Tips for Creating a Good Package

GopherConIndia 2015
20 February 2015

Keiji Yoshida
Software engineer, Recruit Communications Co., Ltd.

Self Introduction

About me

- Keiji Yoshida
- Software engineer at Recruit Communications Co., Ltd. (http://www.rco.recruit.co.jp/) in Tokyo
- Using Go only for my private work

Experience with Go

- Started using Go at the beginning of 2014
- Became a contributor (https://github.com/martini-contrib) to Martini (http://martini.codegangsta.io/), which is a WAF for Go
- Gave a presentation about Martini (http://slides.yoss.si/gocon/martini.html#/) at a Go Conference (http://connpass.com/event/6370/) in Tokyo in May 2014
- Created some packages in Go

Ace - HTML template engine for Go (https://github.com/yosssi/ace)

GCSS - Pure Go CSS Preprocessor (https://github.com/yosssi/gcss)

GMQ - Pure Go MQTT Client (https://github.com/yosssi/gmq)

Goal

Share my knowledge of creating a successful Go package

I have learned some good tips for creating a Go package through trial and error.

This presentation is for beginners of Go who want to create their own Go package.

Following the standard Go coding style

Standard Go coding style

We should follow the standard Go coding style as written in the articles below:

- Effective Go - The Go Programming Language (http://golang.org/doc/effective_go.html)
- CodeReviewComments - go-wiki (https://code.google.com/p/go-wiki/wiki/CodeReviewComments)

There are a lot of rules we have to follow such as:

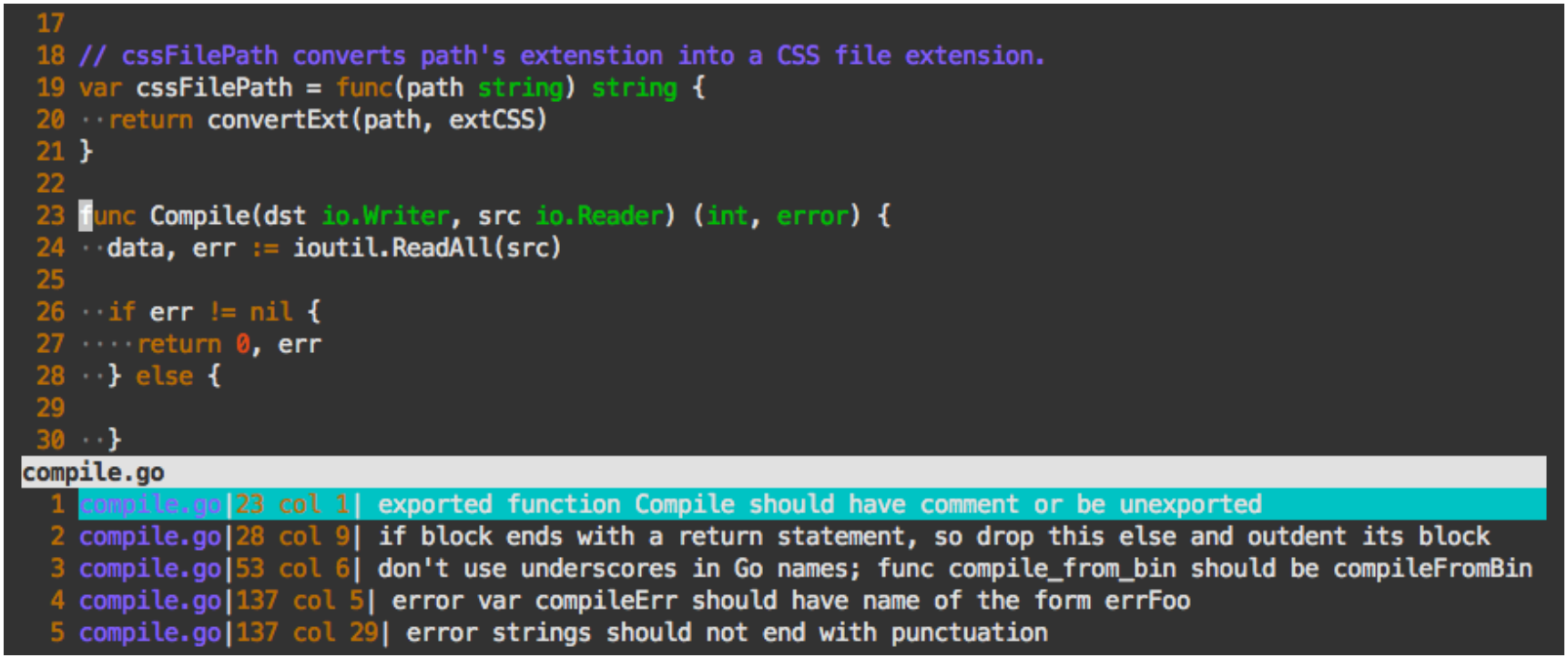
- Every exported (capitalized) name in a program should have a doc comment
- The convention in Go is to use *MixedCaps* or *mixedCaps* rather than underscores to write multiword names
- Error strings should not be capitalized (unless beginning with proper nouns or acronyms) or end with punctuation, since they are usually printed following other context

