The Go Programming Language

Rob Pike
golang.org
Oct 30, 2009

http://golang.org

Google

Go

New

Experimental

Concurrent

Garbage-collected

Systems

Language

Google

Hello, world

```
package main

import "fmt"

func main() {
   fmt.Printf("Hello, 世界\n");
}
```

Google

Who

Robert Griesemer, Ken Thompson, and Rob Pike started the project in late 2007.

By mid 2008 the language was mostly designed and the implementation (compiler, run-time) starting to work.

Ian Lance Taylor and Russ Cox joined in 2008.

Lots of help from many others.

Google

Why

Go fast!

Make programming fun again.

Google

Our changing world

No new major systems language in a decade.

But much has changed:

- sprawling libraries & dependency chains
- dominance of networking
- client/server focus
- massive clusters
- the rise of multi-core CPUs

Major systems languages were not designed with all these factors in mind.

Google

Construction speed

It takes too long to build software.

The tools are slow and are getting slower.

Dependencies are uncontrolled.

Machines have stopped getting faster.

Yet software still grows and grows.

If we stay as we are, before long software construction will be unbearably slow.

Google

Type system tyranny

Robert Griesemer: “Clumsy type systems drive people to dynamically typed languages.”

Clunky typing:

Taints good idea with bad implementation.

Makes programming harder (think of C's **`const`**: well-intentioned but awkward in practice).

Hierarchy is too stringent:

Types in large programs do not easily fall into hierarchies.

Programmers spend too much time deciding tree structure and rearranging inheritance.

You can be productive or safe, not both.

Google

Why a new language?

These problems are endemic and linguistic.

New libraries won't help. (Adding anything is going in the wrong direction.)

Need to start over, thinking about the way programs are written and constructed.

Google

A New Language

Google

Goals

The efficiency of a statically-typed compiled language with the ease of programming of a dynamic language.

Safety: type-safe and memory-safe.

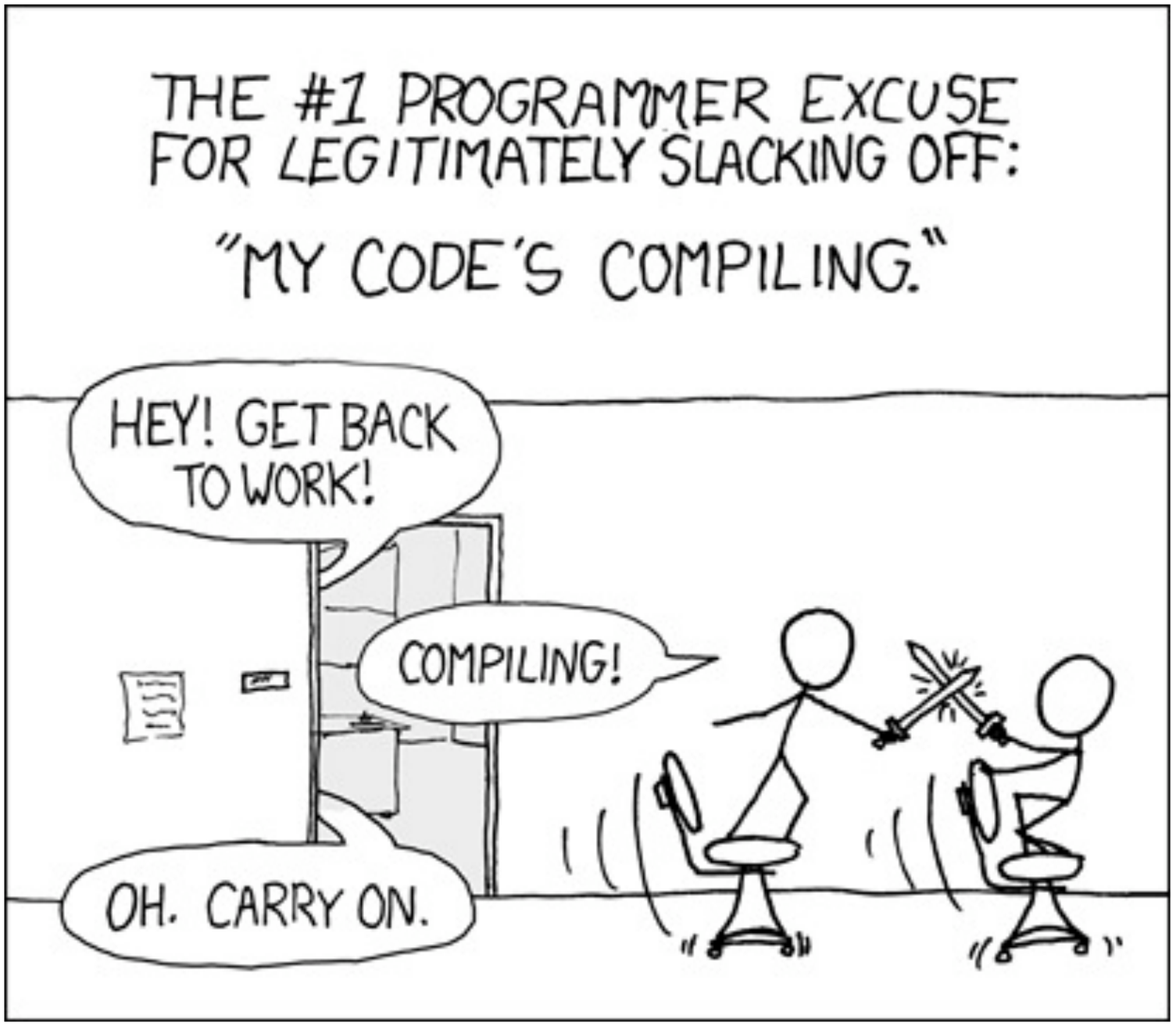
Good support for concurrency and communication.

