Go defer 原理和源码剖析

Go 语言中有一个非常有用的保留字 defer，它可以调用一个函数，该函数的执行被推迟到包裹它的函数返回时执行。

defer 语句调用的函数，要么是因为包裹它的函数执行了 return 语句，到达了函数体的末端，要么是因为对应的 goroutine 发生了 panic。

在实际的 go 语言程序中，defer 语句可以代替其它语言中 try...catch... 的作用，也可以用来处理释放资源等收尾操作，比如关闭文件句柄、关闭数据库连接等。

1. 编译器编译 defer 过程

```
defer dosomething(x)
```

简单来说，执行 defer 语句，实际上是注册了一个稍后执行的函数，确定了函数名和参数，但不会立即调用，而是把调用过程推迟到当前函数 return 或者发生 panic 的时候。

我们先了解一下 defer 相关的数据结构。

1) struct _defer 数据结构

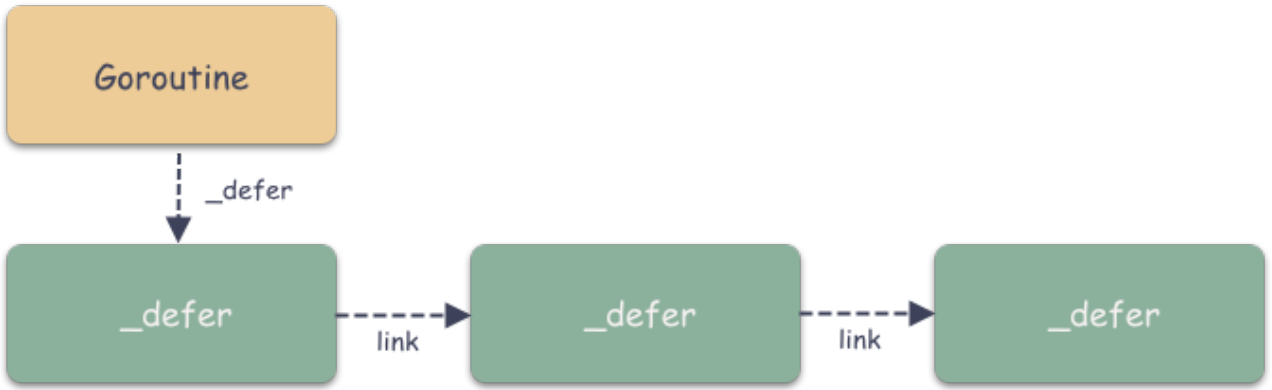
go 语言程序中每一次调用 defer 都生成一个 _defer 结构体。

```
type _defer struct {
    siz     int32 // 参数和返回值的内存大小
    started bool
    heap    bool    // 区分该结构是在栈上分配的，还是对上分配的
    sp        uintptr  // sp 计数器值，栈指针；
    pc        uintptr  // pc 计数器值，程序计数器；
    fn        *funcval // defer 传入的函数地址，也就是延后执行的函数；
    _panic    *_panic  // panic that is running defer
    link      *_defer   // 链表
}
```

我们默认使用了 go 1.13 版本的源代码，其它版本类似。

一个函数内可以有多个 defer 调用，所以自然需要一个数据结构来组织这些 _defer 结构体。_defer 按照对齐规则占用 48 字节的内存。在 _defer 结构体中的 link 字段，这个字段把所有的 _defer 串成一个链表，表头是挂在 Goroutine 的 _defer 字段。

_defer 的链式结构如下：



_defer.siz 用于指定延迟函数的参数和返回值的空间，大小由 _defer.siz 指定，这块内存的值在 defer 关键字执行的时候填充好。

defer 延迟函数的参数是预计算的，在栈上分配空间。每一个 defer 调用在栈上分配的内存布局如下图所示：

其中 _defer 是一个指针，指向一个 struct _defer 对象，它可能分配在栈上，也可能分配在堆上。

2) struct _defer 内存分配

以下是一个使用 defer 的范例，文件名为 test_defer.go：

```
package main

func doDeferFunc(x int) {
    println(x)
}

func doSomething() int {
    var x = 1
    defer doDeferFunc(x)
    x += 2
    return x
}

func main() {
    x := doSomething()
    println(x)
}
```

