Learning Objectives

Learners will be able to…

- **Log in Go programs**
- **Profile a Go program using the web interface**

info

Make Sure You Know

Python or other modern programming language.

Logging

Standard practice in a project is to have logs - messages that show various information about errors, variable states, etc. Logs make it clear what went wrong and make it easier to understand the whole situation.

Consider the `log` package, which is standard for use in Go.

Package `log` implements a simple logging package. It defines a type, Logger, with methods for formatting output. It also has a predefined 'standard' Logger accessible through helper functions Print[f|ln], Fatal[f|ln], and Panic[f|ln], which are easier to use than creating a Logger manually. That logger writes to standard error and prints the date and time of each logged message. Every log message is output on a separate line: if the message being printed does not end in a newline, the logger will add one. The Fatal functions call os.Exit(1) after writing the log message. The Panic functions call panic after writing the log message.

`New` creates a new Logger. The out variable sets the destination to which log data will be written. The prefix appears at the beginning of each generated log line, or after the log header if the Lmsgprefix flag is provided. The flag argument defines the logging properties.

New loggers created with log.New() are concurrently safe. You can share one logger and use it in several goroutines without worrying about possible conflicts between them due to recording messages in the same logger.
If you have multiple loggers using the same write destination, you need to ensure that the underlying Write() method is also safe for concurrent use.

Using the Panic() and Fatal() methods outside of the main() function is best avoided. Instead, it is recommended to return errors that occurred and panic or force terminate the application directly from main() itself.

```go
package main

import (
    "log"
    "os"
)

func main() {
    loggerDebug := log.New(os.Stdout, "[DEBUG]\t",
        log.Ldate|log.Ltime|log.Lshortfile)
    loggerDebug.Print("debug message")
}
```

In this example, we specified the following flags:
`log.Ldate` - the date in the local time zone
`log.Ltime` - the time in the local time zone
`log.Lshortfile` - final file name element and line number

Let's run this program and the output will be:

```
[DEBUG] 2023/09/26 23:18:35 logging.go:20: debug message
```

Logging to file

When creating the logger, we set the destination where the log data will be written, implementing the `io.Writer` interface.
So for logging we can use a file.
Next, we can create two loggers that will write this data to the files `info.log` and `error.log` respectively.

```
func main() {
    logInfoFile, _ := os.OpenFile("info.log",
        os.O_RDWR|os.O_CREATE|os.O_APPEND, 0666)
    loggerInfo := log.New(logInfoFile, "[INFO]\t",
        log.Ldate|log.Ltime)
    defer logInfoFile.Close()

    logErrorFile, _ := os.OpenFile("error.log",
        os.O_RDWR|os.O_CREATE|os.O_APPEND, 0666)
    loggerError := log.New(logErrorFile, "[ERROR]\t",
        log.Ldate|log.Ltime|log.Lshortfile)
    defer logErrorFile.Close()

    loggerInfo.Print("info message")
    loggerError.Print("error message")
}
}
```

Also use buffered I/O whenever possible. This will reduce the number of system calls.
It is usually not necessary to write every logger call to a file - use the bufio package to implement buffered I/O.

```
logInfoFile, _ := os.OpenFile("info.log",
os.O_RDWR|os.O_CREATE|os.O_APPEND, 0666)
loggerInfo := log.New(bufio.NewWriterSize(logInfoFile, 1024*16),
"[INFO]\t", log.Ldate|log.Ltime)
defer logInfo.Close()
```

Profiling

Profiling is the collection of program performance characteristics, such as execution time of individual fragments, memory usage, etc. The tool used to analyze work is called a profiler or profiler.
To understand why a program does not work fast enough, uses too much memory, or uses the processor suboptimally, it is necessary to use profiling.

Go has a powerful built-in profiler that supports profiling of CPU, memory, goroutines, and locks.
You can note such profiling packages “runtime/pprof” and “net/http/pprof” included in the standard distribution of the language.

Consider the package “net/http/pprof”.
All you need to connect the profiler is to import net/http/pprof. the necessary HTTP handlers will be registered automatically when the package is imported (see init() in this package).
Or you will need to manually register several pprof addresses.

```
func main() {
    r := http.NewServeMux()

    r.HandleFunc("/debug/pprof/", pprof.Index)
    r.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
    r.HandleFunc("/debug/pprof/profile", pprof.Profile)
    r.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
    r.HandleFunc("/debug/pprof/trace", pprof.Trace)

    http.ListenAndServe(":8080", r)
}
```

Suboptimal solution example

Let’s write a simple service that will convert a pre-generated `rune` array into a string.

```
package main

import (
    "bytes"
    "fmt"
    "math/rand"
    "net/http"

    "time"

    _ "net/http/pprof"
)

var runeArray []rune

func main() {
    runeArray = generateRuneArray(10000)
    http.HandleFunc("/testProfiling", makeString)
    http.ListenAndServe("localhost:8080", nil)
}

func makeString(w http.ResponseWriter, r *http.Request) {
    var s string
    for _, i := range runeArray {
        s = string(i) + s
    }
    out := fmt.Sprintf("len: %d", len(s))

    fmt.Fprint(w, out)
}

func generateRuneArray(length int) []rune {
    rand.Seed(time.Now().UnixNano())
    dictionary := "abcdefghijklmnopqrstuvwxyz"
    buf := make([]rune, length)
    for i := 0; i < length; i++ {
        buf[i] = rune(dictionary[rand.Intn(len(dictionary))])
    }
    return buf
}
```

Now we can start testing our application.
To do this, we will use the `ab` utility (Apache benchmarking tool).

