**Go performance tool**

Go has a lot of performance tool available for the CPU utilization and time usage.

The common tool is

<https://gitlab.com/steveazz-blog/go-performance-tools-cheat-sheet>

One of the most convinient method to see is use benchmark tool built in go

go test -bench=. -test.benchmem  ./rand/

goos: darwin

goarch: amd64

pkg: gitlab.com/steveazz/blog/go-performance-tools-cheat-sheet/rand

cpu: Intel(R) Core(TM) i7-6820HQ CPU @ 2.70GHz

BenchmarkHitCount100-8              3020            367016 ns/op          269861 B/op       3600 allocs/op

BenchmarkHitCount1000-8              326           3737517 ns/op         2696308 B/op      36005 allocs/op

BenchmarkHitCount100000-8              3         370797178 ns/op        269406189 B/op   3600563 allocs/op

BenchmarkHitCount1000000-8             1        3857843580 ns/op        2697160640 B/op 36006111 allocs/op

PASS

ok      gitlab.com/steveazz/blog/go-performance-tools-cheat-sheet/rand 8.828s

Note: -test.benchmem is an optional flag to show memory allocations

**Comparing Benchmarks**

Go created [perf](https://github.com/golang/perf) which provides [benchstat](https://github.com/golang/perf/tree/master/cmd/benchstat) so that you can compare to benchmark outputs together and it will give you the delta between them.

For example, let’s compare the [main](https://gitlab.com/steveazz-blog/go-performance-tools-cheat-sheet/-/tree/main) and [best](https://gitlab.com/steveazz-blog/go-performance-tools-cheat-sheet/-/tree/best) branches.

# Run benchmarks on `main`

git checkout main

go test -bench=. -test.benchmem -count=5 ./rand/ > old.txt

# Run benchmarks on `best

git checkout best

go test -bench=. -test.benchmem -count=5 ./rand/ > new.txt

# Compare the two benchmark results

benchstat old.txt new.txt

name               old time/op    new time/op    delta

HitCount100-8         366µs ± 0%     103µs ± 0%  -71.89%  (p=0.008 n=5+5)

HitCount1000-8       3.66ms ± 0%    1.06ms ± 5%  -71.13%  (p=0.008 n=5+5)

HitCount100000-8      367ms ± 0%     104ms ± 1%  -71.70%  (p=0.008 n=5+5)

HitCount1000000-8     3.66s ± 0%     1.03s ± 1%  -71.84%  (p=0.016 n=4+5)

name               old alloc/op   new alloc/op   delta

HitCount100-8         270kB ± 0%      53kB ± 0%  -80.36%  (p=0.008 n=5+5)

HitCount1000-8       2.70MB ± 0%    0.53MB ± 0%  -80.39%  (p=0.008 n=5+5)

HitCount100000-8      270MB ± 0%      53MB ± 0%  -80.38%  (p=0.008 n=5+5)

HitCount1000000-8    2.70GB ± 0%    0.53GB ± 0%  -80.39%  (p=0.016 n=4+5)

name               old allocs/op  new allocs/op  delta

HitCount100-8         3.60k ± 0%     1.50k ± 0%  -58.33%  (p=0.008 n=5+5)

HitCount1000-8        36.0k ± 0%     15.0k ± 0%  -58.34%  (p=0.008 n=5+5)

HitCount100000-8      3.60M ± 0%     1.50M ± 0%  -58.34%  (p=0.008 n=5+5)

HitCount1000000-8     36.0M ± 0%     15.0M ± 0%  -58.34%  (p=0.008 n=5+5)

Notice that we pass the -count flag to run the benchmarks multiple times so it can get the mean of the runs.

**Benchmarks**

You can generate profiles using benchmarks that we have in the demo project.

CPU:

go test -bench=. -cpuprofile cpu.prof ./rand/

Memory:

go test -bench=. -memprofile mem.prof ./rand/

**Go Static code analysis**

Static code analysis is the greatest tool to find the issues related to the security, performance, coverage, coding style, and some time even logic running without the running your application