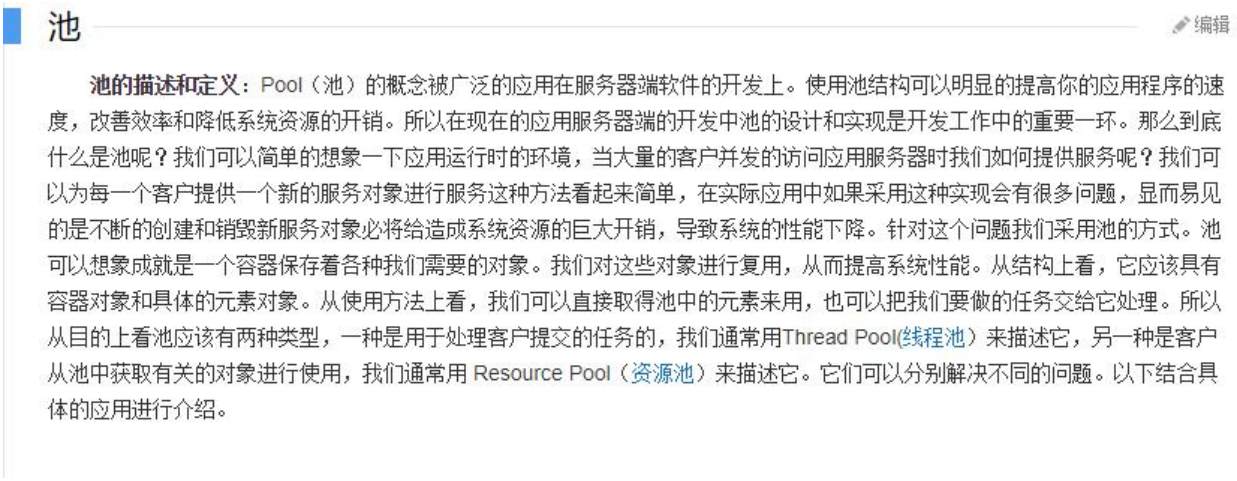
golang 之 sync.pool 思考与理解

作者：xhaoxiong
原文链接：https://ld246.com/article/1538325144163
来源网站：链滴

什么是池?



池

编辑

池的描述和定义：Pool（池）的概念被广泛的应用在服务器端软件的开发上。使用池结构可以明显的提高你的应用程序的速度，改善效率和降低系统资源的开销。所以在现在的应用服务器端的开发中池的设计和实现是开发工作中的重要一环。那么到底什么是池呢？我们可以简单的想象一下应用运行时的环境，当大量的客户并发的访问应用服务器时我们如何提供服务呢？我们可以为每一个客户提供一个新的服务对象进行服务这种方法看起来简单，在实际应用中如果采用这种实现会有很多问题，显而易见的是不断的创建和销毁新服务对象必将给造成系统资源的巨大开销，导致系统的性能下降。针对这个问题我们采用池的方式。池可以想象成就是一个容器保存着各种我们需要的对象。我们对这些对象进行复用，从而提高系统性能。从结构上看，它应该具有容器对象和具体的元素对象。从使用方法上看，我们可以直接取得池中的元素来用，也可以把我们要做的任务交给它处理。所以从目的上看池应该有两种类型，一种是用于处理客户提交的任务的，我们通常用Thread Pool(线程池）来描述它，另一种是客户从池中获取有关的对象进行使用，我们通常用 Resource Pool（资源池）来描述它。它们可以分别解决不同的问题。以下结合具体的应用进行介绍。

由于对池的概念很不清晰，引发了自己对golang中sync.pool的探究理解和学习

关于golang中的池sync.pool(基于1.10.3)

先看一个实例:

```
package main

import (
    "runtime/debug"
    "sync/atomic"
    "sync"
    "fmt"
    "runtime"
)

func main() {
    defer debug.SetGCPercent(debug.SetGCPercent(-1))

    var count int32
    newfun := func() interface{} {
        return atomic.AddInt32(&count, 1)
    }

    pool := sync.Pool{New: newfun}

    v1 := pool.Get()
    fmt.Printf("v1 :%v\n", v1)

    pool.Put(9)
    pool.Put(10)
    pool.Put(11)
    pool.Put(12)

    v2 := pool.Get()
```

```
    fmt.Printf("v2 :%v\n", v2)

    debug.SetGCPercent(100)
    runtime.GC()

    v3 := pool.Get()
    fmt.Printf("v3 :%v\n", v3)

    pool.New = nil

    v4 := pool.Get()
    fmt.Printf("v4 :%v\n", v4)
}
```

