Twelve Go Best Practices

Francesc Campoy Flores
Gopher at Google

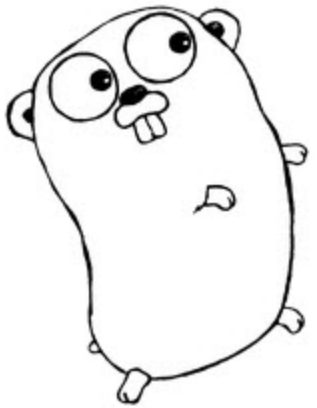
Best practices

From Wikipedia:

```
"A best practice is a method or technique that has consistently shown results superior
to those achieved with other means"
```

Techniques to write Go code that is

- simple,
- readable,
- maintainable.



Some code

```
type Gopher struct {
    Name     string
    Age      int32
    FurColor color.Color
}
```

```
func (g *Gopher) DumpBinary(w io.Writer) error {
    err := binary.Write(w, binary.LittleEndian, int32(len(g.Name)))
    if err == nil {
        _, err := w.Write([]byte(g.Name))
        if err == nil {
            err := binary.Write(w, binary.LittleEndian, g.Age)
            if err == nil {
                return binary.Write(w, binary.LittleEndian, g.FurColor)
            }
            return err
        }
        return err
    }
    return err
}
```

Avoid nesting by handling errors first

```
func (g *Gopher) DumpBinary(w io.Writer) error {
    err := binary.Write(w, binary.LittleEndian, int32(len(g.Name)))
    if err != nil {
        return err
    }
    _, err = w.Write([]byte(g.Name))
    if err != nil {
        return err
    }
    err = binary.Write(w, binary.LittleEndian, g.Age)
    if err != nil {
        return err
    }
    return binary.Write(w, binary.LittleEndian, g.FurColor)
}
```

Less nesting means less cognitive load on the reader

Avoid repetition when possible

Deploy one-off utility types for simpler code

```
type binWriter struct {
    w   io.Writer
    err error
}

// Write writes a value into its writer using little endian.
func (w *binWriter) Write(v interface{}) {
    if w.err != nil {
        return
    }
    w.err = binary.Write(w.w, binary.LittleEndian, v)
}
```

```
func (g *Gopher) DumpBinary(w io.Writer) error {
    bw := &binWriter{w: w}
    bw.Write(int32(len(g.Name)))
    bw.Write([]byte(g.Name))
    bw.Write(g.Age)
    bw.Write(g.FurColor)
    return bw.err
}
```

Type switch to handle special cases

```
// Write writes a value into its writer using little endian.
func (w *binWriter) Write(v interface{}) {
    if w.err != nil {
        return
    }
    switch v.(type) {
    case string:
        s := v.(string)
        w.Write(int32(len(s)))
        w.Write([]byte(s))
    default:
        w.err = binary.Write(w.w, binary.LittleEndian, v)
    }
}
```

```
func (g *Gopher) DumpBinary(w io.Writer) error {
    bw := &binWriter{w: w}
    bw.Write(g.Name)
    bw.Write(g.Age)
    bw.Write(g.FurColor)
    return bw.err
}
```

Type switch with short variable declaration

```
// Write write the given value into the writer using little endian.
func (w *binWriter) Write(v interface{}) {
    if w.err != nil {
        return
    }
    switch v := v.(type) {
    case string:
        w.Write(int32(len(v)))
        w.Write([]byte(v))
    default:
        w.err = binary.Write(w.w, binary.LittleEndian, v)
    }
}
```

Function adapters

```
func init() {
    http.HandleFunc("/", handler)
}

func handler(w http.ResponseWriter, r *http.Request) {
    err := doThis()
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        log.Printf("handling %q: %v", r.RequestURI, err)
        return
    }

    err = doThat()
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        log.Printf("handling %q: %v", r.RequestURI, err)
        return
    }
}
```

Function adapters

```
func init() {
    http.HandleFunc("/", errorHandler(betterHandler))
}

func errorHandler(f func(http.ResponseWriter, *http.Request) error) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        err := f(w, r)
        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
            log.Printf("handling %q: %v", r.RequestURI, err)
        }
    }
}

func betterHandler(w http.ResponseWriter, r *http.Request) error {
    if err := doThis(); err != nil {
        return fmt.Errorf("doing this: %v", err)
    }

    if err := doThat(); err != nil {
        return fmt.Errorf("doing that: %v", err)
    }
    return nil
}
```

Organizing your code

Important code goes first

License information, build tags, package documentation.

Import statements, related groups separated by blank lines.

```
import (
    "fmt"
    "io"
    "log"

    "code.google.com/p/go.net/websocket"
)
```

The rest of the code starting with the most significant types, and ending with helper function and types.

Document your code

Package name, with the associated documentation before.

```
// Package playground registers an HTTP handler at "/compile" that
// proxies requests to the golang.org playground service.
package playground
```

Exported identifiers appear in godoc, they should be documented correctly.

```
// Author represents the person who wrote and/or is presenting the document.
type Author struct {
    Elem []Elem
}

// TextElem returns the first text elements of the author details.
// This is used to display the author' name, job title, and company
// without the contact details.
func (p *Author) TextElem() (elems []Elem) {
```

Generated documentation

Gocode: documenting Go code

Shorter is better

or at least *longer is not always better*.

