极客时间 训练营

Go 测试 —— benchmark 测试

大明

目录

benchmark 测试入门

极客时间 训练营

- 测试以 Bnechmark 为方法开头
- 运行测试的时候，形如普通的测试，但是需要加上 –bench 选项
- 运行选项：
  - –bench 选项：接受一个正则表达式，匹配上的才会执行
  - –benchmem: 输出内存分配
  - –benchtime：运行时间，默认 1s。可以是时间，也可以是次数，例如 1s, 2m, 500x
  - –count：轮数，执行多少轮
- 生成 profile 文件，用于 pprof 工具分析：
  - –cpuprofile=cpu.out
  - –memprofile=mem.out
  - –blockfile=block.out
- 运行结果主要包括：运行次数、单次运行耗时、总耗时

```go
func Fib(n int) int {
    if n < 2 {
        return n
    }
    return Fib(n-1) + Fib(n-2)
}
```

```go
func BenchmarkFib(b *testing.B) {
    Fib( n: 40)
}
```

```
PS D:\workspace\go\src\geekbang\geekbang-go-camp\test> go test `-bench=.
goos: windows
goarch: amd64
pkg: geekbang/geekbang-go-camp/test
cpu: Intel(R) Core(TM) i5-10400F CPU @ 2.90GHz
BenchmarkFib-12         1000000000               0.4424 ns/op
PASS
ok      geekbang/geekbang-go-camp/test  10.318s
PS D:\workspace\go\src\geekbang\geekbang-go-camp\test>
```

目录

内存和 CPU 分析 —— pprof 工具

极客时间 训练营

1. 使用默认的 pprof 工具：go tool pprof cpu.prof

2. 图形界面依赖于 graphviz

3. 常用命令（一般建议优先使用 web 调出图形化界面进行分析）：

- top：列出消耗最高的调用
- list：列出问题代码片段
- peek：查询具体函数的调用关系
- web：图形化界面

```
➜  eorm git:(main) go tool pprof mem.out
File: eorm.test
Type: alloc_space
Time: May 28, 2022 at 11:56am (+08)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)
```

```
Showing top 10 nodes out of 42
      flat  flat%   sum%        cum   cum%
    1025kB 18.15% 18.15%     1025kB 18.15%  runtime.allocm
  902.59kB 15.98% 34.13%  2030.26kB 35.95%  compress/flate.NewWriter
  583.01kB 10.32% 44.45%   583.01kB 10.32%  compress/flate.newDeflateFast (inline)
  544.67kB  9.64% 54.10%  1127.67kB 19.97%  compress/flate.(*compressor).init
  528.17kB  9.35% 63.45%   528.17kB  9.35%  io.copyBuffer
  528.17kB  9.35% 72.80%   528.17kB  9.35%  regexp.(*bitState).reset
  512.20kB  9.07% 81.87%   512.20kB  9.07%  runtime.malg
  512.09kB  9.07% 90.93%   512.09kB  9.07%  compress/gzip.NewWriterLevel
  512.01kB  9.07%   100%   512.01kB  9.07%  github.com/gotomicro/eorm/internal/model.underscoreName
         0     0%   100%  2030.26kB 35.95%  compress/gzip.(*Writer).Write
```

