Go on NetBSD

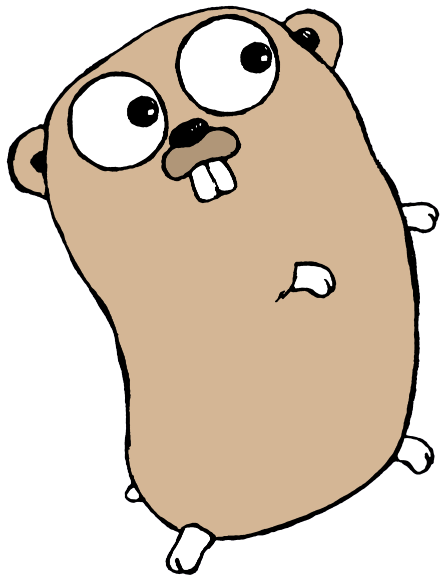
A modern systems programming language

2 February 2013

Benny Siegert
Google Switzerland; The NetBSD Foundation

Agenda

- What is Go?
- Building Go code with the go tool
- Running Go code
- pkgsrc
- Conclusion



What is Go?

A modern systems programming language

Initially developed at Google, open source since 2009.

Initial implementation by Rob Pike, Robert Griesemer, Russ Cox, Ken Thompson.

- compiled
- mostly statically typed
- garbage collected
- provides control over memory layout
- provides access to C APIs (via cgo) and syscalls

Go has powerful concurrency primitives.

Go is:

Simple: concepts are easy to understand

- (the implementation might still be sophisticated)

Orthogonal: concepts mix clearly

- easy to understand and predict what happens

Succinct: no need to predeclare every intention

Safe: misbehavior should be detected

These combine to give expressiveness.

(Source: R. Pike, The Expressiveness of Go (2010), http://talks.golang.org/2010/ExpressivenessOfGo-2010.pdf (http://talks.golang.org/2010/ExpressivenessOfGo-2010.pdf))

Clean

The language is defined by a short and readable **specification**. Read it.

- implemented by two compilers: gc and gccgo (gcc frontend).

The APIs in the standard library are well thought out,
contrary to the "bureaucracy" of C++ or Java:

```
foo::Foo *myFoo = new foo::Foo(foo::FOO_INIT)
```

- but in the original Foo was a longer word

The standard library has "batteries included".

The code has a **standard formatting**, enforced by `gofmt`.
No more discussions about braces and indentation!

Hello World

```
package main

import "fmt"

func main() {
    fmt.Println("Hello World!")
}
```

Run

All code lives in a package (package main is a command).

Semicolons are inserted automatically.

- Opening brace for functions must go on the same line.

Strings are UTF-8, built-in string data type.

Another Hello World

```go
package main

import (
    "flag"
    "fmt"
    "net/http"
)

var addr *string = flag.String("addr", ":8080", "host:port to listen on")

func main() {
    flag.Parse()

    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello World!")
    })

    http.ListenAndServe(*addr, nil)
}
```

Run

`net/http` is not a toy web server! It powers e.g. dl.google.com.

Basic data structures: slices and maps

Slices are a form of dynamic arrays.

```
    a := []int{1, 2, 3, 4} // len(a) = 4, cap(a) = 4
    b := a[2:4]            // b[0] = 3, b[1] = 4

    b = append(b, 5) // b[2] = 5
    _ = b[3]         // out-of-bounds access
```

Run

Strings are immutable; they can be converted to `[]byte` or `[]rune`.

Type-safe hashtables (maps) are built-in.

```
translations := make(map[string]string)
translations["Hello"] = "Bonjour"
```

Object orientation

Objects in Go do not work like they do in C++.
No inheritance, no polymorphy.

They are more similar to objects in Perl 5.
You start from a basic type (struct, int, string, ...) and add methods.

```
package foo

type Number int

func (n Number) Square() Number {
    return n * n
}
```

Methods have a receiver before the name (often a pointer).

Table-driven testing

```go
package foo

import "testing"

var squareTests = []struct {
    num, square Number
}{
    {1, 1},
    {2, 4},
    {256, 65536},
    {-10, 100},
}

func TestSquare(t *testing.T) {
    for _, test := range squareTests {
        actual := test.num.Square()
        if actual != test.square {
            t.Errorf("Square() of %v: got %v, want %v",
                test.num, actual, test.square)
        }
    }
}
```

Table-driven tests (2)

Here is the test run:

```
$ go test
PASS
ok      github.com/bsiegert/talks/go-netbsd/object      0.004s
```

If I deliberately insert a mistake:

```
$ go test
--- FAIL: TestSquare (0.00 seconds)
object_test.go:21:      Square() of -10: got 100, want -100
FAIL
exit status 1
FAIL    github.com/bsiegert/talks/go-netbsd/object      0.004s
```

Finally, useful diagnostics!

Interfaces

Interfaces work on methods, not on data.

```
type Reader interface {
        Read(p []byte) (n int, err error)
}

type Writer interface {
        Write(p []byte) (n int, err error)
}

type ReadWriter interface {
        Reader
        Writer
}
```

Any type that implements these methods fulfills the interface *implicitly* (i.e. no "implements" declarations).

Use the interface instead of a concrete type, e.g. in a function:

```
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
```

Concurrency: goroutines

A goroutine is a sort of lightweight thread. It runs in the same address space, concurrently and independent from the other goroutines.

```
f("Hello World") // f runs; we wait

go f("Hello World") // execution continues while f is running
```

They are much cheaper than threads, you can have thousands of them.

If one goroutine blocks (e.g. on I/O), the others continue to run.
This is easier to reason about than I/O with callbacks, as in node.js.

