GoCN
Go中国社区 GOCN.VIP

Go性能优化之路

腾讯，陈一枭（tensorchen）

2020.01.04 Gopher Meetup Shenzhen

个人简介

腾讯高级工程师

Going框架负责人

tRPC框架Go语言版本负责人

PCG代码委员会Go分会成员

1.何时性能优化

2.如何性能优化

3.性能优化实践

1. 何时性能优化

1.何时开始

每个优化都有成本。此外优化后的代码很少比未优化的版本简单。

“你应该优化吗？”是的，但是只有当问题很重要时，程序真的太慢了，并且能够在保证正确性，稳健性和清晰度的同时变得更快。

假设搜索引擎需要跨越多个数据中心的30,000台机器，这些机器每年的成本约为1,000美元。如果你可以将软件的速度提高一倍，这可以为公司节省每年1500万美元。即使只有一个开发人员花费整整一年时间才能将性能提高也只会付出1%的代价。

权衡优化成本和优化价值。从而决策是否需要改变你的程序。

2.何时停止

优化往往是一个收益递减的游戏。你需要知道何时停止。

目标必须具体。

如果目标是改进CPU，那么什么是可接受的速度。你想要将当前的性能提高2倍吗？10倍？

如果目标是减少内存使用量，对于内存使用情况的变化，可以接受的速度有多慢？你愿意放弃什么来换取较低的空间需求？

2.何时停止

停止优化的信号

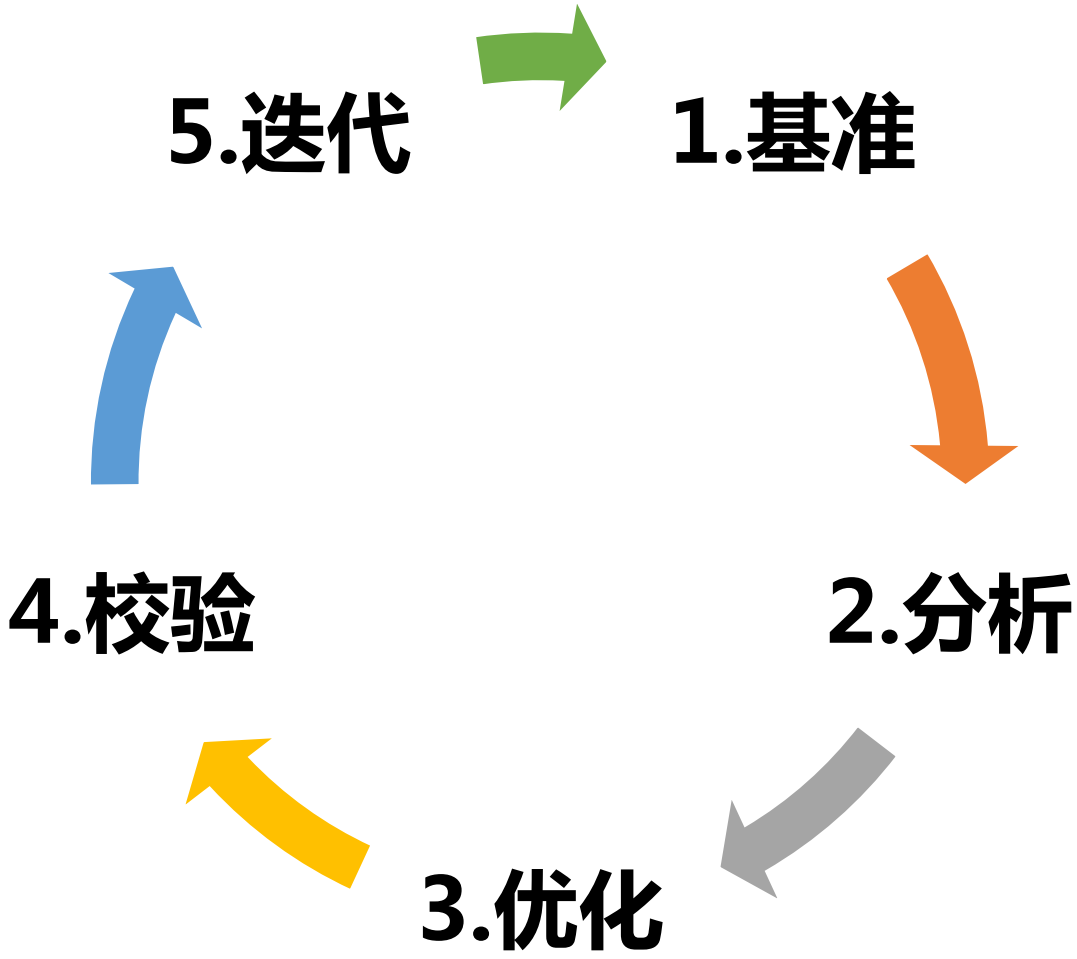
此外，还存在一些具体的信号可以告知我们在优化具体点时已经基本达到性能极限，可以停止了。

