Go语言 sync包

Sync包中除了等待组WaitGroup、互斥锁Mutex、读写互斥锁（RWMutex）还有惰性初始化sync.Once，sync.Map，sync.Cond，sync.Pool等。

sync.Once

`sync.Once`可以保证程序运行期间某段代码只执行一次。

在编程的很多场景下我们需要确保某些操作在高并发的场景下只执行一次，例如只加载一次配置文件、只关闭一次通道等。

Go语言中的sync包中提供了一个针对只执行一次场景的解决方案–sync.Once。

sync.Once只有一个Do方法，其签名如下：

```
func (o *Once) Do(f func()) {}
```

注意：如果要执行的函数f需要传递参数就需要搭配闭包来使用。

```
func main() {
    o := sync.Once{}
    for i := 0; i < 10; i++ {
        o.Do(func() {
            fmt.Println("once")
        })
    }
}
#结果
once
```

作为用于保证函数执行次数的`sync.Once`结构体，使用了互斥锁`sync/atomic`包提供的方法实现了某个函数在程序运行期间只能执行一次的语义。

在使用该结构体时，也需要注意以下问题：

- `sync.Once.Do`方法中传入的函数只会被执行一次，哪怕函数中发生了`panic`
- 两次调用`sync.Once.Do`方法传入不同的函数只会执行第一次传入的函数

sync.Map

原生的 Go Map 在并发读写场景下经常会遇到 panic 的情况。造成的原因是 map 是非线性安全的，并发读写过程中 map 的数据会被写乱。

而一般情况下，解决并发读写 map 的思路是加锁，或者把一个 map 切分成若干个小 map，对 key 进行哈希。
在业界中使用最多并发指出的模式分别是：

- 原生 map + 互斥锁 或者 读写锁
- 标准库 sync.Map (Go 1.9 及之后)

Go 语言原生 map 是非线性安全的，对 map 进行并发读写操作时，需要加锁。在 Go 1.9 引入 sync.map，一种并发安全的 map。

- sync.map 是线性安全的，读取、插入、删除都保持常数级的时间复杂度。
- sync.map 的零值是有效的，并且零值是一个空的 map。在第一次使用之后，不允许被拷贝
- 多个 goroutine 的并发使用是安全的，不需要额外的锁或协程
- 大多数代码应该使用原生的 map，而不是单独的锁或协程控制，以获得更好的类型安全性和维护性

示例：

```
package main

import (
    "fmt"
    "sync"
)

func main()  {
    var m sync.Map
    // 1. 写入
    m.Store("test", 18)
    m.Store("mo", 20)

    // 2. 读取
    age, _ := m.Load("test")
    fmt.Println(age.(int))

    // 3. 遍历
    m.Range(func(key, value interface{}) bool {
        name := key.(string)
        age := value.(int)
        fmt.Println(name, age)
        return true
    })

    // 4. 删除
    m.Delete("test")
    age, ok := m.Load("test")
    fmt.Println(age, ok)

    // 5. 读取或写入
    m.LoadOrStore("mo", 100)
    age, _ = m.Load("mo")
    fmt.Println(age)
}
```

sync.map 适用于读多写少的场景。对于写多的场景，会导致 read map 缓存失效，需要加锁，导致冲突变多；而且由于未命中 read map 次数过多，导致 dirty map 提升为 read map，这是一个 O(N) 的操作，会进一步降低性能。

sync包中提供了一个**开箱即用**的并发安全版map–sync.Map。开箱即用表示不用像内置的map一样使用make函数初始化就能直接使用。同时sync.Map内置了诸如Store、Load、LoadOrStore、Delete、Range等操作方法。

声明

声明一个**变量**名为 myMapName 的 sync.Map。

```
var myMapName sync.Map
```

> Go 语言 sync.Map 无须初始化，直接声明即可使用

Store 增加元素

```
var myMapName sync.Map      //声明一个student Map元素为 <name, age>

myMapName.Store("xiaoming", "6")
myMapName.Store("xiaoqiang", "7")
myMapName.Store("xiaohei", "7")
fmt.Println(myMapName)
```