编译以上代码，加上去除优化和内链选项：

```
go tool compile -N -l test_defer.go
```

导出汇编代码：

```
go tool objdump test_defer.o
```

我们看下编译成的二进制代码：

```
(dlv) disassemble
TEXT main.doSomething(SB) /root/oppo/code/gopher/src/test_defer.go
        test_defer.go:7         0x452600        64488b0c25f8ffffff      mov rcx, qword ptr fs:[0xfffffff8]
        test_defer.go:7         0x452609        483b6110                cmp rsp, qword ptr [rcx+0x10]
        test_defer.go:7         0x45260d        0f8685000000            jbe 0x452698
        test_defer.go:7         0x452613        4883ec50                sub rsp, 0x50
        test_defer.go:7         0x452617        48896c2448              mov qword ptr [rsp+0x48], rbp
        test_defer.go:7         0x45261c        488d6c2448              lea rbp, ptr [rsp+0x48]
        test_defer.go:7         0x452621        48c744245800000000      mov qword ptr [rsp+0x58], 0x0
=>      test_defer.go:8         0x45262a        48c744240801000000      mov qword ptr [rsp+0x8], 0x1
        test_defer.go:9         0x452633        c744241008000000        mov dword ptr [rsp+0x10], 0x8
        test_defer.go:9         0x45263b        488d0586590200          lea rax, ptr [rip+0x25986]
        test_defer.go:9         0x452642        4889442428              mov qword ptr [rsp+0x28], rax
        test_defer.go:9         0x452647        488b442408              mov rax, qword ptr [rsp+0x8]
        test_defer.go:9         0x45264c        4889442440              mov qword ptr [rsp+0x40], rax
        test_defer.go:9         0x452651        488d442410              lea rax, ptr [rsp+0x10]
        test_defer.go:9         0x452656        48890424                mov qword ptr [rsp], rax
        test_defer.go:9         0x45265a        e84116fdff              call $runtime.deferprocStack
        test_defer.go:9         0x45265f        85c0                    test eax, eax
        test_defer.go:9         0x452661        7525                    jnz 0x452688
        test_defer.go:9         0x452663        eb00                    jmp 0x452665
        test_defer.go:10        0x452665        488b442408              mov rax, qword ptr [rsp+0x8]
        test_defer.go:10        0x45266a        4883c002                add rax, 0x2
        test_defer.go:10        0x45266e        4889442408              mov qword ptr [rsp+0x8], rax
        test_defer.go:11        0x452673        4889442458              mov qword ptr [rsp+0x58], rax
        test_defer.go:11        0x452678        90                      nop
        test_defer.go:11        0x452679        e8221cfdff              call $runtime.deferreturn
        test_defer.go:11        0x45267e        488b6c2448              mov rbp, qword ptr [rsp+0x48]
        test_defer.go:11        0x452683        4883c450                add rsp, 0x50
        test_defer.go:11        0x452687        c3                      ret
        test_defer.go:9         0x452688        90                      nop
        test_defer.go:9         0x452689        e8121cfdff              call $runtime.deferreturn
        test_defer.go:9         0x45268e        488b6c2448              mov rbp, qword ptr [rsp+0x48]
        test_defer.go:9         0x452693        4883c450                add rsp, 0x50
        test_defer.go:9         0x452697        c3                      ret
        test_defer.go:7         0x452698        e8637affff              call $runtime.morestack_noctxt
        test_defer.go:7         0x45269d        e95effffff              jmp $main.doSomething
(dlv)
```

从汇编指令我们看到，编译器在遇到 defer 关键字的时候，添加了一些运行库函数： `deferprocStack` 和 `deferreturn` 。

go 1.13 正式版本的发布提升了 defer 的性能，号称针对 defer 场景提升了 30% 的性能。

go 1.13 之前的版本 defer 语句会被编译器翻译成两个过程： 回调注册函数过程：`deferproc` 和 `deferreturn`

。

go 1.13 带来的 deferprocStack 函数，这个函数就是这个 30% 性能提升的核心手段。deferprocStack 和 deferproc 的目的都是注册回调函数，但是不同的是 deferprocStatck 是在栈内存上分配 struct _defer 结构，而 deferproc 这个是需要去堆上分配结构内存的。而我们绝大部分的场景都是可以是在栈上分配的，所以自然整体性能就提升了。栈上分配内存自然是比对上要快太多了，只需要改变 rsp 寄存器的值就可以进行分配。

那么什么时候分配在栈上，什么时候分配在堆上呢?