例如：优化网络I/O服务

如果发现Read/Write操作均消耗在syscall.Syscall则无需继续优化。

(注：前提是必要的syscall.Syscall调用，否则还可以使用缓存Buffer技术减少syscall被调次数)

2. 如何性能优化



1.基准

在尝试改善一段代码的性能之前，首先我们必须了解其当前性能。

优化是一种重构形式。但是，每一步不是改进源代码的某些方面（代码重复，清晰度等），而是可以提高性能的某些方面：降低CPU，内存使用率，延迟等。这种改进通常以可读性为代价。这意味着除了一套**全面的单元测试**（以确保你的更改没有破坏任何内容）之外，你还需要一套很好的**基准测试**，以确保您的更改对性能产生预期的影响。你必须能够验证您的更改是否真的在降低CPU。有时候你认为会改善性能的变化实际上会变成零或负变化。在这些情况下，务必确保撤消修改的程序。

1.基准

golang标准库testing库，提供benchmark功能

执行 go test –bench=. 进行基准测试

```go
func BenchmarkItoaStrconv(b *testing.B) {
  for n := 0; n < b.N; n++ {
    result = strconv.Itoa(n)
  }
}

func BenchmarkItoaFmt(b *testing.B) {
  for n := 0; n < b.N; n++ {
    result = fmt.Sprint(n)
  }
}
```

```
goos: darwin
goarch: amd64
pkg: git.code.oa.com/tensorchen/go-performance/examples/ff
BenchmarkItoaStrconv-12          39405439                29.8 ns/op
BenchmarkItoaFmt-12              14001330                84.3 ns/op
```

2.分析

关注瓶颈。 如果你将运行时间仅占5%的例程速度提高一倍，那么整个挂钟的速度只有2.5%。另一方面，将80%的时间加速10%的例程将使运行时间提高近8%。

定位瓶颈。 工欲善其事，必先利其器，熟练掌握和运用各种性能分析工具。

GoCN
Go中国社区 GOCN.VIP

分析工具：GODEBUG

使用：GODEBUG=gctrace=1 ./xxxx

```
gc 1 @0.047s 20%: 0.006+57+0.004 ms clock, 0.078+85/170/409+0.054 ms cpu, 383->384->382 MB, 384 MB goal, 12 P
gc 2 @0.112s 20%: 0.022+37+0.042 ms clock, 0.26+0.92/110/229+0.51 ms cpu, 383->386->384 MB, 764 MB goal, 12 P
gc 3 @3.416s 1%: 1.9+32+0.003 ms clock, 23+1.6/95/205+0.047 ms cpu, 589->589->427 MB, 768 MB goal, 12 P
gc 4 @8.906s 0%: 0.007+28+0.024 ms clock, 0.087+2.0/83/220+0.29 ms cpu, 739->739->442 MB, 854 MB goal, 12 P
gc 5 @16.672s 0%: 11+134+0.040 ms clock, 133+1.6/251/366+0.48 ms cpu, 911->913->596 MB, 912 MB goal, 12 P
gc 6 @26.326s 0%: 20+210+0.021 ms clock, 250+1.9/353/476+0.25 ms cpu, 1149->1153->716 MB, 1192 MB goal, 12 P
```