Load 查找元素

查找返回两个结果，分别是 value 值和状态（true 或者 false），因为返回的 value 是 interface 类型的，因此 value 我们不可以直接使用，而必须要转换之后才可以使用，返回的 bool 值，表明获取是否成功。

针对 sync.Map 通过 Load 获取不存在的元素时，返回 nil 和 false。

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var myMapName sync.Map //声明一个student Map元素为 <name, age>

    myMapName.Store("xiaoming", "6")
    myMapName.Store("xiaoqiang", "7")
    myMapName.Store("xiaohei", "7")
    fmt.Println(myMapName)

    myMapValue, isFind := myMapName.Load("xiaoming")
    if isFind {
        fmt.Println("myMapValue=", myMapValue) // myMapValue= 6
    } else {
        fmt.Println("Failed to find xiaoming")
    }

    myMapValue, isFind = myMapName.Load("xiaoqiang")
    if isFind {
        fmt.Println("myMapValue=", myMapValue) // myMapValue= 7
    } else {
        fmt.Println("Failed to find xiaoqiang")
    }
```

```
    myMapValue, isFind = myMapName.Load("xiao")
    if isFind {
        fmt.Println("myMapValue=", myMapValue) // Failed to find xiao
    } else {
        fmt.Println("Failed to find xiao")
    }
}
```

Range 遍历

sync.Map 的元素遍历，不能使用 for 循环 或者 for range 循环，要使用 Range 方法并配合一个回调函数进行遍历操作。通过回调函数返回遍历出来的键值对（即 key 和 value ）。

- 当需要继续迭代遍历时，Range 参数中回调函数的返回 true；
- 当需要终止迭代遍历时，Range 参数中回调函数的返回 false。

```
package main

import (
    "fmt"
    "strconv"
    "sync"
)

func main() {
    var myMapName sync.Map //声明一个student Map元素为 <name, age>

    myMapName.Store("xiaoming", "6")
    myMapName.Store("xiaoqiang", "7")
    myMapName.Store("xiaohei", "7")

    var MyName string
    var MyValue uint64

    myMapName.Range(func(key, value interface{}) bool {
        //fmt.Println("key is ", key, "value is ", value)
        MyName = fmt.Sprintf("%v", key)
        MyValue, _ = strconv.ParseUint(fmt.Sprintf("%v", value), 10, 32)
        fmt.Println("key is ", key, "value is ", value)
        return true
    })
    fmt.Println("key is", MyName, "value is", MyValue)
}
#结果
key is  xiaoqiang value is  7
key is  xiaohei value is  7
key is  xiaoming value is  6
key is xiaoming value is 6
```

Delete 删除元素

```
myMapName.Delete("xiaoming")
myMapName.Delete("xiaoqiang")
fmt.Println(myMapName)                //打印当前myMapName 元素（key:value）
```

LoadOrStore 存在即查找否则增加

sync.Map的 LoadOrStore 表示，如果我们获取的 key 存在，那么就返回 key 对应的元素，如果获取的 key 不存在，那么就返回我们设置的值，并且将我们设置的值，存入 map。

语法：

```
func (m *Map) LoadOrStore(key, value interface{}) (actual interface{}, loaded
bool)
```

参数

参数	描述
m	sync.Map 对象。
key	map 中元素的键。
value	map 中元素的值。

返回值

返回值	描述
actual	获取到的值，或者被添加的值。
loaded	key 在 map 中是否存在。

在 map 中获取键为 key 的元素值，如果 key 存在，则将获取的值通过 actual 返回，loaded 返回 true。如果 key 不存在，那么就将 key 和 value 添加加 sync.Map，actual 返回 value，loaded 返回 false。

```
package main

import (
    "fmt"
    "strconv"
    "sync"
)

func main() {
    var myMapName sync.Map //声明一个student Map元素为 <name, age>

    myMapName.Store("xiaoming", "6")
    myMapName.Store("xiaoqiang", "7")
    myMapName.Store("xiaohei", "7")
```

