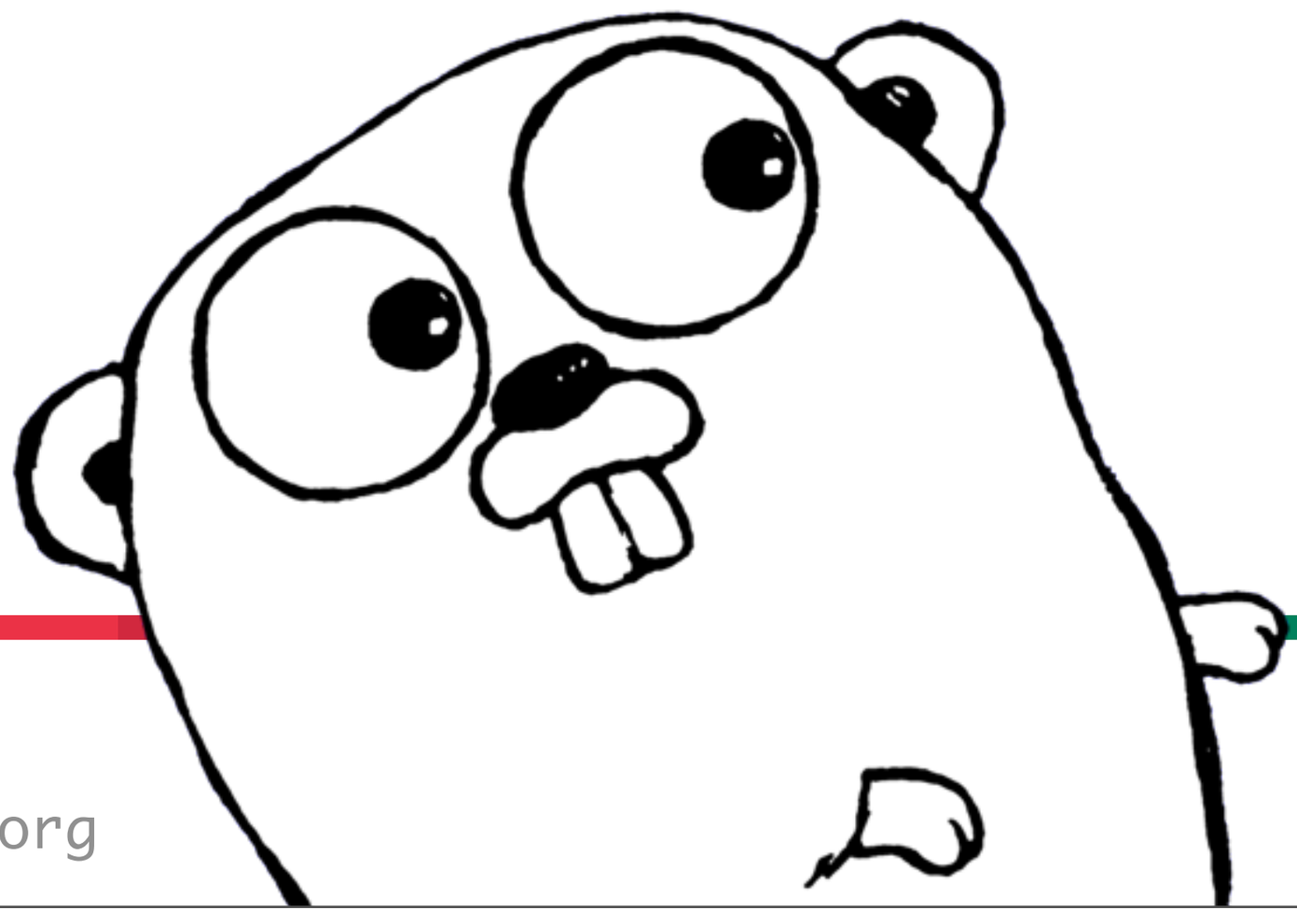
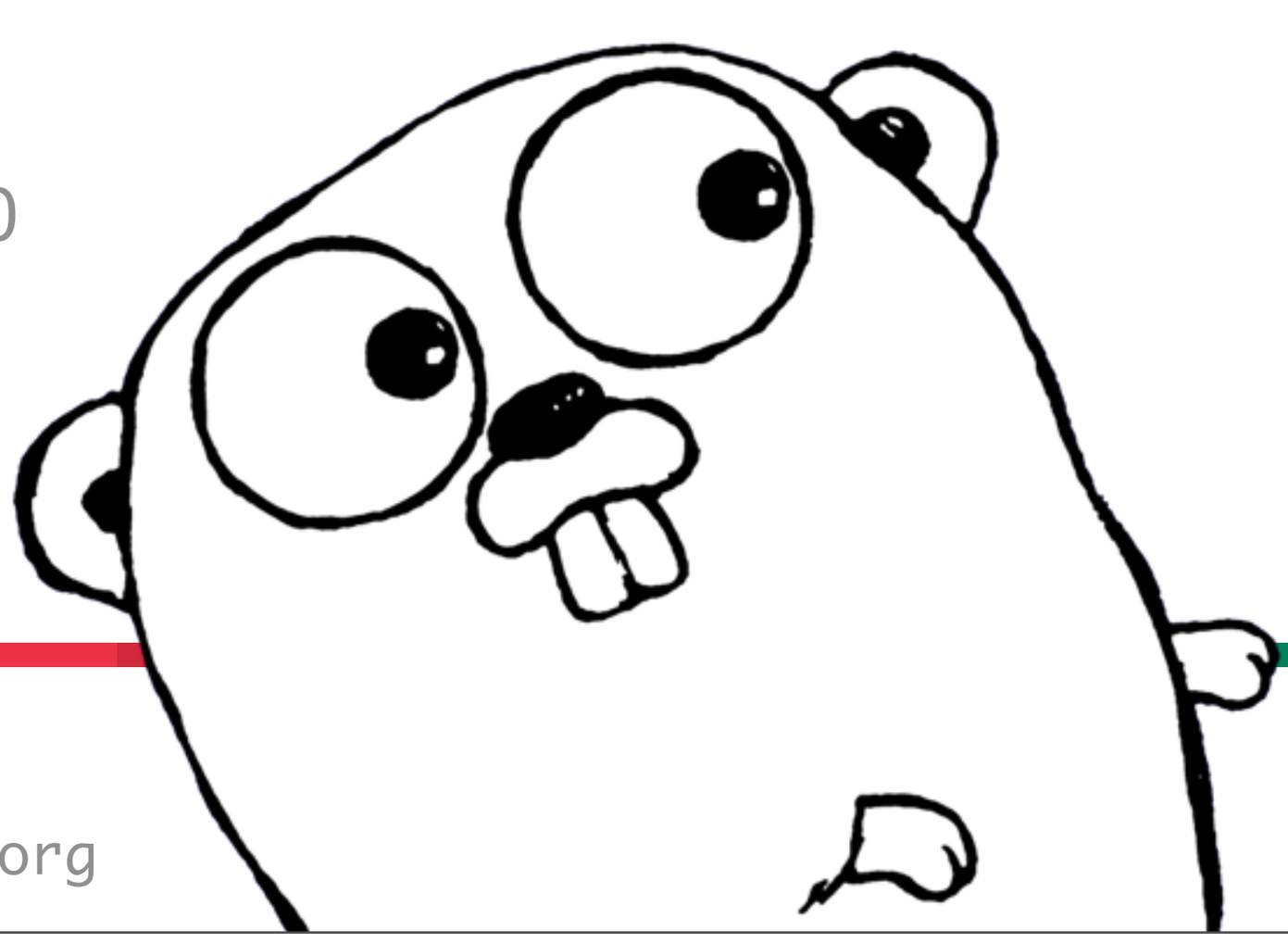


