GOPHER CHINA 2020

中国 上海 / 2020-11.21-22

Go Programming Patterns

陈皓（左耳朵）

2020.11.

The Creators of Golang

Robert Griesemer	Rob Pike	Ken Thompson
Born 1964	Born 1956	Born 1943
Java HotSpot JVM	Unix	Unix/C
V8 Javascript engine	UTF-8	UTF-8
Golang	Golang	Golang

MegaEase
Enterprise Cloud Native Architecture Provider

Self Introduction

- 20+ years working experience for large-scale distributed system architecture and development. Familiar with Cloud Native computing and high concurrency / high availability architecture solution.

- Working Experiences
  - MegaEase – Cloud Native Architecture products as Founder
  - Alibaba – AliCloud, Tmall as principle software engineer.
  - Amazon – Amazon.com as senior software developer.
  - Thomson Reuters as principle software engineer.

Weibo: @左耳朵耗子
Twitter: @haoel
Blog: http://coolshell.cn/

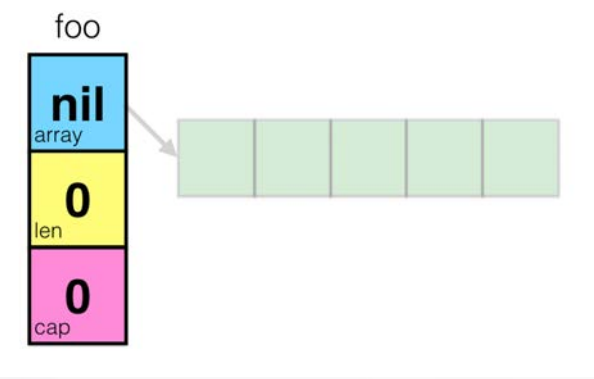
Topics

Basic Coding
Error Handling
Delegation / Embed
Functional Option
Map/Reduce/Filter
Go Generation
Decoration
Kubernetes Visitor
Pipeline

Basic Tips

Slice Internals

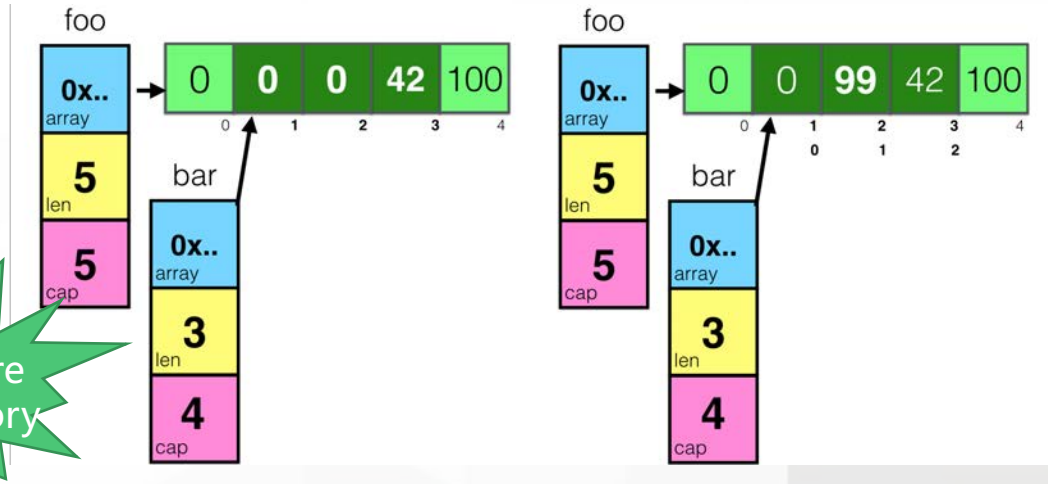
```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```



Slice is a struct

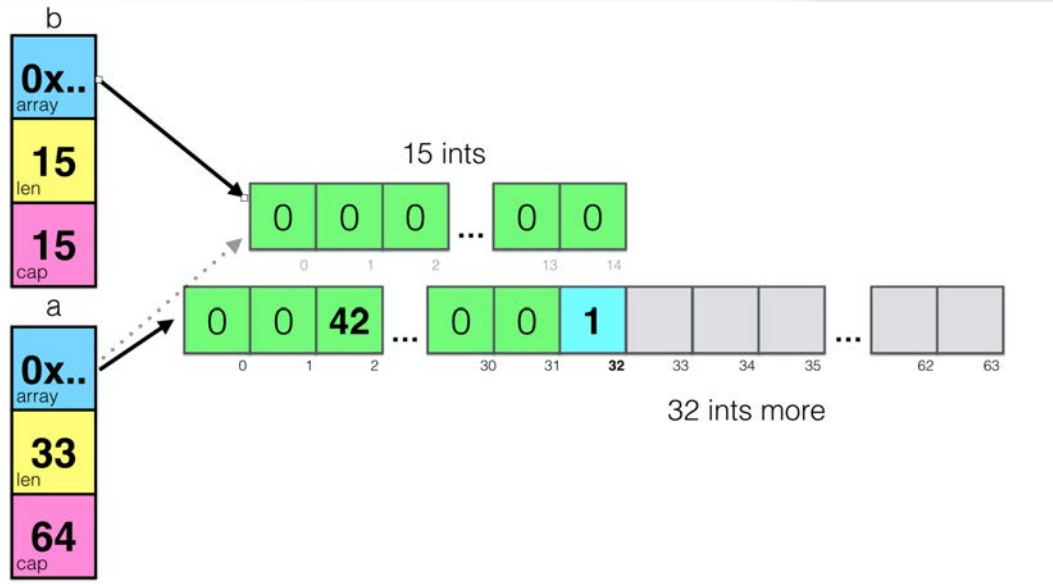
```
foo = make([]int, 5)
foo[3] = 42
foo[4] = 100

bar  := foo[1:4]
bar[1] = 99
```

Slice share the memory



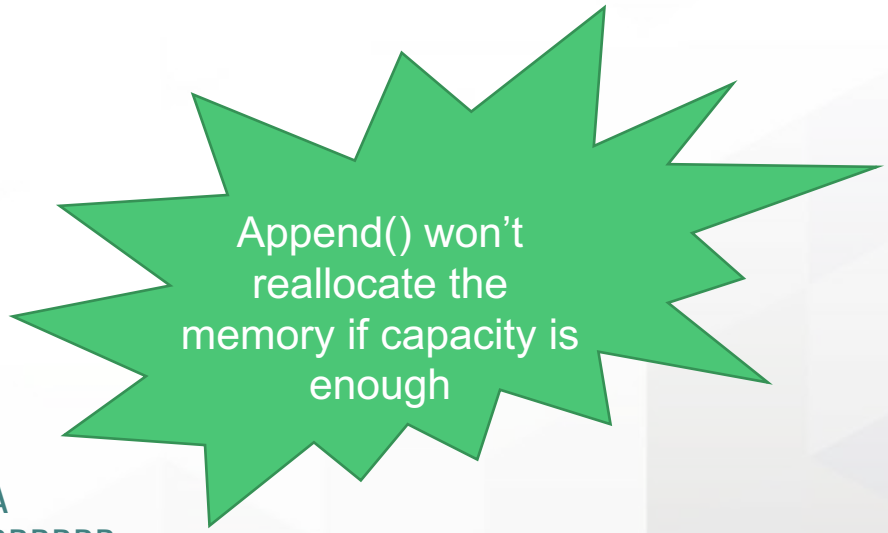