在编译器相关的文件（src/cmd/compile/internal/gc/ssa.go ）里，有个条件判断：

```
func (s *state) stmt(n *Node) {

    case ODEFER:
        d := callDefer
        if n.Esc == EscNever {
            d = callDeferStack
        }
}
```

n.Esc 是 ast.Node 的逃逸分析的结果，那么什么时候 n.Esc 会被置成 EscNever 呢?

这个在逃逸分析的函数 esc 里（src/cmd/compile/internal/gc/esc.go ）：

```
func (e *EscState) esc(n *Node, parent *Node) {

    case ODEFER:
        if e.loopdepth == 1 { // top level
            n.Esc = EscNever // force stack allocation of defer record (see ssa.go)
            break
        }
}
```

这里 e.loopdepth 等于 1的时候，才会设置成 EscNever ， e.loopdepth 字段是用于检测嵌套循环作用域的，换句话说，defer 如果在嵌套作用域的上下文中，那么就可能导致 struct _defer 分配在堆上，如下：

```
package main

func main() {
    for i := 0; i < 10; i++ {
        defer func() {
            _ = i
        }()
    }
}
```

编译器生成的则是 deferproc：

```
(dlv) disassemble
TEXT main.main(SB) /root/oppo/code/gopher/src/test_defer_heap.go
        test_defer_heap.go:3    0x452c20        64488b0c25f8ffffff      mov rcx, qword ptr fs:[0xfffffff8]
        test_defer_heap.go:3    0x452c29        483b6110                cmp rsp, qword ptr [rcx+0x10]
        test_defer_heap.go:3    0x452c2d        0f86a2000000            jbe 0x452cd5
        test_defer_heap.go:3    0x452c33*       4883ec30                sub rsp, 0x30
        test_defer_heap.go:3    0x452c37        48896c2428              mov qword ptr [rsp+0x28], rbp
        test_defer_heap.go:3    0x452c3c        488d6c2428              lea rbp, ptr [rsp+0x28]
=>      test_defer_heap.go:4    0x452c41        488d0518b40000          lea rax, ptr [rip+0xb418]
        test_defer_heap.go:4    0x452c48        48890424                mov qword ptr [rsp], rax
        test_defer_heap.go:4    0x452c4c        e80f81fbff              call $runtime.newobject
        test_defer_heap.go:4    0x452c51        488b442408              mov rax, qword ptr [rsp+0x8]
        test_defer_heap.go:4    0x452c56        4889442420              mov qword ptr [rsp+0x20], rax
        test_defer_heap.go:4    0x452c5b        48c70000000000          mov qword ptr [rax], 0x0
        test_defer_heap.go:4    0x452c62        eb00                    jmp 0x452c64
        test_defer_heap.go:4    0x452c64        488b442420              mov rax, qword ptr [rsp+0x20]
        test_defer_heap.go:4    0x452c69        48833802                cmp qword ptr [rax], 0x2
        test_defer_heap.go:4    0x452c6d        7c02                    jl 0x452c71
        test_defer_heap.go:4    0x452c6f        eb54                    jmp 0x452cc5
        test_defer_heap.go:5    0x452c71        488b442420              mov rax, qword ptr [rsp+0x20]
        test_defer_heap.go:7    0x452c76        4889442418              mov qword ptr [rsp+0x18], rax
        test_defer_heap.go:5    0x452c7b        c7042408000000          mov dword ptr [rsp], 0x8
        test_defer_heap.go:5    0x452c82        488d0d07530200          lea rcx, ptr [rip+0x25307]
        test_defer_heap.go:5    0x452c89        48894c2408              mov qword ptr [rsp+0x8], rcx
        test_defer_heap.go:5    0x452c8e        4889442410              mov qword ptr [rsp+0x10], rax
        test_defer_heap.go:5    0x452c93        e80810fdff              call $runtime.deferproc
        test_defer_heap.go:5    0x452c98        85c0                    test eax, eax
        test_defer_heap.go:5    0x452c9a        7519                    jnz 0x452cb5
        test_defer_heap.go:5    0x452c9c        eb00                    jmp 0x452c9e
        test_defer_heap.go:4    0x452c9e        eb00                    jmp 0x452ca0
        test_defer_heap.go:4    0x452ca0        488b442420              mov rax, qword ptr [rsp+0x20]
        test_defer_heap.go:4    0x452ca5        488b00                  mov rax, qword ptr [rax]
        test_defer_heap.go:4    0x452ca8        488b4c2420              mov rcx, qword ptr [rsp+0x20]
        test_defer_heap.go:4    0x452cad        48ffc0                  inc rax
        test_defer_heap.go:4    0x452cb0        488901                  mov qword ptr [rcx], rax
        test_defer_heap.go:4    0x452cb3        ebaf                    jmp 0x452c64
        test_defer_heap.go:5    0x452cb5        90                      nop
        test_defer_heap.go:5    0x452cb6        e82519fdff              call $runtime.deferreturn
        test_defer_heap.go:5    0x452cbb        488b6c2428              mov rbp, qword ptr [rsp+0x28]
        test_defer_heap.go:5    0x452cc0        4883c430                add rsp, 0x30
        test_defer_heap.go:5    0x452cc4        c3                      ret
        test_defer_heap.go:9    0x452cc5        90                      nop
        test_defer_heap.go:9    0x452cc6        e81519fdff              call $runtime.deferreturn
        test_defer_heap.go:9    0x452ccb        488b6c2428              mov rbp, qword ptr [rsp+0x28]
        test_defer_heap.go:9    0x452cd0        4883c430                add rsp, 0x30
        test_defer_heap.go:9    0x452cd4        c3                      ret
```