http://golang.org

Google

Another Go at Language Design

Rob Pike
OSCON
July 22, 2010



http://golang.org

Google

Who

Russ Cox
Robert Griesemer
Rob Pike
Ian Taylor
Ken Thompson

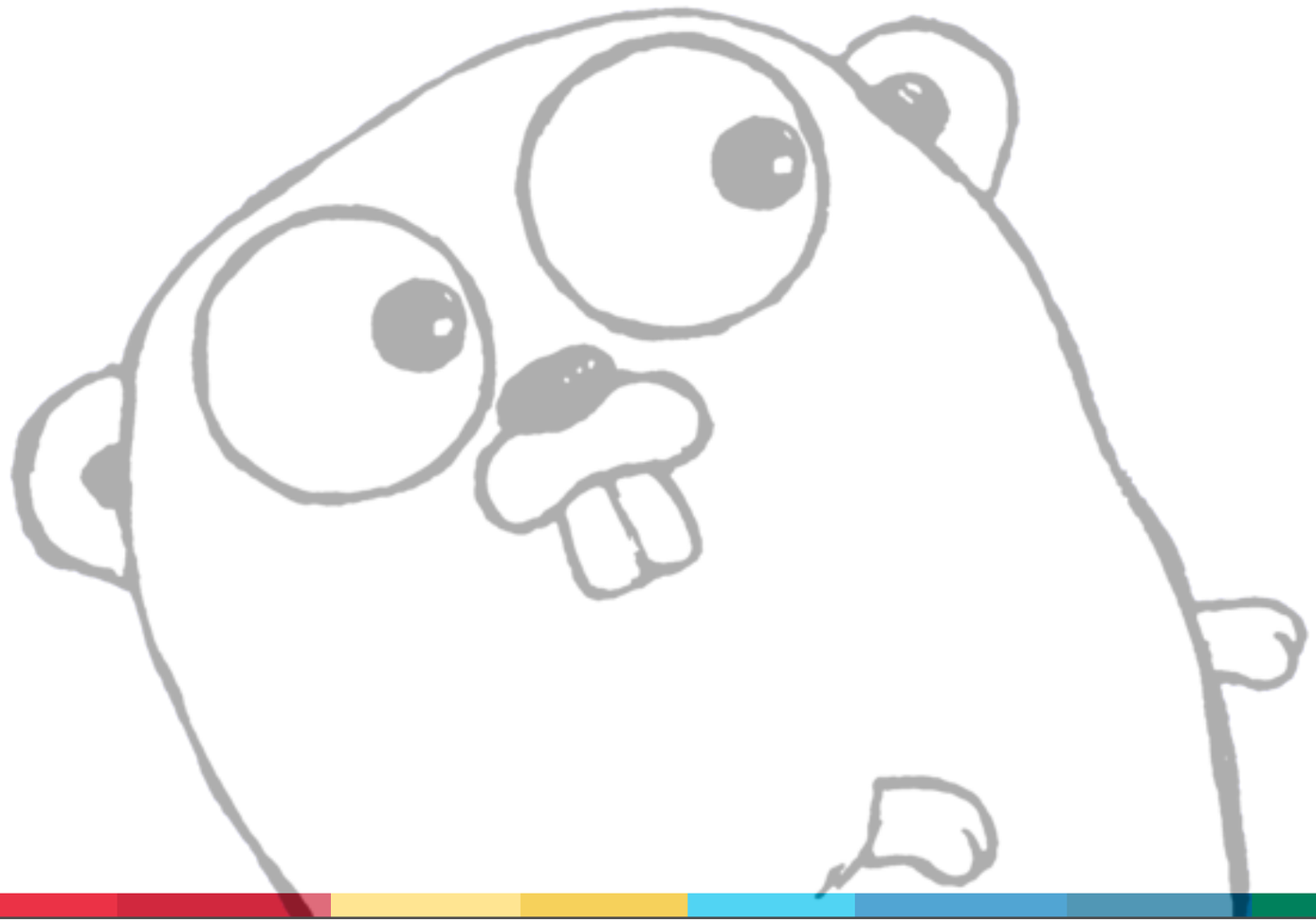
plus David Symonds, Nigel Tao, Andrew Gerrand, Stephen Ma, and others,

plus many contributions from the open source community.

Google

Go

Picking up where we left off the keynote...



Google

How does Go fill the niche?

General purpose

Concise syntax

Expressive type system

Concurrency

Garbage collection

Fast compilation

Efficient execution

Google

The target

Go aims to combine the safety and performance of a statically typed compiled language with the expressiveness and convenience of a dynamically typed interpreted language.

It also aims to be suitable for modern systems - large scale - programming.

Google

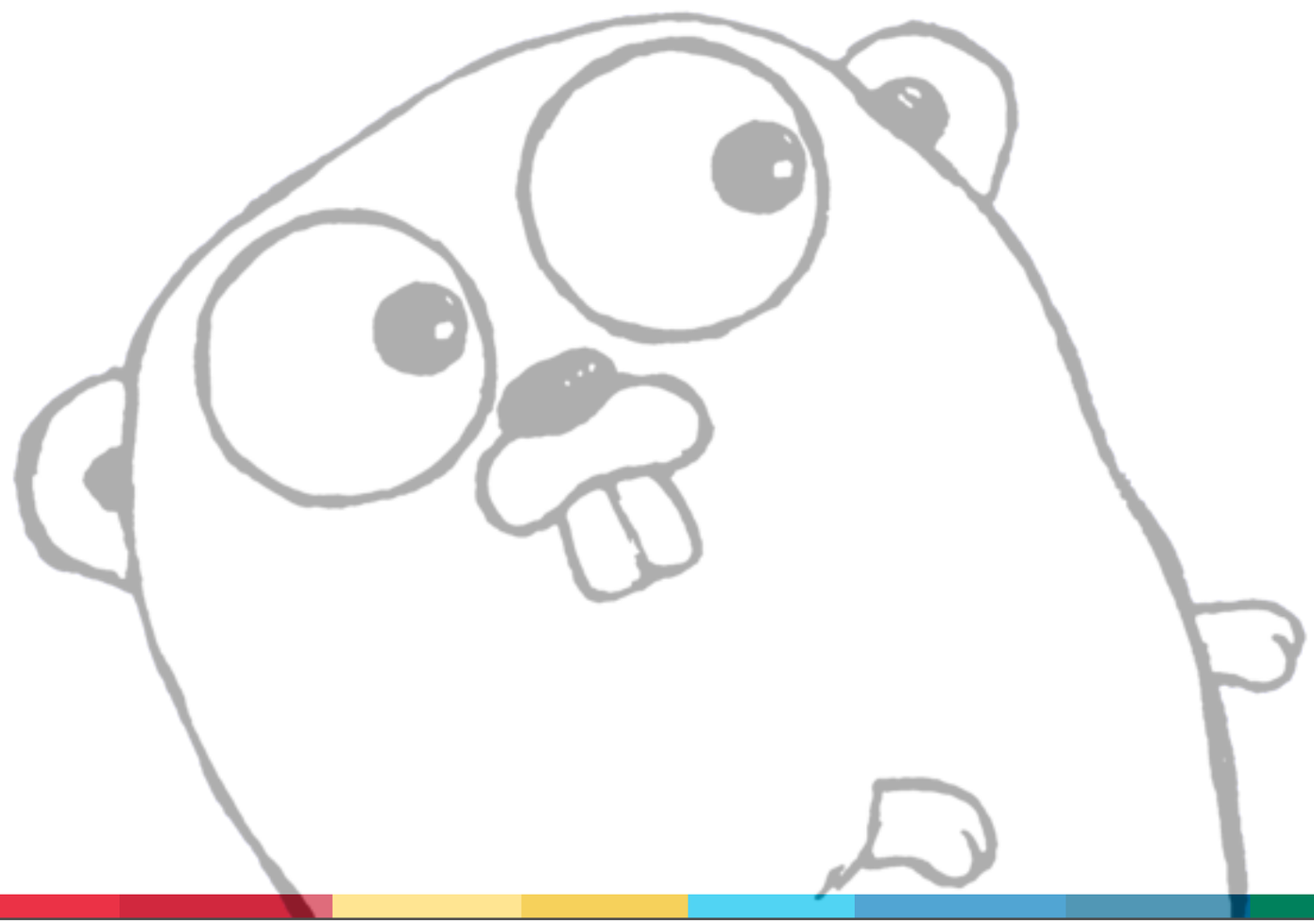
Hello, world 2.0

Serving `http://localhost:8080/world`:

```
package main

import (
    "fmt"
    "http"
)

func handler(c *http.Conn, r *http.Request) {
    fmt.Fprintf(c, "Hello, %s.", r.URL.Path[1:])
}

func main() {
    http.ListenAndServe(":8080",
                        http.HandlerFunc(handler))
}
```


Google

Demo



Google

Why so fast?

New clean compiler worth ~5X compared to gcc.

We want a millionX for large programs, so we need to fix the dependency problem.

In Go, programs compile into packages and each compiled package file imports transitive dependency info.

If `A.go` depends on `B.go` depends on `C.go`:
- compile `C.go`, `B.go`, then `A.go`.
- to recompile `A.go`, compiler reads `B.o` but not `C.o`.

At scale, this can be a huge speedup.

