Golang 常见性能问题总结

Introduction

1.这篇文章总结了一些调优过程遇到以及引发了事故的性能坑，未必适用于所有系统，也不一定任何地方使用这些东西都会引发故障。

2.**使用之前先了解，使用之时要评估，使用之后要测试。**

3.性能瓶颈上，**I.O才是最常见的瓶颈：网络I.O，访问数据库，访问磁盘文件系统等等。**

4.危险等级主要根据是否已经出现故障来主观评估，对于已经在项目中踩过坑的点，去使用的时候应该谨慎评估。

copier.copy/deepcopy等

危险等级：4/5星，避免使用，如果有非用不可的地方，需要评估调用频率和影响。

当出现大对象，或者在结构体中有多层嵌套的时候，deepcopy的表现会差得多，因为在deepcopy中也大量使用了反射reflect来获取对象的类型。可能不同的deepcopy实现方式不一样，部分也有对reflect的性能进行优化，但也只是比直接使用json好，或者是在同类库中脱颖而出。在高频访问时过多的分配内存和引发GC，都会拖慢整体的速度。

测试代码

```
package tricks

import (
        "github.com/json-iterator/go"
        "github.com/mohae/deepcopy"
        "testing"
        "time"

        jzcopier "github.com/jinzhu/copier"
)

type TestStructBigA struct {
        Field0  int
        Field1  int8
        Field2  int16
        Field3  int32
        Field4  int64
        Field5  float32
        Field6  float64
        Field7  string
        Field8  bool
        Field9  time.Time
        Field10 int
        Field11 int8
        Field12 int16
        Field13 int32
        Field14 int64
        Field15 float32
        Field16 float64
        Field17 string
        Field18 bool
        Field19 time.Time
        Field20 int
        Field21 int8
        Field22 int16
        Field23 int32
        Field24 int64
        Field25 float32
        Field26 float64
        Field27 string
        Field28 bool
        Field29 time.Time
        Field30 int
```

```
        Field31 int8
        Field32 int16
        Field33 int32
        Field34 int64
        Field35 float32
        Field36 float64
        Field37 string
        Field38 bool
        Field39 time.Time
        Field40 int
        Field41 int8
        Field42 int16
        Field43 int32
        Field44 int64
        Field45 float32
        Field46 float64
        Field47 string
        Field48 bool
        Field49 time.Time
        Field50 int
        Field51 int8
        Field52 int16
        Field53 int32
        Field54 int64
        Field55 float32
        Field56 float64
        Field57 string
        Field58 bool
        Field59 time.Time
        Field60 int
        Field61 int8
        Field62 int16
        Field63 int32
        Field64 int64
        Field65 float32
        Field66 float64
        Field67 string
        Field68 bool
        Field69 time.Time
        Field70 int
        Field71 int8
        Field72 int16
        Field73 int32
        Field74 int64
        Field75 float32
        Field76 float64
        Field77 string
        Field78 bool
        Field79 time.Time
        Field80 int
        Field81 int8
        Field82 int16
        Field83 int32
        Field84 int64
        Field85 float32
        Field86 float64
        Field87 string
        Field88 bool
        Field89 time.Time
        Field90 int
        Field91 int8
        Field92 int16
        Field93 int32
        Field94 int64
        Field95 float32
        Field96 float64
        Field97 string
        Field98 bool
        Field99 time.Time
}
type TestStructBigB struct {
```

```
Field0  int
Field1  int8
Field2  int16
Field3  int32
Field4  int64
Field5  float32
Field6  float64
Field7  string
Field8  bool
Field9  time.Time
Field10 int
Field11 int8
Field12 int16
Field13 int32
Field14 int64
Field15 float32
Field16 float64
Field17 string
Field18 bool
Field19 time.Time
Field20 int
Field21 int8
Field22 int16
Field23 int32
Field24 int64
Field25 float32
Field26 float64
Field27 string
Field28 bool
Field29 time.Time
Field30 int
Field31 int8
Field32 int16
Field33 int32
Field34 int64
Field35 float32
Field36 float64
Field37 string
Field38 bool
Field39 time.Time
Field40 int
Field41 int8
Field42 int16
Field43 int32
Field44 int64
Field45 float32
Field46 float64
Field47 string
Field48 bool
Field49 time.Time
Field50 int
Field51 int8
Field52 int16
Field53 int32
Field54 int64
Field55 float32
Field56 float64
Field57 string
Field58 bool
Field59 time.Time
Field60 int
Field61 int8
Field62 int16
Field63 int32
Field64 int64
Field65 float32
Field66 float64
Field67 string
Field68 bool
Field69 time.Time
Field70 int
```

```
        Field71 int8
        Field72 int16
        Field73 int32
        Field74 int64
        Field75 float32
        Field76 float64
        Field77 string
        Field78 bool
        Field79 time.Time
        Field80 int
        Field81 int8
        Field82 int16
        Field83 int32
        Field84 int64
        Field85 float32
        Field86 float64
        Field87 string
