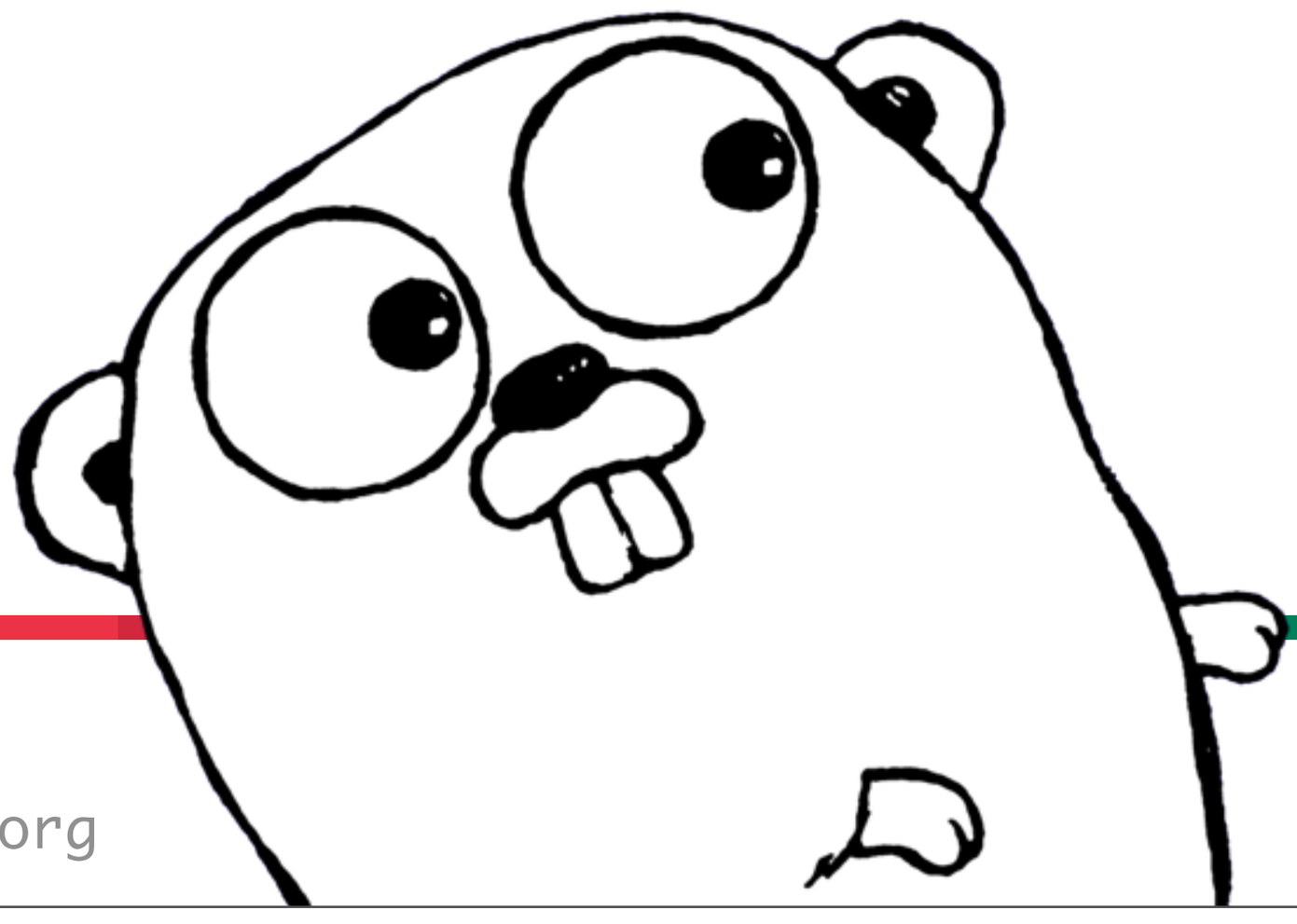


http://golang.org

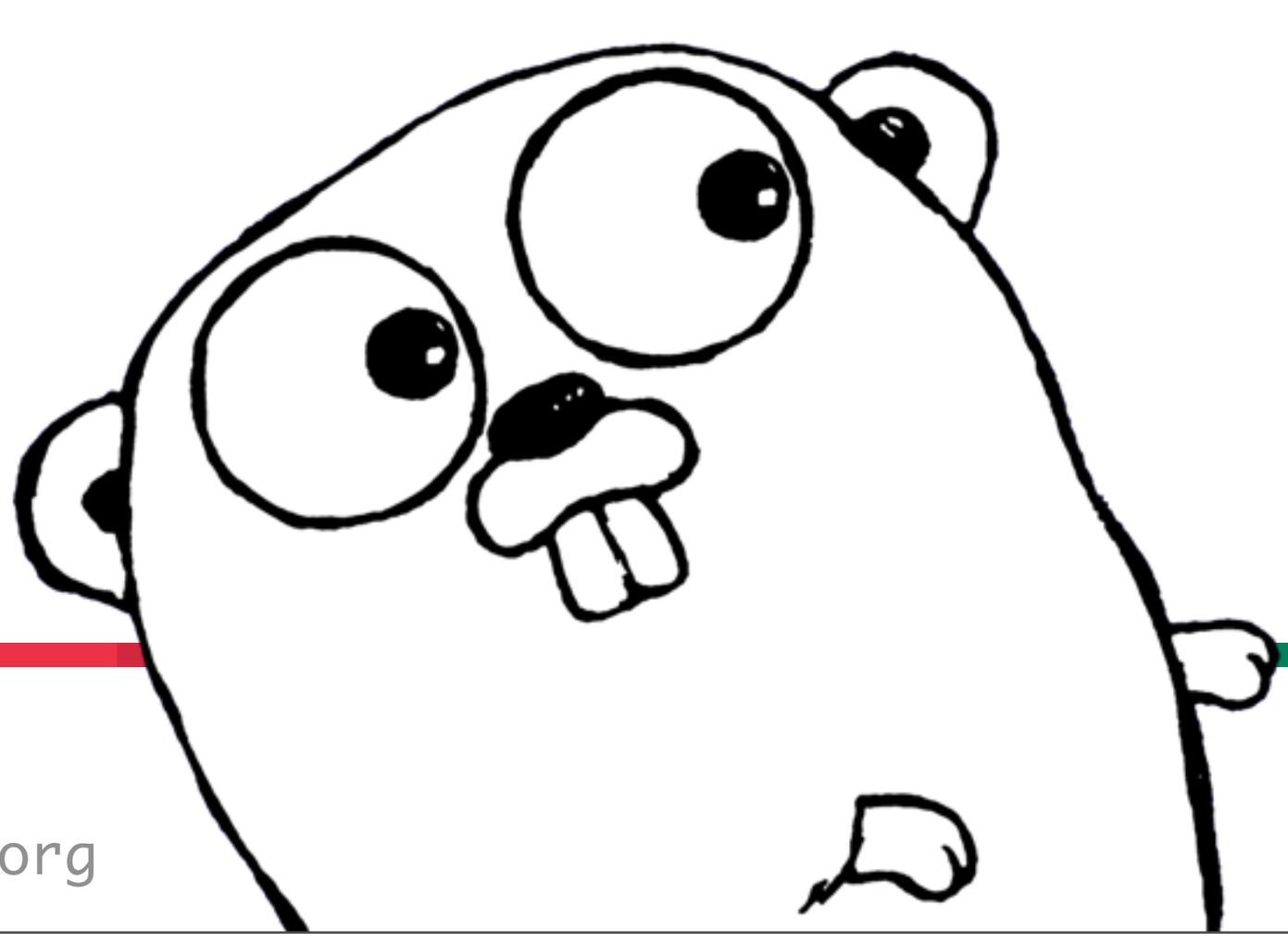
Google

The Expressiveness of Go

Rob Pike

JAOO

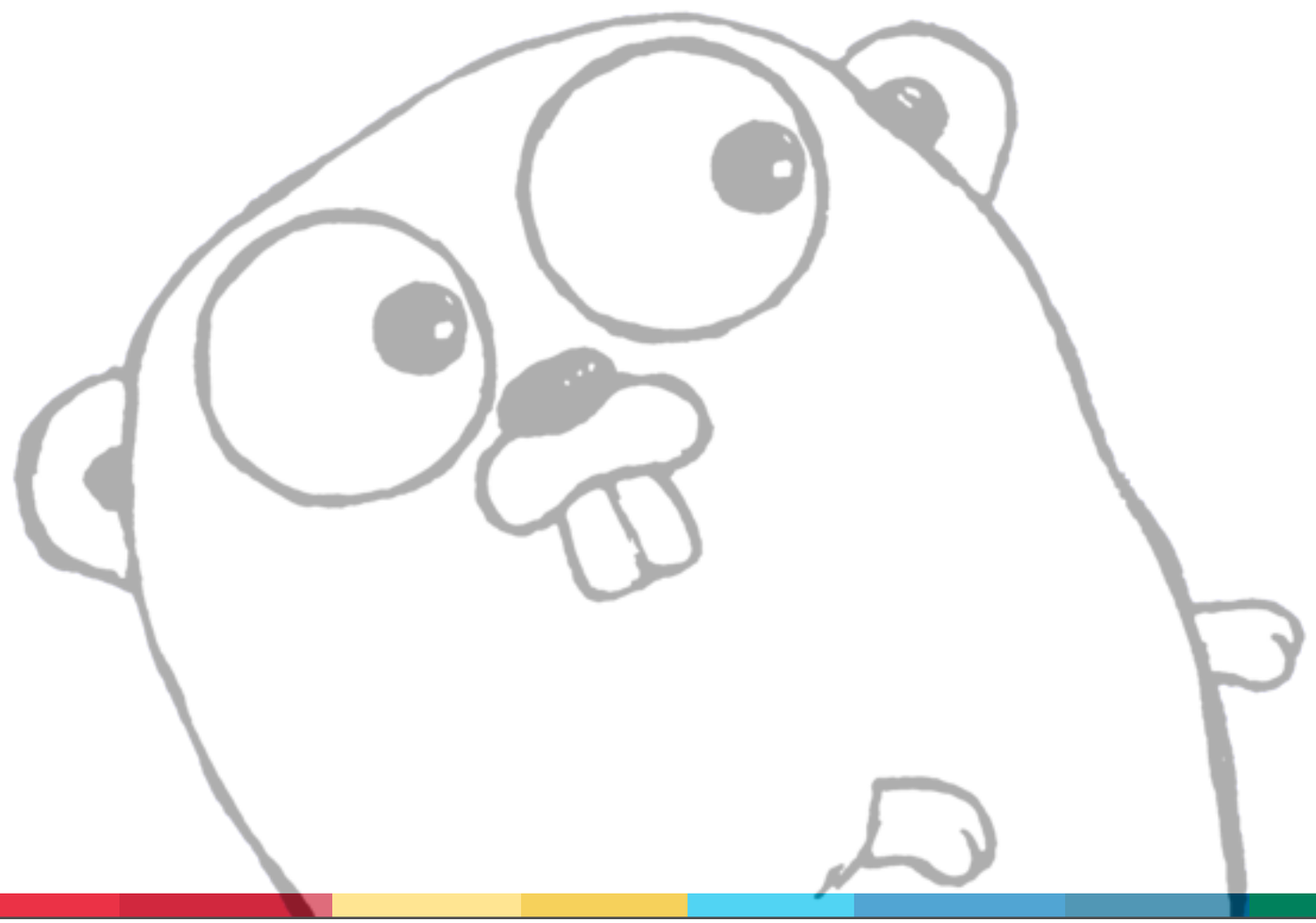
Oct 5, 2010



http://golang.org

Google

Who



Google

Team

Russ Cox
Robert Griesemer
Rob Pike
Ian Taylor
Ken Thompson

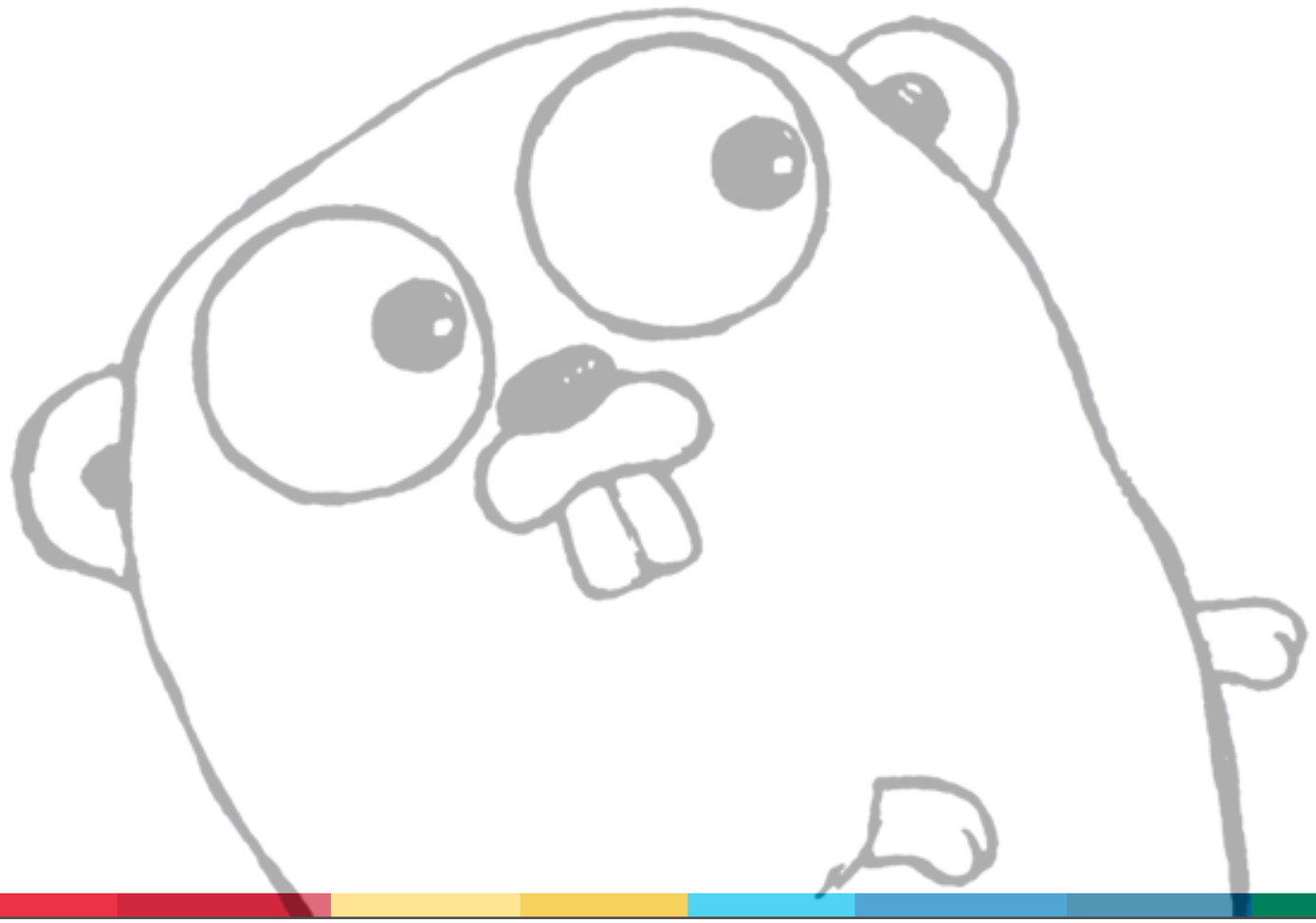
plus David Symonds, Nigel Tao, Andrew Gerrand, Stephen Ma, and others,

plus many contributions from the open source community.

Google

Why

Why a new language?



Google

Why Go?

A response to Google's internal needs:

- efficient large scale programming
- speed of compilation
- distributed systems
- multicore, networked hardware

And a reaction to: "speed and safety or ease of use; pick one."

- complexity, weight, noise (C++, Java)
  vs.
- no static checking (JavaScript, Python)

Go is statically typed and compiled, like C++ or Java (with no VM), but in many ways feels as lightweight and dynamic as JavaScript or Python.

Google

The surprise

It turned out to be a nice general purpose language.

That was unexpected.

The most productive language I've ever used.

And some people agree.

Google

Acceptance

Go was the 2009 TIOBE "Language of the year" two months after it was released and it won an InfoWorld BOSSIE award.

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

Google

Why the success?

