

golang pprof 实战

🕒 2019/04/02. (https://blog.wolfogre.com/redirect/v3/AwZUdKDU24rmS261g_2g_7oSAwM8Cv46xcU7LxImWv3FQQYW3DshxVoWBjvFQi_FrUobxXVQYcU8BkoKxTsGzDw8Bcw8ghxKiG4tGDESAwM8Cv46xcVaFgY7bkEGFtw7If3FPAZNCsU7Bsw8PAXMPIIcSojF) 🛡 Golang (/tags/golang) 🕒 23.3k+ 🎵 21

目录

- ◉ 前言
- 实验准备
- 获取“炸弹”
- 使用 pprof
- 排查 CPU 占用过高
- 排查内存占用过高
- 排查频繁内存回收
- 排查协程泄露
- 排查锁的争用
- 排查阻塞操作
- 思考题
- 最后

前言

如果说在 golang 开发过程进行性能调优，pprof 一定是一个大杀器般的工具。但在网上找到的教程都偏向简略，难寻真的能应用于实战的教程。这也无可厚非，毕竟 pprof 是当程序占用资源异常时才需要启用的工具，而我相信大家的编码水平和排场问题的能力是足够高的，一般不会写出性能极度堪忧的程序，且即使发现有一些资源异常占用，也会通过排查代码快速定位，这也导致 pprof 需要上战场的机会少之又少。即使大家有心想学习使用 pprof，却也常常相忘于江湖。

既然如此，那我就送大家一个性能极度堪忧的“炸弹”程序吧！

这程序没啥正经用途缺极度占用资源，基本覆盖了常见的性能问题。本文就是希望读者能一步一步按照提示，使用 pprof 定位这个程序的性能瓶颈所在，借此学习 pprof 工具的使用方法。

因此，本文是一场“实验课”而非“理论课”，请读者腾出时间，脚踏实地，一步一步随实验步骤进行操作，这会是一个很有趣的冒险，不会很无聊，希望你能喜欢。

实验准备

这里假设你有基本的 golang 开发功底，拥有 golang 开发环境并配置了 \$GOPATH，能熟练阅读简单的代码或进行简单的修改，且知道如何编译运行 golang 程序。此外，需要你大致知道 p prof 是干什么的，有一个基本印象即可，你可以花几分钟时间读一下《Golang 大杀器之性能剖析 PProf》 (https://blog.wolfogre.com/redirect/v3/AouO_9voECGYMC6hksfOB8YSAwM8Cv46x cU7LxImWv3FQhhTHP5qU8VaFgY7xVoWBlrFrUobxXVQYcU8BkoKxTsGxRQGFgrF_wQyMD E4zP8CMDnM_wIxNcw7BswUBhbMPDxzLG4tGDESAwM8Cv46xcVaFgY7bkEGFtw7If3FPaz NCsU7Bsw8PAXMPIIcSojF) 的开头部分，这不会耽误太久。

此外由于你需要运行一个“炸弹”程序，请务必确保你用于做实验的机器有充足的资源，你的机器至少需要：

- 2 核 CPU；
- 2G 内存。

注意，以上只是最低需求，你的机器配置能高于上述要求自然最好。实际运行“炸弹”时，你可以关闭电脑上其他不必要的程序，甚至 IDE 都不用开，我们的实验操作基本上是在命令行里进行的。

此外，务必确保你是在个人机器上运行“炸弹”的，能接受机器死机重启的后果（虽然这发生的概率很低）。请你务必不要在危险的边缘试探，比如在线上服务器运行这个程序。

可能说得你都有点害怕了，为打消你顾虑，我可以剧透一下“炸弹”的情况，让你安心：

- 程序会占用约 2G 内存；
- 程序占用 CPU 最高约 100%（总量按“核数 * 100%”来算）；
- 程序不涉及网络或文件读写；
- 程序除了吃资源之外没有其他危险操作。

且程序所占用的各类资源，均不会随着运行时间的增长而增长，换句话说，只要你把“炸弹”启动并正常运行了一分钟，就基本确认安全了，之后即使运行几天也不会有更多的资源占用，除了有点费电之外。

获取“炸弹”

炸弹程序的代码我已经放到了 GitHub (https://blog.wolfogre.com/redirect/v3/A_4-v86v-9Btg9a9FuRKCCgSAwM8Cv46xcU7LxImWv3FQQYW3DshxTsGzDw8cyzMPIIcSogxEgMDPAr-OsXFWhYGO25BBhbcOyH9xTwGTQrFOwbMPDwFzDyCHEqIxQ) 上，你只需要在终端里运行 `go get` 便可获取，注意加上 `-d` 参数，避免下载后自动安装：

```
go get -d github.com/wolfogre/go-pprof-practice
cd $GOPATH/src/github.com/wolfogre/go-pprof-practice
```

我们可以简单看一下 `main.go` 文件，里面有几个帮助排除性能调问题的关键的点，我加上了些注释方便你理解，如下：

```
package main

import (
    // 略
    _ "net/http/pprof" // 会自动注册 handler 到 http server, 方便通过 http 接口获取
    // 程序运行采样报告
    // 略
)

func main() {
    // 略

    runtime.GOMAXPROCS(1) // 限制 CPU 使用数, 避免过载
    runtime.SetMutexProfileFraction(1) // 开启对锁调用的跟踪
    runtime.SetBlockProfileRate(1) // 开启对阻塞操作的跟踪

    go func() {
        // 启动一个 http server, 注意 pprof 相关的 handler 已经自动注册过了
        if err := http.ListenAndServe(":6060", nil); err != nil {
            log.Fatal(err)
        }
        os.Exit(0)
    }()

    // 略
}
```

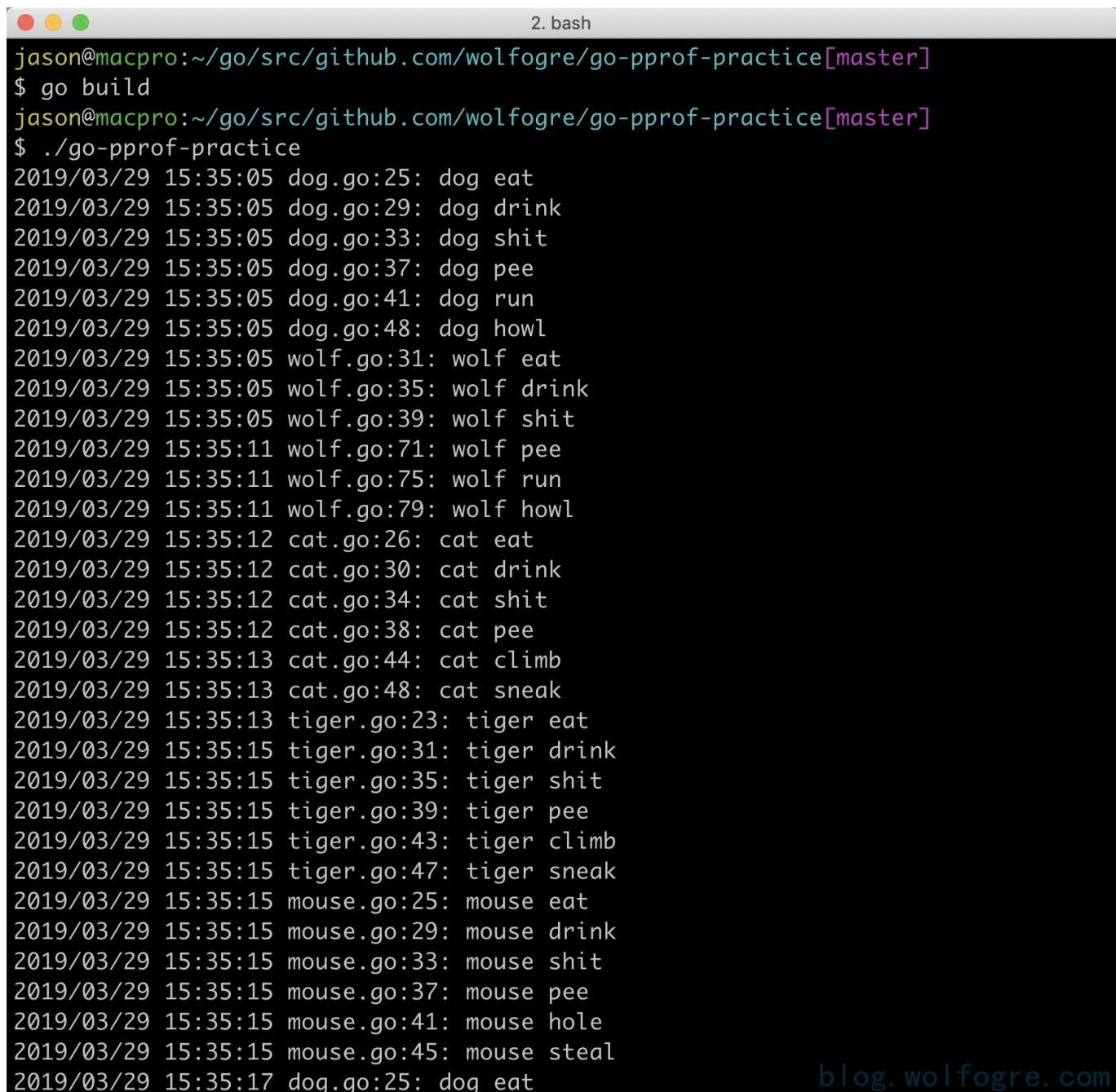
除此之外的其他代码你一律不用看，那些都是我为了模拟一个“逻辑复杂”的程序而编造的，其中大多数的问题很容易通过肉眼发现，但我们需要做的是通过 pprof 来定位代码的问题，所以为了保证实验的趣味性请不要提前阅读代码，可以实验完成后再看。

