Go底层探索(六):延迟函数defer

刘庆辉 猿码记 2023-05-18 17:54 Posted on 北京

收录于合集
#Go进阶 14 #Go 101

> **@注: 以下内容来自《Go语言底层原理剖析》、《Go 语言设计与实现》书中的摘要信息，本人使用版本(`Go1.18`)与书中不一致，源码路径可能会有出入。**

1.介绍

`defer` 是 `Go` 语言中的关键字，也是 `Go` 语言的重要特性之一。 `defer` 的语法形式为 `defer 函数` ，其后必须紧跟一个函数调用或者方法调用。在很多时候， `defer` 后的函数以匿名函数或闭包的形式呈现，例如：

```
defer func(...){
  // 逻辑处理
}
```

2. 特性

2.1 延迟执行

`defer` 后的函数并不会立即执行，而是推迟到了函数结束后执行，示例如下:

```
func deferDelayExec() {
 defer func() {
  fmt.Println("World")
 }()
```

```
 fmt.Println("Hello ")
}
/** 输出
Hello
World
*/
```

2.2 先进后出

在函数体内部，可能出现多个 `defer` 函数。这些 `defer` 函数将按照后入先出（ `last-in first-out, LIFO` ）的顺序执行，这与栈的执行顺序是相同的。

```
func deferExecSort() {
 for i := 1; i <= 5; i++ {
  defer fmt.Println(i)
 }
}
/**输出
5
4
3
2
1
*/
```

2.3 参数预计算

`defer` 的另一个特性是参数的预计算，这一特性时常导致开发者在使用 `defer` 时犯错。因为在大部分时候，我们记住的都是其延迟执行的特性。参数的预计算指当函数到达defer语句时，延迟调用的参数将立即求值，传递到defer函数中的参数将预先被固定，而不会等到函数执行完成后再传递参数到defer中。看下面代码示例:

```
func deferParam() {
 a := 1
```

```
    // 参数a传入
    defer func(b int) {
     fmt.Println("b = ", b+1)
    }(a)
    defer func() {
     fmt.Println("c = ", a+1)
    }()
    a = 99
    fmt.Println("a = ", a)
}
/**输出
a =  99
c =  100
b =  2
*/
```

代码分析:

- **b = 2**: 当函数执行到第一个 `defer` 时，把 `a` 当参数传递 `defer` 函数后**参数将预先被固定**，此时的结果已经求出 `b=1+1` ，等待输出。
- **c = 100**: 当执行到第二个 `defer` 时，并没有把 `a` 传到 `defer` 函数中,由于 `defer` 的延迟特性，要等函数结束后才能执行，函数结束后时， `a` 被赋值为99，所以 `c` 计算的结果是 `99+1 = 100`

3. 常见用途

3.1 资源释放

利用 `defer` 的延迟执行特性， `defer` 一般用于资源的释放和异常的捕获，作为 `Go` 语言的特性之一， `defer` 给 `Go` 代码的书写方式带来了很大的变化。下面的 `CopyFile` 函数用于将文件 `srcName` 的内容复制到文件 `dstName` 中。

```
func CopyFile(srcName string, dstName string) error {
    // 打开源文件
    srcFile, err := os.Open(srcName)
    if err != nil {
```

```
        return err
    }
    // 释放资源
    defer srcFile.Close()

    // 创建目标文件
    dstFile, err := os.Create(dstName)
    if err != nil {
        return err
    }
    // 释放资源
    defer dstFile.Close()

    // 复制文件内容
    _, err = io.Copy(dstFile, srcFile)
    return err
}
```

除了上面常见的操作文件资源以外， `defer` 还常用于锁和锁的释放，实例代码如下:

```
// 不使用defer时
func lockNoUseDefer() {
 ...
 p.Lock()
 if p.count < 10 {
    // 释放锁
  p.Unlock()
  return p.count
 }
 p.count++
 newCount := p.count
  // 释放锁
 p.Unlock()
 return newCount
}

// 使用defer时
func lockUseDefer()  {
 ...
 p.Lock()
```

```
 // 使用defer释放锁
 defer p.Unlock()
 if p.count < 10 {
  return p.count
 }
 p.count++
 return p.count
}
```

> 通过上面代码可以看出,使用 `defer` 后,代码的可读性更好，而且不会因为逻辑复杂而忘了解锁，导致死锁的情况。

3.2 异常捕获

```
func deferCatchError() {
 // 定义一个匿名延迟函数
 defer func() {
  err := recover()
  msg := fmt.Sprintf("err信息: %v",err)
  if err != nil {
   // 程序触发panic时，会被这里捕获
   fmt.Println(msg)
  }
 }()
 // 故意抛出panic
 panic("这里出错了~" )
}

/** 输出
  err信息: 这里出错了~
*/
```

