环的最开始，它会先通过gopark挂起自己，并且把自己push到gcBgMarkWorkerPool中，我们感兴趣的是这些后台协程是如何得到调度的。进一步跟踪gcBgMarkWorkerPool的pop操作，发现有两个地方会调用pop，一个是在gcControllerState类型的findRunnableGCWorker（）方法中，另一个是在调度循环的findrunnable（）函数中。到这里有必要介绍一下后台工作协程的几种不同工作模式。

1. GC Worker的工作模式

在runtime中为GC工作协程定义了3种工作模式，分别有与之对应的常量，如表8-4所示。

表8-4 GC工作协程的3种工作模式及其含义

工作模式	含　义
gcMarkWorkerDedicatedMode	表示该工作协程所在的P专门用来运行这个GC工作协程，并且应该在不被抢占的情况下运行。实际实现的时候，dedicated模式的worker先以可被抢占的模式运行，首次检测到抢占标识时，把本地runq中的所有g都放入全局runq，后续以不被抢占的模式运行，这样可以使本地runq中原有的任务尽量减少延迟
gcMarkWorkerFractionalMode	这种模式的worker主要因为gomaxprocs乘以gcBackgroundUtilizition的结果可能不是整数，不能用整数个dedicated模式的worker实现，剩余的小数部分就由fractional模式的worker来负责
gcMarkWorkerIdleMode	表示当前P没有其他任务可做，处于空闲状态，顺便来执行GC

在以上3种模式中，idle模式的worker由findrunnable（）函数负责调度，当findrunnable（）函数找不到其他可运行的g时，就会从gcBgMarkWorkerPool中pop出一个后台工作协程的g，然后把当前P的gcMarkWorkerMode设置成gcMarkWorkerIdleMode，并返回工作协程的g。

我们更关心的是dedicated和fractional这两种模式是如何调度的，跟踪findRunnableGCWorker（）方法，发现整个runtime中只有schedule（）函数会调用它。第6章我们分析过schedule（）函数的源码，它就是调度循环的具体实现，它会先调用findRunnableGCWorker（）方法来尝试执行GC后台任务，然后才是从runq中取常规g来执行，所以关键点就在于findRunnableGCWorker（）方法了，代码如下：

```
func (c *gcControllerState) findRunnableGCWorker(_p_ *p) *g {
    if gcBlackenEnabled == 0 {
        throw("gcControllerState.findRunnable: blackening not enabled")
    }

    if !gcMarkWorkAvailable(_p_) {
        return nil
    }

    node := (*gcBgMarkWorkerNode)(gcBgMarkWorkerPool.pop())
    if node == nil {
        return nil
    }

    decIfPositive := func(ptr *int64) bool {
        for {
            v := atomic.Loadint64(ptr)
            if v <= 0 {
                return false
            }

            if atomic.Cas64((*uint64)(unsafe.Pointer(ptr)), uint64(v), uint64(v-1)) {
                return true
            }
        }
    }

    if decIfPositive(&c.dedicatedMarkWorkersNeeded) {
        _p_.gcMarkWorkerMode = gcMarkWorkerDedicatedMode
    } else if c.fractionalUtilizationGoal == 0 {
        gcBgMarkWorkerPool.push(&node.node)
        return nil
    } else {
        delta := nanotime() - gcController.markStartTime
        if delta > 0 && float64(_p_.gcFractionalMarkTime)/float64(delta) > c.
fractionalUtilizationGoal {
            gcBgMarkWorkerPool.push(&node.node)
            return nil
        }
        _p_.gcMarkWorkerMode = gcMarkWorkerFractionalMode
    }

    gp := node.gp.ptr()
    casgstatus(gp, _Gwaiting, _Grunnable)
    if trace.enabled {
        traceGoUnpark(gp, 0)
    }
    return gp
}
```

其中gcMarkWorkAvailable（）函数会检查P本地的GC工作队列和全局工作队列，如果已经没有任

务需要处理，就直接返回nil。否则就从gcBgMarkWorkerPool中pop出一个工作协程，如果为nil，则直接返回nil。decIfPositive（）函数基于原子指令CAS实现对一个正整数减一的操作，被用于分配dedicated模式的worker，优先返回dedicated模式的worker。之后再根据c.fractionalUtilizationGoal来调度fractional模式的worker。c.dedicatedMarkWorkersNeeded和c.fractionalUtilizationGoal都是在本轮GC开始时计算出来的，分别表示dedicated、fractional模式的worker会占用多少个P。delta是从本轮开始标记已经过去的时间，用当前P的gcFractionalMarkTime除以delta得到的是当前P运行fractional worker所花时间占总时间的百分比，这样可以把fractional worker分摊到所有P上去执行，尽量使每个P都均衡地分担任务。如果当前P的执行时间已经超过目标值，就返回nil。

最后，我们再来看一下辅助GC，辅助GC虽然不属于GC Worker，但是做的工作也是并发标记工作，所以也需要了解一下。

2．辅助**GC**

辅助GC是通过gcAssistAlloc（）函数完成的，整个runtime中只有一个地方会调用该函数，也就是在mallocgc（）函数中。8.1.6节我们已经知道，g的gcAssistBytes字段记录了当前协程通过辅助GC积累了多少字节的信用值，就像信用卡的额度一样，如果mallocgc（）函数要分配的内存大小在这个信用值的范围内，就不用执行辅助GC，否则就要调用gcAssistAlloc（）函数来执行一部分辅助工作。信用值的存取模型如图8-30所示。

gcAssistAlloc（）函数里有两段代码比较有价值，我们来分析一下。用来计算负债额度和扫描工作量的代码如下：

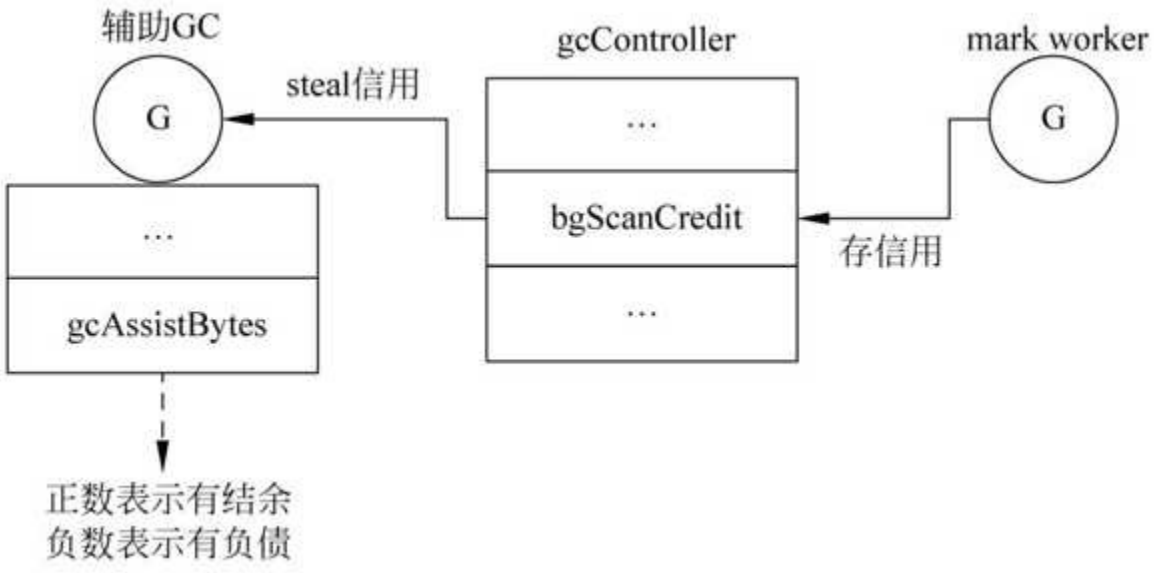

图8-30 辅助GC信用值存取模型

```
assistWorkPerByte := float64frombits(atomic.Load64(&gcController.assistWorkPerByte))
assistBytesPerWork := float64frombits(atomic.Load64(&gcController.assistBytesPerWork))
debtBytes := -gp.gcAssistBytes
scanWork := int64(assistWorkPerByte * float64(debtBytes))
if scanWork < gcOverAssistWork {
    scanWork = gcOverAssistWork
    debtBytes = int64(assistBytesPerWork * float64(scanWork))
}
```

assistWorkPerByte实际上是个float64，表示每分配一字节内存空间应该相应地做多少扫描工作，gcController把它当成uint64进行原子性存取。assistBytesPerWork可以认为是前者的倒数，理解成完成一字节的扫描工作后可以分配多大的内存空间。这两者表示的都是比率，实际的内存分配和扫描

不可能都是一字节一字节的。它们都是在每轮GC开始时被计算好，并且会随着堆扫描的进度一起更新。在mallocgc（）函数中，因为gp.gcAssistBytes＜0，所以才调用了gcAssistAlloc，由此负债额度就是gp.gcAssistBytes的绝对值。预期需要扫描的大小等于debtBytes乘以assistWorkPerByte，如果得到的结果小于gcOverAssistWork，就取gcOverAssistWork的值，该值目前被定义为64KB。也就是至少扫描64KB空间，这样可以避免多次执行而实际的产出过少，就像线程切换频繁造成整体的吞吐量低下。如果把scanWork对齐到gcOverAssistWork，就需要乘以assistBytesPerWork重新计算debtBytes。得到的debtBytes会比本次分配的实际需要大一些，但是没有关系，后面它会被累加到gp.gcAssistBytes中，多出来的部分可供下次分配使用。

还有一段代码用来从gcController窃取扫描信用额度，后台工作协程执行扫描任务积累的信用值会被累加到gcController的bgScanCredit字段，如果该值的大小足够抵消本次的scanWork，则当前协程就不用实际去执行扫描任务了。信用窃取的主要代码如下：

```
bgScanCredit := atomic.Loadint64(&gcController.bgScanCredit)
stolen := int64(0)
if bgScanCredit > 0 {
    if bgScanCredit < scanWork {
        stolen = bgScanCredit
        gp.gcAssistBytes += 1 + int64(assistBytesPerWork * float64(stolen))
    } else {
        stolen = scanWork
        gp.gcAssistBytes += debtBytes
    }
    atomic.Xaddint64(&gcController.bgScanCredit, -stolen)

    scanWork -= stolen

    if scanWork == 0 {
        return
    }
}
```

根据bgScanCredit与scanWork的大小比较，决定是窃取全部还是窃取部分，并且根据实际窃取的大小更新gp.gcAssistBytes，然后从bgScanCredit和scanWork中分别减去窃取的大小。最后，如果scanWork等于0，就不用执行后续的扫描工作了。实现辅助GC主要是为了避免程序过于频繁地分配内存，造成后台工作协程忙不过来，如果程序的内存分配动作不是很频繁，实际上可能根本不会真正去执行辅助扫描。

本节关于GC Worker的分析就到这里，感兴趣的读者可阅读源码了解更多细节。

8.2.6 gctrace

讲了较多干巴巴的理论和代码，本节就来点相关实践，验证一下前面的分析探索。Go的runtime对追踪和调试支持得比较好，例如最常用的pprof，在查找内存泄漏及性能瓶颈时非常方便。垃圾回收方面可以通过GODEBUG环境变量开启gctrace，程序运行时就会输出每轮GC的开始时间、耗时，以及标记终止时的堆大小和标记大小等信息。

接下来我们就用几段实际的代码来演示一下，首先来个简单一点的，代码如下：

```
//第 8 章/code_8_2.go
var p * int64

func main() {
    for i := 0; i < 1000000; i++{
        p = new(int64)
    }
    time.Sleep(time.Second)
}
```

用go build命令构建上述代码，然后通过GODEBUG环境变量指定gctrace＝1来运行可执行文件，得到输出如下：

```
$ GODEBUG='gctrace=1' ./gc_trace.exe
gc 1 @0.022s 0%: 0+0.92+0 ms clock, 0+0/0.92/0+0 ms cpu, 4->4->0 MB, 5 MB goal, 8 P
gc 2 @0.035s 0%: 0+0.99+0 ms clock, 0+0.64/0.64/1.6+0 ms cpu, 4->4->0 MB, 5 MB goal,
8 P
```

