



下载APP



10 | Pool: 性能提升大杀器

2020-11-02 晁岳攀

Go 并发编程实战课

[进入课程 >](#)**讲述: 安晓辉**

时长 28:54 大小 26.47M



你好，我是鸟窝。

Go 是一个自动垃圾回收的编程语言，采用 [三色并发标记算法](#) 标记对象并回收。和其它没有自动垃圾回收的编程语言不同，使用 Go 语言创建对象的时候，我们没有回收 / 释放的心理负担，想用就用，想创建就创建。

但是，如果你想使用 Go 开发一个高性能的应用程序的话，就必须考虑垃圾回收给性能带来的影响，毕竟，Go 的自动垃圾回收机制还是有一个 STW (stop-the-world, 程序暂停) 的时间，而且，大量地创建在堆上的对象，也会影响垃圾回收标记的时间。



所以，一般我们做性能优化的时候，会采用对象池的方式，把不用的对象回收起来，避免被垃圾回收掉，这样使用的时候就不必在堆上重新创建了。

不止如此，像数据库连接、TCP 的长连接，这些连接在创建的时候是一个非常耗时的操作。如果每次都创建一个新的连接对象，耗时较长，很可能整个业务的大部分耗时都花在了创建连接上。

所以，如果我们能把这些连接保存下来，避免每次使用的时候都重新创建，不仅可以大大减少业务的耗时，还能提高应用程序的整体性能。

Go 标准库中提供了一个通用的 Pool 数据结构，也就是 `sync.Pool`，我们使用它可以创建池化的对象。这节课我会详细给你介绍一下 `sync.Pool` 的使用方法、实现原理以及常见的坑，帮助你全方位地掌握标准库的 Pool。

不过，这个类型也有一些使用起来不太方便的地方，就是**它池化的对象可能会被垃圾回收掉**，这对于数据库长连接等场景是不合适的。所以在这一讲中，我会专门介绍其它的一些 Pool，包括 TCP 连接池、数据库连接池等等。

除此之外，我还会专门介绍一个池的应用场景：Worker Pool，或者叫做 goroutine pool，这也是常用的一种并发模式，可以使用有限的 goroutine 资源去处理大量的业务数据。

sync.Pool

首先，我们来学习下标准库提供的 `sync.Pool` 数据类型。

`sync.Pool` 数据类型用来保存一组可独立访问的**临时**对象。请注意这里加粗的“临时”这两个字，它说明了 `sync.Pool` 这个数据类型的特点，也就是说，它池化的对象会在未来的某个时候被毫无预兆地移除掉。而且，如果没有别的对象引用这个被移除的对象的话，这个被移除的对象就会被垃圾回收掉。

因为 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。

有两个知识点你需要记住：

1. `sync.Pool` 本身就是线程安全的，多个 goroutine 可以并发地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

知道了 sync.Pool 这个数据类型的特点，接下来，我们来学习下它的使用方法。其实，这个数据类型不难，它只提供了三个对外的方法：New、Get 和 Put。

1.New

Pool struct 包含一个 New 字段，这个字段的类型是函数 func() interface{}。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。

有趣的是，New 是可变的字段。这就意味着，你可以在程序运行的时候改变创建元素的方法。当然，很少有人会这么做，因为一般我们创建元素的逻辑都是一致的，要创建的也是同一类的元素，所以你在使用 Pool 的时候也没必要玩一些“花活”，在程序运行时更改 New 的值。

2.Get

如果调用这个方法，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。不过，除了返回值是正常实例化的元素，Get 方法的返回值还可能会是一个 nil（Pool.New 字段没有设置，又没有空闲元素可以返回），所以你在使用的时候，可能需要判断。

3.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。

好了，了解了这几个方法，下面我们看看 sync.Pool 最常用的一个场景：buffer 池（缓冲池）。

因为 byte slice 是经常被创建销毁的一类对象，使用 buffer 池可以缓存已经创建的 byte slice，比如，著名的静态网站生成工具 Hugo 中，就包含这样的实现 [bufpool](#)，你可以看一下下面这段代码：

[复制代码](#)

```
1 var buffers = sync.Pool{
2     New: func() interface{} {
3         return new(bytes.Buffer)
4     },
5 }
6
7 func GetBuffer() *bytes.Buffer {
8     return buffers.Get().(*bytes.Buffer)
9 }
10
11 func PutBuffer(buf *bytes.Buffer) {
12     buf.Reset()
13     buffers.Put(buf)
14 }
```

除了 Hugo，这段 buffer 池的代码非常常用。很可能你在阅读其它项目的代码的时候就碰到过，或者是你自己实现 buffer 池的时候也会这么去实现，但是请你注意了，这段代码是有问题的，你一定不要将上面的代码应用到实际的产品中。它可能会有内存泄漏的问题，下面我会重点讲这个问题。

