Investigating Performance

This lesson explains the evaluation criteria of the performance of a Go program using Go tools, which includes time and memory utilization.

WE'LL COVER THE FOLLOWING

Time and memory consumption

A handy script to measure memory and consumption on a Unix-like systems is `xtime`:

```
#!/bin/sh
/usr/bin/time -f '%Uu %Ss %er %MkB %C' "$@"
```

Used on the command-line as `xtime progexec`, where `prog` is a Go executable program, it shows an output like:

```
56.63u 0.26s 56.92r 1642640kB progexec
```

giving the *user time*, *system time*, *real time* and *maximum memory usage*, respectively.

Tuning with `go test`

If the code uses the Go testing package's benchmarking support, we could use the `go test` standard `-cpuprofile` and `-memprofile` flags, causing a CPU or

memory usage profile to be written to the file specified.

```
go test -cpuprofile cpu.prof -memprofile mem.prof -bench
```

This compiles and executes the tests and benchmarks in the current folder. It also writes a CPU profile to **cpu.prof** and a memory profile to **mem.prof**.

Tuning with `pprof`

For a standalone program `progexec`, you have to enable profiling by importing `runtime/pprof`; this package writes the runtime profiling data in the format expected by the `pprof` visualization tool. For CPU profiling, you have to add a few lines of code:

```
var cpuprofile = flag.String("cpuprofile", "", "write cpu profile to file"
)

func main() {
  flag.Parse()
  if *cpuprofile != "" {
    f, err := os.Create(*cpuprofile)
    if err != nil {
      log.Fatal(err)
    }
    defer f.close()
    pprof.StartCPUProfile(f)
    defer pprof.StopCPUProfile()
  }
  ...
}
```

This code defines a flag named `cpuprofile`, calls the Go flag library to parse the command-line flags, and then, if the `cpuprofile` flag has been set on the command line, starts CPU profiling redirected to that file (`os.Create` makes a file with the given name in which the profile will be written). The profiler requires a final call to `StopCPUProfile` to flush any pending writes to the file before the program exits; we use defer to make sure this happens as main returns. Now, run the program with this flag: `progexec -cpuprofile=progexec.prof`, and then you can visualize the profile with:

```
go tool pprof progexec.prof
```

When CPU profiling is enabled, the Go program stops about 100 times per second and records a sample consisting of the program counters on the currently executing Go routine's stack. Some of the interesting commands of this tool are:

- top
- web or web funcname
- list funcname or weblist funcname

top

This shows the 10 most heavily used functions during the execution, an output like:

```
Total: 3099 samples
     626 20.2% 20.2%    626 20.2% scanblock
     309 10.0% 30.2%    2839 91.6% main.FindLoops
...
```

The 5th column is an indicator of how heavy that function is used.

web or web funcname

This command writes a graph of the profile data in SVG format and opens it in a web browser (there is also a `gv` command that writes *PostScript* and opens it in *Ghostview*. For either command, you need `graphviz` installed). The different functions are shown in rectangles (the more it is called the bigger), and the arrows point in the direction of the function calls.

list funcname or weblist funcname

This shows a listing of the code lines in funcname, within the 2nd column the time spent in executing that line, so this gives a very good indication of what code is heaviest in the execution. If you see that the function `runtime.mallocgc` (which both allocates and runs periodic garbage collections) is heavily used, then it is time for memory profiling. To find out why the garbage collector is running so much, we have to find out which code is allocating memory.

For this, the following code has to be added at a judicious place:

```
var memprofile = flag.String("memprofile", "", "write memory profile to th
is file")
...

CallToFunctionWhichAllocatesLotsOfMemory()
if *memprofile != "" {
  f, err := os.Create(*memprofile)
  if err != nil {
    log.Fatal(err)
  }
  pprof.WriteHeapProfile(f)
  f.Close()
  return
}
```

Now, run the program with this flag: `progexec -memprofile=progexec.mprof` and visualize it with:

```
go tool pprof progexec.mprof
```

The same commands apply (top, list funcname etc), but now they measure memory allocation in Mb. Here is a sample output of top:

```
Total: 118.3 MB
     66.1 55.8% 55.8%     103.7 87.7% main.FindLoops
     30.5 25.8% 81.6%     30.5 25.8% main.*LSG·NewLoop
...
```

The topmost functions use the most memory, as seen in the 1st column.

For web applications running on host: port, there is also a standard HTTP interface to profiling data. The following command will start making HTTP requests on the specified port:

```
pprof -http=[host]:[port] source
```

Open a browser using this http://host:port to visualize the interface data. More info can be found here.

You now know how to write a less error prone program, how to catch errors

You now know how to write a less error-prone program, how to catch errors in case a program panics, and how to test a program for performance. There is a quiz on what you have learned in the next lesson.