接着我们需要编译一下这个程序并运行，你不用担心依赖问题，这个程序没有任何外部依赖。

```
go build
./go-pprof-practice
```

运行后注意查看一下资源是否吃紧，机器是否还能扛得住，坚持一分钟，如果确认没问题，咱们再进行下一步。

控制台里应该会不停的打印日志，都是一些“猫狗虎鼠在不停地吃喝拉撒”的屁话，没有意义，不用细看。



```
jason@macpro:~/go/src/github.com/wolfogre/go-pprof-practice[master]
$ go build
jason@macpro:~/go/src/github.com/wolfogre/go-pprof-practice[master]
$ ./go-pprof-practice
2019/03/29 15:35:05 dog.go:25: dog eat
2019/03/29 15:35:05 dog.go:29: dog drink
2019/03/29 15:35:05 dog.go:33: dog shit
2019/03/29 15:35:05 dog.go:37: dog pee
2019/03/29 15:35:05 dog.go:41: dog run
2019/03/29 15:35:05 dog.go:48: dog howl
2019/03/29 15:35:05 wolf.go:31: wolf eat
2019/03/29 15:35:05 wolf.go:35: wolf drink
2019/03/29 15:35:05 wolf.go:39: wolf shit
2019/03/29 15:35:11 wolf.go:71: wolf pee
2019/03/29 15:35:11 wolf.go:75: wolf run
2019/03/29 15:35:11 wolf.go:79: wolf howl
2019/03/29 15:35:12 cat.go:26: cat eat
2019/03/29 15:35:12 cat.go:30: cat drink
2019/03/29 15:35:12 cat.go:34: cat shit
2019/03/29 15:35:12 cat.go:38: cat pee
2019/03/29 15:35:13 cat.go:44: cat climb
2019/03/29 15:35:13 cat.go:48: cat sneak
2019/03/29 15:35:13 tiger.go:23: tiger eat
2019/03/29 15:35:15 tiger.go:31: tiger drink
2019/03/29 15:35:15 tiger.go:35: tiger shit
2019/03/29 15:35:15 tiger.go:39: tiger pee
2019/03/29 15:35:15 tiger.go:43: tiger climb
2019/03/29 15:35:15 tiger.go:47: tiger sneak
2019/03/29 15:35:15 mouse.go:25: mouse eat
2019/03/29 15:35:15 mouse.go:29: mouse drink
2019/03/29 15:35:15 mouse.go:33: mouse shit
2019/03/29 15:35:15 mouse.go:37: mouse pee
2019/03/29 15:35:15 mouse.go:41: mouse hole
2019/03/29 15:35:15 mouse.go:45: mouse steal
2019/03/29 15:35:17 dog.go:25: dog eat
```

blog.wolfogre.com

使用 pprof

保持程序运行，打开浏览器访问 <http://localhost:6060/debug/pprof/>，可以看到如下页面：

/debug/pprof/

Types of profiles available:

Count Profile

Profile Type	Count
allocs	13
block	3
cmdline	0
goroutine	54
heap	13
mutex	1
profile	0
threadcreate	8
trace	0

[full goroutine stack dump](#)

Profile Descriptions:

- allocs: A sampling of all past memory allocations
- block: Stack traces that led to blocking on synchronization primitives
- cmdline: The command line invocation of the current program
- goroutine: Stack traces of all current goroutines
- heap: A sampling of memory allocations of live objects. You can specify the gc GET parameter to run GC before taking the heap sample.
- mutex: Stack traces of holders of contended mutexes
- profile: CPU profile. You can specify the duration in the seconds GET parameter. After you get the profile file, use the go tool pprof command to investigate the profile.
- threadcreate: Stack traces that led to the creation of new OS threads
- trace: A trace of execution of the current program. You can specify the duration in the seconds GET parameter. After you get the trace file, use the go tool trace command to investigate the trace.

blog.wolfogre.com

页面上展示了可用的程序运行采样数据，分别有：

类型	描述	备注
al	内存分配情况的采样信息	可以用浏览器打开，但可读性不高
lo		
cs		

类型	描述	备注
bl	阻塞	可以用浏览器打开，但可读性不高
oc	操作	
ks	情况的采样信息	
c	显示程序启动命令及参数	可以用浏览器打开，这里会显示 <code>./go-pprof-practice</code>
m		
dl		
in		
e		
g	当前所有协程的堆栈信息	可以用浏览器打开，但可读性不高
or		
o		
ut		
in		
e		
h	堆上内存使用情况的采样信息	可以用浏览器打开，但可读性不高
ea		
p		
m	锁争用情况的采样信息	可以用浏览器打开，但可读性不高
ut		
ex		
pr	CPU 占用情况的采样信息	浏览器打开会下载文件
of		
il		
e		

类型	描述	备注
th	系统	可以用浏览器打开，但可读性不高
re	线程	
a	创建	
d	情况	
cr	的采	
ea	样信	
ea	息	
tr	程序	浏览器打开会下载文件，本文不涉及，可另行参阅《深入浅出 Go trace》(https://blog.wolfogre.com/redirect/v3/AwBGKjtUXC4lQ2UdNqHTCoMSAwM8Cv46xcUtPG6RCPoPbv8CcXH9xQrF_wJJOfr_AlNN-IP_AjMyHP8IQUxTtlFBTjhBFgn-UTESAwM8Cv46xcVaFgY7bkEGFtw7If3FPANCsU7Bsw8PAXMPIIcSojF)
ac	运行	
e	跟踪信息	

因为 cmdline 没有什么实验价值，trace 与本文主题关系不大，threadcreate 涉及的情况偏复杂，所以这三个类型的采样信息这里暂且不提。除此之外，其他所有类型的采样信息本文都会涉及到，且炸弹程序已经为每一种类型的采样信息埋藏了一个对应的性能问题，等待你的发现。

由于直接阅读采样信息缺乏直观性，我们需要借助 `go tool pprof` 命令来排查问题，这个命令是 go 原生自带的，所以不用额外安装。

我们先不用完整地学习如何使用这个命令，毕竟那太枯燥了，我们一边实战一边学习。

以下正式开始。

排查 CPU 占用过高

我们首先通过活动监视器（或任务管理器、`top` 命令，取决于你的操作系统和你的喜好），查看一下炸弹程序的 CPU 占用：



可以看到 CPU 占用相当高，这显然是有问题的，我们使用 `go tool pprof` 来排场一下：

```
go tool pprof http://localhost:6060/debug/pprof/profile
```

等待一会儿后，进入一个交互式终端：

```
$ go tool pprof http://localhost:6060/debug/pprof/profile
Fetching profile over HTTP from http://localhost:6060/debug/pprof/profile
Saved profile in /Users/jason/pprof/pprof.samples.cpu.011.pb.gz
Type: cpu
Time: Mar 29, 2019 at 5:30pm (CST)
Duration: 30s, Total samples = 12.47s (41.56%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) |
```

blog.wolfogre.com

输入 `top` 命令，查看 CPU 占用较高的调用：

```
(pprof) top
Showing nodes accounting for 12.37s, 99.20% of 12.47s total
Dropped 19 nodes (cum <= 0.06s)
Showing top 10 nodes out of 17
      flat  flat%  sum%      cum  cum%
11.13s 89.25% 89.25%  11.13s 89.25%  github.com/wolfogre/go-pprof-practic
e/animal/felidae/tiger.(*Tiger).Eat
  0.83s  6.66% 95.91%   0.83s  6.66%  runtime.nanotime
  0.40s  3.21% 99.12%   0.40s  3.21%  runtime.memclrNoHeapPointers
  0.01s  0.08% 99.20%   0.44s  3.53%  runtime.systemstack
  0     0% 99.20%   0.41s  3.29%  github.com/wolfogre/go-pprof-practic
e/animal/canidae/dog.(*Dog).Live
  0     0% 99.20%   0.41s  3.29%  github.com/wolfogre/go-pprof-practic
e/animal/canidae/dog.(*Dog).Run
  0     0% 99.20%   11.13s 89.25%  github.com/wolfogre/go-pprof-practic
e/animal/felidae/tiger.(*Tiger).Live
  0     0% 99.20%   11.54s 92.54%  main.main
  0     0% 99.20%   0.40s  3.21%  runtime.(*mheap).alloc
  0     0% 99.20%   0.41s  3.29%  runtime.largeAlloc
(pprof) |   blog.wolfogre.com
```

很明显，CPU 占用过高是 `github.com/wolfogre/go-pprof-practice/animal/felidae/tiger.(*Tiger).Eat` 造成的。

注：为了保证实验节奏，关于图中 `flat`、`flat%`、`sum%`、`cum`、`cum%` 等参数的含义这里就不展开讲了，你可以先简单理解为数字越大占用情况越严重，之后可以在《Golang 大杀器之性能剖析 PProf》(https://blog.wolfogre.com/redirect/v3/A3jsjsvor3pCsn4x_qmqFKwSAwM8Cv46xcU7LxImWv3F_wdFRERZQopZxVoWBjvFWyYGWsWtTRvFOwaJVMX_BDIwMTjM_wIwOcz_AjE1zP5HB0lU_1AlMjAlRTUlQTQlQTclRTYlOUQlODAlRTUlOTklQTglRTQlQjkLOEIlRTYlODAlQTclRTglODMlQkQlRTUlODklOTYlRTYlOUUlOTAlMjBQUHMsbioYMRIDAzwK_jrFxVoWBjtuQQYW3Dsh_cU8BkoKxTsGzDw8Bcw8ghxKiMU)等其他资料中深入学习。

输入 `list Eat`，查看问题具体在代码的哪一个位置：

```
(pprof) list Eat
Total: 12.47s
ROUTINE ===== github.com/wolfogre/go-pprof-practice/animal/felidae/tiger.(*Tiger).Eat in /Users/jason/go/src/github.com/wolfogre/go-pprof-practice/animal/felidae/tiger/tiger.go
  11.13s    11.13s (flat, cum) 89.25% of Total
      .          .   19:{}
      .          .   20:
      .          .   21:func (t *Tiger) Eat() {
      .          .   22:    log.Println(t.Name(), "eat")
      .          .   23:    loop := 10000000000
  11.13s    11.13s   24:    for i := 0; i < loop; i++ {
      .          .   25:        // do nothing
      .          .   26:    }
      .          .   27:}
      .          .   28:
      .          .   29:func (t *Tiger) Drink() {
(pprof)                                     blog.wolfogre.com
```

