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。

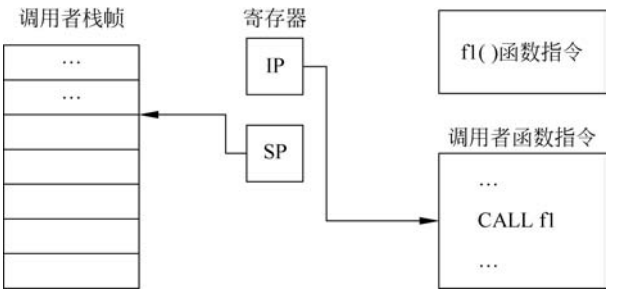

图 3-1　函数调用发生前

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。

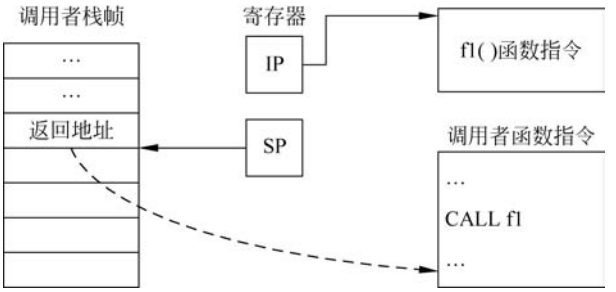

图 3-2 CALL 指令执行后

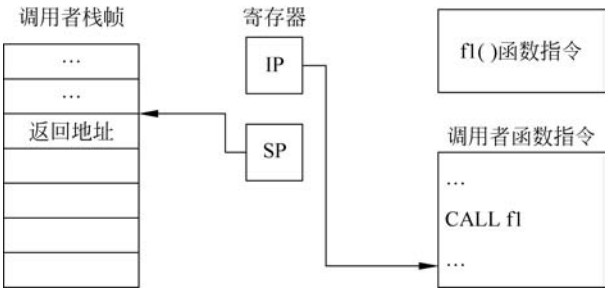

图 3-3 RET 指令执行后

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

6min

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。

(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。

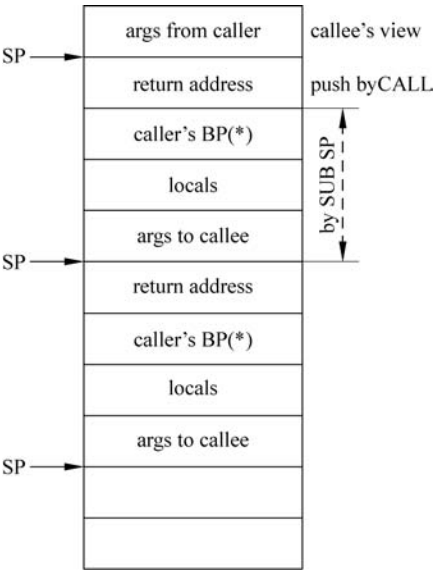

图 3-4 Go 语言函数栈帧布局示意图

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。