Efficient, latency-free garbage collection.

High-speed compilation.

Google

As xkcd observes...



http://xkcd.com/303/

Google

Compilation demo

Google

Design principles

Keep concepts orthogonal.

A few orthogonal features work better than a lot of overlapping ones.

Keep the grammar regular and simple.

Few keywords, parsable without a symbol table.

Reduce typing. Let the language work things out.

No stuttering; don't want to see

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

Avoid bookkeeping.

But keep things safe.

Reduce typing. Keep the type system clear.

No type hierarchy. Too clumsy to write code by constructing type hierarchies.

It can still be object-oriented.

Google

The big picture

Fundamentals:

- Clean, concise syntax.
- Lightweight type system.
- No implicit conversions: keep things explicit.
- Untyped unsized constants: no more **`0x80ULL`**.
- Strict separation of interface and implementation.

Run-time:

- Garbage collection.
- Strings, maps, communication channels.
- Concurrency.

Package model:

- Explicit dependencies to enable faster builds.

Google

New approach: Dependencies

Construction speed depends on managing dependencies.

Explicit dependencies in source allow:

- fast compilation
- fast linking

The Go compiler pulls transitive dependency type info from the object file – but only what it needs.

If **`A.go`** depends on **`B.go`** depends on **`C.go`**:

- compile **`C.go`**, **`B.go`**, then **`A.go`**.
- to compile **`A.go`**, compiler reads **`B.o`** not **`C.o`**.

At scale, this can be a huge speedup.

Google

New approach: Concurrency

Go provides a way to write systems and servers as concurrent, garbage-collected processes (goroutines) with support from the language and run-time.

Language takes care of goroutine management, memory management.

Growing stacks, multiplexing of goroutines onto threads is done automatically.

Concurrency is hard without garbage collection.

Garbage collection is hard without the right language.

Google

... quickly

Google

Basics

```
const N = 1024  // just a number
const str = “this is a 日本語 string\n”

var x, y *float
var ch = '\u1234'

/* Define and use a type, T. */
type T struct { a, b int }
var t0 *T = new(T);
t1 := new(T);  // type taken from expr

// Control structures:
// (no parens, always braces)
if len(str) > 0 { ch = str[0] }
```

Google

Program structure

```
package main

import "os"
import "flag"

var nFlag = flag.Bool("n", false, `no \n`)

func main() {
    flag.Parse();
    s := "";
    for i := 0; i < flag.NArg(); i++ {
        if i > 0 { s += " " }
        s += flag.Arg(i)
    }
    if !*nFlag { s += "\n" }
    os.Stdout.WriteString(s);
}
```

Google

Constants

```
type TZ int
const (
    UTC TZ =  0*60*60;
    EST TZ = -5*60*60; // and so on
)

// iota enumerates:
const (
    bit0, mask0 uint32 = 1<<iota, 1<<iota - 1;
    bit1, mask1 uint32 = 1<<iota, 1<<iota - 1;
    bit2, mask2; // implicitly same text
)

// high precision:
const Ln2= 0.69314718055994530941723212145\
           817656807550013436025525412068000 9
const Log2E= 1/Ln2 // precise reciprocal
```