可以看到，是第 24 行那个一百亿次空循环占用了大量 CPU 时间，至此，问题定位成功！

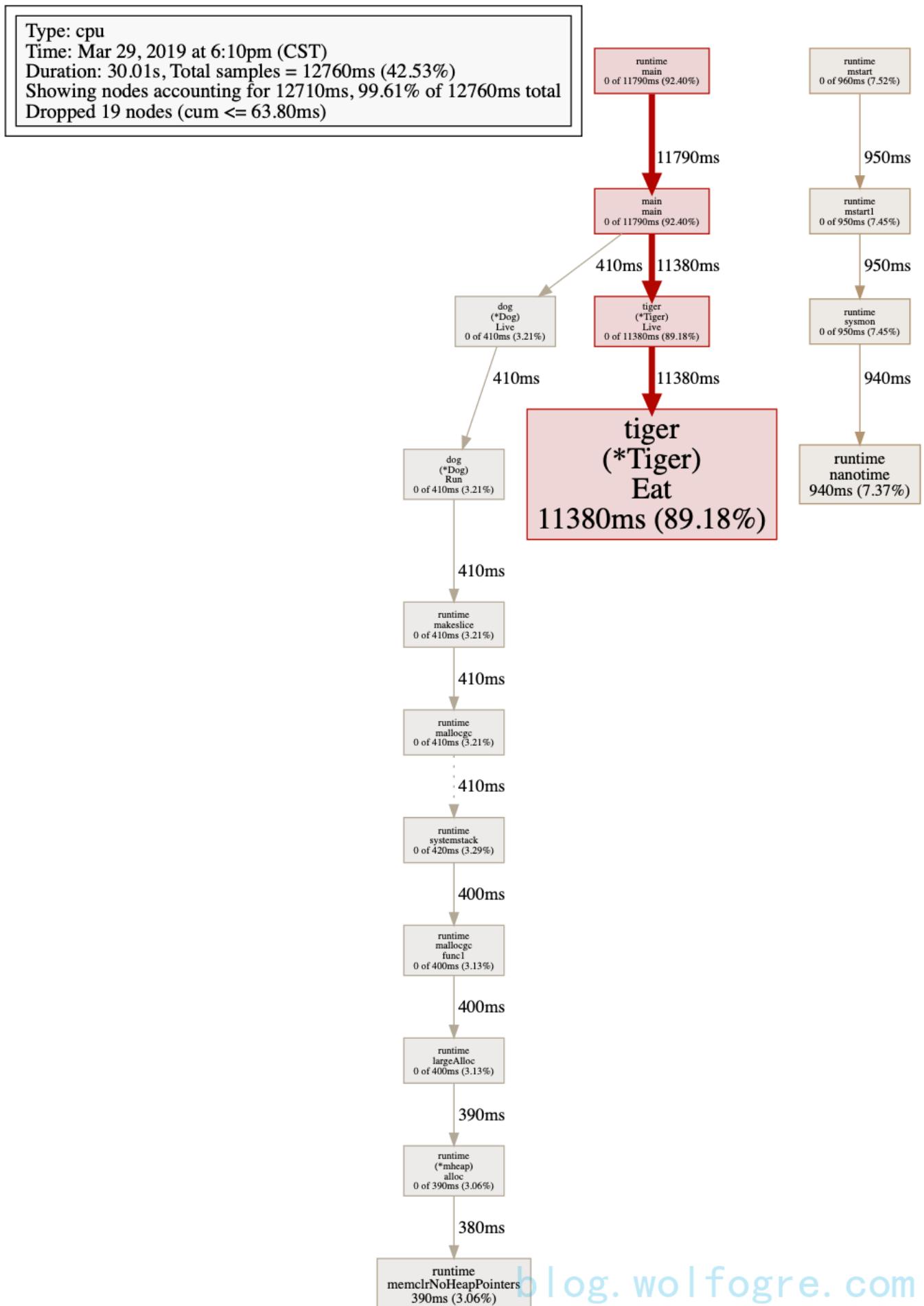
接下来有一个扩展操作：图形化显示调用栈信息，这很酷，但是需要你事先在机器上安装 graphviz，大多数系统上可以轻松安装它：

```
brew install graphviz # for macos
apt install graphviz # for ubuntu
yum install graphviz # for centos
```

或者你也可以访问 graphviz 官网 (https://blog.wolfogre.com/redirect/v3/A421Yoc_xEV4G_G_UO8tV1nMSAwM8Cv46xeU7gjwSbQjbbjsviVpukMUYBkEJFgboxTESAwM8Cv46xcVaFgY7bkEGFtw7If3FPazNCsU7Bsw8PAXMPIIcSojF) 寻找适合自己操作系统的安装方法。

安装完成后，我们继续在上文的交互式终端里输入 web，注意，虽然这个命令的名字叫“web”，但它的实际行为是产生一个 .svg 文件，并调用你的系统里设置的默认打开 .svg 的程序打开它。如果你的系统里打开 .svg 的默认程序并不是浏览器（比如可能是你的代码编辑器），这时候你需要设置一下默认使用浏览器打开 .svg 文件，相信这难不倒你。

你应该可以看到：



图中，`tiger.(*Tiger).Eat` 函数的框特别大，箭头特别粗，pprof 生怕你不知道这个函数的 CPU 占用很高，这张图还包含了很多有趣且有价值的信息，你可以多看一会儿再继续。

至此，这一小节使用 pprof 定位 CPU 占用的实验就结束了，你需要输入 `exit` 退出 pprof 的交互式终端。

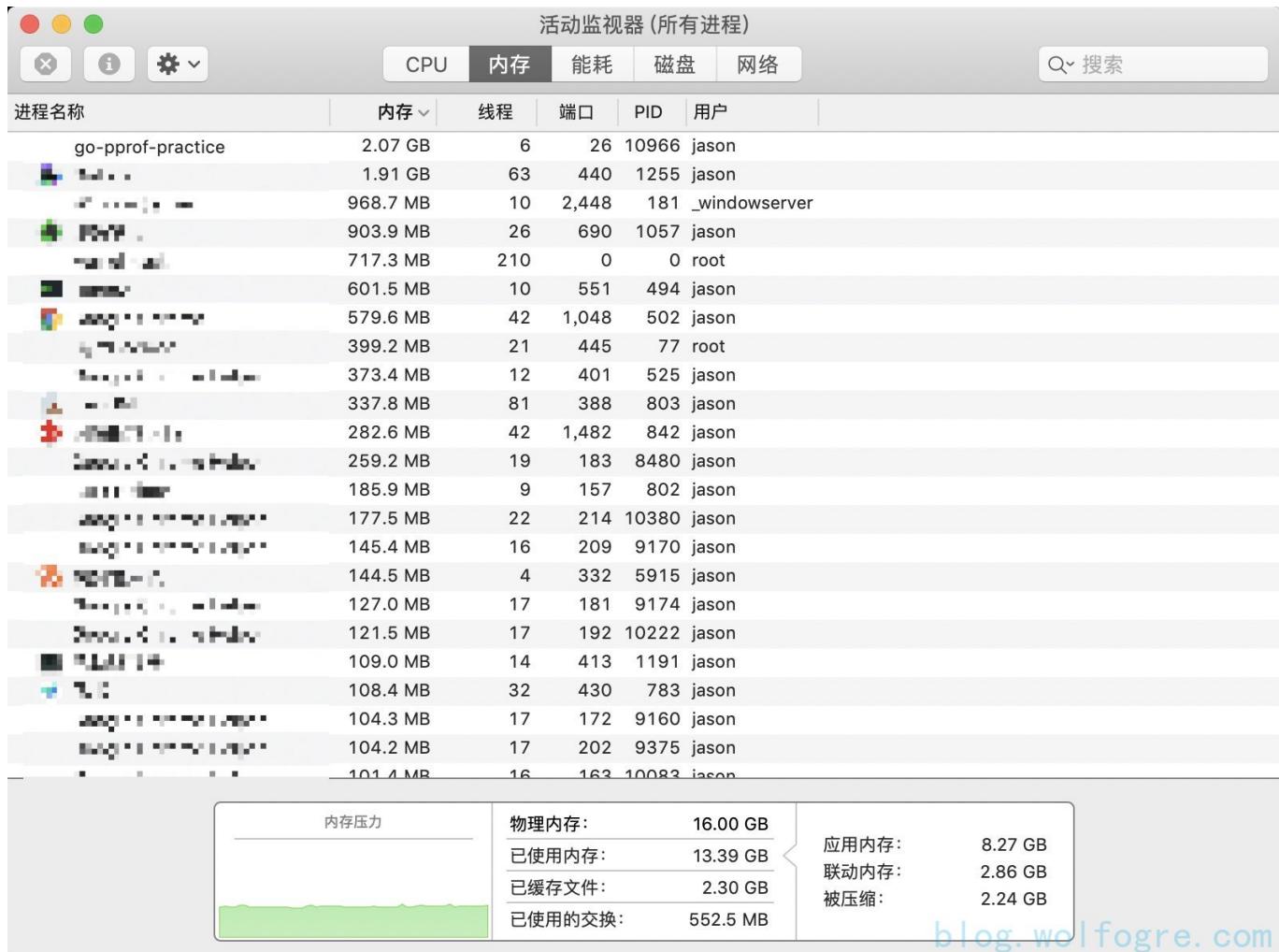
为了方便进行后面的实验，我们修复一下这个问题，不用太麻烦，注释掉相关代码即可：

```
func (t *Tiger) Eat() {
    log.Println(t.Name(), "eat")
    //loop := 100000000000
    //for i := 0; i < loop; i++ {
    //    // do nothing
    //}
}
```

之后修复问题的方法都是注释掉相关的代码，不再赘述。你可能觉得这很粗暴，但要知道，这个实验的重点是如何使用 pprof 定位问题，我们不需要花太多时间在改代码上。

排查内存占用过高

重新编译炸弹程序，再次运行，可以看到 CPU 占用率已经下来了，但是内存的占用率仍然很高：



我们再次运行使用 pprof 命令，注意这次使用的 URL 的结尾是 `heap`：

```
go tool pprof http://localhost:6060/debug/pprof/heap
```