实现原理

了解了 sync.Pool 的基本使用方法，下面我们就来重点学习下它的实现。

Go 1.13 之前的 sync.Pool 的实现有 2 大问题：

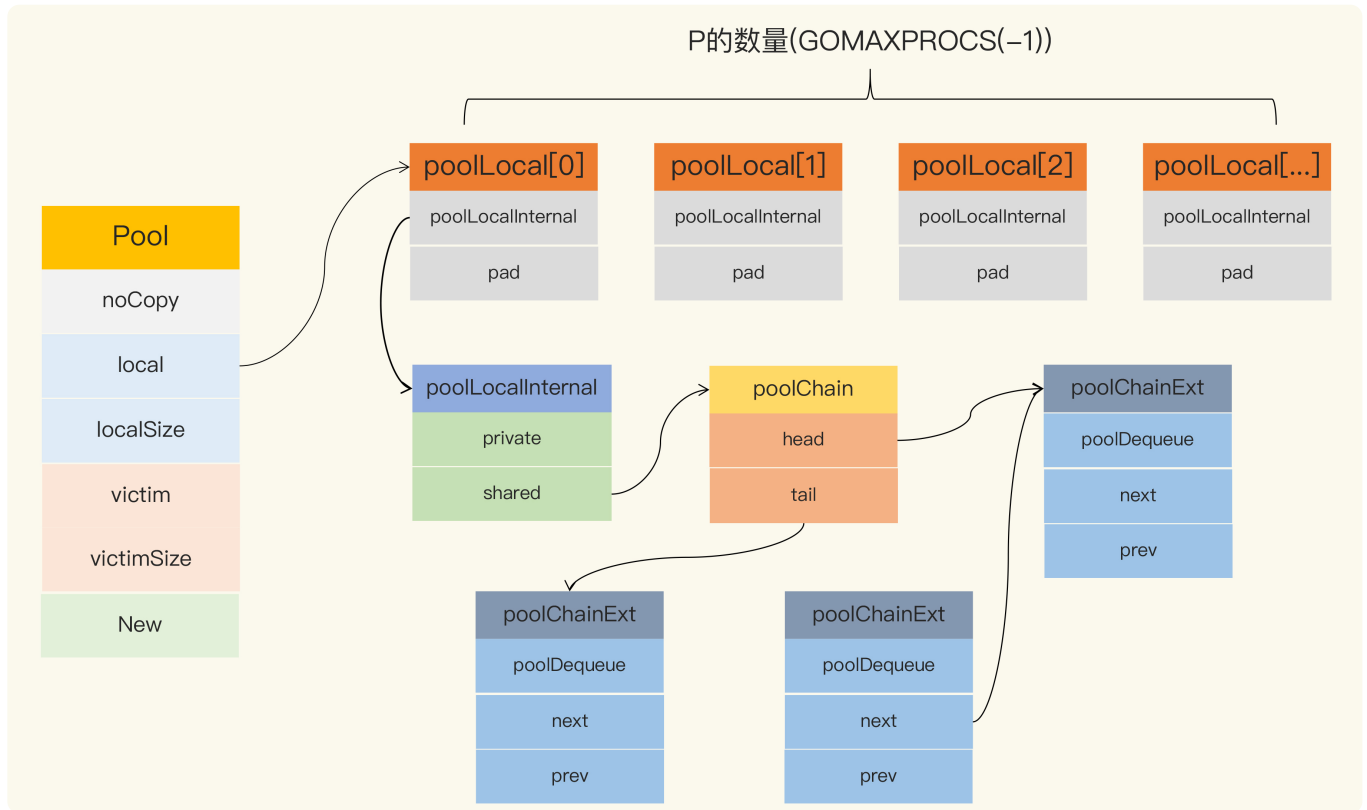
1. 每次 GC 都会回收创建的对象。

如果缓存元素数量太多，就会导致 STW 耗时变长；缓存元素都被回收后，会导致 Get 命中率下降，Get 方法不得不新建很多对象。

2. 底层实现使用了 Mutex，对这个锁并发请求竞争激烈的时候，会导致性能的下降。

在 Go 1.13 中, `sync.Pool` 做了大量的优化。前几讲中我提到过, 提高并发程序性能优化点是尽量不要使用锁, 如果不得已使用了锁, 就把锁 Go 的粒度降到最低。**Go 对 Pool 的优化就是避免使用锁, 同时将加锁的 queue 改成 lock-free 的 queue 的实现, 给即将移除的元素再多一次“复活”的机会。**

当前, `sync.Pool` 的数据结构如下图所示:




`Pool` 最重要的两个字段是 `local` 和 `victim`, 因为它们两个主要用来存储空闲的元素。弄清楚这两个字段的处理逻辑, 你就能完全掌握 `sync.Pool` 的实现了。下面我们来看看这两个字段的关系。

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。

`victim` 中的元素如果被 `Get` 取走, 那么这个元素就很幸运, 因为它又“活”过来了。但是, 如果这个时候 `Get` 的并发不是很大, 元素没有被 `Get` 取走, 那么就会被移除掉, 因为没有别人引用它的话, 就会被垃圾回收掉。