        Field88 bool
        Field89 time.Time
        Field90 int
        Field91 int8
        Field92 int16
        Field93 int32
        Field94 int64
        Field95 float32
        Field96 float64
        Field97 string
        Field98 bool
        Field99 time.Time
}

var json = jsoniter.ConfigCompatibleWithStandardLibrary

func jsonCopy(dest, src interface{}) error {
        bytes, _ := json.Marshal(src)
        return json.Unmarshal(bytes, dest)
}

func BenchmarkJinzhuCopyBig(b *testing.B) {
        var src = TestStructBigA{}
        var dest TestStructBigB
        for i := 0; i < b.N; i++ {
                jzcopier.Copy(&dest, src)
        }
}

func BenchmarkDeepCopyBig(b *testing.B) {
        var src = TestStructBigA{}
        //var dest TestStructBigB
        for i := 0; i < b.N; i++ {
                _ = deepcopy.Copy(src).(TestStructBigA)
        }

}

func BenchmarkJSONCopyBig(b *testing.B) {
        var src = TestStructBigA{}
        var dest TestStructBigB
        for i := 0; i < b.N; i++ {
                jsonCopy(&dest, src)
        }
}

func BenchmarkThisRepoCopyBig(b *testing.B) {
        var src = TestStructBigA{}
        var dest TestStructBigB
        for i := 0; i < b.N; i++ {
                dest.Field0 = src.Field0
                dest.Field1 = src.Field1
                dest.Field2 = src.Field2
                dest.Field3 = src.Field3
```

```
dest.Field4 = src.Field4
dest.Field5 = src.Field5
dest.Field6 = src.Field6
dest.Field7 = src.Field7
dest.Field8 = src.Field8
dest.Field9 = src.Field9
dest.Field10 = src.Field10
dest.Field11 = src.Field11
dest.Field12 = src.Field12
dest.Field13 = src.Field13
dest.Field14 = src.Field14
dest.Field15 = src.Field15
dest.Field16 = src.Field16
dest.Field17 = src.Field17
dest.Field18 = src.Field18
dest.Field19 = src.Field19
dest.Field20 = src.Field20
dest.Field21 = src.Field21
dest.Field22 = src.Field22
dest.Field23 = src.Field23
dest.Field24 = src.Field24
dest.Field25 = src.Field25
dest.Field26 = src.Field26
dest.Field27 = src.Field27
dest.Field28 = src.Field28
dest.Field29 = src.Field29
dest.Field30 = src.Field30
dest.Field31 = src.Field31
dest.Field32 = src.Field32
dest.Field33 = src.Field33
dest.Field34 = src.Field34
dest.Field35 = src.Field35
dest.Field36 = src.Field36
dest.Field37 = src.Field37
dest.Field38 = src.Field38
dest.Field39 = src.Field39
dest.Field40 = src.Field40
dest.Field41 = src.Field41
dest.Field42 = src.Field42
dest.Field43 = src.Field43
dest.Field44 = src.Field44
dest.Field45 = src.Field45
dest.Field46 = src.Field46
dest.Field47 = src.Field47
dest.Field48 = src.Field48
dest.Field49 = src.Field49
dest.Field50 = src.Field50
dest.Field51 = src.Field51
dest.Field52 = src.Field52
dest.Field53 = src.Field53
dest.Field54 = src.Field54
dest.Field55 = src.Field55
dest.Field56 = src.Field56
dest.Field57 = src.Field57
dest.Field58 = src.Field58
dest.Field59 = src.Field59
dest.Field60 = src.Field60
dest.Field61 = src.Field61
dest.Field62 = src.Field62
dest.Field63 = src.Field63
dest.Field64 = src.Field64
dest.Field65 = src.Field65
dest.Field66 = src.Field66
dest.Field67 = src.Field67
dest.Field68 = src.Field68
dest.Field69 = src.Field69
dest.Field70 = src.Field70
dest.Field71 = src.Field71
dest.Field72 = src.Field72
dest.Field73 = src.Field73
dest.Field74 = src.Field74
```

```
                dest.Field75 = src.Field75
                dest.Field76 = src.Field76
                dest.Field77 = src.Field77
                dest.Field78 = src.Field78
                dest.Field79 = src.Field79
                dest.Field80 = src.Field80
                dest.Field81 = src.Field81
                dest.Field82 = src.Field82
                dest.Field83 = src.Field83
                dest.Field84 = src.Field84
                dest.Field85 = src.Field85
                dest.Field86 = src.Field86
                dest.Field87 = src.Field87
                dest.Field88 = src.Field88
                dest.Field89 = src.Field89
                dest.Field90 = src.Field90
                dest.Field91 = src.Field91
                dest.Field92 = src.Field92
                dest.Field93 = src.Field93
                dest.Field94 = src.Field94
                dest.Field95 = src.Field95
                dest.Field96 = src.Field96
                dest.Field97 = src.Field97
                dest.Field98 = src.Field98
                dest.Field99 = src.Field99
        }
}
```

