



下载APP



08 | Once: 一个简约而不简单的并发原语

2020-10-28 晁岳攀

Go 并发编程实战课

[进入课程 >](#)**讲述：安晓辉**

时长 15:50 大小 14.51M



你好，我是鸟窝。

这一讲我来讲一个简单的并发原语：Once。为什么要学习 Once 呢？我先给你答案：**Once 可以用来执行且仅仅执行一次动作，常常用于单例对象的初始化场景。**

那这节课，我们就从对单例对象进行初始化这件事儿说起。

初始化单例资源有很多方法，比如定义 package 级别的变量，这样程序在启动的时候就可以初始化：



```
1 package abc
```

[复制代码](#)

```
2 import time
3
4 var startTime = time.Now()
5
```

或者在 init 函数中进行初始化:

```
1 package abc
2
3 var startTime time.Time
4
5 func init() {
6     startTime = time.Now()
7 }
8
```

[复制代码](#)

又或者在 main 函数开始执行的时候, 执行一个初始化的函数:

```
1 package abc
2
3 var startTime time.Tim
4
5 func initApp() {
6     startTime = time.Now()
7 }
8 func main() {
9     initApp()
10 }
```

[复制代码](#)

这三种方法都是线程安全的, 并且后两种方法还可以根据传入的参数实现定制化的初始化操作。

但是很多时候我们是要延迟进行初始化的, 所以有时候单例资源的初始化, 我们会使用下面的方法:

```
1 package main
2
3 import (
```

[复制代码](#)

```
4     "net"
5     "sync"
6     "time"
7 )
8
9 // 使用互斥锁保证线程(goroutine)安全
10 var connMu sync.Mutex
11 var conn net.Conn
12
13 func getConn() net.Conn {
14     connMu.Lock()
15     defer connMu.Unlock()
16
17     // 返回已创建好的连接
18     if conn != nil {
19         return conn
20     }
21
22     // 创建连接
23     conn, _ = net.DialTimeout("tcp", "baidu.com:80", 10*time.Second)
24     return conn
25 }
26
27 // 使用连接
28 func main() {
29     conn := getConn()
30     if conn == nil {
31         panic("conn is nil")
32     }
33 }
```

这种方式虽然实现起来简单，但是有性能问题。一旦连接创建好，每次请求的时候还是得竞争锁才能读取到这个连接，这是比较浪费资源的，因为连接如果创建好之后，其实就不需要锁的保护了。怎么办呢？


这个时候就可以使用这一讲要介绍的 Once 并发原语了。接下来我会详细介绍 Once 的使用、实现和易错场景。

Once 的使用场景

sync.Once 只暴露了一个方法 Do，你可以多次调用 Do 方法，但是只有第一次调用 Do 方法时 f 参数才会执行，这里的 f 是一个无参数无返回值的函数。


```
1 func (o *Once) Do(f func())
```

因为当且仅当第一次调用 Do 方法的时候参数 f 才会执行，即使第二次、第三次、第 n 次调用时 f 参数的值不一样，也不会被执行，比如下面的例子，虽然 f1 和 f2 是不同的函数，但是第二个函数 f2 就不会执行。

 复制代码

```
1 package main
2
3
4 import (
5     "fmt"
6     "sync"
7 )
8
9 func main() {
10     var once sync.Once
11
12     // 第一个初始化函数
13     f1 := func() {
14         fmt.Println("in f1")
15     }
16     once.Do(f1) // 打印出 in f1
17
18     // 第二个初始化函数
19     f2 := func() {
20         fmt.Println("in f2")
21     }
22     once.Do(f2) // 无输出
23 }
```

因为这里的 f 参数是一个无参数无返回的函数，所以你可能会通过闭包的方式引用外面的参数，比如：

 复制代码

```
1     var addr = "baidu.com"
2
3     var conn net.Conn
4     var err error
5
6     once.Do(func() {
7         conn, err = net.Dial("tcp", addr)
8     })
```