Trim the tree

Large C++ programs (Firefox, OpenOffice, Chromium) have huge build times. On a Mac (OS X 10.5.7, gcc 4.0.1):

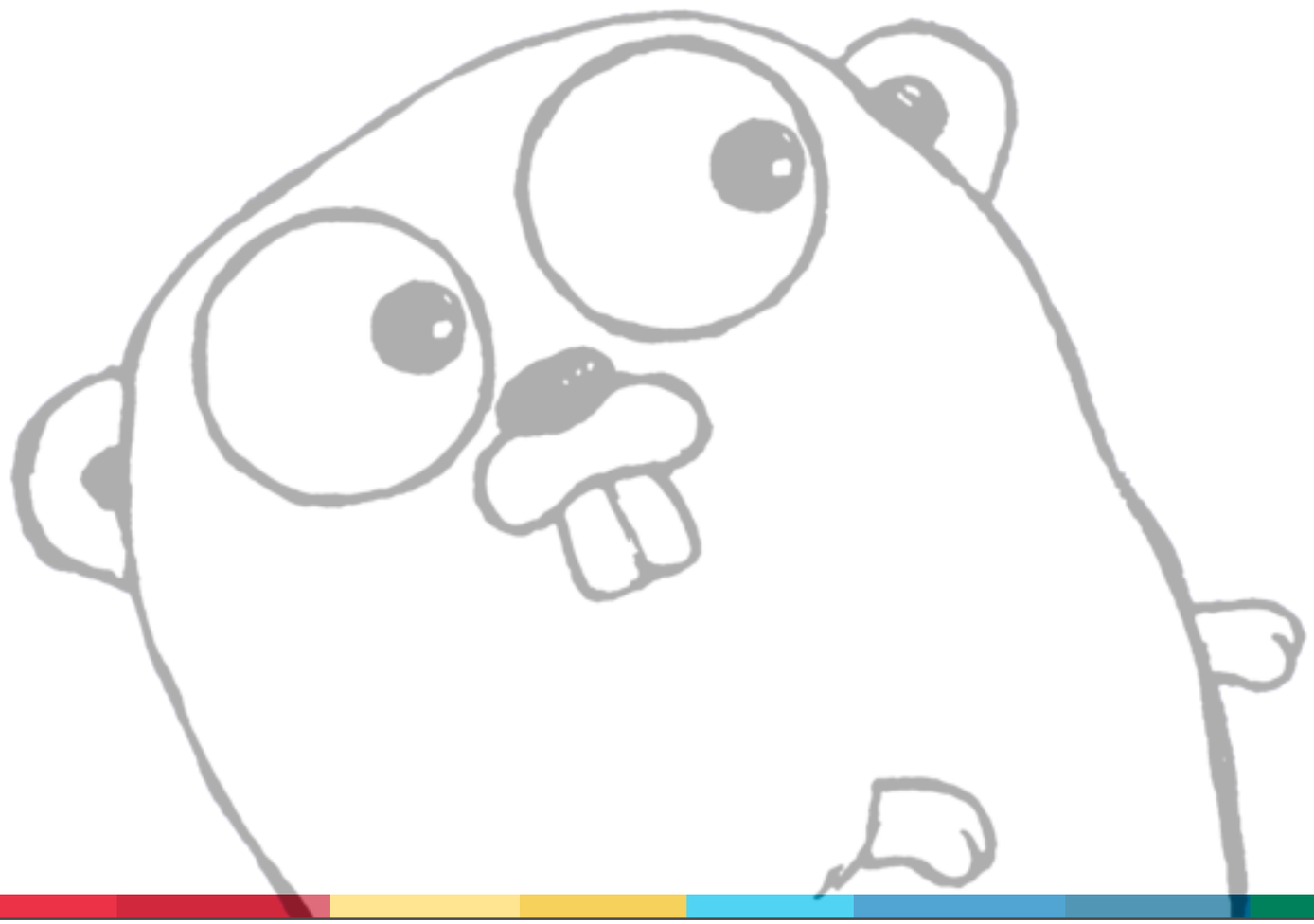
C: `#include <stdio.h>`
reads 360 lines from 9 files
C++: `#include <iostream>`
reads 25,326 lines from 131 files
Objective-C: `#include <Cocoa/Cocoa.h>`
reads 112,047 lines from 689 files
But we haven't done any real work yet!

In Go, `import "fmt"` reads one file:
195 lines summarizing 6 dependent packages

As we scale, the improvement becomes exponential.

A tour of Go

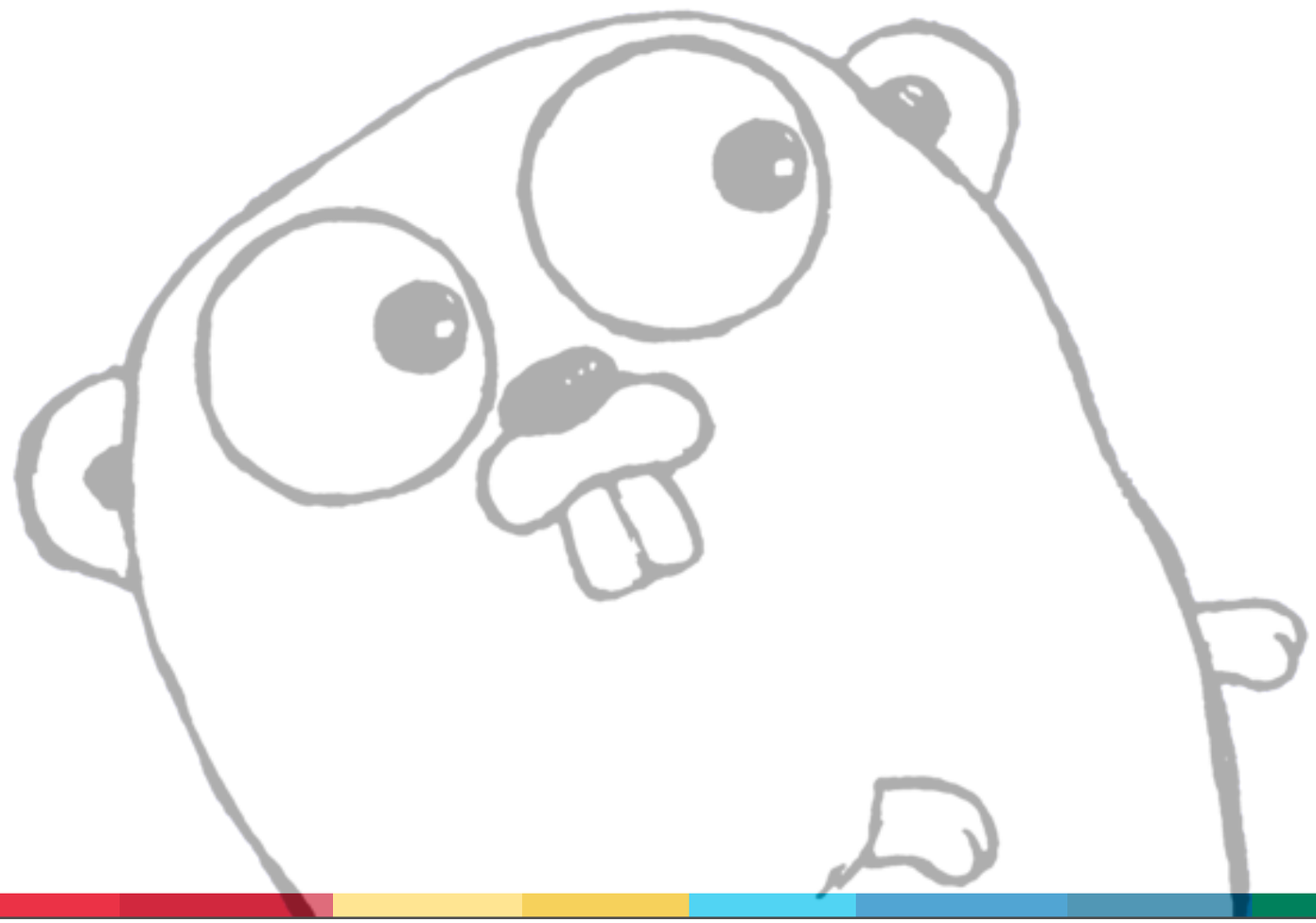
Highlights only



Google

Types

Any type can have methods.
Interfaces are satisfied implicitly.



Google

Expressive type system

Go is an object-oriented language, but unusually so.

There is no such thing as a class.

There is no subclassing.

Any types, even basic types such as integers and strings, can have methods.

Objects implicitly satisfy interfaces, which are just sets of methods.

Any named type can have methods

```
type Day int

var dayName = []string{"Sunday", "Monday"} // and so on

func (d Day) String() string {
    if 0 <= d && int(d) < len(dayName) { return dayName[d] }
    return "NoSuchDay"
}

type Fahrenheit float

func (t Fahrenheit) String() string {
    return fmt.Sprintf("%.1f°F", t)
}
```

Note that these methods do not take a pointer (although they could).