benchmark结果

```
GOROOT=/usr/local/go #gosetup
GOPATH=/Users/yi.liu/GolandProjects:/Users/yi.liu/go #gosetup
/usr/local/go/bin/go test -c -o /private/var/folders/74/8n908j3d3lqcbvymzc7dxgbw0000gn/T
/___gobench_example_tricks example/tricks #gosetup
/private/var/folders/74/8n908j3d3lqcbvymzc7dxgbw0000gn/T/___gobench_example_tricks -test.v -test.bench . -test.
run ^$ #gosetup
goos: darwin
goarch: amd64
pkg: example/tricks
BenchmarkJinzhuCopyBig-16                11953             97112 ns/op
BenchmarkDeepCopyBig-16                 131161              8854 ns/op
BenchmarkJSONCopyBig-16                  48646             24161 ns/op
BenchmarkThisRepoCopyBig-16          1000000000                0.517 ns/op
PASS
```

deepcopy最好不要在高频链路中使用，如果使用一定要控制频率。否则很容易成为性能瓶颈。

反例：

2021.09.01-LCOS-内存异常暴增故障报告

reflect反射

危险等级：4/5星，避免在高频链路中使用，高频率会加剧性能的下降

反射慢的原因是基于golang的api设计，要取得反射的对象类型和对象值需要两套不同的反射函数。

```
//fieldreflect.StructField
type_ := reflect.TypeOf(obj)
field, _ := type_.FieldByName("hello")

//fieldValue
type_ := reflect.ValueOf(obj)
fieldValue := type_.FieldByName("hello")
```

