Going without

Go Sydney Users' Group
November 2018

State

Global State

What do I mean when I say *global state?*

Universe Scope: base types int, string, bool, etc.

Package Scope: variables, types, methods, functions, constants

File scope: imports declarations

Function Scope: variables, types, constants

Block scope: variables, types, constants

Mutable Global State

Why is mutable global state bad?

Coupling

Package level variables exhibit source and run-time coupling.

Any code that can import my package can change the value of a public variable declared in my package.

If my program depends on the type of your package's public variable, if that changes, my program may not compile.

If my program depends on the contents of your package's public variable, if the contents change, my program will compile, but its behaviour may change.

Mutability

If my program depends on the contents of your package's public variable, if the contents change at run time, that's a data race.

A public variable cannot be safely mutated in the presence of multiple goroutines. Anyone attempting to do this must guard the mutation of a public variable using a mutex

Testability

Global variables act like hidden parameters to every method or function a program.

The every global variable in your program can be used to smuggle state between unrelated concerns. If your functions rely on that state, your tests need to closely manage that state.

Global variables are unique, they can only have one value at a time. Wave goodbye to test parallelism

Going without package level variables

What would Go look like if we could no longer declare variables at the package level?

What would be the impact of removing package scoped variable declaraions from Go?

What could we learn about the design of Go programs by doing so?

n.b. I'm only talking about eliminating package level variables, the other top level declarations are untouched.

Is that even possible?


Introduction to the
OLD TESTAMENT

Registration pattern

A registration pattern is followed by several packages in the standard library; `net/http`, `database/sql`, `flag`, and to a lesser extent `log`.

Registation involves an unexported package level map or struct which is mutated by a public function—a textbook singleton.

Registration pattern

Disallowing a package scoped placeholder for this state would remove the side effects in the `image`, `database/sql`, and `crypto` packages to register image decoders, database drivers and cryptographic schemes.

However, this is precisely the spooky action at a distance that the registration pattern encourages.

net/http.DefaultServeMux

When imported net/http/pprof registers a bunch of handlers with, and only with, http.DefaultServeMux

Potentuial security issue, other code cannot use DefaultServeMux—or code that uses it indirectly—without exposing the /debug/pprof endpoint.

The registration side effect makes it difficult to persuade net/http/pprof to register its handlers with a different mux.

Registration promotes confusing APIs

```
package http

// Serve accepts incoming HTTP connections on the listener
// l, creating a new service goroutine for each. The service
// goroutines read requests and then call handler to
// reply to them.
//
// The handler is typically nil, in which case the
// DefaultServeMux is used.
func Serve(l net.Listener, handler Handler) error
```

There's more than one way to do it

```
var l net.Listerner = ...

http.Serve(l, nil)
http.Serve(l, http.DefaultServeMux)
```

One nil good, two nils bad

```
http.Serve(nil, nil) // panic
```

```
const root = http.Dir("/htdocs")
http.Handle("/", http.FileServer(root))
http.ListenAndServe("0.0.0.0:8080", nil)
```

```
const root = http.Dir("/htdocs")
http.Handle("/", http.FileServer(root))
http.ListenAndServe("0.0.0.0:8080", http.DefaultServeMux)
```

```
const root = http.Dir("/htdocs")
mux := http.NewServeMux()
mux.Handle("/", http.FileServer(root))
http.ListenAndServe("0.0.0.0:8080", mux)
```

Eschew registrations

If package scoped variables were no longer used, rather than relying on import side effects, packages should provide a function that registers them with the supplied object.

```
package pprof

// Register registers handlers for
// /debug/pprof.
func Register(mux http.ServeMux)
```

Modules might ruin the party

Avoiding the registry pattern avoids issues with multiple copies of the same package registering themselves during `init()`

This is likely to become more common as projects migrate to Go modules.

If you're unlucky, updating your `go.mod` file causes your application to panic on startup.

Sentinel errors

`io.EOF, sql.ErrNoRows,crypto/x509.ErrUnsupportedAlgorithm, …`

Sentinel errors

Sentinel errors introduce strong source and run-time coupling

To compare the error you have with the error you expect you need to import the package that declares that error.

`IO.EOF` IS A PUBLIC VARIABLE

`io.EOF` is a public variable. Any code that imports the `io` package could change the value of `io.EOF`.

```
package nelson

import "io"

func init() {
  io.EOF = nil // haha!
}
```

```
fmt.Println(io.EOF == io.EOF) // true
x := io.EOF
fmt.Println(io.EOF == x)      // true

io.EOF = fmt.Errorf("whoops")
fmt.Println(io.EOF == io.EOF) // true
fmt.Println(x == io.EOF)      // false
```

`IO.EOF` IS A SINGLETON, NOT A CONSTANT

`io.EOF` behaves like a singleton, not a constant.

```
err := errors.New("EOF")   // io/io.go line 38
fmt.Println(io.EOF == err) // false
```

Sentinel errors without public variables

If we prohibit package scoped variables this would remove the ability to use public variables for sentinel error values.

If we can't declare errors as variables, what could be used to replace them?

Ideally sentinel value should behave as constants. They should be immutable and fungible.

Error interface recap

Any type with an
`Error() string`

method fulfils the `error` interface.