The `Day` type is not the same notion as Java's Integer type: it's really an `int`. There is no box.

Google

Interfaces

```
type Stringer interface {
    String() string
}

func print(args ...Stringer) {
    for i, s := range args {
        if i > 0 { os.Stdout.WriteString(" ") }
        os.Stdout.WriteString(s.String())
    }
}

print(Day(1), Fahrenheit(72.29))
    => Monday 72.3°F
```

Again, these methods do not take a pointer, although another type might define a `String()` method that does, and it too would satisfy `Stringer`.

Google

Empty interface

The empty interface (`interface {}`) has no methods.
Every type satisfies the empty interface.

```
func print(args ...interface{}) {
    for i, arg := range args {
        if i > 0 { os.Stdout.WriteString(" ") }
        switch a := arg.(type) {  // "type switch"
            case Stringer: os.Stdout.WriteString(a.String())
            case int:      os.Stdout.WriteString(itoa(a))
            case string:   os.Stdout.WriteString(a)
            // more types can be used
            default:       os.Stdout.WriteString("????")
        }
    }
}

print(Day(1), "was", Fahrenheit(72.29))
    => Monday was 72.3°F
```


Google

Small and implicit

`Fahrenheit` and `Day` satisfied `Stringer` implicitly; other types might too. A type satisfies an interface simply by implementing its methods. There is no "implements" declaration; interfaces are satisfied implicitly.

It's a form of duck typing, but (usually) checkable at compile time.

An object can (and usually does) satisfy many interfaces simultaneously. For instance, `Fahrenheit` and `Day` satisfy `Stringer` and also the empty interface.

In Go, interfaces are usually small: one or two or even zero methods.

Google

Reader

```
type Reader interface {
    Read(p []byte) (n int, err os.Error)
}
// And similarly for Writer
```

Anything with this `Read` method signature implements `Reader`.

- Sources: files, buffers, network connections, pipes
- Filters: buffers, checksums, decompressors, decrypters

JPEG decoder takes a `Reader`, so it can decode from disk, network, gzipped HTTP, ...

Buffering just wraps a `Reader`:

```
var bufferedInput Reader = bufio.NewReader(os.Stdin)
```

`Fprintf` uses a `Writer`:

```
func Fprintf(w Writer, fmt string, a ...interface{})
```


Google

Interfaces can be retrofitted

Had an existing RPC implementation that used custom wire format. Changed to an interface:

```
type Encoding interface {
    ReadRequestHeader(*Request) os.Error
    ReadRequestBody(interface{}) os.Error
    WriteResponse(*Response, interface{}) os.Error
    Close() os.Error
}
```

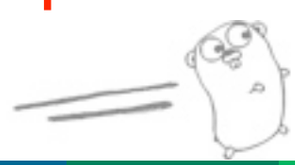
Two functions (send, recv) changed signature. Before:

```
func sendResponse(sending *sync.Mutex, req *Request,
        reply interface{}, enc *gob.Encoder, errmsg string)
```

After (and similarly for receiving):

```
func sendResponse(sending *sync.Mutex, req *Request,
        reply interface{}, enc Encoding, errmsg string)
```

That is almost the whole change to the RPC implementation.



Post facto abstraction

We saw an opportunity: RPC needed only `Encode` and `Decode` methods. Put those in an interface and you've abstracted the codec.

Total time: 20 minutes, including writing and testing the JSON implementation of the interface.

(We also wrote a trivial wrapper to adapt the existing codec for the new `rpc.Encoding` interface.)

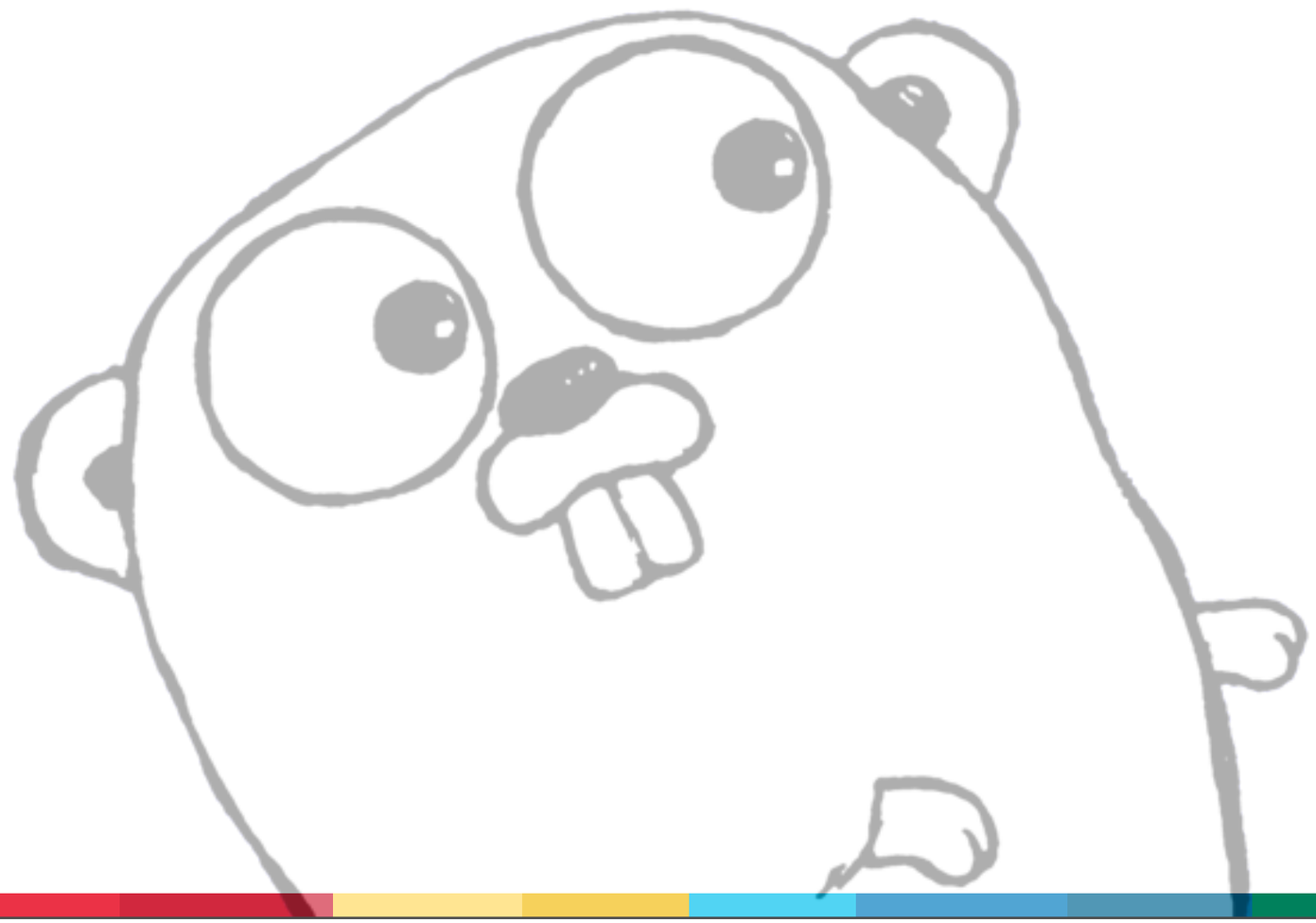
In Java, `RPC` would be refactored into a half-abstract class, subclassed to create `JsonRPC` and `StandardRPC`.

In Go, there is no need to manage a type hierarchy: just pass in an encoding interface stub (and nothing else).

Google

Concurrency

Go is concurrent, in the CSP family.
Channels are first-class.



Google

Concurrency

Systems software must often manage connections and clients.

Go provides independently executing goroutines that communicate and synchronize using channels.

Analogy with Unix: processes connected by pipes. But in Go things are fully typed and lighter weight.

Google

Goroutines

Start a new flow of control with the `go` keyword.
Parallel computation is easy:

```
func main() {
    go expensiveComputation(x, y, z)
    anotherExpensiveComputation(a, b, c)
}
```

Roughly speaking, a goroutine is like a thread, but lighter weight:

- stacks are small, segmented, sized on demand
- goroutines are multiplexed by demand onto true threads
- requires support from language, compiler, runtime
  - can't just be a C++ library

Google

Thread per connection

Doesn't scale in practice, so in most languages we use event-driven callbacks and continuations.

But in Go, a goroutine per connection model scales well.

```
for {
    rw := socket.Accept()
    conn := newConn(rw, handler)
    go conn.serve()
}
```


Google

Channels

Our trivial parallel program again:

```
func main() {
    go expensiveComputation(x, y, z)
    anotherExpensiveComputation(a, b, c)
}
```

Need to know when the computations are done.
Need to know the result.