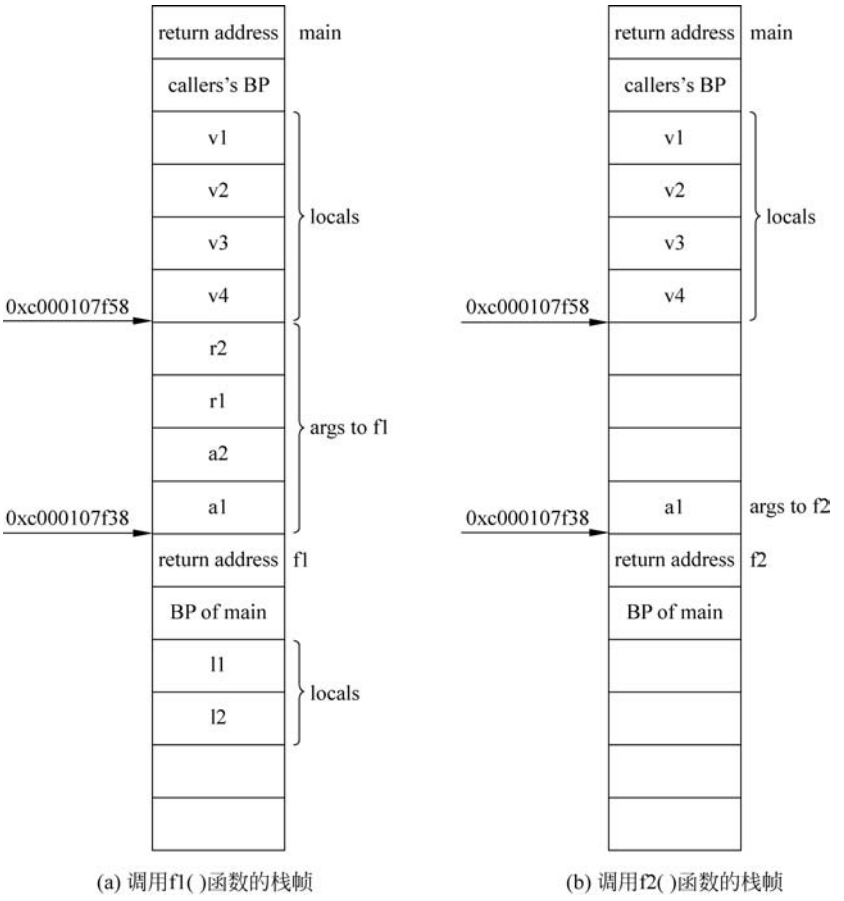

图 3-5　main 调用 f1()函数和 f2()函数的栈帧布局图

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

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。

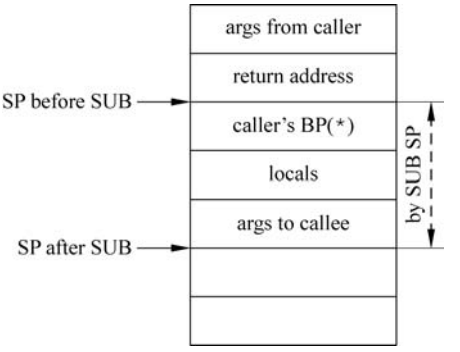

图 3-6 SUB 指令分配整个栈帧

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
```

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。

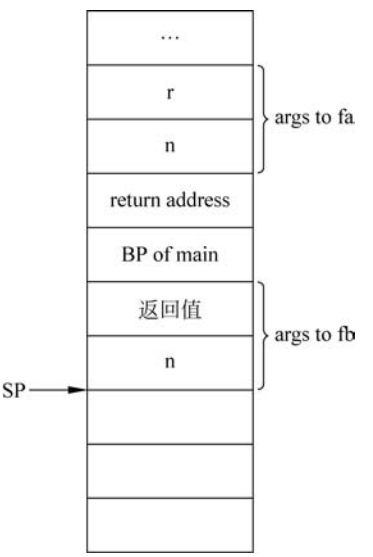

图 3-7　函数 fa 的栈帧布局

(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。

6min

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
}
```