再一次使用 `top`、`list` 来定位问题代码：

```
$ go tool pprof http://localhost:6060/debug/pprof/heap
Fetching profile over HTTP from http://localhost:6060/debug/pprof/heap
Saved profile in /Users/jason/pprof/pprof.alloc_objects.alloc_space.inuse_objects.inuse_space.004.pb.gz
Type: inuse_space
Time: Apr 1, 2019 at 7:35pm (CST)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 1GB, 100% of 1GB total
      flat  flat%  sum%      cum   cum%
      1GB  100%  100%      1GB  100%  github.com/wolfogre/go-pprof-practice/animal/muridae/mouse.(*Mouse).Steal
          0    0%  100%      1GB  100%  github.com/wolfogre/go-pprof-practice/animal/muridae/mouse.(*Mouse).Live
          0    0%  100%      1GB  100%  main.main
          0    0%  100%      1GB  100%  runtime.main
(pprof) list Steal
Total: 1GB
ROUTINE ===== github.com/wolfogre/go-pprof-practice/animal/muridae/mouse.(*Mouse).Steal in /Users/jason/go/src/github.com/wolfogre/go-pprof-practice/animal/muridae/mouse/mouse.go
      1GB      1GB (flat, cum)  100% of Total
      .
      .      45:
      .
      .      46:func (m *Mouse) Steal() {
      .
      .      47:    log.Println(m.Name(), "steal")
      .
      .      48:    max := constant.Gi
      .
      .      49:    for len(m.buffer) * constant.Mi < max {
      1GB      1GB      50:        m.buffer = append(m.buffer, [constant.Mi]byte{})
      .
      .      51:    }
      .
      .      52:}

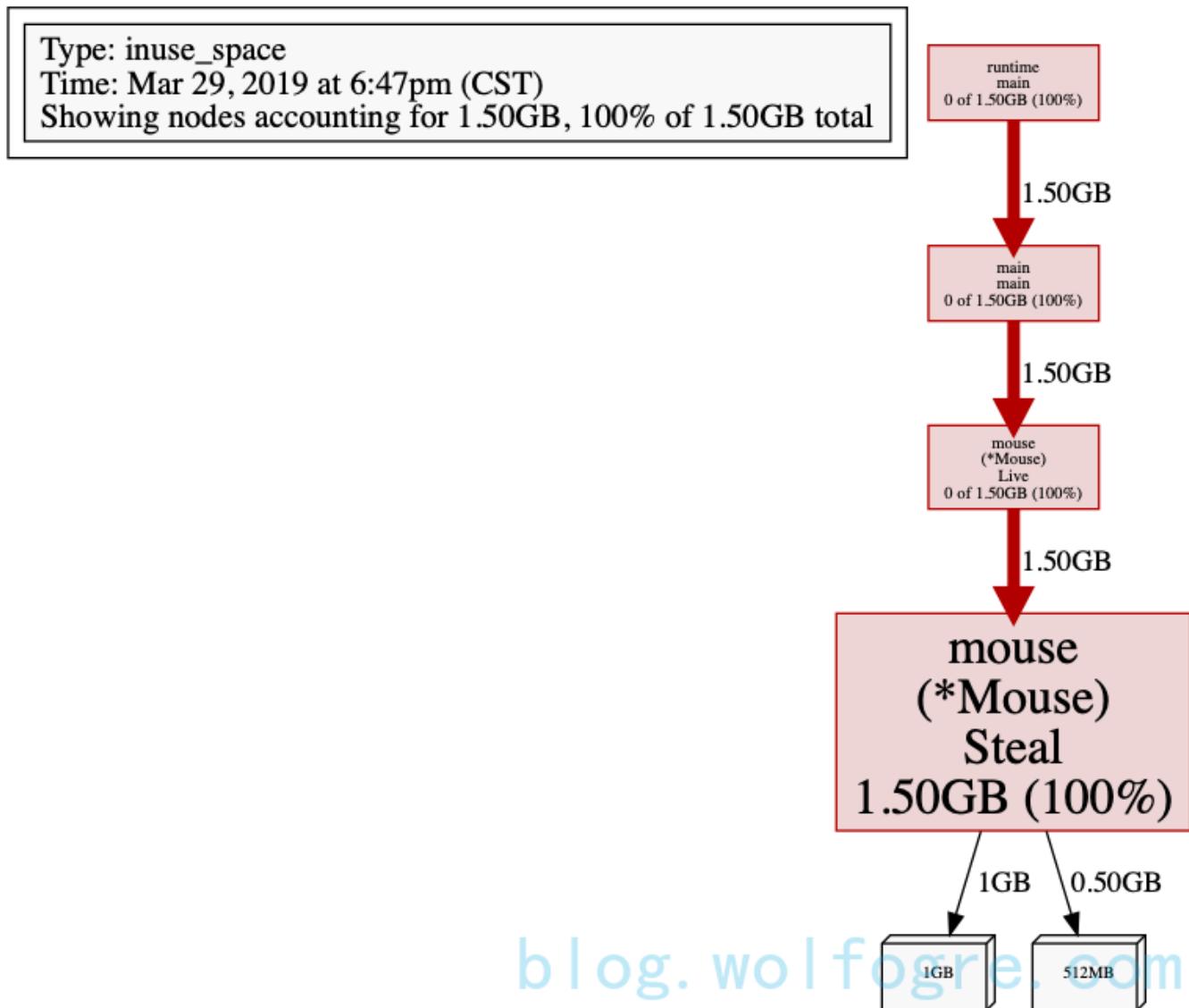
(pprof) blog.wolfogre.com
```

可以看到这次出问题的地方在 `github.com/wolfogre/go-pprof-practice/animal/muridae/mouse.(*Mouse).Steal`，函数内容如下：

```
func (m *Mouse) Steal() {
    log.Println(m.Name(), "steal")
    max := constant.Gi
    for len(m.buffer) * constant.Mi < max {
        m.buffer = append(m.buffer, [constant.Mi]byte{})
    }
}
```

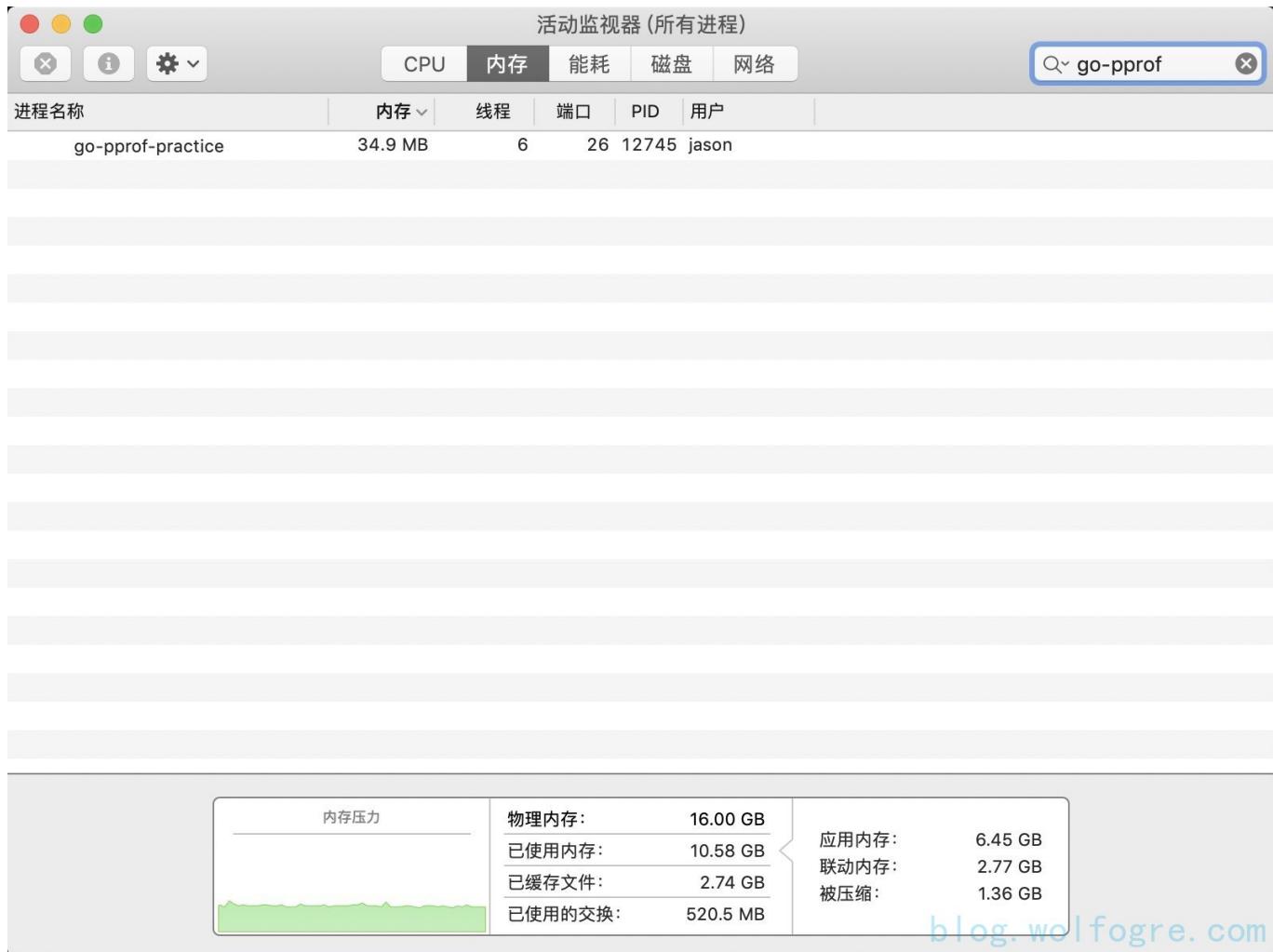
可以看到，这里有个循环会一直向 `m.buffer` 里追加长度为 1 MiB 的数组，直到总容量到达 1 GiB 为止，且一直不释放这些内存，这就难怪会有这么高的内存占用了。

使用 web 来查看图形化展示，可以再次确认问题确实出在这里：



现在我们同样是注释掉相关代码来解决这个问题。

再次编译运行，查看内存占用：



可以看到内存占用已经将到了 35 MB，似乎内存的使用已经恢复正常，一片祥和。

但是，内存相关的性能问题真的已经全部解决了吗？

排查频繁内存回收

你应该知道，频繁的 GC 对 golang 程序性能的影响也是非常严重的。虽然现在这个炸弹程序内存使用量并不高，但这会不会是频繁 GC 之后的假象呢？

为了获取程序运行过程中 GC 日志，我们需要先退出炸弹程序，再在重新启动前赋予一个环境变量，同时为了避免其他日志的干扰，使用 grep 筛选出 GC 日志查看：

```
GODEBUG=gctrace=1 ./go-pprof-practice | grep gc
```