这个过程其实一直在申请reflect.Value这个结构体的空间，这是导致reflect操作构造对象效率低的主要原因。

反例：

2021.11.19-LCOS-CPU暴涨导致服务挂掉

Lock锁

危险等级:3/5星，不恰当的并发不仅无法提高性能，反而因为频繁的调度降低了性能

CPU一个核心在执行程序的时候没有真正物理设备上的并行处理 (Simultaneously Processing), **一个CPU的一个核心永远只会在一个时间段处理一个任务或者一条指令。**那么我们提到的两个场景分别是利用单核CPU时间片 (Time slice) 来进行**抢占式**调度 (Preemption) 和真正的**多核多线程**并行执行。

golang提供两种类型的锁，互斥锁(sync.Mutex)和读写锁(sync.RWMutex)。

按照不同比例的读操作和写操作来测试两种锁的性能。

- 读写比9:1
- 读写比1:9
- 读写比5:5

benchmark结果

```
GOROOT=/usr/local/go #gosetup
GOPATH=/Users/yi.liu/GolandProjects:/Users/yi.liu/go #gosetup
/usr/local/go/bin/go test -c -o /private/var/folders/74/8n908j3d3lqcbvymzc7dxgbw0000gn/T
/___gobench_example_tricks example/tricks #gosetup
/private/var/folders/74/8n908j3d3lqcbvymzc7dxgbw0000gn/T/___gobench_example_tricks -test.v -test.bench . -test.
run ^$ #gosetup
goos: darwin
goarch: amd64
pkg: example/tricks
BenchmarkReadMore-16                    112          10919102 ns/op
BenchmarkReadMoreRW-16                  854           1474592 ns/op
BenchmarkWriteMore-16                   100          10730777 ns/op
BenchmarkWriteMoreRW-16                 100          10052904 ns/op
BenchmarkEqual-16                       100          11454162 ns/op
BenchmarkEqualRW-16                     187           6087196 ns/op
PASS
```

最终可以看到：读多的场景，读写锁比互斥锁快7.4倍；写多的时候，两种锁性能差不多；读写比1:1时，读写锁大概快1.9倍。

如果降低操作的单位时间，那么读写锁的性能优势会被缩小，这是因为锁的阻塞时间变小了，互斥锁的损耗变小。

锁还是Channel

*Effective Go*中有一句话：**通过通信共享内存，而不是通过共享内存而通信。**说明channel作为go的特性之一在并发场景是高优先级的，然而channel并非万能。

Channel更适合于数据流动的场景，而锁其实更擅长数据不流动，位置不变的场景，每个时间段只给一个协程访问数据的权限。

个人认为，可以通过如下三个问题考量：

1.是否转让数据所有权？——把某个数据发送给其他协程。

2.是否分发任务？——每个任务都是一个数据。

3.是否组合数据？——交流异步结果，结果汇总为一个数据。

如果有上述的情况，使用channel是一个好的选择；反之如果只是维护一个状态，或者做一个缓存，那么锁比channel合适。

benchmark

```
import (
        "sync"
        "testing"
)

var mutex = sync.Mutex{}
var ch = make(chan bool, 1)

func UseMutex() {
        mutex.Lock()
        mutex.Unlock()
}
func UseChan() {
        ch <- true
        <-ch
}


GOROOT=/usr/local/go #gosetup
GOPATH=/Users/yi.liu/GolandProjects:/Users/yi.liu/go #gosetup
/usr/local/go/bin/go test -c -o /private/var/folders/74/8n908j3d3lqcbvymzc7dxgbw0000gn/T
/___gobench_lock_test_go example/tricks #gosetup
/private/var/folders/74/8n908j3d3lqcbvymzc7dxgbw0000gn/T/___gobench_lock_test_go -test.v -test.bench
"^BenchmarkUseMutex|BenchmarkUseChan$" -test.run ^$ #gosetup
goos: darwin
goarch: amd64
pkg: example/tricks
BenchmarkUseMutex-16            99958624                10.6 ns/op
BenchmarkUseChan-16             28737012                41.0 ns/op
PASS
```