A Go channel provides the capability: a typed synchronous communications mechanism.

Google

Channels

Goroutines communicate using channels.

```
func computeAndSend(x, y, z int) chan int {
    ch := make(chan int)
    go func() {
        ch <- expensiveComputation(x, y, z)
    }()
    return ch
}

func main() {
    ch := computeAndSend(x, y, z)
    v2 := anotherExpensiveComputation(a, b, c)
    v1 := <-ch
    fmt.Println(v1, v2)
}
```


Google

A worker pool

Traditional approach (C++, etc.) is to communicate by sharing memory and protecting the shared data structures with mutexes (locks).

Server would use shared memory to apportion work:

```
type Work struct {
    x, y, z int
    assigned, done bool
}

type WorkSet struct {
    mu sync.Mutex
    work []*Work
}
```

But not in Go.

Google

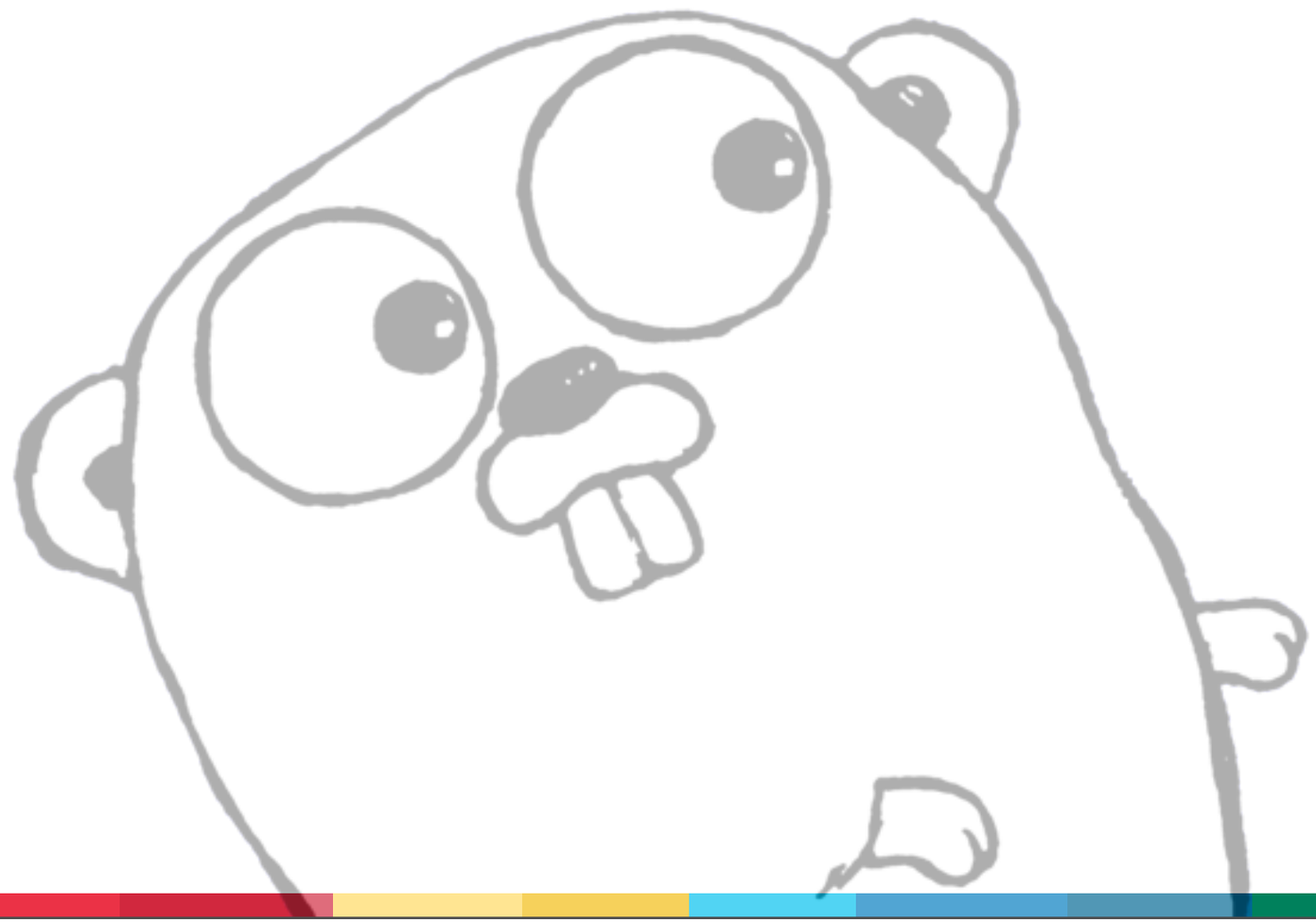
Share memory by communicating

In Go, you reverse the equation.
Channels use <- operator to synchronize and communicate.
Typically don't need or want mutexes.

```
type Work struct { x, y, z int }
func worker(in <-chan *Work, out chan<- *Work) {
    for w := range in {
        w.z = w.x * w.y  // do some work...
        out <- w
    }
}
func main() {
    in, out := make(chan *Work), make(chan *Work)
    for i := 0; i < 10; i++ { go worker(in, out) }
    go sendLotsOfWork(in)
    receiveLotsOfResults(out)
}
```

Memory

Concurrency and abstraction need garbage collection.
Go has pointers, but safely.



Garbage collection

Automatic memory management simplifies life.

GC is critical for concurrent programming; otherwise it's too fussy and error-prone to track ownership as data moves around.

GC also clarifies design. A large part of the design of C and C++ libraries is about deciding who owns memory, who destroys resources. Memory management must be hidden behind the interface.

But garbage collection isn't enough.

Google

Memory safety

Memory in Go is intrinsically safer:

- pointers but no pointer arithmetic
- no dangling pointers (locals move to heap as needed)
- no pointer-to-integer conversions*
- all variables are zero-initialized
- all indexing is bounds-checked

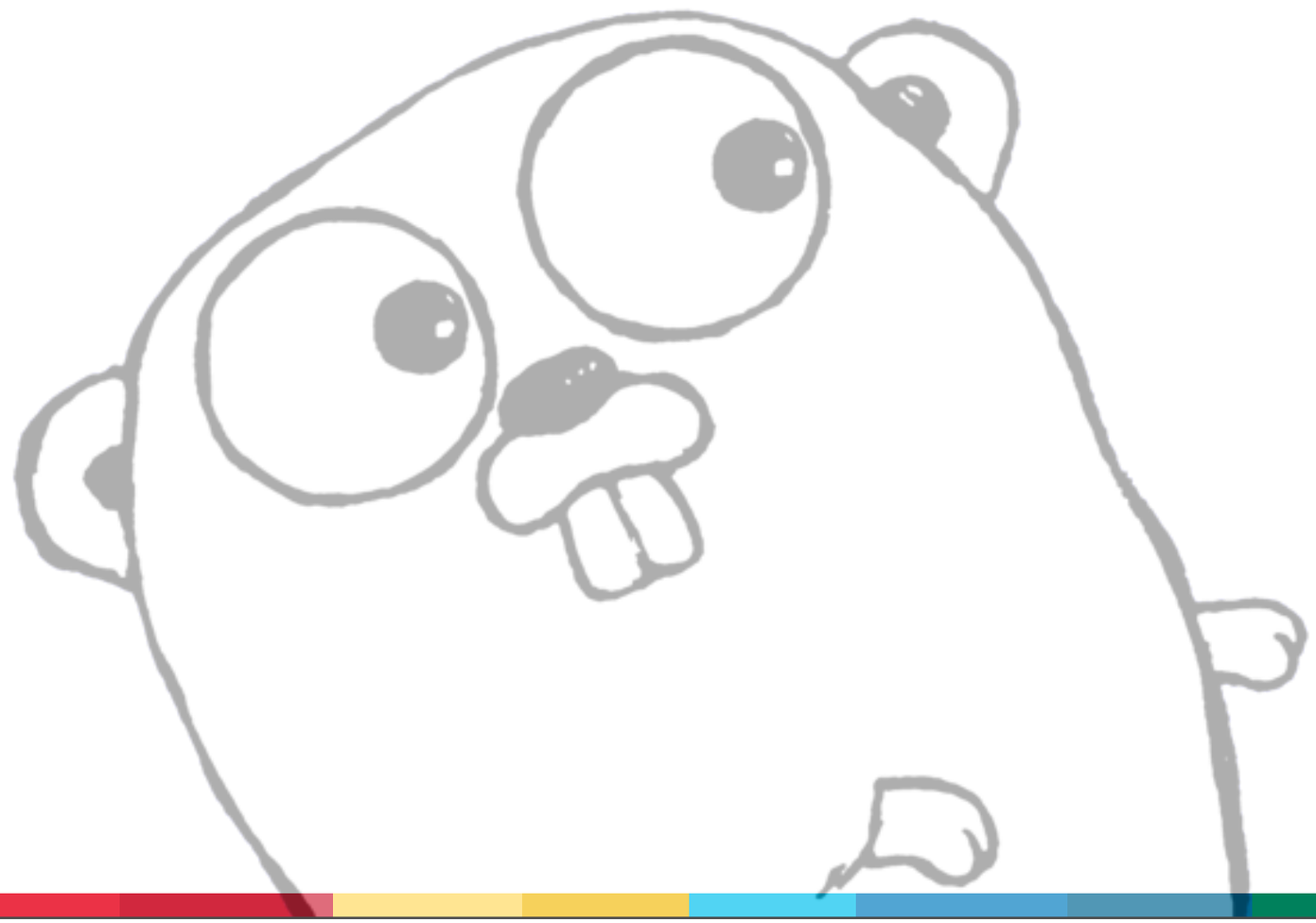
Should have far fewer buffer overflow exploits.