日志输出如下：

```
jason@macpro:~/go/src/github.com/wolfogre/go-pprof-practice[debug]
$ GODEBUG=gctrace=1 ./go-pprof-practice | grep gc
gc 1 @0.002s 1%: 0.004+0.28+0.001 ms clock, 0.004+0.13/0.12/0+0.001 ms cpu, 16->
16->0 MB, 17 MB goal, 1 P
gc 2 @3.023s 0%: 0.007+0.31+0.002 ms clock, 0.007+0.11/0.15/0+0.002 ms cpu, 16->
16->0 MB, 17 MB goal, 1 P
gc 3 @6.035s 0%: 0.010+0.38+0.002 ms clock, 0.010+0.15/0.19/0+0.002 ms cpu, 16->
16->0 MB, 17 MB goal, 1 P
gc 4 @9.049s 0%: 0.010+0.46+0.002 ms clock, 0.010+0.17/0.21/0+0.002 ms cpu, 16->
16->0 MB, 17 MB goal, 1 P
gc 5 @12.068s 0%: 0.007+0.32+0.001 ms clock, 0.007+0.10/0.15/0+0.001 ms cpu, 16-
>16->0 MB, 17 MB goal, 1 P
gc 6 @15.078s 0%: 0.012+0.52+0.002 ms clock, 0.012+0.13/0.21/0+0.002 ms cpu, 16-
>16->0 MB, 17 MB goal, 1 P
gc 7 @18.089s 0%: 0.010+0.60+0.002 ms clock, 0.010+0.15/0.27/0+0.002 ms cpu, 16-
>16->0 MB, 17 MB goal, 1 P
gc 8 @21.103s 0%: 0.011+0.60+0.002 ms clock, 0.011+0.25/0.19/0+0.002 ms cpu, 16-
>16->0 MB, 17 MB goal, 1 P
gc 9 @24.109s 0%: 0.010+0.65+0.002 ms clock, 0.010+0.32/0.14/0+0.002 ms cpu, 16-
>16->0 MB, 17 MB goal, 1 P
gc 10 @27.125s 0%: 0.010+0.59+0.002 ms clock, 0.010+0.23/0.22/0+0.002 ms cpu, 16-
>16->0 MB, 17 MB goal, 1 P
gc 11 @30.140s 0%: 0.010+0.63+0.002 ms clock, 0.010+0.33/0.13/0+0.002 ms cpu, 16-
>16->0 MB, 17 MB goal, 1 P
gc 12 @33.153s 0%: 0.009+0.67+0.002 ms clock, 0.009+0.36/0.15/0+0.002 ms cpu, 16-
>16->0 MB, 17 MB goal, 1 P
gc 13 @36.166s 0%: 0.010+0.63+0.002 ms clock, 0.010+0.35/0.12/0+0.002 ms cpu, 16-
>16->0 MB, 17 MB goal, 1 P
gc 14 @39.181s 0%: 0.010+0.65+0.002 ms clock, 0.010+0.34/0.14/0+0.002 ms cpu, 16-
>16->0 MB, 17 MB goal, 1 P
gc 15 @42.194s 0%: 0.010+0.63+0.002 ms clock, 0.010+0.34/0.13/0+0.002 ms cpu, 16-
>16->0 MB, 17 MB goal, 1 P
gc 16 @45.207s 0%: 0.010+0.61+0.002 ms clock, 0.010+0.30/0.16/0+0.002 ms cpu, 16-
>16->0 MB, 17 MB goal, 1 P
gc 17 @48.221s 0%: 0.008+0.42+0.002 ms clock, 0.008+0.23/0.089/0+0.002 ms cpu,
```

可以看到，GC 差不多每 3 秒就发生一次，且每次 GC 都会从 16MB 清理到几乎 0MB，说明程序在不断的申请内存再释放，这是高性能 golang 程序所不允许的。

如果你希望进一步了解 golang 的 GC 日志可以查看《如何监控 golang 程序的垃圾回收》(https://blog.wolfogre.com/redirect/v3/A9DNco5mRFLA-ZPsfPhLuZDu-oKbuLF_wQyMDE2xf8CMDfF_wIwMcUtHy8qzDsGiVTMOxzFMRIDAzwK_jrFxVoWBjtuQQYW3Dsh_cU8BkoKxTsGzDw8Bcw8ghxKiMU)，为保证实验节奏，这里不做展开。

所以接下来使用 pprof 排查时，我们在乎的不是什么地方在占用大量内存，而是什么地方在不停地申请内存，这两者是有区别的。

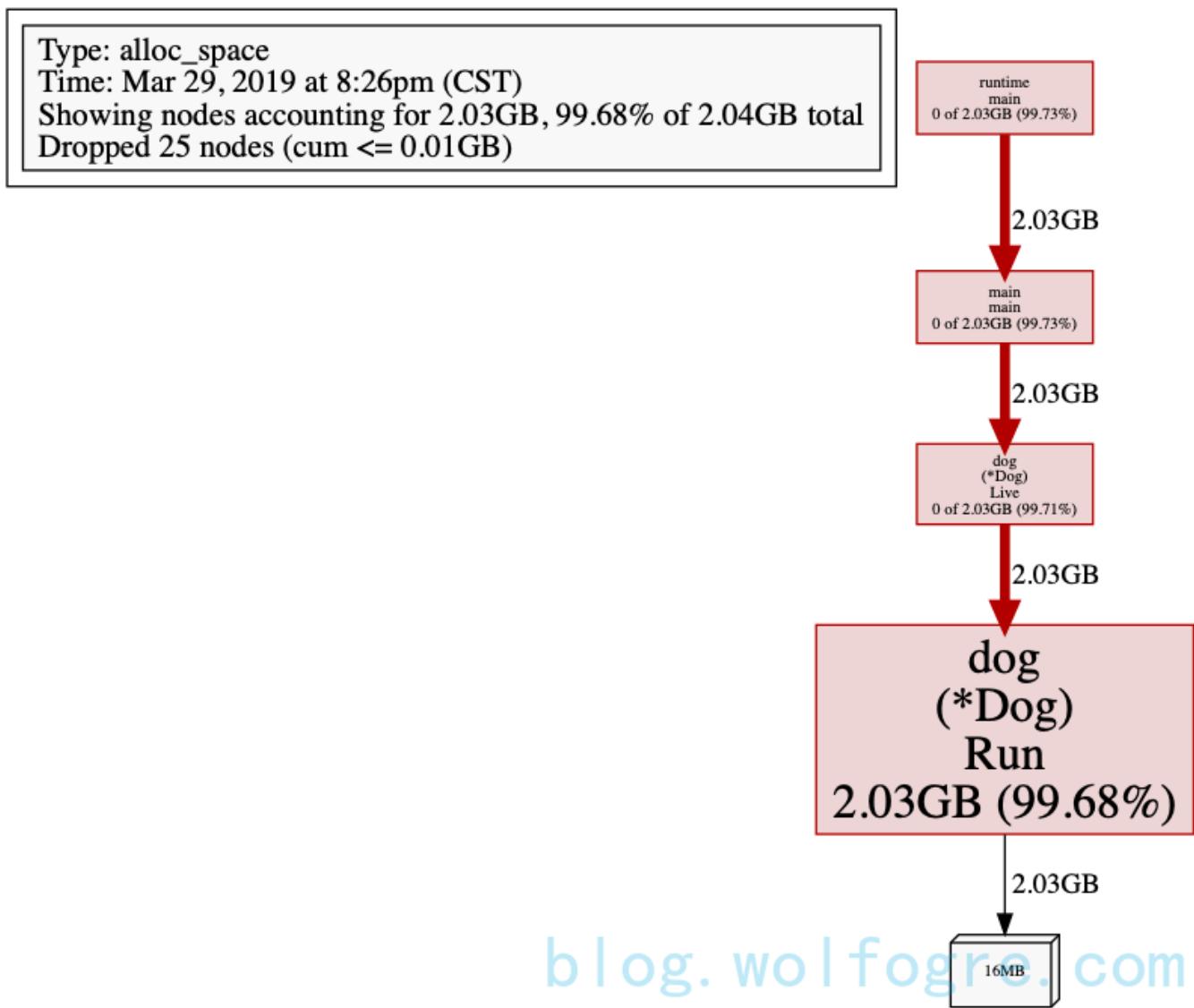
由于内存的申请与释放频度是需要一段时间来统计的，所有我们保证炸弹程序已经运行了几分钟之后，再运行命令：

```
go tool pprof http://localhost:6060/debug/pprof/allocs
```

同样使用 top、list、web 太法：

```
$ go tool pprof http://localhost:6060/debug/pprof/allocs
Fetching profile over HTTP from http://localhost:6060/debug/pprof/allocs
Saved profile in /Users/jason/pprof/pprof.alloc_objects.alloc_space.inuse_objects.inuse_space.027.pb.gz
Type: alloc_space
Time: Mar 29, 2019 at 8:26pm (CST)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 2.03GB, 99.68% of 2.04GB total
Dropped 25 nodes (cum <= 0.01GB)
      flat  flat%  sum%          cum   cum%
  2.03GB 99.68% 99.68%    2.03GB 99.68%  github.com/wolfogre/go-pprof-practice/animal/canidae/dog.(*Dog).Run
      0     0% 99.68%    2.03GB 99.71%  github.com/wolfogre/go-pprof-practice/animal/canidae/dog.(*Dog).Live
      0     0% 99.68%    2.03GB 99.73%  main.main
      0     0% 99.68%    2.03GB 99.73%  runtime.main
(pprof) list Run
Total: 2.04GB
ROUTINE ===== github.com/wolfogre/go-pprof-practice/animal/canidae/dog.(*Dog).Run in /Users/jason/go/src/github.com/wolfogre/go-pprof-practice/animal/canidae/dog/dog.go
  2.03GB    2.03GB (flat, cum) 99.68% of Total
    .          .    38:    log.Println(d.Name(), "pee")
    .          .    39:}
    .          .    40:
    .          .    41:func (d *Dog) Run() {
    .          .    42:    log.Println(d.Name(), "run")
  2.03GB    2.03GB    43:    _ = make([]byte, 16 * constant.Mi)
    .          .    44:}
    .          .    45:
    .          .    46:func (d *Dog) Howl() {
    .          .    47:    log.Println(d.Name(), "howl")
    .          .    48:}

(pprof) |                                     blog.wolfogre.com
```



可以看到 `github.com/wolfogre/go-pprof-practice/animal/canidae/dog.(*Dog).Run` 会进行无意义的内存申请，而这个函数又会被频繁调用，这才导致程序不停地进行 GC：

```
func (d *Dog) Run() {
    log.Println(d.Name(), "run")
    _ = make([]byte, 16 * constant.Mi)
}
```

这里有个小插曲，你可尝试一下将 `16 * constant.Mi` 修改成一个较小的值，重新编译运行，会发现并不会引起频繁 GC，原因是在 golang 里，对象是使用堆内存还是栈内存，由编译器进行逃逸分析并决定，如果对象不会逃逸，便可在使用栈内存，但总有意外，就是对象的尺寸过大时，便不得不使用堆内存。所以这里设置申请 16 MiB 的内存就是为了避免编译器直接在栈上分配，如果那样得话就不会涉及到 GC 了。