解决的问题

设计对象缓存池，除避免内存分配操作开销外，更多的是为了避免分配大量临时对象对垃圾回收器造负面影响。

与调度之间的一些联系

```
    m-p-g(m:n:n)

    m------p----- poolLocal
     *   * |
      * *  g - g
       *     |
      * *    g
     *   *   ...
    m------P----- poolLocal
          |
          |
         g---g
            |
            g
           ...
```

（关于调度和gc模块还没能深入综合理解及其联系，以上仅供参考）

pool的两个特点

1、在本地私有池和本地共享池均获取失败则会从其他p偷一个返回给调用方

2、对象在池中的生命周期取决于垃圾回收任务的下一次执行时间并且从池中获取到的值可能是put进的其中一个值也可能是newfun新生成的一个值，在应用时很容易入坑

包中具体的实现函数和结构体

Pool用local和localSize维护一个动态poolLocal数组。

```
type Pool struct {
    noCopy noCopy

    local     unsafe.Pointer //[P]poolLocal 数组指针
    localSize uintptr        // 数组大小


    New func() interface{} //新建对象函数
}

type poolLocalInternal struct {
    private interface{}   // 私有缓存区
    shared  []interface{} // 公共缓存区
    Mutex                 //
}

type poolLocal struct {
    poolLocalInternal

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

无论是Get()，还是Put()操作都会通过pin来返回与当前P绑定的poolLocal对象，这里面就有初始化关键

```
func (p *Pool) pin() *poolLocal {
    // 返回当前 P.id
    pid := runtime_procPin()


    s := atomic.LoadUintptr(&p.localSize) // load-acquire
    l := p.local          // load-consume
     // 如果 P.id 没有超出数组索引限制，则直接返回
     // 这是考虑到 procresize/GOMAXPROCS 的影响
    if uintptr(pid) < s {
        return indexLocal(l, pid)
    }
    // 没有结果时，会涉及全局加锁操作
    // 比如重新分配数组内存，添加到全局列表
    return p.pinSlow()
}


func (p *Pool) pinSlow() *poolLocal {
```

```
    // M.lock--
    runtime_procUnpin()
    // 加锁
    allPoolsMu.Lock()
    defer allPoolsMu.Unlock()
    pid := runtime_procPin()
    // 再次检查是否符合条件，可能中途已被其他线程调用
    s := p.localSize
    l := p.local
    if uintptr(pid) < s {
        return indexLocal(l, pid)
    }
    // 如果数组为空，新建
    // 将其添加到 allPools，垃圾回收器以此获取所有 Pool 实例
    if p.local == nil {
        allPools = append(allPools, p)
    }
    // 根据 P 数量创建 slice
    size := runtime.GOMAXPROCS(0)
    local := make([]poolLocal, size)
     // 将底层数组起始指针保存到 Pool.local，并设置 P.localSize
    atomic.StorePointer(&p.local, unsafe.Pointer(&local[0]))
    atomic.StoreUintptr(&p.localSize, uintptr(size))
    // 返回本次所需的 poolLocal
    return &local[pid]
}

func indexLocal(l unsafe.Pointer, i int) *poolLocal {
    lp := unsafe.Pointer(uintptr(l) + uintptr(i)*unsafe.Sizeof(poolLocal{}))
    return (*poolLocal)(lp)
}
```

1、Get()获取对象时 优先从private空间获取 ->没有则加锁从share空间获取(从尾部开始获取)->没再new func新的对象(此对象不会放回池中)

2、注意：Get操作后(在返回之前就会将它从池中删除)，缓存对象彻底与Pool失去引用关联，需要自Put放回。