下面的代码是垃圾回收时 `sync.Pool` 的处理逻辑:

 复制代码

```

1 func poolCleanup() {
2     // 丢弃当前victim, STW所以不用加锁
3     for _, p := range oldPools {
4         p.victim = nil
5         p.victimSize = 0
6     }
7
8     // 将local复制给victim, 并将原local置为nil
9     for _, p := range allPools {
10        p.victim = p.local
11        p.victimSize = p.localSize
12        p.local = nil
13        p.localSize = 0
14    }
15
16    oldPools, allPools = allPools, nil
17 }

```

在这段代码中，你需要关注一下 local 字段，因为所有当前主要的空闲可用的元素都存放在 local 字段中，请求元素时也是优先从 local 字段中查找可用的元素。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題。

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 方法

我们来看看 Get 方法的具体实现原理。

 复制代码

```

1 func (p *Pool) Get() interface{} {
2     // 把当前goroutine固定在当前的P上
3     l, pid := p.pin()
4     x := l.private // 优先从local的private字段取，快速
5     l.private = nil

```

```

6     if x == nil {
7         // 从当前的local.shared弹出一个，注意是从head读取并移除
8         x, _ = l.shared.popHead()
9         if x == nil { // 如果没有，则去偷一个
10             x = p.getSlow(pid)
11         }
12     }
13     runtime_procUnpin()
14     // 如果没有获取到，尝试使用New函数生成一个新的
15     if x == nil && p.New != nil {
16         x = p.New()
17     }
18     return x
19 }

```

我来给你解释下这段代码。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。

下面的代码是 getSlow 方法的主要逻辑：

```

1 func (p *Pool) getSlow(pid int) interface{} {
2
3     size := atomic.LoadUintptr(&p.localSize)
4     locals := p.local
5     // 从其它proc中尝试偷取一个元素
6     for i := 0; i < int(size); i++ {
7         l := indexLocal(locals, (pid+i+1)%int(size))
8         if x, _ := l.shared.popTail(); x != nil {
9             return x
10        }
11    }
12
13    // 如果其它proc也没有可用元素，那么尝试从vintim中获取

```

 复制代码

```

14     size = atomic.LoadUintptr(&p.victimSize)
15     if uintptr(pid) >= size {
16         return nil
17     }
18     locals = p.victim
19     l := indexLocal(locals, pid)
20     if x := l.private; x != nil { // 同样的逻辑, 先从victim中的local private获取
21         l.private = nil
22         return x
23     }
24     for i := 0; i < int(size); i++ { // 从victim其它proc尝试偷取
25         l := indexLocal(locals, (pid+i)%int(size))
26         if x, _ := l.shared.popTail(); x != nil {
27             return x
28         }
29     }
30
31     // 如果victim中都没有, 则把这个victim标记为空, 以后的查找可以快速跳过了
32     atomic.StoreUintptr(&p.victimSize, 0)
33
34     return nil
35 }

```

这里我没列出 pin 代码的实现, 你只需要知道, pin 方法会将此 goroutine 固定在当前的 P 上, 避免查找元素期间被其它的 P 执行。固定的好处就是查找元素期间直接得到跟这个 P 相关的 local。有一点需要注意的是, pin 方法在执行的时候, 如果跟这个 P 相关的 local 还没有创建, 或者运行时 P 的数量被修改了的话, 就会新创建 local。

Put 方法

我们来看看 Put 方法的具体实现原理。

 复制代码

```

1 func (p *Pool) Put(x interface{}) {
2     if x == nil { // nil值直接丢弃
3         return
4     }
5     l, _ := p.pin()
6     if l.private == nil { // 如果本地private没有值, 直接设置这个值即可
7         l.private = x
8         x = nil
9     }
10    if x != nil { // 否则加入到本地队列中
11        l.shared.pushHead(x)
12    }
13    runtime_procUnpin()

```



```
14 }
```

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。

sync.Pool 的坑

到这里，我们就掌握了 sync.Pool 的使用方法和实现原理，接下来，我要再和你聊聊容易踩的两个坑，分别是内存泄漏和内存浪费。

内存泄漏

这节课刚开始的时候，我讲到，可以使用 sync.Pool 做 buffer 池，但是，如果用刚刚的那种方式做 buffer 池的话，可能会有内存泄漏的风险。为啥这么说呢？我们来分析一下。

取出来的 bytes.Buffer 在使用的时候，我们可以往这个元素中增加大量的 byte 数据，这会导致底层的 byte slice 的容量可能会变得很大。这个时候，即使 Reset 再放回到池中，这些 byte slice 的容量不会改变，所占的空间依然很大。而且，因为 Pool 回收的机制，这些大的 Buffer 可能不被回收，而是会一直占用很大的空间，这属于内存泄漏的问题。