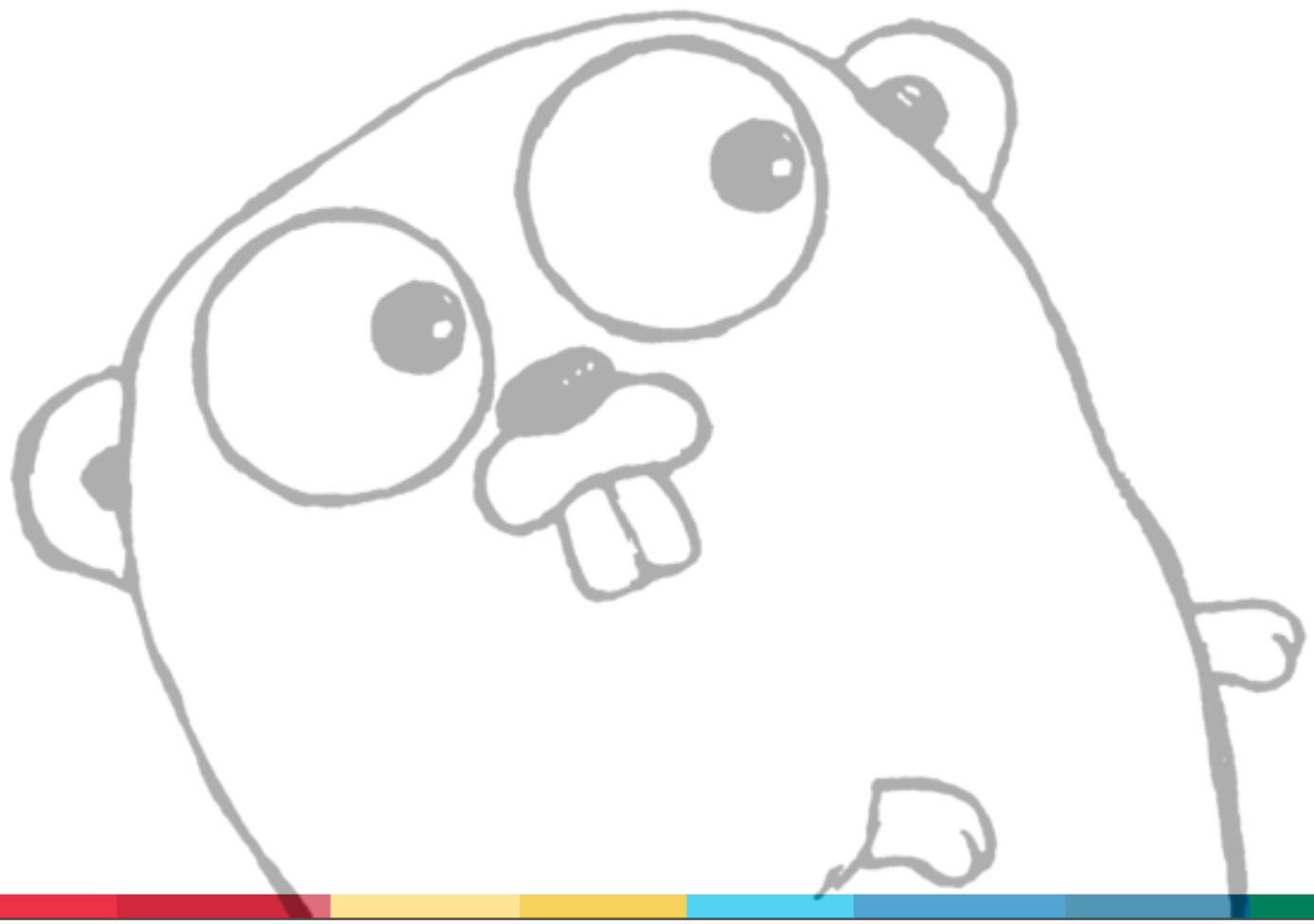
This acceptance was a pleasant surprise.

But in retrospect, the way we approached the design was important to the expressiveness and productivity of Go.

Google

Principles

The axioms of Go's design



Google

Principles

Go is:

Simple
- concepts are easy to understand
  - (the implementation might still be sophisticated)

Orthogonal
- concepts mix cleanly
- easy to understand and to predict what happens

Succinct
- no need to predeclare every intention

Safe
- misbehavior should be detected

These combine to give expressiveness.

Simplicity

Number of keywords is an approximate measure of complexity.

C (K&R)	K&R	32
C++	1991	48
Java	3rd edition	50
C#	2010	77
C++0x	2010	72+11*
JavaScript	ECMA-262	26+16*
Python	2.7	31
Pascal	ISO	35
Modula-2	1980	40
Oberon	1990	32
Go	2010	25

* extra count is for reserved words and alternate spellings

Respect

Go trusts the programmer to write down what is meant.

In turn, Go tries to respect the programmer's intentions.

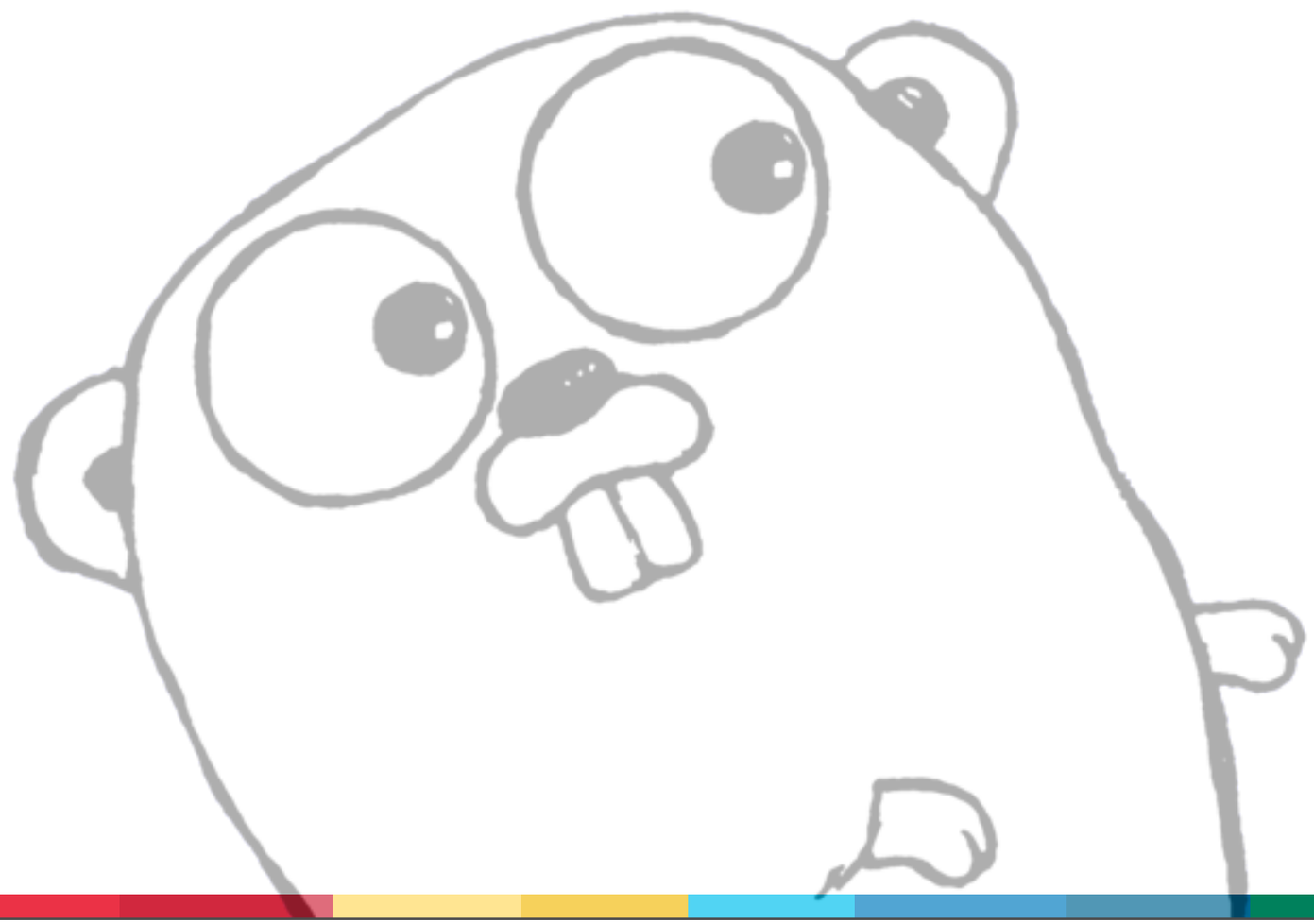
It's possible to be safe and fun.

There's a difference between seat belts and training wheels.

