Go: The Idea Behind Sync.Pool

Hang Chu

August 16, 2021

Overview

Introduction

Motivation

- Repeated allocation and deallocation of objects that are expensive to create and have short lifetimes is wasteful.
- **Solution**: Create an object pool to better manage resources.
- **Object Pool Pattern**:
  - Initialize a pool of ready-to-use objects.
  - Unneeded used objects are returned to the pool.
  - If the pool is empty, a new object will be created and returned upon request.
  - The pool must be responsible for resetting used objects' state.

Introduce sync.Pool

- Many ways to implement the object pooling pattern, but focus on using sync.Pool today.
- How sync.Pool works: caching temporarily unused objects for later use.

Implementation

sync.Pool in action

- New returns a new object if pool is empty.
- personPool has 2 methods:
  - Get: takes an object out of the pool.
  - Put: returns the unused object back to the pool.
- How to measure efficiency of sync.Pool?

```go
type Person struct {
    Name string
}

// initializes pool
var personPool = sync.Pool{
    // returns a new object if pool is empty
    // when called by personPool.Put()
    New: func() interface{} { return new(Person) },
}

func main() {
    // gets a new instance
    newPerson := personPool.Get().(*Person)

    // defers release function so the instance
    // can be used again
    defer personPool.Put(newPerson)

    // uses the instance
    newPerson.Name = "Jack"
}
```

Figure 1: A simple sync.Pool example

Benchmark

Benchmarking two functions

- (De)allocation objects with and without a pool.
- Outer loop runs b.N times, with 10,000 allocations/operation (inner loop).

```go
func BenchmarkWithoutPool(b *testing.B) {
    var p *Person
    b.ReportAllocs()
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        for j := 0; j < 10000; j++ {
        p = new(Person)
        p.Age = 23
        }
    }
}

func BenchmarkWithPool(b *testing.B) {
    var p *Person
    b.ReportAllocs()
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        for j := 0; j < 10000; j++ {
        p = personPool.Get().(*Person)
        p.Age = 23
        personPool.Put(p)
        }
    }
}
```

Figure 2: sync.Pool benchmark

```
goos: linux
goarch: amd64
BenchmarkWithoutPool-4          4904         236638 ns/op          80000 B/op      10000 allocs/op
BenchmarkWithPool-4             5253         249298 ns/op              0 B/op          0 allocs/op
PASS
ok      _/home/jodi/tfs-03/lec-06/presentation  4.294s
```

Figure 3: Benchmark results

- BenchmarkWithoutPool took 80,000B to allocate 10,000 objects per operation, while BenmarkWithPool didn't take any.
- Not the best example to show the disadvantages of sync.Pool due to negligible difference in running time per operation between two functions.

No such thing as a free lunch

- Test performance of sync.Pool in a parallel setting.
- The sync.Pool version took **16X** more time per operation to run than the allocation one.

```go
func BenchmarkPool(b *testing.B) {
    var p sync.Pool
    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            p.Put( x: 1)
            p.Get()
        }
    })
}

func BenchmarkAllocation(b *testing.B) {
    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            i := 0
            i = i
        }
    })
}
```

Figure 4: Benchmarking sync.Pool and simple Allocation

```
BenchmarkPool-4          100000000      10.9 ns/op
BenchmarkAllocation-4    1000000000     0.674 ns/op
```

Figure 5: Benchmarking results

How sync.Pool works under the hood

- sync.Pool has 2 containers: local pool and victim cache.
- package sync registers init function to the runtime to clean the pool, which will be triggered by the garbage collector
- When GC is triggered, victim cache is cleared and objects from local pool will be moved there.
- Both new and returned objects are put in local pool.

```
func init() {
   runtime_registerPoolCleanup(poolCleanup)
}
```

Figure 6: sync package's init function

```
func poolCleanup() {
   // Drop victim caches from all pools.
   for _, p := range oldPools {
      p.victim = nil
      p.victimSize = 0
   }

   // Move primary cache to victim cache.
   for _, p := range allPools {
      p.victim = p.local
      p.victimSize = p.localSize
      p.local = nil
      p.localSize = 0
   }

   oldPools, allPools = allPools, nil
}
```

Figure 7: Pool cleanup function

Conclusion

Conclusion

- Need to create expensive objects a lot of times? Use object pooling, more specifically sync.Pool.
- Object pooling is slower than simple (de)allocation under certain conditions.