It is difficult to memorize all coding rules and follow them during coding.

Using Golint

Golint (https://github.com/golang/lint) is a linter for Go source code.

This tool checks Go source code to see if the source code is following the standard Go coding style or not.



Using goimports instead of gofmt

goimports (http://godoc.org/code.google.com/p/go.tools/cmd/goimports) acts the same as gofmt but in addition to code formatting, it also updates Go import lines, adds missing ones and removes unreferenced ones.

goimports also organizes import lines in groups with blank lines between them. The standard library packages are in the first group.

```
package main

import (
    "fmt"
    "hash/adler32"
    "os"

    "appengine/user"
    "appengine/foo"

    "code.google.com/p/x/y"
    "github.com/foo/bar"
)
```

Making our Go packages extensible

Exporting identifiers only when needed

We should not export all identifiers because exported identifiers are difficult to modify

```
// PrivateStruct is used only in this package.
type PrivateStruct struct{}

// PublicFunc is used not only in this package but also other packages.
func PublicFunc() {}
```

Only export identifiers when necessary.

```
// privateStruct is used only in this package.
type privateStruct struct{}

// PublicFunc is used not only in this package but also other packages.
func PublicFunc() {}
```

Using Options struct as a parameter 1/5

Suppose we define a Dog struct and its construct function as following:

```
type Dog struct {
    name string
}

func NewDog(name string) *Dog {
    d := &Dog{name: name}
    return d // Some initialization might be done before returning `d`.
}

func main() {
    d := NewDog("Taro")
}
```

Using Options struct as a parameter 2/5

If we need to add an age field to the Dog struct, we have to create a new function:

```
type Dog struct {
    name string
    age  int
}

func (d *Dog) SetAge(age int) *Dog {
    d.age = age
    return d
}

func NewDog(name string) *Dog {
    d := &Dog{name: name}
    return d // Some initialization might be done before returning `d`.
}

func main() {
    d := NewDog("Taro").SetAge(7)
}
```

We cannot change the NewDog function's signature because it has been exported.

Using Options struct as a parameter 3/5

After repeating this kind of modification, disaster strikes:

```
type Dog struct {
    name    string
    age     int
    sex     string
    species string
}

func (d *Dog) SetSex(sex string) *Dog {
    d.sex = sex
    return d
}

func (d *Dog) SetSpecies(species string) *Dog {
    d.species = species
    return d
}

func main() {
    d := NewDog("Taro").SetAge(7).SetSex("male").SetSpecies("Dachshund")
}
```

Using Options struct as a parameter 4/5

We can avoid adding a new function when a new field is added by using Options struct.

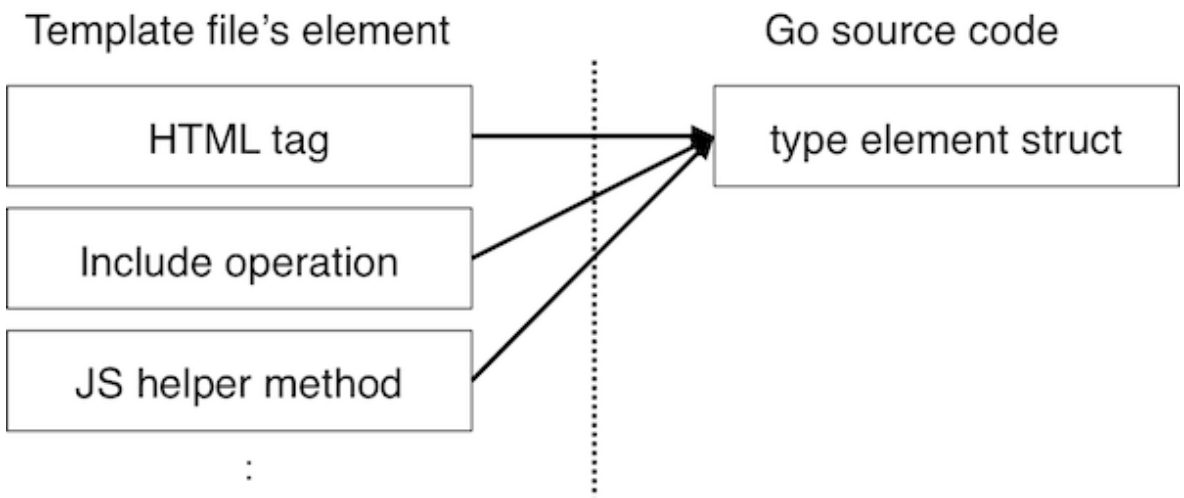
```
type Dog struct {
    name string
}

type Options struct {
    Name string
}

func NewDog(opts *Options) *Dog {
    if opts == nil {
        opts = &Options{}
    }
    d := &Dog{name: opts.Name}
    return d // Some initialization might be done before returning `d`.
}

func main() {
    d := NewDog(&Options{Name: "Taro"})
}
```

