Use sync.Pool in Golang to reduce GC pressure

1 Preface

In a nutshell: save and reuse temporary objects, reduce memory allocation, and reduce GC pressure.

sync.Pool works roughly as follows.

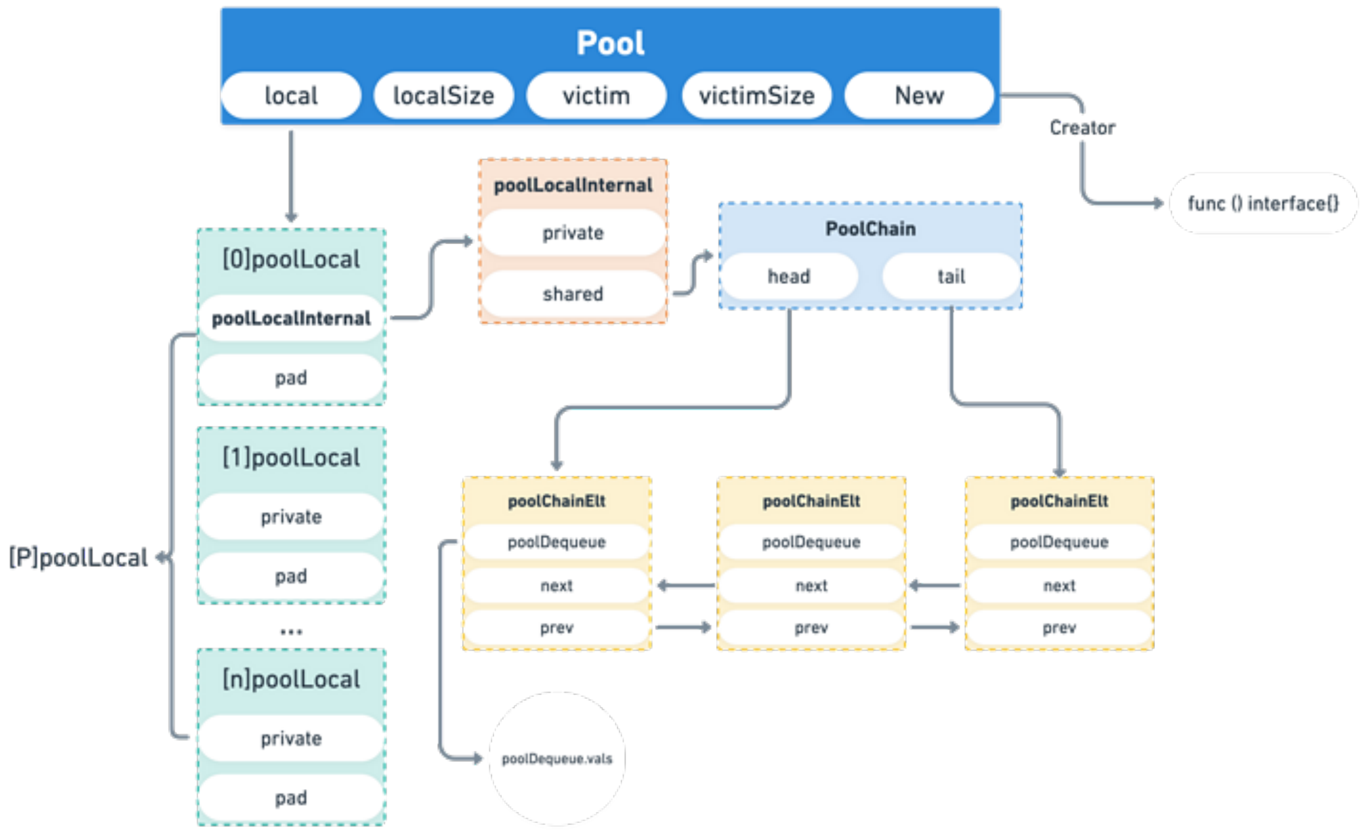
A local object pool poolLocal is created for each P to minimize concurrency conflicts, taking advantage of GMP features.

Each poolLocal has a private object, and access to private objects is given priority to avoid complex logic.

Use `pin` to lock the current P during Get and Put to prevent the goroutine from being preempted and causing program chaos.

Use object stealing mechanism to fetch objects from other P’s local object pool and victim during Get.

Make full use of CPU Cache feature to improve program performance.



As a simple example.

```
type Student struct {
    Name   string
    Age    int32
    Remark [1024]byte
}

```

```go
var buf, _ = json.Marshal(Student{Name: "Geektutu", Age: 25})

func unmarsh() {
    stu := &Student{}
    json.Unmarshal(buf, stu)
}
```

The deserialization of json is very common in text parsing and network communication, and when the program is very concurrent, a large number of temporary objects need to be created in a short time. These objects are allocated on the heap, which will put a lot of pressure on the GC and seriously affect the performance of the program.