而且在实际的使用中，绝大多数情况下，你会使用闭包的方式去初始化外部的一个资源。

你看，Once 的使用场景很明确，所以，在标准库内部实现中也常常能看到 Once 的身影。

比如标准库内部 [cache](#) 的实现上，就使用了 Once 初始化 Cache 资源，包括 defaultDir 值的获取：

[复制代码](#)

```
1  func Default() *Cache { // 获取默认的Cache
2  defaultOnce.Do(initDefaultCache) // 初始化cache
3  return defaultCache
4  }
5
6  // 定义一个全局的cache变量，使用Once初始化，所以也定义了一个Once变量
7  var (
8      defaultOnce sync.Once
9      defaultCache *Cache
10 )
11
12 func initDefaultCache() { //初始化cache,也就是Once.Do使用的f函数
13     .....
14     defaultCache = c
15 }
16
17 // 其它一些Once初始化的变量，比如defaultDir
18 var (
19     defaultDirOnce sync.Once
20     defaultDir     string
21     defaultDirErr  error
22 )
23
24
```

还有一些测试的时候初始化测试的资源（[export_windows_test](#)）：

[复制代码](#)

```
1  // 测试window系统调用时区相关函数
2  func ForceAusFromTZIForTesting() {
3      ResetLocalOnceForTest()
4      // 使用Once执行一次初始化
5      localOnce.Do(func() { initLocalFromTZI(&aus) })
6  }
```

除此之外，还有保证只调用一次 `copyenv` 的 `envOnce`，`strings` 包下的 `Replacer`，`time` 包中的 [测试](#)，Go 拉取库时的 [proxy](#)，`net.pipe`，`crc64`，`Regexp`，...，数不胜数。我给你重点介绍一下很值得我们学习的 `math/big/sqrt.go` 中实现的一个数据结构，它通过 `Once` 封装了一个只初始化一次的值：

[复制代码](#)

```
1 // 值是3.0或者0.0的一个数据结构
2 var threeOnce struct {
3     sync.Once
4     v *Float
5 }
6
7 // 返回此数据结构的值，如果还没有初始化为3.0，则初始化
8 func three() *Float {
9     threeOnce.Do(func() { // 使用Once初始化
10         threeOnce.v = NewFloat(3.0)
11     })
12     return threeOnce.v
13 }
```

它将 `sync.Once` 和 `*Float` 封装成一个对象，提供了只初始化一次的值 `v`。你看它的 `three` 方法的实现，虽然每次都调用 `threeOnce.Do` 方法，但是参数只会被调用一次。

当你使用 `Once` 的时候，你也可以尝试采用这种结构，将值和 `Once` 封装成一个新的数据结构，提供只初始化一次的值。

总结一下 `Once` 并发原语解决的问题和使用场景：**`Once` 常常用来初始化单例资源，或者并发访问只需初始化一次的共享资源，或者在测试的时候初始化一次测试资源。**

了解了 `Once` 的使用场景，那应该怎样实现一个 `Once` 呢？

如何实现一个 `Once`？

很多人认为实现一个 `Once` 一样的并发原语很简单，只需使用一个 `flag` 标记是否初始化过即可，最多是用 `atomic` 原子操作这个 `flag`，比如下面的实现：

[复制代码](#)

```
1 type Once struct {
2     done uint32
3 }
4
5
6 func (o *Once) Do(f func()) {
7     if !atomic.CompareAndSwapUint32(&o.done, 0, 1) {
8         return
9     }
10    f()
11 }
```

这确实是一种实现方式，但是，这个实现有一个很大的问题，就是如果参数 `f` 执行很慢的话，后续调用 `Do` 方法的 goroutine 虽然看到 `done` 已经设置为执行过了，但是获取某些初始化资源的时候可能会得到空的资源，因为 `f` 还没有执行完。