可以看到保证并发安全的情况下，互斥锁要更优秀一些。

序列化、反序列化

危险等级：3.5/5星，序列化只有遇到大的结构并且高频的时候才会变成性能瓶颈

无论是gob还是json，本质上都利用了reflect的功能，既然reflect的性能本身有限制，那么序列化被滥用一定是会影响性能的。

理论上gob的性能会比json要好一些，但是泛用性更差，基本只适用于go - go的交互、go - C的交互。当对象很小的时候，尽量避免使用序列化/反序列化组合而是直接去赋值，不然会有些得不偿失。在高频接口的链路中出现过多的序列化操作也会产生性能瓶颈。

在使用json的时候，使用第三方的json库一般情况是要优于golang自带的encoding/json库，因为或多或少都有一些针对性优化以提升操作性能。同样地，protobuf的数据交互也不推荐直接使用官方的protobuf库（一样使用了反射导致降低了效率）。

在当前的SLS业务中，product → lane → line的关系会导致request body和response的放大，这对序列化/反序列化的操作而言是加重负担的，如果每一层链路都在执行这个操作就需要考虑性能的损失。

反例：

2021.11.04 压测导致Icos系统负载高，接口超时使得点线流向的channel没有返回导致PMS无法下单

字符串

危险等级：1/5星，一般业务上不会产生频繁的字符串操作

string类型在go里不可变，所以在拼接过程中不可避免的产生对象分配和值拷贝。直接使用字符串相加或者常用的fmt.Sprintf放在高频场合其实是比较占用CPU时钟的。

测试代码

```
import (
        "bytes"
        "fmt"
        "strings"
        "testing"
)

const (
        sss = "hello world!"
        cnt = 10000
)

var expected = strings.Repeat(sss, cnt)

func BenchmarkStringConcat(b *testing.B) {
        var result string
        for n := 0; n < b.N; n++ {
                var str string
                for i := 0; i < cnt; i++ {
                        str += sss
                }
                result = str
        }
        b.StopTimer()
        if result != expected {
                b.Errorf("unexpected result; got=%s, want=%s", string(result), expected)
        }
}

func BenchmarkStringSprintf(b *testing.B) {
        var result string
        for n := 0; n < b.N; n++ {
                var str string
                for i := 0; i < cnt; i++ {
                        str = fmt.Sprintf("%s%s", str, sss)
                }
                result = str
        }
        b.StopTimer()
        if result != expected {
                b.Errorf("unexpected result; got=%s, want=%s", string(result), expected)
        }
}

func BenchmarkStringJoin(b *testing.B) {
        var result string
        for n := 0; n < b.N; n++ {
                var str string
                for i := 0; i < cnt; i++ {
                        str = strings.Join([]string{str, sss}, "")
                }
                result = str
        }
        b.StopTimer()
        if result != expected {
                b.Errorf("unexpected result; got=%s, want=%s", string(result), expected)
        }
}

func BenchmarkBufferWrite(b *testing.B) {
        var result string
        for n := 0; n < b.N; n++ {
                buf := new(bytes.Buffer)
                for i := 0; i < cnt; i++ {
```

```
                        buf.WriteString(sss)
                }
                result = buf.String()
        }
        b.StopTimer()
        if result != expected {
                b.Errorf("unexpected result; got=%s, want=%s", string(result), expected)
        }
}

func BenchmarkBytesAppend(b *testing.B) {
        var result string
        for n := 0; n < b.N; n++ {
                var bbb []byte

                for i := 0; i < cnt; i++ {
                        bbb = append(bbb, sss...)
                }
                result = string(bbb)
        }
        b.StopTimer()
        if result != expected {
                b.Errorf("unexpected result; got=%s, want=%s", string(result), expected)
        }
}

func BenchmarkStringCopy(b *testing.B) {
        var result string
        for n := 0; n < b.N; n++ {
                tsl := len(sss) * cnt
                bs := make([]byte, tsl)
                bl := 0

                for bl < tsl {
                        bl += copy(bs[bl:], sss)
                }

                result = string(bs)
        }
        b.StopTimer()
        if result != expected {
                b.Errorf("unexpected result; got=%s, want=%s", string(result), expected)
        }
}

func BenchmarkStringBuilder(b *testing.B) {
        var result string
        for n := 0; n < b.N; n++ {
                var builder strings.Builder

                for i := 0; i < cnt; i++ {
                        builder.WriteString(sss)
                }

                result = builder.String()
        }
        b.StopTimer()
        if result != expected {
                b.Errorf("unexpected result; got=%s, want=%s", string(result), expected)
        }
}
```