输出的这两行日志包含较多信息，我们一项一项来梳理。拿第一条日志来分析，开头处gc 1中的数字1是序号，表示这是当前进程第一次GC。接下来的@0.022s表示GC开始的时刻，也就是程序开始执行的22ms后。后面的0%表示GC占用CPU的比例，0+0.92+0分别是清扫终止、并发标记和标记终止这3个阶段耗费的时间，以毫秒为单位。后面的0+0/0.92/0+0进一步细化地给出了清扫终止、辅助GC、dedicated加fractional标记、idle标记，以及标记终止这5项所耗费的CPU时间。4->4->0 MB对应标记开始时堆的大小、标记结束时堆的大小和标记结束时实际标记的空间大小。5 MB goal表示预期标记结束时的堆大小。8 P是本轮GC时runtime中P的数量。

初次触发GC的堆大小的临界值是4MB，这与之前的代码分析一致。因为首次标记后存活的堆大小是0MB，所以临界值保持在4MB没有升高。我们的代码分配了100万个int64，总计消耗8MB的堆空间，所以总共发生了两次GC。

我们稍微改动一下测试代码，把循环次数由100万次改成1000万次，这样一来GC次数就变多了，在笔者的计算机上发生了19次。虽然次数变多了，但是标记前后的堆大小和实际标记的大小一直是4->4->0MB，goal也一直保持在5MB。我们可以试着通过GOGC环境变量把gcpercent改得大一些。gcpercent的默认值是100，意味着堆大小比上次标记大小增长了一倍时触发下次GC。我们把它改成300，也就是增长三倍时触发下次GC。得到的输出如下：

```
$ GODEBUG='gctrace=1' GOGC=300 ./gc_trace.exe
gc 1 @0.049s 0%: 0+0.99+0 ms clock, 0+0/0/0+0 ms cpu, 12->12->0 MB, 13 MB goal, 8 P
gc 2 @0.091s 0%: 0+1.9+0 ms clock, 0+0.99/0.99/1.9+0 ms cpu, 12->12->0 MB, 13 MB
goal, 8 P
gc 3 @0.144s 0%: 0+0.54+0 ms clock, 0+0/1.0/0+0 ms cpu, 12->12->0 MB, 13 MB goal, 8 P
gc 4 @0.195s 0%: 0+1.2+0 ms clock, 0+1.0/0.27/0.27+0 ms cpu, 12->12->0 MB, 13 MB
goal, 8 P
gc 5 @0.254s 0%: 0+0.57+0.40 ms clock, 0+0/0.57/0.57+3.2 ms cpu, 12->12->0 MB, 13 MB
goal, 8 P
gc 6 @0.310s 0%: 0+0.70+0 ms clock, 0+0/0/0+0 ms cpu, 12->12->0 MB, 13 MB goal, 8 P
```

原来的4MB变成了12MB，也就是把首次触发的临界值调高了三倍。每次标记之后实际标记的大小都是0MB，因为我们的代码把每次循环分配的int64的地址都赋给了同一个包级别指针，这样后面的指针就会覆盖前面的指针，最后一次之前的指针都会变成不可达。我们再稍微修改一下代码逻辑，用一个包级别的指针数组使分配的int64全都可达，代码如下：

```
//第 8 章/code_8_3.go
var pa [10000000] * int64

func main() {
    for i := 0 ; i < 10000000; i++{
        pa[i] = new(int64)
    }
    time.Sleep(time.Second)
}
```

包级别变量pa是个大小为1000万的int64指针数组，我们循环分配1000万个int64，通过pa数组保证它们都可达。运行得到的输出如下：

```
$ GODEBUG = 'gctrace = 1' ./gc_trace.exe
gc 1 @0.026s 17 % : 0 + 23 + 0 ms clock, 0 + 22/46/113 + 0 ms cpu, 4 -> 4 -> 3 MB, 5 MB goal, 8 P
gc 2 @0.062s 15 % : 0 + 10 + 0 ms clock, 0 + 9.4/10/56 + 0 ms cpu, 7 -> 7 -> 7 MB, 8 MB goal, 8 P
gc 3 @0.095s 12 % : 0 + 15 + 0 ms clock, 0 + 5.0/17/77 + 0 ms cpu, 14 -> 14 -> 14 MB, 15 MB goal,
8 P
gc 4 @0.158s 12 % : 0 + 25 + 0 ms clock, 0 + 16/48/120 + 0 ms cpu, 29 -> 29 -> 29 MB, 30 MB goal,
8 P
gc 5 @0.288s 11 % : 0 + 51 + 0 ms clock, 0 + 34/93/263 + 0 ms cpu, 58 -> 59 -> 59 MB, 59 MB goal,
8 P
```

这次共发生了5次GC，可以看到每次标记开始时的堆大小相对于上次实际标记的大小基本上成二倍关系，并且GC占用CPU的百分比显著增加。不过由于所有分配均可达，造成每次标记终止时的堆大小和实际标记大小基本相等。我们再修改一下代码，让一半分配可达，另一半不可达，具体的代码如下：

```
//第 8 章/code_8_4.go
var pa [10000000] * int64

func main() {
    for i := 0; i < 10000000; i++{
        pa[i] = new(int64)
        pa[i] = new(int64)
    }
    time.Sleep(time.Second)
}
```

我们在每轮循环中连续分配两个int64，地址都赋给pa[i]，这样后面的指针就会覆盖掉前面的指针，从而实现我们想要的一半可达的效果。这次运行得到的输出如下：

```
$ GODEBUG=gctrace=1 ./gc_trace.exe
gc 1 @0.049s 21%: 0+72+0 ms clock, 0+70/141/348+0 ms cpu, 4->4->3 MB, 5 MB goal, 8 P
gc 2 @0.154s 18%: 0.99+18+0 ms clock, 7.9+17/18/100+0 ms cpu, 7->7->7 MB, 8 MB goal,
8 P
gc 3 @0.238s 14%: 0.99+23+0 ms clock, 7.9+22/23/139+0 ms cpu, 14->14->14 MB, 15 MB
goal, 8 P
gc 4 @0.369s 11%: 0+27+0 ms clock, 0+9.8/52/131+0 ms cpu, 27->28->28 MB, 28 MB goal,
8 P
gc 5 @0.581s 9%: 0+65+0 ms clock, 0+18/113/336+0 ms cpu, 55->56->56 MB, 57 MB goal,
8 P
gc 6 @0.985s 7%: 0+52+0 ms clock, 0+16/89/281+0 ms cpu, 110->113->113 MB, 113 MB
goal, 8 P
```

堆的大小确实增加了，但是实际标记的大小还是和堆大小基本一致，这是怎么回事呢？对了，我们忘了tiny allocator，小于16字节且noscan的内存块会用tiny分配器进行组合分配。这里相邻的两次int64分配肯定被tiny allocator组合分配了，所以只要其中一个还是可达的，整个内存块就会被标记。我们可以再修改一下代码来绕过tiny分配器，可选分配scan型内存，或分配不小于16字节的内存块，我们选择前者，改成分配*int64，代码如下：

```
//第8章/code_8_5.go
var pa [10000000]**int64

func main() {
    for i := 0; i < 10000000; i++{
        pa[i] = new(*int64)
        pa[i] = new(*int64)
    }
    time.Sleep(time.Second)
}
```

这次不是noscan分配了，所以不能使用tiny allocator。实际运行后的输出如下：

```
$ GODEBUG=gctrace=1 ./gc_trace.exe
gc 1 @0.030s 19%: 0.21+72+0 ms clock, 1.6+20/142/347+0 ms cpu, 4->5->2 MB, 5 MB
goal, 8 P
gc 2 @0.132s 17%: 0+25+0 ms clock, 0+10/48/122+0 ms cpu, 5->5->4 MB, 6 MB goal, 8 P
gc 3 @0.198s 15%: 0+36+0.13 ms clock, 0+9.9/68/176+1.1 ms cpu, 8->8->6 MB, 9 MB
goal, 8 P
gc 4 @0.300s 15%: 0.99+48+0 ms clock, 7.9+23/94/245+0 ms cpu, 12->12->9 MB, 13 MB
goal, 8 P
gc 5 @0.443s 14%: 0+64+0 ms clock, 0+34/128/321+0 ms cpu, 18->19->14 MB, 19 MB goal,
8 P
gc 6 @0.643s 13%: 0+87+0 ms clock, 0+52/173/431+0 ms cpu, 28->29->21 MB, 29 MB goal,
8 P
gc 7 @0.911s 13%: 0+112+0 ms clock, 0+59/223/559+0 ms cpu, 42->44->33 MB, 43 MB
goal, 8 P
gc 8 @1.234s 12%: 0+76+0 ms clock, 0+40/151/388+0 ms cpu, 64->65->49 MB, 66 MB goal,
8 P
gc 9 @1.500s 12%: 0+119+0 ms clock, 0+66/238/593+0 ms cpu, 96->99->75 MB, 98 MB
goal, 8 P
```

这次标记终止时的堆大小和实际标记大小终于不相等了，也就是有可回收的空间了。关于gctrace的探索就到这里，大家可以设计更有意思的测试代码进行实验。

使用Go提供的trace包结合trace工具，还能以图形化的形式更直观地展示各种追踪数据，其中就包含与堆和GC相关的信息。我们再来简单地看一下，准备的示例代码如下：

```
//第 8 章/code_8_6.go
var pa [10000000] * int64

func main() {
    if err : = trace.Start(os.Stdout); err != nil {
        log.Fatalln(err)
    }
    defer trace.Stop()
    for i : = 0 ; i < 10000000; i++{
        pa[i] = new(int64)
        pa[i] = new(int64)
    }
}
```

代码会把trace数据输出到标准输出，我们需要收集这些数据以备进一步分析，所以运行可执行文件时要把标准输出重定向到一个文件，命令如下：

```
$ ./gc_trace.exe > out.dat
```

然后使用trace工具来分析out.dat文件，命令如下：

```
$ go tool trace out.dat
```

该命令会自动打开一个浏览器窗口，在打开的页面中单击View trace链接，然后就能看到如图8-31所示的图形界面了。

可以看到在程序运行的整个生命周期中堆的大小变化和GC运行的时段，以及各个P在不同时段分别在执行什么。笔者的计算机是8核CPU，所以可以看到8个P中有两个在执行dedicated工作协程，还可以看到有的P在某个时段执行了辅助GC等。