Google

An example

A complete (if simple) web server



Google

Hello, world 2.0

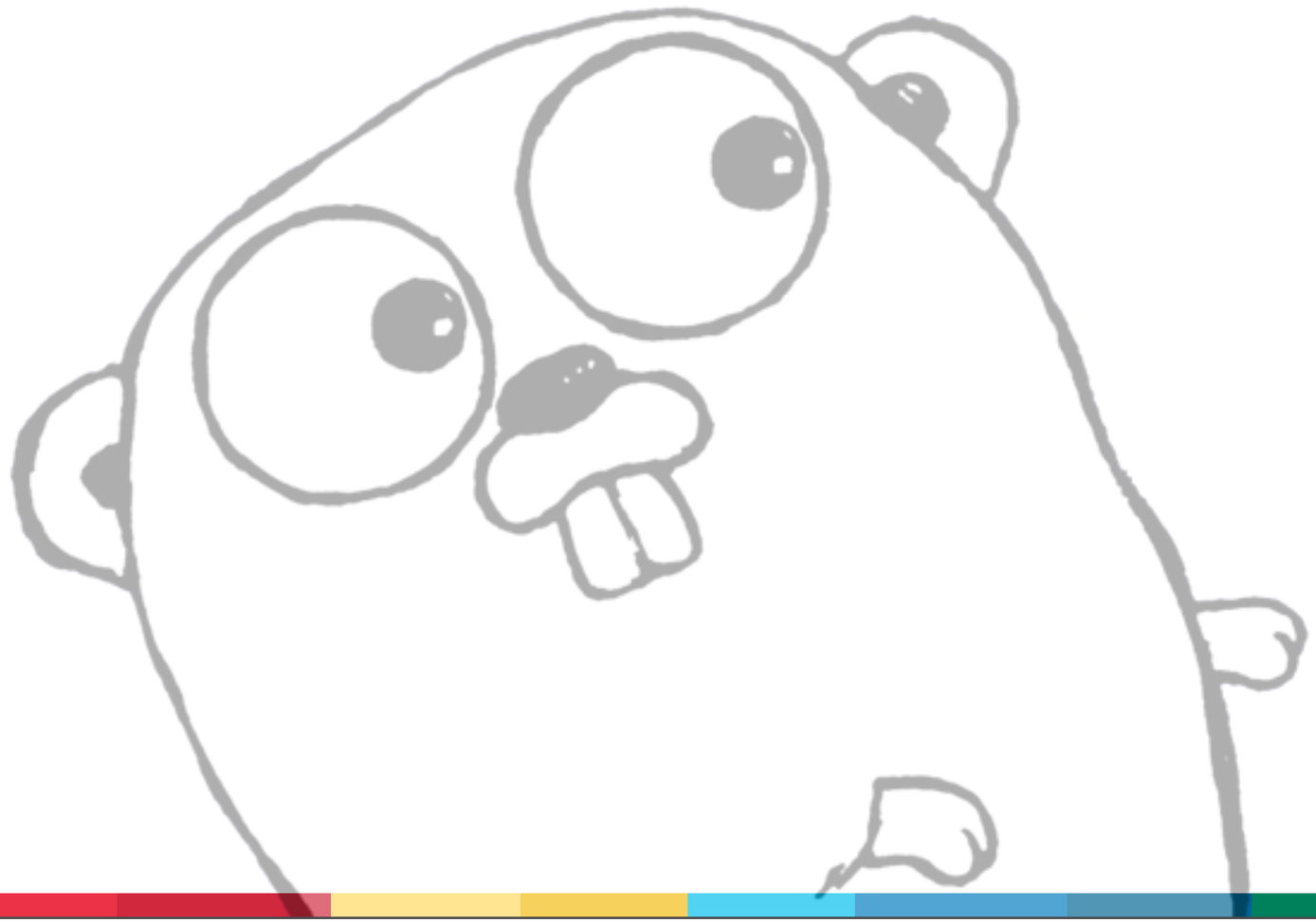
Serving `http://localhost:8080/world`:

```
package main

import (
    "fmt"
    "http"
)

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

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


Google

Stories

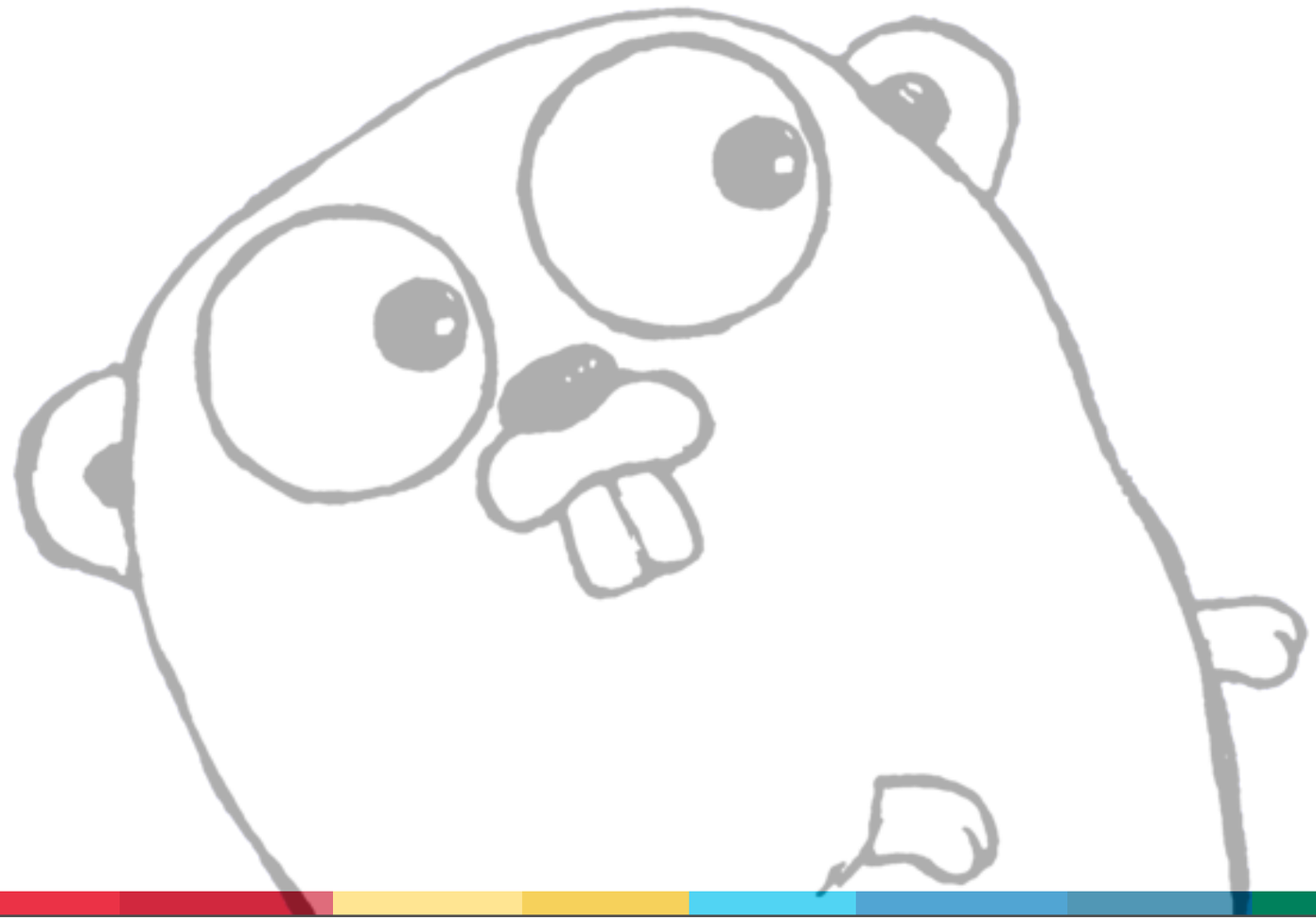
A few design tales that illustrate how the principles play out.
Nowhere near a complete tour of Go.



Google

Basics

Some fundamentals



Google

Starting points

Go started with a few important simplifications relative to C/C++, informed by our experience with those languages:

There are pointers but no pointer arithmetic
- pointers are important to performance, pointer arithmetic not.
- although it's OK to point inside a struct.
  - important to control layout of memory, avoid allocation

Increment/decrement (`p++`) are statements, not expressions.
- no confusion about order of evaluation

Addresses last as long as they are needed.
- take the address of a local variable, the implementation guarantees the memory survives while it's referenced.

No implicit numerical conversions (`float` to `int`, etc.).
- C's "usual arithmetic conversions" are a minefield.

Google

Constants are ideal

Implicit conversions often involve constants (`sin(Pi/4)`), but Go mitigates the issue by having nicer constants.

Constants are "ideal numbers": no size or sign, hence no `L` or `U` or `UL` endings.

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

They are just numbers that can be assigned to variables; no conversions necessary. A typed element in the expression sets the true type of the constant. Here `1e9` becomes `int64`.