我们同样注释掉问题代码，重新编译执行，可以看到这一次，程序的 GC 频度要低很多，以至于短时间内都看不到 GC 日志了：

```
jason@macpro:~/go/src/github.com/wolfogre/go-pprof-practice[debug]
$ go build
jason@macpro:~/go/src/github.com/wolfogre/go-pprof-practice[debug]
$ GODEBUG=gctrace=1 ./go-pprof-practice | grep gc
```

blog.wolfogre.com

排查协程泄露

由于 golang 自带内存回收，所以一般不会发生内存泄露。但凡事都有例外，在 golang 中，协程本身是可能泄露的，或者叫协程失控，进而导致内存泄露。

我们在浏览器里可以看到，此时程序的协程数已经多达 106 条：

← → C ⓘ localhost:6060/debug/pprof/

/debug/pprof/

Types of profiles available:

Count Profile

0 [allocs](#)

3 [block](#)

0 [cmdline](#)

106 [goroutine](#)

0 [heap](#)

1 [mutex](#)

0 [profile](#)

7 [threadcreate](#)

0 [trace](#)

[full goroutine stack dump](#) blog.wolfogre.com

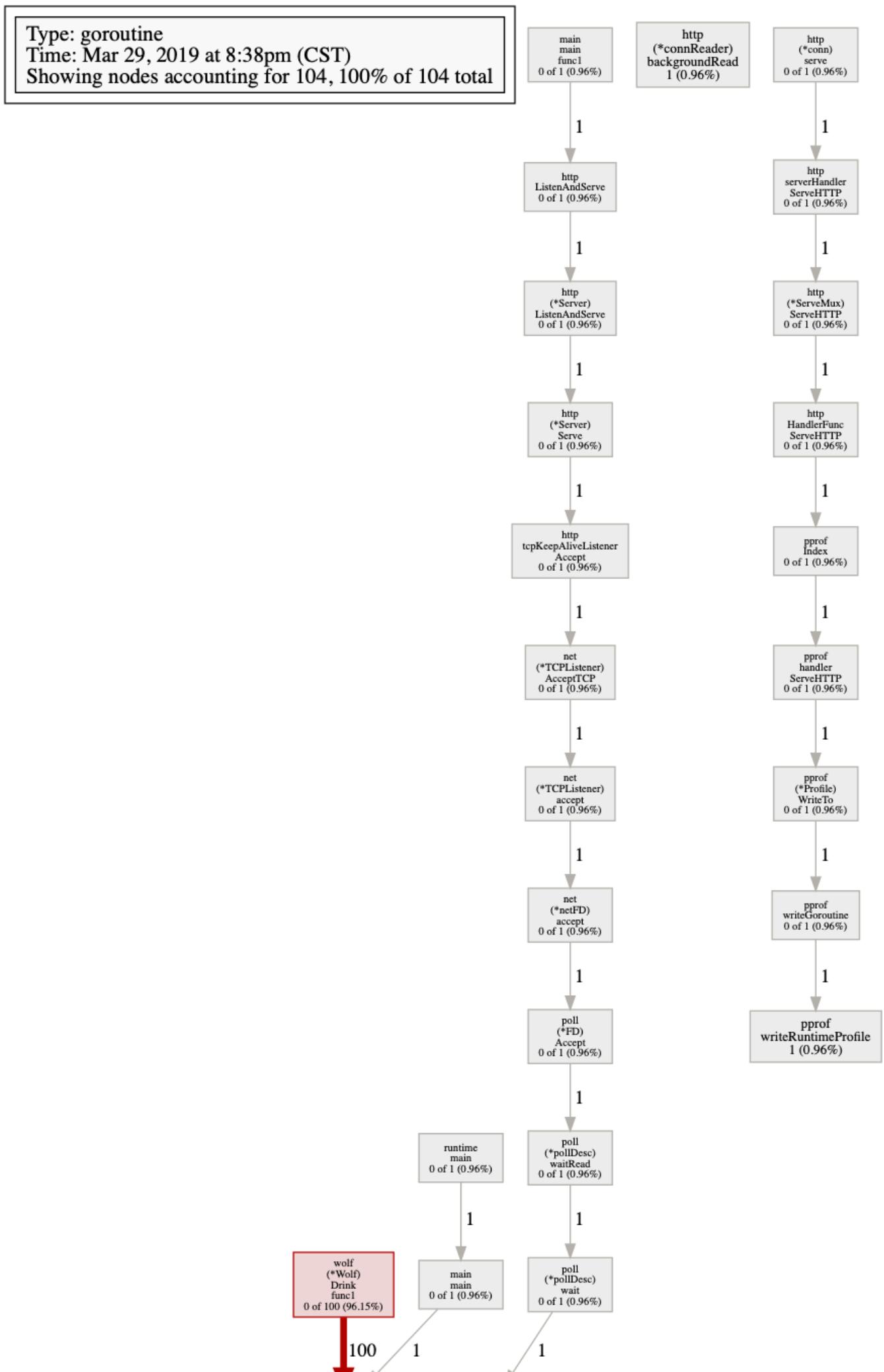
虽然 106 条并不算多，但对于这样一个小程序来说，似乎还是不正常的。为求安心，我们再次是用 pprof 来排查一下：

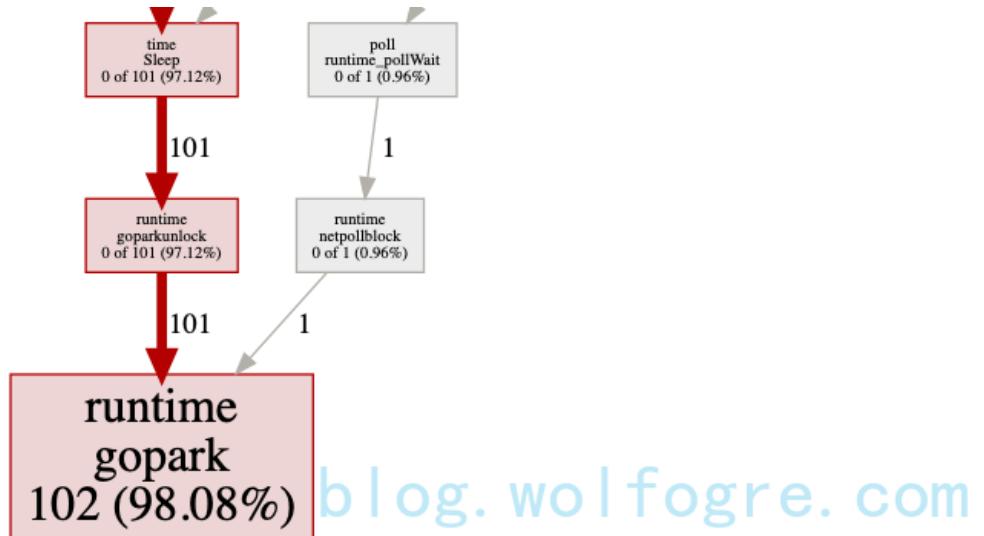
```
go tool pprof http://localhost:6060/debug/pprof/goroutine
```

同样是 top、list、web 大法：

```
$ go tool pprof http://localhost:6060/debug/pprof/goroutine
Fetching profile over HTTP from http://localhost:6060/debug/pprof/goroutine
Saved profile in /Users/jason/pprof/pprof.goroutine.001.pb.gz
Type: goroutine
Time: Mar 29, 2019 at 8:38pm (CST)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 104, 100% of 104 total
Showing top 10 nodes out of 29
      flat  flat%  sum%      cum  cum%
  102 98.08% 98.08%    102 98.08%  runtime.gopark
      1  0.96% 99.04%        1  0.96%  net/http.(*connReader).backgroundRea
d
      1  0.96% 100%        1  0.96%  runtime/pprof.writeRuntimeProfile
      0    0% 100%    100 96.15%  github.com/wolfogre/go-pprof-practic
e/animal/canidae/wolf.(*Wolf).Drink.func1
      0    0% 100%        1  0.96%  internal/poll.(*FD).Accept
      0    0% 100%        1  0.96%  internal/poll.(*pollDesc).wait
      0    0% 100%        1  0.96%  internal/poll.(*pollDesc).waitRead
      0    0% 100%        1  0.96%  internal/poll.runtime_pollWait
      0    0% 100%        1  0.96%  main.main
      0    0% 100%        1  0.96%  main.main.func1
(pprof) list Drink
Total: 104
ROUTINE ===== github.com/wolfogre/go-pprof-practice/animal/ca
nidae/wolf.(*Wolf).Drink.func1 in /Users/jason/go/src/github.com/wolfogre/go-ppr
of-practice/animal/canidae/wolf/wolf.go
      0          100 (flat, cum) 96.15% of Total
      .          .
      .          29:
      .          .
      .          30:func (w *Wolf) Drink() {
      .          .
      .          31:    log.Println(w.Name(), "drink")
      .          .
      .          32:    for i := 0; i < 10; i++ {
      .          .
      .          33:        go func() {
      .          .
      .          100   34:            time.Sleep(30 * time.Second)
      .          .
      .          35:        }()
      .          .
      .          36:    }
      .          .
      .          37:}
      .          .
      .          38:
      .          .
      .          39:func (w *Wolf) Shit() {
```

blog.wolfogre.com





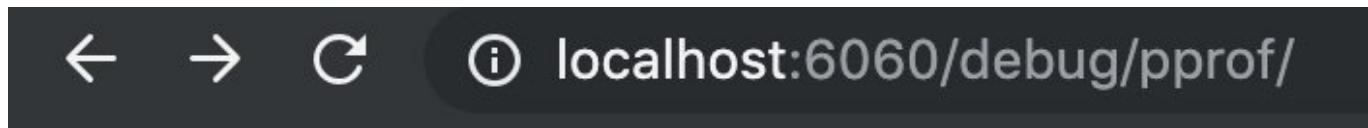
可能这次问题藏得比较隐晦，但仔细观察还是不难发现，问题在于 `github.com/wolfogre/go-pprof-practice/animal/canidae/wolf.(*Wolf).Drink` 在不停地创建没有实际作用的协程：

```

func (w *Wolf) Drink() {
    log.Println(w.Name(), "drink")
    for i := 0; i < 10; i++ {
        go func() {
            time.Sleep(30 * time.Second)
        }()
    }
}
  
```

可以看到，`Drink` 函数每次会释放 10 个协程出去，每个协程会睡眠 30 秒再退出，而 `Drink` 函数又会被反复调用，这才导致大量协程泄露，试想一下，如果释放出的协程会永久阻塞，那么泄露的协程数便会持续增加，内存的占用也会持续增加，那迟早是会被操作系统杀死的。

