Highlights of Go 1.1

May 29, 2013

John Graham-Cumming

Method Values

- Function value bound to specific receiver

```
type Prefixer struct {
  prefix string
}

func (p Prefixer) Add(s string) string {
  return p.prefix + ": " + s
}

func main() {
  hawaii := Prefixer{"Aloha”}
  fmt.Printf("%s\n", hawaii.Add("Welcome to Honolulu"))

  adder := hawaii.Add
  fmt.Printf("%s\n", adder("Welcome to Honolulu"))
}
```

Change to `return` handling

- `return` statement not needed at end of functions if function termination is unambiguous

```
func even() int {
  for {
    if i := rand.Intn(100); i%2 == 0 {
      return i
    }
  }
}
```

- Worth reading the specification on ‘terminating statements’
- http://golang.org/ref/spec#Terminating_statements

bufio.Scanner

- Simple, fast type for doing command tasks like reading `os.Stdin` line by line, or reading word by word from a file

```
scanner := bufio.NewScanner(os.Stdin)

for scanner.Scan() {
    fmt.Println(scanner.Text()
}

if err := scanner.Err(); err != nil {
    // Handle error
}
```

- Built in scanners for lines, words, characters and runes
- Can provide a custom scanner function

Size of `int`

- On 64-bit platforms `int` and `uint` are now 64 bits
  - Elsewhere they are 32-bits.

- Couple of consequences:
  - If your code was relying on them being 32-bits then you may have trouble

```
x := ^uint32(0)     // x is 0xffffffff
i := int(x)         // i is -1 on 32-bit systems,
                    // 0xffffffff on 64-bit

fmt.Println(i)
```

  - Slice indexes are `int`s which means they can have 2 billion members

Heap size and Platforms

- On 64-bit machines heap can now be tens of GB
- No change on 32-bit
- If you were running off tip in recent months you already had massive heaps

- Experimental support for: `linux/arm`, `freebsd/arm`, `netbsd/386`, `amd64` and `arm`, `openbsd/386` and `amd64`

Nanosecond timing

- FreeBSD, Linux, NetBSD, OpenBSD, OS X `time` package has nanosecond precision
- `time.Round` and `time.Truncate` to round up/down to nearest multiples of any `time.Duration`

```
t, _ := time.Parse("2006 Jan 02 15:04:05", "2012 Dec
07 12:15:30.918273645")
trunc := []time.Duration{
  time.Nanosecond, time.Microsecond,
  time.Millisecond, time.Second,
  2 * time.Second, time.Minute,
  10 * time.Minute, time.Hour,
}
for _, d := range trunc {
  fmt.Printf("t.Truncate(%6s) = %s\n", d,
t.Truncate(d).Format("15:04:05.999999999"))
}
```

Performance

- Claimed 30 to 40% performance increase over Go 1.0.3
- Best analysis by Dave Cheney:
  - http://dave.cheney.net/2013/05/21/go-11-performance-improvements
  - http://dave.cheney.net/2013/05/25/go-11-performance-improvements-part-2
  - http://dave.cheney.net/2013/05/28/go-11-performance-improvements-part-3
- Dave’s summary: 30-40% performance increase is real

Performance Highlights

- Code generation improvements across all three gc compilers
- Improvements to inlining
- Reduction in stack usage
- Parallel garbage collector.
- More precise garbage collection, which reduces the size of the heap, leading to lower GC pause times.
- New runtime scheduler; tight integration of the scheduler with the net package
- Parts of the runtime and standard library have been rewritten in assembly to take advantage of specific bulk move or crypto instructions.

Race Detector

- http://golang.org/doc/articles/race_detector.html
- Detects concurrent access by two goroutines to the same variable where one access is write

```
func main() {
  c := make(chan bool)
  m := make(map[string]string)

  go func() {
    m["1"] = "a" // First conflicting access.
    c <- true
  }()

  m["2"] = "b" // Second conflicting access.
  <-c
}
```

- `go build -race example.go`

Full release notes

- http://golang.org/doc/go1.1
- Go 1.1 is backwards compatible with Go 1.0.3
- Go 1.2 targeted for December 1