* Package `unsafe` allows this but labels the code as dangerous; used mainly in some low-level libraries.

Google

Designed as a systems language

Can do speed-critical things well (e.g. crypto).
Can control memory layout, work with bits.



Google

Systems language

By systems language, we mean suitable for writing systems software.

- web servers
- web browsers
- web crawlers
- search indexers
- databases
- compilers
- programming tools (debuggers, analyzers, ...)
- IDEs
- operating systems (maybe)

...

Google

Systems programming

From http://loadcode.blogspot.com/2009/12/go-vs-java.html

"[Git] is known to be very fast. It is written in C. A Java version JGit was made. It was considerably slower. Handling of memory and lack of unsigned types [were] some of the important reasons."

Shawn O. Pearce wrote on the git mailing list:

"JGit struggles with not having an efficient way to represent a SHA-1. C can just say "unsigned char[20]" and have it inline into the container's memory allocation. A byte[20] in Java will cost an *additional* 16 bytes of memory, and be slower to access because the bytes themselves are in a different area of memory from the container object."

Google

Control of bits and memory

Like C, Go has

- full set of unsigned types
- bit-level operations
- programmer control of memory layout

```
type T struct {
    x   int
    buf [20]byte
    ...
}
```

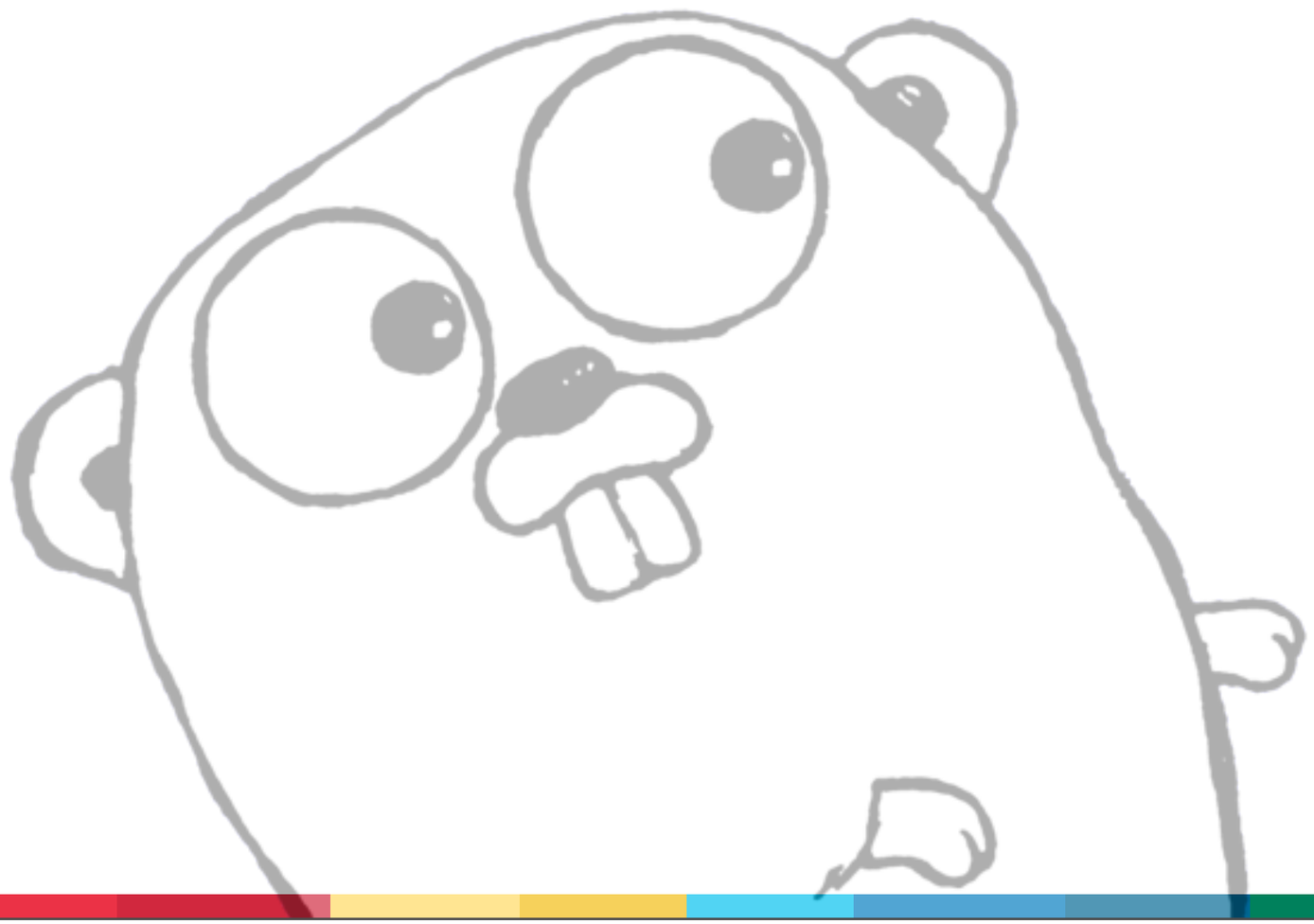
- pointers to inner values

```
p := &t.buf
```


Google

But a nice general-purpose language

Clean, simple, consistent, expressive.
Productive.



Google

Simplicity and clarity

Go's design aims for being easy to use, which means it must be easy to understand, even if that sometimes contradicts superficial ease of use.

Some examples:

No implicit numeric conversions, although the way constants work ameliorates the inconvenience. (Next slide.)

No method overloading. For a given type, there is only one method with that name.

There is no "public" or "private" label. Instead, items with `UpperCaseNames` are visible to clients; `lowerCaseNames` are not.

Google

Constants

Numeric constants are "ideal numbers": no size or signed/unsigned distinction, hence no L or U or UL endings.

```
077 // octal
0xFEEDBEEEEEEEEEEEEEEEEEEEEF // hexadecimal
1 << 100
```

Syntax of literal determines default type:

```
1.234e5  // float
1e2      // float
100      // int
```

But they are just numbers that can be used at will and assigned to variables with no conversions necessary.

```
seconds := time.Nanoseconds()/1e9  // result has integer type
```


High precision constants

Arithmetic with constants is high precision. Only when assigned to a variable are they rounded or truncated to fit.

```
const MaxUint = 1<<32 - 1
const Ln2     = 0.693147180559945309417232121458176\
                568075500134360255254120680009
const Log2E   = 1/Ln2 // accurate reciprocal
var x float64 = Log2E // rounded to nearest float64 value
```

The value assigned to `x` will be as precise as possible in a 64-bit `float`.

Google

And much more...