我们注释掉问题代码，重新编译运行可以看到，协程数已经降到 4 条了：



/debug/pprof/

Types of profiles available:

Count Profile

1 [allocs](#)

2 [block](#)

0 [cmdline](#)

4 [goroutine](#)

1 [heap](#)

1 [mutex](#)

0 [profile](#)

7 [threadcreate](#)

0 [trace](#)

[full goroutine stack dump](#) blog.wolfogre.com

排查锁的争用

到目前为止，我们已经解决这个炸弹程序的所有资源占用问题，但是事情还没有完，我们需要进一步排查那些会导致程序运行慢的性能问题，这些问题可能并不会导致资源占用，但会让程序效率低下，这同样是高性能程序所忌讳的。

我们首先想到的就是程序中是否有不合理的锁的争用，我们倒一倒，回头看看上一张图，虽然协程数已经降到 4 条，但还显示有一个 mutex 存在争用问题。

相信到这里，你已经触类旁通了，无需多言，开整。

```
go tool pprof http://localhost:6060/debug/pprof/mutex
```

同样是 top、list、web 大法：

```
$ go tool pprof http://localhost:6060/debug/pprof/mutex
Fetching profile over HTTP from http://localhost:6060/debug/pprof/mutex
Saved profile in /Users/jason/pprof/pprof.contentions.delay.012.pb.gz
Type: delay
Time: Mar 29, 2019 at 9:07pm (CST)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 3.86mins, 100% of 3.86mins total
      flat  flat%    sum%      cum   cum%
  3.86mins  100%  100%  3.86mins  100% sync.(*Mutex).Unlock
      0      0%  100%  3.86mins  100% github.com/wolfogre/go-pprof-practice/animal/canidae/wolf.(*Wolf).Howl.func1
(pprof) list Howl
Total: 3.86mins
ROUTINE ===== github.com/wolfogre/go-pprof-practice/animal/canidae/wolf.(*Wolf).Howl.func1 in /Users/jason/go/src/github.com/wolfogre/go-pprof-practice/animal/canidae/wolf/wolf.go
      0  3.86mins (flat, cum)  100% of Total
      .          .      53:
      .          .      54:     m := &sync.Mutex{}
      .          .      55:     m.Lock()
      .          .      56:     go func() {
      .          .          57:             time.Sleep(time.Second)
      .          .          58:             m.Unlock()
      .          .          59:     }()
      .          .          60:     m.Lock()
      .          .          61:}

(pprof)
```

blog.wolfogre.com

Type: delay
 Time: Mar 29, 2019 at 9:07pm (CST)
 Showing nodes accounting for 3.86mins, 100% of 3.86mins total

wolf
 (*Wolf)
 Howl
 func1
 0 of 3.86mins (100%)

3.86mins

sync
 (*Mutex)
 Unlock
 3.86mins (100%)

可以看出来这问题出在 `github.com/wolfogre/go-pprof-practice/animal/canidae/wolf.(*Wolf).Howl`。但要知道，在代码中使用锁是无可非议的，并不是所有的锁都会被标记有问题，我们看看这个有问题的锁那儿触雷了。

```
func (w *Wolf) Howl() {
    log.Println(w.Name(), "howl")

    m := &sync.Mutex{}
    m.Lock()
    go func() {
        time.Sleep(time.Second)
        m.Unlock()
    }()
    m.Lock()
}
```

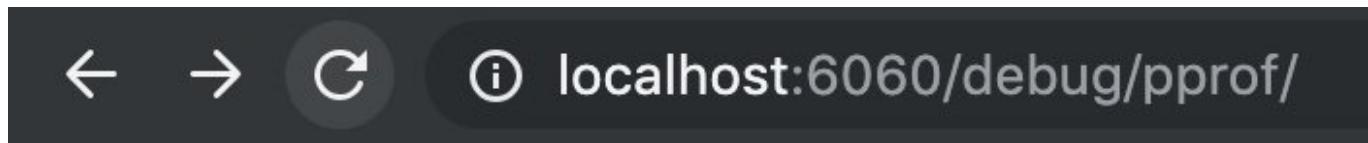
可以看到，这个锁由主协程 Lock，并启动子协程去 Unlock，主协程会阻塞在第二次 Lock 这儿等待子协程完成任务，但由于子协程足足睡眠了一秒，导致主协程等待这个锁释放足足等了一秒钟。虽然这可能是实际的业务需要，逻辑上说得通，并不一定真的是性能瓶颈，但既然它出现在我写的“炸弹”里，就肯定不是什么“业务需要”啦。

所以，我们注释掉这段问题代码，重新编译执行，继续。

排查阻塞操作

好了，我们开始排查最后一个问题。

在程序中，除了锁的争用会导致阻塞之外，很多逻辑都会导致阻塞。



/debug/pprof/

Types of profiles available:

Count Profile

0 [allocs](#)

2 [block](#)

0 [cmdline](#)

4 [goroutine](#)

0 [heap](#)

0 [mutex](#)

0 [profile](#)

7 [threadcreate](#)

0 [trace](#)

[full goroutine stack dump](#) blog.wolfogre.com

可以看到，这里仍有 2 个阻塞操作，虽然不一定是有问题的，但我们保证程序性能，我们还是要老老实实排查确认一下才对。

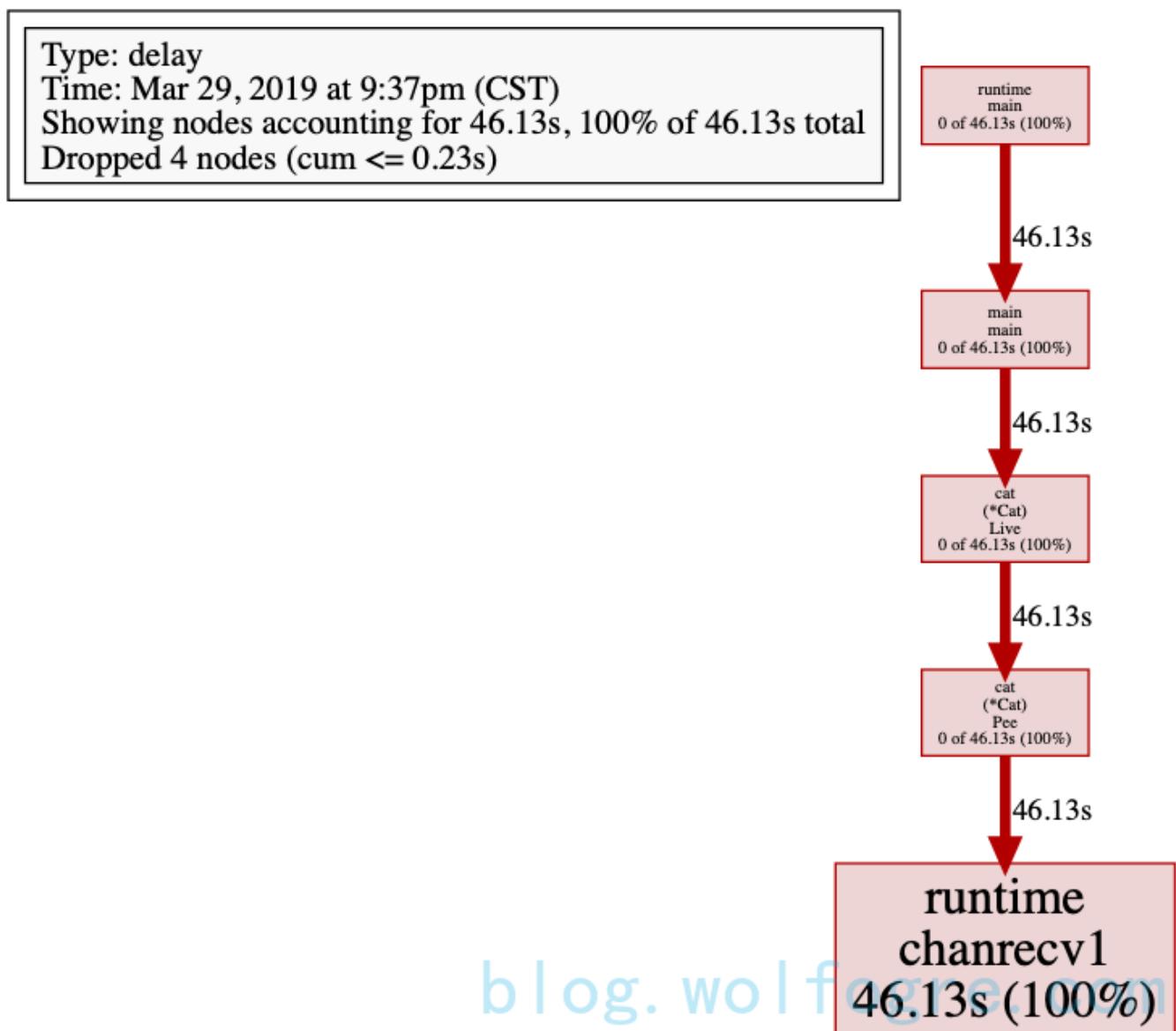
```
go tool pprof http://localhost:6060/debug/pprof/block
```

top、list、web，你懂得。

```
$ go tool pprof http://localhost:6060/debug/pprof/block
Fetching profile over HTTP from http://localhost:6060/debug/pprof/block
Saved profile in /Users/jason/pprof/pprof.contentions.delay.017.pb.gz
Type: delay
Time: Mar 29, 2019 at 9:40pm (CST)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 2.21mins, 100% of 2.21mins total
Dropped 4 nodes (cum <= 0.01mins)
      flat  flat%  sum%      cum  cum%
2.21mins  100%  100%  2.21mins  100%  runtime.chanrecv1
      0    0%  100%  2.21mins  100%  github.com/wolfogre/go-pprof-practic
e/animal/felidae/cat.(*Cat).Live
      0    0%  100%  2.21mins  100%  github.com/wolfogre/go-pprof-practic
e/animal/felidae/cat.(*Cat).Pee
      0    0%  100%  2.21mins  100%  main.main
      0    0%  100%  2.21mins  100%  runtime.main
(pprof) list Pee
Total: 2.21mins
ROUTINE ===== github.com/wolfogre/go-pprof-practice/animal/fe
lidae/cat.(*Cat).Pee in /Users/jason/go/src/github.com/wolfogre/go-pprof-practic
e/animal/felidae/cat/cat.go
      0  2.21mins (flat, cum)  100% of Total
      .          . 34:{}
      .          . 35:{}
      .          . 36:func (c *Cat) Pee() {
      .          . 37:    log.Println(c.Name(), "pee")
      .          . 38:{}
      .  2.21mins 39:    <-time.After(time.Second)
      .          . 40:{}
      .          . 41:{}
      .          . 42:func (c *Cat) Climb() {
      .          . 43:    log.Println(c.Name(), "climb")
      .          . 44:}

(pprof)
```

blog.wolfogre.com



可以看到，阻塞操作位于 [github.com/wolfogre/go-pprof-practice/animal/felidae/cat.\(*Cat\).Pee](https://github.com/wolfogre/go-pprof-practice/animal/felidae/cat.(*Cat).Pee) :

```

func (c *Cat) Pee() {
    log.Println(c.Name(), "pee")

    <-time.After(time.Second)
}
  
