Golang性能分析优化

> Go Profiling and Optimization

概述

go有着强大的runtime profiling工具，例如`pprof`

- CPU profiles (stack sampling)：记录cpu以及调用栈
- 内存 Heap profiles using allocation profiling
- block profiles， traces

用法

- 在test或benchtest中`go test . -bench . -cpuprofile prof.cpu`，然后使用go tool pprof prof.cpu 进行分析
- `uber/go-torch`: 根据Go's pprof的结果，构造输出火焰图

go pprof

- top10 top10的占用entries
- -cum cumulative累积的权重
- svg 可视化输出为svg图

实战

> 最近在做pingcap的talent-plan，实验一即是编写并行的merge_sort，要求比单线程quick_sort性能更优，输入数据为16m个int64的数据，并进行profile

首先第一版并没有对merge_sort进行并行度的区分，开启了过多的goroutine，在时间上非常耗时。明显比普通内置sort要慢很多。

两种优化策略

> 两种'[]'优化策略主要思想一致即，如何充分利用多核优势

- 混合排序优化：可以设置当数组元素个数小于某个阈值K之后，直接使用内置的排序，不再进行递归

- 协程粒度优化：可以设置当数组元素小于某个阈值K之后，不再开启新的协程

1. 内置sort进行profile

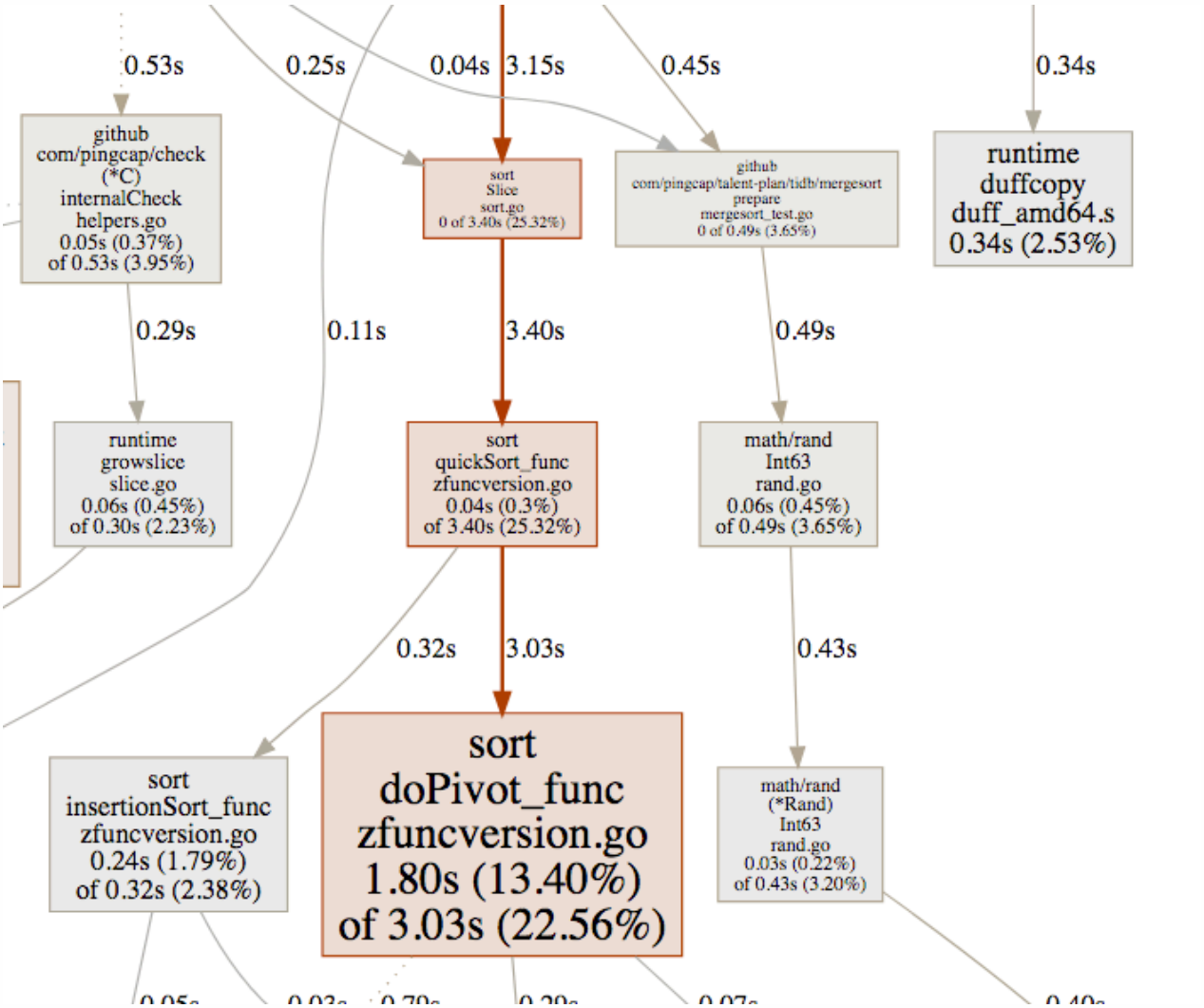
总耗时如下图：可以看到总耗时**5.88s**。

```
➜  mergesort git:(master) ✗ go test -bench BenchmarkNormalSort -cpuprofile normal.out
OK: 1 passed
1 2 3 4 5 6 9 goos: darwin
goarch: amd64
pkg: github.com/pingcap/talent-plan/tidb/mergesort
BenchmarkNormalSort-4                  1        3994045955 ns/op
PASS
ok      github.com/pingcap/talent-plan/tidb/mergesort   5.889s
```

