

Mutex 互斥锁

在并发编程中，如果程序中的一部分会被并发访问或修改，那么，为了避免并发访问导致意想不到的结果，这部分程序需要被保护起来，这部分被保护起来的程序，就叫做临界区

临界区就是一个被共享的资源，或者说是一个整体的一组共享资源，比如对数据库的访问、对某一个共享数据结构的操作、对一个 I/O 设备的使用、对一个连接池中的连接的调用，等等

Mutex 的基本使用方法

互斥锁 Mutex 就提供两个方法 Lock 和 Unlock：进入临界区之前调用 Lock 方法，退出临界区的时候调用 Unlock 方法：

```
1 package main
2
3 import (
4     "fmt"
5     "sync"
6 )
7
8 func main() {
9     var counter MyCounter
10    // 使用WaitGroup等待10个goroutine完成
11    var wg sync.WaitGroup
12    wg.Add(10)
13    for i := 0; i < 10; i++ {
14        go func() {
15            defer wg.Done()
16            // 执行10万次累加
17            for j := 0; j < 100000; j++ {
18                counter.Incr() // 受到锁保护的方法
19            }
20        }()
21    }
22    wg.Wait()
23    fmt.Println(counter.Count())
24
25
26 }
27 // 一个线程安全的计数器
28 type MyCounter struct {
29     mu sync.Mutex
```

```

30     count uint64
31 }
32 // 使用写锁保护
33 func (c *MyCounter) Incr() {
34     c.mu.Lock()
35     c.count++
36     c.mu.Unlock()
37 }
38 // 使用读锁保护
39 func (c *MyCounter) Count() uint64 {
40     c.mu.Lock()
41     defer c.mu.Unlock()
42     return c.count
43 }
44

```

Mutex 的实现原理

字段 sema: 是个信号量变量，用来控制等待 goroutine 的阻塞休眠和唤醒

字段state: 32位，最低三位分别表示mutexLocked、mutexWoken、mutexStarving，所以剩下的位置，用来表示可以有 $1 < (32-3)$ 个Goroutine 等待互斥锁的释放

```

1  type Mutex struct {
2      state int32
3      sema uint32
4  }
5  const (
6      mutexLocked = 1 << iota // mutex is locked
7      mutexWoken           //唤醒标记2
8      mutexStarving        // 饥饿模式4
9      mutexWaiterShift = iota // 位移数3
10     starvationThresholdNs = 1e6 // 阻塞时间阈值1ms
11 )

```

100

010

001

Waiter

Starving

Woken

Locked

加锁流程:

```

1 func (m *Mutex) Lock() {

```

```

2 // Fast path: grab unlocked mutex.
3 if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
4     return
5 }
6 // Slow path (outlined so that the fast path can be inlined)
7 m.lockSlow()
8 }

```

加锁的时候，一开始会通过CAS看一下能不能直接获取锁，如果可以的话，那么直接获取锁成功

```

1 func (m *Mutex) lockSlow() {
2     var waitStartTime int64 // 等待时间
3     starving := false       // 饥饿标记
4     awoke := false          // 唤醒标记
5     iter := 0                // 自旋次数
6     old := m.state
7     for {
8         // Don't spin in starvation mode, ownership is handed off to waiters
9         // so we won't be able to acquire the mutex anyway.
10        if old&(mutexLocked|mutexStarving) == mutexLocked && runtime_canSpin(iter) {
11            //可以知道当前锁的状态必须是上锁，并且不能处于饥饿状态，这个判断才为true，然后再看看iter
            //是否满足次数的限制，如果都为true，那么则往下继续。
12            // Active spinning makes sense.
13            // Try to set mutexWoken flag to inform Unlock
14            // to not wake other blocked goroutines.
15            // 设置上的Woken标记
16            if !awoke && old&mutexWoken == 0 && old>>mutexWaiterShift != 0 &&
17                atomic.CompareAndSwapInt32(&m.state, old, old|mutexWoken) {
18                awoke = true
19            }
20            runtime_doSpin()
21            iter++
22            old = m.state
23            continue
24        }
25        ...
26    }

```

tip:按位与的运算规则是，如果两数对应的二进制位都为 1，那么结果为 1，否则结果为 0。按位或的运算规则是如果两数对应的二进制位有一个为 1，那么结果为 1，否则结果为 0

满足四个条件设置唤醒标记：

- ☐ 首先判断了awoke是不是唤醒状态；
- ☐ `old&mutexWoken == 0`为真表示没有其他正在唤醒的节点；
- ☐ `old>>mutexWaiterShift != 0`表明当前有正在等待的goroutine；
- ☐ CAS将state的mutexWoken状态位设置为`old|mutexWoken`，即为1是否成功

如果被标记，哪怕现在锁上面的阻塞队列不为空，也不做唤醒。直接return，让自旋着的goroutine有更大机会抢到锁。

```
1    new := old
2    // Don't try to acquire starving mutex, new arriving goroutines must queue.
3    if old&mutexStarving == 0 {
4        // 如果当前不是饥饿模式，那么将mutexLocked状态位设置1，表示加锁
5        new |= mutexLocked
6    }
7    if old&(mutexLocked|mutexStarving) != 0 {
8        // 如果当前被锁定或者处于饥饿模式，则waiter加一，表示等待一个等待计数
9        new += 1 << mutexWaiterShift
10   }
11   // The current goroutine switches mutex to starvation mode.
12   // But if the mutex is currently unlocked, don't do the switch.
13   // Unlock expects that starving mutex has waiters, which will not
14   // be true in this case.
15   if starving && old&mutexLocked != 0 {
16       // 如果当前goroutine是饥饿状态，并且old已经上锁了，那么mutexStarving状态位设置为1，将
       当前goroutine设置为饥饿状态
17       new |= mutexStarving
18   }
19   if awake {
20       // The goroutine has been woken from sleep,
21       // so we need to reset the flag in either case.
22       if new&mutexWoken == 0 {
23           throw("sync: inconsistent mutex state")
24       }
25       消除mutexWoken标志位
26       new &^= mutexWoken
27   }
```