Try to find the **shortest name that is self explanatory**.

- Prefer `MarshalIndent` to `MarshalWithIndentation`.

Don't forget that the package name will appear before the identifier you chose.

- In package `encoding/json` we find the type `Encoder`, not `JSONEncoder`.
- It is referred as `json.Encoder`.

Packages with multiple files

Should you split a package into multiple files?

- Avoid very long files

The `net/http` package from the standard library contains 15734 lines in 47 files.

- Separate code and tests

`net/http/cookie.go` and `net/http/cookie_test.go` are both part of the `http` package.

Test code is compiled **only** at test time.

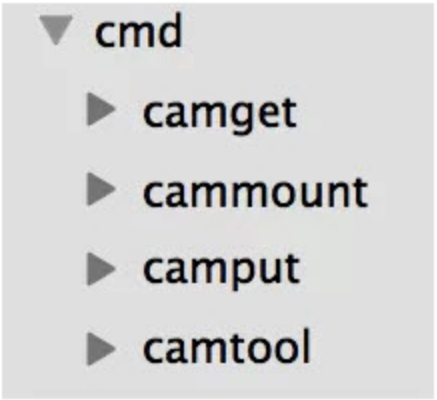
- Separated package documentation

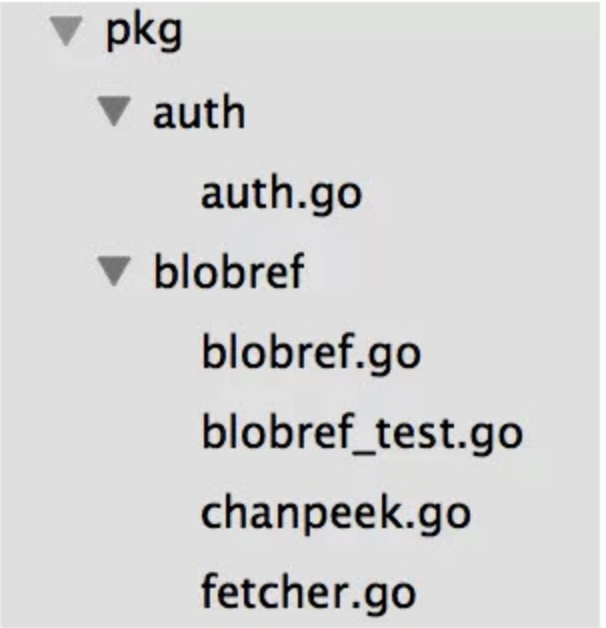
When we have more than one file in a package, it's convention to create a `doc.go` containing the package documentation.

Make your packages "go get"-able

Some packages are potentially reusable, some others are not.

A package defining some network protocol might be reused while one defining an executable command may not.





github.com/bradfitz/camlistore

APIs

Ask for what you need

Let's use the Gopher type from before

```
type Gopher struct {
    Name     string
    Age      int32
    FurColor color.Color
}
```

We could define this method

```
func (g *Gopher) DumpToFile(f *os.File) error {
```

But using a concrete type makes this code difficult to test, so we use an interface.

```
func (g *Gopher) DumpToReadWriter(rw io.ReadWriter) error {
```

And, since we're using an interface, we should ask only for the methods we need.

```
func (g *Gopher) DumpToWriter(f io.Writer) error {
```

Keep independent packages independent

```
import (
    "code.google.com/p/go.talks/2013/bestpractices/funcdraw/drawer"
    "code.google.com/p/go.talks/2013/bestpractices/funcdraw/parser"
)
```

```
    // Parse the text into an executable function.
    f, err := parser.Parse(text)
    if err != nil {
        log.Fatalf("parse %q: %v", text, err)
    }

    // Create an image plotting the function.
    m := drawer.Draw(f, *width, *height, *xmin, *xmax)

    // Encode the image into the standard output.
    err = png.Encode(os.Stdout, m)
    if err != nil {
        log.Fatalf("encode image: %v", err)
    }
```

Parsing

```
type ParsedFunc struct {
    text string
    eval func(float64) float64
}

func Parse(text string) (*ParsedFunc, error) {
    f, err := parse(text)
    if err != nil {
        return nil, err
    }
    return &ParsedFunc{text: text, eval: f}, nil
}

func (f *ParsedFunc) Eval(x float64) float64 { return f.eval(x) }
func (f *ParsedFunc) String() string         { return f.text }
```

Drawing

```
import (
    "image"

    "code.google.com/p/go.talks/2013/bestpractices/funcdraw/parser"
)

// Draw draws an image showing a rendering of the passed ParsedFunc.
func DrawParsedFunc(f parser.ParsedFunc) image.Image {
```

Avoid dependency by using an interface.

```
import "image"

// Function represent a drawable mathematical function.
type Function interface {
    Eval(float64) float64
}

// Draw draws an image showing a rendering of the passed Function.
func Draw(f Function) image.Image {
```