```
a := make([]int, 32)
b := a[1:16]
a = append(a, 1)
a[2] = 42
```

Append() could reallocate the memory



Slices Overlapped

```
func main() {
    path := []byte("AAAA/BBBBBBBBB")
    sepIndex := bytes.IndexByte(path,'/')

    dir1 := path[:sepIndex]
    dir2 := path[sepIndex+1:]

    fmt.Println("dir1 =>",string(dir1)) //prints: dir1 => AAAA
    fmt.Println("dir2 =>",string(dir2)) //prints: dir2 => BBBBBBBBB


    dir1 = append(dir1,"suffix"...)

    fmt.Println("dir1 =>",string(dir1)) //prints: dir1 => AAAAsuffix
    fmt.Println("dir2 =>",string(dir2)) //prints: dir2 => uffixBBBB
}
```



Slices Overlapped

```
func main() {
    path := []byte("AAAA/BBBBBBBBB")
    sepIndex := bytes.IndexByte(path,'/')

    dir1 := path[:sepIndex:sepIndex]
    dir2 := path[sepIndex+1:]

    fmt.Println("dir1 =>",string(dir1)) //prints: dir1 => AAAA
    fmt.Println("dir2 =>",string(dir2)) //prints: dir2 => BBBBBBBBB


    dir1 = append(dir1,"suffix"...)

    fmt.Println("dir1 =>",string(dir1)) //prints: dir1 => AAAAsuffix
    fmt.Println("dir2 =>",string(dir2)) //prints: dir2 => BBBBBBBBB
}
```

Full Slice Expression

dir1 := path[:sepIndex:sepIndex]

Limited Capacity

The appending cause
a new buffer allocation

Deep Comparison

```
import (
    "fmt"
    "reflect"
)


type data struct {
    num int                     //ok
    checks [10]func() bool      //not comparable
    doit func() bool            //not comparable
    m map[string] string        //not comparable
    bytes []byte                //not comparable
}
```

```
func main() {

    v1 := data{}
    v2 := data{}
    fmt.Println("v1 == v2:",reflect.DeepEqual(v1,v2))
    //prints: v1 == v2: true


    m1 := map[string]string{"one": "a","two": "b"}
    m2 := map[string]string{"two": "b", "one": "a"}
    fmt.Println("m1 == m2:",reflect.DeepEqual(m1, m2))
    //prints: m1 == m2: true


    s1 := []int{1, 2, 3}
    s2 := []int{1, 2, 3}
    fmt.Println("s1 == s2:",reflect.DeepEqual(s1, s2))
    //prints: s1 == s2: true
}
```

Function vs Receiver

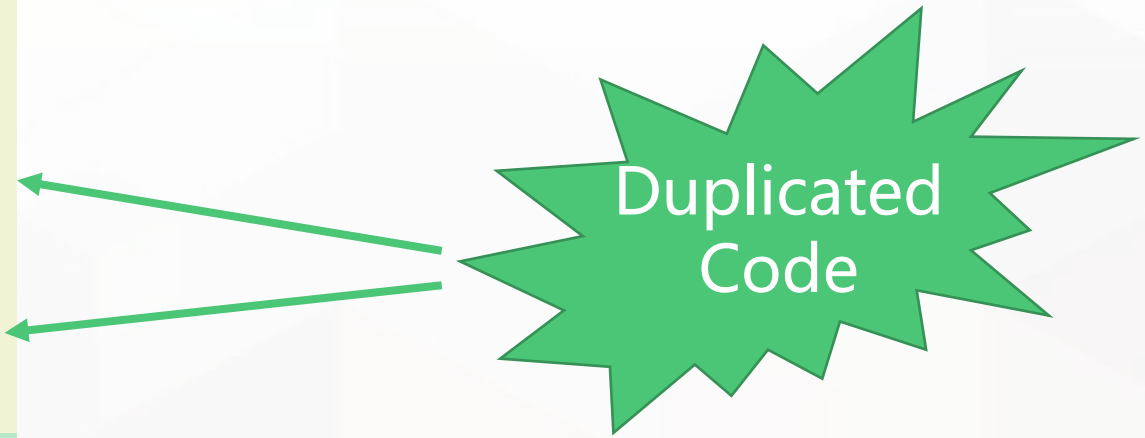
```go
func PrintPerson(p *Person) {
    fmt.Printf("Name=%s, Sexual=%s, Age=%d\n",
        p.Name, p.Sexual, p.Age)
}


func main() {
    var p = Person{
        Name: "Hao Chen",
        Sexual: "Male",
        Age: 44,
    }


    PrintPerson(&p)
}
```

```go
func (p *Person) Print() {
    fmt.Printf("Name=%s, Sexual=%s, Age=%d\n",
        p.Name, p.Sexual, p.Age)
}


func main() {
    var p = Person{
        Name: "Hao Chen",
        Sexual: "Male",
        Age: 44,
    }


    p.Print()
}
```

Interface Patterns

```go
type Country struct {
    Name string
}

type City struct {
    Name string
}



type Printable interface {
    PrintStr()
}

func (c Country) PrintStr() {
    fmt.Println(c.Name)
}

func (c City) PrintStr() {
    fmt.Println(c.Name)
}
```

```go
c1 := Country {"China"}
c2 := City {"Beijing"}

c1.PrintStr()
c2.PrintStr()
```



Interface Patterns

```go
type Country struct {
    Name string
}

type City struct {
    Name string
}



type Printable interface {
    PrintStr()
}

func (c Country) PrintStr() {
    fmt.Println(c.Name)
}

func (c City) PrintStr() {
    fmt.Println(c.Name)
}
```

```go
c1 := Country {"China"}
c2 := City {"Beijing"}

c1.PrintStr()
c2.PrintStr()
```

```go
type WithName struct {
    Name string
}

type Country struct {
    WithName
}

type City struct {
    WithName
}



type Printable interface {
    PrintStr()
}

func (w WithName) PrintStr() {
    fmt.Println(w.Name)
}
```

```go
c1 := Country {WithName{ "China"}}
c2 := City { WithName{"Beijing"}}

c1.PrintStr()
c2.PrintStr()
```

Using a shared struct

Initialization is a bit mess

Interface Patterns

```go
type Country struct {
    Name string
}

type City struct {
    Name string
}



type Printable interface {
    PrintStr()
}

func (c Country) PrintStr() {
    fmt.Println(c.Name)
}

func (c City) PrintStr() {
    fmt.Println(c.Name)
}
```

```go
c1 := Country {"China"}
c2 := City {"Beijing"}

c1.PrintStr()
c2.PrintStr()
```

```go
type WithName struct {
    Name string
}

type Country struct {
    WithName
}

type City struct {
    WithName
}



type Printable interface {
    PrintStr()
}

func (w WithName) PrintStr() {
    fmt.Println(w.Name)
}
```

```go
c1 := Country {WithName{ "China"}}
c2 := City { WithName{"Beijing"}}

c1.PrintStr()
c2.PrintStr()
```

```go
type Country struct {
    Name string
}

type City struct {
    Name string
}

type Stringable interface {
    ToString() string
}

func (c Country) ToString() string {
    return "Country = " + c.Name
}

func (c City) ToString() string{
    return "City = " + c.Name
}

func PrintStr(p Stringable) {
    fmt.Println(p.ToString())
}
```

```go
d1 := Country {"USA"}
d2 := City{"Los Angeles"}

PrintStr(d1)
PrintStr(d2)
```

io.Read
ioutil.ReadAll

Golden Rule

Program to an interface not an implementation

Verify Interface Compliance