GC开始 GC结束 Live堆 Goal堆

问：为何GC操作清理无用对象，但是GC结束时堆大小比GC开始时还大？

答：GC过程清理无用对象，但不一定会真正释放回操作系统

问：Live heap，Goal heap怎么理解？

答：- Live heap（活动堆）：进程中正在真正使用堆大小（排除无用对象和剩余堆空间）
- Goal heap（目标堆）：进程中从操作系统真正申请的堆内存大小，（保持足够的可用堆空间来处理大多数分配，而不必频繁从OS请求更多内存）

分析工具：go tool pprof

两种方式：

1. runtime/pprof （通常用于工具分析）
2. net/http/pprof（通常用于常驻服务进程分析）

服务端分析使用流程：

1. 如右图，导入net/http/pprof，启动http服务
2. 获取profile文件并启动web服务在浏览器段分析观察。

```
import (
  "net/http"
  _ "net/http/pprof"
)

func init() {
  go func() {
    log.Fatal(http.ListenAndServe("127.0.0.1:6060", nil))
  }()
}

func main() {
  log.Fatal(transport.EchoTCP(":10003", func(conn net.Conn) {
    io.Copy(conn, conn)
  }))
}
```

导入 net/http/pprof

启动http服务

CPU分析：go tool pprof -http=:32000 http://127.0.0.1:6060/debug/pprof/profile?second=10s

内存分析：go tool pprof -http=:32000 http://127.0.0.1:6060/debug/pprof/heap

pprof: Top

pprof VIEW SAMPLE REFINE Search regexp

Flat	Flat%	Sum%	Cum	Cum%	Name
20s	48.73%	48.73%	20.05s	48.85%	syscall.syscall
5.07s	12.35%	61.09%	5.07s	12.35%	runtime.pthread_cond_wait
4.02s	9.80%	70.88%	4.03s	9.82%	runtime.kevent
2.77s	6.75%	77.63%	2.77s	6.75%	runtime.nanotime
2.71s	6.60%	84.23%	2.71s	6.60%	runtime.pthread_cond_signal
2.20s	5.36%	89.60%	22.34s	54.43%	main.main.func1
1.49s	3.63%	93.23%	1.50s	3.65%	runtime.usleep
1.12s	2.73%	95.96%	5.16s	12.57%	runtime.netpoll
0.69s	1.68%	97.64%	0.69s	1.68%	runtime.pthread_cond_timedwait_relative_np
0.39s	0.95%	98.59%	1.08s	2.63%	runtime.notetsleep
0.05s	0.12%	98.71%	13.92s	33.92%	runtime.findrunnable
0.02s	0.05%	98.76%	4.07s	9.92%	runtime.sysmon
0.01s	0.02%	98.78%	5.09s	12.40%	runtime.stopm
0.01s	0.02%	98.81%	5.77s	14.06%	runtime.semasleep
0.01s	0.02%	98.83%	14.18s	34.55%	runtime.schedule
0.01s	0.02%	98.85%	14.19s	34.58%	runtime.park_m
0.01s	0.02%	98.88%	3.94s	9.60%	internal/poll.(*FD).Write
0.01s	0.02%	98.90%	16.20s	39.47%	internal/poll.(*FD).Read

- Flat：函数自身运行耗时
- Flat%：函数自身耗时比例
- Sum%：累计耗时比例
- Cum：函数自身+其调用函数耗时
- Cum%：函数自身+其调用函数比例
- Name：函数名

