1.sync.Pool 基本使用

https://golang.org/pkg/sync/

- sync.Pool的设计目的是存放已经分配但暂时不用的对象，供以后使用，减轻gc的代价
- 存储在Pool中的对象会随时被gc自动回收，因此Pool中对象的缓存期限为两次gc之间
- 不能定义sync.Pool的大小，其大小仅仅受限于内存的大小
- sync.Pool支持多协程之间共享

sync.Pool的使用非常简单，新建一个Pool对象时，提供一个New函数，表示当池中没有对象时，如何生成对象。然后使用Get和Put函数从Pool中取和存放临时对象。下面有一个简单的实例，直接运行是会打印两次“new an object”,去掉runtime.GC(),会发现只会调用一次New函数，实现了对象重用的目的。

```
package main

import (
    "fmt"
    "runtime"
    "sync"
)

func main() {
    p := &sync.Pool{
        New: func() interface{} {
            fmt.Println("new an object")
            return 0
        },
    }

    a := p.Get().(int)
    a = 100
    p.Put(a)
    runtime.GC()
    b := p.Get().(int)
    fmt.Println(a, b)
}
```

2.sync.Pool 如何支持多协程共享？

sync.Pool支持多协程共享，为了尽量减少竞争和加锁的操作，golang在设计的时候为每个P（核）都分配了一个子池，每个子池包含一个私有对象和共享列表。 私有对象只有对应的P能够访问，共享列表是与其它P共享的。

在golang的GMP调度模型中，协程G最终会被调度到某个固定的核P上。当一个协程执行Pool的get或者put方法时，会首先对改核P上的子池进行操作。一个P同一时间只能执行一个goroutine，因此对私有对象存取操作是不需要加锁的。共享列表是和其他P分享的，因此操作共享列表是需要加锁的。例如一个协程希望从某个Pool中获取对象，它包含以下几个步骤：

1. 判断协程所在的核P中的私有对象是否为空，如果非常则返回，并将改核P的私有对象置为空
2. 如果协程所在的核P中的私有对象为空，就去改核P的共享列表中获取对象（需要加锁）
3. 如果协程所在的核P中的共享列表为空，就去其它核的共享列表中获取对象（需要加锁）

4. 如果所有的核的共享列表都为空，就会通过New函数产生一个新的对象
在sync.Pool的源码中，每个核P的子池的结构如下所示：

```
// Local per-P Pool appendix.
type poolLocalInternal struct {
    private interface{}   // Can be used only by the respective P.
    shared  []interface{} // Can be used by any P.
    Mutex                 // Protects shared.
}
```

更加细致的sync.Pool源码分析，可参考http://jack-nie.github.io/go/golang-sync-pool.html

3.为什么不使用sync.pool实现连接池？

刚开始接触到sync.pool时，很容易让人联想到连接池的概念，但是经过仔细分析后发现sync.pool并不是适合作为连接池，主要有以下两个原因：
- 连接池的大小通常是固定的且受限制的，而sync.Pool是无法控制缓存对象的数量，他只受限于内存大小
- sync.Pool对象缓存的期限在两次gc之间,这点也和连接池非常不符合

golang中连接池通常利用channel的缓存特性实现。当需要连接时，从channel中获取，如果池中没有连接时，将阻塞或者新建连接，新建连接的数量不能超过某个限制。

https://github.com/goctx/generic-pool基于channel提供了一个通用连接池的实现

```
package pool

import (
    "errors"
    "io"
    "sync"
    "time"
)

var (
    ErrInvalidConfig = errors.New("invalid pool config")
    ErrPoolClosed    = errors.New("pool closed")
)

type Poolable interface {
    io.Closer
    GetActiveTime() time.Time
}

type factory func() (Poolable, error)

type Pool interface {
    Acquire() (Poolable, error) // 获取资源
    Release(Poolable) error     // 释放资源
    Close(Poolable) error       // 关闭资源
    Shutdown() error            // 关闭池
}
```

```
type GenericPool struct {
    sync.Mutex
    pool        chan Poolable
    maxOpen     int  // 池中最大资源数
    numOpen     int  // 当前池中资源数
    minOpen     int  // 池中最少资源数
    closed      bool // 池是否已关闭
    maxLifetime time.Duration
    factory     factory // 创建连接的方法
}

func NewGenericPool(minOpen, maxOpen int, maxLifetime time.Duration, factory factory) (*GenericPool,
    if maxOpen <= 0 || minOpen > maxOpen {
        return nil, ErrInvalidConfig
    }
    p := &GenericPool{
        maxOpen:     maxOpen,
        minOpen:     minOpen,
        maxLifetime: maxLifetime,
        factory:     factory,
        pool:        make(chan Poolable, maxOpen),
    }

    for i := 0; i < minOpen; i++ {
        closer, err := factory()
        if err != nil {
            continue
        }
        p.numOpen++
        p.pool <- closer
    }
    return p, nil
}

func (p *GenericPool) Acquire() (Poolable, error) {
    if p.closed {
        return nil, ErrPoolClosed
    }
    for {
        closer, err := p.getOrCreate()
        if err != nil {
            return nil, err
        }
        // 如果设置了超时且当前连接的活跃时间+超时时间早于现在，则当前连接已过期
        if p.maxLifetime > 0 && closer.GetActiveTime().Add(time.Duration(p.maxLifetime)).Before(time.
            p.Close(closer)
            continue
        }
        return closer, nil
    }
}
```

```
func (p *GenericPool) getOrCreate() (Poolable, error) {
    select {
    case closer := <-p.pool:
        return closer, nil
    default:
    }
    p.Lock()
    if p.numOpen >= p.maxOpen {
        closer := <-p.pool
        p.Unlock()
        return closer, nil
    }
    // 新建连接
    closer, err := p.factory()
    if err != nil {
        p.Unlock()
        return nil, err
    }
    p.numOpen++
    p.Unlock()
    return closer, nil
}

// 释放单个资源到连接池
func (p *GenericPool) Release(closer Poolable) error {
    if p.closed {
        return ErrPoolClosed
    }
    p.Lock()
    p.pool <- closer
    p.Unlock()
    return nil
}

// 关闭单个资源
func (p *GenericPool) Close(closer Poolable) error {
    p.Lock()
    closer.Close()
    p.numOpen--
    p.Unlock()
    return nil
}

// 关闭连接池，释放所有资源
func (p *GenericPool) Shutdown() error {
    if p.closed {
        return ErrPoolClosed
    }
    p.Lock()
    close(p.pool)
    for closer := range p.pool {
```

```
        closer.Close()
        p.numOpen--
    }
    p.closed = true
    p.Unlock()
    return nil
}
```

参考：

[1].https://blog.csdn.net/yongjian_lian/article/details/42058893
[2].https://segmentfault.com/a/1190000013089363
[3].http://jack-nie.github.io/go/golang-sync-pool.html