内存和 CPU 分析 —— pprof 工具

1. 使用默认的 pprof 工具：go tool pprof cpu.prof

2. 图形界面依赖于 graphviz

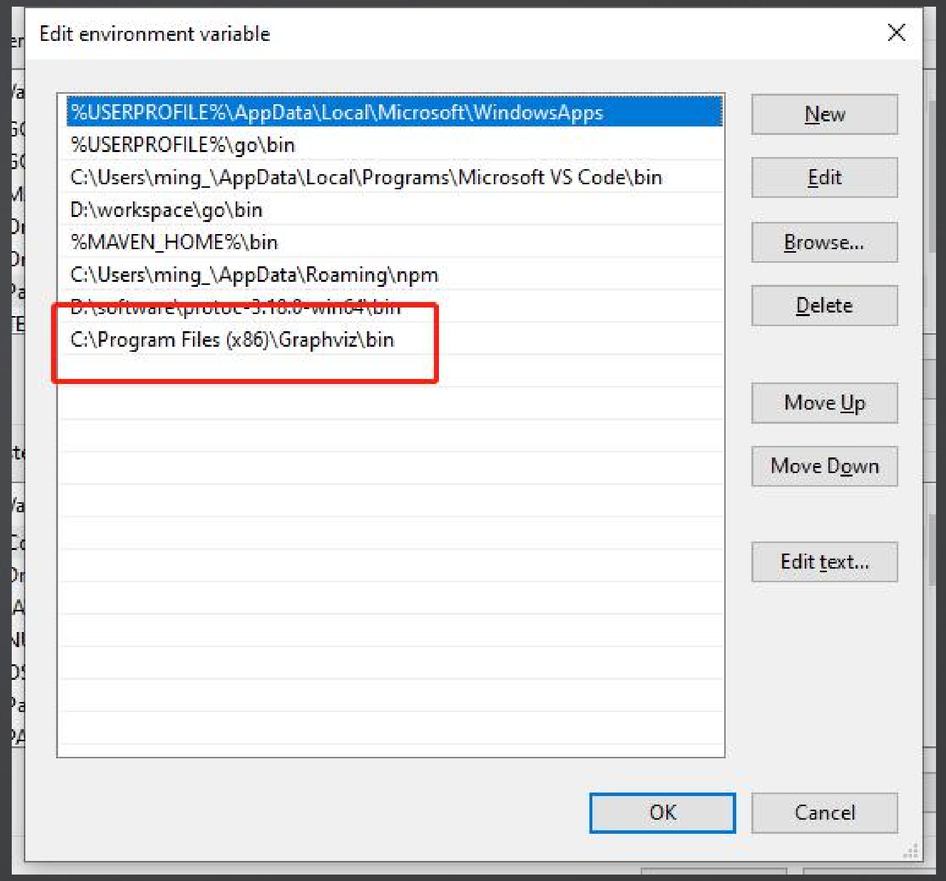
3. 常用命令：
   - top：列出消耗最高的调用
   - list：列出问题代码片段
   - peek：查询具体函数的调用关系
   - web：图形化界面

```
(pprof) list runtime.allocm
Total: 5.52MB
ROUTINE ======================== runtime.allocm in /home/mindeng/s
       1MB        1MB (flat, cum) 18.15% of Total
         .          .   1738:           }
         .          .   1739:           sched.freem = newList
         .          .   1740:           unlock(&sched.lock)
         .          .   1741:   }
         .          .   1742:
       1MB        1MB   1743:   mp := new(m)
         .          .   1744:   mp.mstartfn = fn
         .          .   1745:   mcommoninit(mp, id)
         .          .   1746:
```