所以，一个正确的 `Once` 实现要使用一个互斥锁，这样初始化的时候如果有并发的 goroutine，就会进入 `doSlow` 方法。互斥锁的机制保证只有一个 goroutine 进行初始化，同时利用双检查的机制（double-checking），再次判断 `o.done` 是否为 0，如果为 0，则是第一次执行，执行完毕后，就将 `o.done` 设置为 1，然后释放锁。

即使此时有多个 goroutine 同时进入了 `doSlow` 方法，因为双检查的机制，后续的 goroutine 会看到 `o.done` 的值为 1，也不会再次执行 `f`。

这样既保证了并发的 goroutine 会等待 `f` 完成，而且还不会多次执行 `f`。

[复制代码](#)

```
1 type Once struct {
2     done uint32
3     m     Mutex
4 }
5
6 func (o *Once) Do(f func()) {
7     if atomic.LoadUint32(&o.done) == 0 {
8         o.doSlow(f)
9     }
10 }
11
12
13 func (o *Once) doSlow(f func()) {
14     o.m.Lock()
15     defer o.m.Unlock()
16     // 双检查
17     if o.done == 0 {
```

```
18         defer atomic.StoreUint32(&o.done, 1)
19         f()
20     }
21 }
```

好了，到这里我们就了解了 Once 的使用场景，很明确，同时呢，也感受到 Once 的实现也是相对简单的。在实践中，其实很少会出现错误使用 Once 的情况，但是就像墨菲定律说的，凡是可能出错的事就一定会出错。使用 Once 也有可能出现两种错误场景，尽管非常罕见。我这里提前讲给你，咱打个预防针。

使用 Once 可能出现的 2 种错误

第一种错误：死锁

你已经知道了 Do 方法会执行一次 f，但是如果 f 中再次调用这个 Once 的 Do 方法的话，就会导致死锁的情况出现。这还不是无限递归的情况，而是的确确的 Lock 的递归调用导致的死锁。

[复制代码](#)

```
1 func main() {
2     var once sync.Once
3     once.Do(func() {
4         once.Do(func() {
5             fmt.Println("初始化")
6         })
7     })
8 }
```


当然，想要避免这种情况的出现，就不要在 f 参数中调用当前的这个 Once，不管是直接的还是间接的。

第二种错误：未初始化

如果 f 方法执行的时候 panic，或者 f 执行初始化资源的时候失败了，这个时候，Once 还是会认为初次执行已经成功了，即使再次调用 Do 方法，也不会再次执行 f。

比如下面的例子，由于一些防火墙的原因，googleConn 并没有被正确的初始化，后面如果想当然认为既然执行了 Do 方法 googleConn 就已经初始化的话，会抛出空指针的错


误:

 复制代码

```
1 func main() {
2     var once sync.Once
3     var googleConn net.Conn // 到Google网站的一个连接
4
5     once.Do(func() {
6         // 建立到google.com的连接, 有可能因为网络的原因, googleConn并没有建立成功, 此时
7         googleConn, _ = net.Dial("tcp", "google.com:80")
8     })
9     // 发送http请求
10    googleConn.Write([]byte("GET / HTTP/1.1\r\nHost: google.com\r\n Accept: */
11    io.Copy(os.Stdout, googleConn)
12 }
```

既然执行过 `Once.Do` 方法也可能因为函数执行失败的原因未初始化资源, 并且以后也没机会再次初始化资源, 那么这种初始化未完成的问题该怎么解决呢?

这里我来告诉你一招独家秘笈, 我们可以**自己实现一个类似 `Once` 的并发原语**, 既可以返回当前调用 `Do` 方法是否正确完成, 还可以在初始化失败后调用 `Do` 方法再次尝试初始化, 直到初始化成功才不再初始化了。