```
seconds := time.Nanoseconds()/1e9
```

High precision constants

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

```
const Ln2= 0.69314718055994530941723212214581\
           76568075500134360255254120680009
const Log2E= 1/Ln2 // accurate reciprocal
var x float64 = Log2E
```

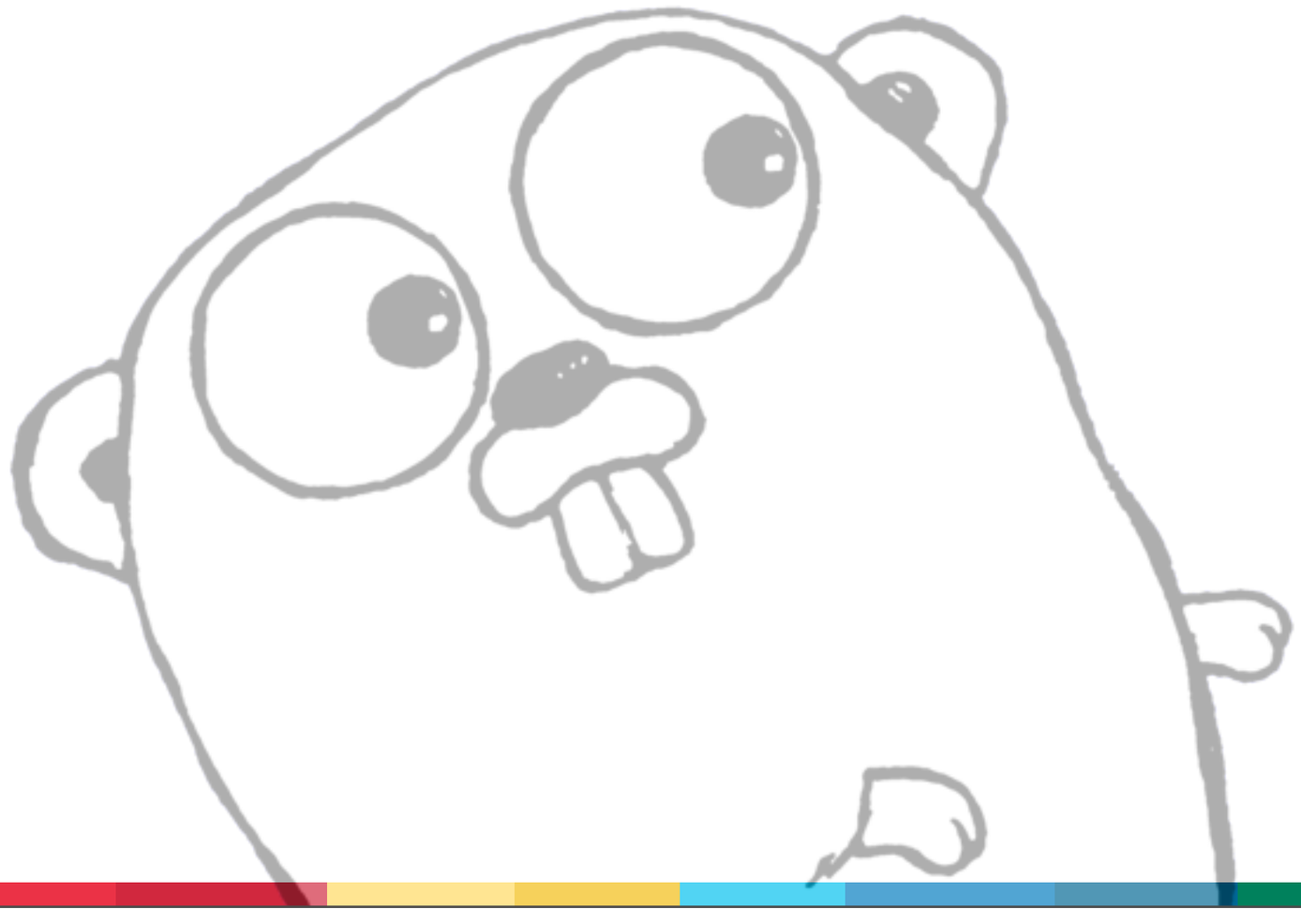
The value assigned to `x` will be as precise as possible in a 64-bit floating point number.

Simple, clear model. Simple constant syntax. Constants orthogonal (nearly) to type system.

Google

Types and data

Structs, methods, and interfaces



Google

Structs

Structs describe (and control) the layout of data.

Some early proposals declared methods in the struct, but that idea was dropped. Instead methods are declared like ordinary functions, outside the type, and with a "receiver".

```
type Point struct { x, y float }
```

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

The `(p Point)` declares the receiver (no automatic "`this`" variable; also notice `p` is not a pointer, although it could be.)

Methods are not mixed with the data definition.
They are orthogonal to types.

Google

Methods are orthogonal to types

Orthogonality of methods allows any type to have them.

```
type Vector []float
func (v Vector) Abs() float {
  sumOfSquares := 0.0
  for i := range v {
    sumOfSquares += v[i]*v[i]
  }
  return math.Sqrt(sumOfSquares)
}
```

It also allows receivers to be values or pointers:

```
func (p *Point) Scale(ratio float) {
  p.x, p.y = ratio*p.x, ratio*p.y
}
```

Orthogonality leads to generality.

Google

Interfaces

Interfaces are just sets of methods; work for any type.

```
type Abser interface {
  Abs() float
}

var a Abser
a = Point{3, 4}
print(a.Abs())
a = Vector{1, 2, 3, 4}
print(a.Abs())
```

Interfaces are satisfied implicitly. `Point` and `Vector` do not declare that they implement `Abser`, they just do.

@mjmoriarity: The way Go handles interfaces is the way I wish every language handled them.

Google

Interfaces are abstract, other types are concrete

In some of our early proposals, interfaces could contain data, but this conflated representation and behavior.
Made them distinct:

- concrete type such as structs define data
- interfaces define behavior

As with methods, now anything can satisfy an interface.

```
type Value float  // basic type (not boxed)
func (v Value) Abs() float {
  if v < 0 { v = -v }
  return v
}

a = Value(-27)
print(a.Abs())
```


Google

Interfaces are satisfied implicitly

`Point` and `Vector` satisfied `Abser` 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. It's also another form of orthogonality.

An object satisfies multiple interfaces simultaneously. `Point` and `Vector` satisfy `Abser` and also the empty interface: `interface{}`, which is satisfied by any value (analogous to C++ `void*` or Java `Object`)

In Go, interfaces are usually one or two (or zero) methods.

Reader

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

Anything with a `Read` method 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

Hello, world 2.0

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

```
package main

import (
    "fmt"
    "http"
)

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

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


Google

Interfaces enable retrofitting

Had an existing RPC implementation that used custom wire format. Wanted to use JSON. Defined an interface pair:

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

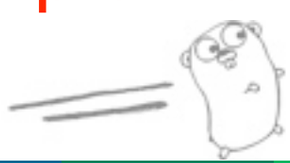
Two functions (send, receive) changed signature. Before:

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

After (and similarly for receiving):

```
func sendResponse(sending *sync.Mutex, req *Request,
        reply interface{}, enc Encoding, err 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`. You'd have to plan ahead.

In Go it's simpler and the design adapts through experience.

Google

Principles redux

Types and data examples:

Simple
- interfaces are just method sets

Orthogonal
- representation (data) and behavior (methods) are independent
- empty interface can represent any value

Succinct
- no `implements` declarations; interfaces just satisfy

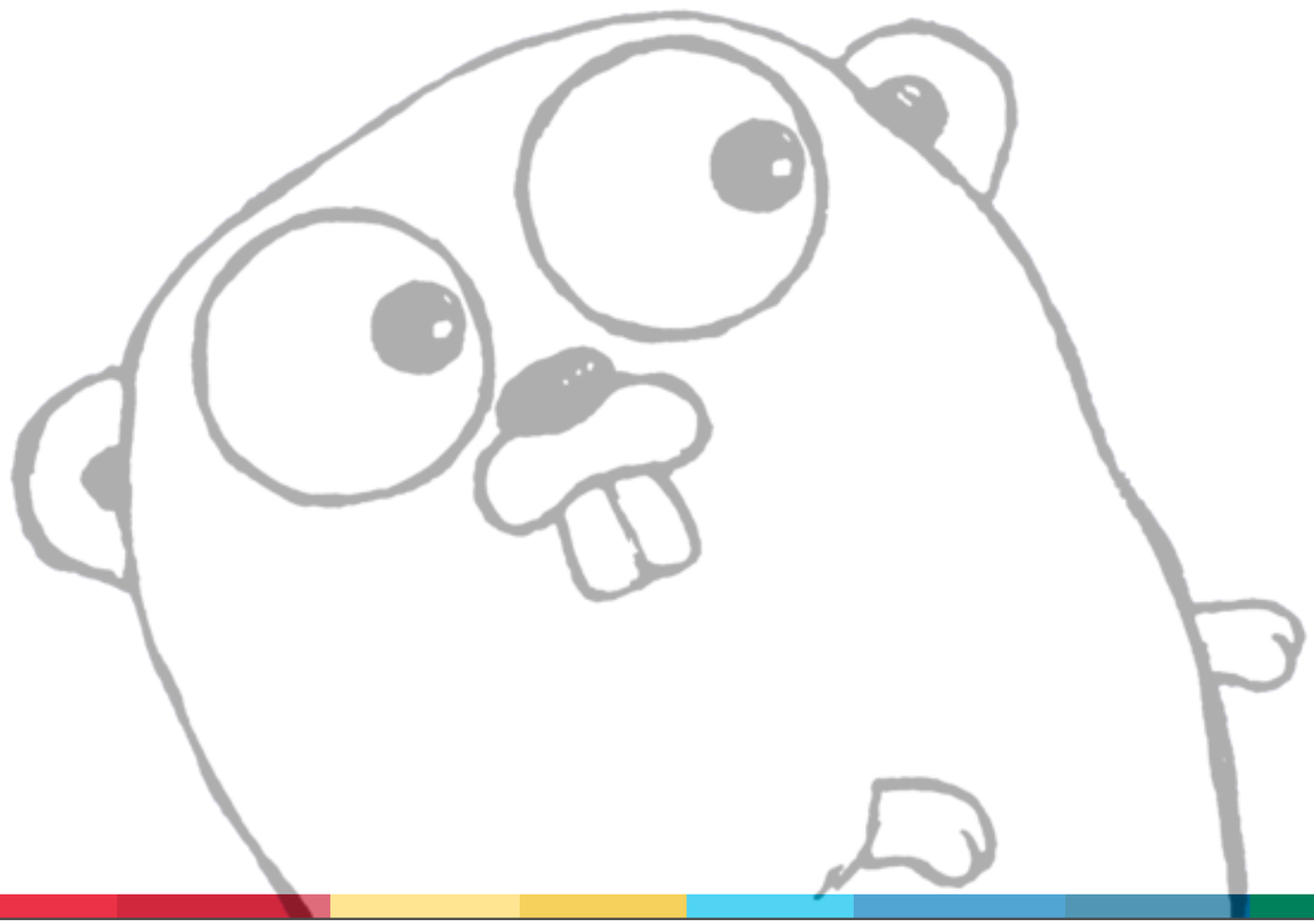
Safe
- static type checking

Expressiveness: implicit satisfaction lets pieces fit together seamlessly - and after the fact.

Google

Names

Visibility



Visibility

With methods not declared as part of the type definition, how to say private/public?

Long design debate with several suggestions:
- placement in the file
- `export` keyword
- name marker (e.g. `Point{ +x, +y int }`)

Resolution (anguished decision): Don't make it part of the declaration, make it part of the name. After all, that's what you see whenever you use it!

Case of first character determines visibility outside package:

```
ThisMethodIsPublic
thisOneIsNot
```


Google

Visibility is orthogonal to type

One of the best decisions in the language, yet really hard to make!

Lose control over how to use case, e.g. can't use it to distinguish Types from variables.

In return, though:

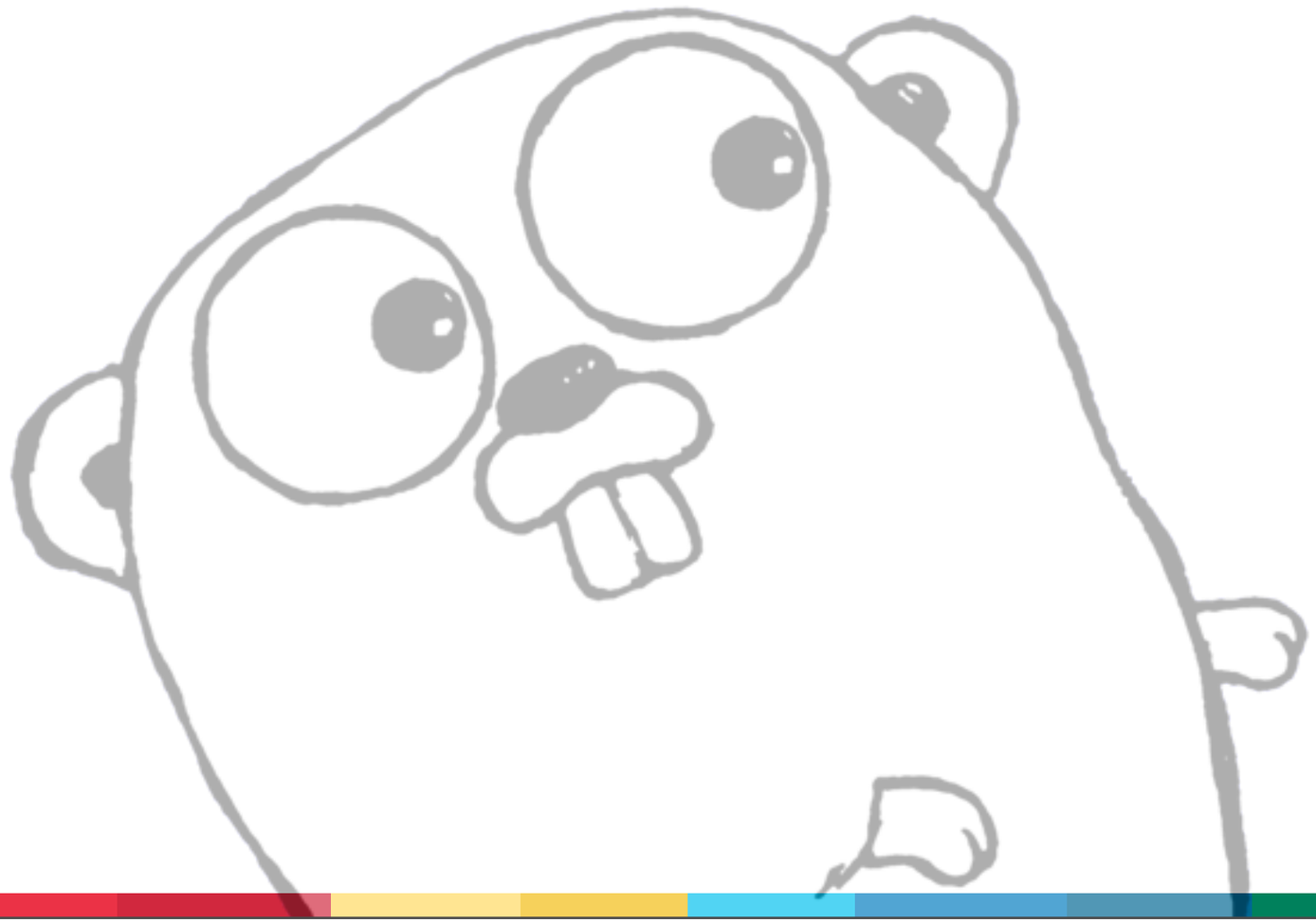
- extremely simple, easy rule to understand
- consequences clear
- see a name, can see whether it's public without going to the declaration
- can make any type, variable, method, function, or constant public or not with the same mechanism

Orthogonality again!

Google

Concurrency and closures

Goroutines, channels, stacks and closures



Google

The model

Go has native support for concurrent operations in the CSP tradition.

Two styles of concurrency:

- deterministic
  - order of operations is well-defined
- non-deterministic
  - use mutual exclusion, order of operations undefined

Go's goroutines and channels promote deterministic concurrency (e.g. channels with one sender, one receiver), which is easier to reason about.

Simpler for the programmer.

Google

Go concurrency basics

Start a goroutine:

```
go f()
```

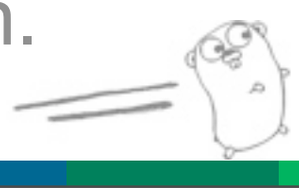
Channel send (arrow points in direction of flow):

```
ch := make(chan int)
go fn(ch)
ch <- 32
```

Channel receive:

```
value = <-ch
```

Channels are unbuffered (synchronous) by default, which combines synchronization with communication.



A simple worker pool

A unit of work:

```
type Work struct { x, y, result int }
```

A worker task:

```
func worker(input <-chan *Work, output chan<- *Work) {
    for w := range input {
        w.result = expensiveComputation(w.x, w.y)
        out <- w
    }
}
```

Driver:

```
func Run() {
    in, out := make(chan *Work), make(chan *Work)
    for i := 0; i < 10; i++ { go worker(in, out) }
    go sendLotsOfWork(in)
    receiveLotsOfResults(out)
}
```


No low-level synchronization needed.

Google

Secondary support

To make concurrency feasible, need several things:

- language support (axiomatic)
- garbage collection (near axiomatic)

To make it good, you need:

- stack management
- closures

Google

Stacks

Goroutines have "segmented stacks":

```
go f()
```

starts `f()` executing concurrently on a new (small) stack.

Stack grows and shrinks as needed.

No programmer concern about stack size.
No possibility for stack overflow.

A couple of instructions of overhead on each function call, a huge improvement in simplicity and expressiveness.

Concurrent execution is orthogonal to everything else.

Google

Launching a goroutine

Start a service, return a channel to communicate with it:

```
func (s *Service) Start() chan<- request {
  ch := make(chan request)
  go s.serve(ch)  // s.serve runs concurrently
  return ch       // returns immediately
}
```

A common pattern, given channels as first-class values.

Google

Closures

Closures are just local functions

```
func Compose(f, g func(x float) float)
                  func(x float) float {
  return func(x float) float {
    return f(g(x))
  }
}

func main() {
  fmt.Println(Compose(sin, cos)(math.Pi/8))
}
```

Fit easily into implementation since local variables already move to heap when necessary.

Google

Closures and concurrency

Query servers in replicated database, return first response.