```
    fmt.Println(myMapName.LoadOrStore("baozi", "18"))   //元素不存在
    fmt.Println(myMapName.LoadOrStore("xiaoming", "6")) //元素存在

    var MyName string
    var MyValue uint64

    myMapName.Range(func(key, value interface{}) bool {
        //fmt.Println("key is ", key, "value is ", value)
        MyName = fmt.Sprintf("%v", key)
        MyValue, _ = strconv.ParseUint(fmt.Sprintf("%v", value), 10, 32)
        fmt.Println("key is ", key, "value is ", value)
        return true
    })
    fmt.Println("key is", MyName, "value is", MyValue)
}
#结果
18 false
6 true
key is  xiaoming value is  6
key is  xiaoqiang value is  7
key is  xiaohei value is  7
key is  baozi value is  18
key is baozi value is 18
```

sync.Cond

sync 包中的 Cond 实现了一种条件变量，可以使用多个 Reader 等待公共资源。

每个 Cond 都会关联一个 Lock ,当修改条件或者调用 Wait 方法，必须加锁，保护 Condition。

sync.Cond 条件变量是用来协调想要共享资源的那些 goroutine, 当共享资源的状态发生变化时，可以被用来通知被互斥锁阻塞的 gorountine。

sync.Mutex 通常用来保护临界区和共享资源，条件变量 sync.Cond 用来协调想要访问的共享资源。

有一个协程正在接收数据，其他协程必须等待这个协程接收完数据，才能读取到正确的数据。

上述情形下，如果单纯的使用 channel 或者互斥锁，只能有一个协程可以等待，并读取到数据，没办法通知其他协程也读取数据。

这个时候怎么办?

- 可以用一个全局变量标识第一个协程是否接收数据完毕，剩下的协程反复检查该变量的值，直到读取到数据。
- 也可创建多个 channel, 每个协程阻塞在一个 Channel 上，由接收数据的协程在数据接收完毕后，挨个通知。

然后 Go 中其实内置来一个 sync.Cond 来解决这个问题。

sync.Cond 方法

- NewCond 创建实例

```
func NewCond(l Locker) *Cond
```

NewCond 创建实例需要关联一个锁。

- Broadcast 广播唤醒所有

```
func (c *Cond) Broadcast()
```

Broadcast 唤醒所有等待条件变量 c 的 goroutine，无需锁保护。

- Signal 唤醒一个协程

```
func (c *Cond) Signal()
```

Signal 只唤醒任意1个等待条件变量 c 的 goroutine，无需锁保护。方法会唤醒队列最前面的。

- Wait 等待，陷入休眠状态

```
func (c *Cond) Wait()
```

调用 Wait 会自动释放锁 c.L，并挂起调用者所在的 goroutine，因此当前协程会阻塞在 Wait 方法调用的地方。如果其他协程调用了 Signal 或 Broadcast 唤醒了该协程，Wait 方法结束阻塞时，并重新给 c.L 加锁，并且继续执行 Wait 后面的代码

示例

`sync.Cond` 是条件变量，可以让一组协程都在满足特定条件时被唤醒。每一个 `sync.Cond` 结构体在初始化时都需要传入一个互斥锁。

```
package main

import (
    "fmt"
    "os"
    "os/signal"
    "sync"
    "sync/atomic"
    "time"
)

var status uint32

func listen(c *sync.Cond) {
    c.L.Lock()                              // 加锁
    for atomic.LoadUint32(&status) != 1 { //原子操作
        c.Wait() // 等待并挂起调用者所在的协程
    }
    fmt.Println("listenning")
    c.L.Unlock() // 解锁
}

func broadcast(c *sync.Cond) {
    c.L.Lock()
    atomic.StoreUint32(&status, 1)
    c.Broadcast() // Broadcast 唤醒所有等待条件变量 c 的 goroutine
    c.L.Unlock()
```