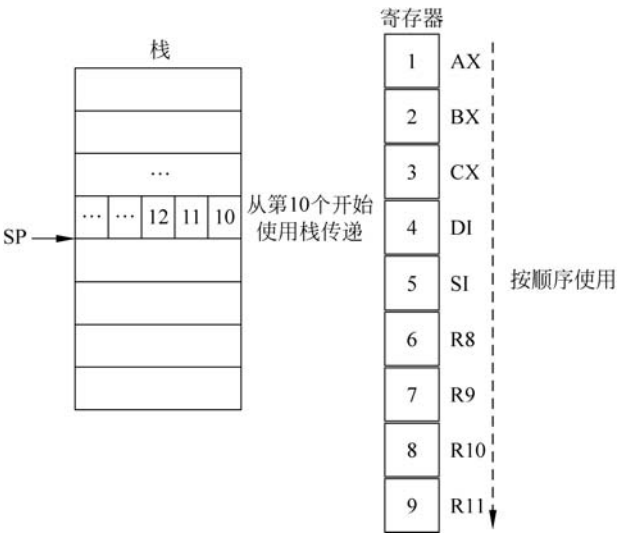

图 3-8 Go 1.17 中 in12()函数入参的传递方式

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。

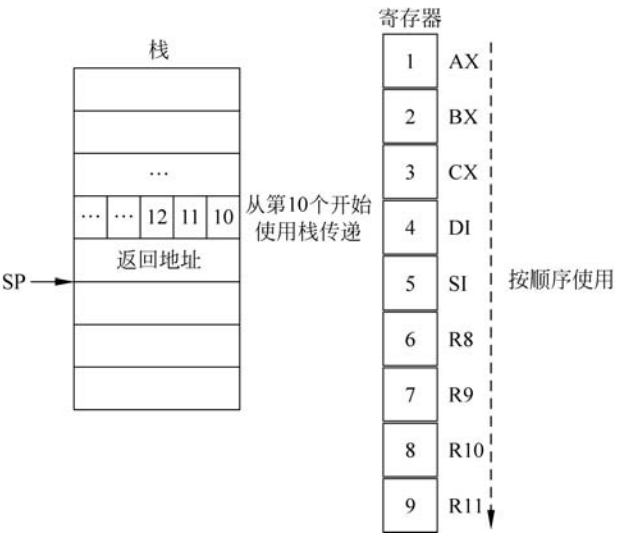

图 3-9　Go 1.17 中 out12()函数返回值的传递方式

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。

5min

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              48c744240800000000      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。