消除mutexWoken标志位是因为后续流程很有可能当前线程会被挂起,就需要等待其他释放锁的goroutine来唤醒，如果unlock的时候发现mutexWoken的位置不是0，则就不会去唤醒，则该线程就无法再醒来加锁。

```

1    if atomic.CompareAndSwapInt32(&m.state, old, new) {
2        if old&(mutexLocked|mutexStarving) == 0 {
3            1.如果原来状态没有上锁，也没有饥饿，那么直接返回，表示获取到锁
4            break // locked the mutex with CAS
5        }
6        // If we were already waiting before, queue at the front of the queue.
7        // 2.判断waitStartTime是否已经初始化过,queueLifo为false，说明是新来的goroutine，
    如果是true，则是旧的goroutine
8        queueLifo := waitStartTime != 0
9        // 3.如果等待时间为0，那么初始化等待时间
10       if waitStartTime == 0 {
11           waitStartTime = runtime.nanotime()
12       }
13       // 4.阻塞等待，当前goroutine进行休眠（如果不是新的goroutine来抢占锁，那么加入到等待
    队列头部）
14       runtime.SemacquireMutex(&m.sema, queueLifo, 1)
15       // 5.唤醒之后检查锁是否应该处于饥饿状态（？）
16       starving = starving || runtime.nanotime()-waitStartTime >
starvationThresholdNs //
17       old = m.state
18       // 6.判断是否已经处于饥饿状态
19       if old&mutexStarving != 0 {
20           // 表明已经是饥饿状态
21           // If this goroutine was woken and mutex is in starvation mode,
22           // ownership was handed off to us but mutex is in somewhat
23           // inconsistent state: mutexLocked is not set and we are still
24           // accounted as waiter. Fix that.
25           if old&(mutexLocked|mutexWoken) != 0 || old>>mutexWaiterShift == 0 {
26               throw("sync: inconsistent mutex state")
27           }
28           // 7.加锁并且将waiter数减1
29           delta := int32(mutexLocked - 1<<mutexWaiterShift)
30           if !starving || old>>mutexWaiterShift == 1 {
31               // Exit starvation mode.
32               // Critical to do it here and consider wait time.
33               // Starvation mode is so inefficient, that two goroutines
34               // can go lock-step infinitely once they switch mutex
35               // to starvation mode.
36               // 8.如果当前goroutine不是饥饿状态，就从饥饿模式切换会正常模式
37               delta -= mutexStarving

```

```

38         }
39         atomic.AddInt32(&m.state, delta)
40         break
41     }
42     awoke = true
43     iter = 0
44 } else {
45     old = m.state
46 }
47 }

```

1. 首先会判断old状态，如果没有饥饿，也没有获取到锁，那么直接返回，因为这种情况在进入这段代码之前会将new状态设置为mutexLocked，表示已经获取到锁。这里还判断了一下old状态不能为饥饿状态，否则也不能获取到锁；
2. 判断waitStartTime是否已经初始化过了，如果是新的goroutine来抢占锁，那么queueLifo会返回false；如果不是新的goroutine来抢占锁，那么加入到等待队列头部，这样等待最久的 goroutine 优先能够获取到锁；
3. 被唤醒一开始是需要判断一下当前的starving状态以及等待的时间如果超过了1ms，那么会将starving设置为true；

解锁流程

```

1 func (m *Mutex) Unlock() {
2     if race.Enabled {
3         _ = m.state
4         race.Release(unsafe.Pointer(m))
5     }
6
7     // Fast path: drop lock bit.
8     new := atomic.AddInt32(&m.state, -mutexLocked)
9     if new != 0 {
10         // Outlined slow path to allow inlining the fast path.
11         // To hide unlockSlow during tracing we skip one extra frame when tracing
12         // GoUnblock.
13         m.unlockSlow(new)
14     }
15 }

```

主要就是AddInt32重新设置state的mutexLocked位为0，然后判断新的state值是否不为0，不为0则调用unlockSlow方法。

```

1 func (m *Mutex) unlockSlow(new int32) {

```

```

2  if (new+mutexLocked)&mutexLocked == 0 {
3      throw("sync: unlock of unlocked mutex")
4  }
5  if new&mutexStarving == 0 {
6      // 正常模式
7      old := new
8      for {
9          // If there are no waiters or a goroutine has already
10         // been woken or grabbed the lock, no need to wake anyone.
11         // In starvation mode ownership is directly handed off from unlocking
12         // goroutine to the next waiter. We are not part of this chain,
13         // since we did not observe mutexStarving when we unlocked the mutex above.
14         // So get off the way.
15         // 如果没有 waiter, 或者已经有在处理的情况, 直接返回
16         if old>>mutexWaiterShift == 0 || old&(mutexLocked|mutexWoken|mutexStarving) !=
0 {
17             return
18         }
19         // Grab the right to wake someone.
20         // waiter 数减 1, mutexWoken 标志设置上, 通过 CAS 更新 state 的值
21         new = (old - 1<<mutexWaiterShift) | mutexWoken
22         if atomic.CompareAndSwapInt32(&m.state, old, new) {
23             runtime_Semrelease(&m.sema, false, 1)
24             return
25         }
26         old = m.state
27     }
28 } else {
29     // Starving mode: handoff mutex ownership to the next waiter, and yield
30     // our time slice so that the next waiter can start to run immediately.
31     // Note: mutexLocked is not set, the waiter will set it after wakeup.
32     // But mutex is still considered locked if mutexStarving is set,
33     // so new coming goroutines won't acquire it.
34     // 饥饿模式 直接唤醒等待队列中的 waiter
35     runtime_Semrelease(&m.sema, true, 1)
36 }
37 }