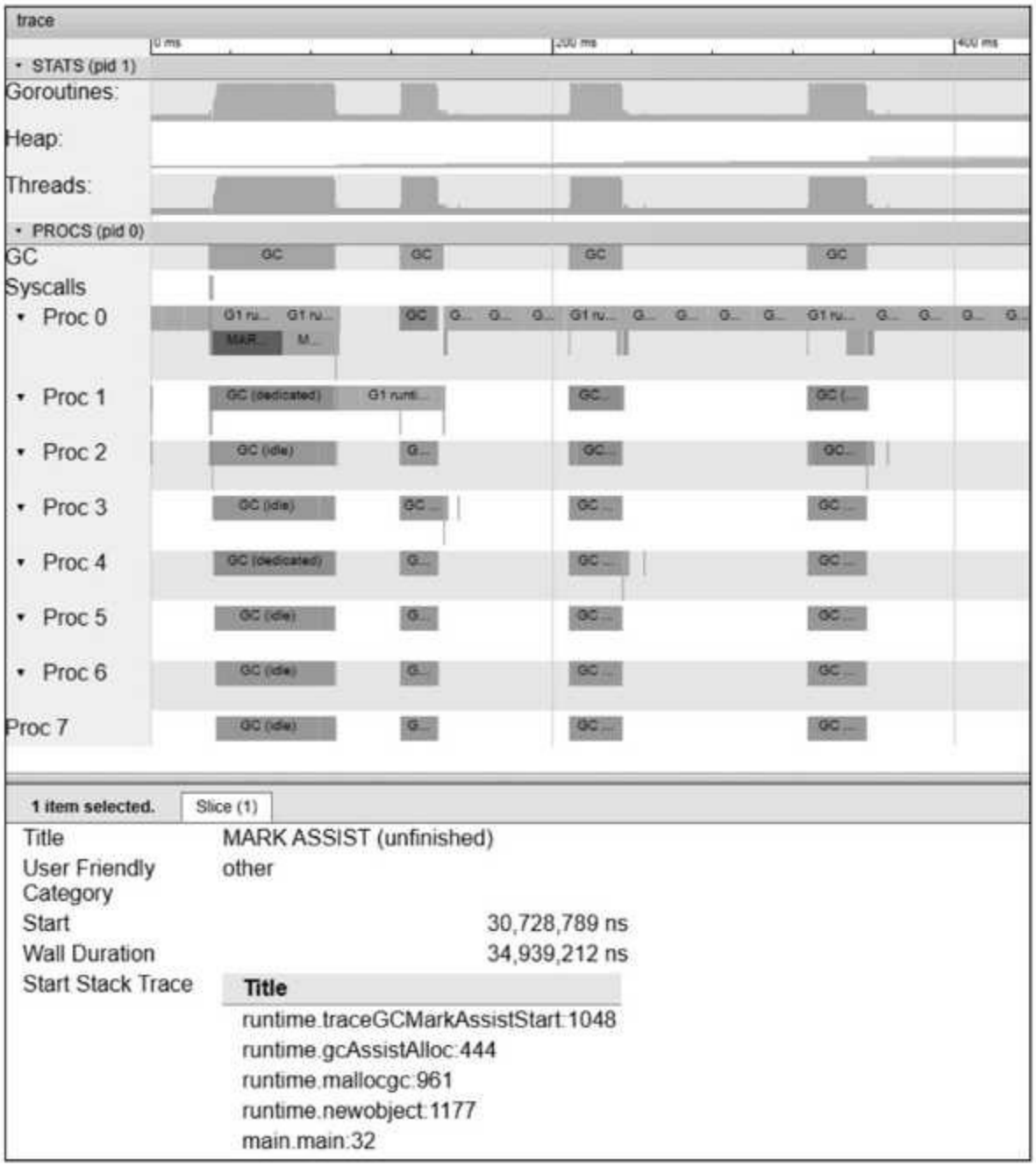

图8-31 trace命令图形界面

8.3 本章小结

本章主要探索了Go的堆内存管理，写作时主要参考了Go 1.16版及之前几个版本的runtime。前半部分以内存分配为主线，首先了解了借鉴自tcmalloc的基于sizeclasses的空闲链表，然后重点分析了最关键的基于arena和span的内存空间管理，以及用于管理和本地缓存mspan的mcentral和mcache结构，最后梳理了mallocgc（）函数的主要流程和3种内存分配策略。后半部分探索了垃圾回收，在了解了GC的大致流程之后，重点分析了GC root、三色抽象、写屏障等几个关键概念。最后介绍了gctrace的用法，为大家进一步探索GC提供了一个思路，更多精彩发现期待大家动手探索。

第9章
栈

现代的计算机组成基本上是基于栈的，从单片机到服务器多核心处理器，都是按照栈的思想来设计的，各种常用的编程语言也是如此。从Go语言的角度来看，goroutine的栈以栈帧的形式提供了函数局部变量的存储空间，又为函数调用时参数和返回值的传递提供了载体。考虑到函数调用与返回本身类似于入栈出栈的操作，因此栈是最适合的数据结构。下面，我们就来看一看runtime是如何管理goroutine的栈的。

9.1 栈分配

栈分配要研究的是goroutine的栈是何时被分配的，以及是怎样进行分配的。首先，要说何时被分配，当然是在goroutine被创建的时候。在第6章分析goroutine创建过程的时候，我们知道newproc1（）函数会先尝试通过调用gfget（）函数获取一个空闲的g，如果无法获取，就调用malg来分配一个全新的g。

gfget（）函数从空闲链表中获取的g可能带有栈，也可能不带栈，因为goroutine退出运行的时候，如果栈的大小不等于初始大小（增长过），就会被释放，因此，gfget（）函数需要检测得到的g有没有栈，并为不带栈的g分配一个初始的栈。至于malg（）函数，因为是全新分配的g，所以肯定需要为它分配一个栈。

通过runtime源码可以得知，gfget（）函数和malg（）函数中分配的栈大小都是2KB。至于具体的栈空间分配工作，是由stackalloc（）函数来完成的。分配细节还得从runtime的初始化说起。

9.1.1 栈分配初始化

经过第6章的分析，我们已经知道整个初始化过程由schedinit（）函数负责，其中与栈相关的初始化是通过stackinit（）函数完成的。stackinit（）函数会初始化两个用于栈分配的全局对象，一个是栈缓冲池stackpool，另一个是专门用来分配大栈的stackLarge。其中stackpool的定义代码如下：

```
var stackpool [_NumStackOrders]struct {
    item stackpoolItem
    //省略掉用于内存对齐的填充空间
}
```

在Linux环境下，_NumStackOrders的值为4，也就是说stackpool实际上是一个长度为4的数组。数组元素类型是一个结构体，结构体中包含一个stackpoolItem类型的item字段和用于内存对齐的填充空间。填充空间的作用是把整个结构体的大小对齐到平台Cache Line的大小，以便最大限度地优化存取速度，因为与逻辑不相关，这里就省略掉了。stackpoolItem结构的定义代码如下：

```
//go:notinheap
type stackpoolItem struct {
    mu   mutex
    span mSpanList
}
```

其中mSpanList是一个由mspan构成的双向链表，mutex用来保护这个链表，真正的栈内存由链表中的mspan来提供。stackpool的结构如图9-1所示。

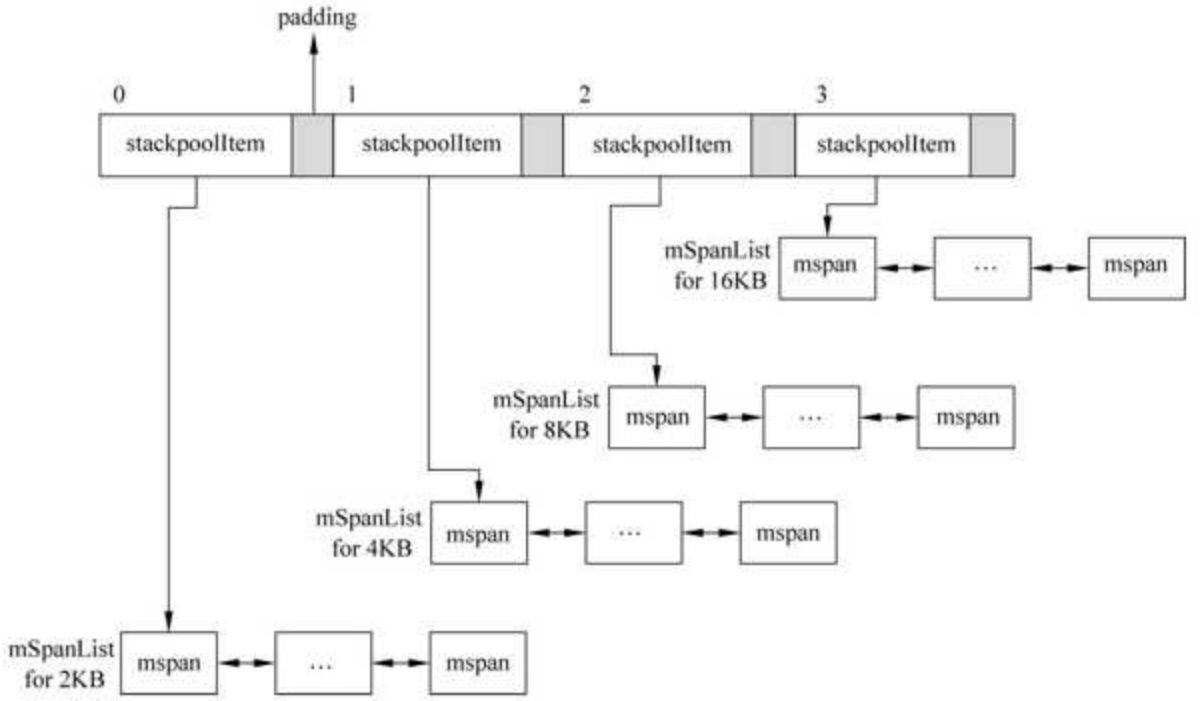

图9-1 stackpool结构示意图

stackpool数组的4个链表分别用来分配大小为2KB、4KB、8KB和16KB的栈，更大的栈空间由stackLarge来分配。stackLarge的定义代码如下：

```
var stackLarge struct {
    lock mutex
    free [heapAddrBits - pageShift]mSpanList
}
```

在amd64架构的Linux环境下，heapAddrBits的值是48，pageShift的值是13，所以free字段就是个长度为25的mSpanList数组。下标为0的链表对应_PageSize，用来分配8KB的空间，后续依次翻倍，如图9-2所示。

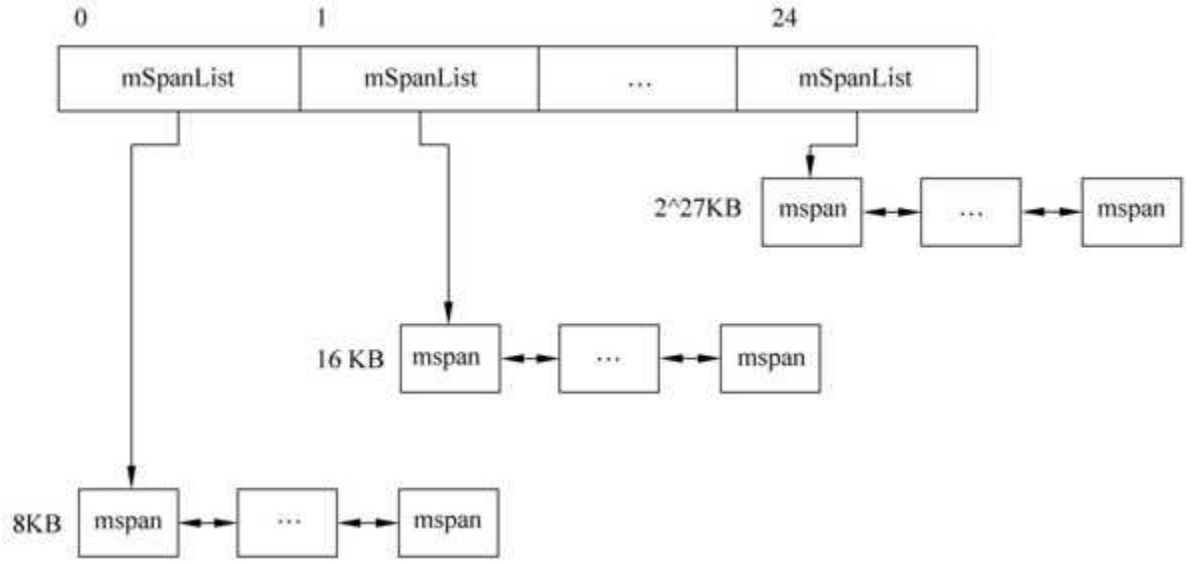

图9-2 stackLarge结构示意图

由于在实际运行中对大于16KB的栈需求较少，所以这些针对不同大小的链表共用一把锁就可以了。像stackpool则不然，因为使用频率较高，所以要为每个链表配一把锁。

stackinit（）函数只是校验了_StackCacheSize必须被定义为_PageSize的整倍数，并把stackpool和stackLarge中的链表都初始化为空链表，函数的代码如下：

```
func stackinit() {
    if _StackCacheSize& _PageMask != 0 {
        throw("cache size must be a multiple of page size")
    }
    for i := range stackpool {
        stackpool[i].item.span.init()
        lockInit(&stackpool[i].item.mu, lockRankStackpool)
    }
    for i := range stackLarge.free {
        stackLarge.free[i].init()
        lockInit(&stackLarge.lock, lockRankStackLarge)
    }
}
```