> 程序在运行时,任意地方都可能会发生 `panic` 异常，例如算术除0错误、内存无效访问、数组越界等，一旦发生 `panic` 异常,就会导致整个程序异常退出，这不是我们想见到的结

> 果，通常我们希望程序能够继续正常执行，其他编程语言会提供 `try..catch` 的语法，但 `go` 不支持 `try..catch` 的语法,只能通过 `defer + recover` 来实现；

4. 返回值陷阱

除了前面提到的参数预计算， `defer` 还有一种非常容易犯错的场景: **涉及与返回值参数结合。**

4.1 先看示例

```
func f1() (n int) {
 n = 1
 defer func() {
  n++
 }()
 return n
}

func f2() int {
 n := 1
 defer func() {
  n++
 }()
 return n
}

func TestRun(t *testing.T) {
 fmt.Println("f1 = ", f1())
 fmt.Println("f2 = ", f2())
}
```

运行输出如下:

```
=== RUN   TestRun
f1 =  2
f2 =  1
```

```
--- PASS: TestRun (0.00s)
PASS
```

4.2 defer、return、返回值

在讲三者的执行顺序前，先了解下 `return` 返回值的运行机制，`return xx` 并非原子操作，在编译后会被分为**赋值**、**返回值**两个指令。

当与 `defer` 结合使用时，三者的执行顺序：

- `return 赋值` 最先执行，即先将结果写入返回值中；
- 接着 `defer` 开始执行一些收尾工作；
- 最后函数携带当前返回值退出（即返回值）。

所以结论是：第一步先return赋值，第二步再执行defer，第三步执行空的return。但是在有名与无名的函数返回值的情况下会有些区别：

4.3 无名函数返回

如果函数的返回值是无名的（不带命名返回值）如上例中的 `f2()` ，则 `go` 语言会在执行 `return` 指令时,创建一个**临时变量**保存返回值，最后返回。结合 `f2` 理解如下:

```
// 这里返回值是无名
func f2() int {
  // 第一步:return赋值；创建了一个临时变量保存返回值
 n := 1
  // 后续defer对n操作，这里的n并不是返回值变量
 defer func() {
  n++
 }()
  // 空的return,这一步是将第一步中的临时变量保存的值返回
 return
}
```

a.分析下代码运行:

上例代码一共执行3步操作:

- **return 赋值：**因为返回值没有命名，所以 `return` 默认指定了一个返回值（假设为 `s` ），实际运行可以理解如下:

```
n := 1
// 这里的s指的是临时变量
s := n
```

- **`defer` 操作:** 后续的 `defer` 操作都是针对 `n` 进行的，并不会影响返回值 `s` ，所以 `s` 始终都是等于 `1` .

- **空return返回**：大部分人都会被 `return n` 给误导，明明返回的是 `n` ,为什么最后结果不对呢，实际上最后的返回 `rentun n` 最后会变成 `return` 。用代码理解如下:

```
// 定义临时变量s，s是最终返回值
var s int
// 赋值
n := 1
s = n
return s
```

4.4 有名函数返回

有名返回值的函数,由于返回值变量已经提前定义，所以运行过程中并不会再创建临时变量，后续 `defer` 操作的变量都是返回值变量,结合 `f1` 理解如下:

```
// 定义了返回值变量
func f1() (n int) {
  // 直接操作返回值
 n = 1
 defer func() {
```

```
    // 这里操作的也是返回值
    n++
  }()
  return n
}
```

5. 数据结构

5.1 字段释义

`defer` 关键字在 `Go` 语言源代码中对应的数据结构如下:

```
type _defer struct {
  siz       int32 // 参数和结果的内存大小
  started   bool
  openDefer bool //表示当前 defer 是否经过开放编码的优化
  sp        uintptr //栈指针
  pc        uintptr //调用方的程序计数器
  fn        *funcval //defer 关键字中传入的函数
  _panic    *_panic // 触发延迟调用的结构体
  link      *_defer // 使用此字段串成链表
    ...
}
```

5.2 串成链表

> `runtime._defer` 结构体是延迟调用链表上的一个元素，所有的结构体都会通过 `link` 字段串联成链表。

串联成链示意图

6. 执行机制

6.1 三种机制

中间代码生成阶段的 `cmd/compile/internal/gc.state.stmt` 会负责处理程序中的 `defer`，该函数会根据条件的不同，使用三种不同的机制处理该关键字：

```
func (s *state) stmt(n *Node) {
 ...
 switch n.Op {
    ...
 case ODEFER:
  if s.hasOpenDefers {
   s.openDeferRecord(n.Left) // 开放编码
  } else {
   d := callDefer // 堆分配
   if n.Esc == EscNever {
    d = callDeferStack // 栈分配
   }
   s.callResult(n.Left, d)
  }
 }
}
```