```
func (p *Pool) Get() interface{} {
    if race.Enabled {
        race.Disable()
    }
    l := p.pin()
    x := l.private
    l.private = nil
    runtime_procUnpin()
    if x == nil {
        l.Lock()
        last := len(l.shared) - 1
        if last >= 0 {
```

```
            x = l.shared[last]
            l.shared = l.shared[:last]
        }
        l.Unlock()
        if x == nil {
            x = p.getSlow()
        }
    }
    if race.Enabled {
        race.Enable()
        if x != nil {
            race.Acquire(poolRaceAddr(x))
        }
    }
    if x == nil && p.New != nil {
        x = p.New()
    }
    return x
}

func (p *Pool) getSlow() (x interface{}) {

    size := atomic.LoadUintptr(&p.localSize) // load-acquire
    local := p.local                         // load-consume
    // 尝试从其他procs获取一个对象
    pid := runtime_procPin()
    runtime_procUnpin()
    for i := 0; i < int(size); i++ {
        l := indexLocal(local, (pid+i+1)%int(size))
        l.Lock()
        last := len(l.shared) - 1
        if last >= 0 {
            x = l.shared[last]
            l.shared = l.shared[:last]
            l.Unlock()
            break
        }
        l.Unlock()
    }
    return x
}
```

Put()优先放入private空间->其次再考虑share空间

```
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
    }
    runtime_procUnpin()
    if x != nil {
        l.Lock()
        l.shared = append(l.shared, x)
        l.Unlock()
    }
    if race.Enabled {
        race.Enable()
    }
}
```

由poolCleanup()可知其操作很简单粗暴清空，并且其需要额外注册runtime_registerPoolCleanup()的

```
func poolCleanup() {
    // This function is called with the world stopped, at the beginning of a garbage collection.
    // It must not allocate and probably should not call any runtime functions.
    // Defensively zero out everything, 2 reasons:
    // 1. To prevent false retention of whole Pools.
    // 2. If GC happens while a goroutine works with l.shared in Put/Get,
    //    it will retain whole Pool. So next cycle memory consumption would be doubled.
    for i, p := range allPools {
        allPools[i] = nil
        for i := 0; i < int(p.localSize); i++ {
            l := indexLocal(p.local, i)
            l.private = nil
            for j := range l.shared {
                l.shared[j] = nil
            }
            l.shared = nil
        }
        p.local = nil
        p.localSize = 0
    }
    allPools = []*Pool{}
}

var (
    allPoolsMu Mutex
```

```
    allPools  []*Pool
)

func init() {
    runtime_registerPoolCleanup(poolCleanup)
}

func indexLocal(l unsafe.Pointer, i int) *poolLocal {
    lp := unsafe.Pointer(uintptr(l) + uintptr(i)*unsafe.Sizeof(poolLocal{}))
    return (*poolLocal)(lp)
}

// Implemented in runtime.
func runtime_registerPoolCleanup(cleanup func())
func runtime_procPin() int
func runtime_procUnpin()
```

在1.5版本中的一点点区别

pool.go

```
type Pool struct {
        local     unsafe.Pointer     // [P]poolLocal 数组指针
        localSize uintptr           // 数组内 poolLocal 的数量
        New func() interface{}      // 新建对象函数
}
type poolLocal struct {
        private interface{}         // 私有缓存区
        shared  []interface{}       // 可共享缓存区
        Mutex
        pad     [128]byte
}
```

pool.go

```
func indexLocal(l unsafe.Pointer, i int) *poolLocal {
        // 不去考虑 Pool.local，也就是 l 参数实际数组的长度，反正也不会超过 100
0000
        // 直接将其转换成大数组，然后按索引号返回 poolLocal 即可
        return &(*[1000000]poolLocal)(l)[i]
}
```

不要觉得无厘头，这种做法在C里很常见，甚至你在某些操作系统的源码里也会看到类似的东西。这么做不用去考虑P数量的变化，或者对_MaxGomaxprocs的修改，直接以性能优先。

以上总结：sync.Pool的定位不是做类似连接池的东西，它的用途仅仅是增加对象重用的几率，减少g的负担，在开销方面也不是很低,在调度方面和gc方面还需要串着多看，多理解其原理才行。

参考资料:

《Go并发编程第二版-郝林》

《Go语言学习笔记-雨痕》