目前_StackCacheSize被定义为32768，而_PageSize则是8192。接下来可以看一看stackalloc（）函数是如何分配空间的。

9.1.2 栈分配逻辑

负责分配栈空间的stackalloc（）函数的原型如下：

```
func stackalloc(n uint32) stack
```

参数n表示要分配的栈空间的大小，它必须是2的幂。返回的stack结构用来表示分配的栈空间，hi字段是高地址，也就是栈空间的上界，lo表示空间下界，代码如下：

```
type stack struct {
    lo uintptr
    hi uintptr
}
```

stackalloc（）函数内部会对小于32KB的分配和32KB及以上的分配区分处理，我们先来看一下小于32KB时的处理逻辑。

1．小于**32KB**的栈分配

由于参数n必须是2的幂，所以也就是针对16KB及以下的分配。主要的处理逻辑的代码如下：

```
order := uint8(0)
n2 := n
for n2 > _FixedStack {
    order++
    n2 >>= 1
}
var x gclinkptr
if stackNoCache != 0 || thisg.m.p == 0 || thisg.m.preemptoff != "" {
    lock(&stackpool[order].item.mu)
    x = stackpoolalloc(order)
    unlock(&stackpool[order].item.mu)
} else {
    c := thisg.m.p.ptr().mcache
    x = c.stackcache[order].list
    if x.ptr() == nil {
        stackcacherefill(c, order)
        x = c.stackcache[order].list
    }
    c.stackcache[order].list = x.ptr().next
    c.stackcache[order].size -= uintptr(n)
}
v = unsafe.Pointer(x)
```

开始的for循环让order＝log2（n/_FixedStack），在Linux上，_FixedStack被定义为2KB，所以参数n＝2KB时order为0，n＝4KB时order为1，以此类推，8KB对应2，16KB对应3。在后续的分配过程中，order对应stackcache和stackpool数组的下标。

接下来的if语句会优先使用当前P的mcache中的stackcache进行分配，它是个数组，大小与stackpool相同，也是_NumStackOrders。实际上就是stackpool的一个本地缓存，不用加锁，效率更高。

不过，有几种情况不能使用stackcache，stackNoCache不为0时，表示runtime构建的时候关闭了stackcache，当前M没有绑定的P时自然无法使用，还有就是GC正在运行的时候，即m.preemptoff不为空时，会存在并发问题。stackcache数组元素的类型是结构体，具体的定义代码如下：

```
type stackfreelist struct {
    list gclinkptr
    size uintptr
}
```

gclinkptr专门用来构造内存块链表，它会把每个节点最初的一个指针大小的内存用作指向下一个节点的指针。因为最小的栈也有2KB，所以list字段可以很安全地基于gclinkptr把它们连成一个链表，size字段记录的是链表的长度。当本地stackcache中某个链表空了的时候，stackcacherefill（）函数会循环调用stackpoolalloc（）函数从stackpool中对应的链表中取一些节点过来，不是按个数，而是按照空间大小为_StackCacheSize的一半，也就是每次16KB。

在不能使用stackcache来分配的时候，stackalloc（）函数会直接调用stackpoolalloc（）函数在stackpool中分配。stackpoolalloc（）函数的主要代码如下：

```
func stackpoolalloc(order uint8) gclinkptr {
    list := &stackpool[order].item.span
    s := list.first
    lockWithRankMayAcquire(&mheap_.lock, lockRankMheap)
    if s == nil {
        s = mheap_.allocManual(_StackCacheSize>>_PageShift, spanAllocStack)
        //...
        s.elemsize = _FixedStack << order
        for i := uintptr(0); i < _StackCacheSize; i += s.elemsize {
            x := gclinkptr(s.base() + i)
            x.ptr().next = s.manualFreeList
            s.manualFreeList = x
        }
        list.insert(s)
    }
    x := s.manualFreeList
    //...
    s.manualFreeList = x.ptr().next
    s.allocCount++
    if s.manualFreeList.ptr() == nil {
        list.remove(s)
    }
    return x
}
```

先尝试从stackpool中取得与目标大小对应的链表，如果链表为空，就从堆上分配一个大小等于_StackCacheSize的mspan，手动将其划分成目标大小的内存块，添加到manualFreeList中，然后把新的mspan添加到stackpool对应的链表中。最终的栈是从mspan中分配的，实际上就是从manualFreeList中取出一个内存块，并增加allocCount计数，如果mspan已经没有剩余空间了，就把它从stackpool中移除。

上述是16KB及以下大小的栈分配，主要逻辑如图9-3所示。

2. 大于或等于**32KB**的栈分配

32KB及以上大小的栈分配的主要代码如下：

```
var s *mspan
npage := uintptr(n) >> _PageShift
log2npage := stacklog2(npage)

lock(&stackLarge.lock)
if !stackLarge.free[log2npage].isEmpty() {
    s = stackLarge.free[log2npage].first
    stackLarge.free[log2npage].remove(s)
}
unlock(&stackLarge.lock)

lockWithRankMayAcquire(&mheap_.lock, lockRankMheap)

if s == nil {
    s = mheap_.allocManual(npage, spanAllocStack)
    if s == nil {
        throw("out of memory")
    }
    osStackAlloc(s)
    s.elemsize = uintptr(n)
}
v = unsafe.Pointer(s.base())
```

通过把参数n右移_PageShift位，计算出整页面数npage。再通过stacklog2用npage对2做对数运算，得到log2npage用作stackLarge.free数组的下标，从对应的链表中取出一个mspan。如果链表为空，就从堆上分配一个大小为npage个页面的mpan。最后，把mspan中的所有内存用作栈，分配工作就完成了。

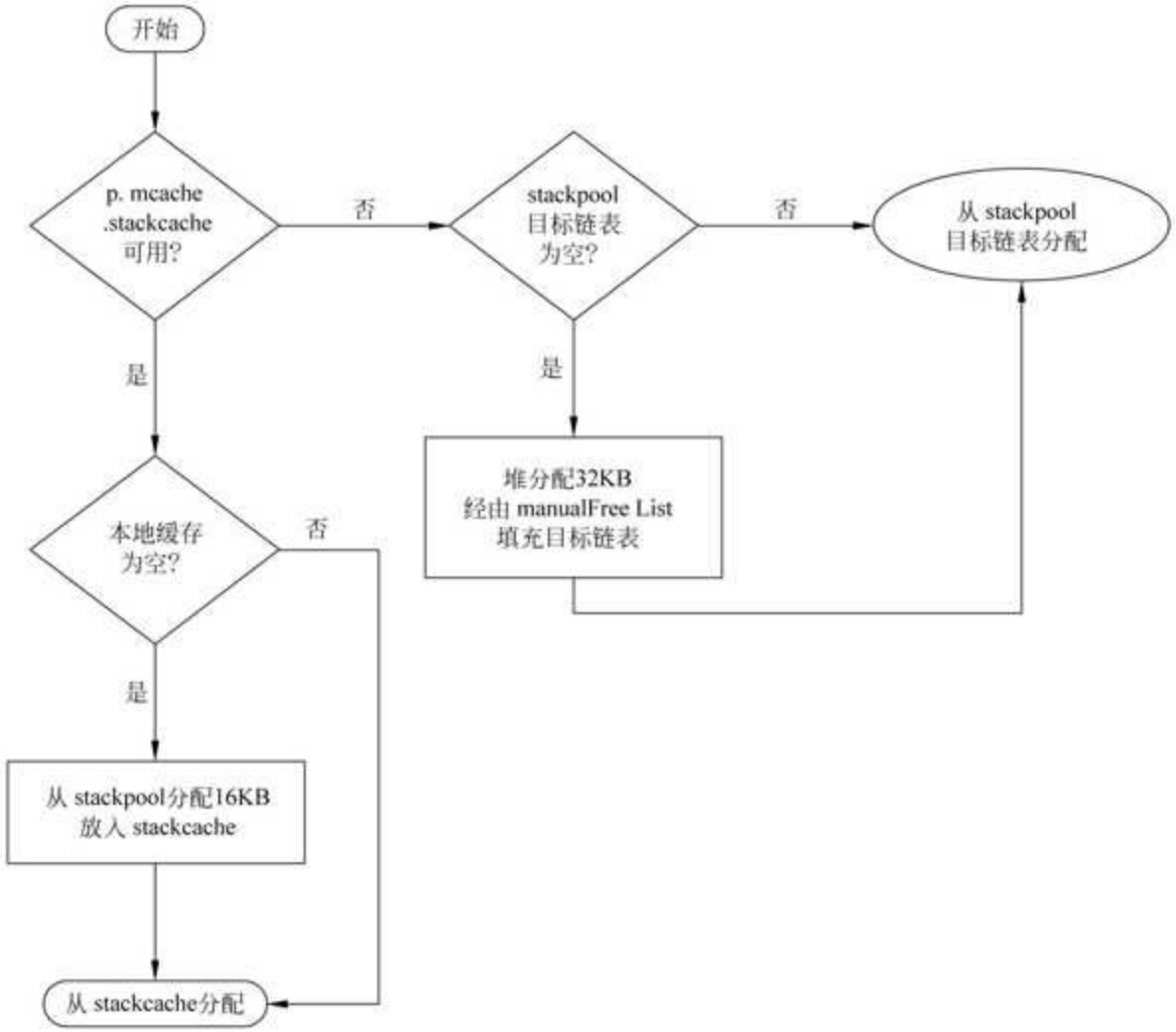

图9-3 16KB及以下大小的栈分配

3. 栈分配逻辑总结

我们总结一下整个分配逻辑，如果栈大小小于32KB，就从stackpool中分配。首先尝试本地缓存p.mcache.stackcache，缓存若为空就调用stackcacherefill（）函数从stackpool中分配16KB空间放入本地缓存。如果不能使用本地缓存，就调用stackpoolalloc（）函数直接从stackpool中分配。stackpoolalloc（）函数会按需从堆中分配32KB的内存，并划分成目标大小的块，添加到stackpool对应的链表中。如果栈大小大于或等于32KB，先检查一下stackLarge对应的链表中有没有，如果没有就直接堆分配。

goroutine栈的初始分配都发生在创建阶段，由gfget（）函数或malg（）函数调用stackalloc（）函数分配一个最小的栈，但是stackalloc（）函数并不只是在这里被调用，在运行阶段栈空间需要增长的时候，会调用该函数重新分配更大的栈空间，这也是9.2节中要研究的内容。

9.2 栈增长

在复杂的业务逻辑中，函数调用层级往往也会很深，栈空间会随着函数调用层级的加深而不断消耗。初始的2KB栈空间很可能会不够用，所以需要实现一种运行阶段动态增长的机制。goroutine的栈增长是通过编译器和runtime合作实现的，编译器会在函数的头部安插检测代码，检查当前剩余的栈空间是否够用，在不够用的时候调用runtime中的相关函数来增长栈空间。

9.2.1 栈增长检测代码

本书至此，我们已不止一次见到过栈增长检测代码，笔者多次给出对应的伪代码如下：

```
func fibonacci(n int) int {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    return fibonacci(n-1) + fibonacci(n-2)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

其实这只是几种检测代码中的一种，根据runtime源码可以得知，编译器安插在函数头部的栈增长检测代码一共有3种形式，根据当前函数栈帧的大小来确定选用哪一种，接下来我们就逐个来看一下。

1. 第一种形式的栈增长检测

第一种栈增长检测形式针对函数栈帧大小不超过_StackSmall（128字节）时，属于较小栈帧的情况。只要栈指针SP的位置没有超过stackguard0的界限，就不用进行栈增长。也就是说，在stackguard0以下有128字节空间可供安全使用。检测代码直接比较栈指针SP和stackguard0，代码如下：

```
if SP <= gp.stackguard0 {
    goto morestack
}
```

我们可以通过反编译一个栈帧为128字节的函数来实际验证一下，在amd64+Linux环境下编译一个示例，代码如下：