分析得到：48.73%消耗在syscall.syscall

但缺乏源码相关，调用关系信息。

GoCN
Go中国社区 GOCN.VIP

pprof: source

pprof VIEW SAMPLE REFINE Search regexp

syscall.syscall

/usr/local/Cellar/go/1.13.5/libexec/src/runtime/sys_darwin.go

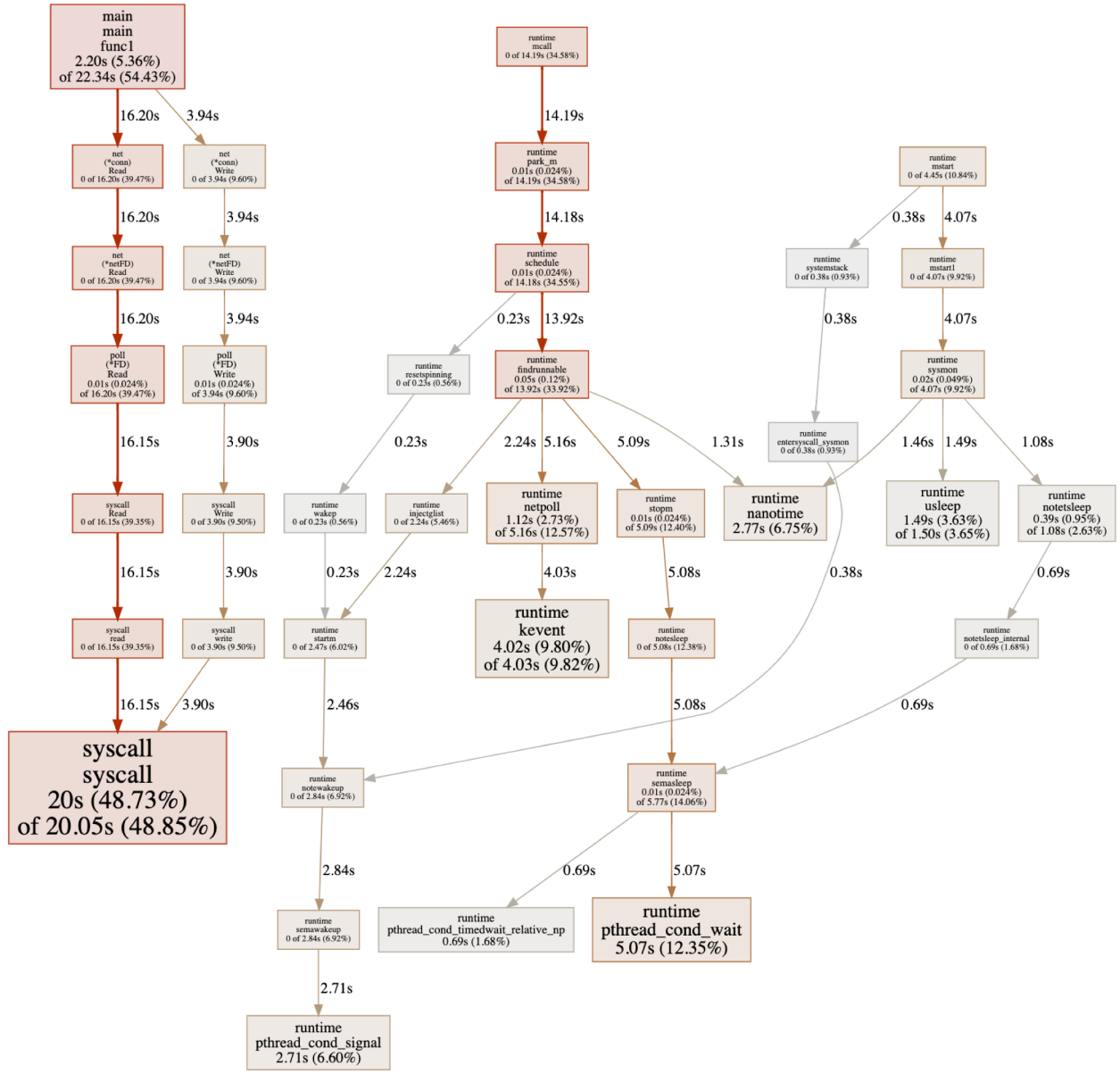
```
Total:          20s     20.05s (flat, cum) 48.85%
   58            .          .
   59            .          .           //go:linkname syscall_syscall syscall.syscall
   60            .          .           //go:nosplit
   61            .          .           //go:cgo_unsafe_args
   62            .          .           func syscall_syscall(fn, a1, a2, a3 uintptr) (r1, r2, err uintptr) {
   63            .       30ms                entersyscall()
   64            .       20ms                libcCall(unsafe.Pointer(funcPC(syscall)), unsafe.Pointer(&fn))
   65          20s        20s                exitsyscall()
   66            .          .                return
   67            .          .           }
   68            .          .           func syscall()
   69            .          .
   70            .          .           //go:linkname syscall_syscall6 syscall.syscall6
```