There are other aspects of Go that make it easy and expressive yet scalable and efficient:

- clear package structure
- clear rules about how a program begins execution
- top-level initializing functions and values
- composite values

```
var freq = map[string]float{"C4":261.626, "A4":440}  // etc.
```

- tagged elements in initializers

```
var s = Point{x:27, y:-13.2}
```

- function literals and closures

```
go func() { for { c1 <- <-c2 } }()
```

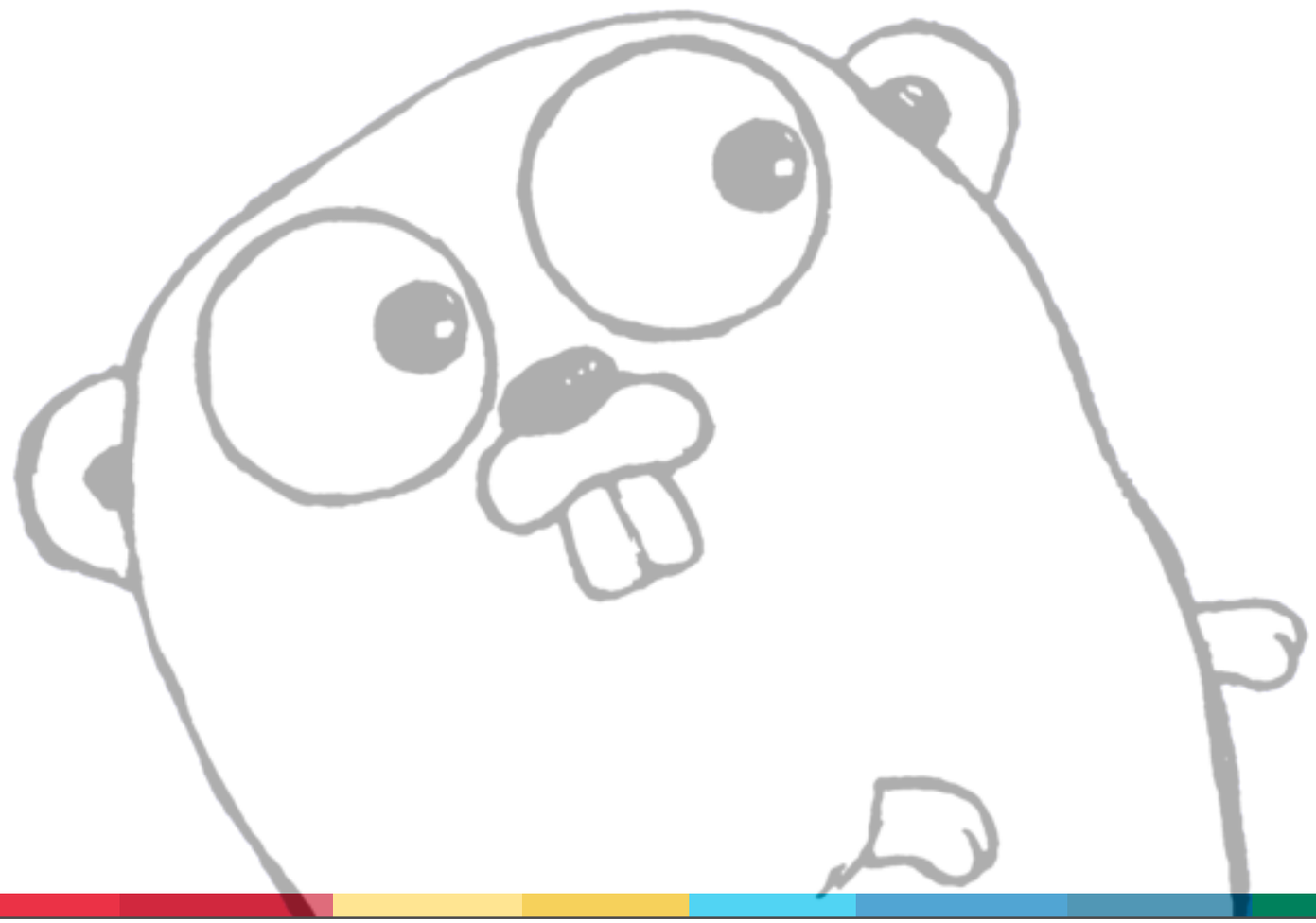
- reflection
- and more....

Plus automatic document generation and formatting.

Google

Summary

Go is different.
Go is productive.
Go is fun.

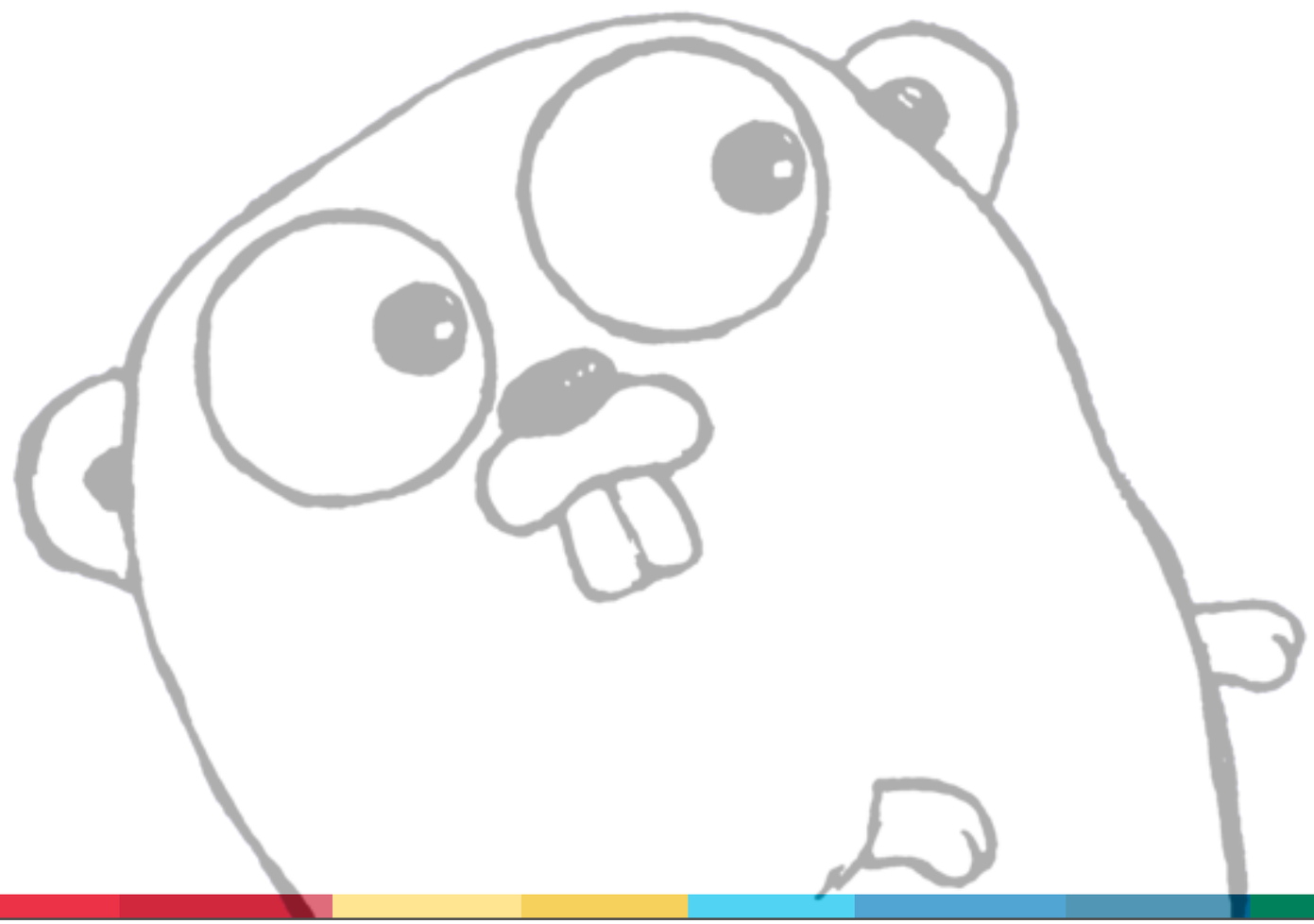


Google

Go is different

- Go is object-oriented not type-oriented
  - inheritance is not primary
  - methods on any type, but no classes or subclasses
- Go is (mostly) implicit not explicit
  - types are inferred not declared
  - objects have interfaces but they are derived, not specified
- Go is concurrent not parallel
  - intended for program structure, not max performance
  - but still can keep all the cores humming nicely
  - ... and many programs are more nicely expressed with concurrent ideas even if not parallel at all

Status



Google

Implementation

The language is designed and usable. Two compiler suites:

Gc, written in C, generates OK code very quickly.
- unusual design based on the Plan 9 compiler suite

Gccgo, written in C++, generates good code more slowly
- uses GCC's code generator and tools

Libraries good and growing, but some pieces are still preliminary.

Garbage collector works fine (simple mark and sweep) but is being rewritten for more concurrency, less latency.

Available for Linux etc., Mac OS X. Windows port working.

All available as open source; see `http://golang.org`.

Google

Acceptance

Go was the 2009 TIOBE "Language of the year" two months after it was released.

Year	Winner
2009	Go
2008	C
2007	Python
2006	Ruby
2005	Java
2004	PHP
2003	C++

Testimonials

"I have reimplemented a networking project from Scala to Go. Scala code is 6000 lines. Go is about 3000.