Using Options struct as a parameter 5/5

```
type Dog struct {
    name string
    age  int
}

type Options struct {
    Name string
    Age  int
}

func NewDog(opts *Options) *Dog {
    if opts == nil {
        opts = &Options{}
    }
    d := &Dog{name: opts.Name, age: opts.Age}
    return d // Some initialization might be done before returning `d`.
}

func main() {
    d := NewDog(&Options{Name: "Taro", Age: 7})
}
```

Dividing a larger struct into smaller ones 1/7

When I started creating an HTML template engine, I created a single struct representing all of the template file's elements.



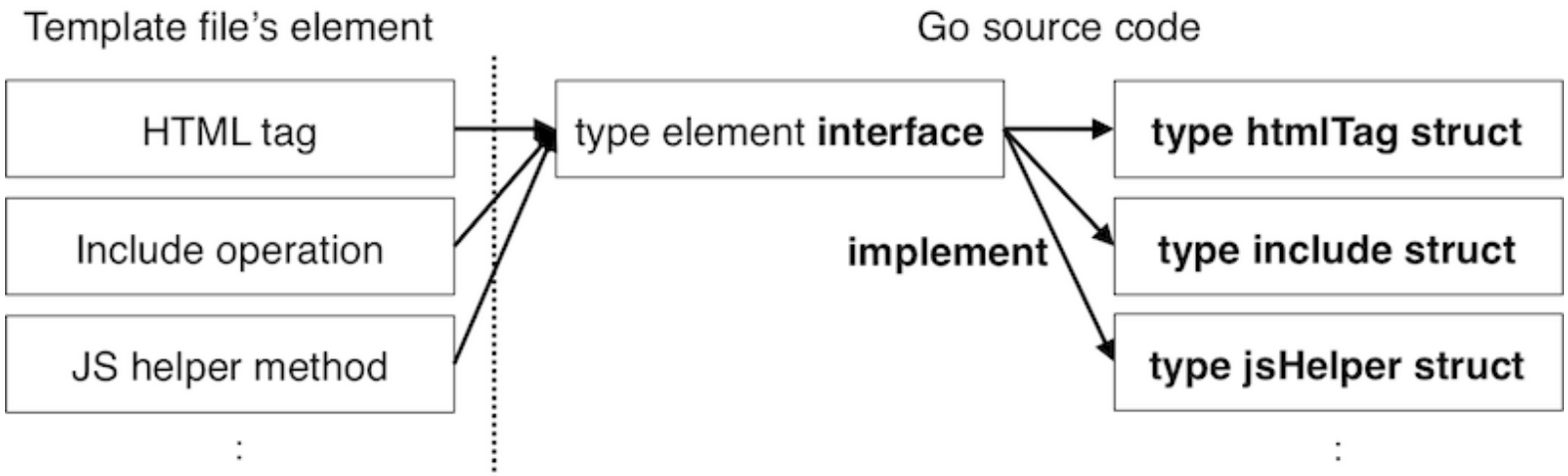
Dividing a larger struct into smaller ones 2/7

The `element` struct became very large because it contained all element types' processing and lost its maintainability.

```
// element represents a template file's element.
type element struct {
    eType string // eType represents a type of the element.
}

// WriteTo writes the element's content to the writer.
func (e *element) WriteTo(w io.Writer) (int64, error) {
    switch e.eType {
    case "HTMLTag":
        // Write its HTML content.
    case "IncludeOperation":
        // Load the other template and write the content.
    case "JSHelper":
        // Write its JavaScript content.
    }
}
```

Dividing a larger struct into smaller ones 3/7

I divided the `element` struct into smaller ones and created an `element` interface.