```

在正常模式下, 如果没有 waiter, 或者mutexLocked、mutexStarving、mutexWoken有一个不为零说明已经有其他goroutine在处理了, 直接返回; 如果互斥锁存在等待者, 那么通过

runtime_Semrelease直接唤醒等待队列中的 waiter;

正常模式拥有更好的性能，因为即使有等待抢锁的 waiter，goroutine 也可以连续多次获取到锁。饥饿模式是对公平性和性能的一种平衡，它避免了某些 goroutine 长时间的等待锁。在饥饿模式下，优先对待的是那些一直在等待的 waiter。

饥饿模式和正常模式 是怎么去看？

03 | Mutex: 4种易错场景大盘点

Lock/Unlock 不是成对出现、Copy 已使用的 Mutex、重入和死锁

Mutex 不是可重入的锁

方案一：通过 hacker 的方式获取到 goroutine id，记录下获取锁的 goroutine id，它可以实现 Locker 接口。

方案二：调用 Lock/Unlock 方法时，由 goroutine 提供一个 token，用来标识它自己，而不是我们通过 hacker 的方式获取到 goroutine id，但是，这样一来，就不满足Locker 接口了

两个或两个以上的进程（或线程，goroutine）在执行过程中，因争夺共享资源而处于一种互相等待的状态，

死锁产生的必要条件:

- 1.互斥：至少一个资源是被排他性独享的，其他线程必须处于等待状态，直到资源被释放
- 2.持有和等待：goroutine 持有一个资源，并且还在请求其它 goroutine 持有的资源.
- 3.不可剥夺：资源只能由持有它的 goroutine 来释放。
- 4.环路等待：一般来说，存在一组等待进程， $P=\{P1, P2, \dots, PN\}$ ，P1 等待 P2 持有的资源，P2 等待 P3 持有的资源，依此类推，最后是 PN 等待 P1 持有的资源，这就形成了一个环路等待的死结。

解决死锁问题:

- 1.一次性申请，2.申请不到，主动释放，3.按序申请。

04 | Mutex: 骇客编程，如何拓展额外功能

实现 TryLock

获取等待者的数量等指标

实现一个线程安全的队列

RWMutex

实现方法:

Lock/Unlock: 写操作时调用的方法

RLock/RUnlock: 读操作时调用的方法

RLocker: 这个方法的作用是为读操作返回一个 Locker 接口的对象

如果你遇到可以明确区分 reader 和 writer goroutine 的场景，且有大量的并发读、少量的并发写，并且有强烈的性能需求，你就可以考虑使用读写锁 RWMutex 替换 Mutex

实现原理

RWMutex 包含一个 Mutex，以及四个辅助字段 writerSem、readerSem、readerCount 和 readerWait

```
1 type RWMutex struct {
2     w Mutex // 互斥锁解决多个writer的竞争
3     writerSem uint32 // writer信号量
4     readerSem uint32 // reader信号量
5     readerCount int32 // reader的数量
6     readerWait int32 // writer等待完成的reader的数量
7 }
8 const rwmutexMaxReaders = 1 << 30
```

rwmutexMaxReaders，定义了最大的 reader 数量

RLock/RUnlock 的实现

1.如果有读的goroutine，则将readerCount + 1，调用 RUnlock 的时候，我们需要将 Reader 的计数减去 1

2.如果有 writer 竞争锁或者持有锁时，那么，readerCount 不仅仅承担着 reader的计数功能，还能够标识当前是否有 writer 竞争或持有锁。在这种情况下，阻塞等待锁的释放。

readerCount 这个字段有双重含义

没有 writer 竞争或持有锁时，readerCount 和我们正常理解的 reader 的计数是一样的；但是，如果有 writer 竞争锁或者持有锁时，那么，readerCount 不仅仅承担着 reader的计数功能，还能够标识当前是否有 writer 竞争或持有锁

3.如果它是负值，就表示当前有 writer 竞争锁，在这种情况下，还会调用 rUnlockSlow 方法 (readerWait-1)，检查是不是reader 都释放读锁了，如果读锁都释放了，那么可以唤醒请求写锁的 writer。

```
1 func (rw *RWMutex) RLock() {
2     if atomic.AddInt32(&rw.readerCount, 1) < 0 {
3         // A writer is pending, wait for it.
4         runtime_SemacquireMutex(&rw.readerSem, false, 0)
5     }
6 }
7 func (rw *RWMutex) RUnlock() {
```

```

8     if r := atomic.AddInt32(&rw.readerCount, -1); r < 0 {
9         // Outlined slow-path to allow the fast-path to be inlined
10        rw.rUnlockSlow(r)
11    }
12 }
13 func (rw *RWMutex) rUnlockSlow(r int32) {
14     if r+1 == 0 || r+1 == -rwmutexMaxReaders {
15         race.Enable()
16         throw("sync: RUnlock of unlocked RWMutex")
17     }
18     // A writer is pending.
19     if atomic.AddInt32(&rw.readerWait, -1) == 0 {
20         // The last reader unblocks the writer.
21         runtime_Semrelease(&rw.writerSem, false, 1)
22     }
23 }

```

Lock()实现逻辑:

1. 首先获取互斥锁，解决其他write竞争的问题
2. 其次反转readerCount 字段，把它从原来的正整数 readerCount(>=0) 修改为负数 (readerCount-rwmutexMaxReaders) 让这个字段保持两个含义（既保存了 reader 的数量，又表示当前有 writer)

readercount = 2

max=30

readerCount = -28

r= readerCount+max = 2

3.如果 r 不是 0，就说明当前有持有读锁的 reader，RWMutex 需要把这个当前 readerCount 赋值给 readerWait 字段保存下来，同时，这个 writer 进入阻塞等待状态