内存和 CPU 分析 —— google pprof 工具

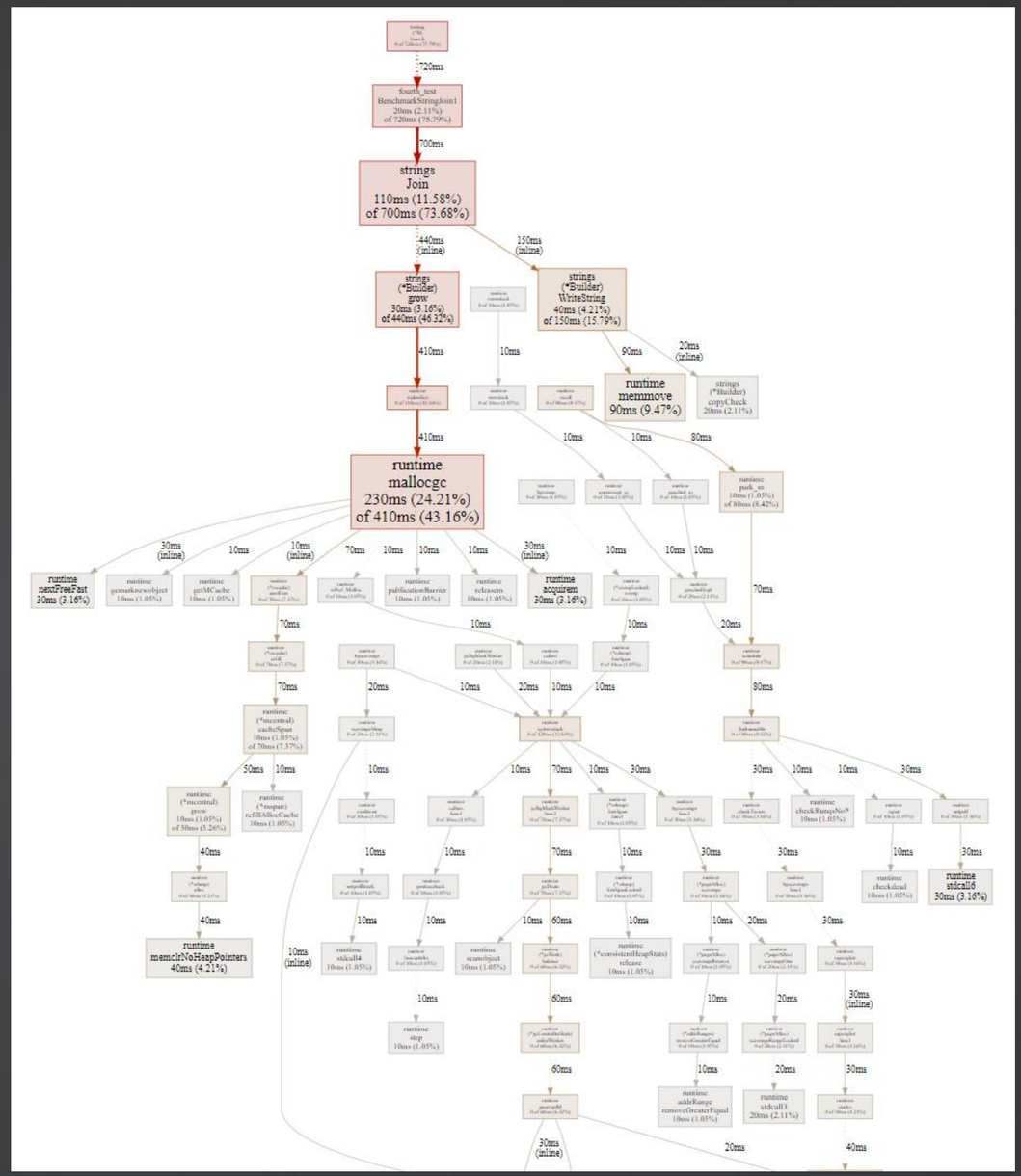
使用默认的 pprof 工具：go tool pprof cpu.prof

1. 使用 pprof 工具来分析：go get –u github.com/google/pprof
2. 安装 graphviz
3. web 界面： pprof –http=:8080 cpu.prof



内存和 CPU 分析 —— 调用链路

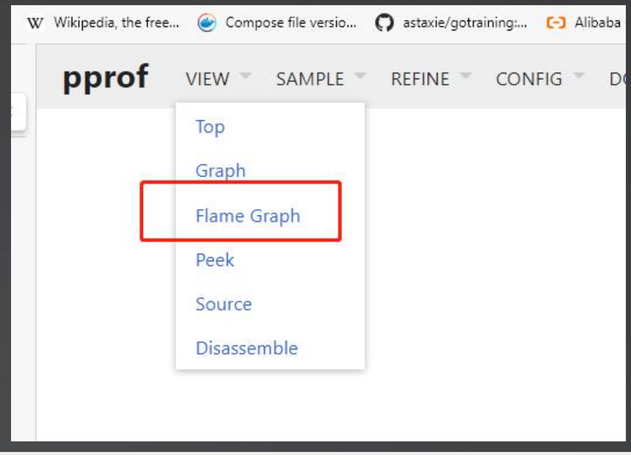
1. 中间标红的就是关键路径，或者说瓶颈所在的路径
2. 在一般的企业级应用中，这条路径都是因为内存分配引起的
3. 除了关键路径，一些开销比较大的分支路径也要关心
4. 优化关键路径性价比最高（可以做到数量级提升）。但是有时候关键路径优化不动，那么可以尝试优化其它分支路径