This includes primitive types like `string`,
specifically constant strings.

Constant errors

```
type Error string

func (e Error) Error() string {
        return string(e)
}
```

Constant errors are constants

```
const err = Error("EOF")
const err2 = errorString{"EOF"} //
const initializer errorString literal
is not a constant
```

Constant errors are immutable

```
const err Error = "EOF"
err = Error("not EOF")
// error, cannot assign to err
```

Constant errors are equal if their values are equal

```
const err = Error("EOF")
fmt.Println(err == Error("EOF")) // true
```

```
type Reader struct{}

func (r *Reader) Read([]byte) (int, error) {
        return 0, errors.New("eof")
}

func main() {
        var eof = errors.New("eof")
        var r Reader
        _, err := r.Read([]byte{})
        fmt.Println(err == eof) // false
}
```

```
type Reader struct{}

func (r *Reader) Read([]byte) (int, error) {
        return 0, Error("eof")
}

func main() {
        const eof Error = "eof"
        var r Reader
        _, err := r.Read([]byte{})
        fmt.Println(err == eof) // true
}
```

Could we change the definition of io.EOF to be a constant?

It turns out that this compiles just fine and passes almost all the tests (when I first did this experiment two years ago ./all.bash passed cleanly).

It's not worth fixing the one failing test to prove a point.

But this does not mean you cannot use a constant error type if you need to declare a sentinel error.

What about private error variables?

```
package sql

var errDBClosed = errors.New("sql:
database is closed")
```

Interface satisfaction assertions

The interface satisfaction idiom

```
var _ SomeInterface = new(SomeType)
```

occurred at least 19 times in the standard library.

```
func TestSomeTypeImplementsSomeInterface(t *testing.T) {
        // won't compile if SomeType does not
        // implement SomeInterface
        var _ SomeInterface = new(SomeType)
}
```

```
func TestSomeTypeImplementsSomeInterface(t *testing.T) {
        var i interface{} = new(SomeType)
        if _, ok := i.(SomeInterface); !ok {
                t.Fatalf(“%t doesn’t implement SomeInterface", i)
        }
}
```

Real singletons

`os.Stdin, os.Stdout, os.Stderr`

```
package os

var (
	Stdin  = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
	Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
	Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
)
```

```
type readfd int

func (r readfd) Read(buf []byte) (int, error) {
    return syscall.Read(int(r), buf)
}

type writefd int

func (w writefd) Write(buf []byte) (int, error) {
    return syscall.Write(int(w), buf)
}

const (
    Stdin  = readfd(0)
    Stdout = writefd(1)
    Stderr = writefd(2)
)

func main() {
    fmt.Fprintf(Stdout, "Hello world")
}
```

A bridge too far?

Is it possible to eliminate package level variables? No

Should we act as if it was? Yes

A MIDDLE GROUND

Avoid public variables. A variable that can be changed by any party that knows its name is a red flag.

Replace public variable declarations with constants where possible

Replace public variables with function parameters and striuct fields.

Use interfaces to declare—without stipulating how—the behaviour your function requires.

One more thing

```
func main() {
    stop := make(chan int)
    go func() {
        for {
            select {
            case <-stop:
                return
            default:
                p, _ := filepath.Abs("file.txt")
                fmt.Println(p)
            }
        }
    }()

    d1, _ := ioutil.TempDir("", "")
    d2, _ := ioutil.TempDir("", "")
    for i := 0; i < 20; i++ {
        os.Chdir(d1)
        os.Chdir(d2)
    }
}
```

```
% go run chdir.go
/Users/dfc/iCloud/presentations/file.txt
/Users/dfc/iCloud/presentations/file.txt
/private/var/folders/by/3gf34_z95zg05cyj744_vhx40000gn/T/409841509/file.txt
/private/var/folders/by/3gf34_z95zg05cyj744_vhx40000gn/T/409841509/file.txt
/private/var/folders/by/3gf34_z95zg05cyj744_vhx40000gn/T/155712896/file.txt
/private/var/folders/by/3gf34_z95zg05cyj744_vhx40000gn/T/409841509/file.txt
/private/var/folders/by/3gf34_z95zg05cyj744_vhx40000gn/T/409841509/file.txt
/private/var/folders/by/3gf34_z95zg05cyj744_vhx40000gn/T/155712896/file.txt
```

```
package main

import "log"

func main() {
        if err := Main(); err != nil {
                log.Fatal(err)
        }
}

func Main() error {
        // actual main function
}
```

```
type Context struct {
        Stdin          io.Reader
        Stdout, Stderr io.Writer
        Workdir string
}

func main() {
        wd, err := os.Getwd()
        check(err)

        ctx := &Context{
                Stdin:   os.Stdin,
                Stdout:  os.Stdout,
                Stderr:  os.Stderr,
                Workdir: wd,
        }
        err = Main(ctx)
        check(err)
}

func check(err error) {
        if err != nil {
                log.Fatal(err)
        }
}
```

Are these *really* singletons?

```
os.Stdin, os.Stdout, os.Stderr
os.Getwd()
os.Args()
os.Getenv()
time.Now()
```

Thank you!

Have a happy and safe new year!
Thank you for supporting the meet up!

Image: Egon Elbre