当 defer 外层出现显式（for）或者隐式（goto）的时候，将会导致 struct _defer 结构体分配在堆上，性能就会变差，这个编程的时候要注意。

编译器就能决定 _defer 结构体分配在栈上还是堆上，对应函数分别是 deferprocStatck 和 deferproc 函数，这两个函数都很简单，目的一致：分配出 struct _defer 的内存结构，把回调函数初始化进去，挂到链表中。

3) deferprocStack 栈上分配

deferprocStack 函数做了哪些事情呢?

```
// 进入这个函数之前，就已经在栈上分配好了内存结构
func deferprocStack(d *_defer) {
    gp := getg()

    // siz 和 fn 在进入这个函数之前已经赋值
    d.started = false
    // 表明是栈的内存
    d.heap = false
    // 获取到 caller 函数的 rsp 寄存器值，并赋值到 _defer 结构 sp 字段中
    d.sp = getcallersp()
    // 获取到 caller 函数的 rip 寄存器值，并赋值到 _defer 结构 pc 字段中
    // 根据函数调用的原理，我们就知道 caller 的压栈的 pc (rip) 值就是 deferprocStack 的下一条指令
    d.pc = getcallerpc()

    // 把这个 _defer 结构作为一个节点，挂到 goroutine 的链表中
    *(*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))
    // 注意，特殊的返回，不会触发延迟调用的函数
    return0()
}
```

小结：

- 由于是栈上分配内存的，所以调用到 deferprocStack 之前，编译器就已经把 struct _defer 结构的函数准备好了；
- _defer.heap 字段用来标识这个结构体分配在栈上；

- 保存上下文，把 caller 函数的 rsp，pc（rip） 寄存器的值保存到 _defer 结构体；
- _defer 作为一个节点挂接到链表。注意：表头是 goroutine 结构的 _defer 字段，而在一个协程任务中大部分有多次函数调用的，所以这个链表会挂接一个调用栈上的 _defer 结构，执行的时候按照 rsp 来过滤区分；

4) deferproc 堆上分配

堆上分配的函数为 deferproc ，简化逻辑如下：

```
func deferproc(siz int32, fn *funcval) {
  // arguments of fn fullow fn
    // 获取 caller 函数的 rsp 寄存器值
    sp := getcallersp()
    argp := uintptr(unsafe.Pointer(&fn)) + unsafe.Sizeof(fn)
    // 获取 caller 函数的 pc（rip） 寄存器值
    callerpc := getcallerpc()

    // 分配 struct _defer 内存结构
    d := newdefer(siz)
    if d._panic != nil {
        throw("deferproc: d.panic != nil after newdefer")
    }
    // _defer 结构体初始化
    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))
    }
    // 注意，特殊的返回，不会触发延迟调用的函数
    return0()
}
```