即使是 Go 的标准库，在内存泄漏这个问题上也栽了几次坑，比如 [issue 23199](#)、[@dsnet](#) 提供了一个简单的可重现的例子，演示了内存泄漏的问题。再比如 encoding、json 中类似的问题：将容量已经变得很大的 Buffer 再放回 Pool 中，导致内存泄漏。后来在元素放回时，增加了检查逻辑，改成放回的超过一定大小的 buffer，就直接丢弃掉，不再放到池中，如下所示：

```
288 func putEncodeState(e *encodeState) {
289     » // Proper usage of a sync.Pool requires each entry to have approximately
290     » // the same memory cost. To obtain this property when the stored type
291     » // contains a variably-sized buffer, we add a hard limit on the maximum buffer
292     » // to place back in the pool.
293     » //
294     » // See https://golang.org/issue/23199
295     » const maxSize = 1 << 16 // 64KiB
296     » if e.Cap() > maxSize {
297     »     » return
298     » }
299     » encodeStatePool.Put(e)
300 }
301
```

package fmt 中也有这个问题，修改方法是一样的，超过一定大小的 buffer，就直接丢弃了：

```
140 // Tree saves used pp structs in ppFree; avoids an allocation per invocation.
141 func (p *pp) free() {
142     »    // Proper usage of a sync.Pool requires each entry to have approximately
143     »    // the same memory cost. To obtain this property when the stored type
144     »    // contains a variably-sized buffer, we add a hard limit on the maximum buffer
145     »    // to place back in the pool.
146     »    //
147     »    // See https://golang.org/issue/23199
148     »    if cap(p.buf) > 64<<10 {
149     »        »    return
150     »    }
151
152     »    p.buf = p.buf[:0]
153     »    p.arg = nil
154     »    p.value = reflect.Value{}
155     »    ppFree.Put(p)
156 }
```

在使用 sync.Pool 回收 buffer 的时候，**一定要检查回收的对象的大小**。如果 buffer 太大，就不要回收了，否则就太浪费了。

内存浪费

除了内存泄漏以外，还有一种浪费的情况，就是池子中的 buffer 都比较大，但在实际使用的时候，很多时候只需要一个小的 buffer，这也是一种浪费现象。接下来，我就讲解一下这种情况的处理方法。

要做到物尽其用，尽可能不浪费的话，我们可以将 buffer 池分成几层。首先，小于 512 byte 的元素的 buffer 占一个池子；其次，小于 1K byte 大小的元素占一个池子；再次，小于 4K byte 大小的元素占一个池子。这样分成几个池子以后，就可以根据需要，到所需大小的池子中获取 buffer 了。

在标准库 net/http/server.go 中的代码中，就提供了 2K 和 4K 两个 writer 的池子。你可以看看下面这段代码：

```

814  var (
815      bufioReaderPool  sync.Pool
816      bufioWriter2kPool sync.Pool
817      bufioWriter4kPool sync.Pool
818  )
819
820  var copyBufPool = sync.Pool{
821      New: func() interface{} {
822          b := make([]byte, 32*1024)
823          return &b
824      },
825  }
826
827  func bufioWriterPool(size int) *sync.Pool {
828      switch size {
829      case 2 << 10:
830          return &bufioWriter2kPool
831      case 4 << 10:
832          return &bufioWriter4kPool
833      }
834      return nil
835  }
836

```

YouTube 开源的知名项目 vitess 中提供了 [bucketpool](#) 的实现，它提供了更加通用的多层 buffer 池。你在使用的时候，只需要指定池子的最大和最小尺寸，vitess 就会自动计算出合适的池子数。而且，当你调用 Get 方法的时候，只需要传入你要获取的 buffer 的大小，就可以了。下面这段代码就描述了这个过程，你可以看看：

```

type Pool
    ◦ func New(minSize, maxSize int) *Pool
    ◦ func (p *Pool) Get(size int) *[]byte
    ◦ func (p *Pool) Put(b *[]byte)

```

第三方库

除了这种分层的为了节省空间的 buffer 设计外，还有其它的一些第三方的库也会提供 buffer 池的功能。接下来我带你熟悉几个常用的第三方的库。

1. [bytebufferpool](#)

这是 fasthttp 作者 valyala 提供的一个 buffer 池，基本功能和 sync.Pool 相同。它的底层也是使用 sync.Pool 实现的，包括会检测最大的 buffer，超过最大尺寸的 buffer，就会被丢弃。

valyala 一向很擅长挖掘系统的性能，这个库也不例外。它提供了校准（calibrate，用来动态调整创建元素的权重）的机制，可以“智能”地调整 Pool 的 defaultSize 和 maxSize。一般来说，我们使用 buffer size 的场景比较固定，所用 buffer 的大小会集中在某个范围里。有了校准的特性，bytebufferpool 就能够偏重于创建这个范围大小的 buffer，从而节省空间。

2. [oxtoacart/bpool](#)

