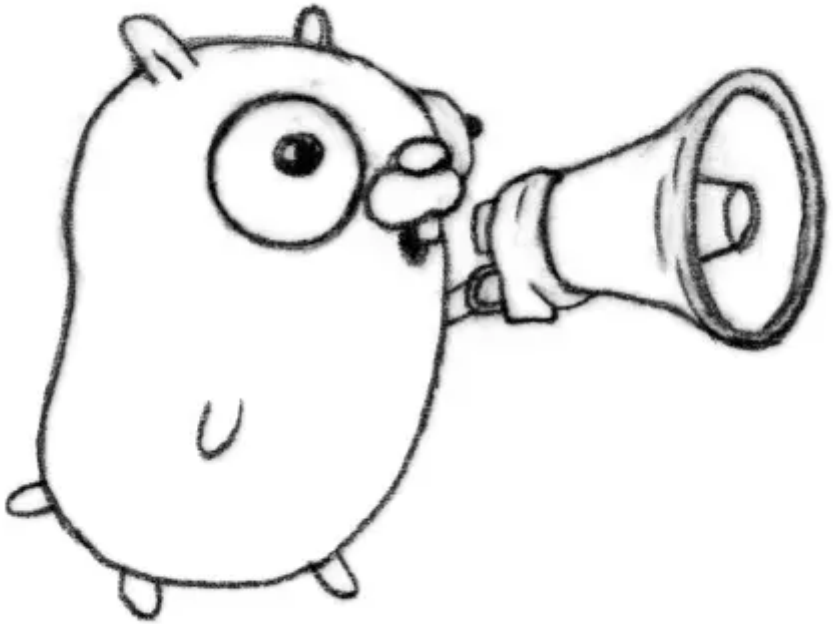
This is Go

vienna.go
26 November 2019

Rodolfo Carvalho

SENTRY ❤ GO

This is Go: a personal tale on what makes Go unique



The Go gopher was designed by Renee French (https://reneefrench.blogspot.com)

PyCon 2007, Computer (Programming) Literacy

Robert M. Lefkowitz (https://github.com/r0ml)

The art of communicating through symbols ideas about reality

Photo by Óscar Ardèvol (https://unsplash.com/photos/e8_2PJXeg84)

#1 Simplicity

"Simplicity is the real reason why Go is successful" — Rob Pike, dotGo 2015

youtu.be/rFejpH_tAHM (https://youtu.be/rFejpH_tAHM)

Simplicity

```
package main

import "fmt"

func main() {
    s := "Hello World"
    fmt.Println(s)
}
```

- Packages
- Fully Qualified Imports
- Static Typing with Type Inference
- Single Dispatch
- Capitalization is Meaningful

Simplicity

- Fits your brain
- Small set of orthogonal features
- Easy to learn
- Easy to read
- Go is boring (https://youtu.be/4Dr8FXs9aJM)!
- Great for larger teams

The art of COMMUNICATING through symbols ideas about reality

#2 gofmt

Gopher Artwork by Ashley McNamara (https://github.com/ashleymcnamara/gophers)

#3 Error values, no exceptions

blog.golang.org/errors-are-values (https://blog.golang.org/errors-are-values)

- Explicit, on-the-spot, error handling
- No especial syntax to learn
- Think about errors upfront

```
_, err = fd.Write(p0[a:b])
if err != nil {
    return err
}
_, err = fd.Write(p1[c:d])
if err != nil {
    return err
}
_, err = fd.Write(p2[e:f])
if err != nil {
    return err
}
// and so on
```

Programming your error handling

```
type errWriter struct {
    w   io.Writer
    err error
}
func (ew *errWriter) write(buf []byte) {
    if ew.err != nil {
        return
    }
    _, ew.err = ew.w.Write(buf)
}
```

Use it:

```
ew := &errWriter{w: fd}
ew.write(p0[a:b])
ew.write(p1[c:d])
ew.write(p2[e:f])
// and so on
if ew.err != nil {
    return ew.err
}
```

The art of communicating through symbols IDEAS about reality

#4 Interfaces

Go interfaces enable post-facto abstraction.

- Python: can't write it down and type-check, you just duck type
- Java: must declare *a priori*, 3rd-party code cannot implement your interfaces
- Implicitly satisfiable interfaces

Frequently used: `io.Reader`, `io.Writer`, `fmt.Stringer`, `interface{}`

Example:

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

The bigger the interface, the weaker the abstraction

Keep them small, and compose!

```
type ReadWriter interface {
    Reader
    Writer
}
```

To implement an interface

Implement methods with the appropriate signatures:

```
type File struct {
    Name string
}

func (f *File) Read(p []byte) (n int, err error) {
    return len(p), nil
}

func (f *File) Write(p []byte) (n int, err error) {
    return len(p), nil
}

func (f *File) String() string {
    return f.Name
}
```

Nothing more.

To define your own interface

Declare the method signatures:

```
type Greeter interface {
    Greet(string) error
}
```

Swiss-knife interface

go-review.googlesource.com/#/c/20763/ (https://go-review.googlesource.com/#/c/20763/)

```
// go/src/net/http/request.go
func (l *maxBytesReader) tooLarge() (n int, err error) {
    if !l.stopped {
        l.stopped = true

        // The server code and client code both use
        // maxBytesReader. This "requestTooLarge" check is
        // only used by the server code. To prevent binaries
        // which only using the HTTP Client code (such as
        // cmd/go) from also linking in the HTTP server, don't
        // use a static type assertion to the server
        // "*response" type. Check this interface instead:
        type requestTooLarger interface {
            requestTooLarge()
        }
        if res, ok := l.w.(requestTooLarger); ok {
            res.requestTooLarge()
        }
    }
    return 0, errors.New("http: request body too large")
}
```

The art of communicating through symbols ideas about REALITY

#5 go test

Because there are no exceptions and because errors are (interface) values, it is easy to write concise table-driven tests.

Define tests:

```
type atoi64Test struct {
    in  string
    out int64
    err error
}

var atoi64tests = []atoi64Test{
    {"", 0, ErrSyntax},
    {"0", 0, nil},
    // ...
    {"9223372036854775809", 1<<63 - 1, ErrRange},
    {"-9223372036854775809", -1 << 63, ErrRange},
}
```

Test them all

```
func TestParseInt64(t *testing.T) {
    for i := range atoi64tests {
        test := &atoi64tests[i]
        out, err := ParseInt(test.in, 10, 64)
        if test.out != out || !reflect.DeepEqual(test.err, err) {
            t.Errorf("Atoi64(%q) = %v, %v want %v, %v",
                test.in, out, err, test.out, test.err)
        }
    }
}
```

Even if a single test case fails, all other tests can still be run.

There are no *assertions*, no *exceptions* to control the test flow, and no need for the "single assert per test method" rule.

More

- benchmarks
- profiling
- tracing
- race detector
- ...

This is Go

“Go is an attempt to combine the **safety** and **performance** of statically typed compiled languages with the **expressiveness** and **convenience**, and the **fun**, of a dynamic typed interpreted language.”

— Rob Pike, OSCON 2010

youtu.be/5kj5ApnhPAE (https://youtu.be/5kj5ApnhPAE)

Recap

Go strives for **simplicity**

Go has great tooling

Go helps you think about errors

Go interfaces allow for composing abstractions

Go empowers us to communicate through symbols ideas about reality

Thank you

Rodolfo Carvalho

rhcarvalho@gmail.com (mailto:rhcarvalho@gmail.com)

https://www.rodolfocarvalho.net (https://www.rodolfocarvalho.net)