Maximum number of goroutines running in parallel is configurable
(e.g. one per core).

Concurrency: channels

Channels are type-safe "pipes" to transfer data between goroutines.

"Don't communicate by sharing memory -- share memory by communicating."

They are also a synchronization point.

```
timer := make(chan bool)
go func() {
    time.Sleep(deltaT)
    timer <- true
}()
// Do something else; when ready, receive.
// Receive will block until timer delivers.
<-timer
```

Easily implement worker pools, parallelize computations, etc.

More information: R. Pike, "Concurrency is not parallelism", http://talks.golang.org/2012/waza.slide (http://talks.golang.org/2012/waza.slide).

"Self-documenting" code: godoc

godoc extracts and generates documentation for Go programs, using comments in the source code.

```
// Package strings implements simple functions to manipulate strings.
package strings

// Count counts the number of non-overlapping instances of sep in s.
func Count(s, sep string) int {
    // …
}
```

http://golang.org (http://golang.org) runs godoc on Google App Engine.

`godoc -http=:6060` runs the server locally.

`godoc foo` shows the documentation on the console (similar to a manpage).

Commands often have a doc.go containing only documentation.

Building Code With the go Tool

GOROOT and GOPATH

The default build tool is called go. It uses `$GOROOT` and `$GOPATH`.

- **GOROOT** contains the standard Go tree (source + compiled form).
- **GOPATH** is a colon-separated list of "user paths". It *must* be set by the developer.

Even after building, the source code is needed for godoc and for building dependent packages.

GOPATH example

GOPATH=/home/user/gocode

```
/home/user/gocode/
    src/
        myproject/
            foo/        (go code in package foo)
                x.go
            server/     (go code in package main)
                y.go
    bin/
        server          (installed command)
    pkg/
        netbsd_amd64/
            myproject/
                foo.a   (installed package object)
```

Conventions for remote repos

```
import (
    "code.google.com/p/go.image/tiff"
    "github.com/mattn/go-gtk"
    "launchpad.net/goamz/ec2"
)
```

Import path == URL (more or less). Supports free hosters and custom remote repositories.

`go get github.com/user/repo/package` installs these dependencies
(if you have git, hg, bzr installed).

- fetch, build, install
- supports recursive fetching of dependencies

Running Go code

A word on compilers

The gc suite is the most often used compiler suite. It compiles insanely fast.

- supports i386, amd64, arm
- Linux, FreeBSD, OpenBSD, NetBSD, Windows
- easy to cross-compile for other platforms

gccgo is a Go frontend for gcc, included in gcc 4.7.x.

- supports all platforms gcc supports
- better optimizations
- may not have the latest standard libraries
- has fewer users

Go packages

All Go code lives in a package.
Compiling a package main produces an executable binary.

Other packages are compiled to static libraries (.a files).

- contain code **and** the exported interface
- contain all dependencies
- .a files from different compilers (and different compiler versions) are incompatible.

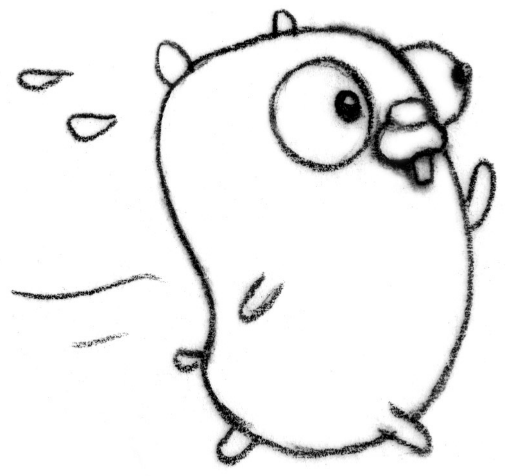
Running a server written in Go

Currently, Go programs cannot daemonize, so they run in the foreground.

- this is harder than it looks

My suggestion: run it under daemontools

- log on stdout, collect logs with multilog



Daemontools example: continuous builder

/service/builder/run contains:

```
#!/bin/sh
exec 2>&1
exec setuidgid builder envdir env /service/builder/builder -commit netbsd-amd64-bsiegert
```

/service/builder/env/HOME contains:

```
/home/builder
```

/service/builder/log/run contains:

```
#!/bin/sh
exec setuidgid buildlog multilog ./main
```

Just copy the executable to /service/builder/builder, done!

pkgsrc

wip/go

Go is available in pkgsrc-wip as `wip/go`.

It installs source + binaries to `$PREFIX/go`.

The package supports

- NetBSD
- Linux
- OpenBSD (untested)
- Mac OS X (currently broken)

on i386 and x86_64.

NetBSD support is not in a stable release yet. (Go 1.1 will have it.)

pkgsrc and software written in Go

Two cases: **libraries** and **executables**.

Executables are easy, as they are statically linked.

- may link dynamically against C libraries
- no *runtime* dependencies to Go libs
- need source (or part of it) for godoc

Libraries?

- have to be recompiled each time you upgrade the Go compiler
- why not make source-only packages and compile during postinstall?
- how to support building with gccgo?
- need to design a `go/package.mk`.

Conclusion

Conclusion

Try Go!
It does not look very revolutionary on first glance, but it is addictive.

Try Go on NetBSD.

Try wip/go and report any problems.

Thank you

Benny Siegert
Google Switzerland; The NetBSD Foundation
mailto:bsiegert@google.com (mailto:bsiegert@google.com)
mailto:bsiegert@netbsd.org (mailto:bsiegert@netbsd.org)
http://www.mirbsd.org/wlog-10.htm (http://www.mirbsd.org/wlog-10.htm)