这也是比较常用的 buffer 池，它提供了以下几种类型的 buffer。

bpool.BufferPool: 提供一个固定元素数量的 buffer 池，元素类型是 bytes.Buffer，如果超过这个数量，Put 的时候就丢弃，如果池中的元素都被取光了，会新建一个返回。Put 回去的时候，不会检测 buffer 的大小。

bpool.BytesPool: 提供一个固定元素数量的 byte slice 池，元素类型是 byte slice。Put 回去的时候不检测 slice 的大小。

bpool.SizedBufferPool: 提供一个固定元素数量的 buffer 池，如果超过这个数量，Put 的时候就丢弃，如果池中的元素都被取光了，会新建一个返回。Put 回去的时候，会检测 buffer 的大小，超过指定的大小的话，就会创建一个新的满足条件的 buffer 放回去。

bpool 最大的特色就是能够保持池子中元素的数量，一旦 Put 的数量多于它的阈值，就会自动丢弃，而 sync.Pool 是一个没有限制的池子，只要 Put 就会收进去。

bpool 是基于 Channel 实现的，不像 sync.Pool 为了提高性能而做了很多优化，所以，在性能上比不过 sync.Pool。不过，它提供了限制 Pool 容量的功能，所以，如果你想控制

Pool 的容量的话，可以考虑这个库。

连接池

Pool 的另一个很常用的一个场景就是保持 TCP 的连接。一个 TCP 的连接创建，需要三次握手等过程，如果是 TLS 的，还会需要更多的步骤，如果加上身份认证等逻辑的话，耗时会更长。所以，为了避免每次通讯的时候都新创建连接，我们一般会建立一个连接的池子，预先把连接创建好，或者是逐步把连接放在池子中，减少连接创建的耗时，从而提高系统的性能。

事实上，我们很少会使用 sync.Pool 去池化连接对象，原因就在于，sync.Pool 会无通知地在某个时候就把连接移除垃圾回收掉了，而我们的场景是需要长久保持这个连接，所以，我们一般会使用其它方法来池化连接，比如接下来我要讲到的几种需要保持长连接的 Pool。

标准库中的 http client 池

标准库的 http.Client 是一个 http client 的库，可以用它来访问 web 服务器。为了提高性能，这个 Client 的实现也是通过池的方法来缓存一定数量的连接，以便后续重用这些连接。

http.Client 实现连接池的代码是在 Transport 类型中，它使用 idleConn 保存持久化的可重用的长连接：

```
95  type Transport struct {
96      idleMu      sync.Mutex
97      closeIdle   bool                // user has requested to close all idle conns
98      idleConn    map[connectMethodKey][]*persistConn // most recently used at end
99      idleConnWait map[connectMethodKey]wantConnQueue // waiting getConns
100     idleLRU      connLRU
101
102     reqMu      sync.Mutex
103     reqCanceler map[cancelKey]func(error)
104
105     altMu      sync.Mutex // guards changing altProto only
106     altProto   atomic.Value // of nil or map[string]RoundTripper, key is URI scheme
107
108     connsPerHostMu sync.Mutex
109     connsPerHost   map[connectMethodKey]int
110     connsPerHostWait map[connectMethodKey]wantConnQueue // waiting getConns
```

TCP 连接池

最常用的一个 TCP 连接池是 fatih 开发的 [fatih/pool](#)，虽然这个项目已经被 fatih 归档 (Archived)，不再维护了，但是因为它相当稳定了，我们可以开箱即用。即使你有一些特殊的需求，也可以 fork 它，然后自己再做修改。

它的使用套路如下：

[复制代码](#)

```
1 // 工厂模式，提供创建连接的工厂方法
2 factory := func() (net.Conn, error) { return net.Dial("tcp", "127.0.0.1:4000") }
3
4 // 创建一个tcp池，提供初始容量和最大容量以及工厂方法
5 p, err := pool.NewChannelPool(5, 30, factory)
6
7 // 获取一个连接
8 conn, err := p.Get()
9
10 // Close并不会真正关闭这个连接，而是把它放回池子，所以你不必显式地Put这个对象到池中
11 conn.Close()
12
13 // 通过调用MarkUnusable，Close的时候就会真正关闭底层的tcp的连接了
14 if pc, ok := conn.(*pool.PoolConn); ok {
15     pc.MarkUnusable()
16     pc.Close()
17 }
18
19 // 关闭池子就会关闭=池子中的所有的tcp连接
20 p.Close()
21
22 // 当前池子中的连接的数量
23 current := p.Len()
```

虽然我一直在说 TCP，但是它管理的是更通用的 net.Conn，不局限于 TCP 连接。

它通过把 net.Conn 包装成 PoolConn，实现了拦截 net.Conn 的 Close 方法，避免了真正地关闭底层连接，而是把这个连接放回到池中：

[复制代码](#)

```
1 type PoolConn struct {
2     net.Conn
3     mu      sync.RWMutex
4     c       *channelPool
5     unusable bool
6 }
```



```

7
8    //拦截Close
9    func (p *PoolConn) Close() error {
10        p.mu.RLock()
11        defer p.mu.RUnlock()
12
13        if p.unusable {
14            if p.Conn != nil {
15                return p.Conn.Close()
16            }
17            return nil
18        }
19        return p.c.put(p.Conn)
20    }