使用如下命令：`go test -bench BenchmarkNormalSort -cpuprofile normal.out` 进行cpu profile。

```
➜  mergesort git:(master) ✗ go tool pprof normal.out
Type: cpu
Time: Apr 6, 2019 at 1:21pm (CST)
Duration: 8.90s, Total samples = 13.43s (150.96%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top10
Showing nodes accounting for 8080ms, 60.16% of 13430ms total
Dropped 118 nodes (cum <= 67.15ms)
Showing top 10 nodes out of 125
      flat  flat%   sum%        cum   cum%
    2260ms 16.83% 16.83%     2260ms 16.83%  runtime.usleep /usr/local/Cellar/go/1.9.1/libexec/src/runtime/sys_darwin_amd64.s
    1800ms 13.40% 30.23%     3030ms 22.56%  sort.doPivot_func /usr/local/Cellar/go/1.9.1/libexec/src/sort/zfuncversion.go
     980ms  7.30% 37.53%     3820ms 28.44%  github.com/pingcap/talent-plan/tidb/mergesort.mergeSort.func2 /Users/wangruoxuan/go/src/github.com/pingcap/talent-plan/tidb/mergesort/mergesort.go
     840ms  6.25% 43.78%      840ms  6.25%  github.com/pingcap/talent-plan/tidb/mergesort.BenchmarkNormalSort.func1 /Users/wangruoxuan/go/src/github.com/pingcap/talent-plan/tidb/mergesort/bench_test.go
     560ms  4.17% 47.95%     1220ms  9.08%  github.com/pingcap/talent-plan/tidb/mergesort.merge /Users/wangruoxuan/go/src/github.com/pingcap/talent-plan/tidb/mergesort/mergesort.go
     460ms  3.43% 51.38%      460ms  3.43%  runtime.memclrNoHeapPointers /usr/local/Cellar/go/1.9.1/libexec/src/runtime/memclr_amd64.s
     340ms  2.53% 53.91%      340ms  2.53%  runtime.duffcopy /usr/local/Cellar/go/1.9.1/libexec/src/runtime/duff_amd64.s
     320ms  2.38% 56.29%      320ms  2.38%  reflect.Swapper.func5 /usr/local/Cellar/go/1.9.1/libexec/src/reflect/swapper.go
     270ms  2.01% 58.30%     1170ms  8.71%  runtime.gentraceback /usr/local/Cellar/go/1.9.1/libexec/src/runtime/traceback.go
     250ms  1.86% 60.16%      500ms  3.72%  runtime.scanobject /usr/local/Cellar/go/1.9.1/libexec/src/runtime/mgcmark.go
(pprof)
```

输出成svg图截取部分如下，很直观明了的看到golang内置的sort排序，主要cpu消耗在快排的关键Pivot函数上面，一切是符合预期的，当然标准库一般没什么问题。



2. K==1 merge sort profile

总耗时如下图，可以看到总耗时异常的高**165s**。

```
➜ mergesort git:(master) ✗ go test -bench BenchmarkMergeSort -cpuprofile mergesort_1.out
OK: 1 passed
1 2 3 4 5 6 9 goos: darwin
goarch: amd64
pkg: github.com/pingcap/talent-plan/tidb/mergesort
BenchmarkMergeSort-4           1        158076396429 ns/op
PASS
ok      github.com/pingcap/talent-plan/tidb/mergesort   165.344s
```

仅pprof分析，由下图可以看到各个入口的累计耗时都要比内置sort高很多，更直观的输出svg也可以看到结果

```
Showing top 10 nodes out of 74
      flat  flat%   sum%        cum   cum%
   159.22s 45.95% 45.95%    159.22s 45.95%  runtime/pprof.lostProfileEvent /usr/local/Cellar/go/1.9.1/libexec/src/runtime/pprof/proto.g
o
     0.25s 0.072% 46.02%    124.40s 35.90%  github.com/pingcap/talent-plan/tidb/mergesort.mergeSort /Users/wangruoxuan/go/src/github.co
m/pingcap/talent-plan/tidb/mergesort/mergesort.go
     0.14s  0.04% 46.06%     92.54s 26.71%  runtime.systemstack /usr/local/Cellar/go/1.9.1/libexec/src/runtime/asm_amd64.s
     6.61s  1.91% 47.97%     82.49s 23.81%  github.com/pingcap/talent-plan/tidb/mergesort.mergeSort.func1 /Users/wangruoxuan/go/src/git
hub.com/pingcap/talent-plan/tidb/mergesort/mergesort.go
    10.05s  2.90% 50.87%     72.71s 20.98%  github.com/pingcap/talent-plan/tidb/mergesort.mergeSort.func2 /Users/wangruoxuan/go/src/git
hub.com/pingcap/talent-plan/tidb/mergesort/mergesort.go
     0.67s  0.19% 51.06%     72.37s 20.88%  runtime.mallocgc /usr/local/Cellar/go/1.9.1/libexec/src/runtime/malloc.go
     0.06s 0.017% 51.08%     51.54s 14.87%  runtime.newobject /usr/local/Cellar/go/1.9.1/libexec/src/runtime/malloc.go
     0.15s 0.043% 51.12%     50.52s 14.58%  runtime.gcAssistAlloc /usr/local/Cellar/go/1.9.1/libexec/src/runtime/mgcmark.go
     0.02s 0.0058% 51.13%     40.84s 11.79%  runtime.markroot.func1 /usr/local/Cellar/go/1.9.1/libexec/src/runtime/mgcmark.go
     1.13s  0.33% 51.45%     40.82s 11.78%  runtime.scang /usr/local/Cellar/go/1.9.1/libexec/src/runtime/proc.go
```