```
type Shape interface {
    Sides() int
    Area() int
}

type Square struct {
    len int
}

func (s* Square) Sides() int {
    return 4
}


func main() {
    s := Square{len: 5}
    fmt.Printf("%d\n",s.Sides())
}
```

Checking a type whether implement all of methods

```
var _ Shape = (*Square)(nil)
```

cannot use (*Square)(nil) (type *Square) as type Shape in assignment:
*Square does not implement Shape (missing Area method)

```
func  (s* Square) Area() int {
    return s.len * s.len
}
```

Time

Time is difficult!

Always use `time.Time` and `time.Duration`

- Command-line flags: flag supports `time.Duration` via `time.ParseDuration`
- JSON: encoding/json supports encoding `time.Time` as an RFC 3339 string via its UnmarshalJSON method
- SQL: database/sql supports converting DATETIME or TIMESTAMP columns into `time.Time` and back if the underlying driver supports it
- YAML: gopkg.in/yaml.v2 supports time.Time as an RFC 3339 string, and `time.Duration` via `time.ParseDuration`.

If you cannot use time.Time, use string with format defined in RFC3339

Performance

Prefer strconv over fmt

```
for i := 0; i < b.N; i++ {
    s := fmt.Sprint(rand.Int())
}
```

143 ns/op

```
for i := 0; i < b.N; i++ {
    s := strconv.Itoa(rand.Int())
}
```

64.2 ns/op

Avoid string-to-byte conversion

```
for i := 0; i < b.N; i++ {
    w.Write([]byte("Hello world"))
}
```

22.2 ns/op

```
data := []byte("Hello world")
for i := 0; i < b.N; i++ {
    w.Write(data)
}
```

3.25 ns/op

Specifying Slice Capacity

```
for n := 0; n < b.N; n++ {
    data := make([]int, 0)
    for k := 0; k < size; k++{
        data = append(data, k)
    }
}
```

100000000 2.48s

```
for n := 0; n < b.N; n++ {
    data := make([]int, 0, size)
    for k := 0; k < size; k++{
        data = append(data, k)
    }
}
```

100000000 0.21s

Use StringBuffer or StringBuilder

```
var strLen int = 30000

var str string
for n := 0; n < strLen; n++ {
    str += "x"
}
```

12.7 ns/op

```
var builder strings.Builder
for n := 0; n < strLen; n++ {
    builder.WriteString("x")
}
```

0.0265 ns/op

```
var buffer bytes.Buffer
for n := 0; n < strLen; n++ {
    buffer.WriteString("x")
}
```

0.0088 ns/op

Performance

Make multiple I/O operations asynchronous
running in parallel, Use sync.WaitGroup to synchronize multiple operations.

Avoid memory allocation in hot code
Not only requires additional CPU cycles, but will also make the garbage collector busy. Using use sync.Pool to reuse objects whenever possible, especially in program hot spots.

Favor lock-free algorithms
Avoiding mutexes whenever possible. Lock-free alternatives to some common data structures are available , using sync/Atomic package

Use buffered I/O
Accessing disk for every byte is inefficient; reading and writing bigger chunks of data greatly improves the speed. Using bufio.NewWriter() or bufio.NewReader() could help

Use compiled regular expressions for repeated matching
It is inefficient to compile the same regular expression before every matching.

Use Protocol Buffers instead of JSON
JSON uses reflection, which is relatively slow due to the amount of work it does. Using protobuf or msgp.

Use int keys instead of string keys for maps
If the program relies heavily on maps, using int keys might be meaningful

Further Readings

Effective Go
https://golang.org/doc/effective_go.html

Uber Go Style
https://github.com/uber-go/guide/blob/master/style.md

50 Shades of Go: Traps, Gotchas, and Common Mistakes for New Golang Devs
http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/

Go Advice
https://github.com/cristaloleg/go-advice

Practical Go Benchmarks
https://www.instana.com/blog/practical-golang-benchmarks/

Benchmarks of Go serialization methods
https://github.com/alecthomas/go_serialization_benchmarks

Debugging performance issues in Go programs
https://github.com/golang/go/wiki/Performance

Go code refactoring: the 23x performance hunt
https://medium.com/@val_deleplace/go-code-refactoring-the-23x-performance-hunt-156746b522f7

Delegation / Embed

Delegation – Example 1

Start with Widget and Label

```
type Widget struct {
    X, Y int
}


type Label struct {
    Widget        // Embedding (delegation)
    Text   string // Aggregation
}
```

If the member name conflict, need solve it

```
func (label Label) Paint() {
      fmt.Printf("%p:Label.Paint(%q)\n", &label, label.Text)
}
```

Two Interfaces

```
type Painter interface {
    Paint()
}

type Clicker interface {
    Click()
}
```

Delegation – Example 1

New Component - Button

```
type Button struct {
    Label // Embedding (delegation)
}

func NewButton(x, y int, text string) Button {
    return Button{Label{Widget{x, y}, text}}
}

func (button Button) Paint() { // Override
    fmt.Printf("Button.Paint(%s)\n", button.Text)
}

func (button Button) Click() {
    fmt.Printf("Button.Click(%s)\n", button.Text)
}
```

New Component – ListBox

```
type ListBox struct {
    Widget          // Embedding (delegation)
    Texts  []string // Aggregation
    Index  int      // Aggregation
}

func (listBox ListBox) Paint() {
    fmt.Printf("ListBox.Paint(%q)\n", listBox.Texts)
}

func (listBox ListBox) Click() {
    fmt.Printf("ListBox.Click(%q)\n", listBox.Texts)
}
```

Delegation – Example 1 -Polymorphism

```
button1 := Button{Label{Widget{10, 70}, "OK"}}
button2 := NewButton(50, 70, "Cancel")
listBox := ListBox{Widget{10, 40},
    []string{"AL", "AK", "AZ", "AR"}, 0}

for _, painter := range []Painter{label, listBox, button1, button2} {
    painter.Paint()
}
```

```
Label.Paint("State")
ListBox.Paint(["AL" "AK" "AZ" "AR"])
Button.Paint("OK")
Button.Paint("Cancel")
```

```
for _, widget := range []interface{}{label, listBox, button1, button2} {
    if clicker, ok := widget.(Clicker); ok {
        clicker.Click()
    }
}
```

```
ListBox.Click(["AL" "AK" "AZ" "AR"])
Button.Click("OK")
Button.Click("Cancel")
```

Delegation – Example 2

An Integer Set with Add(), Delete(), Contains(), String() method