```

它的 Pool 是通过 Channel 实现的，空闲的连接放入到 Channel 中，这也是 Channel 的一个应用场景：

```

1    type channelPool struct {
2        // 存储连接池的channel
3        mu      sync.RWMutex
4        conns   chan net.Conn
5
6
7        // net.Conn 的产生器
8        factory Factory
9    }

```

[复制代码](#)

数据库连接池

标准库 sql.DB 还提供了一个通用的数据库的连接池，通过 MaxOpenConns 和 MaxIdleConns 控制最大的连接数和最大的 idle 的连接数。默认的 MaxIdleConns 是 2，这个数对于数据库相关的应用来说太小了，我们一般都会调整它。

```

func (db *DB) SetConnMaxIdleTime(d time.Duration)
func (db *DB) SetConnMaxLifetime(d time.Duration)
func (db *DB) SetMaxIdleConns(n int)
func (db *DB) SetMaxOpenConns(n int)

```

DB 的 freeConn 保存了 idle 的连接，这样，当我们获取数据库连接的时候，它就会优先尝试从 freeConn 获取已有的连接（[🔗conn](#)）。

```

402 type DB struct {
403     // Atomic access only. At top of struct to prevent mis-alignment
404     // on 32-bit platforms. Of type time.Duration.
405     waitDuration int64 // Total time waited for new connections.
406
407     connector driver.Connector
408     // numClosed is an atomic counter which represents a total number of
409     // closed connections. Stmt.openStmt checks it before cleaning closed
410     // connections in Stmt.css.
411     numClosed uint64
412
413     mu sync.Mutex // protects following fields
414     freeConn []*driver.Conn


```

Memcached Client 连接池

Brad Fitzpatrick 是知名缓存库 Memcached 的原作者，前 Go 团队成员。

[gomemcache](#) 是他使用 Go 开发的 Memcached 的客户端，其中也用了连接池的方式池化 Memcached 的连接。接下来让我们看看它的连接池的实现。

gomemcache Client 有一个 freeconn 的字段，用来保存空闲的连接。当一个请求使用完之后，它会调用 putFreeConn 放回到池子中，请求的时候，调用 getFreeConn 优先查询 freeConn 中是否有可用的连接。它采用 Mutex+Slice 实现 Pool:

 复制代码

```

1 // 放回一个待重用的连接
2 func (c *Client) putFreeConn(addr net.Addr, cn *conn) {
3     c.lk.Lock()
4     defer c.lk.Unlock()
5     if c.freeconn == nil { // 如果对象为空, 创建一个map对象
6         c.freeconn = make(map[string][]*conn)
7     }
8     freelist := c.freeconn[addr.String()] //得到此地址的连接列表
9     if len(freelist) >= c.maxIdleConns() { //如果连接已满, 关闭, 不再放入
10         cn.nc.Close()
11         return
12     }
13     c.freeconn[addr.String()] = append(freelist, cn) // 加入到空闲列表中
14 }
15
16 // 得到一个空闲连接
17 func (c *Client) getFreeConn(addr net.Addr) (cn *conn, ok bool) {
18     c.lk.Lock()
19     defer c.lk.Unlock()
20

```

```
21     if c.freeconn == nil {
22         return nil, false
23     }
24     freelist, ok := c.freeconn[addr.String()]
25     if !ok || len(freelist) == 0 { // 没有此地址的空闲列表, 或者列表为空
26         return nil, false
27     }
28     cn = freelist[len(freelist)-1] // 取出尾部的空闲连接
29     c.freeconn[addr.String()] = freelist[:len(freelist)-1]
30     return cn, true
31 }
```

Worker Pool

最后, 我再讲一个 Pool 应用得非常广泛的场景。

你已经知道, goroutine 是一个很轻量级的“纤程”, 在一个服务器上可以创建十几万甚至几十万的 goroutine。但是“可以”和“合适”之间还是有区别的, 你会在应用中让几十万的 goroutine 一直跑吗? 基本上是不会的。

一个 goroutine 初始的栈大小是 2048 个字节, 并且在需要的时候可以扩展到 1GB (具体的内容你可以课下看看代码中的配置: [🔗不同的架构最大数会不同](#)), 所以, 大量的 goroutine 还是很耗资源的。同时, 大量的 goroutine 对于调度和垃圾回收的耗时还是会有影响的, 因此, goroutine 并不是越多越好。

有的时候, 我们就会创建一个 Worker Pool 来减少 goroutine 的使用。比如, 我们实现一个 TCP 服务器, 如果每一个连接都要由一个独立的 goroutine 去处理的话, 在大量连接的情况下, 就会创建大量的 goroutine, 这个时候, 我们就可以创建一个固定数量的 goroutine (Worker), 由这一组 Worker 去处理连接, 比如 fasthttp 中的 [🔗Worker Pool](#)。