svg图不便展示，可以看到主要耗时由两部分：`runtime mallocgc 72s`，`runtime system stack 92s`, 主要原因是16M（2^24）个数据，会开启个1+2+4+ 2^3 ... + 2^{24} 个即 2^{25} - 1个线程，也会造成频繁的gc。虽说goroutine非常高效但是如此庞大数量的goroutine，势必会导致过度的系统调用和gc，并不能充分利用多核优势，程序的消耗主要浪费在了线程调度，goroutine调度上面，因此程序设计不合理。

3. 混合排序优化

定义最优解：使用之前介绍的混合排序优化，在机器：4c8g/darwin的配置下，2^{24} 的数据量下求的最优的K使得排序耗时最短。

首先在开始求最优解之前，我们先设置K=1000进行验证，耗时**4.62s**，目前已经可以说明问题了，使用混合排序优化已经有了很大的提升。

结合svg图, 可以看到系统用于`runtime mallocgc 1.52s runtime system stack 1.56s`有明显的下降，且使用了内置sort, `sort doPiovt_func 0.93s`。

```
➜  mergesort git:(master) ✗ go test -bench BenchmarkMergeSort -cpuprofile mergesort_1000.out
OK: 1 passed
1 2 3 4 5 6 9 goos: darwin
goarch: amd64
pkg: github.com/pingcap/talent-plan/tidb/mergesort
BenchmarkMergeSort-4           1        2356991531 ns/op
PASS
ok      github.com/pingcap/talent-plan/tidb/mergesort   4.621s
```

求最优解思路：K从特定的start开始设置不同的step步长，计算每次的特定的K下的耗时，总的耗时曲线应该是先减后增，最优解即为波谷。

具体实现如下：

```
func FindOptimalK(start, step int) int {
  numElements := 16 << 20
  src := make([]int64, numElements)
  original := make([]int64, numElements)
  prepare(original)

  min := 163 * time.Second
  res:=start

  for k := start; k < numElements; k += step {
    st := time.Now()
    Thresh = k // 更新K
    copy(src, original)
    MergeSort(src)
    cost := time.Now().Sub(st)
    fmt.Printf("%d cost: %.2fs\n", Thresh, cost.Seconds())
    if cost < min {
      min = cost
      res = k //选取波谷的位置即为最优解
    } else {

    }
  }
  return res
```

```
}
```

结果如下：

```
1000 cost: 2.35s
101000 cost: 1.68s
201000 cost: 1.65s
301000 cost: 1.58s
401000 cost: 1.58s
501000 cost: 1.53s
601000 cost: 1.59s
701000 cost: 1.53s
801000 cost: 1.58s
901000 cost: 1.57s
1001000 cost: 1.61s
1101000 cost: 1.64s
1201000 cost: 1.55s
1301000 cost: 1.59s
1401000 cost: 1.56s
1501000 cost: 1.61s
1601000 cost: 1.67s
1701000 cost: 1.63s
1801000 cost: 1.62s
1901000 cost: 1.52s
2001000 cost: 1.63s
1001000 cost: 1.65s
2001000 cost: 1.74s
3001000 cost: 1.58s
4001000 cost: 1.57s
5001000 cost: 1.65s
6001000 cost: 1.61s
7001000 cost: 1.55s
8001000 cost: 1.62s
9001000 cost: 2.07s
10001000 cost: 2.05s
11001000 cost: 2.04s
12001000 cost: 2.04s
13001000 cost: 2.05s
14001000 cost: 2.06s
15001000 cost: 2.05s
16001000 cost: 2.18s
Result: 7001000--- PASS: TestFindOptimalK (32.49s)
PASS
```

总结

理想很美好，由于线程调度，等一些随机性的原因，最优解K很难找到，不过可以暂时认定一个阈值，在当前数量级上面，10w~1000w的k都能获得非常不错的性能。从svg图上面可以看到对比k=100000.svg,k=10000000.svg。

Reference

- youtube Go Profiling and Optimization