```
}

func main() {
    // 创建一个实例
    c := sync.NewCond(&sync.Mutex{})
    for i := 0; i < 10; i++ {
        go listen(c) // 10个协程通过sync.Cond.Wait等待特定条件的满足
    }
    time.Sleep(time.Second)
    go broadcast(c) // 调用sync.Cond.Broadcast唤醒所有陷入等待的协程

    ch := make(chan os.Signal, 1)
    signal.Notify(ch, os.Interrupt)
    <-ch
}
```

调用`sync.Cond.Broadcast`方法后，上述代码会打印出 10 次 "listenning" 并结束调用。

`sync.Cond`不是一个常用的同步机制，但是在条件长时间无法满足时，与使用`for {}`进行忙碌等待相比，`sync.Cond`能够让出处理器的使用权，提供 CPU 的利用率。使用时需要注意以下问题：

- `Wait`在调用之前一定要上锁，否则会触发`panic`，程序崩溃
- `Signal`唤醒的 GG 都是队列最前面、等待最久的goroutine
- `Broadcast`会按照一定顺序广播通知等待的全部goroutine

sync.Pool

我们通常用golang来构建高并发场景下的应用，但是由于golang内建的GC机制会影响应用的性能，为了减少GC，golang提供了对象重用的机制，也就是sync.Pool对象池。 sync.Pool是可伸缩的，并发安全的。其大小仅受限于内存的大小，可以被看作是一个存放可重用对象的值的容器。 设计的目的是存放已经分配的但是暂时不用的对象，在需要用到的时候直接从pool中取。

任何存放区其中的值可以在任何时候被删除而不通知，在高负载下可以动态的扩容，在不活跃时对象池会收缩。

sync.Pool首先声明了两个结构体，如下：

```
// Local per-P Pool appendix.
type poolLocalInternal struct {
  private interface{} // Can be used only by the respective P.
  shared  poolChain   // Local P can pushHead/popHead; any P can popTail.
}

type poolLocal struct {
  poolLocalInternal

  // Prevents false sharing on widespread platforms with
  // 128 mod (cache line size) = 0 .
  pad [128 - unsafe.Sizeof(poolLocalInternal{})%128]byte
}
```

为了使得可以在多个goroutine中高效的使用并发，sync.Pool会为每个P(对应CPU，这里有点像GMP模型)都分配一个本地池，当执行Get或者Put操作的时候，会先将goroutine和某个P的对象池关联，再对该池进行操作。

每个P的对象池分为私有对象和共享列表对象，私有对象只能被特定的P访问，共享列表对象可以被任何P访问。因为同一时刻一个P只能执行一个goroutine，所以无需加锁，但是对共享列表对象进行操作时，因为可能有多个goroutine同时操作，即并发操作，所以需要加锁。

需要注意的是 poolLocal 结构体中有个 pad 成员，其目的是为了防止false sharing。cache使用中常见的一个问题是false sharing。当不同的线程同时读写同一个 cache line上不同数据时就可能发生false sharing。false sharing会导致多核处理器上严重的系统性能下降。

sync.Pool的Put和Get方法

sync.Pool 有两个公开的方法，一个是Get，另一个是Put。

Put方法

我们先来看一下Put方法的源码，如下：

```
// Put adds x to the pool.
func (p *Pool) Put(x interface{}) {
  if x == nil {
    return
  }
  if race.Enabled {
    if fastrand()%4 == 0 {
      // Randomly drop x on floor.
      return
    }
    race.ReleaseMerge(poolRaceAddr(x))
    race.Disable()
  }
  l, _ := p.pin()
  if l.private == nil {
    l.private = x
    x = nil
  }
  if x != nil {
    l.shared.pushHead(x)
  }
  runtime_procUnpin()
  if race.Enabled {
    race.Enable()
  }
}
```

阅读以上Put方法的源码可以知道： - 如果Put放入的值为空，则直接 return 了，不会执行下面的逻辑了； - 如果不为空，则继续检查当前goroutine的private是否设置对象池私有值，如果没有则将x赋值给该私有成员，并将x设置为nil； - 如果当前goroutine的private私有值已经被赋值过了，那么将该值追加到共享列表。

Get方法

我们再来看下Get方法的源码，如下：