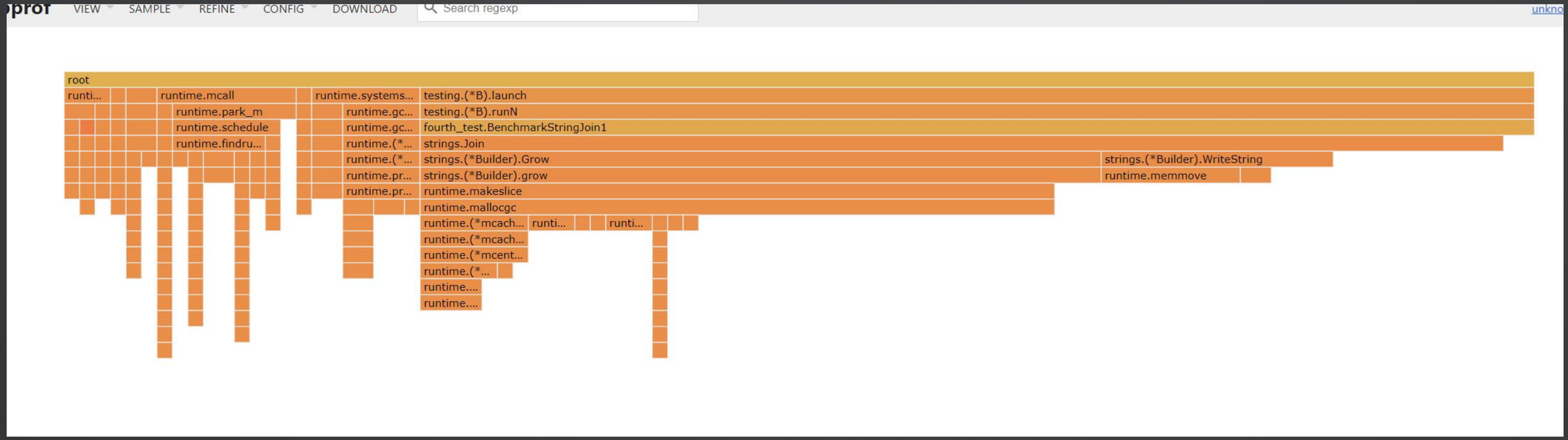


内存和 CPU 分析 —— 火焰图

长度代表的就是资源消耗占比。

例如 JOIN 方法，主要由 Grow 和 WriteString 组成，其中 Grow 很长，代表它是性能瓶颈。





内存和 CPU 分析 —— net/http/pprof

net/http/pprof 提供了可视化界面来查看程序的各种指标：

- 如果本身是 web 应用，可以只引入 pprof 包
- 最佳实践是为 pprof 准备一个专门的端口，并且该端口只可以在内网访问

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

func main() {
    go func() {
        log.Println(http.ListenAndServe(addr: "localhost:6060", handler: nil))
    }()
    beego.Run()
}
```

匿名引入pprof 包

至少要启动一个端口

```
/debug/pprof/

Types of profiles available:
Count Profile
5     allocs
0     block
0     cmdline
7     goroutine
5     heap
0     mutex
0     profile
7     threadcreate
0     trace
full goroutine stack dump
```

内存和 CPU 分析 —— net/http/pprof

极客时间 训练营

可以考虑结合 go tool pprof 工具来使用（难以手工解读）：

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

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

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