pprof: graph

Graph视图呈现程序函数调用树，

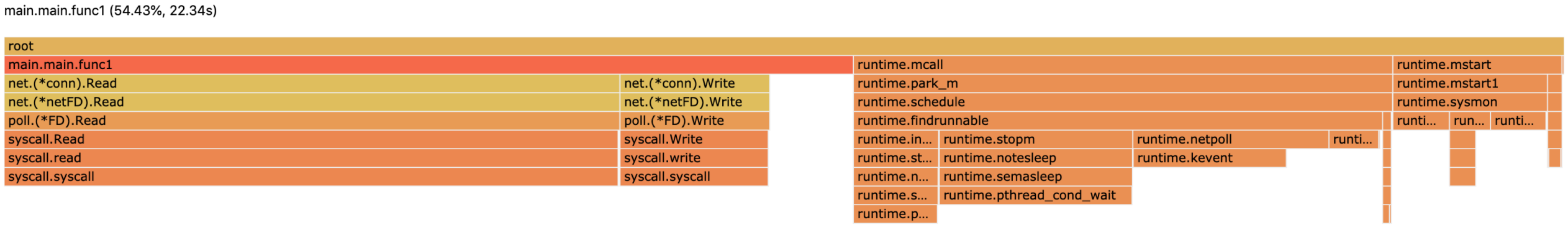
十分方便跟踪定位业务消耗源头。

方框越大代表函数自身消耗越大。



GoCN
Go中国社区 GOCN.VIP

pprof: 火焰图



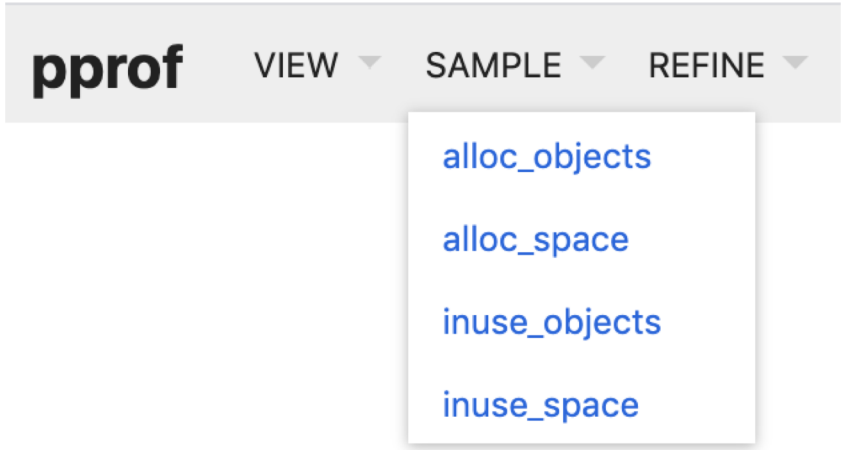
x 轴显示的是在该性能指标分析中所**占用的资源量**，也就是横向越宽，则意味着在该指标中占用的资源越多。