无论是string相加，fmt.Sprintf还是join，比起bytes.Buffer或strings.Builder这些要慢很多，在高频使用的场合下不太合适。

*转换不同类型是尽量使用strconv显式的指定类型转换，而不是直接fmt.Sprintf去动态解析。

benchmark

```
GOROOT=/usr/local/go #gosetup
GOPATH=/Users/yi.liu/GolandProjects:/Users/yi.liu/go #gosetup
/usr/local/go/bin/go test -c -o /private/var/folders/74/8n908j3d3lqcbvymzc7dxgbw0000gn/T
/___gobench_string_test_go example #gosetup
/private/var/folders/74/8n908j3d3lqcbvymzc7dxgbw0000gn/T/___gobench_string_test_go -test.v -test.bench
"^BenchmarkStringConcat|BenchmarkStringSprintf|BenchmarkStringJoin|BenchmarkBufferWrite|BenchmarkBytesAppend|Ben
chmarkStringCopy|BenchmarkStringBuilder$" -test.run ^$ #gosetup
goos: darwin
goarch: amd64
pkg: example
BenchmarkStringConcat-16                  21          52217861 ns/op
BenchmarkStringSprintf-16                 12         102727589 ns/op
BenchmarkStringJoin-16                    18          62943255 ns/op
BenchmarkBufferWrite-16                10000            106835 ns/op
BenchmarkBytesAppend-16                16249             69185 ns/op
BenchmarkStringCopy-16                 17965             61900 ns/op
BenchmarkStringBuilder-16              15510             79083 ns/op
PASS
```

简单来说，**在只读场景里，字符串很合适，即便是做了重新分片。但是有其它操作的话，使用[]bytes会是更好的选择。**

strings.Builder的局限

1.输入类型必须是string，所以当业务代码中混合类型转换时要把输入都变为string类型（float/int等等）。

2.在不知道要拼接的字符串规模时，builder可能会多次扩容甚至引起GC，这是得不偿失的。

最好是预先估计一个要拼接的字符串规模，利用Grow()方法给builder预先分配空间。

Apollo配置读取

在go-chassis框架集成的apollo客户端中， 在一个高频链路中反复读取配置比较消耗性能。（这一点由Lcos系统发现）

在之前的agollo集成中似乎没有这个问题。

Slice

危险等级：1/5星，性能的极值差距并不会特别大

slice和Map比较常见，由于使用范围比较广，场景较多，可优化的空间并不是很大。

slice底层是数组，slice本身是引用类型。因此有两个地方容易出现问题：

1.当slice的容量用完再继续添加元素时需要扩容，而这个扩容会把申请新的空间，把老的内容复制到新的空间，这是一个非常耗时的操作。

2.只用slice的其中一部分时，整个slice的内存并不会释放，比如每次取末尾2个元素的操作，实际上是re-slice，原来的slice内存仍然被占据直至GC发生。