```
func Query(conns []Conn, query string) Result {
  ch := make(chan Result, 1)  // buffer of 1 item
  for _, conn := range conns {
    go func(c Conn) {
      _ = ch <- c.DoQuery(query)
    }(conn)
  }
  return <-ch
}
```

A complex behavior expressed succinctly because the building blocks (concurrency, communication, closures, garbage collection, ...) are orthogonal.

Google

Principles redux

Concurrency and closure examples:

Simple
- stacks just work; goroutines too cheap to meter

Orthogonal
- concurrency orthogonal to rest of language
- orthogonality of functions make closures easy

Succinct
- `go f()`
- closure syntax clear

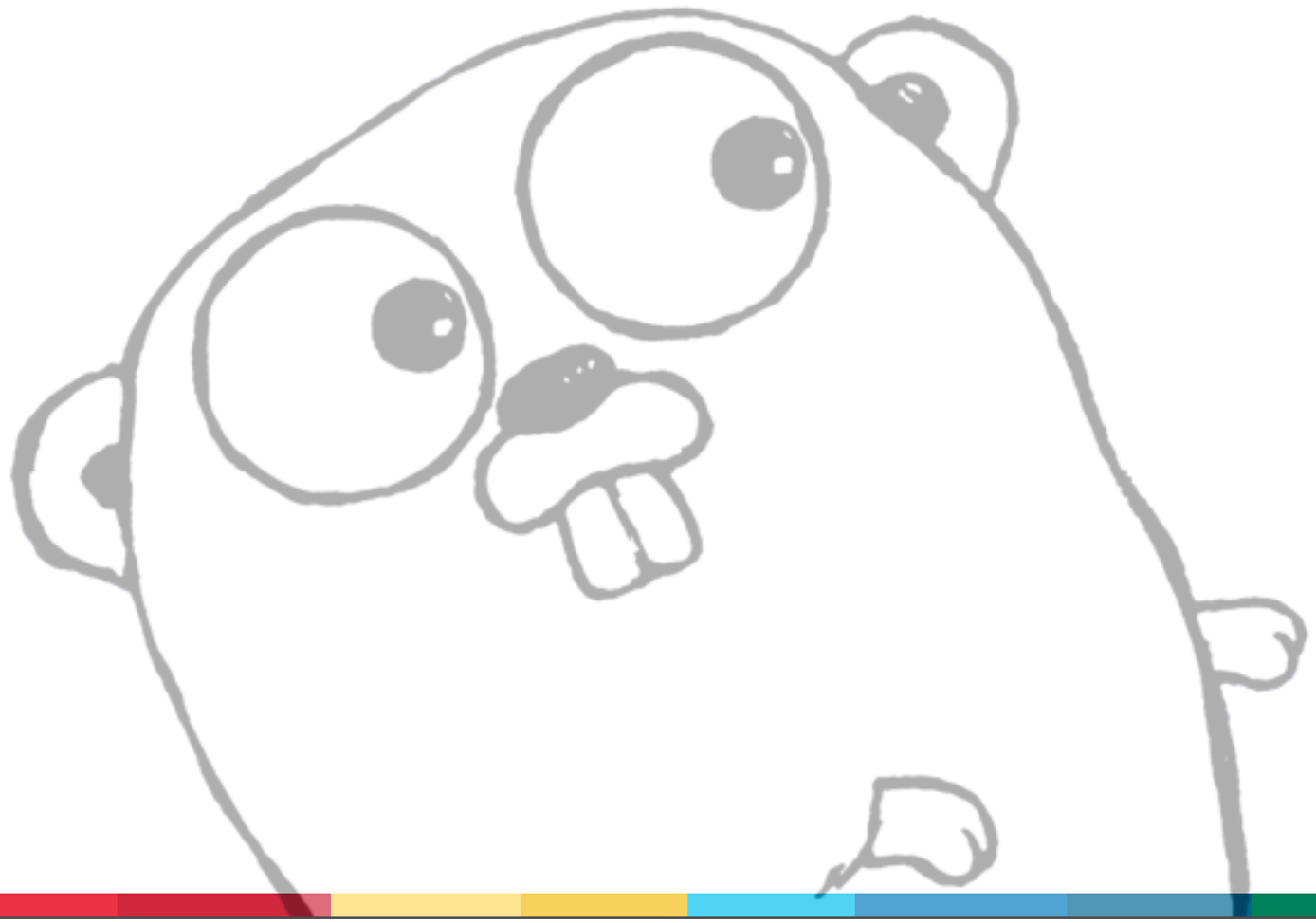
Safe
- no stack overflows
- garbage collection avoids many concurrency problems

Expressiveness: complex behaviors easily expressed.

Google

Conclusion

Expressiveness comes from orthogonal composition of simple concepts.



Google

Conclusion

Go is not a small language (goroutines, channels, garbage collection, methods, interfaces, closures, ...) but it is an expressive and comprehensible one.

Expressiveness comes from orthogonal composition of constructs.

Comprehensibility comes from simple constructs that interact in easily understood ways.

Build a language from simple orthogonal building blocks and you have a language that will be easy and productive to use.

The surprises you discover will be pleasant ones.

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`.

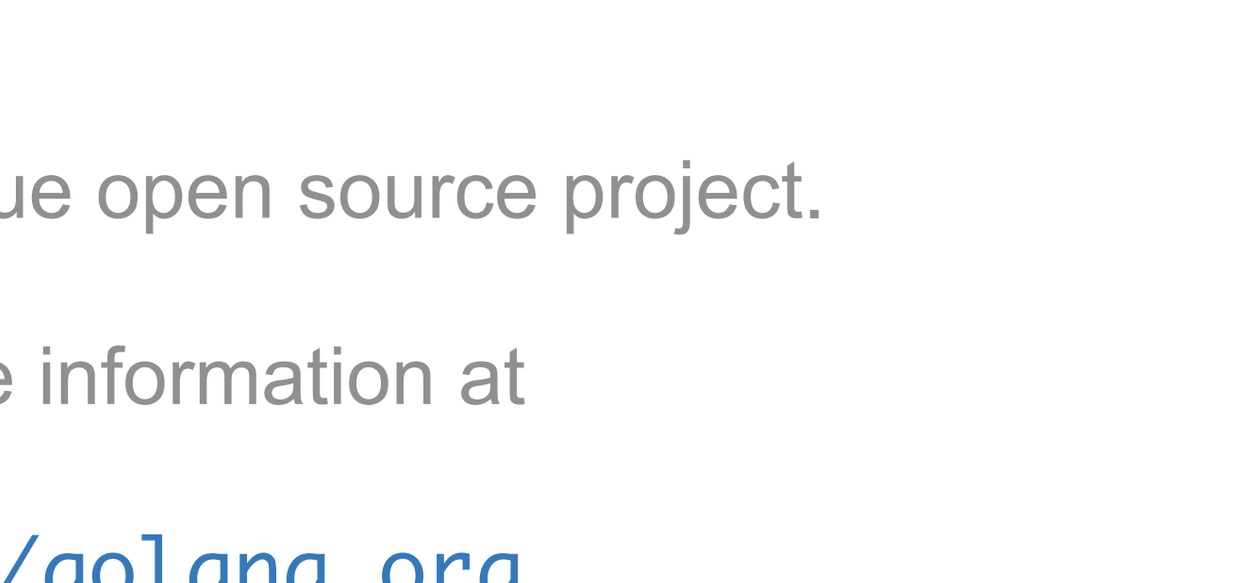
Try it out

This is a true open source project.

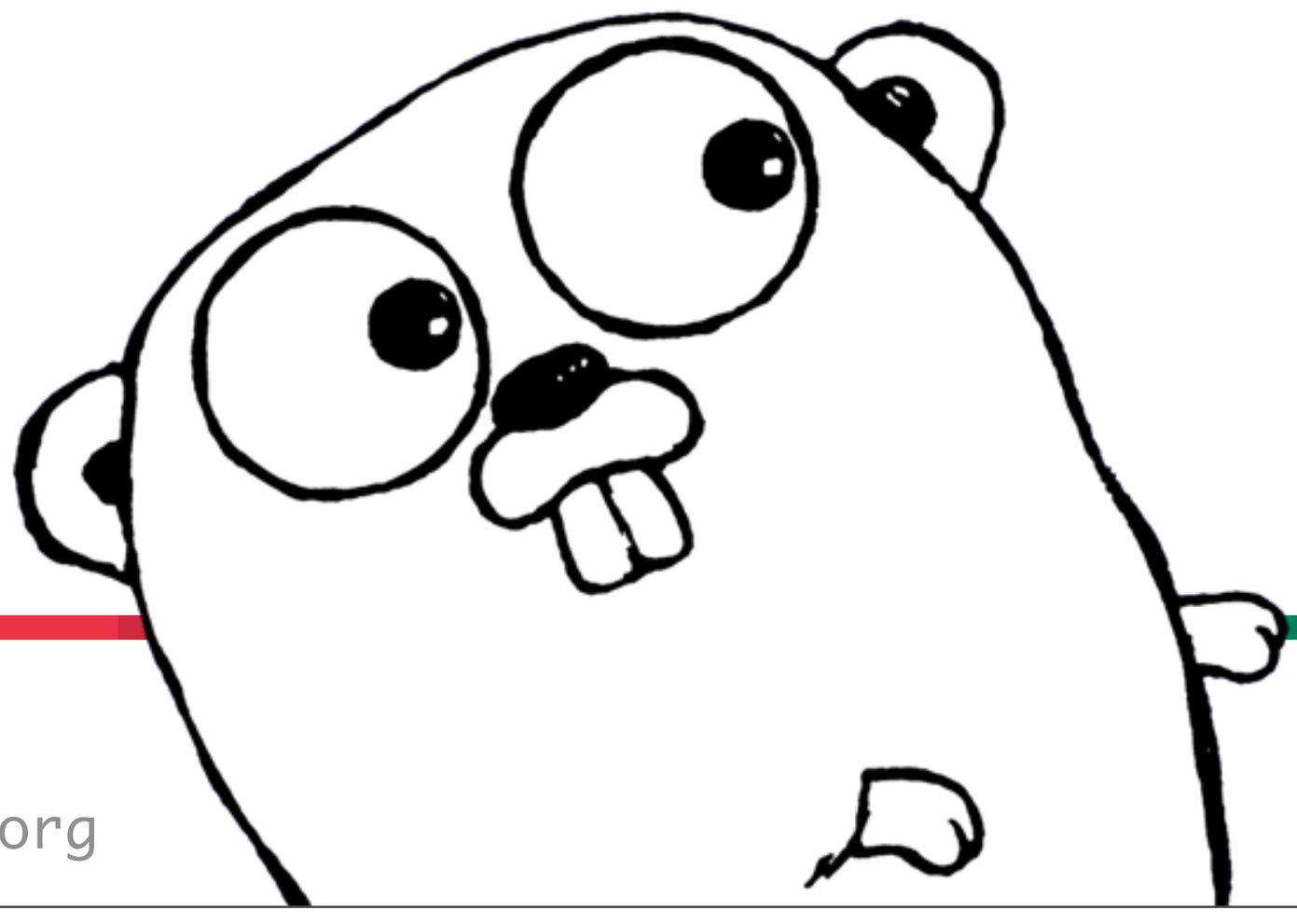
Much more information at

`http://golang.org`

including full source, documentation, and a playground that lets you try Go code right from the browser.



Google



http://golang.org

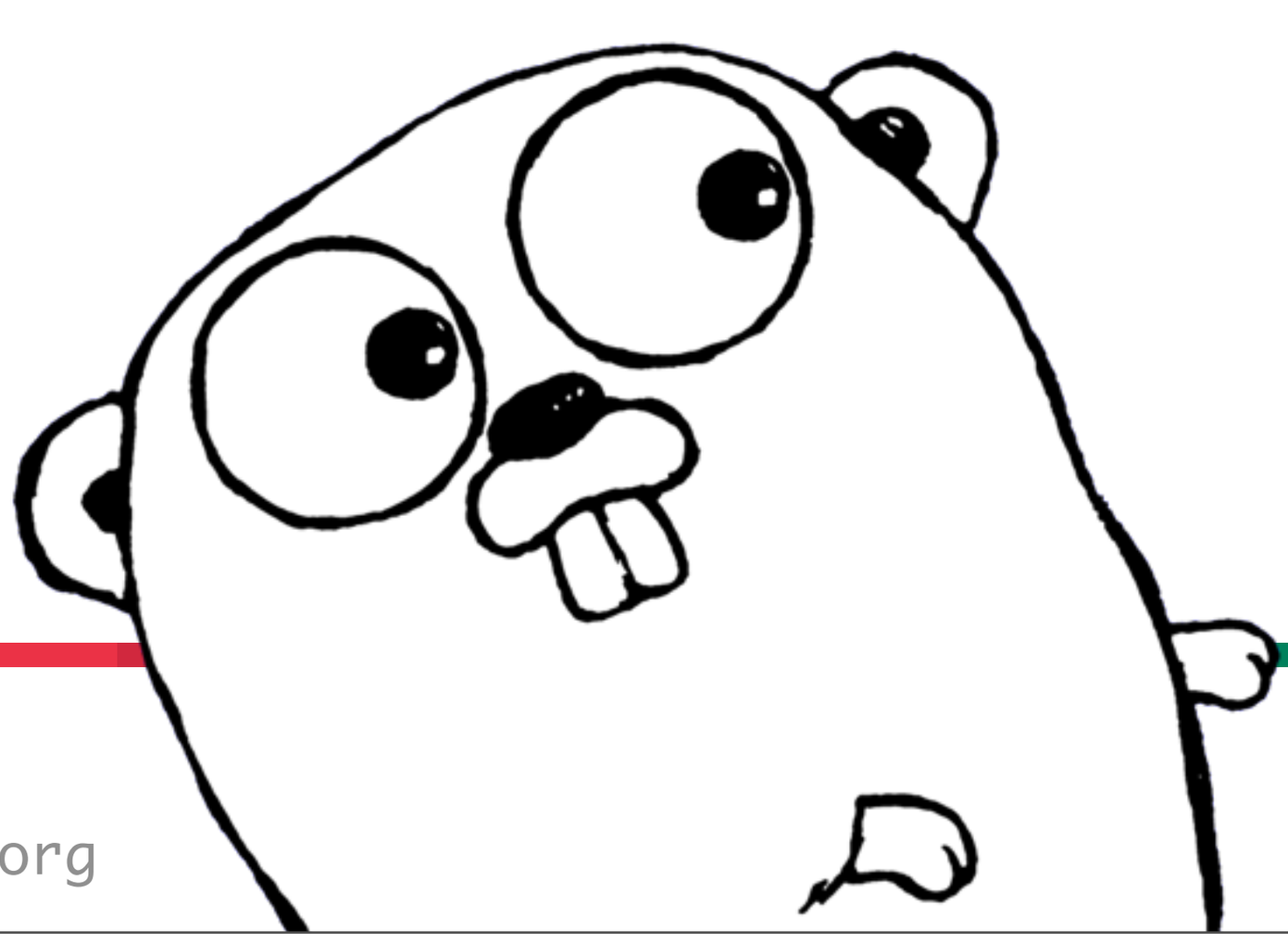
Google

The Expressiveness of Go

Rob Pike

JAOO

Oct 5, 2010



http://golang.org

Google