Testing

Using an interface instead of a concrete type makes testing easier.

```
package drawer

import (
    "math"
    "testing"
)

type TestFunc func(float64) float64

func (f TestFunc) Eval(x float64) float64 { return f(x) }

var (
    ident = TestFunc(func(x float64) float64 { return x })
    sin   = TestFunc(math.Sin)
)

func TestDraw_Ident(t *testing.T) {
    m := Draw(ident)
    // Verify obtained image.
```

Avoid concurrency in your API

```
func doConcurrently(job string, err chan error) {
    go func() {
        fmt.Println("doing job", job)
        time.Sleep(1 * time.Second)
        err <- errors.New("something went wrong!")
    }()
}

func main() {
    jobs := []string{"one", "two", "three"}

    errc := make(chan error)
    for _, job := range jobs {
        doConcurrently(job, errc)
    }
    for _ = range jobs {
        if err := <-errc; err != nil {
            fmt.Println(err)
        }
    }
}
```

Run

What if we want to use it sequentially?

Avoid concurrency in your API

```
func do(job string) error {
    fmt.Println("doing job", job)
    time.Sleep(1 * time.Second)
    return errors.New("something went wrong!")
}

func main() {
    jobs := []string{"one", "two", "three"}

    errc := make(chan error)
    for _, job := range jobs {
        go func(job string) {
            errc <- do(job)
        }(job)
    }
    for _ = range jobs {
        if err := <-errc; err != nil {
            fmt.Println(err)
        }
    }
}
```

Run

Expose synchronous APIs, calling them concurrently is easy.

Best practices for concurrency

Use goroutines to manage state

Use a chan or a struct with a chan to communicate with a goroutine

```
type Server struct{ quit chan bool }

func NewServer() *Server {
    s := &Server{make(chan bool)}
    go s.run()
    return s
}

func (s *Server) run() {
    for {
        select {
        case <-s.quit:
            fmt.Println("finishing task")
            time.Sleep(time.Second)
            fmt.Println("task done")
            s.quit <- true
            return
        case <-time.After(time.Second):
            fmt.Println("running task")
        }
    }
}
```

Use goroutines to manage state (continued)

```
func (s *Server) Stop() {
    fmt.Println("server stopping")
    s.quit <- true
    <-s.quit
    fmt.Println("server stopped")
}

func main() {
    s := NewServer()
    time.Sleep(2 * time.Second)
    s.Stop()
}
```

Run

Avoid goroutine leaks with buffered chans

```
func sendMsg(msg, addr string) error {
    conn, err := net.Dial("tcp", addr)
    if err != nil {
        return err
    }
    defer conn.Close()
    _, err = fmt.Fprint(conn, msg)
    return err
}
```

```
func main() {
    addr := []string{"localhost:8080", "http://google.com"}
    err := broadcastMsg("hi", addr)

    time.Sleep(time.Second)

    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("everything went fine")
}
```

Avoid goroutine leaks with buffered chans (continued)

```
func broadcastMsg(msg string, addrs []string) error {
    errc := make(chan error)
    for _, addr := range addrs {
        go func(addr string) {
            errc <- sendMsg(msg, addr)
            fmt.Println("done")
        }(addr)
    }

    for _ = range addrs {
        if err := <-errc; err != nil {
            return err
        }
    }
    return nil
}
```

Run

- the goroutine is blocked on the chan write
- the goroutine holds a reference to the chan
- the chan will never be garbage collected

Avoid goroutines leaks with buffered chans (continued)

```
func broadcastMsg(msg string, addrs []string) error {
    errc := make(chan error, len(addrs))
    for _, addr := range addrs {
        go func(addr string) {
            errc <- sendMsg(msg, addr)
            fmt.Println("done")
        }(addr)
    }

    for _ = range addrs {
        if err := <-errc; err != nil {
            return err
        }
    }
    return nil
}
```

Run

- what if we can't predict the capacity of the channel?

Avoid goroutines leaks with quit chan

```
func broadcastMsg(msg string, addrs []string) error {
    errc := make(chan error)
    quit := make(chan struct{})

    defer close(quit)

    for _, addr := range addrs {
        go func(addr string) {
            select {
            case errc <- sendMsg(msg, addr):
                fmt.Println("done")
            case <-quit:
                fmt.Println("quit")
            }
        }(addr)
    }

    for _ = range addrs {
        if err := <-errc; err != nil {
            return err
        }
    }
    return nil
}
```

Run

Twelve best practices

1. Avoid nesting by handling errors first
2. Avoid repetition when possible
3. Important code goes first
4. Document your code
5. Shorter is better
6. Packages with multiple files
7. Make your packages "go get"-able
8. Ask for what you need
9. Keep independent packages independent
10. Avoid concurrency in your API
11. Use goroutines to manage state
12. Avoid goroutine leaks

Some links

Resources

- Go homepage golang.org
- Go interactive tour tour.golang.org

Other talks

- Lexical scanning with Go video
- Concurrency is not parallelism video
- Go concurrency patterns video
- Advanced Go concurrency patterns video

Thank you

Francesc Campoy Flores
Gopher at Google
@campoy83
http://campoy.cat/+

http://golang.org