```

你应该可以看懂，不同于睡眠一秒，这里是从一个 channel 里读数据时，发生了阻塞，直到这个 channel 在一秒后才有数据读出，这就导致程序阻塞了一秒而非睡眠了一秒。

这里有个疑点，就是上文中是可以看到有两个阻塞操作的，但这里只排查出了一个，我没有找到其准确原因，但怀疑另一个阻塞操作是程序监听端口提供 porof 查询时，涉及到 IO 操作发生了阻塞，即阻塞在对 HTTP 端口的监听上，但我没有进一步考证。

不管怎样，恭喜你完整地完成了这个实验。

思考题

另有一些问题，虽然比较重要，但碍于篇幅（其实是我偷懒），就以思考题的形式留给大家了。

1. 每次进入交互式终端，都会提示“type ‘help’ for commands, ‘o’ for options”，你有尝试过查看有哪些命令和哪些选项吗？有重点了解一下“sample_index”这个选项吗？
2. 关于内存的指标，有申请对象数、使用对象数、申请空间大小、使用空间大小，本文用的是什么指标？如何查看不同的指标？（提示：在内存实验中，试试查看、修改“sample_index”选项。）
3. 你有听说过火焰图吗？要不要在试验中生成一下火焰图？
4. main 函数中的 `runtime.SetMutexProfileFraction` 和 `runtime.SetBlockProfileRate` 是如何影响指标采样的？它们的参数的含义是什么？
5. 评论区有很多很有价值的提问，你有注意到吗？

最后

碍于我的水平有限，实验中还有很多奇怪的细节我只能暂时熟视无睹（比如“排查内存占用过高”一节中，为什么实际申请的是 1.5 GiB 内存），如果这些奇怪的细节你也发现了，并痛斥我假装睁眼瞎，那么我的目的就达到了……

——还记得吗，本文的目的是让你熟悉使用 pprof，消除对它的陌生感，并能借用它进一步得了解 golang。而你通过这次试验，发现了程序的很多行为不同于你以往的认知或假设，并抱着好奇心，开始向比深处更深处迈进，那么，我何尝不觉得这是本文的功德呢？

与君共勉。

← 记·刚过去的三个月 (<https://blog.wolfogre.com/posts/passed-three-months/>)

彻底搞懂 golang 里的 iota → (<https://blog.wolfogre.com/posts/golang-iota/>)

21 Comments - powered by [utteranc.es](#)

blundel commented on 2019年6月14日

感谢，对pprof有了最基本的了解和使用，但还是浅显了点，期待下一篇。

duncanwang commented on 2019年8月10日

超级棒，我注明出处，转载一下哦。

yanjinbin commented on 2019年11月7日

为什么我的web svg没有dog 函数分支啊

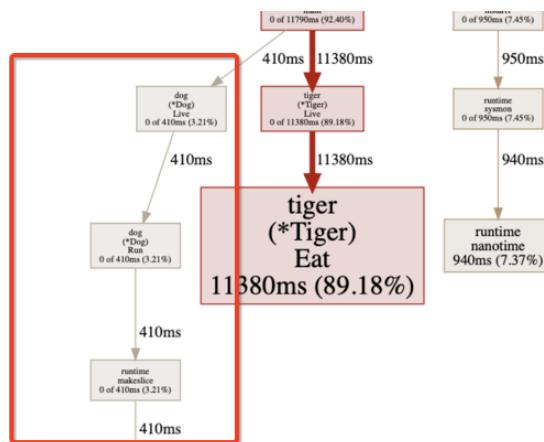
wolfogre commented on 2019年11月7日

Owner

@yanjinbin 可以说明下是哪一步不符合预期吗？如果是“排查频繁内存回收”那一步，请确认下有没有让程序先运行几分钟再执行 go pprof，时间间隔过短的话可能从 profile 里看不出东西。

yanjinbin commented on 2019年11月7日

执行 go pprof，时间间隔过短的话可能从 profile 里看不出东西。

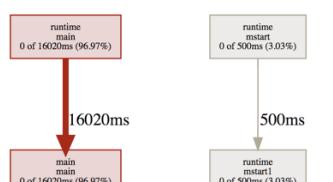


cpu内存过高 就这个红色标注的分支 我web生成不了
我的如下

```

cpu
Nov 7, 2019 at 5:10pm (CST)
ion: 30s, Total samples = 16520ms (55.06%)
ing nodes accounting for 16480ms, 99.76% of 16520ms total
ed 21 nodes (cum <= 82.60ms)

```



yanjinbin commented on 2019年11月7日

还有 火焰图🔥 跟这个pprof差不多吧 为什么uber的 go torch要说
1.11以上都整合到pprof上面去了呢 纳闷

wolfogre commented on 2019年11月8日

Owner

@yanjinbin 这是正常现象，因为我写的代码只是靠疯狂循环来提升
CPU占用率，并不能精准控制占用多少，况且 pprof 采样也会有一

些误差，你可以看到我的 tiger 占用是 89.18%，而你的是 96.85%，而这个树形调用栈图只会显示占用较高的调用栈，不是所有的调用栈，那样的话图就没法看了，所以你的图里，dog 的 CPU 占用实际上是被认为太小了所以被忽略了。这不重要，重要的 tiger 被正确发现了。

所以火焰图的优点就在这里，它是另一种图形化展示调用栈资源占用的方式，不仅能突出资源占用的大户，对于占用量不高的调用栈也不会完全丢失细节，但可能没有树形调用栈图这么直白易懂。

yanjinbin commented on 2019年11月8日

@yanjinbin 这是正常现象，因为我写的代码只是靠疯狂循环来提升CPU占用率，并不能精准控制占用多少，况且 pprof 采样也会有一些误差，你可以看到我的 tiger 占用是 89.18%，而你的是 96.85%，而这个树形调用栈图只会显示占用较高的调用栈，不是所有的调用栈，那样的话图就没法看了，所以你的图里，dog 的 CPU 占用实际上是被认为太小了所以被忽略了。这不重要，重要的 tiger 被正确发现了。

所以火焰图的优点就在这里，它是另一种图形化展示调用栈资源占用的方式，不仅能突出资源占用的大户，对于占用量不高的调用栈也不会完全丢失细节，但可能没有树形调用栈图这么直白易懂。

嗯 谢谢，我现在想知道的就是 火焰图 到底有没有可以用1.13版本的pprof展现呢 之前是uber go torch项目

yanjinbin commented on 2019年11月8日

楼主 现在的火焰图 还是用uber的go torch吗 不是说整合到pprof上面去了？

yanjinbin commented on 2019年11月8日

1数据采样： go tool pprof pprof [http://127.0.0.1:\[端口号\]/debug/pprof/profile](http://127.0.0.1:[端口号]/debug/pprof/profile) -seconds [采样间隔时间数字]

```
go tool pprof pprof http://127.0.0.1:6060/debug/pprof/profile
-seCONDS 10
```

2生成火焰图： go tool pprof -http=:8081 ~/pprof/[文件路径名].pb.gz

wolfogre commented on 2019年11月8日

Owner

@yanjinbin go-torch 已经废弃了， pprof 整合了火焰图这事儿我之前确实不知道。

刚试了一下你写的步骤， 赞👍。不过有个笔误哈，你想说的应该是：

```
go tool pprof --seconds [采样间隔时间数字] http://127.0.0.1:
```

wolfogre commented on 2019年12月25日

Owner

来自 @wenchaopeng 于 issue #1 的提问：

pprof统计的内存与PC上显示的内存不符合

你好，看了你写的关于pprof的博客，写的很好，不过有一个问题，你那个炸弹程序占用内存大概2GB，我在linux下跑了，通过htop查看，确实在2GB左右，但是通过pprof查看，只能查到1GB内存占用，请教一下这是为什么，两边内存对不上，请解惑！

好问题哈。是这样的，代码中一共有两处有意消耗内存，一处是这里：

```
func (m *Mouse) Steal() {
    log.Println(m.Name(), "steal")
    max := constant.Gi
    for len(m.buffer) * constant.Mi < max {
        m.buffer = append(m.buffer, [constant.Mi
    }
}
```

lieinsun commented on 2020年1月7日

哇哦，超级棒的文章，教程向的博客一顿好找啊

wdd817 commented on 2020年3月16日

讲道理，粗浅但是特别棒的入门文章，👍

Tracy-Hu commented on 2020年8月4日

优质文章o(￣▽￣)d 关于排查阻塞部分，我按照步骤一步步到这里后，block数量显示为1而不是文中的2啦，所以小编最后的疑点是不是没问题的~另外，web生成的图，里面的虚线代表什么意思呀？麻

烦小编帮忙解答下Thanks♪(·ω·)/

SeasonWoo commented on 2021年8月3日

写的很棒，对于入门真的特别有帮助！

kevinx1 commented on 2021年9月1日

很全面，学习中！

chaggle commented on 2022年3月9日

好耶！

jieyuefeng commented on 2022年3月12日

你好，文中《Golang 大杀器之性能剖析 PProf》链接更新到了
<https://github.com/eddycjy/blog/blob/master/content/posts/go/tools/2018-09-15-go-tool-pprof.md>

lyfwfm commented on 2022年5月21日

请问该方法能在生产环境使用吗？我担心pprof连上生产环境进程
(因为进程此时已经cpu和内存很高了)，导致进程直接挂掉。

© 2016-2022 Wolfogre's Blog · Powered by Hugo (<http://gohugo.io>).

皖ICP备16005679 (<https://beian.miit.gov.cn/>)号

最近维护于2022年07月19日

累计 286.5k+ 次真实访问