```
//第 9 章/code_9_1.go
//go:noinline
func test(i int) byte {
    var b [104]byte
    for x := range b {
        b[x] = byte(x)
    }
    return b[i%len(b)]
}
```

函数的逻辑并不重要，有两点需要简单说明一下：

（1）noinline注释用来避免函数被编译器内联优化掉，那样就不能反编译了。

（2）声明一个104字节的数组b，使上述函数的栈帧正好凑足128字节，函数栈帧的分配如图9-4所示。

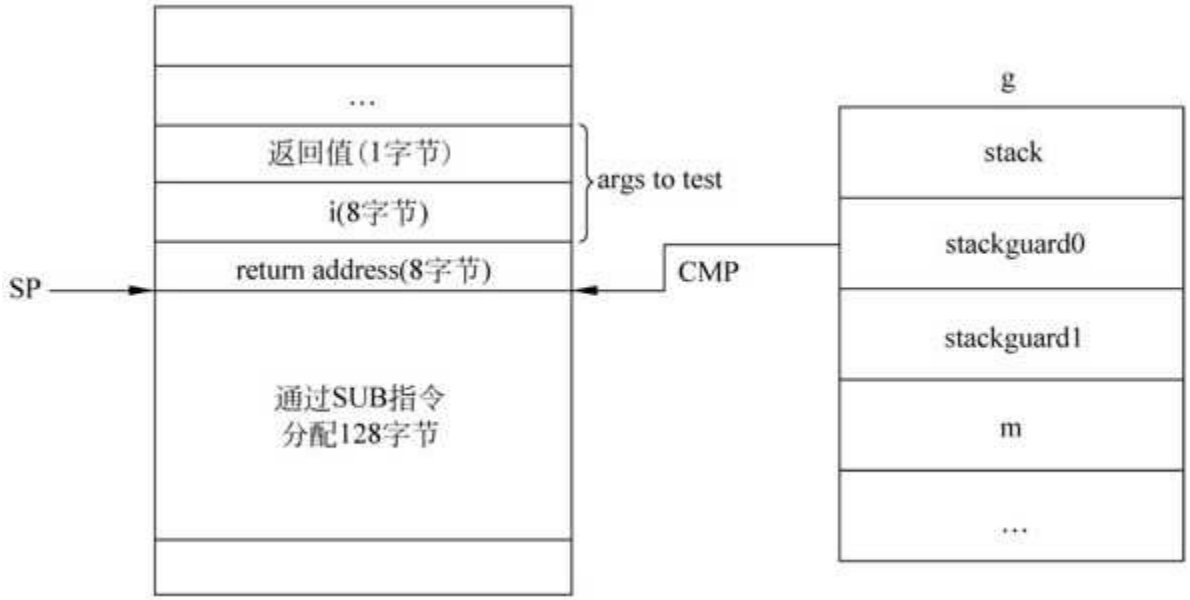

图9-4 示例函数栈帧布局

反编译之后，汇编代码的一头一尾就是栈增长检测代码，汇编代码如下：

```
$ go tool objdump -S -s '^main.test$' stack
TEXT main.test(SB) C:/go/current/gopath/src/fengyoulin.com/stack/main.go
func test(i int) byte {
  0x45ede0      64488b0c25f8ffffff    MOVQ FS:0xfffffff8, CX      //第1条指令
  0x45ede9      483b6110              CMPQ 0x10(CX), SP           //第2条指令
  0x45eded      0f86a7000000          JBE 0x45ee9a                //第3条指令
  0x45edf3      4883c480              ADDQ $-0x80, SP             //第4条指令
  0x45edf7      48896c2478            MOVQ BP, 0x78(SP)
  0x45edfc      488d6c2478            LEAQ 0x78(SP), BP
  ...
  0x45ee9a      e8c1aeffff            CALL runtime.morestack_noctxt(SB)
  0x45ee9f      90                    NOPL
  0x45eea0      e93bffffff            JMP main.test(SB)
```

第一条指令MOVQ把当前协程g的地址放到CX寄存器中，加上16字节偏移就是stackguard0字段的地址，第二条指令CMPQ直接比较stackguard0和栈指针寄存器SP。第四条指令ADDQ把栈指针向下移动了0x80字节，对应128字节的栈帧大小。用伪代码来描述上述的栈增长检测逻辑就是我们之前多次见过的这种形式，伪代码如下：

```
func test(i int) byte {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    //... 这里是函数逻辑
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

2. 第二种形式的检测代码

接下来再看一看第二种形式的检测代码，源码注释中说，当函数栈帧大小大于_StackSmall并且小于_StackBig的时候，会采用第二种形式的检测代码。根据笔者的测试，在栈帧大小等于_StackBig的时候也会采用这种形式，所以正确的范围应该是在栈帧大于128字节，并且不超过4096字节的时候。我们把上述test（）函数中数组b的大小改成4072，这样就能够构造一个4096字节的栈帧，再次反编译之后得到的检测代码如下：

```
$ go tool objdump -S -s '^main.test$' stack
TEXT main.test(SB) C:/go/current/gopath/src/fengyoulin.com/stack/main.go
func test(i int) byte {
  0x45ede0      64488b0c25f8ffffff      MOVQ FS:0xfffffff8, CX          //第 1 条指令
  0x45ede9      488d842480f0ffff        LEAQ 0xfffff080(SP), AX         //第 2 条指令
  0x45edf1      483b4110                CMPQ 0x10(CX), AX               //第 3 条指令
  0x45edf5      0f86a5000000            JBE 0x45eea0
  0x45edfb      4881ec00100000          SUBQ $0x1000, SP
  0x45ee02      4889ac24f80f0000        MOVQ BP, 0xff8(SP)
  0x45ee0a      488dac24f80f0000        LEAQ 0xff8(SP), BP
  ...
  0x45ee9c      0f1f4000                NOPL 0(AX)
  0x45eea0      e8bbaeffff              CALL runtime.morestack_noctxt(SB)
  0x45eea5      e936ffffff              JMP main.test(SB)
```

第二条指令LEAQ用SP减去3968，把结果放到了AX寄存器中。第三条指令CMPQ把AX寄存器的值和stackguard0进行比较。这里的3968是由栈帧大小减去_StackSmall得到的，如图9-5所示。

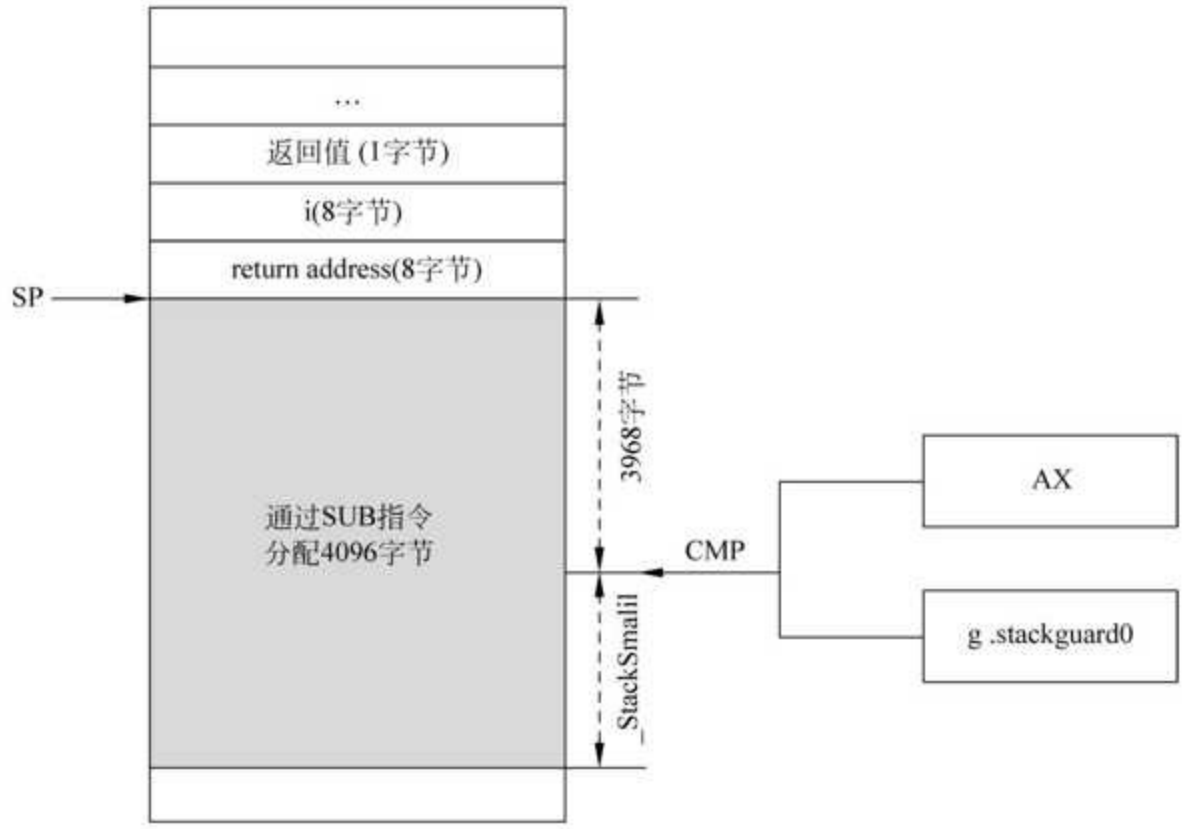

图9-5 第二种形式示例函数栈帧布局

整个检测逻辑对应的伪代码如下：

```
func test(i int) byte {
entry:
    gp := getg()
    if SP - (framesize - _StackSmall) <= gp.stackguard0 {
        goto morestack
    }
    //... 这里是函数逻辑
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

减去_StackSmall是因为stackguard0以下128字节是可以安全使用的，此范围以内不用进行栈增长。在framesize小于或等于_StackSmall的时候，括号内部是0或一个负数，SP减去它相当于加上一个非负数。只要SP大于stackguard0，加上一个小于128的非负数肯定也会大于stackguard0，所以第一种形式的检测代码可以看作第二种的简化版本。

3. 第三种形式的检测代码

第三种形式的检测代码，也是最后一种，在函数栈帧大小超过4096字节时，会使用这种形式。还是基于test（）函数，我们把数组b的大小改成4080，这样栈帧大小就变成了4104。反编译之后得到的汇编代码如下：

```
$ go tool objdump -S -s '^main.test$' stack
TEXT main.test(SB) C:/go/current/gopath/src/fengyoulin.com/stack/main.go
func test(i int) byte {
  0x45ede0      64488b0c25f8ffffff      MOVQ FS:0xfffffff8, CX      //第 1 条指令
  0x45ede9      488b7110                MOVQ 0x10(CX), SI           //第 2 条指令
  0x45eded      4881fedefaffff          CMPQ $-0x522, SI            //第 3 条指令
  0x45edf4      0f84b5000000            JE 0x45eeaf                 //第 4 条指令
  0x45edfa      488d8424a0030000        LEAQ 0x3a0(SP), AX          //第 5 条指令
  0x45ee02      4829f0                  SUBQ SI, AX                 //第 6 条指令
  0x45ee05      483d28130000            CMPQ $0x1328, AX            //第 7 条指令
  0x45ee0b      0f869e000000            JBE 0x45eeaf
  0x45ee11      4881ec08100000          SUBQ $0x1008, SP
  0x45ee18      4889ac2400100000        MOVQ BP, 0x1000(SP)
  0x45ee20      488dac2400100000        LEAQ 0x1000(SP), BP
  ...
  0x45eeaf      e8acaeffff              CALL runtime.morestack_noctxt(SB)
  0x45eeb4      e927ffffff              JMP main.test(SB)
```

此处省略掉了函数逻辑，只保留了栈增长代码。第三条指令中的-0x522对应常量stackPreempt，第五条指令中的0x3a0对应常量_StackGuard，而第七条指令中的0x1328是由栈帧大小0x1008加上_StackGuard再减去_StackSmall后得到的。这次的检测逻辑比之前的两种要复杂一点，转换成伪代码如下：

```
func test(i int) byte {
entry:
    gp := getg()
    if SP == stackPreempt {
        goto morestack
    }
    if SP + _StackGuard - gp.stackguard0 <= framesize + _StackGuard - _StackSmall {
        goto morestack
    }
    //... 这里是函数逻辑
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