比较推荐的方法：**知道capacity时预先分配好，避免反复申请内存空间和元素复制；复用slice。**

元素包含struct的情况

需要注意for和range的区别，如果使用了类型为[]struct{}的数组，那么遍历下标时for和range的开销差不多，遍历元素时，for的效率要比range好得多。

测试代码

```
type Item struct {
        id  int
        val [4096]byte
}

func BenchmarkForStruct(b *testing.B) {
        var items [1024]Item
        for i := 0; i < b.N; i++ {
                length := len(items)
                var tmp int
                for k := 0; k < length; k++ {
                        tmp = items[k].id
                }
                _ = tmp
        }
}

func BenchmarkRangeIndexStruct(b *testing.B) {
        var items [1024]Item
        for i := 0; i < b.N; i++ {
                var tmp int
                for k := range items {
                        tmp = items[k].id
                }
                _ = tmp
        }
}

func BenchmarkRangeStruct(b *testing.B) {
        var items [1024]Item
        for i := 0; i < b.N; i++ {
                var tmp int
                for _, item := range items {
                        tmp = item.id
                }
                _ = tmp
        }
}
```

benchmark

```
GOROOT=/usr/local/go #gosetup
GOPATH=/Users/yi.liu/GolandProjects:/Users/yi.liu/go #gosetup
/usr/local/go/bin/go test -c -o /private/var/folders/74/8n908j3d3lqcbvymzc7dxgbw0000gn/T
/___gobench_loop_test_go example/tricks #gosetup
/private/var/folders/74/8n908j3d3lqcbvymzc7dxgbw0000gn/T/___gobench_loop_test_go -test.v -test.bench
"^BenchmarkForStruct|BenchmarkRangeIndexStruct|BenchmarkRangeStruct$" -test.run ^$ #gosetup
goos: darwin
goarch: amd64
pkg: example/tricks
BenchmarkForStruct-16                    3602899               296 ns/op
BenchmarkRangeIndexStruct-16             2212918               549 ns/op
BenchmarkRangeStruct-16                     4620            266875 ns/op
PASS
```

如果把类型换成[]*struct{}，那么for和range的性能才会差不多。

所以在设计batch接口的时候，**尽量使用[]*struct{}去传递参数，而不是[]struct{}**

e.g

```
func BatchGetLineServiceableAreaV2WithCache(ctx context.Context, batchReqList []
*BatchGetLineServiceableAreaRequestV2, scene int) ([]*BatchGetLineServiceableAreaResponse, retcode.
RetcodeMessager) {}
```

GC

危险等级：一般不太需要从这方面着手优化

一般情况下会适用两个原则：**内存分配比计算更昂贵。栈对象比堆对象代价低。**

通过go trace看看程序运行的GC开销，如果GC时间过长说明需要回收的临时对象太多。

可以考虑这些方法：

1. 调整GC频率。不让程序本身频繁的GC，代价是常驻内存会升高，对于高内存消耗的项目不适用。
2. 复用对象。上述的string slice等等就可以通过对象池复用来减少重复的分配，sync.Pool的使用也比较简单，而且可以动态扩容，Go 1.13优化了驻留的时间不再会因为GC快速的回收。
3. 在时间和精力允许的情况下，对一些关键链路的函数做逃逸分析，小对象不使用指针，尽量降低GC的压力，让这些对象不逃逸到堆上。

   原因：

   a. 栈对象随着函数结束而回收，不是所有的局部变量都是栈对象，有可能会逃逸到堆上（返回之后被上层引用等等）。
   b. Golang的GC会有STW阶段，停止所有goroutine的工作，所以过多的堆对象增加了触发GC的次数和单次GC扫描的时间等等，无形中拖累了整体性能。

更多...

不同的项目面临的性能瓶颈往往不同，善用性能分析工具比较重要：pprof, trace, GC分析等等。

调优在抽象层次上由高到低：优先关注数据结构，算法，以及正确的解耦合；做完这些事情再去细看输入输出，批处理，并发，标准库的使用，内存管理等。