Dividing a larger struct into smaller ones 4/7

I could divide the single large processing into smaller processing that made my package more maintainable.

```
type element interface {
    io.WriterTo
}

type htmlTag struct{}

func (h *htmlTag) WriteTo(w io.Writer) (int64, error) {
    // Write the html tag's content to the writer.
}

type include struct{}

func (i *include) WriteTo(w io.Writer) (int64, error) {
    // Load the other template and write the content.
}
```

Dividing a larger struct into smaller ones 5/7

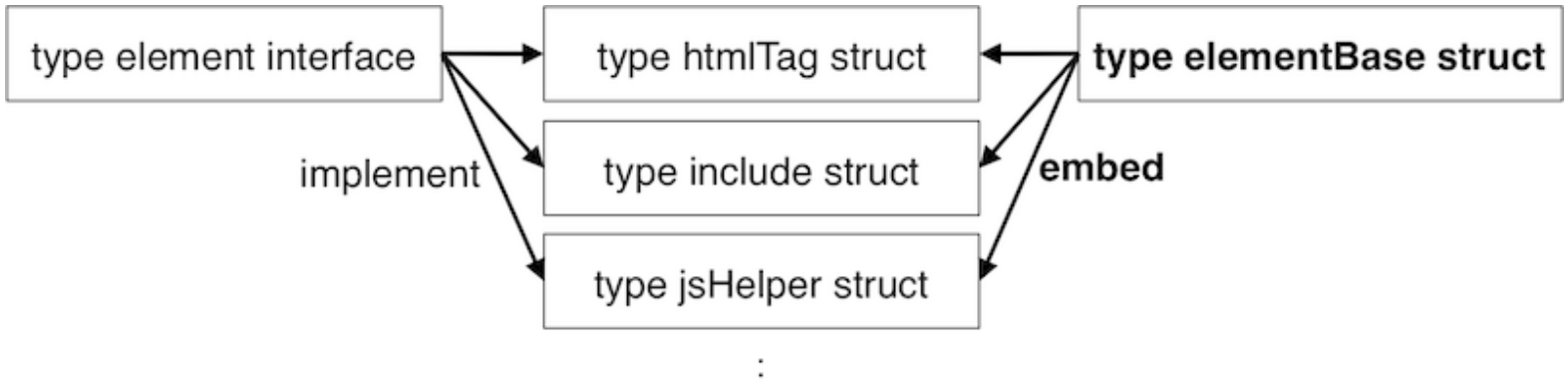
In addition, each element had its child elements just like each HTML tag did.

```
type element interface {
    io.WriterTo
    Append(child element)
}

type htmlTag struct {
    children []element // children is a common field among elements.
}

func (h *htmlTag) Append(child element) { // Append is a common method among elements.
    h.children = append(h.children, child)
}

type include struct {
    children []element
}

func (i *include) Append(child element) {
    i.children = append(i.children, child)
}
```

Dividing a larger struct into smaller ones 6/7

I created an `elementBase` struct which had fields and methods common among elements.

I embedded the `elementBase` struct into the structs which implemented the `element` interface.



Dividing a larger struct into smaller ones 7/7

By having embedded the `elementBase` struct into the structs which implemented the `element` interface, I could get rid of the duplicated fields and methods.

```
// elementBase has fields and methods which are common among elements.
type elementBase struct {
    children []element
}

func (e *elementBase) Append(child element) {
    e.children = append(e.children, child)
}

type htmlTag struct {
    elementBase // embed the elementBase struct into the htmlTag struct.
}

type include struct {
    elementBase // embed the `elementBase` struct into the include struct.
}
```

Giving users the freedom of using our package

Giving users the option to call APIs sequentially or concurrently 1/3

When I released my CSS preprocessing package like Sass (http://sass-lang.com/) which compiles a simplified CSS format file into a pure CSS format file, I made the `Compile` function always run concurrently because it involved many I/O blocking processes.

```
func Compile(path string) <-chan string {
    chPath := make(chan string)

    go func() {
        // Read the simplified CSS format file specified by the paramter,
        // compile it into a pure CSS format data and
        // write the result to a new file.

        // Send the result file's path to the chPath channel.
        chPath <- resultFilePath
    }()

    return chPath
}
```

But I noticed we could not use this function sequentially if we wanted to do so.