Since version 1.3, Go has provided a mechanism for object reuse, the sync.Pool. sync.Pool is scalable and concurrency-safe, and its size is limited only by the size of memory. sync.Pool is used to store values that have been allocated but not used, and may be used in the future. This allows the system to reuse existing objects without having to go through memory allocation again, reducing the pressure on GC and thus improving system performance.

The size of the sync.Pool is scalable and will be dynamically expanded at high load, and objects stored in the pool will be automatically cleaned up if they become inactive.

2 How to use

The use of sync.Pool is very simple.

2.1 Declaring a pool of objects

You only need to implement the New function. If there are no objects in the pool, the New function will be called to create them.

```go
var studentPool = sync.Pool{
    New: func() interface{} {
        return new(Student)
    },
}
```

2.2 Get & Put

```go
stu := studentPool.Get().(*Student)
json.Unmarshal(buf, stu)
studentPool.Put(stu)
```

`Get()` is used to get an object from the object pool, because the return value is `interface{}` and therefore requires a type conversion.

`Put()`, on the other hand, returns the object pool after the object is used.

3 Performance tests

3.1 struct deserialization

```go
func BenchmarkUnmarshal(b *testing.B) {
    for n := 0; n < b.N; n++ {
        stu := &Student{}
        json.Unmarshal(buf, stu)
    }
}

func BenchmarkUnmarshalWithPool(b *testing.B) {
    for n := 0; n < b.N; n++ {
        stu := studentPool.Get().(*Student)
        json.Unmarshal(buf, stu)
        studentPool.Put(stu)
    }
}
```

The test results are as follows.

```
$ go test -bench . -benchmem
goos: darwin
goarch: amd64
pkg: example/hpg-sync-pool
BenchmarkUnmarshal-8           1993    559768 ns/op    5096 B/op 7 allocs/op
BenchmarkUnmarshalWithPool-8   1976    550223 ns/op     234 B/op 6 allocs/op
PASS
ok      example/hpg-sync-pool   2.334s
```

In this example, because the Student structure has a small memory footprint, memory allocation takes almost no time at all. The standard library json deserialization uses reflection, which is less efficient and takes up most of the time, so the final execution time between the two methods is almost unchanged. However, the memory footprint is an order of magnitude worse. After using `sync.Pool`, the memory footprint is only 234/5096 = 1/22 of unused, which has a big impact on GC.

3.2 bytes.Buffer

```go
var bufferPool = sync.Pool{
    New: func() interface{} {
        return &bytes.Buffer{}
    },
}

var data = make([]byte, 10000)

func BenchmarkBufferWithPool(b *testing.B) {
    for n := 0; n < b.N; n++ {
        buf := bufferPool.Get().(*bytes.Buffer)
        buf.Write(data)
        buf.Reset()
        bufferPool.Put(buf)
    }
}

func BenchmarkBuffer(b *testing.B) {
    for n := 0; n < b.N; n++ {
        var buf bytes.Buffer
        buf.Write(data)
    }
}
```

The test results are as follows.

```
BenchmarkBufferWithPool-8    8778160     133 ns/op        0 B/op    0 allocs/op
BenchmarkBuffer-8             906572    1299 ns/op    10240 B/op    1 allocs/op
```

This example creates a pool of `bytes.Buffer` objects and performs only one simple `Write` operation at a time, which is a pure memory mover and takes almost negligible time. Memory allocation and reclamation, on the other hand, take up more time and therefore have a greater impact on the overall performance of the program.

4 Use in standard libraries

4.1 fmt.Printf

The Go language standard library also makes extensive use of `sync.Pool`, such as `fmt` and `encoding/json`.

The following is the source code for `fmt.Printf` (go/src/fmt/print.go).

```
// go 1.13.6

// pp is used to store a printer's state and is reused with sync.Pool to avoid
allocations.
type pp struct {
    buf buffer
    ...
}

var ppFree = sync.Pool{
    New: func() interface{} { return new(pp) },
}

// newPrinter allocates a new pp struct or grabs a cached one.
func newPrinter() *pp {
    p := ppFree.Get().(*pp)
    p.panicking = false
    p.erroring = false
    p.wrapErrs = false
    p.fmt.init(&p.buf)
    return p
}

// free saves used pp structs in ppFree; avoids an allocation per invocation.
func (p *pp) free() {
    if cap(p.buf) > 64<<10 {
        return
    }

    p.buf = p.buf[:0]
    p.arg = nil
    p.value = reflect.Value{}
    p.wrappedErr = nil
    ppFree.Put(p)
}

func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error) {
```

```go
	p := newPrinter()
	p.doPrintf(format, a)
	n, err = w.Write(p.buf)
	p.free()
	return
}

// Printf formats according to a format specifier and writes to standard output.
// It returns the number of bytes written and any write error encountered.
func Printf(format string, a ...interface{}) (n int, err error) {
	return Fprintf(os.Stdout, format, a...)
}
```

Calls to `fmt.Printf` are very frequent. Using `sync.Pool` to reuse pp objects can greatly improve performance, reduce memory usage, and reduce GC pressure.