4.每当一个 reader 释放读锁的时候（调用 RUnlock 方法时），readerWait 字段就减 1，直到所有的活跃的 reader 都释放了读锁，才会唤醒这个 writer

```

1 func (rw *RWMutex) Lock() {
2     // First, resolve competition with other writers.
3     rw.w.Lock()
4     // Announce to readers there is a pending writer.
5     r := atomic.AddInt32(&rw.readerCount, -rwmutexMaxReaders) + rwmutexMaxReaders
6     // Wait for active readers.
7     if r != 0 && atomic.AddInt32(&rw.readerWait, r) != 0 {
8         runtime_SemacquireMutex(&rw.writerSem, false, 0)
9     }

```

Unlock()实现逻辑:

1. 当一个 writer 释放锁的时候，它会再次反转 readerCount 字段，这里的反转方法就是给它增加 `rwmutexMaxReaders` 这个常数值
2. 既然 writer 要释放锁了，那么就需要唤醒之后新来的 reader
3. 在 `RWMutex` 的 `Unlock` 返回之前，需要把内部的互斥锁释放

```

1 func (rw *RWMutex) Unlock() {
2     if race.Enabled {
3         _ = rw.w.state
4         race.Release(unsafe.Pointer(&rw.readerSem))
5         race.Disable()
6     }
7
8     // Announce to readers there is no active writer.
9     r := atomic.AddInt32(&rw.readerCount, rwmutexMaxReaders)
10    if r >= rwmutexMaxReaders {
11        race.Enable()
12        throw("sync: Unlock of unlocked RWMutex")
13    }
14    // Unblock blocked readers, if any.
15    for i := 0; i < int(r); i++ {
16        runtime_Semrelease(&rw.readerSem, false, 0)
17    }
18    // Allow other writers to proceed.
19    rw.w.Unlock()
20 }

```

21 注意点：在 `Lock` 方法中，是先获取内部互斥锁，才会修改的其他字段；

22 而在 `Unlock` 方法中，是先修改的其他字段，才会释放内部互斥锁，这样才能保证字段的修改也受到互斥锁的保护。

WaitGroup

比如，我们要完成一个大的任务，需要使用并行的 goroutine 执行三个小任务，只有这三个小任务都完成，我们才能去执行后面的任务。如果通过轮询的方式定时询问三个小任务是否完成，会存在两个问题：一是，性能比较低，因为三个小任务可能早就完成了，却要等很长时间才被轮询到；二是，会有很多无谓的轮询，空耗 CPU 资源

`Add`，用来设置 `WaitGroup` 的计数值；

Done, 用来将 WaitGroup 的计数值减 1, 其实就是调用了 Add(-1)

Wait, 调用这个方法的 goroutine 会一直阻塞, 直到 WaitGroup 的计数值变为 0

实现原理

```
1 type WaitGroup struct {
2     // 避免复制使用的一个技巧, 可以告诉vet工具违反了复制使用的规则
3     noCopy noCopy
4     state1 [3]uint32
5 }
```

在 64 位环境下, state1 的第一个元素是 waiter 数, 第二个元素是 WaitGroup 的计数值, 第三个元素是信号量

在 32 位环境下, 那么 state1 的第一个元素是信号量, 后两个元素分别是 waiter 数和计数值。

// 补源码

Cond

Go 标准库提供 Cond 原语的目的是, 为等待 / 通知场景下的并发问题提供支持

Cond 通常应用于等待某个条件的一组 goroutine, 等条件变为 true 的时候, 其中一个 goroutine 或者所有的 goroutine 都会被唤醒执行。

补demo.go实例

基本用法:

sync.NewCond()方法: 返回一个带锁的**Cond**

Wait 方法, 会把调用者 Caller 放入 Cond 的等待队列中并阻塞, 直到被 Signal 或者Broadcast 的方法从等待队列中移除并唤醒

Broadcast 方法, 允许调用者 Caller 唤醒所有等待此 Cond 的 goroutine

Signal 方法, 允许调用者 Caller 唤醒一个等待此 Cond 的 goroutine

实现方法:

```
1 type Cond struct {
2     noCopy noCopy
3     L Locker // 当观察或者修改等待条件的时候需要加锁
4     notify notifyList // 等待队列
5     checker copyChecker // 是一个辅助结构, 可以在运行时检查 Cond 是否被复制使用。
6 }
```

Signal 和 Broadcast 只涉及到 notifyList 数据结构, 不涉及到锁。

Wait 把调用者加入到等待队列时会释放锁，在被唤醒之后还会请求锁。在阻塞休眠期间，调用者是不持有锁的，这样能让其他 goroutine 有机会检查或者更新等待变量,调用 cond.Wait 方法之前一定要加锁

// 检查一下--notifyList主要存的是什么？等待goroutine是怎么存储的

Cond 有三点特性是 Channel 无法替代的:

- 1.Cond 和一个 Locker 关联，可以利用这个 Locker 对相关的依赖条件更改提供保护
- 2.Cond 可以同时支持 Signal 和 Broadcast 方法，而 Channel 只能同时支持其中一种。
- 3.Cond 的 Broadcast 方法可以被重复调用。等待条件再次变成不满足的状态后，我们又可以调用 Broadcast 再次唤醒等待的 goroutine.