y 轴表示调用栈，每一层都是一个函数。调用栈越深，火焰就越高（深），底部就是正在执行的函数，上方都是它的父函数。

火焰图最底部的边表示被占用的CPU时间，宽度越大占比越大。通常从底部最宽边向上观察父函数，直到你的业务代码函数。

pprof：内存分析

可视化分配的字节数或分配的对象数



alloc_objects:收集自程序启动以来，累计的分配对象数

alloc_space: 收集自程序启动以来，累计的分配空间

inuse_objects:收集统计实时，正在使用的分配对象数

inuse_space: 收集统计实时，正在使用的分配空间

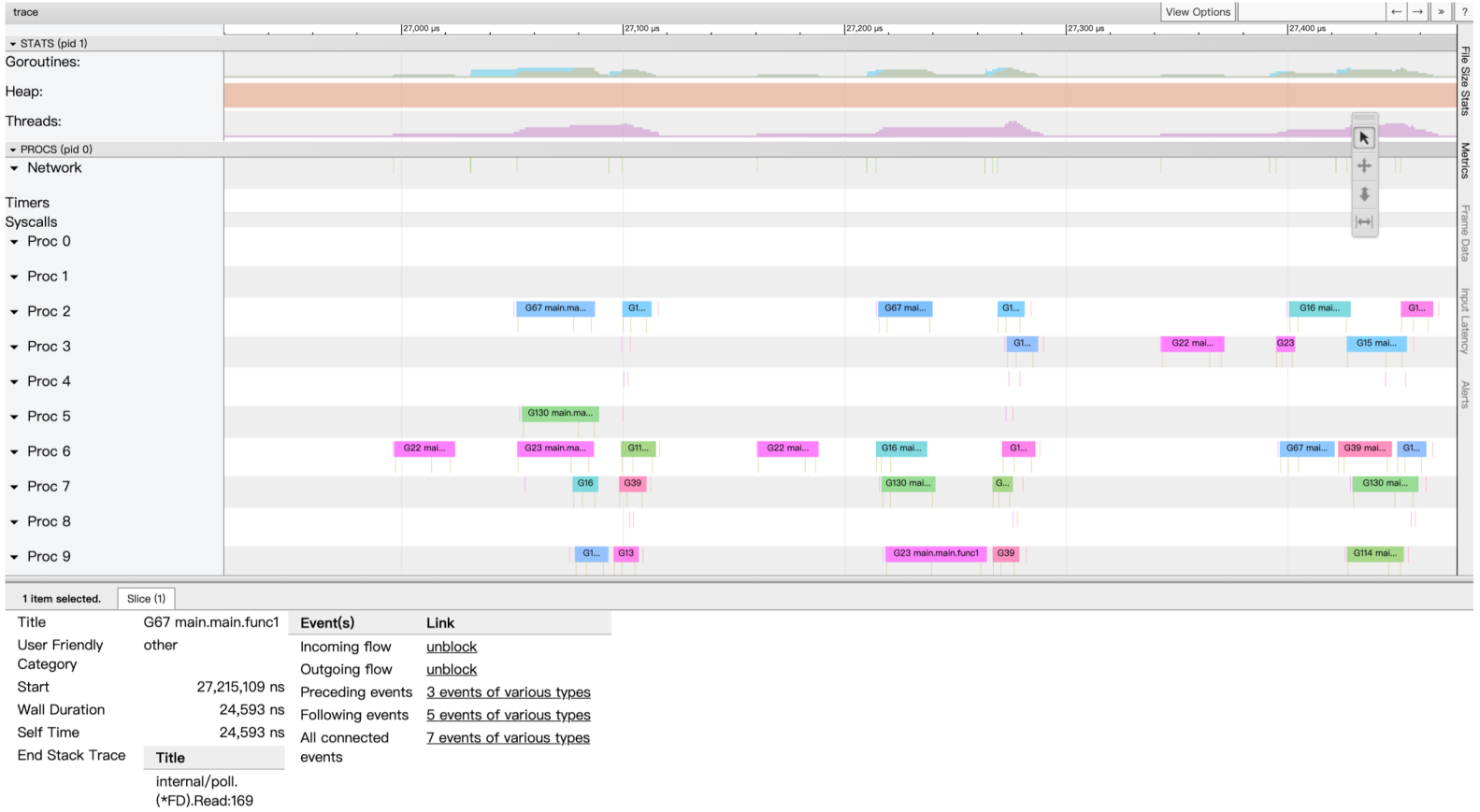
如果**要减少内存消耗**，则需要查看--inuse_space 在正常程序运行期间收集的配置文件。

如果**要提高执行速度**，请查看--alloc_objects大量运行时间后或程序结束时收集的概要文件。

分析工具：go tool trace

Trace分析：用来分析程序动态执行的情况，并且分析的精度达到纳秒级别

1. curl –o trace.out http://127.0.0.1:5050/debug/pprof/trace?seconds=2
2. go tool trace trace.out