SP和stackguard0都是无符号整型，因为内存地址不存在负数，相应的大小比较也是针对无符号整型的，JBE是无符号比较对应的跳转指令。无符号整型运算需要格外注意Wrap Around问题（环回问题），也就是一个数减去比自己大一些的数，会得到一个极大的正数，因此，要在两侧都加上_StackGuard，避免因为SP小于stackguard0造成减法结果环回。_StackGuard表示stackguard0到栈底的距离，在Linux下是928字节，SP加_StackGuard肯定大于stackguard0。

如图9-6所示，如果将两侧的变量进行移动，并且消除_StackGuard，就会发现和第二种形式是等价的，只不过这种变形后的比较不兼容stackPreempt，所以要前置单独判断。

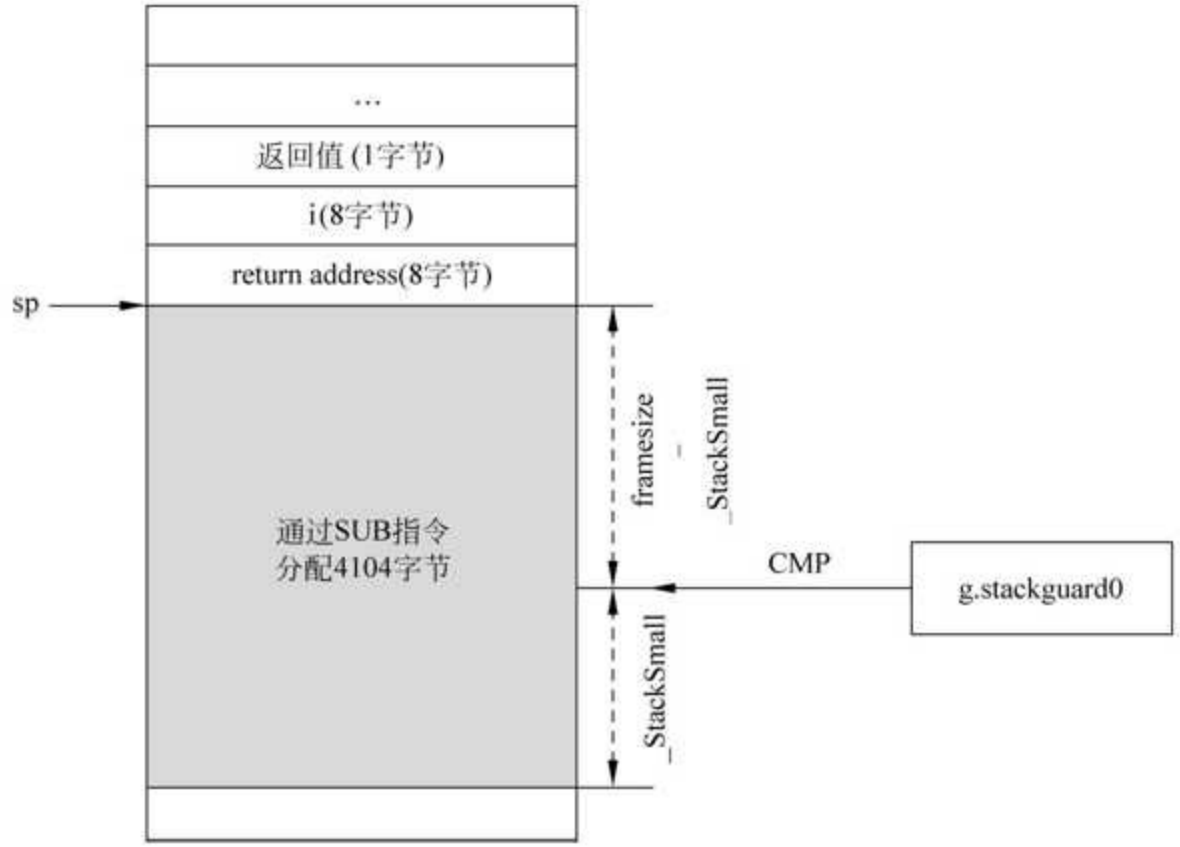

图9-6 第3种形式示例函数栈帧布局

至此，编译器安插的三种形式的栈增长检测代码都讲解过了，本质上都是判断栈指针SP向下移动栈帧大小framesize字节以后，不会超过stackguard0以下_StackSmall的位置。第三种形式最为接近，其他两种形式分别是在此基础上的简化和变形。

看完了编译器安插的代码，接下来研究一下runtime中用来执行栈增长的函数。

9.2.2 栈增长函数

通过9.2.1节的反汇编可以发现，负责进行栈增长的是runtime.morestack_noctxt（）函数，该函数是用汇编语言实现的，代码如下：

```
TEXT runtime·morestack_noctxt(SB),NOSPLIT,$0
    MOVL    $0, DX
    JMP     runtime·morestack(SB)
```

它只不过是把DX寄存器清零，然后跳转到runtime.morestack（）函数。noctxt是no context的缩写，指的是没有闭包上下文。在第3章讲解Function Value的时候我们已经知道，背后可能是个闭包，也可能是个普通的函数，Go统一支持它们，而栈增长会区分闭包和普通的函数，它们各自会调用不同的函数进行栈增长。

首先，我们准备一个闭包函数，代码如下：

```
//第 9 章/code_9_2.go
func mc(l int) func(i int) byte {
    return func(i int) byte {
        b := make([]byte, l)
        for x := range b {
            b[x] = byte(x)
        }
        return b[i%len(b)]
    }
}
```

反编译的时候需要指定内部闭包函数，对于这种匿名的函数，Go的反编译工具会对它们进行编号。例如这里是mc（）函数里的第1个匿名函数，名字是mc.func1。反编译之后在得到的栈增长代码中调用的是runtime.morestack（）函数，代码如下：

```
$ go tool objdump -S -s '^main.mc.func1$' stack
TEXT main.mc.func1(SB) C:/go/current/gopath/src/fengyoulin.com/stack/main.go
        return func(i int) byte {
  0x45ef60      64488b0c25f8ffffff    MOVQ FS:0xfffffff8, CX
  0x45ef69      483b6110              CMPQ 0x10(CX), SP
  0x45ef6d      0f8687000000          JBE 0x45effa
  0x45ef73      4883ec30              SUBQ $0x30, SP
  0x45ef77      48896c2428            MOVQ BP, 0x28(SP)
  0x45ef7c      488d6c2428            LEAQ 0x28(SP), BP
  ...
  0x45effa      e8c1acffff            CALL runtime.morestack(SB)
  0x45efff      90                    NOPL
  0x45f000      e95bffffff            JMP main.mc.func1(SB)
```

阶段性总结一下，闭包函数内部如果需要栈增长，会直接调用runtime.morestack（）函数，而一般的函数会调用runtime.morestack_noctxt（）函数，它会先显式地将DX寄存器清零，然后调用morestack（）函数。

morestack（）函数也是一个用汇编语言实现的函数，它会先进行一些检查工作，因为不能增长g0和gsignal的栈，所以它会先把调用者的PC、SP等存入g.sched中，然后调用newstack（）函数来增长栈。后半部分的代码如下：

```
//Set g->sched to context in f.
MOVQ    0(SP), AX //f's PC
MOVQ    AX, (g_sched+gobuf_pc)(SI)
MOVQ    SI, (g_sched+gobuf_g)(SI)
LEAQ    8(SP), AX //f's SP
MOVQ    AX, (g_sched+gobuf_sp)(SI)
MOVQ    BP, (g_sched+gobuf_bp)(SI)
MOVQ    DX, (g_sched+gobuf_ctxt)(SI)

//Call newstack on m->g0's stack.
MOVQ    m_g0(BX), BX
MOVQ    BX, g(CX)
MOVQ    (g_sched+gobuf_sp)(BX), SP
CALL    runtime·newstack(SB)
CALL    runtime·abort(SB)
```

需要注意的是newstack（）函数是不会返回的，它的执行流程如图9-7所示。newstack（）函数并不一定会执行栈增长，在stackguard0等于常量stackPreempt时会调用gopreempt_m（）函数让出CPU。至于正常的栈增长逻辑，newstack（）函数先把当前的栈空间大小乘以2，并把协程状态置为_Gcopystack，接下来调用copystack（）函数完成新空间分配及旧栈上数据的复制，最后将协程状态恢复为_Grunning并通过gogo（&g.sched）来恢复协程运行。

copystack（）函数真正完成了新空间分配和旧数据复制，其中有很多比较重要的细节，接下来就把最主要的逻辑摘选出来，分段进行分析。

第一部分代码如下：

```
old := gp.stack
used := old.hi - gp.sched.sp
new := stackalloc(uint32(newsize))
var adjinfo adjustinfo
adjinfo.old = old
adjinfo.delta = new.hi - old.hi
```

把当前协程的旧有栈空间范围记录在old中，计算出实际已使用的空间大小并存储在变量used中，分配新的栈空间new，根据新旧栈空间的栈底做减法得出要调整的偏移量。

第二部分代码如下：

```
ncopy := used
if !gp.activeStackChans {
    if newsize < old.hi-old.lo && atomic.Load8(&gp.parkingOnChan) != 0 {
        throw("racy sudog adjustment due to parking on channel")
    }

    adjustsudogs(gp, &adjinfo)
} else {
    adjinfo.sghi = findsghi(gp, old)
    ncopy -= syncadjustsudogs(gp, used, &adjinfo)
}
memmove(unsafe.Pointer(new.hi-ncopy), unsafe.Pointer(old.hi-ncopy), ncopy)
```

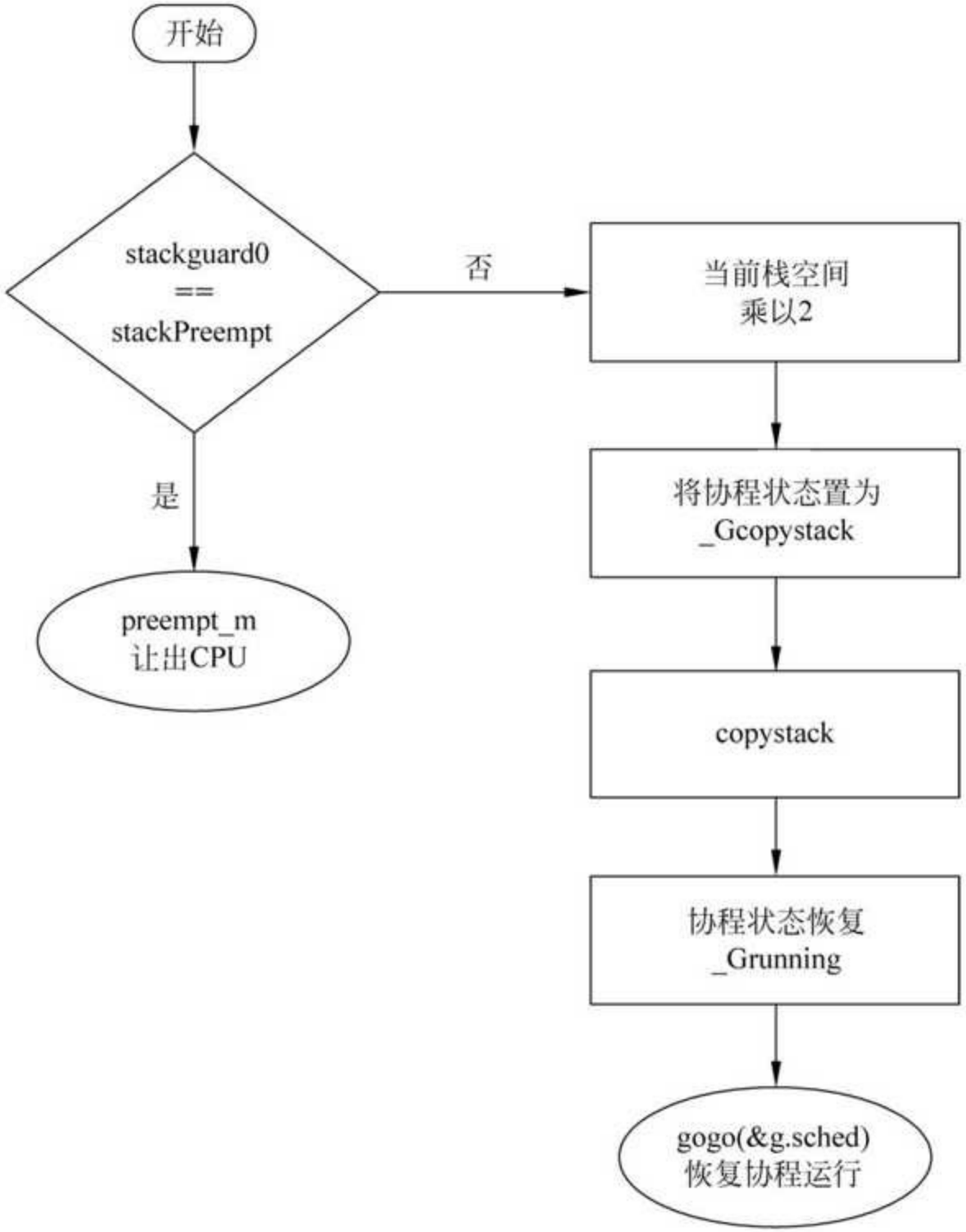

图9-7 newstack（）函数执行流程

ncopy是接下来要复制的栈区间大小，默认等于已经使用的区间大小used。activeStackChans表明存在未加锁的channel指向正在被移动的栈，需要先对这些channel加锁，然后才能安全地对栈进行操作。当newsize小于旧有栈空间大小时，表明在进行栈收缩操作，而parkingOnChan表示当前协程正在等待channel通信，此时不允许进行栈收缩，但是可以进行增长。adjustsudogs（）函数用来调整当前协程的waiting链表，它会把每个sudog节点的elem指针都加上delta偏移量，使它们都指向新的栈空间，如图9-8所示。