WaitGroup 和 Cond 是有区别的：WaitGroup 是主 goroutine 等待确定数量的子goroutine 完成任务；而 Cond 是等待某个条件满足，这个条件的修改可以被任意多的goroutine 更新，而且 Cond 的 Wait 不关心也不知道其他 goroutine 的数量，只关心等待条件

Once

使用场景：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

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

实现

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

```
type Once struct {
    done uint32
}

func (o *Once) Do(f func()) {
    if !atomic.CompareAndSwapUint32(&o.done, 0, 1) {
        return
    }
    f()
}
```

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

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

```
1 type Once struct {  
2     done uint32  
3     Mutex  
4 }
```

sync.Map

使用场景：

- 1.只会增长的缓存系统中，一个 key 只写入一次而被读很多次；
- 2.多个 goroutine 为不相交的键集读、写和重写键值对

基本方法：

Store、Load 和 Delete 这三个核心函数的操作都是先从 read 字段中处理的，因为读取read 字段的时候不用加锁

sync.Map 的实现

```
1 type Map struct {  
2     mu Mutex // 基本上你可以把它看成一个安全的只读的map  
3     // 它包含的元素其实也是通过原子操作更新的，但是已删除的entry就需要加锁操作了  
4     read atomic.Value  
5     // readOnly// 包含需要加锁才能访问的元素  
6     // 包括所有在read字段中但未被expunged（删除）的元素以及新加的元素  
7     dirty map[interface{}]*entry  
8     // 记录从read中读取miss的次数，一旦miss数和dirty长度一样了，就会把dirty提升为read  
9     misses int  
10 }
```

通过冗余的两个数据结构（只读的 read 字段、可写的 dirty），来减少加锁对性能的影响。对只读字段（read）的操作不需要加锁。优先从 read 字段读取、更新、删除，因为对 read 字段的读取不需要锁

sync.Map类型在内部使用了大量的原子操作来存取键和值，并使用了两个原生的map作为存储介质。其中一个原生map被存在了sync.Map的read字段中，该字段是sync/atomic.Value类型的。我们在后面简称它为只读字典。不过，只读字典虽然不会增减其中的键，但却允许变更其中的键所对应的值。由read字段的类型可知，sync.Map在替换只读字典的时候根本用不着锁。

sync.Map中的另一个原生字典由它的dirty字段代表。它存储键值对的方式与read字段中的原生字典一致，它的键类型也是interface{}。

sync.Map在查找指定的键所对应的值的时候，总会先去只读字典中寻找，并不需要锁定互斥锁。只有当确定“只读字典中没有，但脏字典中可能会有这个键”的时候，它才会在锁的保护下去访问脏字典。

sync.Map在存储键值对的时候，它才会在锁的保护下把键值对存储到脏字典中。

对于删除键值对，sync.Map会先去检查只读字典中是否有对应的键。存在于只读字典中的时候，才会被用标记为“已删除”的方式进行逻辑删除，而不会直接被物理删除。。如果没有，脏字典中可能有，那么它就会在锁的保护下，试图从脏字典中删掉该键值对。

只读字典和脏字典之间是会互相转换的。在脏字典中查找键值对次数足够多的时候，sync.Map会把脏字典直接作为只读字典，保存在它的read字段中，然后把代表脏字典的dirty字段的值置为nil。

在这之后，一旦再有新的键值对存入，它就会依据只读字典去重建脏字典。这个时候，它会把只读字典中已被逻辑删除的键值对过滤掉。

由于只读字典中键的集合不能被改变，所以其中的键值对有时候可能是不全的。相反，脏字典中的键值对集合总是完全的，并且其中不会包含已被逻辑删除的键值对。

因此，可以看出，在读操作有很多但写操作却很少的情况下，并发安全字典的性能往往会更好。在几个写操作当中，新增键值对的操作对并发安全字典的性能影响是最大的，其次是删除操作，最后才是修改操作。

Pool

sync.Pool 数据类型用来保存一组可独立访问的临时对象

Pool 可以有效地减少新对象的申请，从而提高程序性能，所以 Go 内部库也用到了sync.Pool，比如fmt包，它会使用一个动态大小的buffer池做输出缓存，当大量的goroutine并发输出的时候，就会创建比较多的buffer，并且在不需要的时候回收掉。

sync.pool特点:

- 1、sync.pool本身是线程安全的，多个goroutine可以并发的调用它的方法存取对象
- 2、sync.Pool不可在使用之后再复制使用

sync.pool的使用方法:

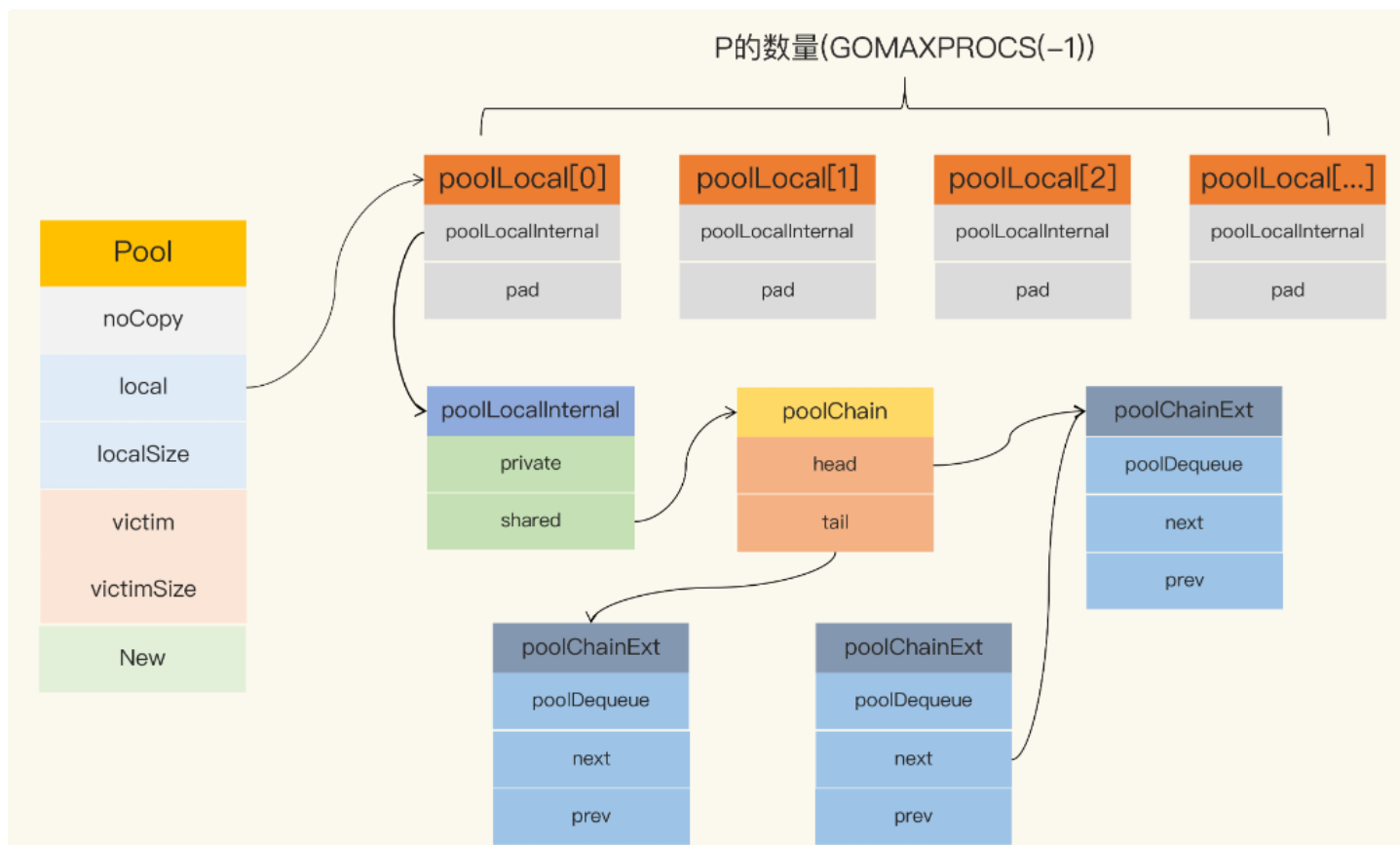
new: 当调用Pool的Get方法从池中获取元素，没有更多的空闲元素可返回时，就会调用这个New方法来创建新的元素。如果你没有设置New字段，没有更多的空闲元素可返回时，Get方法将返回

nil，表明当前没有可用的元素。

get: 调用这个方法，就会从 Pool取走一个元素，这也就意味着，这个元素会从 Pool 中移除，返回给调用者

put: 这个方法用于将一个元素返还给 Pool，Pool 会把这个元素保存到池中，并且可以复用。但如果 Put 一个 nil 值，Pool 就会忽略这个值

实现原理



每次垃圾回收的时候，Pool 会把 victim 中的对象移除，然后把 local 的数据给 victim，这样的话，local 就会被清空，而 victim 就像一个垃圾分拣站，里面的东西可能会被当做垃圾丢弃了，但是里面有用的东西也可能被捡回来重新使用

local 字段包含一个poolLocalInternal 字段，并提供 CPU 缓存对齐，从而避免 false sharing 而 poolLocalInternal 也包含两个字段：private 和 shared

private，代表一个缓存的元素，而且只能由相应的一个 P 存取。因为一个 P 同时只能执行一个 goroutine，所以不会有并发的问题

shared，可以由任意的 P 访问，但是只有本地的 P 才能 pushHead/popHead，其它 P 可以 popTail，相当于只有一个本地的 P 作为生产者（Producer），多个 P 作为消费者（Consumer），它是使用一个 local-free 的 queue 列表实现的

get函数的过程：

首先，从本地的 private 字段中获取可用元素，因为没有锁，获取元素的过程会非常快，如果没有获取到，就尝试从本地的 shared 获取一个，如果还没有，会使用 getSlow 方法去其它的 shared 中“偷”一个。最后，如果没有获取到，就尝试使用 New 函数创建一个新的。

这里的重点是 getSlow 方法，我们来分析下。看名字也就知道了，它的耗时可能比较长。它首先要遍历所有的 local，尝试从它们的 shared 弹出一个元素。如果还没找到一个，那么，就开始对 victim 下手了。在 victim 中查询可用元素的逻辑还是一样的，先从对应的 victim 的 private 查找，如果查不到，就再从其它 victim 的 shared 中查找。

put方法的逻辑：

Put 的逻辑相对简单，优先设置本地 private，如果 private 字段已经有值了，那么就把此元素 push 到本地队列中

第三方库：

1. [bytebufferpool](#)

2. [oxtoacart/bpool](#)

Context

应用场景：

1. 上下文信息传递（request-scoped），比如处理 http 请求、在请求处理链路上传递信息
2. 控制子 goroutine 的运行
3. 超时控制的方法调用
4. 可以取消的方法调用

基本使用方法：

Deadline：这个方法会返回这个 Context 被取消的截止日期。如果没有设置截止日期，ok 的值是 false

Done：Done 方法返回一个 Channel 对象，如果 Done 没有被 close，Err 方法返回 nil；如果 Done 被 close，Err 方法会返回 Done 被 close 的原因

Err：

Value：Value 返回此 ctx 中和指定的 key 相关联的 value

Context 中实现了 2 个常用的生成顶层 Context 的方法：

- 1、context.Background()：返回一个非 nil 的、空的 Context，没有任何值，不会被cancel，不会超时，没有截止日期。一般用在主函数、初始化、测试以及创建根Context 的时候。
- 2、context.TODO()：返回一个非 nil 的、空的 Context，没有任何值，不会被 cancel，不会超时，没有截止日期。当你不清楚是否该用 Context，或者目前还不知道要传递一些什么上下文信息的时候，就可以使用这个方法