```
type IntSet struct {
    data map[int]bool
}
func NewIntSet() IntSet {
    return IntSet{make(map[int]bool)}
}
func (set *IntSet) Add(x int) {
    set.data[x] = true
}
func (set *IntSet) Delete(x int) {
    delete(set.data, x)
}
func (set *IntSet) Contains(x int) bool {
    return set.data[x]
}
```

```
// Satisfies fmt.Stringer interface
func (set *IntSet) String() string {
    if len(set.data) == 0 {
        return "{}"
    }
    ints := make([]int, 0, len(set.data))
    for i := range set.data {
        ints = append(ints, i)
    }
    sort.Ints(ints)
    parts := make([]string, 0, len(ints))
    for _, i := range ints {
        parts = append(parts, fmt.Sprint(i))
    }
    return "{" + strings.Join(parts, ",") + "}"
}
```

```
ints := NewIntSet()
for _, i := range []int{1, 3, 5, 7} {
    ints.Add(i)
    fmt.Println(ints)
}
for _, i := range []int{1, 2, 3, 4, 5, 6, 7} {
    fmt.Print(i, ints.Contains(i), " ")
    ints.Delete(i)
    fmt.Println(ints)
}
```

Delegation – Example 2

Adding the Undo Feature for IntSet

```
type UndoableIntSet struct { // Poor style
    IntSet    // Embedding (delegation)
    functions []func()
}

func NewUndoableIntSet() UndoableIntSet {
    return UndoableIntSet{NewIntSet(), nil}
}


func (set *UndoableIntSet) Add(x int) { // Override
    if !set.Contains(x) {
        set.data[x] = true
        set.functions = append(set.functions, func() { set.Delete(x) })
    } else {
        set.functions = append(set.functions, nil)
    }
}


func (set *UndoableIntSet) Delete(x int) { // Override
    if set.Contains(x) {
        delete(set.data, x)
        set.functions = append(set.functions, func() { set.Add(x) })
    } else {
        set.functions = append(set.functions, nil)
    }
}
```

UndoableIntSet works, however it is not generic

```
func (set *UndoableIntSet) Undo() error {
    if len(set.functions) == 0 {
        return errors.New("No functions to undo")
    }
    index := len(set.functions) - 1
    if function := set.functions[index]; function != nil {
        function()
        // Free closure for garbage collection
        set.functions[index] = nil
    }
    set.functions = set.functions[:index]
    return nil
}
```

```
ints := NewUndoableIntSet()
for _, i := range []int{1, 3, 5, 7} {
    ints.Add(i)
    fmt.Println(ints)
}
for _, i := range []int{1, 2, 3, 4, 5, 6, 7} {
    fmt.Println(i, ints.Contains(i), " ")
    ints.Delete(i)
    fmt.Println(ints)
}

for {
    if err := ints.Undo(); err != nil {
        break
    }
    fmt.Println(ints)
}
```

Delegation – Example2 – IoC

Implement function stack

```
type Undo []func()

func (undo *Undo) Add(function func()) {
    *undo = append(*undo, function)
}

func (undo *Undo) Undo() error {
    functions := *undo
    if len(functions) == 0 {
        return errors.New("No functions to undo")
    }
    index := len(functions) - 1
    if function := functions[index]; function != nil {
        function()
        // Free closure for garbage collection
        functions[index] = nil
    }
    *undo = functions[:index]
    return nil
}
```

Depend on function signature(interface)

New Version of Integer Set

```
type IntSet struct {
    data map[int]bool
    undo Undo
}

func NewIntSet() IntSet {
    return IntSet{data: make(map[int]bool)}
}

func (set *IntSet) Undo() error {
    return set.undo.Undo()
}

func (set *IntSet) Contains(x int) bool {
    return set.data[x]
}
```

```
func (set *IntSet) Add(x int) {
    if !set.Contains(x) {
        set.data[x] = true
        set.undo.Add(func() { set.Delete(x) })
    } else {
        set.undo.Add(nil)
    }
}

func (set *IntSet) Delete(x int) {
    if set.Contains(x) {
        delete(set.data, x)
        set.undo.Add(func() { set.Add(x) })
    } else {
        set.undo.Add(nil)
    }
}
```

In Add() function, add the Delete() function into Undo stack
In Delete() function, add the Add() function into Undo stack
In Undo() function, simply run Undo() and it would pop up the function to execute

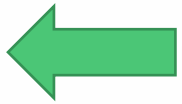
Error Handling

if err != nil Checking Hell

```
func parse(r io.Reader) (*Point, error) {

    var p Point

    if err := binary.Read(r, binary.BigEndian, &p.Longitude); err != nil {
        return nil, err
    }


    if err := binary.Read(r, binary.BigEndian, &p.Latitude); err != nil {
        return nil, err
    }


    if err := binary.Read(r, binary.BigEndian, &p.Distance); err != nil {
        return nil, err
    }


    if err := binary.Read(r, binary.BigEndian, &p.ElevationGain); err != nil {
        return nil, err
    }

    if err := binary.Read(r, binary.BigEndian, &p.ElevationLoss); err != nil {
        return nil, err
    }
}
```

Functional Solution

```
func parse(r io.Reader) (*Point, error) {
    var p Point
    var err error
    read := func(data interface{}) {
        if err != nil {
            return
        }
        err = binary.Read(r, binary.BigEndian, data)
    }

    read(&p.Longitude)
    read(&p.Latitude)
    read(&p.Distance)
    read(&p.ElevationGain)
    read(&p.ElevationLoss)


    if err != nil {
        return &p, err
    }
    return &p, nil
}
```



Closure for errors

The only one problem -
it needs an `err` variable and inline function

Learning from bufio.Scanner

```
scanner := bufio.NewScanner(input)

for scanner.Scan() {
    token := scanner.Text()
    // process token
}

if err := scanner.Err(); err != nil {
    // process the error
}
```

Its Scan method performs the underlying I/O, which can of course lead to an error.

Yet the Scan method does not expose an error at all. Instead, it returns a boolean.

And a separate method, to be run at the end of the scan, reports whether an error occurred.

Using an Error Object

```
type Reader struct {
    r   io.Reader
    err error
}


func (r *Reader) read(data interface{}) {
    if r.err == nil {
        r.err = binary.Read(r.r, binary.BigEndian, data)
    }
}
```

The only one problem – We can't make it generic
We have to define a wrapper struct for each type.

Two Articles

Golang Error Handling lesson by Rob Pike
http://jxck.hatenablog.com/entry/golang-error-handling-lesson-by-rob-pike
Errors are values
https://blog.golang.org/errors-are-values

```
func parse(input io.Reader) (*Point, error) {
    var p Point
    r := Reader{r: input}


    r.read(&p.Longitude)
    r.read(&p.Latitude)
    r.read(&p.Distance)
    r.read(&p.ElevationGain)
    r.read(&p.ElevationLoss)


    if r.err != nil {
        return nil, r.err
    }


    return &p, nil
}
```

One More Thing- Error Context

```
if err != nil {
    return err
}
```

Add context to an errors

```
if err != nil {
   return fmt.Errorf("something failed: %v", err)
}
```

A good package help to add the Stack

```
import "github.com/pkg/errors"
if err != nil {
    return errors.Wrap(err, "read failed")
}
```

De Facto Standard

Functional Options

Configuration Problem

A Server Configuration needs so many options, but not all of them are mandatory

