Mutex

01 | Mutex：如何解决资源并发访问问题

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的意想不到的结果，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：

```
func main() {
    // 封装好的计数器
    var counter Countervar
    wg sync.WaitGroupwg.Add(10)
    // 启动10个goroutine
    for i := 0; i < 10; i++ {
        go func() {
            defer wg.Done()
            // 执行10万次累加
            for j := 0; j < 100000; j++ {
                counter.Incr() // 受到锁保护的方法
            }
        }()
    }
    wg.Wait()
    fmt.Println(counter.Count())
}
// 线程安全的计数器类型
type Counter struct {
    CounterType intName
    stringmu sync.Mutex
    count uint64
}
// 加1的方法，内部使用互斥锁保护
func (c *Counter) Incr() {
    c.mu.Lock()
    c.count++
    c.mu.Unlock()
}
```

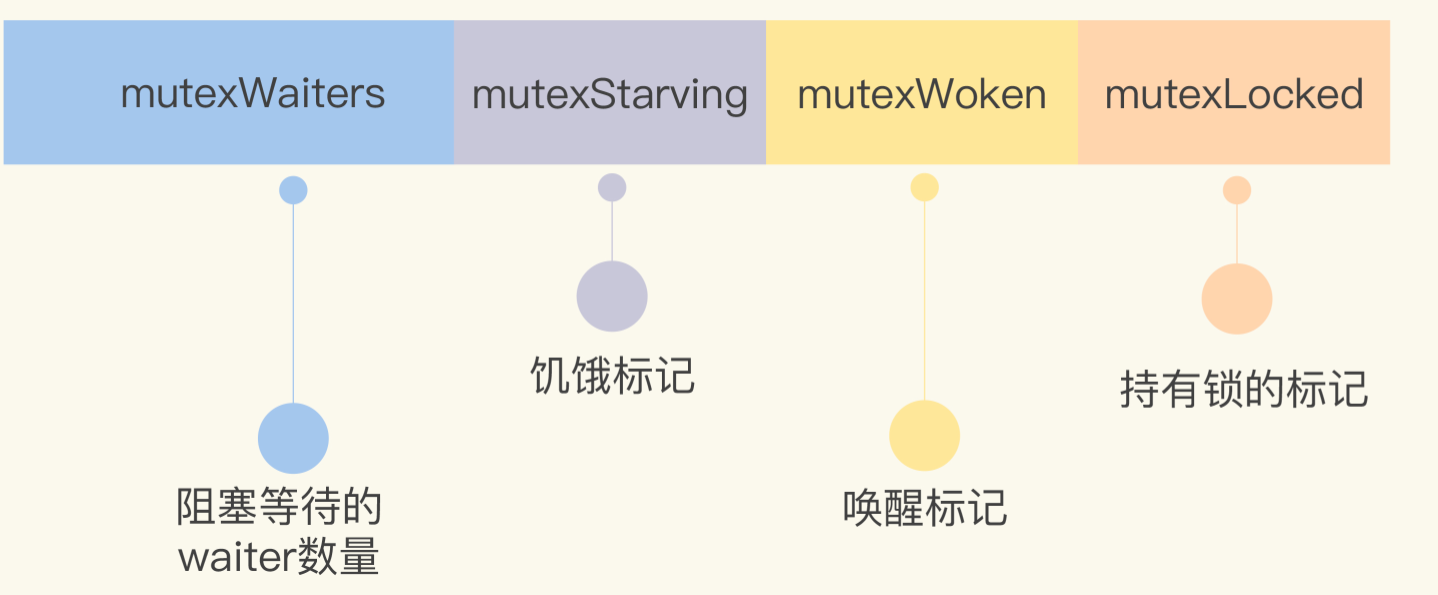
```
30 // 得到计数器的值，也需要锁保护
31 func (c *Counter) Count() uint64 {
32     c.mu.Lock()
33     defer c.mu.Unlock()
34     return c.count
35 }
36
```

02 | Mutex：庖丁解牛看实现

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：四部分：

```
1 type Mutex struct {
2     state int32
3     sema uint32
4 }
```



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。

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，…，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

```
type RWMutex struct {
    w Mutex // 互斥锁解决多个writer的竞争
    writerSem uint32 // writer信号量
    readerSem uint32 // reader信号量
    readerCount int32 // reader的数量
    readerWait int32 // writer等待完成的reader的数量
}
```

```
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goroute，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 。

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）

reader = 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

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

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；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