Let’s launch our application.

In a new terminal, execute `ab -k -c 8 -n 5000 "localhost:8080/testProfiling"`, where `-n 5000` is the number of requests.

Immediately after launching in another terminal, let's run the profiler `go tool pprof ./bin/profiling-example http://127.0.0.1:8080/debug/pprof/profile`, where `./bin/profiling-example` is the path to our compiled applications.

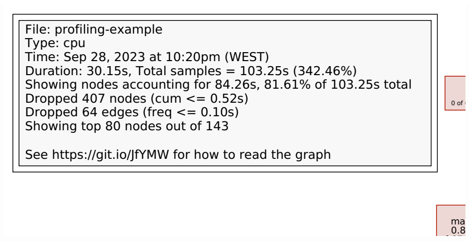
In the output of the `ab` utility we will find the line `Requests per second`. Let's remember this value and focus on it.

```
Requests per second:    93.07 [#/sec] (mean)
```

Now let's go back to `go tool pprof`.
It uses sampling to determine which functions are spending the most CPU time. Go runtime stops execution every ten milliseconds and records the current call stack of all running goroutines. The CPU profiler runs for 30 seconds by default.
After some time, pprof will go into interactive mode.
For help, use `help`.
`top` - to see a list of functions that, as a percentage, were most present in the resulting sample.
In our case we will see something similar:

```
Showing nodes accounting for 61.07s, 59.15% of 103.25s total
Dropped 407 nodes (cum <= 0.52s)
Showing top 10 nodes out of 143
      flat  flat%   sum%        cum   cum%
    29.97s 29.03% 29.03%     29.97s 29.03%  runtime.memmove
     9.16s  8.87% 37.90%      9.16s  8.87%  runtime.nanotime
     5.41s  5.24% 43.14%     12.06s 11.68%  runtime.scanobject
     4.68s  4.53% 47.67%      4.68s  4.53%  runtime.futex
     2.92s  2.83% 50.50%      2.92s  2.83%  runtime.madvise
     2.33s  2.26% 52.76%      3.43s  3.32%  runtime.scanblock
     1.89s  1.83% 54.59%      1.89s  1.83%
        runtime.heapBits.nextFast (inline)
     1.62s  1.57% 56.15%      2.94s  2.85%  runtime.findObject
     1.60s  1.55% 57.70%      1.69s  1.64%
        runtime.writeHeapBits.flush
     1.49s  1.44% 59.15%      1.49s  1.44%  runtime.procyield
```

You can see this more clearly using the `web` command (you may need to install `apt install graphviz`). It generates a call graph in SVG format and opens it in a web browser.



File: profiling-example
Type: cpu
Time: Sep 28, 2023 at 10:20pm (WEST)
Duration: 30.15s, Total samples = 103.25s (342.46%)
Showing nodes accounting for 84.26s, 81.61% of 103.25s total
Dropped 407 nodes (cum <= 0.52s)
Dropped 64 edges (freq <= 0.10s)
Showing top 80 nodes out of 143

See https://git.io/JfYMW for how to read the graph

pprof

You can also use a separate function in more detail using the `list` command.
We use the command `list makeString` After which we get something like this output:

```
Total: 103.25s
ROUTINE ======================== main.makeString in
        /home/.../profiling-example/cmd/main.go
     840ms     67.41s (flat, cum) 65.29% of Total
         .          .     22:func makeString(w
        http.ResponseWriter, r *http.Request) {
         .          .     23:   var s string
     270ms      270ms     24:   for _, i := range runeArray {
     480ms     66.94s     25:       s = string(i) + s
         .          .     26:   }
      80ms      170ms     27:   out := fmt.Sprintf("len: %d",
        len(s))
         .          .     28:
      10ms       30ms     29:   fmt.Fprint(w, out)
         .          .     30:}
         .          .     31:
```

Further optimization and checking

After analyzing the data from `pprof` we can conclude that most of the resources are spent on the `makeString` function.
Also pay attention to the line `480ms 66.94s 25: s = string(i) + s`.
Since each iteration of the loop results in a new line being allocated in memory, we see that in our case this is where we have the highest costs.
Clearly, using `bytes.Buffer{}` would be a better solution in our case.

Let's rewrite the `makeString` function as follows:

```
func makeString(w http.ResponseWriter, r *http.Request) {
    var s string
    buf := bytes.Buffer{}
    for _, i := range runeArray {
        buf.WriteRune(i)
    }
    buf.WriteString(s)
    s = buf.String()
    out := fmt.Sprintf("len: %d", len(s))

    fmt.Fprint(w, out)
}
```

Then we will compile and run our program again and run the utility `ab -k -c 8 -n 5000 "localhost:8080/testProfiling"`.
Now the `Requests per second` indicator is much higher.

```
Requests per second: 20667.65 [#/sec] (mean)
```

Let's remember that in the previous version of our application we had 93.07.
Thus, profiling is a good way to find out the real performance of your application and find bottlenecks that need to be optimized.