```
type Server struct {
    Addr     string
    Port     int
    Protocol string
    Timeout  time.Duration
    MaxConns int
    TLS      *tls.Config
}
```

We need a number of New Server function for different senarios

```
func NewServer(addr string, port int) (*Server, error) {
   //...
}

func NewTLSServer(addr string, port int, tls *tls.Config) (*Server, error) {
    //...
}

func NewServerWithTimeout(addr string, port int, timeout time.Duration) (*Server, error) {
    //...
}

func NewTLSServerWithMaxConnAndTimeout(addr string, port int, maxconns int, timeout time.Duration, tls *tls.Config) (*Server,
error) {
    //...
}
```

One Popular Solution

Using a dedicated Configuration structure

```go
type Config struct {
    Protocol string
    Timeout  time.Duration
    Maxconns int
    TLS      *tls.Config
}

type Server struct {
    Addr string
    Port int
    Conf *Config
}
```

```go
func NewServer(addr string, port int, conf *Config) (*Server, error) {
    //...
}

//Using the default configuratrion
srv1, _ := NewServer("localhost", 9000, nil)

conf := ServerConfig{Protocol:"tcp", Timeout: 60*time.Duration}
srv2, _ := NewServer("locahost", 9000, &conf)
```

The Config parameter is mandatory, and all of the varies configuration share a same function signature
And the Config parameter would be empty or null.

Functional Option

```go
type Option func(*Server)


func Protocol(p string) Option {
    return func(s *Server) {
        s.Protocol = p
    }
}

func Timeout(timeout time.Duration) Option {
    return func(s *Server) {
        s.Timeout = timeout
    }
}

func MaxConns(maxconns int) Option {
    return func(s *Server) {
        s.MaxConns = maxconns
    }
}

func TLS(tls *tls.Config) Option {
    return func(s *Server) {
        s.TLS = tls
    }
}
```

```go
func NewServer(addr string, port int,
               options ...func(*Server)) (*Server, error) {


    srv := Server{
        Addr:     addr,
        Port:     port,
        Protocol: "tcp",
        Timeout:  30 * time.Second,
        MaxConns: 1000,
        TLS:      nil,
    }
    for _, option := range options {
        option(&srv)
    }
    //...
    return &srv, nil
}
```

```go
s1, _ := NewServer("localhost", 1024)
s2, _ := NewServer("localhost", 2048, Protocol("udp"))
s3, _ := NewServer("0.0.0.0", 8080, Timeout(300*time.Second), MaxConns(1000))
```

One Article

“Self referential functions and design” by Rob Pike
http://commandcenter.blogspot.com.au/2014/01/self-referential-functions-and-design.html

Functional Option

- Sensible defaults
- Highly configurable
- Easy to maintain
- Self documenting
- Safe for newcomers
- No need nil or an empty value

Map/Reduce/Filter

Basic Map Reduce – Example 1

Higher Order Function

```go
func MapUpCase(arr []string, fn func(s string) string) []string {
    var newArray = []string{}
    for _, it := range arr {
        newArray = append(newArray, fn(it))
    }
    return newArray
}


func MapLen(arr []string, fn func(s string) int) []int {
    var newArray = []int{}
    for _, it := range arr {
        newArray = append(newArray, fn(it))
    }
    return newArray
}
```

```go
var list = []string{"Hao", "Chen", "MegaEase"}
x := MapUpCase(list, func(s string) string {
    return strings.ToUpper(s)
})
fmt.Printf("%v\n", x)
//["HAO", "CHEN", "MEGAEASE"]

y := MapLen(list, func(s string) int {
    return len(s)
})
fmt.Printf("%v\n", y)
//[3, 4, 8]
```

```go
func Reduce(arr []string, fn func(s string) int) int {
    sum := 0
    for _, it := range arr {
        sum += fn(it)
    }
    return sum
}
```

```go
var list = []string{"Hao", "Chen", "MegaEase"}

x := Reduce(list, func(s string) int {
    return len(s)
})
fmt.Printf("%v\n", x)
// 15
```

```go
func Filter(arr []int, fn func(n int) bool) []int {
    var newArray = []int{}
    for _, it := range arr {
        if fn(it) {
            newArray = append(newArray, it)
        }
    }
    return newArray
}
```

```go
var intset = []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
out := Filter(intset, func(n int) bool {
   return n%2 == 1
})
fmt.Printf("%v\n", out)

out = Filter(intset, func(n int) bool {
    return n > 5
})
fmt.Printf("%v\n", out)
```

Basic Map Reduce – Example 2

```
type Employee struct {
    Name     string
    Age      int
    Vacation int
    Salary   int
}

var list = []Employee{
    {"Hao", 44, 0, 8000},
    {"Bob", 34, 10, 5000},
    {"Alice", 23, 5, 9000},
    {"Jack", 26, 0, 4000},
    {"Tom", 48, 9, 7500},
    {"Marry", 29, 0, 6000},
    {"Mike", 32, 8, 4000},
}
```

```
func EmployeeCountIf(list []Employee, fn func(e *Employee) bool) int {
    count := 0
    for i, _ := range list {
        if fn(&list[i]) {
            count += 1
        }
    }
    return count
}
```

```
old := EmployeeCountIf(list, func(e *Employee) bool {
    return e.Age > 40
})
fmt.Printf("old people: %d\n", old)
//old people: 2
```

```
high_pay := EmployeeCountIf(list, func(e *Employee) bool {
    return e.Salary >= 6000
})
fmt.Printf("High Salary people: %d\n", high_pay)
//High Salary people: 4
```

```
func EmployeeFilterIn(list []Employee, fn func(e *Employee) bool) []Employee {
    var newList []Employee
    for i, _ := range list {
        if fn(&list[i]) {
            newList = append(newList, list[i])
        }
    }
    return newList
}
```

```
no_vacation := EmployeeFilterIn(list, func(e *Employee) bool {
    return e.Vacation == 0
})
fmt.Printf("People no vacation: %v\n", no_vacation)
//People no vacation: [{Hao 44 0 8000} {Jack 26 0 4000} {Marry 29 0 6000}]
```

```
func EmployeeSumIf(list []Employee, fn func(e *Employee) int) int {
    var sum = 0
    for i, _ := range list {
        sum += fn(&list[i])
    }
    return sum
}
```

```
total_pay := EmployeeSumIf(list, func(e *Employee) int {
    return e.Salary
})

fmt.Printf("Total Salary: %d\n", total_pay)
//Total Salary: 43500
```

```
younger_pay := EmployeeSumIf(list, func(e *Employee) int {
    if e.Age < 30 {
        return e.Salary
    } else {
        return 0
    }
})
```

Generic Map

```go
func Transform(slice, fn interface{}) interface{} {
    return transform(slice, fn, false)
}
```

```go
func TransformInPlace(slice, fn interface{}) interface{} {
    return transform(slice, fn, true)
}
```

```go
func transform(slice, function interface{}, inPlace bool) interface{} {
    //check the `slice` type is Slice
    sliceInType := reflect.ValueOf(slice)
    if sliceInType.Kind() != reflect.Slice {
        panic("transform: not slice")
    }