> 堆分配、栈分配和开放编码是处理 `defer` 关键字的三种机制，早期的 `Go` 语言会在堆上分配 `runtime._defer` 结构体，不过该实现的性能较差，`Go` 语言在 `1.13` 中引入栈上分配的结构体，减少了 30% 的额外开销，并在 `1.14` 中引入了基于开放编码的 `defer`，使得该关键字的额外开销可以忽略不计。

6.2 堆分配

从上述源码可以看出:堆分配是默认的兜底执行方案，当该方案被启用时，编译器不仅将 `defer` 关键字都转换成 `runtime.deferproc` 函数，它还会通过以下三个步骤为所有调用 `defer`

的函数末尾插入 runtime.deferreturn 的函数调用：

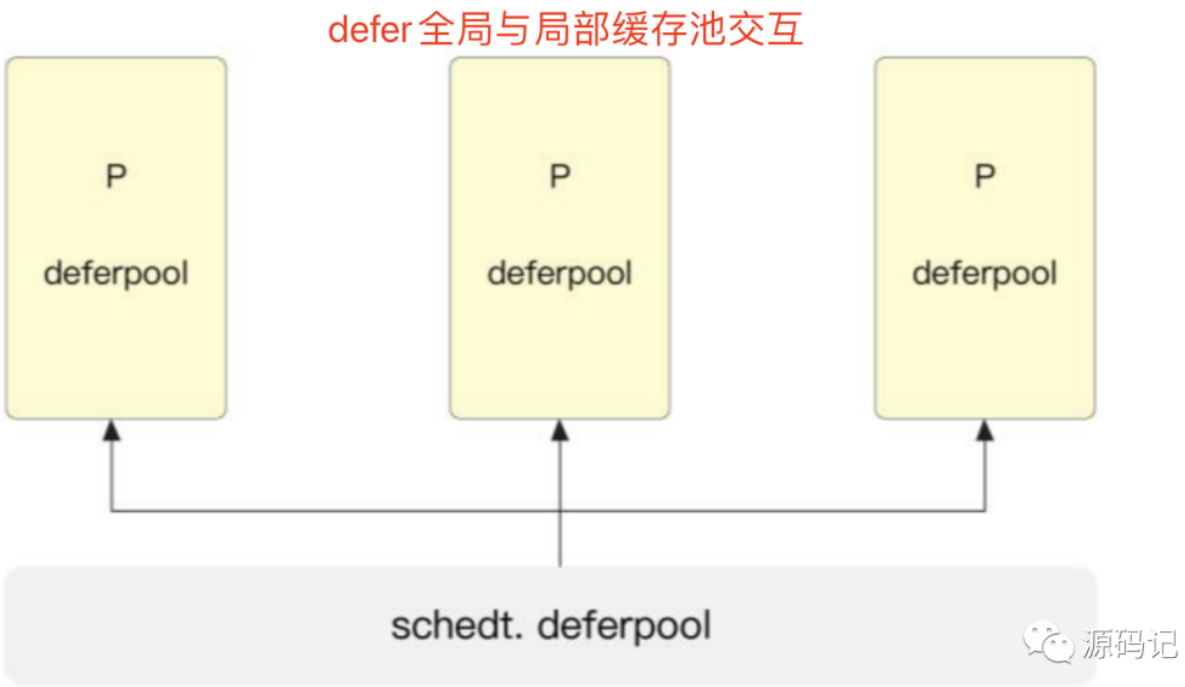
- 步骤一：在遇到 ODEFER 节点时会执行 Curfn.Func.SetHasDefer(true) 设置当前函数的 hasdefer 属性；实现代码位置：cmd/compile/internal/gc.walkstmt
- 步骤二：执行 s.hasdefer = fn.Func.HasDefer() 更新 state 的 hasdefer；实现代码位置：cmd/compile/internal/gc.buildssa
- 步骤三：根据 state 的 hasdefer 在函数返回之前插入 runtime.deferreturn 的函数调用；实现代码位置：cmd/compile/internal/gc.state.exit

> deferproc 和 deferreturn
>
> - runtime.deferproc 负责创建新的延迟调用；
> - runtime.deferreturn 负责在函数调用结束时执行所有的延迟调用；

6.2.1 申请_defer机制

runtime.deferproc 中 runtime.newdefer 的作用是想尽办法获得 runtime._defer 结构体，这里包含三种路径：

1. 从全局缓存池 sched.deferpool 中取出结构体并将该结构体追加到当前逻辑处理器 P 局部缓存池中；
2. 从逻辑处理器 P 局部缓存池 P.deferpool 中取出结构体；
3. 通过 runtime.mallocgc 在堆上创建一个新的结构体；



- 当 `defer` 执行完毕被销毁后，会重新回到局部缓存池中;
- 当局部缓存池容纳了足够的对象时，会将 `_defer` 结构体放入全局缓存池。
- 存储在全局和局部缓存池中的对象如果没有被使用，则最终在垃圾回收阶段被销毁。