9min

3.3 Function Value

8min

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。

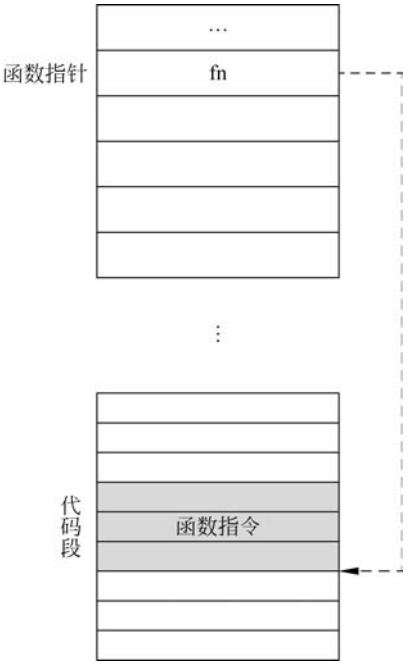

图 3-10 函数指针

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。

(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。

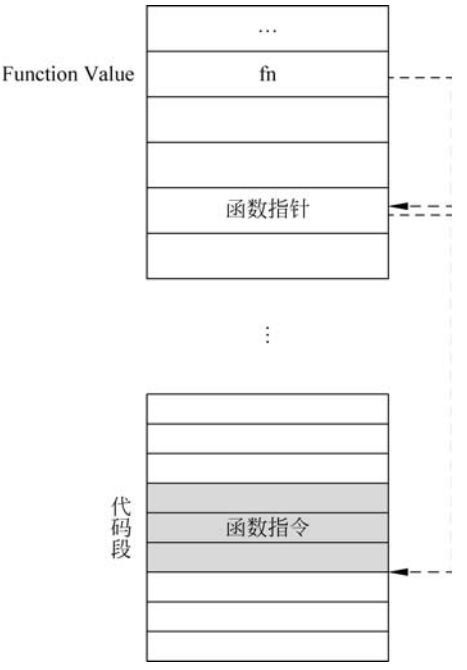

图 3-11 Function Value

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。

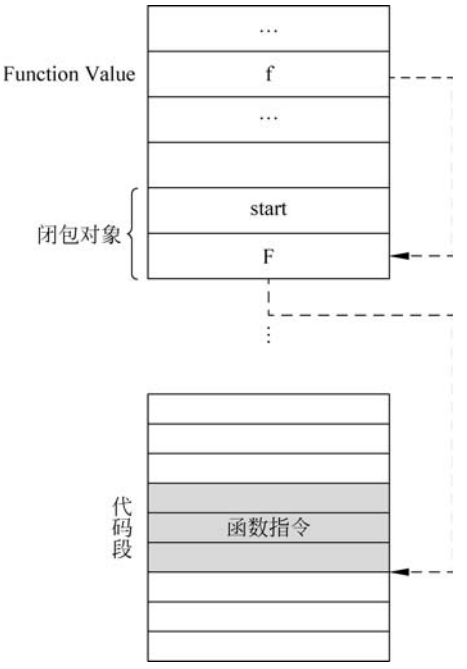

图 3-12 Function Value 和闭包对象

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。

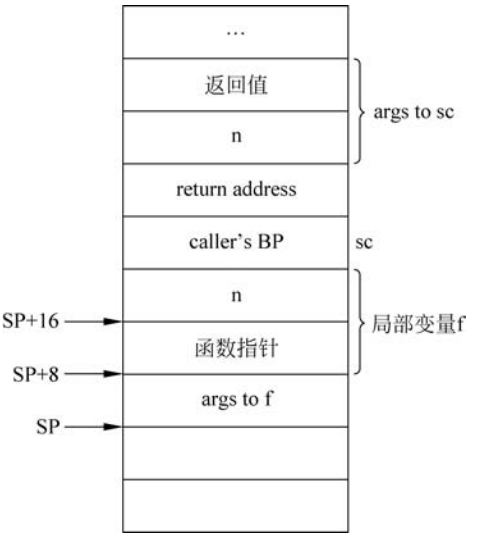

图 3-13　sc()函数中构造的闭包对象 f

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。

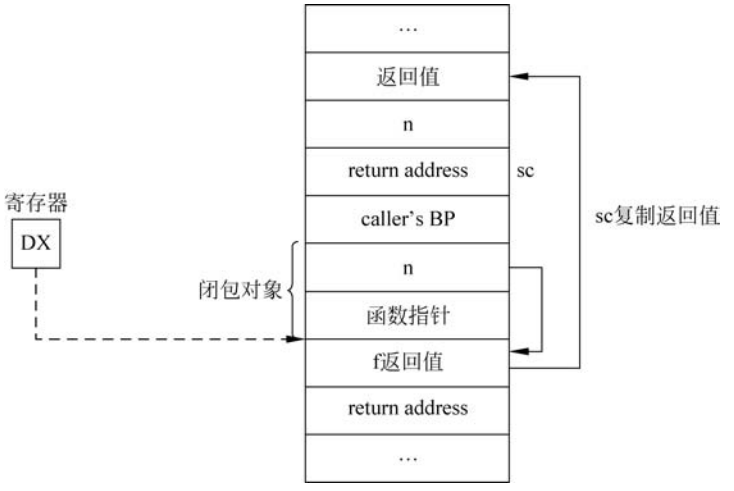

图 3-14　调用闭包函数 f()