 复制代码

```
1 // 一个功能更加强大的Once
2 type Once struct {
3     m sync.Mutex
4     done uint32
5 }
6 // 传入的函数f有返回值error, 如果初始化失败, 需要返回失败的error
7 // Do方法会把这个error返回给调用者
8 func (o *Once) Do(f func() error) error {
9     if atomic.LoadUint32(&o.done) == 1 { //fast path
10        return nil
11    }
12    return o.slowDo(f)
13 }
14 // 如果还没有初始化
15 func (o *Once) slowDo(f func() error) error {
16     o.m.Lock()
17     defer o.m.Unlock()
18     var err error
19     if o.done == 0 { // 双检查, 还没有初始化
20         err = f()
```

```
21         if err == nil { // 初始化成功才将标记置为已初始化
22             atomic.StoreUint32(&o.done, 1)
23         }
24     }
25     return err
26 }
```

我们所做的改变就是 Do 方法和参数 f 函数都会返回 error，如果 f 执行失败，会把这个错误信息返回。

对 slowDo 方法也做了调整，如果 f 调用失败，我们不会更改 done 字段的值，这样后续 degoroutine 还会继续调用 f。如果 f 执行成功，才会修改 done 的值为 1。

可以说，真是一顿操作猛如虎，我们使用 Once 有点得心应手的感觉了。等等，还有个问题，我们怎么查询是否初始化过呢？


目前的 Once 实现可以保证你调用任意次数的 once.Do 方法，它只会执行这个方法一次。但是，有时候我们需要打一个标记。如果初始化后我们就去执行其它的操作，标准库的 Once 并不会告诉你是否初始化完成了，只是让你放心大胆地去执行 Do 方法，所以，**你还需要一个辅助变量，自己去检查是否初始化过了**，比如通过下面的代码中的 inited 字段：

[复制代码](#)

```
1  type AnimalStore struct {once    sync.Once;inited uint32}
2  func (a *AnimalStore) Init() // 可以被并发调用
3      a.once.Do(func() {
4          longOperationSetupDbOpenFilesQueuesEtc()
5          atomic.StoreUint32(&a.inited, 1)
6      })
7  }
8  func (a *AnimalStore) CountOfCats() (int, error) { // 另外一个goroutine
9      if atomic.LoadUint32(&a.inited) == 0 { // 初始化后才会执行真正的业务逻辑
10         return 0, NotYetInitedError
11     }
12         //Real operation
13 }
```

当然，通过这段代码，我们可以解决这类问题，但是，如果官方的 Once 类型有 Done 这样一个方法的话，我们就可以直接使用了。这是有人在 Go 代码库中提出的一个

issue([🔗 #41690](#))。对于这类问题，一般都会被建议采用其它类型，或者自己去扩展。我们可以尝试扩展这个并发原语：

 复制代码

```
1 // Once 是一个扩展的sync.Once类型，提供了一个Done方法
2 type Once struct {
3     sync.Once
4 }
5
6 // Done 返回此Once是否执行过
7 // 如果执行过则返回true
8 // 如果没有执行过或者正在执行，返回false
9 func (o *Once) Done() bool {
10     return atomic.LoadUint32((*uint32)(unsafe.Pointer(&o.Once))) == 1
11 }
12
13 func main() {
14     var flag Once
15     fmt.Println(flag.Done()) //false
16
17     flag.Do(func() {
18         time.Sleep(time.Second)
19     })
20
21     fmt.Println(flag.Done()) //true
22 }
```

好了，到这里关于并发原语 Once 的内容我讲得就差不多了。最后呢，和你分享一个 Once 的踩坑案例。

其实啊，使用 Once 真的不容易犯错，想犯错都很困难，因为很少有人会傻傻地在初始化函数 f 中递归调用 f，这种死锁的现象几乎不会发生。另外如果函数初始化不成功，我们一般会 panic，或者在使用的时候做检查，会及早发现这个问题，在初始化函数中加强代码。

所以查看大部分的 Go 项目，几乎找不到 Once 的错误使用场景，不过我还是发现了一个。这个 issue 先从另外一个需求 ([🔗 go#25955](#)) 谈起。