```
heap profile: 12: 1064976 [17: 3133456] @ heap/1048576    heap，贼难懂
1: 1048576 [1: 1048576] @ 0xaa9f70 0xaa9e86 0xaa5067 0xc7c045 0xc7ce89 0x858483 0x85ae95 0x85d254 0x8572dc 0x471381
#       0xaa9f6f        runtime/pprof.writeGoroutineStacks+0x4f /home/mindeng/software/go/src/runtime/pprof/pprof.go:692
#       0xaa9e85        runtime/pprof.writeGoroutine+0x45       /home/mindeng/software/go/src/runtime/pprof/pprof.go:683
#       0xaa5066        runtime/pprof.(*Profile).WriteTo+0xa6   /home/mindeng/software/go/src/runtime/pprof/pprof.go:332
#       0xc7c044        net/http/pprof.handler.ServeHTTP+0x444  /home/mindeng/software/go/src/net/http/pprof/pprof.go:253
#       0xc7ce88        net/http/pprof.Index+0xe8               /home/mindeng/software/go/src/net/http/pprof/pprof.go:371
#       0x858482        net/http.HandlerFunc.ServeHTTP+0x42     /home/mindeng/software/go/src/net/http/server.go:2084
#       0x85ae94        net/http.(*ServeMux).ServeHTTP+0x134    /home/mindeng/software/go/src/net/http/server.go:2462
#       0x85d253        net/http.serverHandler.ServeHTTP+0x473  /home/mindeng/software/go/src/net/http/server.go:2916
#       0x8572db        net/http.(*conn).serve+0x193b           /home/mindeng/software/go/src/net/http/server.go:1966

1: 6528 [1: 6528] @ 0x56c1a5 0x56be05 0x496bf8 0x496a85 0x56c7ed 0xb7267a 0xc7dd5b 0xc7d3c9 0x858483 0x85ae95 0x85d254 0x8572
#       0x56c1a4        strings.(*Replacer).build+0x304         /home/mindeng/software/go/src/strings/replace.go:75
#       0x56be04        strings.(*Replacer).buildOnce+0x24      /home/mindeng/software/go/src/strings/replace.go:40
#       0x496bf7        sync.(*Once).doSlow+0x137               /home/mindeng/software/go/src/sync/once.go:68
#       0x496a84        sync.(*Once).Do+0x44                    /home/mindeng/software/go/src/sync/once.go:59
#       0x56c7ec        strings.(*Replacer).Replace+0x6c        /home/mindeng/software/go/src/strings/replace.go:96
#       0xb72679        html.EscapeString+0x39                  /home/mindeng/software/go/src/html/escape.go:179
#       0xc7dd5a        net/http/pprof.indexTmplExecute+0x57a   /home/mindeng/software/go/src/net/http/pprof/pprof.go:429
#       0xc7d3c8        net/http/pprof.Index+0x628              /home/mindeng/software/go/src/net/http/pprof/pprof.go:402
#       0x858482        net/http.HandlerFunc.ServeHTTP+0x42     /home/mindeng/software/go/src/net/http/server.go:2084
#       0x85ae94        net/http.(*ServeMux).ServeHTTP+0x134    /home/mindeng/software/go/src/net/http/server.go:2462
#       0x85d253        net/http.serverHandler.ServeHTTP+0x473  /home/mindeng/software/go/src/net/http/server.go:2916
#       0x8572db        net/http.(*conn).serve+0x193b           /home/mindeng/software/go/src/net/http/server.go:1966

4: 4096 [4: 4096] @ 0x4431ea 0x4439a5 0x443fa5 0x44437a 0x445a85 0x446039 0x442a52 0x44293e 0x46f0c5
```

内存和 CPU 分析 —— trace

- trace 数据可以从前面 net/http/pprof 里面下载下来
- 或者使用 trace 包来采集
- 或者使用 –trace 的测试标记位
- 使用 go tool trace trace.out 命令来解析

```
    "runtime/trace"
)

func main() {
    trace.Start(os.Stderr)
    defer trace.Stop()
    fmt.Println(a...: "hello")
}
```

这里可以设置为某个文件