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

5min

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。


9min

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：

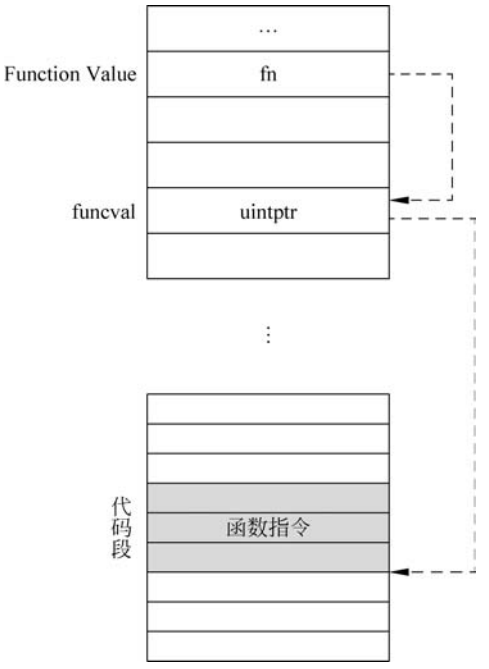

图 3-15 funcval 对 Function Value 两级指针的支持

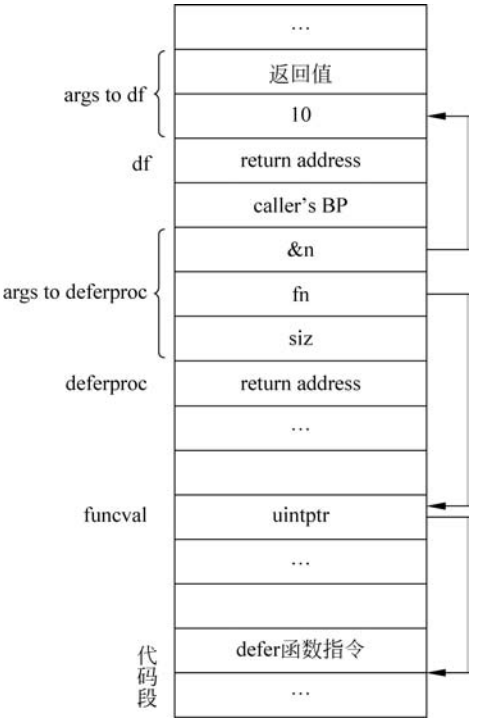

图 3-16 df()函数调用 deferproc 时的栈帧

```
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：

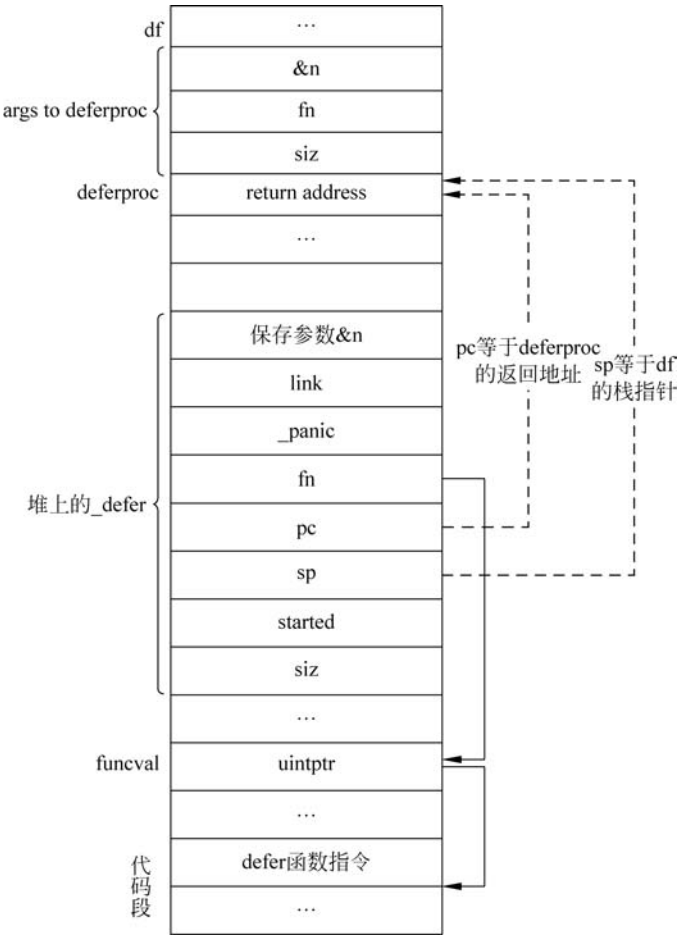

图 3-17 deferproc 执行中为_defer 赋值