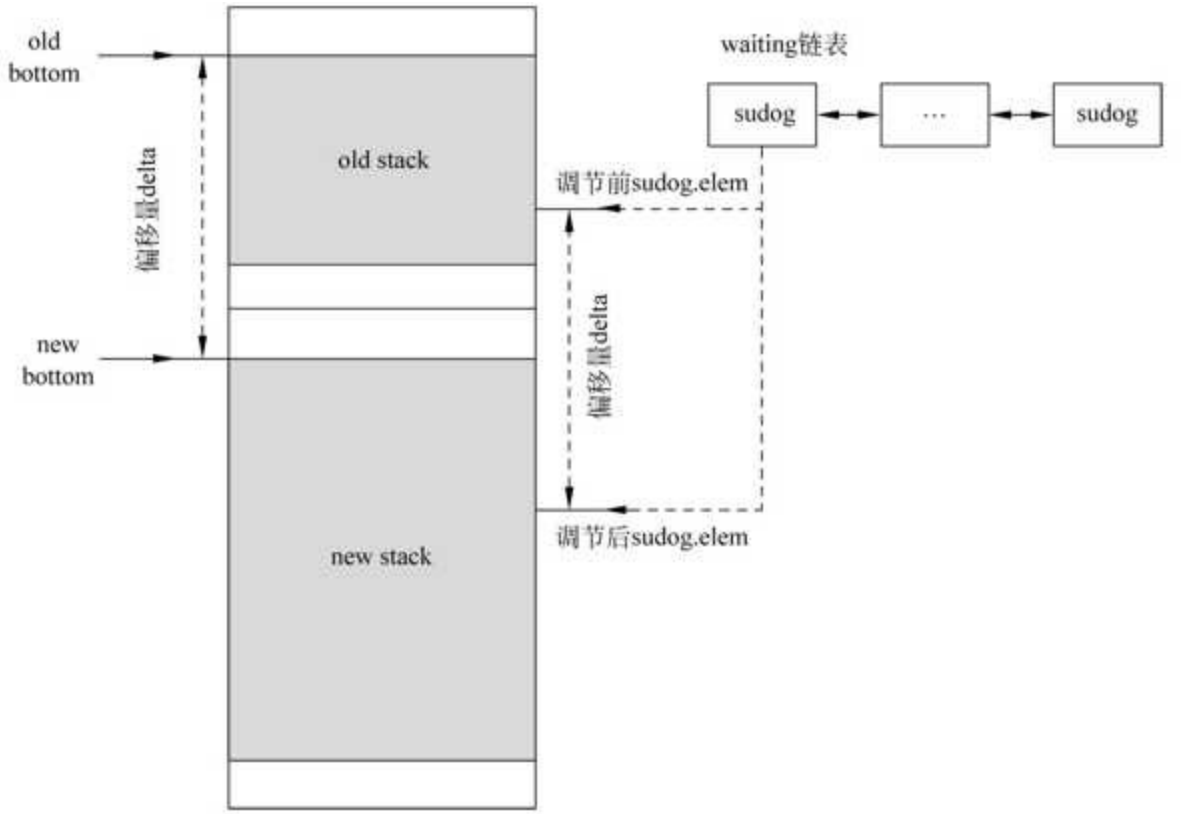

图9-8 栈增长时waiting链表调整示意图

findsghi（）函数会遍历当前协程的waiting链表，找出所有sudog的elem指针中值最大的那个。因为栈是向下增长的，如果协程因为channel通信而发生等待，则channel一般会指向最近的栈帧，所以对于从栈顶到sghi的这段区间必须谨慎操作。

syncadjustsudogs（）函数会对所有的channel加锁，然后调用adjustsudogs（）函数对waiting链表中的sudog进行调整，并通过memmove（）函数复制栈顶到sghi这段区间的栈内存，最后释放所有channel的锁，并返回复制的栈区间的大小。ncopy中要减去syncadjustsudogs已经复制的区间大小，剩下的栈内存就可以不用加锁了，可直接通过memmove（）函数进行复制。

第三部分代码如下：

```
adjustctxt(gp, &adjinfo)
adjustdefers(gp, &adjinfo)
adjustpanics(gp, &adjinfo)
if adjinfo.sghi != 0 {
    adjinfo.sghi += adjinfo.delta
}
```

adjustctxt（）函数负责对闭包上下文进行调整，实际上是把gp.sched.ctxt加上偏移量delta，如果启用了frame pointer，则该函数也会调整gp.sched.bp。adjustdefers（）函数负责调整gp._defer链表中每个_defer结构的各个字段，以及后面追加的函数参数。adjustpanics（）函数主要用于调整gp._panic，使它指向新的栈。

第四部分代码如下：

```
gp.stack = new
gp.stackguard0 = new.lo + _StackGuard
gp.sched.sp = new.hi - used
gp.stktopsp += adjinfo.delta
gentraceback(^uintptr(0), ^uintptr(0), 0, gp, 0, nil, 0x7fffffff, adjustframe, noescape
(unsafe.Pointer(&adjinfo)), 0)
stackfree(old)
```

使用新的栈空间new替换掉旧的栈空间，并更新stackguard0、sched.sp和stktopsp，让它们指向新的栈空间。通过gentraceback（）函数回调adjustframe（）函数，对新栈上的地址类变量进行修正。adjustframe（）函数会调用adjustpointers（）函数，后者在修改栈上的指针时对于栈顶到sghi这段区间内的指针会使用CAS操作以保证安全。最后通过stackfree（）函数释放旧的栈，这样copystack（）函数就完成任务了。

9.3 栈收缩

在9.2节中分析copystack（）函数源码的时候，我们发现它不仅支持栈增长，也可以执行栈收缩。就像它的名字一样，copystack（）函数只是复制并移动栈，当newsize比原来的栈空间更小时，实际上执行的是一次栈收缩。

在runtime中有个专门负责栈收缩的函数，即shrinkstack（）函数。它会进行一些校验，然后用当前栈大小的一半作为newsize调用copystack（）函数。在runtime中有两个地方会调用shrinkstack（）函数，一个是在scanstack（）函数中，另一个是在newstack（）函数中。GC的markroot（）函数会调用scanstack（）函数，scanstack（）函数又会调用shrinkstack（）函数，代码如下：

```
if isShrinkStackSafe(gp) {
    shrinkstack(gp)
} else {
    gp.preemptShrink = true
}
```

如果当前能够安全地执行栈收缩，则scanstack（）函数就会直接调用shrinkstack（）函数，否则就设置preemptShrink标识。在newstack（）函数中检测到stackPreempt之后，在让出CPU之前还会检查preemptShrink，如果值为true就会先进行栈收缩，代码如下：

```
if preempt {
    if gp.preemptShrink {
        gp.preemptShrink = false
        shrinkstack(gp)
    }

    //省略部分代码
    gopreempt_m(gp) //让出 CPU
}
```

也就是说，唯一发起栈收缩的地方是GC的scanstack（）函数。如果安全就会立即进行栈收缩，否则就设置preemptShrink标识，等到newstack（）函数检测到该标志再调用shrinkstack（）函数收缩栈。整体来看，newstack（）函数的名字也是很有道理的，因为它既可以执行栈增长，也可以执行栈收缩。

最后还有一点比较重要，需要分析一下，也就是在什么情况下能够安全地执行栈收缩，这就要看一看isShrinkStackSafe（）的源码，代码如下：

```
func isShrinkStackSafe(gp *g) bool {
    return gp.syscallsp == 0 &&
        !gp.asyncSafePoint &&
        atomic.Load8(&gp.parkingOnChan) == 0
}
```

首先判断gp.syscallsp是否等于0，如果gp.syscallsp等于0，则说明当前没有在执行系统调用，系统调用可能会有一些指针指向协程的栈，并且很多参数经过强制类型转换，无法得到最内层栈帧精确的指针位图。其次要判断asyncSafePoint是否等于false，如果asyncSafePoint等于true，则表明当前协程处在异步抢占中，这种情况下也无法得到最内层栈帧精确的指针位图。最后通过原子性的Load操作判断parkingOnChan是否等于0，在parkingOnChan不等于0的时候，表示协程正在调用gopark（）函数在某个channel上挂起等待，但是还没设置activeStackChans的值，在这个时间窗口内也不能执行栈收缩，因为copystack（）函数依赖activeStackChans的值来决定是否需要加锁，在这个时间窗口内会出现错误。

9.4 栈释放

本节我们来关注一下栈空间的释放，主要指的是goroutine的栈，在什么时候及是如何被回收的。通过源码来分析比较容易找到用来释放栈空间的函数。与分配栈空间的stackalloc（）函数对应，stackfree（）函数用来释放栈空间。通过stackfree（）函数的源码，我们基本上能了解栈空间是如何被释放的，再通过分析对stackfree（）函数的引用，就能知道栈空间是何时被释放的。

stackfree（）函数的处理逻辑和stackalloc（）函数是对应的，也是把16KB及以下和32KB及以上的栈空间分开处理的。

9.4.1 小于或等于16KB的栈空间

我们先来看一看不超过16KB的栈是如何被回收的，主要逻辑代码如下：

```
v := unsafe.Pointer(stk.lo)
n := stk.hi - stk.lo
order := uint8(0)
n2 := n
for n2 > _FixedStack {
    order++
    n2 >>= 1
}
x := gclinkptr(v)
if stackNoCache != 0 || gp.m.p == 0 || gp.m.preemptoff != "" {
    lock(&stackpool[order].item.mu)
    stackpoolfree(x, order)
    unlock(&stackpool[order].item.mu)
} else {
    c := gp.m.p.ptr().mcache
    if c.stackcache[order].size >= _StackCacheSize {
    stackcacherelease(c, order)
    }
    x.ptr().next = c.stackcache[order].list
    c.stackcache[order].list = x
    c.stackcache[order].size += n
}
```

同样先计算出log2（n/_FixedStack）并赋值给order，如果当前可以操作stackcache，就把要释放的栈内存放到stackcache对应的链表中，提前检测对应链表中空间总大小是否达到或超过了32KB，通过stackcacherelease（）函数把多余的内存放回到stackpool中，只保留_StackCacheSize的一半，也就是16KB。如果当前不能操作stackcache，就直接调用stackpoolfree（）函数，把要释放的内存直接放到stackpool对应的链表中。stackpoolfree（）函数释放后会检查对应的mspan是否完全空闲，并调用堆释放函数把完全空闲的mspan释放。