Worker 的实现也是五花八门的:

有些是在后台默默执行的, 不需要等待返回结果;

有些需要等待一批任务执行完;

有些 Worker Pool 的生命周期和程序一样长;

有些只是临时使用，执行完毕后，Pool 就销毁了。

大部分的 Worker Pool 都是通过 Channel 来缓存任务的，因为 Channel 能够比较方便地实现并发的保护，有的是多个 Worker 共享同一个任务 Channel，有些是每个 Worker 都有一个独立的 Channel。

综合下来，精挑细选，我给你推荐三款易用的 Worker Pool，这三个 Worker Pool 的 API 设计简单，也比较相似，易于和项目集成，而且提供的功能也是我们常用的功能。

🔗 [gammazero/workerpool](#): gammazero/workerpool 可以无限制地提交任务，提供了更便利的 Submit 和 SubmitWait 方法提交任务，还可以提供当前的 worker 数和任务数以及关闭 Pool 的功能。

🔗 [ivpusic/grpool](#): grpool 创建 Pool 的时候需要提供 Worker 的数量和等待执行的任务的最大数量，任务的提交是直接往 Channel 放入任务。

🔗 [dpaks/goworkers](#): dpaks/goworkers 提供了更便利的 Submi 方法提交任务以及 Worker 数、任务数等查询方法、关闭 Pool 的方法。它的任务的执行结果需要在 ResultChan 和 ErrChan 中去获取，没有提供阻塞的方法，但是它可以在初始化的时候设置 Worker 的数量和任务数。

类似的 Worker Pool 的实现非常多，比如还有 🔗 [panjf2000/ants](#)、🔗 [Jeffail/tunny](#)、🔗 [benmanns/goworker](#)、🔗 [go-playground/pool](#)、🔗 [Sherifabdinaby/gpool](#) 等第三方库。🔗 [pond](#) 也是一个非常不错的 Worker Pool，关注度目前不是很高，但是功能非常齐全。

其实，你也可以自己去开发自己的 Worker Pool，但是，对于我这种“懒惰”的人来说，只要满足我的实际需求，我还是倾向于从这个几个常用的库中选择一个来使用。所以，我建议你也从常用的库中进行选择。