```
go run main.go 2> trace.out
```

内存和 CPU 分析 —— trace

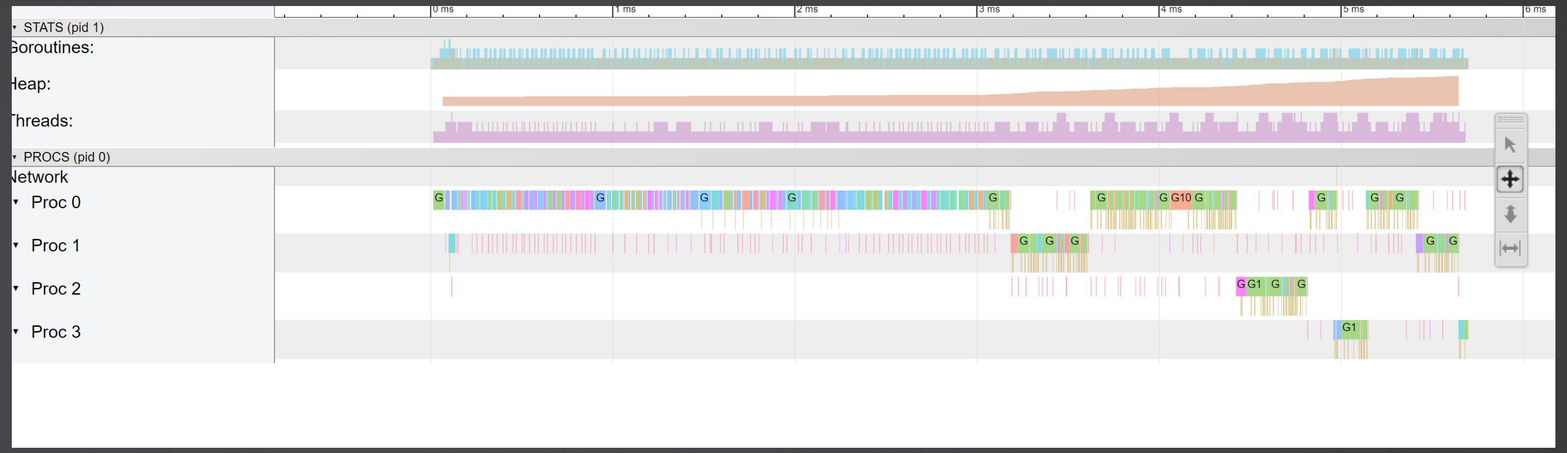
- trace 数据可以从前面 net/http/pprof 里面下载下来
- 或者使用 trace 包来采集
- 使用 go tool trace trace.out 命令来解析
- 常用数据：
  - View trace：查看 trace
  - Goroutine analysis：goroutine 分析
  - Network blocking profile：网络阻塞
  - Synchronization blocking profile：同步阻塞
  - Syscall blocking profile：系统调用阻塞
  - Scheduler latency profile：调度延迟