6.3 栈分配

从 `defer` 堆分配的过程可以看出，即便有全局和局部缓存池策略，由于涉及堆与栈参数的复制等操作，堆分配仍然比直接调用效率低下。

`Go` 语言团队在 `1.13` 中对 `defer` 关键字进行了优化，当该关键字在函数体中最多执行一次时，编译期间的 `cmd/compile/internal/gc.state.call` 会将结构体分配到栈上并调用 `runtime.deferprocStack` 。

因为在编译期间我们已经创建了 `runtime._defer` 结构体，所以在运行期间 `runtime.deferprocStack` 只需要设置一些未在编译期间初始化的字段，就可以将栈上的 `runtime._defer` 追加到函数的链表上:

```
func deferprocStack(d *_defer) {
 gp := getg()
 d.started = false
```

```
    d.heap = false // 栈上分配的 _defer
    d.openDefer = false
    d.sp = getcallersp()
    d.pc = getcallerpc()
    d.framepc = 0
    d.varp = 0
    *(*uintptr)(unsafe.Pointer(&d._panic)) = 0
    *(*uintptr)(unsafe.Pointer(&d.fd)) = 0
    *(*uintptr)(unsafe.Pointer(&d.link)) = uintptr(unsafe.Pointer(gp._defer))
    *(*uintptr)(unsafe.Pointer(&gp._defer)) = uintptr(unsafe.Pointer(d))

    return0()
}
```

> 除了分配位置的不同，栈上分配和堆上分配的 `runtime._defer` 并没有本质的不同，而该方法可以适用于绝大多数的场景，与堆上分配的 `runtime._defer` 相比，该方法可以将 `defer` 关键字的额外开销降低大约30%。

6.4 开放编码

`Go` 语言在 `1.14` 中通过开放编码（`Open Coded`）实现 `defer` 关键字，该设计使用代码内联优化 `defer` 关键的额外开销并引入函数数据 `funcdata` 管理 `panic` 的调用，该优化可以将 `defer` 的调用开销从 `1.13` 版本的 `~35ns` 降低至 `~6ns` 左右：

```
With normal (stack-allocated) defers only:         35.4  ns/op
With open-coded defers:                             5.6  ns/op
Cost of function call alone (remove defer keyword): 4.4  ns/op
```

然而开放编码作为一种优化 `defer` 关键字的方法，它不是在所有的场景下都会开启的，开放编码只会在满足以下的条件时启用：

- 函数的 `defer` 数量少于或者等于 8 个；
- 函数的 `defer` 关键字不能在循环中执行；

- 函数的 `return` 语句与 `defer` 语句的乘积小于或者等于 `15` 个；

初看上述几个条件可能会觉得不明所以，但是当我们深入理解基于开放编码的优化就可以明白上述限制背后的原因。

6.4.1 defer <=8

`Go` 语言会在编译期间就确定是否启用开放编码，在编译器生成中间代码之前，我们会使用 `cmd/compile/internal/gc.walkstmt` 修改已经生成的抽象语法树，设置函数体上的 `OpenCodedDeferDisallowed` 属性：

```
const maxOpenDefers = 8

func walkstmt(n *Node) *Node {
 switch n.Op {
 case ODEFER:
  Curfn.Func.SetHasDefer(true)
  Curfn.Func.numDefers++
    //数量>8个禁用开放编码优化
  if Curfn.Func.numDefers > maxOpenDefers {
   Curfn.Func.SetOpenCodedDeferDisallowed(true)
  }
  if n.Esc != EscNever {
      //defer处于for循环中，会禁用开放编码优化
   Curfn.Func.SetOpenCodedDeferDisallowed(true)
  }
  fallthrough
 ...
 }
}
```

> 通过上述源码可以发现: 如果函数中 `defer` 关键字的数量多于 8 个或者 `defer` 关键字处于 `for` 循环中，那么我们在这里都会禁用开放编码优化

6.4.2 retun语句 * defer < 15

在 SSA 中间代码生成阶段的 cmd/compile/internal/gc.buildssa 中，我们也能够看到启用开放编码优化的其他条件，也就是返回语句的数量与 defer 数量的乘积需要小于 15：

```
func buildssa(fn *Node, worker int) *ssa.Func {
 ...
 s.hasOpenDefers = s.hasdefer && !s.curfn.Func.OpenCodedDeferDisallowed()
 ...
 if s.hasOpenDefers &&
  s.curfn.Func.numReturns*s.curfn.Func.numDefers > 15 {
    // 如果大于15 决定当前函数是否应该使用开放编码优化 defer 关键字
  s.hasOpenDefers = false
 }
 ...
}
```


收录于合集 #Go 101

上一篇 · Go底层探索(五):哈希表Map-扩容[下篇]