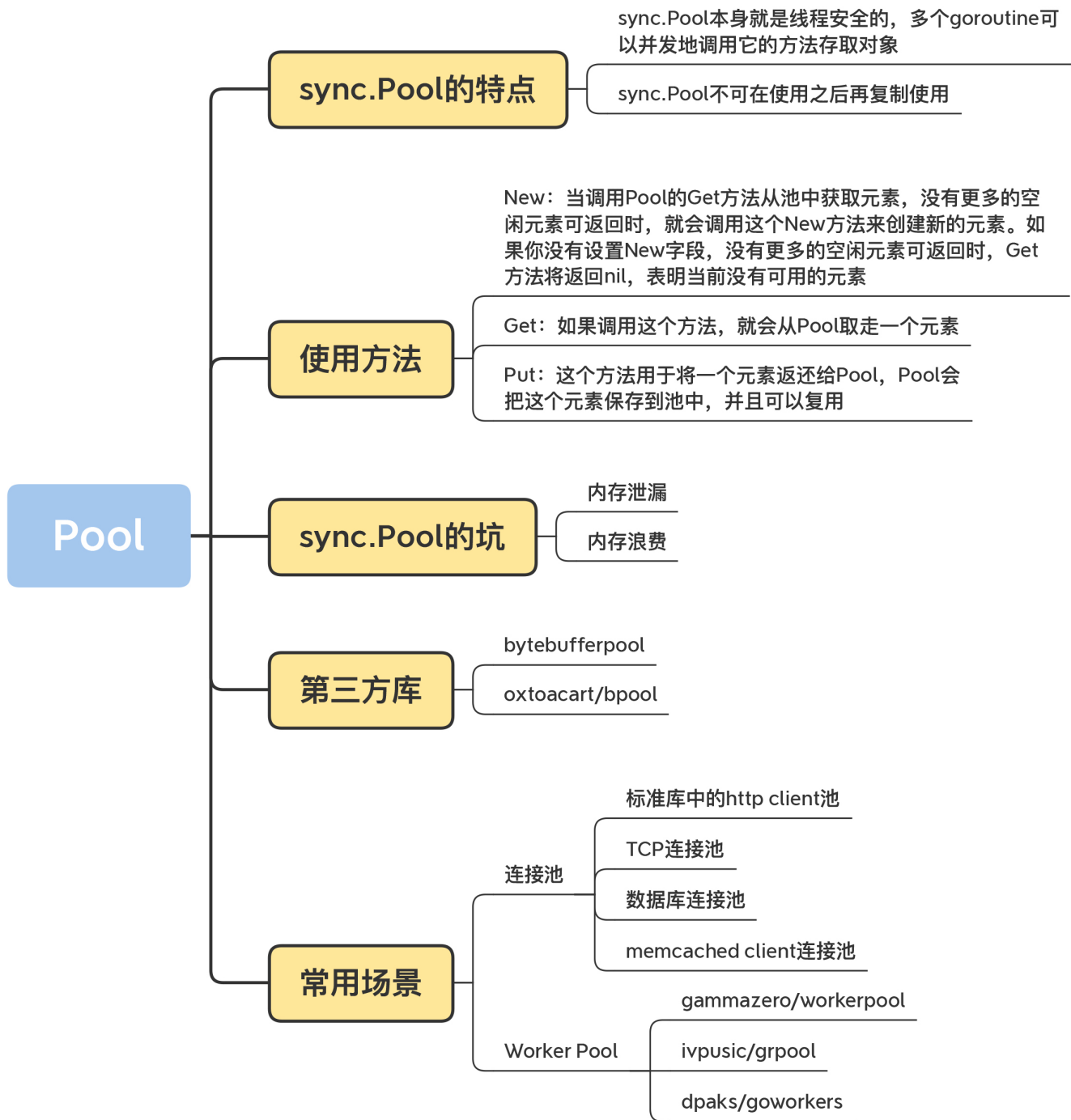
总结

Pool 是一个通用的概念，也是解决对象重用和预先分配的一个常用的优化手段。即使你自己没在项目中直接使用过，但肯定在使用其它库的时候，就享受到应用 Pool 的好处了，比如数据库的访问、http API 的请求等等。

我们一般不会在程序一开始的时候就开始考虑优化，而是等项目开发到一个阶段，或者快结束的时候，才全面地考虑程序中的优化点，而 Pool 就是常用的一个优化手段。如果你发现程序中有一种 GC 耗时特别高，有大量的相同类型的临时对象，不断地被创建销毁，这时，你就可以考虑看看，是不是可以通过池化的手段重用这些对象。

另外，在分布式系统或者微服务框架中，可能会有大量的并发 Client 请求，如果 Client 的耗时占比很大，你也可以考虑池化 Client，以便重用。

如果你发现系统中的 goroutine 数量非常多，程序的内存资源占用比较大，而且整体系统的耗时和 GC 也比较高，我建议你看看，是否能够通过 Worker Pool 解决大量 goroutine 的问题，从而降低这些指标。



思考题

在标准库 net/rpc 包中，Server 端需要解析大量客户端的请求（[Request](#)），这些短暂使用的 Request 是可以重用的。请你检查相关的代码，看看 Go 开发者都使用了什么样的方式来重用这些对象。

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 09 | map：如何实现线程安全的map类型？

下一篇 11 | Context：信息穿透上下文

精选留言 (8)

写留言



Junes

2020-11-02

分享一下我的理解，主要分为回收和获取两个函数：

```
func (server *Server) freeRequest(req *Request) {
    server.reqLock.Lock()
    // 将req放在freeReq的头部，指向原先的链表头...
    展开
```



4

**末班车**

2020-11-02

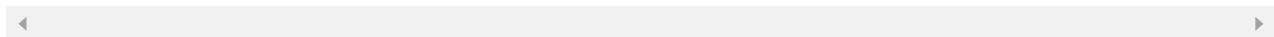
之前用到去看过，好像是通过一个链表的形式，把request存起来，最新的在链表的头，最旧的在链表的尾部，可是不懂的是，为什么每次取出了req，还要重新赋零值呢，这和我每次new一个有什么区别么？求大佬指点。

展开 ▾

作者回复: 好问题。

重新赋零值相当于reset,避免先前的垃圾数据影响这个request。

new会在堆上新创建一个对象。



💬 1

👍 4

**Yayu**

2020-11-03

谢谢老师，喜欢老师这篇文章中通过外链的方式列出一些老师常用的三方库，很有用！

💬

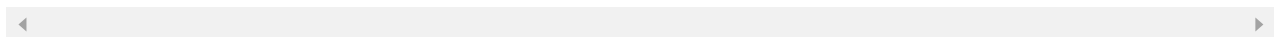
👍 1

**那一刻**

2020-11-03

请问老师, sync.Pool会有内存泄漏，怎么理解因为 Pool 回收的机制，这些大的 Buffer 可能不被回收？

作者回复: 对



💬

👍

**党**

2020-11-03

可以用池化技术做任务队列么?尤其是worker pool这几个库

💬

👍

**党**

2020-11-03

那像websocke这种长连接，每个ws用一个goroutine来维护，是不是就没必要用池化技术了。

💬 1

👍

**橙子888**



2020-11-02

打卡。

展开 ∨

**虫子樱桃**

2020-11-02

思考题的奥秘感觉在这两个函数

```\n

```
// ServeRequest is like ServeCodec but synchronously serves a single request.
```

```
// It does not close the codec upon completion.
```

```
func (server *Server) ServeRequest(codec ServerCodec) error {...
```

展开 ∨