9.4.2 大于或等于32KB的栈空间

针对32KB及以上大小的栈空间释放的相关代码如下：

```
s := spanOfUnchecked(uintptr(v))
if s.state.get() != mSpanManual {
    println(hex(s.base()), v)
    throw("bad span state")
}
if gcphase == _GCoff {
    osStackFree(s)
    mheap_.freeManual(s, spanAllocStack)
} else {
    log2npage := stacklog2(s.npages)
    lock(&stackLarge.lock)
    stackLarge.free[log2npage].insert(s)
    unlock(&stackLarge.lock)
}
```

先通过栈空间的起始地址（低地址）找到对应的mspan，如果当前处于GC的清理阶段，就直接调用堆释放函数释放该mspan。若GC正在运行，为了避免栈空间被重用发生冲突，就先把它放入stackLarge的free链表中。

stackfree（）函数的逻辑到这里就梳理完了，那么该函数何时会被调用呢？

9.4.3 栈释放时机

通过分析源码中的调用关系，笔者发现会有两个地方调用该函数来释放常规goroutine的栈。常规goroutine，指的是除了g0、gsignal这类特殊协程之外，那些通过newproc（）函数创建的goroutine。这两处调用stackfree（）函数的地方，一处是在gfput（）函数中，代码如下：

```
if stksize != _FixedStack {
    stackfree(gp.stack)
    gp.stack.lo = 0
    gp.stack.hi = 0
    gp.stackguard0 = 0
}
```

如图9-9所示，该逻辑把大小不等于_FixedStack的栈都释放，这个大小也正是初始分配时的栈大小，因为shrinkstack（）函数不会把栈收缩到比这更小，所以该逻辑是把所有增长过的栈都释放，其目的是节省内存空间。

另一处调用stackfree（）来释放常规goroutine栈空间的地方在markrootFreeGStacks（）函数中，这个函数整体不算复杂，具体代码如下：

```
func markrootFreeGStacks() {
    lock(&sched.gFree.lock)
    list := sched.gFree.stack
    sched.gFree.stack = gList{}
    unlock(&sched.gFree.lock)
    if list.empty() {
        return
    }

    q := gQueue{list.head, list.head}
    for gp := list.head.ptr(); gp != nil; gp = gp.schedlink.ptr() {
        stackfree(gp.stack)
        gp.stack.lo = 0
        gp.stack.hi = 0
        q.tail.set(gp)
    }

    lock(&sched.gFree.lock)
    sched.gFree.noStack.pushAll(q)
    unlock(&sched.gFree.lock)
}
```

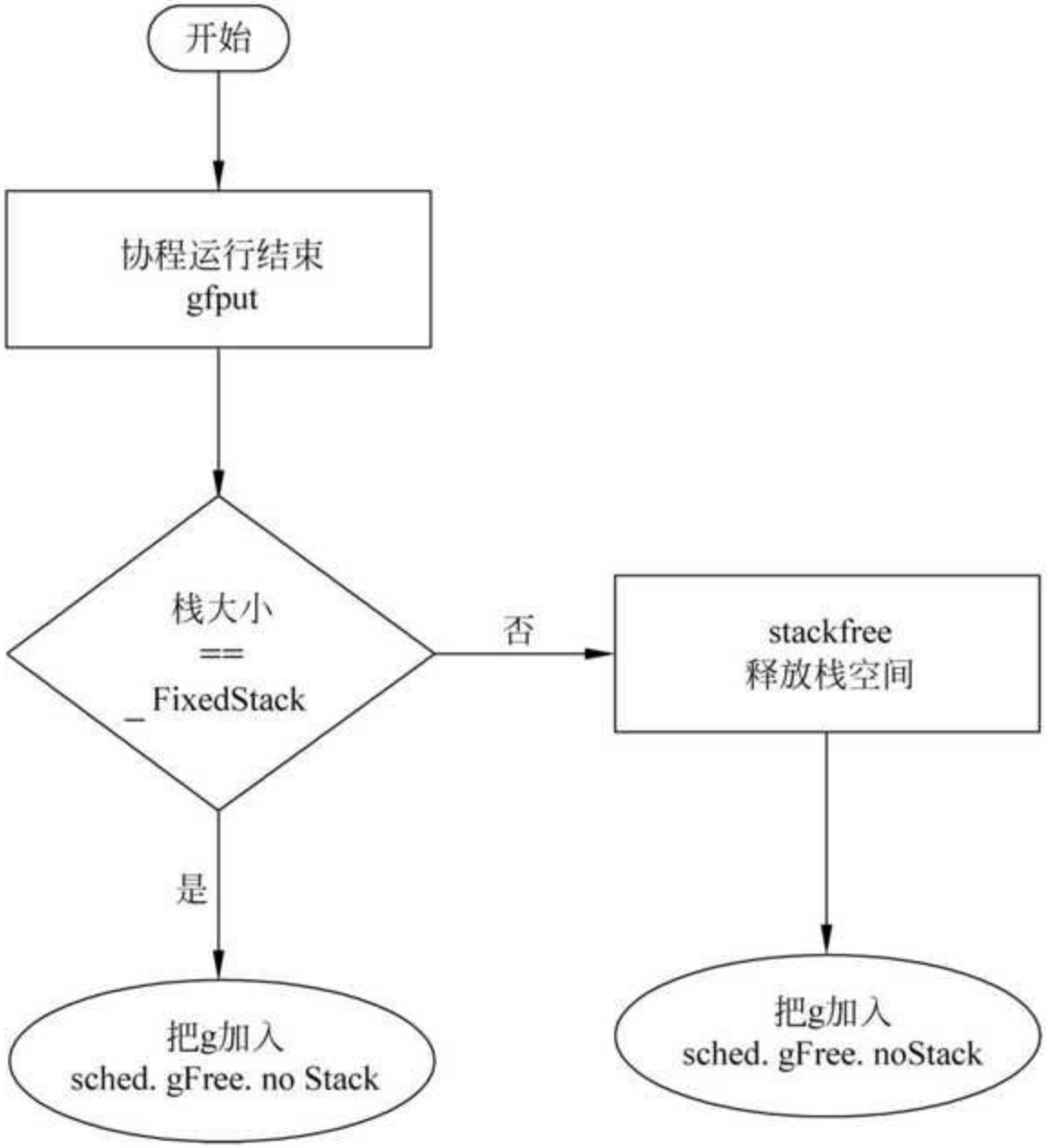

图9-9 常规栈释放的第1个时机gfput

在锁的保护下，首先获取sched.gFree.stack链表并存到本地变量list中，sched.gFree.stack就是有栈的那个全局空闲g链表。获取链表后把原链表清空，遍历获取的链表list，调用stackfree（）函数逐个释放栈，然后将gp.stack清零并把gp放入队列q中。最后把队列q全部push到sched.gFree.noStack中，也就是没有栈的那个全局空闲g链表，主要逻辑如图9-10所示。

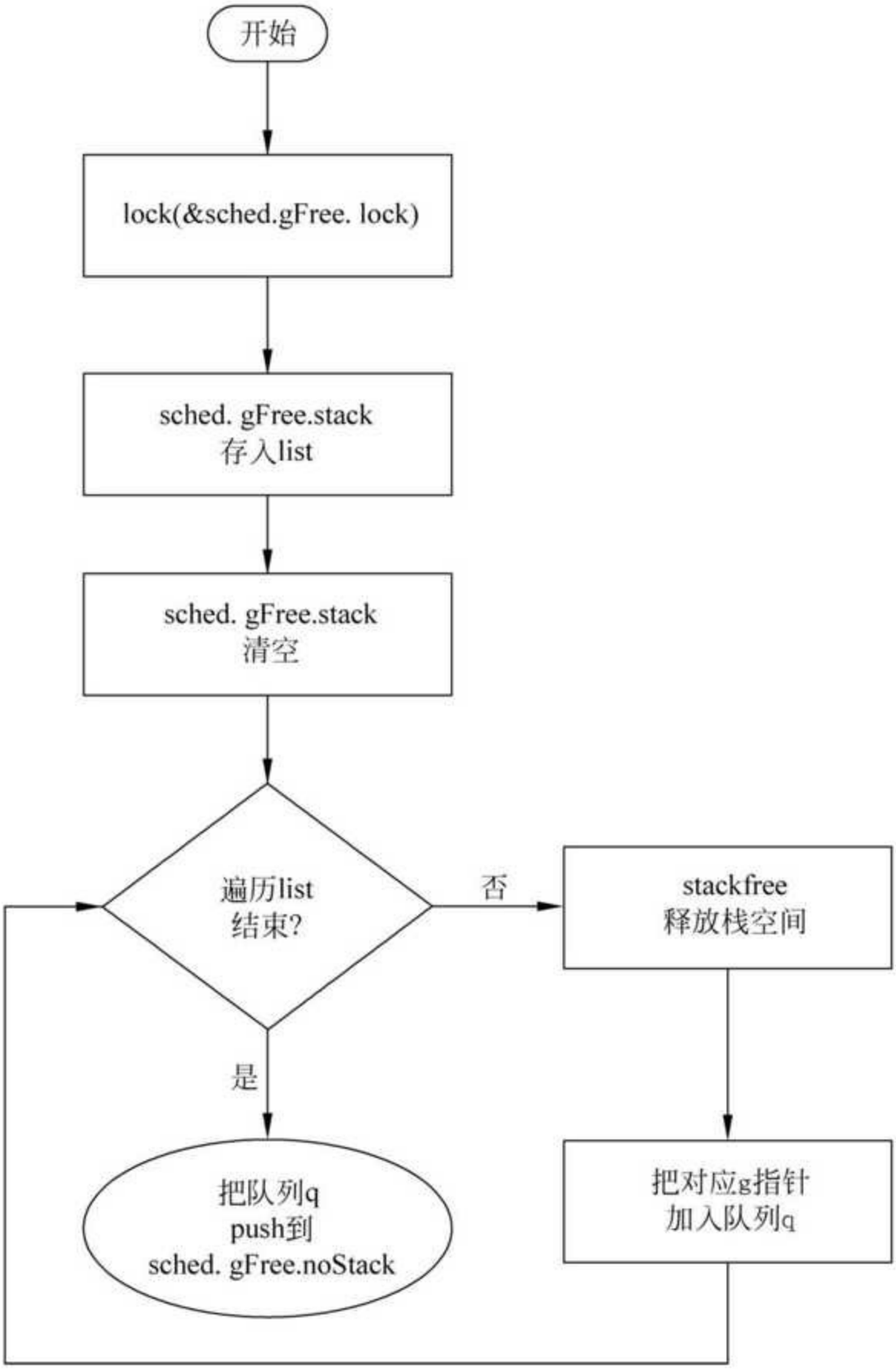

图9-10 常规栈释放的第2个时机markrootFreeGStacks

进一步追踪markrootFreeGStacks（）函数的调用者，发现只有一个地方会调用它，即markroot（）函数，也就是说源头是GC，所以常规goroutine栈的释放，一是发生在协程运行结束时，gfput会把增长过的栈释放，栈没有增长过的g会被放入sched.gFree.stack中；二是GC会处理sched.gFree.stack链表，把这里面所有g的栈都释放，然后把它们放入sched.gFree.noStack链表中。

9.5 本章小结

至此，关于goroutine栈内存管理的探索就告一段落了。我们了解了栈空间是如何分配与释放的，几个关键词是stackcache、stackpool和stackLarge。还知道了newstack（）函数既能进行栈增长，又能进行栈收缩，shrinkstack（）函数只负责栈收缩，这两者都是基于copystack（）函数实现的。copystack（）函数会分配新的栈空间，复制旧的栈数据并通过一系列adjustxxx（）函数进行指针修正，最后释放旧的栈空间。GC会发起栈收缩，以及释放sched.gFree.stack中所有g的栈空间。

考虑到栈增长的复杂性，应该还是有一定开销的，因此，对于栈深度较大的逻辑，应该避免频繁地创建和销毁协程，可以尝试结合有缓冲channel实现一个简单的协程池。

资源链接

此二维码为付费二维码