    //check the function signature
    fn := reflect.ValueOf(function)
    elemType := sliceInType.Type().Elem()
    if !verifyFuncSignature(fn, elemType, nil) {
        panic("trasform: function must be of type func(" + sliceInType.Type().Elem().String() + ") outputElemType")
    }

    sliceOutType := sliceInType
    if !inPlace {
        sliceOutType = reflect.MakeSlice(reflect.SliceOf(fn.Type().Out(0)), sliceInType.Len(), sliceInType.Len())
    }
    for i := 0; i < sliceInType.Len(); i++ {
        sliceOutType.Index(i).Set(fn.Call([]reflect.Value{sliceInType.Index(i)})[0])
    }
    return sliceOutType.Interface()

}
```

Verify Function Signature

```go
func verifyFuncSignature(fn reflect.Value, types ...reflect.Type) bool {

    //Check it is a funciton
    if fn.Kind() != reflect.Func {
        return false
    }
    // NumIn() - returns a function type's input parameter count.
    // NumOut() - returns a function type's output parameter count.
    if (fn.Type().NumIn() != len(types)-1) || (fn.Type().NumOut() != 1) {
        return false
    }
    // In() - returns the type of a function type's i'th input parameter.
    for i := 0; i < len(types)-1; i++ {
        if fn.Type().In(i) != types[i] {
            return false
        }
    }
    // Out() - returns the type of a function type's i'th output parameter.
    outType := types[len(types)-1]
    if outType != nil && fn.Type().Out(0) != outType {
        return false
    }
    return true
}
```

Generic Map Test

```go
func TestTransformString(t *testing.T) {
    list := []string{"1", "2", "3", "4", "5", "6"}
    expect := []string{"111","222","333","444","555","666"}
    result := Transform(list, func(a string) string{
        return a +a +a
    })

    if !reflect.DeepEqual(expect, result) {
        t.Fatalf("Transform failed: expect %v got %v",
            expect, result)
    }
}
```

```go
func TestTransformnPlace(t *testing.T) {
    list := []int{1, 2, 3, 4, 5, 6, 7, 8, 9}
    expect := []int{3, 6, 9, 12, 15, 18, 21, 24, 27}
    TransformInPlace(list, func (a int) int {
        return a*3
    })

    if !reflect.DeepEqual(expect, list) {
        t.Fatalf("Transform failed: expect %v got %v",
                expect, list)
    }
}
```

```go
func TestMapEmployee(t *testing.T) {
    var list = []Employee{
        {"Hao", 44, 0, 8000},
        {"Bob", 34, 10, 5000},
        {"Alice", 23, 5, 9000},
        {"Jack", 26, 0, 4000},
        {"Tom", 48, 9, 7500},
    }
    var expect = []Employee{
        {"Hao", 45, 0, 9000},
        {"Bob", 35, 10, 6000},
        {"Alice", 24, 5, 10000},
        {"Jack", 27, 0, 5000},
        {"Tom", 49, 9, 8500},
    }

    result := TransformInPlace(list, func(e Employee) Employee {
        e.Salary += 1000
        e.Age += 1
        return e
    })

    if !reflect.DeepEqual(expect, list) {
        t.Fatalf("Transform failed: expect %v got %v",
            expect, list)
    }
}
```

Generic Reduce

```
func Reduce(slice, pairFunc, zero interface{}) interface{} {
    sliceInType := reflect.ValueOf(slice)
    if sliceInType.Kind() != reflect.Slice {
        panic("reduce: wrong type, not slice")
    }

    len := sliceInType.Len()
    if len == 0 {
        return zero
    } else if len == 1 {
        return sliceInType.Index(0)
    }

    elemType := sliceInType.Type().Elem()
    fn := reflect.ValueOf(pairFunc)
    if !verifyFuncSignature(fn, elemType, elemType, elemType) {
    t := elemType.String()
        panic("reduce: function must be of type func(" + t +
            ", " + t + ") " + t)
    }

    var ins [2]reflect.Value
    ins[0] = sliceInType.Index(0)
    ins[1] = sliceInType.Index(1)
    out := fn.Call(ins[:])[0]

    for i := 2; i < len; i++ {
        ins[0] = out
        ins[1] = sliceInType.Index(i)
        out = fn.Call(ins[:])[0]
    }
    return out.Interface()
}
```

```
func mul(a, b int) int {
    return a * b
}

a := make([]int, 10)
for i := range a {
        a[i] = i + 1
}
// Compute 10!
out := Reduce(a, mul, 1).(int)
```

```
var list = []Employee{
    {"Hao", 44, 0, 8000},
    {"Bob", 34, 10, 5000},
    {"Alice", 23, 5, 9000},
    {"Jack", 26, 0, 4000},
    {"Tom", 48, 9, 7500},
    {"Marry", 29, 0, 6000},
    {"Mike", 32, 8, 4000},
}
result := Reduce(list, func(a, b Employee) Employee {
    return Employee{"Total Salary",0,0,a.Salary + b.Salary}
}, 0)

expect := 43500
if result.(Employee).Salary != expect {
    t.Fatalf("expected %v got %v", expect, result)
}
```

Generic Filter

```go
var boolType = reflect.ValueOf(true).Type()

func filter(slice, function interface{}, inPlace bool) (interface{}, int) {
    sliceInType := reflect.ValueOf(slice)
    if sliceInType.Kind() != reflect.Slice {
        panic("filter: wrong type, not a slice")
    }

    fn := reflect.ValueOf(function)
    elemType := sliceInType.Type().Elem()
    if !verifyFuncSignature(fn, elemType, boolType) {
        panic("filter: function must be of type func(" +
             elemType.String() + ") bool")
    }

    var which []int
    for i := 0; i < sliceInType.Len(); i++ {
        if fn.Call([]reflect.Value{sliceInType.Index(i)})[0].Bool() {
            which = append(which, i)
        }
    }

    out := sliceInType

    if !inPlace {
        out = reflect.MakeSlice(sliceInType.Type(), len(which), len(which))
    }
    for i := range which {
        out.Index(i).Set(sliceInType.Index(which[i]))
    }

    return out.Interface(), len(which)
}
```

```go
func Filter(slice, fn interface{}) interface{}
{
    result, _ := filter(slice, fn, false)
    return result
}
```

```go
func FilterInPlace(slicePtr, fn interface{}) {
    in := reflect.ValueOf(slicePtr)
    if in.Kind() != reflect.Ptr {
        panic("FilterInPlace: wrong type, " +
            "not a pointer to slice")
    }
    _, n := filter(in.Elem().Interface(), fn, true)
   in.Elem().SetLen(n)
}
```

```go
func isEven(a int) bool {
    return a%2 == 0
}
```

```go
func isOddString(s string) bool {
    i,_:= strconv.ParseInt(s, 10, 32)
    return i%2 == 1
}
```

```go
a := []int{1, 2, 3, 4, 5, 6, 7, 8, 9}
result := Filter(a, isEven)
//{2, 4, 6, 8}
```

```go
s := []string{"1","2","3","4","5","6","7","8","9"}

FilterInPlace(&s, isOddString)
// {"1", "3", "5", "7", "9"}
```

Notes

Warnning

Reflection is generally **SLOW** and should be avoided for latency-sensitive applications.

Why Go haven't Map/Reduce?

I wanted to see how hard it was to implement this sort of thing in Go, with as nice an API as I could manage. It wasn't hard.

Having written it a couple of years ago, I haven't had occasion to use it once. Instead, I just use "for" loops.

Personally, I think Rob doesn't write business code…

You shouldn't use it either.

https://github.com/robpike/filter

Type Assert & Reflection

Type Assert