Once 的踩坑案例

go#25955 有网友提出一个需求，希望 Once 提供一个 Reset 方法，能够将 Once 重置为初始化的状态。比如下面的例子，St 通过两个 Once 控制它的 Open/Close 状态。但是在 Close 之后再调用 Open 的话，不会再执行 init 函数，因为 Once 只会执行一次初始化函数。

[复制代码](#)

```
1 type St struct {
2     openOnce *sync.Once
3     closeOnce *sync.Once
4 }
5
6 func(st *St) Open(){
7     st.openOnce.Do(func() { ... }) // init
8     ...
9 }
10
11 func(st *St) Close(){
12     st.closeOnce.Do(func() { ... }) // deinit
13     ...
14 }
```

所以提交这个 Issue 的开发者希望 Once 增加一个 Reset 方法，Reset 之后再调用 once.Do 又可以初始化了。

Go 的核心开发者 Ian Lance Taylor 给他了一个简单的解决方案。在这个例子中，只使用一个 ponce *sync.Once 做初始化，Reset 的时候给 ponce 这个变量赋值一个新的 Once 实例即可 (ponce = new(sync.Once))。Once 的本意就是执行一次，所以 Reset 破坏了这个并发原语的本意。

这个解决方案一点都没问题，可以很好地解决这位开发者的需求。Docker 较早的版本 (1.11.2) 中使用了它们的一个网络库 libnetwork，这个网络库在使用 Once 的时候就使用 Ian Lance Taylor 介绍的方法，但是不幸的是，它的 Reset 方法中又改变了 Once 指针的值，导致程序 panic 了。原始逻辑比较复杂，一个简化版可重现的 [代码](#)如下：

[复制代码](#)

```
1 package main
2
3 import (
4     "fmt"
5     "sync"
```

```
6     "time"
7 )
8
9 // 一个组合的并发原语
10 type MuOnce struct {
11     sync.RWMutex
12     sync.Once
13     mtime time.Time
14     vals []string
15 }
16
17 // 相当于reset方法, 会将m.Once重新复制一个Once
18 func (m *MuOnce) refresh() {
19     m.Lock()
20     defer m.Unlock()
21     m.Once = sync.Once{}
22     m.mtime = time.Now()
23     m.vals = []string{m.mtime.String()}
24 }
25
26 // 获取某个初始化的值, 如果超过某个时间, 会reset Once
27 func (m *MuOnce) strings() []string {
28     now := time.Now()
29     m.RLock()
30     if now.After(m.mtime) {
31         defer m.Do(m.refresh) // 使用refresh函数重新初始化
32     }
33     vals := m.vals
34     m.RUnlock()
35     return vals
36 }
37
38 func main() {
39     fmt.Println("Hello, playground")
40     m := new(MuOnce)
41     fmt.Println(m.strings())
42     fmt.Println(m.strings())
43 }
```

如果你执行这段代码就会 panic:

```

Hello, playground
fatal error: sync: unlock of unlocked mutex

goroutine 1 [running]:
runtime.throw(0x4cd034, 0x1e)
    /usr/local/go-faketime/src/runtime/panic.go:1116 +0x72 fp=0xc000187d80 sp=0xc000187d50 pc=0x432532
sync.throw(0x4cd034, 0x1e)
    /usr/local/go-faketime/src/runtime/panic.go:1102 +0x35 fp=0xc000187da0 sp=0xc000187d80 pc=0x45f535
sync.(*Mutex).unlockSlow(0xc00005219c, 0xc0ffffff)
    /usr/local/go-faketime/src/sync/mutex.go:196 +0xd8 fp=0xc000187dc8 sp=0xc000187da0 pc=0x471d58
sync.(*Mutex).Unlock(0xc00005219c)
    /usr/local/go-faketime/src/sync/mutex.go:190 +0x48 fp=0xc000187de8 sp=0xc000187dc8 pc=0x471c68
sync.(*Once).doSlow(0xc000052198, 0xc000068ec8)
    /usr/local/go-faketime/src/sync/once.go:68 +0x9a fp=0xc000187e38 sp=0xc000187de8 pc=0x471e7a
sync.(*Once).Do(0xc000052198, 0xc000068ec8)
    /usr/local/go-faketime/src/sync/once.go:57 +0x45 fp=0xc000187e58 sp=0xc000187e38 pc=0x471dc5
main.(*MuOnce).strings(0xc000052180, 0x0, 0x0, 0x0)
    /tmp/sandbox855343333/prog.go:32 +0x186 fp=0xc000187f00 sp=0xc000187e58 pc=0x4a3a06
main.main()
    /tmp/sandbox855343333/prog.go:38 +0x9c fp=0xc000187f88 sp=0xc000187f00 pc=0x4a3b5c