```
//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。

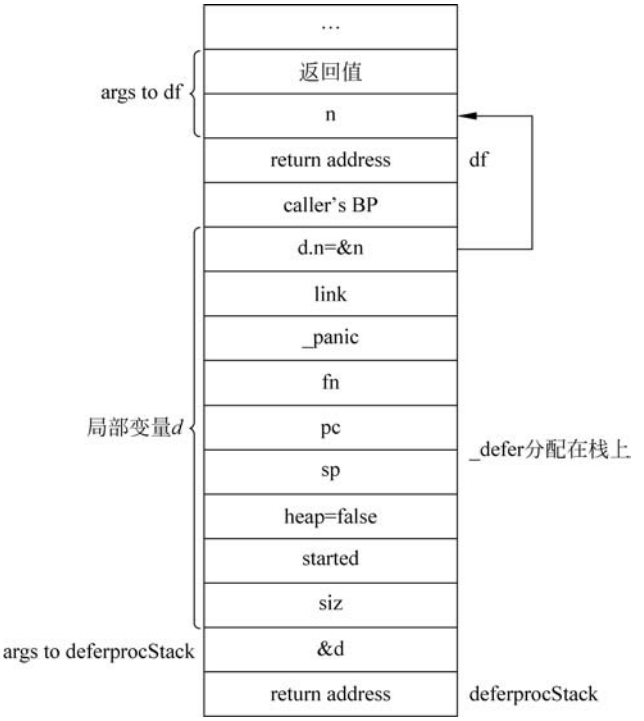

图 3-18 df()函数调用 deferprocStack()时的栈帧

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。

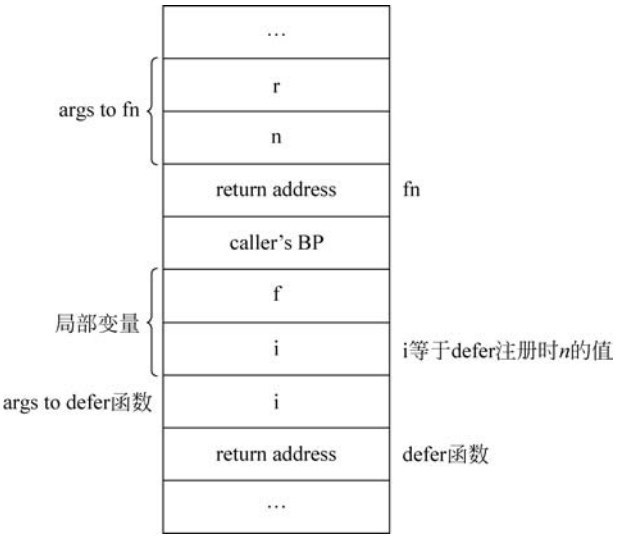

图 3-19 fn()函数通过 open coded defer 的方式调用 defer 函数

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。

10min

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。

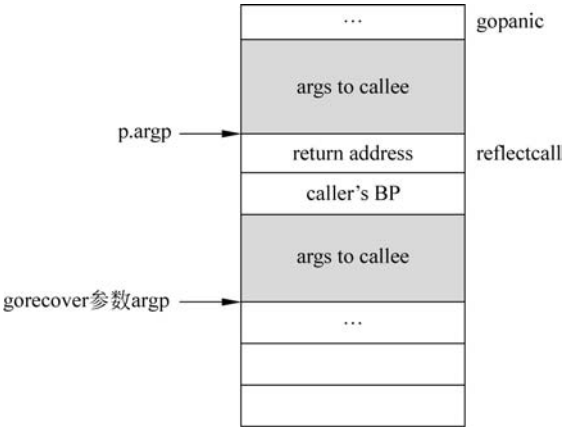

图 3-20　p.argp 和 gorecover()函数参数 argp 的关系

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。

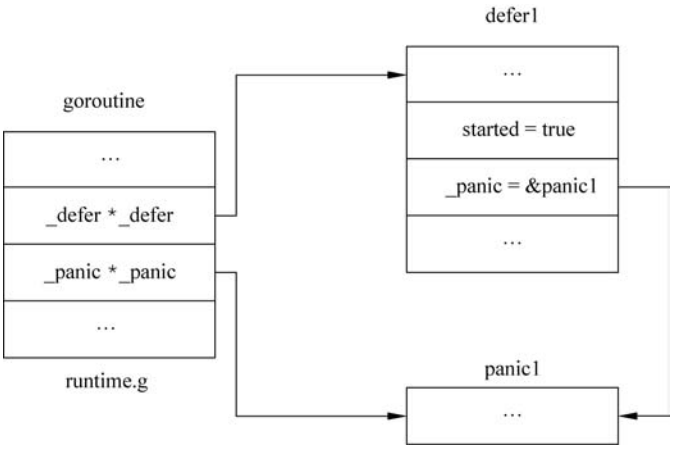

图 3-21　panic2 执行前的_defer 链表和_panic 链表