小结：

- 与栈上分配不同，struct _defer 结构是在该函数里分配的，调用 newdefer 分配结构体，newdefer 函数则是先去 poul 缓存池里看一眼，有就直接取用，没有就调用 mallocgc 从堆上分配内存；
- deferproc 接受入参 siz，fn ，这两个参数分别标识延迟函数的参数和返回值的内存大小，延迟函数地址；
- _defer.heap 字段用来标识这个结构体分配在堆上；
- 保存上下文，把 caller 函数的 rsp，pc（rip） 寄存器的值保存到 _defer 结构体；
- _defer 作为一个节点挂接到链表；

5) 执行 defer 函数链

编译器遇到 defer 语句，会插入两个函数：

- 分配函数：deferproc 或者 deferprocStack ；
- 执行函数：deferreturn 。

包裹 defer 语句的函数退出的时候，由 deferreturn 负责执行所有的延迟调用链。

```
func deferreturn(arg0 uintptr) {
    gp := getg()
    // 获取到最前的 _defer 节点
    d := gp._defer
    // 函数递归终止条件（d 链表遍历完成）
    if d == nil {
        return
    }
    // 获取 caller 函数的 rsp 寄存器值
    sp := getcallersp()
    if d.sp != sp {
        // 如果 _defer.sp 和 caller 的 sp 值不一致，那么直接返回；
        // 因为，就说明这个 _defer 结构不是在该 caller 函数注册的
        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))
        }
        // 获取到延迟回调函数地址
        fn := d.fn
        d.fn = nil
        // 把当前 _defer 节点从链表中摘除
        gp._defer = d.link
        // 释放 _defer 内存（主要是堆上才会需要处理，栈上的随着函数执行完，栈收缩就回收了）
        freedefer(d)
        // 执行延迟回调函数
        jmpdefer(fn, uintptr(unsafe.Pointer(&arg0)))
    }
```

代码说明：

- 遍历 defer 链表，一个个执行，顺序链表从前往后执行，执行一个摘除一个，直到链表为空；
- jmpdefer 负责跳转到延迟回调函数执行指令，执行结束之后，跳转回 deferreturn 里执行；
- _defer.sp 的值可以用来判断哪些是当前 caller 函数注册的，这样就能保证只执行自己函数注册的延迟回调函数；
  - 例如，a() -> b() -> c() ， a 调用 b，b 调用 c ，而 a，b，c 三个函数都有 defer 注册延迟函数，那么自然是 c()函数返回的时候，执行 c 的回调；

2. defer 传递参数

1) 预计算参数

在前面描述 _defer 数据结构的时候说到内存结构如下：

_defer 在栈上作为一个 header，延迟回调函数（ defer ）的参数和返回值紧接着 _defer 放置，而这个参数值是在 defer 执行的时候就设置好了，也就是预计算参数，而非等到执行 defer 函数的时候才去获取。

举个例子，执行 defer func(x, y) 的时候，x，y 这两个实参是计算的出来的，Go 中的函数调用都是值传递。那么就会把 x，y 的值拷贝到 _defer 结构体之后。再看个例子：

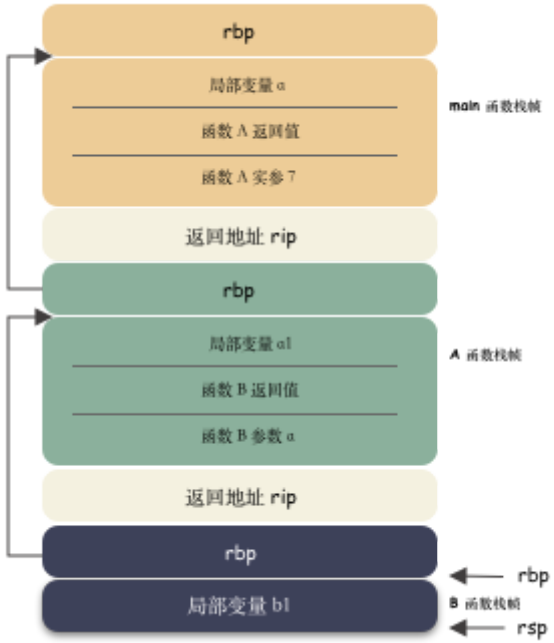
```
package main

func main() {
    var x = 1
    defer println(x)
    x += 2
    return
}
```