```

原因在于第 31 行执行 `m.Once.Do` 方法的时候，使用的是 `m.Once` 的指针，然后调用 `m.refresh`，在执行 `m.refresh` 的时候 `Once` 内部的 `Mutex` 首先会加锁（可以再翻看一下这一讲的 `Once` 的实现原理），但是，在 `refresh` 中更改了 `Once` 指针的值之后，结果在执行完 `refresh` 释放锁的时候，释放的是一个刚初始化未加锁的 `Mutex`，所以就 panic 了。

如果你还不太明白，我再给你简化成一个更简单的例子：

[复制代码](#)

```

1 package main
2
3
4 import (
5     "sync"
6 )
7
8 type Once struct {
9     m sync.Mutex
10 }
11
12 func (o *Once) doSlow() {
13     o.m.Lock()
14     defer o.m.Unlock()
15
16     // 这里更新的o指针的值!!!!!!!，会导致上一行Unlock出错
17     *o = Once{}
18 }
19
20 func main() {
21     var once Once
22     once.doSlow()
23 }

```

doSlow 方法就演示了这个错误。Ian Lance Taylor 介绍的 Reset 方法没有错误，但是你在使用的时候千万别再初始化函数中 Reset 这个 Once，否则势必会导致 Unlock 一个未加锁的 Mutex 的错误。

总的来说，这还是对 Once 的实现机制不熟悉，又进行复杂使用导致的错误。不过最新版的 libnetwork 相关的地方已经去掉了 Once 的使用了。所以，我带你一起来看这个案例，主要目的还是想巩固一下我们对 Once 的理解。

总结

今天我们一起学习了 Once，我们常常使用它来实现单例模式。

单例是 23 种设计模式之一，也是常常引起争议的设计模式之一，甚至有人把它归为反模式。为什么说它是反模式呢，我拿标准库中的单例模式给你介绍下。

因为 Go 没有 immutable 类型，导致我们声明的全局变量都是可变的，别的地方或者第三方库可以随意更改这些变量。比如 package io 中定义了几个全局变量，比如 io.EOF：

```
1 var EOF = errors.New("EOF")
```

[复制代码](#)

因为它是一个 package 级别的变量，我们可以在程序中偷偷把它改了，这会导致一些依赖 io.EOF 这个变量做判断的代码出错。

```
1 io.EOF = errors.New("我们自己定义的EOF")
```