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。

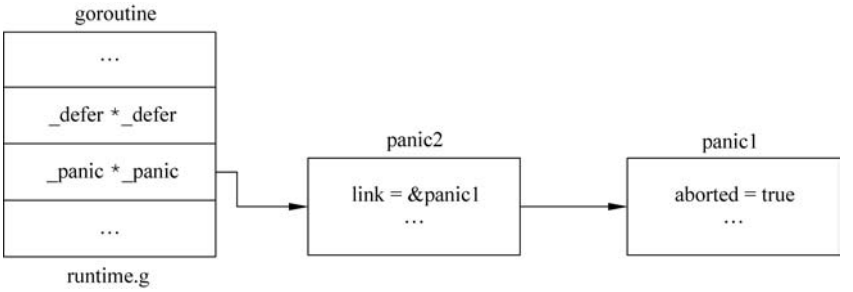

图 3-22　panic2 执行后的_defer 链表和_panic 链表

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。

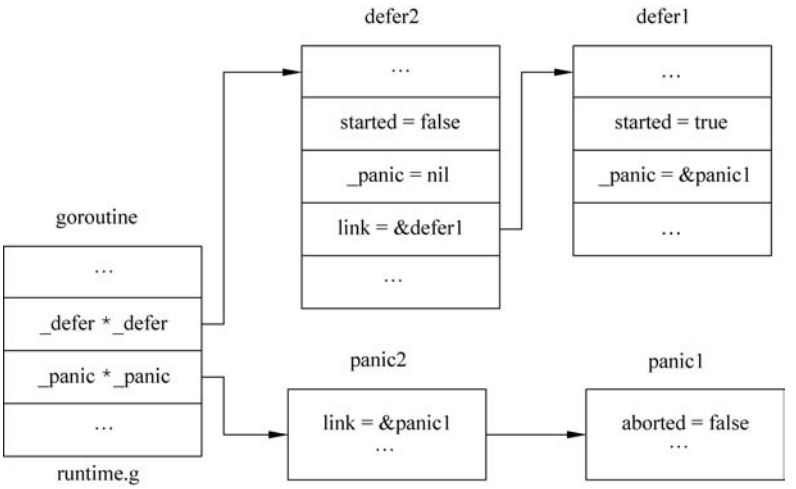

图 3-23 defer2 执行前的_defer 链表和_panic 链表

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。

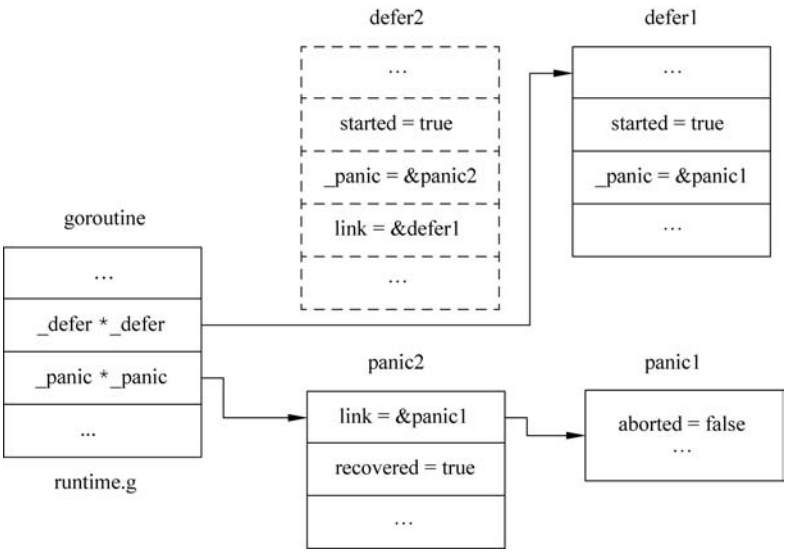

图 3-24　defer2 结束后的_defer 链表和_panic 链表

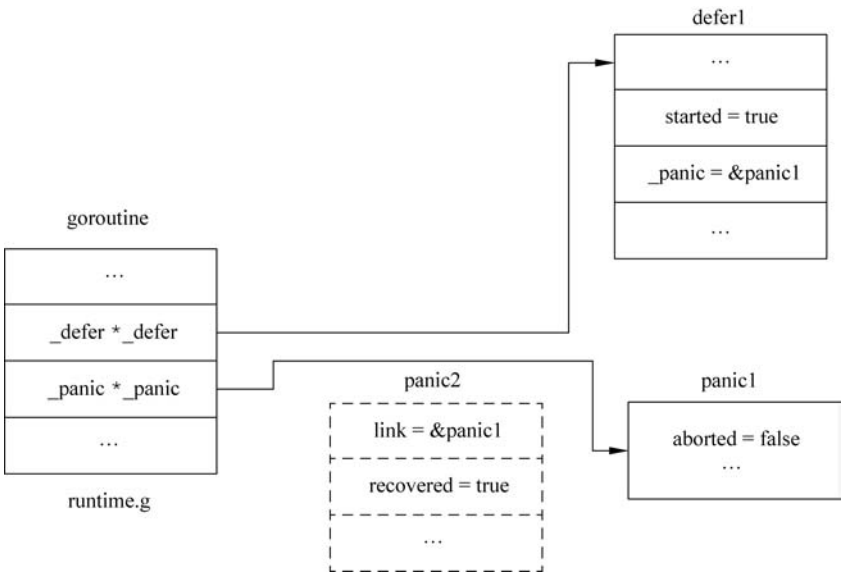

图 3-25　panic2 恢复后的_defer 链表和_panic 链表

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 addOneOpenDeferFrame()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。