Giving users the option to call APIs sequentially or concurrently 2/3

I changed the function to always run sequentially.

```
func Compile(path string) string {
    // Read the simplified CSS format file specified by the paramter,
    // compile it into a pure CSS format data and
    // write the result to a new file.

    // Return the result file's path.
    return resultFilePath
}
```

Giving users the option to call APIs sequentially or concurrently 3/3

Users can choose to call the function sequentially or concurrently.

```
func main() {
    // Call the `Compile` function sequentially.
    resultPath1 := Compile("file1")
    resultPath2 := Compile("file2")

    // Calling it concurrently is also easy.
    originalFilePaths := []string{"file3", "file4", "file5"}
    chResultPath := make(chan string)

    for _, path := range originalFilePaths {
        go func(path string) {
            chResultPath <- Compile(path)
        }(path)
    }

    for resultPath := range chResultPath {
        fmt.Println(resultPath)
    }
}
```

Using an interface as a function parameter 1/3

The precedent `Compile` function gets a file path as a parameter and loads the file specified by the parameter.

```
func Compile(path string) string {
    // Read the simplified CSS format file specified by the paramter,
    // compile it into a pure CSS format data and
    // write the result to a new file.

    // Return the result file's path.
    return resultFilePath
}
```

We cannot load binary data from memory instead of a file by using this function because all it can get is a file path.

Using an interface as a function parameter 2/3

I changed the function parameter from the file path string to an `io.Reader` interface so that users could pass to the function any data they wanted to load and compile.

```
func Compile(src io.Reader) string {
    // Read the simplified CSS format data from the reader,
    // compile it into a pure CSS format data and
    // write the result to a new file.

    // Return the result file's path.
    return resultFilePath
}
```

The `Compile` function writes the result data only to a file on disk. Users cannot write the result data to other locations.

Using an interface as a function parameter 3/3

I added an `io.Writer` interface parameter to the function so that users could write the result data to anywhere they liked.

```
func Compile(dst io.Writer, src io.Reader) (int, error) {
    // Read the simplified CSS format data from the reader,
    // compile it into a pure CSS format data.

    // Write the result data to the writer.
    return dst.Write(resultData)
}
```

We can make our functions much more flexible and useful to users by using an interface as a function parameter.

Testing

We should aim to accomplish 100% of the test coverage

- To detect the regression when we modify our package
- To keep up with the updated versions of Go language and the packages which our package depends on
- To detect the race condition in our package

Replacing a global variable during testing 1/2

Replacing a global variable's value with another value is helpful in accomplishing 100% of the test coverage.

For example, we cannot test the block which has `os.Exit` because it terminates the testing process.

```
func main() {
    if len(os.Args) < 2 {
        os.Exit(1) // Can not test this block because os.Exit terminates the process.
    }
}
```

Replacement of a global variable during testing 2/2

By defining a global variable which has value of os.Exit and replacing it with a function which does not terminate the process, we can test the block in which `os.Exit` was originally called.

```
var exit = os.Exit // Define a global variable.

func main() {
    if len(os.Args) < 2 {
        exit(1) // Can test this block by replacing the value of the exit function.
    }
}
```

```
func Test_main(t *testing.T) {
    defer func(orig func(int)) {
        exit = orig // Restore the value of the exit global variable.
    }(exit)

    exit = func(_ int) {} // Replace the value of the exit function with another one.

    main() // Execute the test.
}
```

Table driven tests

Table driven tests (https://code.google.com/p/go-wiki/wiki/TableDrivenTests) are useful for writing many test cases in simple order.

```
var testCases = []struct {
    in  string
    out string
}{
    {"a", "A"},
    {"b", "B"},
    {"c", "C"},
    {"d", "D"},
    {"e", "E"},
}

func TestToUpper(t *testing.T) {
    for _, tc := range testCases {
        if result := ToUpper(tc.in); result != tc.out {
            t.Errorf("ToUpper(%q) => %q, want %q", tc.in, result, tc.out)
        }
    }
}
```

Testing our Go code automatically on CI services

Windows

- AppVeyor (https://ci.appveyor.com)

Linux

- wercker (http://wercker.com/)
- drone.io (https://drone.io/)
- Travis CI (https://travis-ci.org/)
- CircleCI (https://circleci.com/)

Test Coverage Report and Statistics

- Coveralls (https://coveralls.io/)
- codecov (https://codecov.io/)

Thank you

Keiji Yoshida
Software engineer, Recruit Communications Co., Ltd.
yoshida_keiji@r.recruit.co.jp (mailto:yoshida_keiji@r.recruit.co.jp)
http://yoss.si/ (http://yoss.si/)
@_yosssi (http://twitter.com/_yosssi)