```
func (p *Pool) Get() interface{} {
  if race.Enabled {
    race.Disable()
```

```
  }
  l, pid := p.pin()
  x := l.private
  l.private = nil
  if x == nil {
    // Try to pop the head of the local shard. We prefer
    // the head over the tail for temporal locality of
    // reuse.
    x, _ = l.shared.popHead()
    if x == nil {
      x = p.getSlow(pid)
    }
  }
  runtime_procUnpin()
  if race.Enabled {
    race.Enable()
    if x != nil {
      race.Acquire(poolRaceAddr(x))
    }
  }
  if x == nil && p.New != nil {
    x = p.New()
  }
  return x
}
```

阅读以上Get方法的源码，可以知道： - 首先尝试从本地P对应的那个对象池中获取一个对象值, 并从对象池中删掉该值。 - 如果从本地对象池中获取失败，则从共享列表中获取，并从共享列表中删除该值。 - 如果从共享列表中获取失败，则会从其它P的对象池中“偷”一个过来，并删除共享池中的该值（就是源码中14行的p.getSlow()）。 - 如果还是失败，那么直接通过 New() 分配一个返回值，注意这个分配的值不会被放入对象池中。New()是返回用户注册的New函数的值，如果用户未注册New，那么默认返回nil。

init函数

最后我们来看一下init函数，如下：

```
func init() {
  funtime_registerPoolCleanup(poolCleanup)
}
```

可以看到在init的时候注册了一个PoolCleanup函数，他会清除掉sync.Pool中的所有的缓存的对象，这个注册函数会在每次GC的时候运行，所以sync.Pool中的值只在两次GC中间的时段有效。

sync.Pool使用示例

示例代码：

```
package main
import (
    "fmt"
    "sync"
)
// 定义一个 Person 结构体，有Name和Age变量
type Person struct {
```

```
    Name string
    Age int
}
// 初始化sync.Pool，new函数就是创建Person结构体
func initPool() *sync.Pool {
    return &sync.Pool{
        New: func() interface{} {
            fmt.Println("创建一个 person.")
            return &Person{}
        },
    }
}
// 主函数，入口函数
func main() {
    pool := initPool()
    person := pool.Get().(*Person)
    fmt.Println("首次从sync.Pool中获取person：", person)
    person.Name = "Jack"
    person.Age = 23
    pool.Put(person)
    fmt.Println("设置的对象Name: ", person.Name)
    fmt.Println("设置的对象Age: ", person.Age)
    fmt.Println("Pool 中有一个对象，调用Get方法获取：", pool.Get().(*Person))
    fmt.Println("Pool 中没有对象了，再次调用Get方法：", pool.Get().(*Person))
}
```

运行结果如下所示：

```
创建一个 person.
首次从sync.Pool中获取person：&{ 0}
设置的对象Name:  Jack
设置的对象Age:  23
Pool 中有一个对象，调用Get方法获取：&{Jack 23}
创建一个 person.
Pool 中没有对象了，再次调用Get方法:  &{ 0}
```

通过以上的源码及其示例，我们可以知道： - Get方法并不会对获取到的对象值做任何的保证，因为放入本地对象池中的值有可能会在任何时候被删除，而得不到通知。 - 放入共享池中的值有可能被其他的goroutine拿走，所以对象池比较适合用来存储一些临时切状态无关的数据，但是不适合用来存储数据库连接的实例，因为存入对象池的值有可能会在垃圾回收时被删除掉，这违反了数据库连接池建立的初衷。

由此可知，Golang的对象池严格意义上来说是一个临时的对象池，适用于储存一些会在goroutine间分享的临时对象。主要作用是减少GC，提高性能。在Golang中最常见的使用场景就是fmt包中的输出缓冲区了。