这个程序输出是什么呢？是 1 ，还是 3 ？答案是：1 。defer 执行的函数是 println ， println 参数是 x ， x 的值传进去的值则是在 defer 语句执行的时候就确认了的。

2) defer 的参数准备

defer 延迟函数执行的参数已经保存在和 _defer 一起的连续内存块了。那么执行 defer 函数的时候，参数是哪里来呢？当然不是直接去 _defer 的地址找。因为这里是走的标准的函数调用。

在 Go 语言中，一个函数的参数由 caller 函数准备好，比如说，一个 main() -> A(7) -> B(a) 形成类似以下的栈帧：



所以，deferreturn 除了跳转到 defer 函数指令，还需要做一个事情：把 defer 延迟回调函数需要的参数准备好（空间和值）。那么就是如下代码来做的视线：

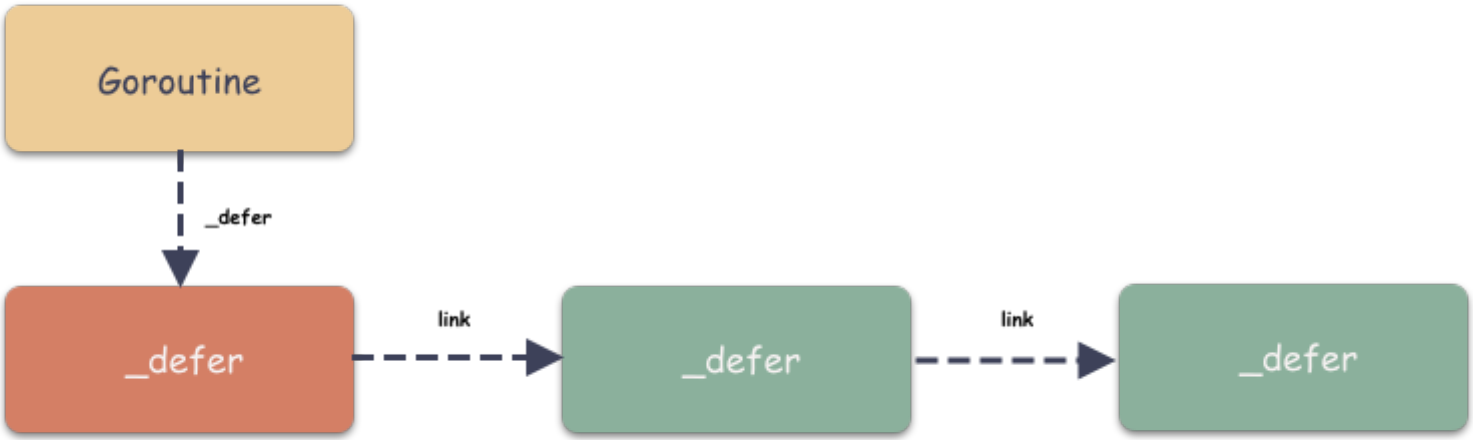
```
func deferreturn(arg0 uintptr) {

    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))
    }
```

```
    }
```

arg0 就是 caller 用来放置 defer 参数和返回值的栈地址。这段代码的意思就是，把 _defer 预先的准备好的参数，copy 到 caller 栈帧的某个地址（arg0）。

3. 执行多条 defer

前面已经详细说明了，_defer 是一个链表，表头是 goroutine._defer 结构。一个协程的函数注册的是挂同一个链表，执行的时候按照 rsp 来区分函数。并且，这个链表是把新元素插在表头，而执行的时候是从前往后执行，所以这里导致了一个 LIFO 的特性，也就是先注册的 defer 函数后执行。



4. defer 和 return 运行顺序

包含 defer 语句的函数返回时，先设置返回值还是先执行 defer 函数?