Google

Values and types

```
weekend := []string{ "Saturday", "Sunday" }

timeZones := map[string]TZ {
    "UTC":UTC, "EST":EST, "CST":CST, //...
}

func add(a, b int) int { return a+b }
```

```
type Op func (int, int) int

type RPC struct {
    a, b int;
    op Op;
    result *int;
}
rpc := RPC{ 1, 2, add, new(int) };
```

Google

Methods

```
type Point struct {
  X, Y float  // Upper case means exported
}

func (p *Point) Scale(s float) {
  p.X *= s; p.Y *= s;  // p is explicit
}
```

```
func (p *Point) Abs() float {
  return math.Sqrt(p.X*p.X + p.Y*p.Y)
}

x := &Point{ 3, 4 };
x.Scale(5);
```

Google

Methods for any user type

```
package main
import "fmt"

type TZ int
const (
 HOUR TZ = 60*60; UTC TZ = 0*HOUR; EST TZ = -5*HOUR; //...
)
var timeZones = map[string]TZ { "UTC": UTC, "EST": EST, }
func (tz TZ) String() string {  // Method on TZ (not ptr)
  for name, zone := range timeZones {
    if tz == zone { return name }
  }
  return fmt.Sprintf("%+d:%02d", tz/3600, (tz%3600)/60);
}
func main() {
  fmt.Println(EST);  // Print* know about method String()
  fmt.Println(5*HOUR/2);
}
// Output (two lines) EST +2:30
```

Google

Interfaces

```
type Magnitude interface {
  Abs() float;  // among other things
}

var m Magnitude;
m = x;  // x is type *Point, has method Abs()
mag := m.Abs();

type Point3 struct { X, Y, Z float }
func (p *Point3) Abs() float {
  return math.Sqrt(p.X*p.X + p.Y*p.Y + p.Z*p.Z)
}
m = &Point3{ 3, 4, 5 };
mag += m.Abs();

type Polar struct { R, Θ float }
func (p Polar) Abs() float { return p.R }
m = Polar{ 2.0, PI/2 };
mag += m.Abs();
```

Google

Interfaces for generality

Package **`io`** defines the **`Writer`** interface:

```
type Writer interface {
    Write(p []byte) (n int, err os.Error)
}
```

Any type with that method can be written to: files, pipes, network connections, buffers, ... On the other hand, anything that needs to write can just specify **`io.Writer`**.

For instance, **`fmt.Fprintf`** takes **`io.Writer`** as first argument.

For instance, **`bufio.NewWriter`** takes an **`io.Writer`** in, buffers it, satisfies **`io.Writer`** out.

And so on...

Google

Putting it together

```
package main

import (
   "bufio";
   "fmt";
   "os";
)

func main() {
   // unbuffered
   fmt.Fprintf(os.Stdout, "%s, ", "hello");
   // buffered: os.Stdout implements io.Writer
   buf := bufio.NewWriter(os.Stdout);
   // and now so does buf.
   fmt.Fprintf(buf, "%s\n", "world!");
   buf.Flush();
}
```

Google

Communication channels

```
var c chan string;

c = make(chan string);

c <- "Hello";  // infix send

// in a different goroutine
greeting := <-c; // prefix receive

cc := new(chan chan string);

cc <- c;  // handing off a capability
```

Google

Goroutines

```
x := longCalculation(17); // runs too long

c := make(chan int);
func wrapper(a int, c chan int) {
    result := longCalculation(a);
    c <- result;
}
go wrapper(17, c);

// do something for a while; then...
x := <-c;
```

Google

A multiplexed server

```
type Request struct {
    a, b    int;
    replyc  chan int;  // reply channel inside the Request
}

type binOp func(a, b int) int

func run(op binOp, req *request) {
    req.replyc <- op(req.a, req.b)
}

func server(op binOp, service chan *request) {
    for {
        req := <-service; // requests arrive here
        go run(op, req);  // don't wait for op
    }
}

func StartServer(op binOp) chan *request {
    reqChan := make(chan *request);
    go server(op, reqChan);
    return reqChan;
}
```

Google

The client

```
// Start server; receive a channel on which
// to send requests.
server := StartServer(
          func(a, b int) int {return a+b});

// Create requests
req1 := &Request{23,45, make(chan int)};
req2 := &Request{-17,1<<4, make(chan int)};

// Send them in arbitrary order
server <- req1; server <- req2;

// Wait for the answers in arbitrary order
fmt.Printf("Answer2: %d\n", <-req2.replyc);
fmt.Printf("Answer1: %d\n", <-req1.replyc);
```

Google

Select

A **select** is like a **switch** statement in which the cases are communications. A simple example uses a second channel to tear down the server.

```
func server(op binOp, service chan *request,
            quit chan bool) {
    for {
        select {
        case req := <-service:
            go run(op, req);  // don't wait
        case <-quit:
            return;
        }
    }
}
```

Google

And more...