```
type WaitGroup struct {
// 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
noCopy noCopy
state1 [3]uint32
}
```

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。

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。

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

看demo.go实例

实现方法：

```
type Cond struct {
    noCopy noCopy
    L Locker  // 当观察或者修改等待条件的时候需要加锁
```

```
    notify notifyList  // 等待队列
    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。
}
```

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

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

```
type Once struct {
    done uint32m Mutex
}
```

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

```go
type Map struct {
    mu Mutex// 基本上你可以把它看成一个安全的只读的map
    // 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
    read atomic.Value
    // readOnly// 包含需要加锁才能访问的元素
    // 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
    dirty map[interface{}]*entry
    // 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
    misses int
}
```

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goroutetine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

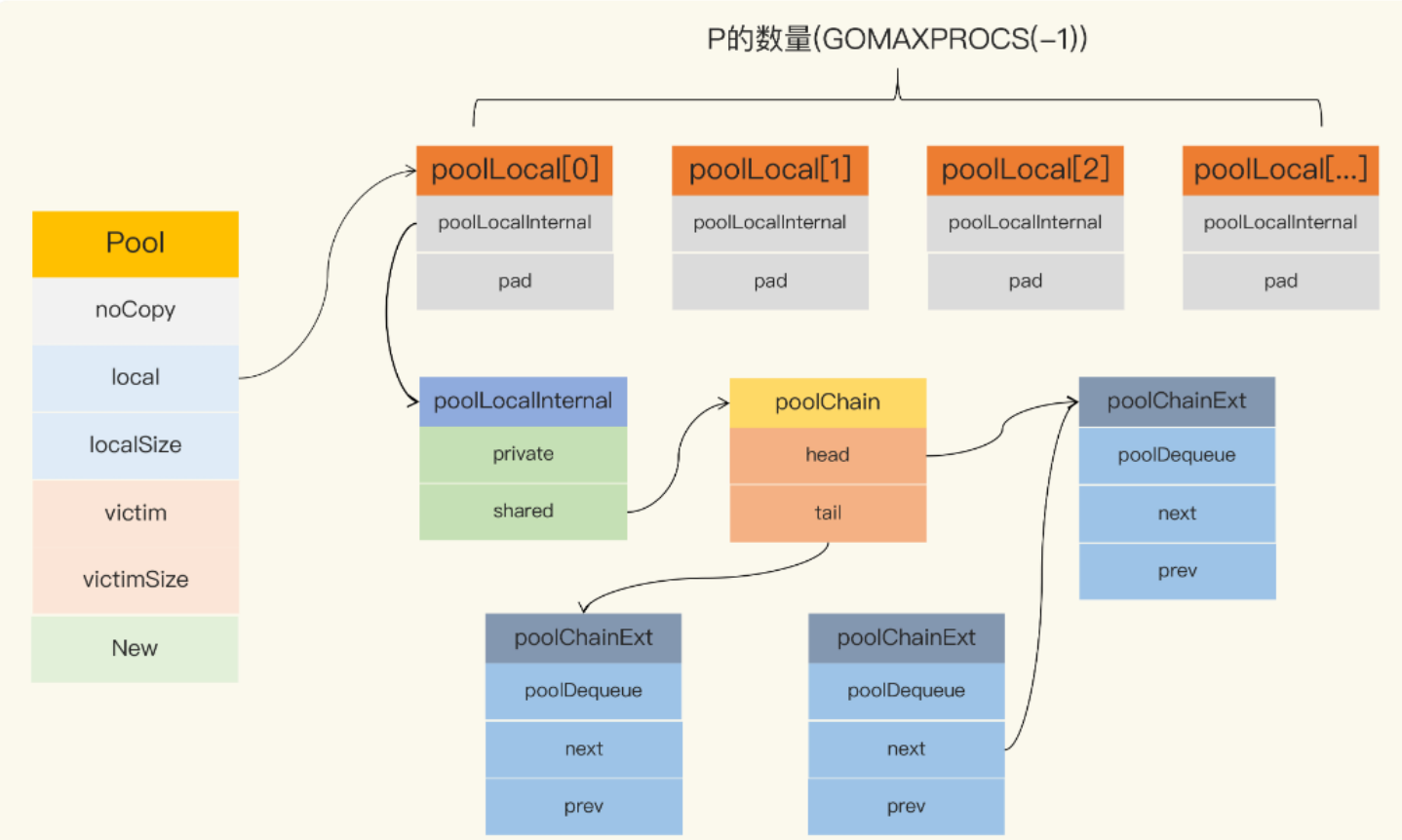
sync.pool的使用方法：

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 vintim 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

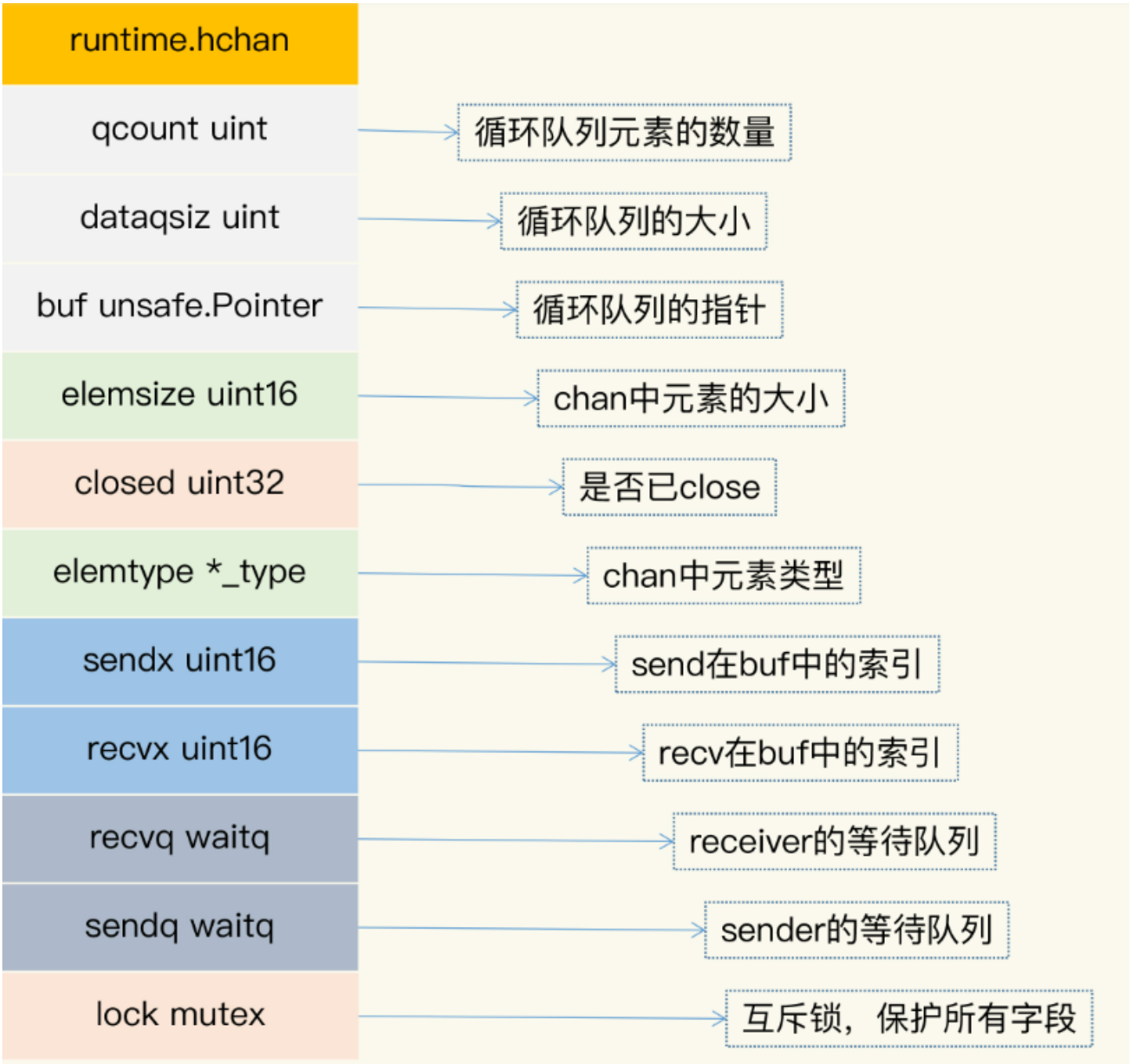
atomic

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：

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

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）