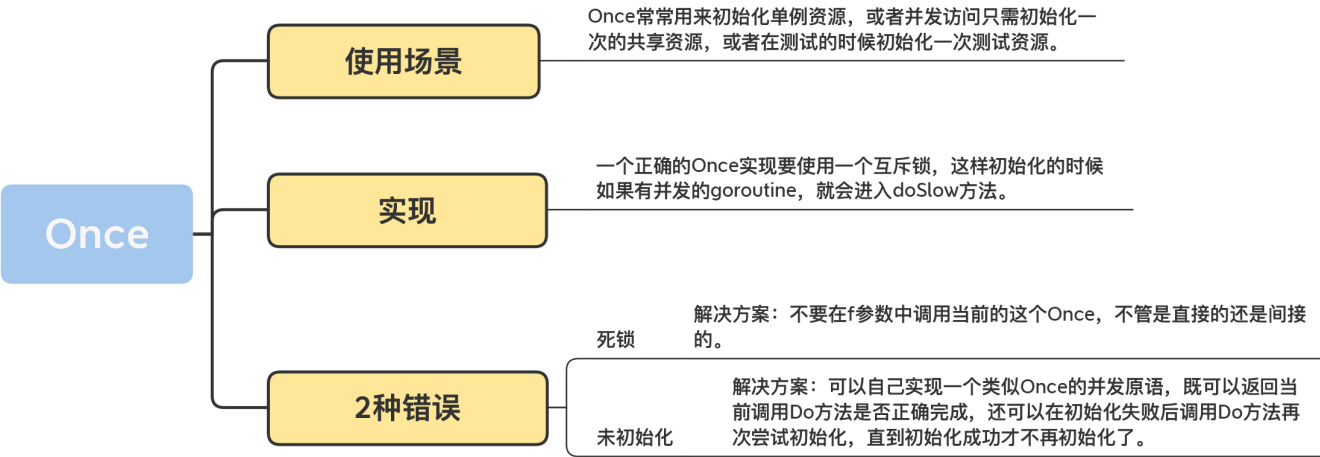
[复制代码](#)

从我个人的角度来说，一些单例（全局变量）的确很方便，比如 Buffer 池或者连接池，所以有时候我们也不要谈虎色变。虽然有人把单例模式称之为反模式，但毕竟只能代表一部分开发者的观点，否则也不会把它列在 23 种设计模式中了。

如果你真的担心这个 package 级别的变量被人修改，你可以不把它们暴露出来，而是提供一个只读的 GetXXX 的方法，这样别人就不会进行修改了。

而且，Once 不只应用于单例模式，一些变量在也需要在使用的时候做延迟初始化，所以也是可以使用 Once 处理这些场景的。

总而言之，Once 的应用场景还是很广泛的。**一旦你遇到只需要初始化一次的场景，首先想到的就应该是 Once 并发原语。**



思考题

1. 我已经分析了几个并发原语的实现，你可能注意到总是有些 slowXXXX 的方法，从 XXXX 方法中单独抽取出来，你明白为什么要这么做吗，有什么好处？
2. Once 在第一次使用之后，还能复制给其它变量使用吗？

欢迎在留言区写下你的思考和答案，我们一起交流讨论。如果你觉得有所收获，也欢迎你把今天的内容分享给你的朋友或同事。

提建议

Go 并发编程实战课

鸟窝带你攻克并发编程难题

晁岳攀 (鸟窝)

前微博技术专家

知名微服务框架 rpcx 的作者



新版升级: 点击「 请朋友读」, 20位好友免费读, 邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 07 | Cond: 条件变量的实现机制及避坑指南

精选留言 (4)

 写留言



Linuxer

2020-10-28

第一个思考题: Linux内核也有很多这种fast code path和slow code path, 我想这样划分是不是内聚性更好, 实现更清晰呢,从linux性能分析来看, 貌似更多关注点是在slow code path

第二个思考题: 应该不可以吧, Once的内部状态已经被改变了

展开 ∨

作者回复: fast path的一个好处是此方法可以内联



1



大漠胡萝卜

2020-10-28

```go

```
atomic.LoadUint32((*uint32)(unsafe.Pointer(&o.Once))) == 1
...
```

这个是怎么判断执行完的呢

展开



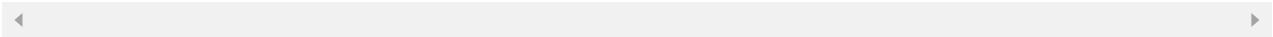
柠檬鱼也是鱼

2020-10-28

once为什么不直接加锁，还需要加多一个 双重检测呢？这块不太懂，望老师解答，我的理解是，调用do()之后直接上锁，等执行完f()再解锁不就行了吗

展开

作者回复: 因为有并发初始化的问题



橙子888

2020-10-28

打卡。

展开