Even though Go does not have the power of abbreviation, the flexible type system seems to out-run Scala when the programs start getting longer.

Hence, Go produces much shorter code asymptotically."

- Petar Maymounkov

"Go is unique because of the set of things it does well. It has areas for improvement, but for my needs it is the best match I've found when compared to: C, C++, Erlang, Python, Ruby, C#, D, Java, and Scala."

- Hans Stimer

Utility

For those on the team, it's the main day-to-day language now. It has rough spots but mostly in the libraries, which are improving fast.

Productivity seems much higher. (I get behind on mail much more often.) Most builds take a fraction of a second.

Starting to be used inside Google for some production work.

We haven't built truly large software in Go yet, but all indicators are positive.

Google

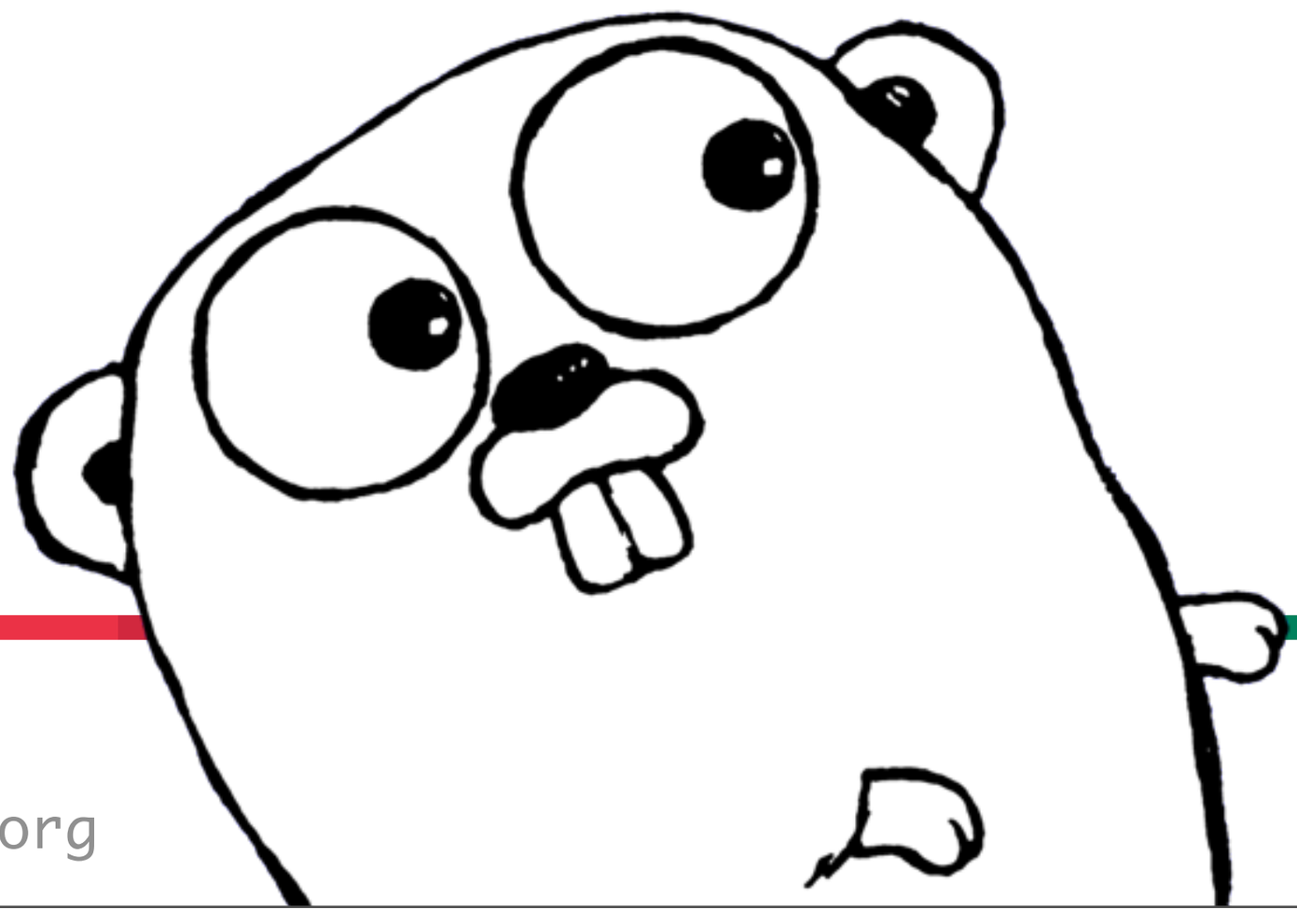
Try it out

This is a true open source project.

Full source, documentation and much more at

`http://golang.org`

Google

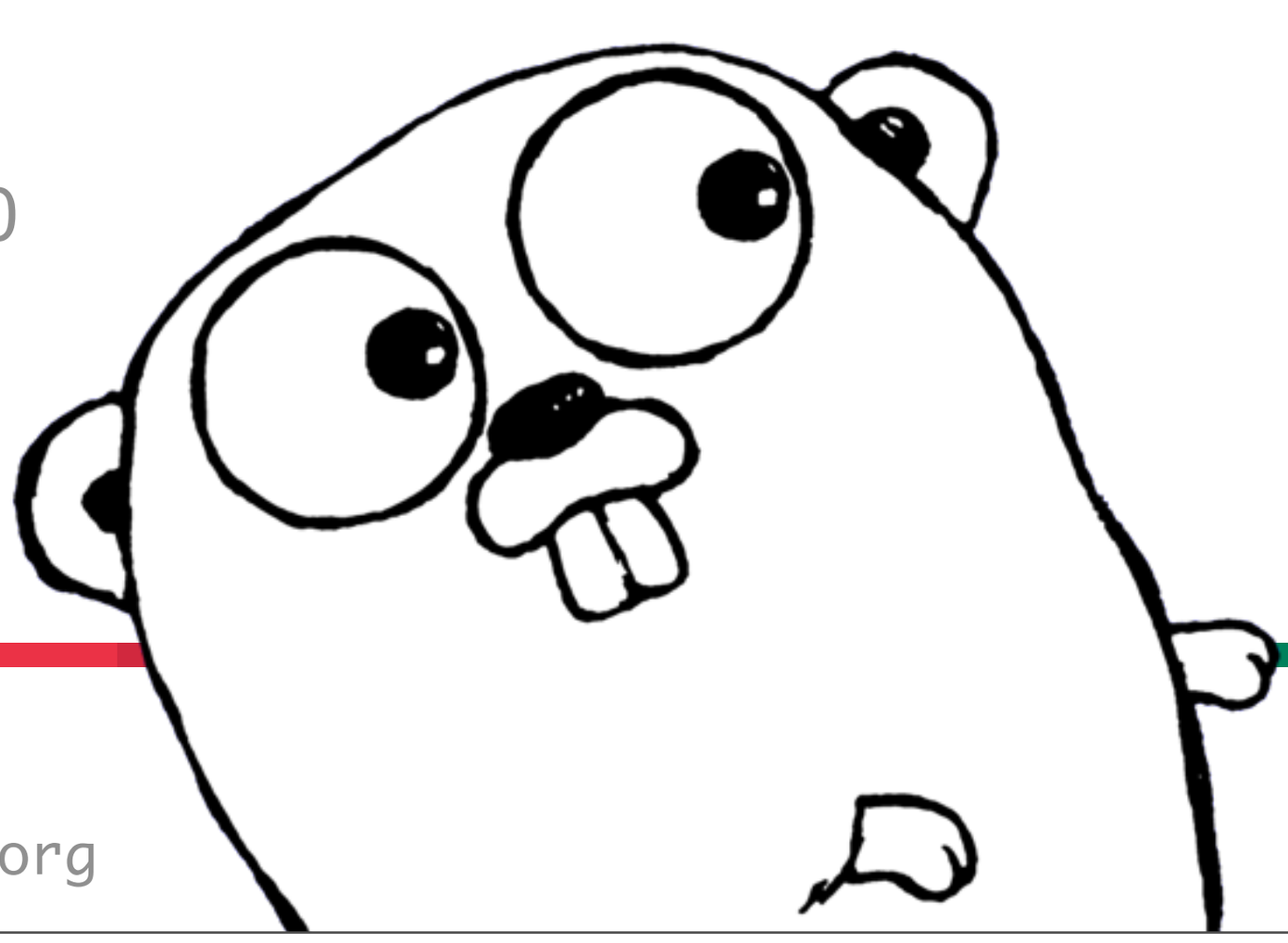


http://golang.org

Google

Another Go at Language Design

Rob Pike
OSCON
July 22, 2010



http://golang.org

Google