使用规则：

- 1、一般函数使用 Context 的时候，会把这个参数放在第一个参数的位置
- 2、从来不把 nil 当做 Context 类型的参数值，可以使用 context.Background() 创建一个空的上下文对象，也不要使用 nil
- 3、Context 只用来临时做函数之间的上下文透传，不能持久化 Context 或者把 Context 长久保存。把 Context 持久化到数据库、本地文件或者全局变量、缓存中都是错误的用法

- 4、key 的类型不应该是字符串类型或者其它内建类型，否则容易在包之间使用 Context 时候产生冲突。使用 WithValue 时，key 的类型应该是自己定义的类型
- 5、常常使用 struct{}作为底层类型定义 key 的类型。对于 exported key 的静态类型，常常是接口或者指针。这样可以尽量减少内存分配

创建特殊用途 Context 的方法

- 1、WithValue 基于 parent Context 生成一个新的 Context，保存了一个 key-value 键值对。它常常用来传递上下文。
- 2、WithCancel 方法返回 parent 的副本，只是副本中的 Done Channel 是新建的对象，它的类型是 cancelCtx
- 3、WithTimeout 其实是和 WithDeadline 一样，只不过一个参数是超时时间，一个参数是截止时间。超时时间加上当前时间，其实就是截止时间
- 4、WithDeadline 会返回一个 parent 的副本，并且设置了一个不晚于参数 d 的截止时间，类型为 timerCtx

atomic

atomic 操作的对象是一个地址，你需要把可寻址的变量的地址作为参数传递给方法，而不是把变量的值传递给方法

add: Add 方法就是给第一个参数地址中的值增加一个 delta 值

cas: 这个方法会比较当前 addr 地址里的值是不是 old，如果不等于 old，就返回 false；如果等于 old，就把此地址的值替换成 new 值，返回 true。这就相当于“判断相等才替换”

Swap: 如果不需要比较旧值，只是比较粗暴地替换的话

Load: Load 方法会取出 addr 地址中的值

Store: Store 方法会把一个值存入到指定的 addr 地址中

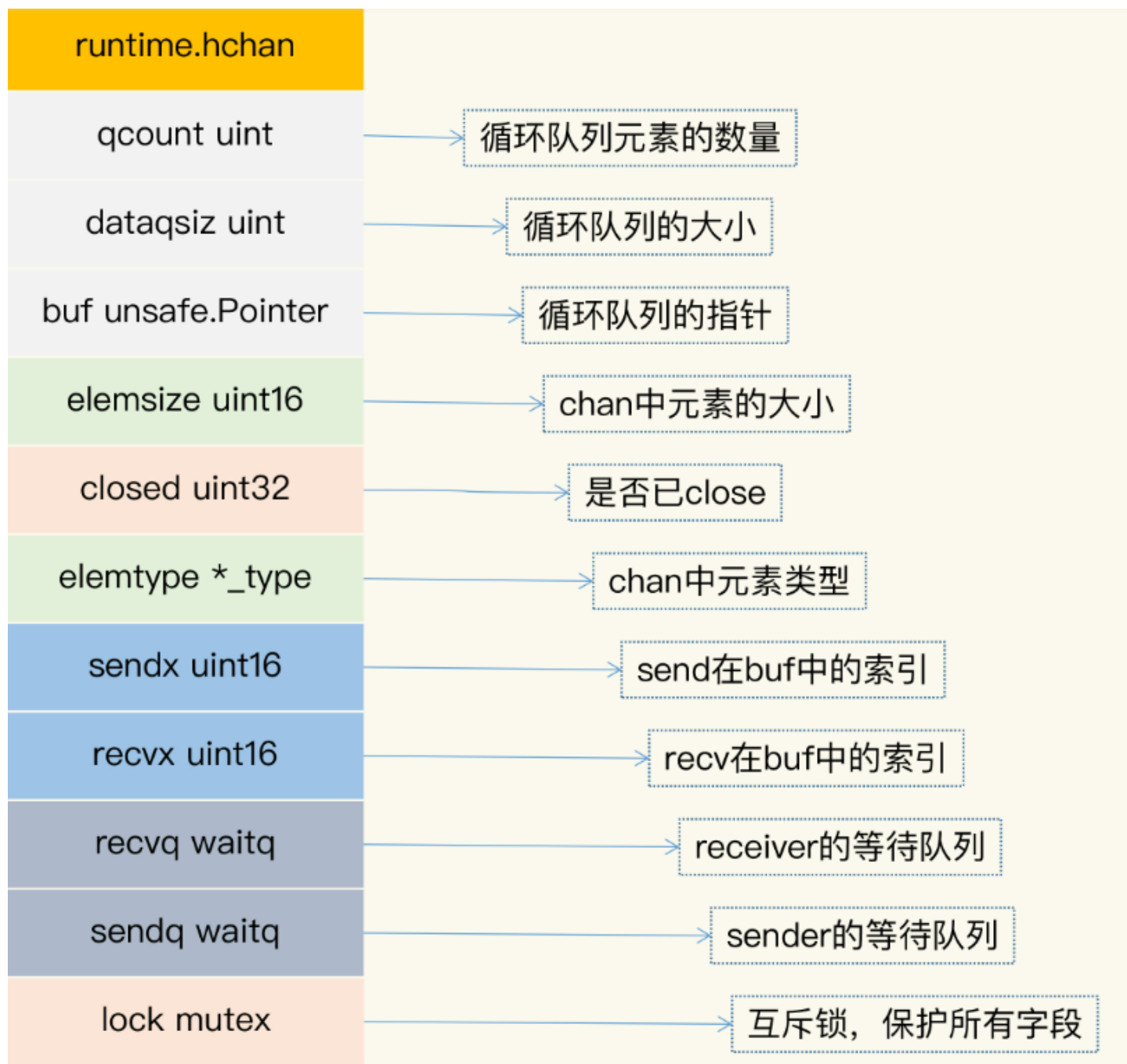
Value 类型：见演示文稿

Channel

应用场景：

1. 数据交流
2. 任务编排
3. 数据传递
4. 信号通知
5. 锁

实现机制：



使用方法:

- 1、共享资源的并发访问使用传统并发原语
- 2、复杂的任务编排和消息传递使用 Channel;
- 3、消息通知机制使用 Channel，除非只想 signal 一个 goroutine，才使用 Cond
- 4、简单等待所有任务的完成用 WaitGroup，也有 Channel 的推崇者用 Channel，都可以
- 5、需要和 Select 语句结合，使用 Channel
- 6、需要和超时配合时，使用 Channel 和 Context

使用反射操作 Channel

通过 reflect.Select 函数，你可以将一组运行时的 case clause 传入，当作参数执行。Go 的 select 是伪随机的，它可以在执行的 case 中随机选择一个 case，并把选择的这个 case 的索引 (chosen) 返

回，如果没有可用的 case 返回，会返回一个 bool 类型的返回值，这个返回值用来表示是否有 case 成功被选择。如果是 recv case，还会返回接收的元素
见demo

Go 内存模型

happens-before在一个 goroutine 内部，程序的执行顺序和它们的代码指定的顺序是一样的，即使编译器或者 CPU 重排了读写顺序，从行为上来看，也和代码指定的顺序一样

1. 在 Go 语言中，对变量进行零值的初始化就是一个写操作。
2. 如果对超过机器 word（64bit、32bit 或者其它）大小的值进行读写，那么，就可以看作是对拆成 word 大小的几个读写无序进行。
3. Go 并不提供直接的 CPU 屏障（CPU fence）来提示编译器或者 CPU 保证顺序性，而是使用不同架构的内存屏障指令来实现统一的并发原语

Semaphore

信号量（Semaphore）是用来控制多个 goroutine 同时访问多个资源的并发原语

Go 在它的扩展包中提供了信号量`semaphore`，不过这个信号量的类型名并不叫 Semaphore，而是叫 Weighted

1. Acquire 方法：相当于 P 操作，你可以一次获取多个资源，如果没有足够多的资源，调用者就会被阻塞。它的第一个参数是 Context，这就意味着，你可以通过 Context 增加超时或者 cancel 的机制。如果是正常获取了资源，就返回 nil；否则，就返回 ctx.Err()，信号量不改变。
2. Release 方法：相当于 V 操作，可以将 n 个资源释放，返还给信号量。
3. TryAcquire 方法：尝试获取 n 个资源，但是它不会阻塞，要么成功获取 n 个资源，返回 true，要么一个也不获取，返回 false

信号量 Weighted 的数据结构：

```
1 type Weighted struct {
2     size int64 // 最大资源数
3     cur  int64 // 当前已被使用的资源
4     mu   sync.Mutex // 互斥锁，对字段的保护
5     waiters list.List // 等待队列
6 }
```

其它信号量的实现

- 1、使用一个buffer 为 n 的 Channel 很容易实现信号量
- 2、marusama/semaphore也实现了一个可以动态更改资源容量的信号量

SingleFlight

SingleFlight 的作用是将并发请求合并成一个请求，以减少对下层服务的压力

sync.Once 主要是用在单次初始化场景中，而 SingleFlight 主要用在合并并发请求的场景中，尤其是缓存场景

在面对秒杀等大并发请求的场景，而且这些请求都是读请求时，你就可以把这些请求合并为一个请求，这样，你就可以将后端服务的压力从 n 降到 1。尤其是在面对后端是数据库这样的服务的时候，采用 SingleFlight 可以极大地提高性能

SingleFlight 使用互斥锁 Mutex 和 Map 来实现。Mutex 提供并发时的读写保护，Map 用来保存同一个 key 的正在处理 (in flight) 的请求

Do: 这个方法执行一个函数，并返回函数执行的结果。你需要提供一个 key，对于同一个 key，在同一时间只有一个在执行，同一个 key 并发的请求会等待。第一个执行的请求返回的结果，就是它的返回结果。函数 fn 是一个无参的函数，返回一个结果或者 error，而 Do 方法会返回函数执行的结果或者是 error，shared 会指示 v 是否返回给多个请求

DoChan: 类似 Do 方法，只不过是返回一个 chan，等 fn 函数执行完，产生了结果以后，就能从这个 chan 中接收这个结果。

Forget: 告诉 Group 忘记这个 key。这样一来，之后这个 key 请求会执行 f，而不是等待前一个未完成的 fn 函数的结果

用 SingleFlight 来解决缓存击穿问题再合适不过了

CyclicBarrier

CyclicBarrier 是一个可重用的栅栏并发原语，用来控制一组请求同时执行的数据结构

CyclicBarrier 更适合用在“固定数量的 goroutine 等待同一个执行点”的场景中，而且在放行 goroutine 之后，CyclicBarrier 可以重复利用，不像 WaitGroup 重用的时候，必须小心翼翼地避免 panic

WaitGroup 更适合用在“一个 goroutine 等待一组 goroutine 到达同一个执行点”的场景中，或者是不需要重用的场景中

- 1、第一个是 New 方法，它只需要一个参数，来指定循环栅栏参与者的数量；
- 2、第二个方法是 NewWithAction，它额外提供一个函数，可以在每一次到达执行点的时候执行一次。具体的时间点是在最后一个参与者到达之后，但是其它的参与者还未被放行之前。我们可以利用它，做放行之前的一些共享状态的更新等操作

分组操作

ErrGroup

在使用 ErrGroup 时，我们要用到三个方法，分别是 WithContext、Go 和 Wait

分布式环境的并发原语（etcd）