```
//Container is a generic container, accepting anything.
type Container []interface{}

//Put adds an element to the container.
func (c *Container) Put(elem interface{}) {
    *c = append(*c, elem)
}

//Get gets an element from the container.
func (c *Container) Get() interface{} {
    elem := (*c)[0]
    *c = (*c)[1:]
    return elem
}

intContainer := &Container{}
intContainer.Put(7)
intContainer.Put(42)

// assert that the actual type is int
elem, ok := intContainer.Get().(int)
if !ok {
    fmt.Println("Unable to read an int from intContainer")
}

fmt.Printf("assertExample: %d (%T)\n", elem, elem)
```

Reflection

```
type Cabinet struct {
    s reflect.Value
}

func NewCabinet(t reflect.Type) *Cabinet {
    return &Cabinet{
        s: reflect.MakeSlice(reflect.SliceOf(t), 0, 10),
    }
}

func (c *Cabinet) Put(val interface{}) {
    if reflect.ValueOf(val).Type() != c.s.Type().Elem() {
        panic(fmt.Sprintf("Put: cannot put a %T into "+
            "a slice of %s", val, c.s.Type().Elem()))
    }
    c.s = reflect.Append(c.s, reflect.ValueOf(val))
}

func (c *Cabinet) Get(retref interface{}) {
    retref = c.s.Index(0)
    c.s = c.s.Slice(1, c.s.Len())
}

f := 3.14152
c := NewCabinet(reflect.TypeOf(f))
c.Put(f)
```

Can we do better?

Can we do something like C++ template,
which could generate the code for specify data type ?

```
template <class T>
T GetMax (T a, T b)  {
   T result;
   result = (a>b)? a : b;
   return (result);
}
```

```
int main () {
   int i=5, j=6, k;
   long l=10, m=5, n;
   k=GetMax<int>(i,j);
   n=GetMax<long>(l,m);
   cout << k << endl; cout << n << endl;
   return 0;
}
```

Go Generation

Go Generate – Example 1

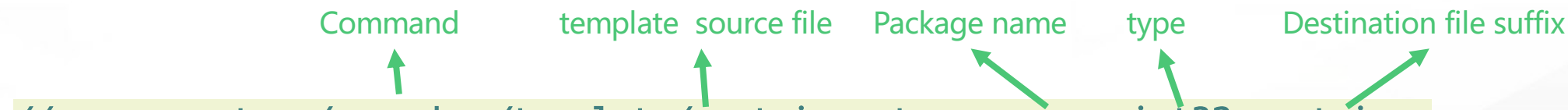
Template – container.tmp.go

```
package PACKAGE_NAME

type GENERIC_NAMEContainer struct {
    s []GENERIC_TYPE
}

func NewGENERIC_NAMEContainer() *GENERIC_NAMEContainer {
    return &GENERIC_NAMEContainer{s: []GENERIC_TYPE{}}
}

func (c *GENERIC_NAMEContainer) Put(val GENERIC_TYPE) {
    c.s = append(c.s, val)
}

func (c *GENERIC_NAMEContainer) Get() GENERIC_TYPE {
    r := c.s[0]
    c.s = c.s[1:]
    return r
}
```

Script – gen.sh

```
#!/bin/bash

set -e

SRC_FILE=${1}
PACKAGE=${2}
TYPE=${3}
DES=${4}
#uppcase the first char
PREFIX="$(tr '[:lower:]' '[:upper:]' <<< ${TYPE:0:1})${TYPE:1}"


DES_FILE=$(echo ${TYPE}| tr '[:upper:]' '[:lower:]')_${DES}.go

sed 's/PACKAGE_NAME/'"${PACKAGE}"'/g' ${SRC_FILE} | \
    sed 's/GENERIC_TYPE/'"${TYPE}"'/g' | \
    sed 's/GENERIC_NAME/'"${PREFIX}"'/g' > ${DES_FILE}
```

Go Generate – Example 1




```
//go:generate ./gen.sh ./template/container.tmp.go gen uint32 container
func generateUint32Example() {
    var u uint32 = 42
    c := NewUint32Container()
    c.Put(u)
    v := c.Get()
    fmt.Printf("generateExample: %d (%T)\n", v, v)
}


//go:generate ./gen.sh ./template/container.tmp.go gen string container
func generateStringExample() {
    var s string = "Hello"
    c := NewStringContainer()
    c.Put(s)
    v := c.Get()
    fmt.Printf("generateExample: %s (%T)\n", v, v)
}
```

Go Generate – Example 1

uint32_container.go

```go
package gen

type Uint32Container struct {
    s []uint32
}

func NewUint32Container() *Uint32Container {
    return &Uint32Container{s: []uint32{}}
}

func (c *Uint32Container) Put(val uint32) {
    c.s = append(c.s, val)
}

func (c *Uint32Container) Get() uint32 {
    r := c.s[0]
    c.s = c.s[1:]
    return r
}
```

string_container.go

```go
package gen

type StringContainer struct {
    s []string
}

func NewStringContainer() *StringContainer {
    return &StringContainer{s: []string{}}
}

func (c *StringContainer) Put(val string) {
    c.s = append(c.s, val)
}

func (c *StringContainer) Get() string {
    r := c.s[0]
    c.s = c.s[1:]
    return r
}
```

After run " go generate" command, two source code go file would be generated

Go Generate – Example 2

Template – filter.tmp.go

```
package PACKAGE_NAME


type GENERIC_NAMEList []GENERIC_TYPE


type GENERIC_NAMEToBool func(*GENERIC_TYPE) bool


func (al GENERIC_NAMEList) Filter(f GENERIC_NAMEToBool) GENERIC_NAMEList {
    var ret GENERIC_NAMEList
    for _, a := range al {
        if f(&a) {
            ret = append(ret, a)
        }
    }
    return ret
}
```

Go Generate – Example 2

```go
//go:generate ./gen.sh ./template/filter.tmp.go gen int filter
func filterIntArrayExample() {
    a := IntList{1, 2, 3, 4, 5, 6, 7, 8, 9}
    b := a.Filter(func(i *int) bool {
        return *i%2 == 0
    })


    fmt.Println(b)
}
```

Int_filter.go

```go
package gen

type IntList []int

type IntToBool func(*int) bool

func (al IntList) Filter(f IntToBool) IntList {
    var ret IntList
    for _, a := range al {
        if f(&a) {
            ret = append(ret, a)
        }
    }
    return ret
}
```

Go Generate – Example 2

```go
//go:generate ./gen.sh ./template/filter.tmp.go gen Employee filter
func filterEmployeeExample() {

var list = EmployeeList{
    {"Hao", 44, 0, 8000},
    {"Bob", 34, 10, 5000},
    {"Alice", 23, 5, 9000},
    {"Jack", 26, 0, 4000},
    {"Tom", 48, 9, 7500},
}


var filter EmployeeList
filter = list.Filter(func(e *Employee) bool {
        return e.Age > 40
})


fmt.Println("----- Employee.Age > 40 ------")
for _, e := range filter {
        fmt.Println(e)
}
```