3.优化

根据分析，修改系统的该部分实现以优化消除瓶颈。

少做些。少做一次。更快地做。

只有如上三种优化，最大的收益来自1，但我们将所有时间都花在3上 。

——Michael Fromberger

不要去做：只为你使用的东西付费，而不是你可以使用的东西

4.校验

从新测量基准测试并进行对比，需要使用benchstat或等效的统计测试，
而不能只是用眼睛去看

安装benchstat：go get golang.org/x/perf/cmd/benchstat

使用benchstat：第一版使用fmt.Sprint，第二版本优化后使用strconv.Itoa

1. 统计10次第一版性能：go test -bench=MyItoa -count=10 | tee version1.txt
2. 统计10次第二版性能：go test -bench=MyItoa -count=10 | tee version2.txt
3. 对比版本一二： benchstat version1.txt version2.txt

```
$ benchstat version1.txt version2.txt
name        old time/op  new time/op  delta
MyItoa-12   80.2ns ± 1%  30.4ns ± 3%  -62.07%  (p=0.000 n=10+10)
```

性能损耗降低 62%

GoCN
Go中国社区 GOCN.VIP

Go性能优化实践

GoCN
Go中国社区 GOCN.VIP

使用sync.Pool复用对象

本质：定期进行GC处理的用户定义的对象列表。

原理：复用已分配对象，减少分配数量，降低GC压力。

注意⚠：必须**重置被复用对象**，将对象从池中放回前或者取出后。避免使用到脏数据。

注意⚠：保证**使用后放回池**中，与任何手动内存管理方案一样，对sync.Pool的错误使用可能会导致使用后释放错误。重点注意**变量逃逸**后的释放操作。

典型示例:

利用sync.Pool实现接受UDP请求数据buf缓冲区，**避免[]byte的频繁分配与释放。**

在实际开发中，buf缓冲区可能会被多个函数应用，必须要注意Get和Put的一一对应。

udpPool自身可以作为全局变量，更好的方式是实现为Server中的成员变量。

```go
var udpPool = sync.Pool{New: func() interface{} {
  return make([]byte, defaultUDPBufferSize)
}}

func EchoUDP(address string) error {
  for {
    buf := udpPool.Get().([]byte)
    num, addr, err := u.ReadFrom(buf)
    if err != nil {
      udpPool.Put(buf[:defaultUDPBufferSize])
      if netErr, ok := err.(net.Error); ok && netErr.Temporary() {
        continue
      }
      return err
    }
    go handleUDP(u, buf[:num], addr)
  }
}

func handleUDP(u *net.UDPConn, buf []byte, addr net.Addr) {
  _, err := u.WriteTo(buf, addr)
  if err != nil {
  }
  udpPool.Put(buf[:defaultUDPBufferSize])
  return
}
```

GoCN
Go中国社区 GOCN.VIP

使用成员变量复用对象

典型示例:

TCP服务端，将每个buf缓冲区和TCP Conn绑定，每次此Conn读取数据均复用此buf

1. 这种复用比sync.Pool更契合具体业务场景，性能也是最高
2. 当客户端以**短连接请求业务，复用效果将不再存在。**

```
type framer struct{}

// ReadFrame 从io reader拆分出完整数据桢
func (f *framer) ReadFrame(reader io.Reader) (msgbuf []byte, err error) {
    head := make([]byte, frameHeadLen)
    _, err = io.ReadFull(reader, head[:frameHeadLen])
    totalLen := binary.BigEndian.Uint32(head[4:8])
    msg := make([]byte, totalLen)
    copy(msg, head[:])
    _, err = io.ReadFull(reader, msg[frameHeadLen:totalLen])
    return msg[:totalLen], err: nil
}
```

```
type framer struct {
    reader io.Reader
    head   [16]byte
    msg    []byte
}

// ReadFrame 从io reader拆分出完整数据桢
func (f *framer) ReadFrame() (msgbuf []byte, err error) {
    var num int
    num, err = io.ReadFull(f.reader, f.head[:frameHeadLen])
    totalLen := binary.BigEndian.Uint32(f.head[4:8])
    if int(totalLen) > len(f.msg) {
        f.msg = make([]byte, totalLen)
    }
    copy(f.msg, f.head[:])
    num, err = io.ReadFull(f.reader, f.msg[frameHeadLen:totalLen])
    return f.msg[:totalLen], err: nil
}
```

写时复制代替互斥锁

应用场景：受保护的数据不会经常被修改，并且可以对其进行复制。

实现：使用atomic.Value保证数据的加载和存储操作原子性

注意⚠：指针赋值是原子的吗？不是

注意⚠：slice数据类型多Goroutine并发读写会导致程序异常吗？会，只是不那么容易发生。

写时复制代替互斥锁

典型场景:

服务或SDK配置信息不会经常被修改，并且可以对其进行复制。

```
// SvrInfo SDK配置信息
type SvrInfo struct {
  SvrInfoType  SvrInfoType
  FrameSvrInfo FrameSvrInfo
  CommSvrInfo  CommSvrInfo
  PolarisInfo  PolarisInfo

  // 如下字段会定期从远端配置系统读取并更新
  // 读写锁保护如下字段
  mutex         sync.RWMutex
  ConfigVersion int
  PrefixInfo    PrefixInfo
  AttaInfo      ConfigInfo
}
```

```
// SvrInfo SDK配置信息
type SvrInfo struct {
  SvrInfoType SvrInfoType
  FrameSvrInfo *FrameSvrInfo
  CommSvrInfo *CommSvrInfo
  PolarisInfo *PolarisInfo

  Config atomic.Value // RemoteConfig
}
```

配置信息**动静分离**，
聚合动态字段并使用atomic.Value进行保护

分区：减少共享数据结构争用

原理：减少加锁粒度

```
// Partition 分区数据结构，使用读写锁保护本段数据
type Partition struct {
    sync.RWMutex
    m map[string]string
}

const partCount = 64  // 分区个数
var m [partCount]Partition // 整体受保护的数据"m"

func Find(k string) string {
    idx := hash(k) % partCount  // hash函数寻址分区索引
    part := &m[idx]
    part.RLock()  // 加锁保护本段数据
    v := part.m[k]
    part.RUnlock() // 释放锁
    return v
}
```

避免包含指针结构体作为map的key

原理：在垃圾回收期间，运行时扫描包含指针的对象，并对其进行追踪。

优化方案：在生产用例中，您需要在插入map之前将字符串散列为整数。

```
var pointers = map[string]int{}

func main() {
  for i := 0; i < 10000000; i++ {
    pointers[strconv.Itoa(i)] = i
  }

  for {
    timeGC()
    time.Sleep(1 * time.Second)
  }
}

// $
gc took: 84.462016ms
gc took: 83.274798ms
gc took: 84.366206ms
gc took: 84.640187ms
```

```
type Entity struct {
  A int
  B float64
}
var entities = map[Entity]int{}

func main() {
  for i := 0; i < 10000000; i++ {
    entities[Entity{
      A: i,
      B: float64(i),
    }] = i
  }
  for {
    timeGC()
    time.Sleep(1 * time.Second)
  }
}

// $
gc took: 4.539228ms
gc took: 3.037675ms
gc took: 2.670415ms
gc took: 3.349965ms
```

GoCN
Go中国社区 GOCN.VIP

使用 strings.Builder 拼接字符串

原理：写入字节缓冲区，仅在调用String()构建器时，才实际创建字符串。

```go
func BuildStrRaw(strs []string) string {
  var s string

  for _, v := range strs {
    s += v
  }

  return s
}
```

```go
func BuildStrBuilder(strs []string) string {
  b := strings.Builder{}
  b.Grow(128)  //预分配128字节

  for _, v := range strs {
    b.WriteString(v)
  }

  return b.String()
}
```

```
goos: darwin
goarch: amd64
pkg: git.code.oa.com/tensorchen/go-performance/examples/strings
BenchmarkStringBuildRaw-12              3777170               297 ns/op             216 B/op          8 allocs/op
BenchmarkStringBuildBuilder-12         15131961                71.3 ns/op           128 B/op          1 allocs/op
PASS
ok      git.code.oa.com/tensorchen/go-performance/examples/strings      2.615s
```

其它

1. 使用strconv代替fmt
2. 避免[]byte和string的转换
3. 利用代码生成避免反射
4. 多读少写，使用sync.RWMutex代替sync.Mutex

总结

1. 优化也是要付出人力等成本，明确何时停止性能优化
2. 优先使用CPU Profile，当发现定位困难时，尝试使用MEM Profile。
3. 实践出真知，快速实验
4. 具体问题具体分析结合实际场景，进行特定的优化。

Thanks