1) 函数的调用过程

要理解这个过程，首先要知道函数调用的过程：

- go 的一行**函数调用**语句其实非原子操作，对应多行汇编指令，包括 1）参数设置，2） call 指令执行；
- 其中 call 汇编指令的内容也有两个：返回地址压栈（会导致 rsp 值往下增长，rsp-0x8），callee 函数地址加载到 pc 寄存器；
- go 的一行**函数返回 return**语句其实也非原子操作，对应多行汇编指令，包括 1）**返回值设置** 和 2）ret 指令执行；
- 其中 ret 汇编指令的内容是两个，指令 pc 寄存器恢复为 rsp 栈顶保存的地址，rsp 往上缩减，rsp+0x8；
- 参数设置在 caller 函数里，返回值设置在 callee 函数里；
- rsp, rbp 两个寄存器是栈帧的最重要的两个寄存器，这两个值划定了栈帧；

最重要的一点：Go 的 return 的语句调用是个复合操作，可以对应一下两个操作序列：

- 设置返回值
- ret 指令跳转到 caller 函数

2) return 之后是先返回值还是先执行 defer 函数?

Golang 官方文档有明确说明：

> That is, if the surrounding function returns through an explicit return statement, deferred functions are executed **after any result parameters are set by that return statement** but **before the function returns to its caller**.

也就是说，defer 的函数链调用是在设置了返回值之后，但是在运行指令上下文返回到 caller 函数之前。

所以含有 defer 注册的函数，执行 return 语句之后，对应执行三个操作序列：

- 设置返回值
- 执行 defer 链表
- ret 指令跳转到 caller 函数

那么，根据这个原理我们来解析如下的行为：

```
func f1 () (r int) {
    t := 1
    defer func() {
        t = t + 5
    }()
    return t
}
```

```
func f2() (r int) {
    defer func(r int) {
        r = r + 5
    }(r)
    return 1
}

func f3() (r int) {
    defer func () {
        r = r + 5
    } ()
    return 1
}
```

这三个函数的返回值分别是多少?

答案：f1() -> 1，f2() -> 1，f3() -> 6 。

a) 函数 f1 执行 return t 语句之后：

- 设置返回值 r = t，这个时候局部变量 t 的值等于 1，所以 r = 1；
- 执行 defer 函数，t = t+5 ，之后局部变量 t 的值为 6；
- 执行汇编 ret 指令，跳转到 caller 函数；

所以，f1() 的返回值是 1 ；

b) 函数 f2 执行 return 1 语句之后：

- 设置返回值 r = 1 ；
- 执行 defer 函数，defer 函数传入的参数是 r，r 在预计算参数的时候值为 0，Go 传参为值传递，0 赋值给了匿名函数的参数变量，所以 ，r = r+5 ，匿名函数的参数变量 r 的值为 5；
- 执行汇编 ret 指令，跳转到 caller 函数；

所以，f2() 的返回值还是 1 ；

c) 函数 f3 执行 return 1 语句之后：

- 设置返回值 r = 1；
- 执行 defer 函数，r = r+5 ，之后返回值变量 r 的值为 6（这是个闭包函数，注意和 f2 区分）；

- 执行汇编 ret 指令，跳转到 caller 函数；

所以，f1() 的返回值是 6 。

5. 总结

- defer 关键字执行对应 _defer 数据结构，在 go1.1 - go1.12 期间一直是堆上分配，在 go1.13 之后优化成栈上分配 _defer 结构，性能提升明显；
- _defer 大部分场景是分配在栈上的，但是遇到循环嵌套的场景会分配到堆上，所以编程时要注意 defer 使用场景，否则可能出性能问题；
- _defer 对应一个注册的延迟回调函数（defer），defer 函数的参数和返回值紧跟 _defer，可以理解成 header，_defer 和函数参数，返回值所在内存是一块连续的空间，其中 _defer.siz 指明参数和返回值的所占空间大小；
- 同一个协程里 defer 注册的函数，都挂在一个链表中，表头为 goroutine._defer；
  - 新元素插入在最前面，遍历执行的时候则是从前往后执行。所以 defer 注册函数具有 LIFO 的特性，也就是后注册的先执行；
  - 不同的函数都在这个链表上，以 _defer.sp 区分；
  - defer 的参数是预计算的，也就是在 defer 关键字执行的时候，参数就确认，赋值在 _defer 的内存块后面。执行的时候，copy 到栈帧对应的位置上；
  - return 对应 3 个动作的复合操作：设置返回值、执行 defer 函数链表、ret 指令跳转。

参考：编程宝库 go 语言教程。

posted on 2021-11-12 14:11