```
➜  hello go tool trace trace.out
2022/05/28 13:27:10 Parsing trace...
2022/05/28 13:27:10 Splitting trace...
2022/05/28 13:27:10 Opening browser. T
```

View trace
Goroutine analysis
Network blocking profile (⬇)
Synchronization blocking profile (⬇)
Syscall blocking profile (⬇)
Scheduler latency profile (⬇)
User-defined tasks
User-defined regions
Minimum mutator utilization

内存和 CPU 分析 —— view trace

- timeline：时间线，表达执行时间
- Heap：内存占用，主要分析 goroutine 哪个消耗比较大，可以用来辅助分析内存逃逸
- Goroutines：执行中的 goroutine 和可执行的 goroutine
- Threads：系统线程
- PROCS：Go 概念中的 processor



个人感觉不太好用这个，虽然提供了很丰富的信息，
但是大多数时候都用不上

内存和 CPU 分析 —— view trace thread

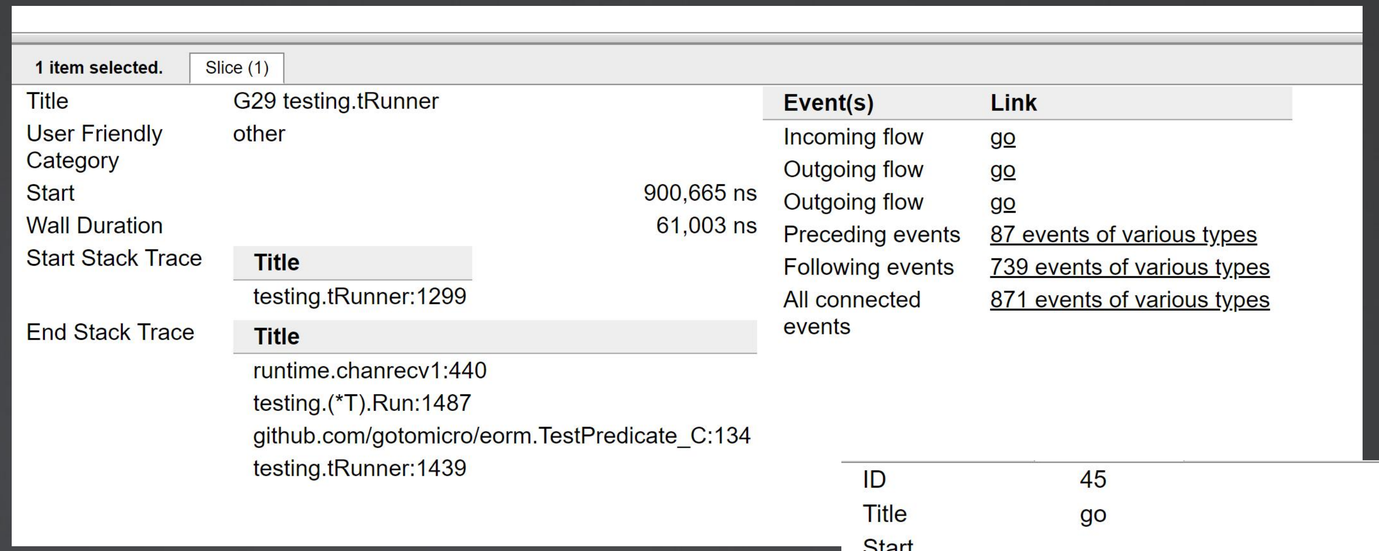
可以理解为采样的时候，处于不同状态的线程的数量。

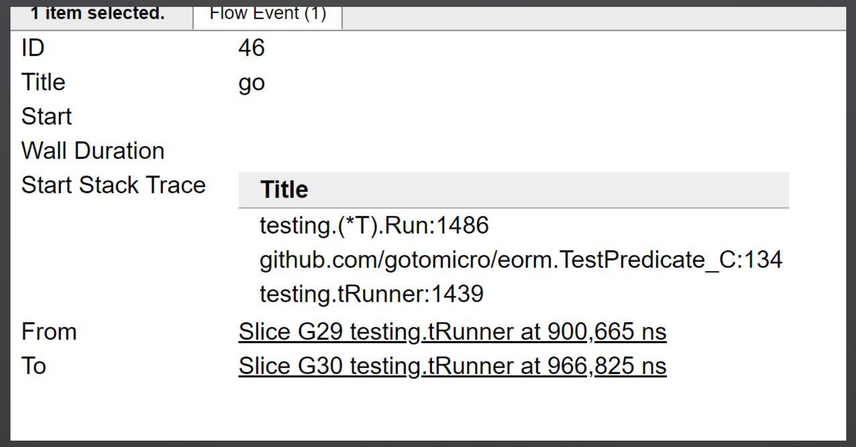
2 items selected. Counter Samples (2)

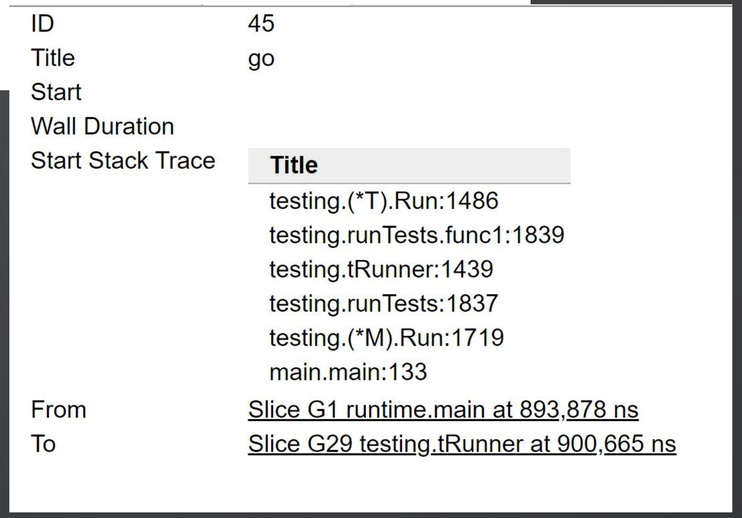
Counter	Series	Time	Value
Threads	InSyscall	0.149464	0
Threads	Running	0.149464	2

内存和 CPU 分析 —— view trace processor

- Title：名字
- Start：开始时间点
- Wall Duration：持续时间
- Stack Trace：开始时刻或者结束时刻的栈
- Events：发生了什么
  - 可以用来分析 goroutine 调度，例如从哪个 goroutine 切换到哪个 gotouine







目录

Go 逃逸分析

在 Go 里面，对象可以被分配到栈或者堆上。分配到堆上的，被称为内存逃逸。

可能的原因（不是必然引起逃逸，而是可能逃逸。是否逃逸还跟执行上下文有关）：

- 指针逃逸：如方法返回局部变量指针
- interface{} 逃逸：如使用 interface{} 作为参数或者返回值
- 接口逃逸：如以接口作为返回值
- 大对象：大对象会直接分配到堆上
- 栈空间不足
- 闭包：闭包内部引用了外部变量
- channel 传递指针

一般可以使用 gcflags=-m 来分析内存逃逸

Go 逃逸分析——指针逃逸

指针逃逸：如方法返回局部变量指针