No time to talk about:

- package construction
- initialization
- reflection
- dynamic types
- embedding
- iterators
- testing

Google

Chaining

```
package main

import ("flag"; "fmt")

var ngoroutine = flag.Int("n", 100000, "how many")

func f(left, right chan int) { left <- 1 + <-right }

func main() {
    flag.Parse();
    leftmost := make(chan int);
    var left, right chan int = nil, leftmost;
    for i := 0; i < *ngoroutine; i++ {
        left, right = right, make(chan int);
        go f(left, right);
    }
    right <- 0;  // bang!
    x := <-leftmost;  // wait for completion
    fmt.Println(x);   // 100000
}
```

Google

Concurrency demo

Google

Status

Google

Compilers

Two variants:

6g/8g/5g (Ken Thompson)

more experimental.
generates OK code very quickly.
not GCC-linkable but has FFI support.

gccgo (Ian Taylor)

Go front end for GCC.
generates good code not as quickly.

Both support 32- and 64-bit x86, plus ARM.

Performance: typically within 10-20% of C.

Google

Run-time

Run-time handles memory allocation and collection, stack handling, goroutines, channels, slices, maps, reflection, and more.

Solid but improving.

6g has good goroutine support, muxes them onto threads, implements segmented stacks.

Gccgo is (for a little while yet) lacking segmented stacks, allocates one goroutine per thread.

Google

Garbage collector

6g has a simple but effective mark-and-sweep collector. Work is underway to develop the ideas in IBM's Recycler garbage collector* to build a very efficient, low-latency concurrent collector.

Gccgo at the moment has no collector; the new collector is being developed for both compilers.

* http://www.research.ibm.com/people/d/dfb/papers.html

Google

Libraries

Lots of libraries but plenty still needed.
Some (e.g. regexp) work fine but are too simple.

OS, I/O, files
math (sin(x) etc.)
strings, Unicode, regular expressions
reflection
command-line flags, logging
hashes, crypto
testing (plus testing tool, **`gotest`**)
networking, HTTP, RPC
HTML (and more general) templates

...and lots more.

Google

Godoc and Gofmt

`Godoc`:

documentation server, analogous to **`javadoc`**
but easier on the programmer. Can run yourself
but live at:

http://golang.org/ (top-level; serves all docs)
http://golang.org/pkg/ (package docs)
http://golang.org/src/ (source code)

`Gofmt`:

pretty-printer; all code in the repository has
been formatted by it.

Google

Debugger

A custom debugger is underway; not quite ready yet (but close).

Gccgo users can invoke **gdb** but the symbol table makes it look like C and there's no knowledge of the run-time.

Google

What about generics?

Go does not have generic types, etc.

We don't yet understand the right semantics for them given Go's type system but we're still thinking. They will add complexity so must be done right.

Generics would definitely be useful, though maps, slices, and interfaces address many common use cases.

Collections can be built using the empty interface, at the cost of manual unboxing.

In short: not yet.

Google

What about ...?

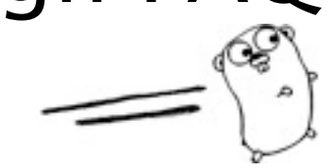
Your favorite feature may be missing (or present in a different form, such as **`enum`** vs. **`iota`**).

Perhaps it's not a good fit for the kind of language Go aims to be or contradicts Go's design goals.

Perhaps it hasn't been a priority for us while we explore other areas of the language.

We expect people to notice missing features. Don't let that stop you from exploring the features that Go does have.

For more info, there's a language design FAQ on the web site.



Google

Conclusion

Google

A call for action

It's early yet but promising.
A very comfortable and productive language.

Lots of documents on the web:
  specification, tutorial, "Effective Go", FAQs, more
Full open-source implementations.

Want to try it?
Want to help?
Want to build libraries or tools?

http://golang.org

Google

The Go Programming Language

Rob Pike
golang.org
Oct 30, 2009

http://golang.org

Google