```
2
3   type User struct {
4       Name string
5   }
6
7   func ReturnPointer() *User {
8       return &User{
9           Name: "Tom",
10      }
11  }
```

```
# geekbang/geekbang-go-camp/third/escape
.\escape.go:7:6: can inline ReturnPointer
.\escape.go:8:9: &User{...} escapes to heap
```

Go 逃逸分析——interface{} 逃逸

极客时间 训练营

interface{} 逃逸：如使用 interface{} 作为参数或者返回值

```
15      func Printf() {
16          str := "Hello"
17          fmt.Println(str)
18      }
19
```

```
PS D:\workspace\go\src\geekbang\geekbang-go-camp\third\escape
# geekbang/geekbang-go-camp/third/escape
.\escape.go:17:13: inlining call to fmt.Println
.\escape.go:17:13: str escapes to heap
.\escape.go:17:13: []interface {}{...} does not escape
```

Go 逃逸分析——接口逃逸

极客时间 训练营

接口逃逸：如以接口作为返回值

```go
29  func ReturnUserV1() User {
30      u := User{}
31      return u
32  }
33
34  func ReturnUserV2() Animal {
35      u := User{}
36      return u
37  }
```

```
# geekbang/geekbang-go-camp/third/escape
.\escape.go:13:6: can inline User.Eat
.\escape.go:14:12: inlining call to fmt.Printf
.\escape.go:29:6: can inline ReturnUserV1
.\escape.go:34:6: can inline ReturnUserV2
.\escape.go:13:7: u does not escape
.\escape.go:36:2: u escapes to heap
<autogenerated>:1: leaking param: .this
<autogenerated>:1: inlining call to User.Eat
<autogenerated>:1: inlining call to fmt.Printf
<autogenerated>:1: .this does not escape
<autogenerated>:1: leaking param content: .this
<autogenerated>:1: .this does not escape
```

Go 逃逸分析——闭包

闭包：闭包引用外部变量

```
37  func CountFn() func() int {
38      n := 0
39      return func() int {
40          n++
41          return n
42      }
43  }
```

```
PS D:\workspace\go\src\geekbang\geekbang-go-camp\thir
# geekbang/geekbang-go-camp/third/escape
.\escape.go:37:6: can inline CountFn
.\escape.go:39:9: can inline CountFn.func1
.\escape.go:38:2: moved to heap: n
.\escape.go:39:9: func literal escapes to heap
<autogenerated>:1: leaking param: .this
```

Go 逃逸分析—— channel 传递指针

极客时间 训练营

channel 传递指针

```
44
45  func Channel() chan *User {
46      ch := make(chan *User, 1)
47      a := &User{
48          Name: "Tom",
49      }
50      ch <- a
51      return ch
52  }
53
```

```
<autogenerated>:1: leaking param: .this
PS D:\workspace\go\src\geekbang\geekbang-go-camp\
# geekbang/geekbang-go-camp/third/escape
.\escape.go:45:6: can inline Channel
.\escape.go:47:7: &User{...} escapes to heap
<autogenerated>:1: leaking param: .this
```

THANKS