```go
type Employee struct {
    Name     string
    Age      int
    Vacation int
    Salary   int
}
```

employee_filter.go

```go
package gen

type EmployeeList []Employee

type EmployeeToBool func(*Employee) bool

func (al EmployeeList) Filter(f EmployeeToBool) EmployeeList
{
    var ret EmployeeList
    for _, a := range al {
        if f(&a) {
                ret = append(ret, a)
        }
    }
    return ret
}
```

The 3rd-Pary Generate Libs

Genny - https://github.com/cheekybits/genny
Generic - https://github.com/taylorchu/generic
GenGen - https://github.com/joeshaw/gengen
Gen - https://github.com/clipperhouse/gen

Pros / Cons

Pros / Cons

Copy & Paste

Pros:
- Quick
- Needs no external libraries or tools

Cons:
- Code bloat
- Breaks the DRY principle

Code Generation

Pros:
- Clear code possible (depending on the tool),
- Compile-time type checking
- Allow writing tests against the generic code template
- No runtime overhead

Cons:
- Possible binary bloat,
- requires an extra build step and a third party tool

Type Assertions

Pros:
- Code stays quite clear
- No needs external libraries or tools

Cons:
- No compile-time type checking
- Runtime overhead from converting to/from interfaces
- Caller is required to do the type assertion

Interfaces & Reflection

Pros:
- Versatile & Flexible
- No needs external libraries or tools

Cons:
- Reflection makes code much more complicated.
- Runtime overhead

Decoration

Basice Example 1

```
func decorator(f func(s string)) func(s string) {
    return func(s string) {
        fmt.Println("Started")
        f(s)
        fmt.Println("Done")
    }
}

func Hello(s string) {
    fmt.Println(s)
}

func main() {
        decorator(Hello)("Hello, World!")
}
```

Return an function wrapped another function with same parameters

Basic Example 2

```go
func Sum1(start, end int64) int64 {
  var sum int64
  sum = 0
  if start > end {
    start, end = end, start
  }
  for i := start; i <= end; i++ {
    sum += i
  }
  return sum
}


func Sum2(start, end int64) int64 {
  if start > end {
    start, end = end, start
  }
  return (end - start + 1) * (end + start) / 2
}
```

```go
type SumFunc func(int64, int64) int64

func getFunctionName(i interface{}) string {
  return runtime.FuncForPC(reflect.ValueOf(i).Pointer()).Name()
}

func timedSumFunc(f SumFunc) SumFunc {
  return func(start, end int64) int64 {
    defer func(t time.Time) {
      fmt.Printf("--- Time Elapsed (%s): %v ---\n",
          getFunctionName(f), time.Since(t))
    }(time.Now())
    return f(start, end)
  }
}
```

```go
sum1 := timedSumFunc(Sum1)
sum2 := timedSumFunc(Sum2)
fmt.Printf("%d, %d\n", sum1(1, 10000000), sum2(1, 10000000))
```

HTTP Server Example

```go
func WithServerHeader(h http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        log.Println("--->WithServerHeader()")
        w.Header().Set("Server", "HelloServer v0.0.1")
        h(w, r)
    }
}
func WithAuthCookie(h http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        log.Println("--->WithAuthCookie()")
        cookie := &http.Cookie{Name: "Auth", Value: "Pass", Path: "/"}
        http.SetCookie(w, cookie)
        h(w, r)
    }
}
func WithBasicAuth(h http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        log.Println("--->WithBasicAuth()")
        cookie, err := r.Cookie("Auth")
        if err != nil || cookie.Value != "Pass" {
            w.WriteHeader(http.StatusForbidden)
            return
        }
        h(w, r)
    }
}
```

```go
func WithDebugLog(h http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        log.Println("--->WithDebugLog")
        r.ParseForm()
        log.Println(r.Form)
        log.Println("path", r.URL.Path)
        log.Println("scheme", r.URL.Scheme)
        log.Println(r.Form["url_long"])
        for k, v := range r.Form {
            log.Println("key:", k)
            log.Println("val:", strings.Join(v, ""))
        }
        h(w, r)
    }
}
```

```go
func hello(w http.ResponseWriter, r *http.Request) {
    log.Printf("Recieved Request %s from %s\n", r.URL.Path, r.RemoteAddr)
    fmt.Fprintf(w, "Hello, World! "+r.URL.Path)
}
func main() {
    http.HandleFunc("/v1/hello", WithServerHeader(WithAuthCookie(hello)))
    http.HandleFunc("/v2/hello", WithServerHeader(WithBasicAuth(hello)))
    http.HandleFunc("/v3/hello", WithServerHeader(WithBasicAuth(WithDebugLog(hello))))
    err := http.ListenAndServe(":8080", nil)
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```

Generic Decorator

```
func Decorator(decoPtr, fn interface{}) (err error) {
    var decoratedFunc, targetFunc reflect.Value
    if decoPtr == nil ||
      reflect.TypeOf(decoPtr).Kind() != reflect.Ptr ||
      reflect.ValueOf(decoPtr).Elem().Kind() != reflect.Func {
        err = fmt.Errorf("Need a function porinter!")
        return
    }

    decoratedFunc = reflect.ValueOf(decoPtr).Elem()
    targetFunc = reflect.ValueOf(fn)
    if targetFunc.Kind() != reflect.Func {
        err = fmt.Errorf("Need a function!")
        return
    }

    v := reflect.MakeFunc(targetFunc.Type(),
            func(in []reflect.Value) (out []reflect.Value) {
                fmt.Println("before")
                if targetFunc.Type().IsVariadic() {
                    out = targetFunc.CallSlice(in)
                } else {
                    out = targetFunc.Call(in)
                }
                fmt.Println("after")
                Return
            }
         )

    decoratedFunc.Set(v)
    return
}
```

decoPtr - output parameter
the new function has been decorated

fn – input parameter
the function need be decorated

```
func bar(a, b string) string {
    fmt.Printf("%s, %s \n", a, b)
    return a + b
}


mybar := bar
err := Decorator(&mybar, bar)
if err != nil {
    panic(err)
}
fmt.Println(mybar("hello,", "world!"))
```

Kubernetes Visitor

Visitor

```
type VisitorFunc func(*Info, error) error

type Visitor interface {
    Visit(VisitorFunc) error
}

type Info struct {
    Namespace   string
    Name        string
    OtherThings string
}
func (info *Info) Visit(fn VisitorFunc) error {
       return fn(info, nil)
}
```

```
type OtherThingsVisitor struct {
    visitor Visitor
}

func (v OtherThingsVisitor) Visit(fn VisitorFunc) error {
    return v.visitor.Visit(func(info *Info, err error) error {
        fmt.Println("OtherThingsVisitor() before call function")
        err = fn(info, err)
        if err == nil {
            fmt.Printf("==> OtherThings=%s\n", info.OtherThings)
        }
        fmt.Println("OtherThingsVisitor() after call function")
        return err
    })
}
```

```
type LogVisitor struct {
    visitor Visitor
}


func (v LogVisitor) Visit(fn VisitorFunc) error {
    return v.visitor.Visit(func(info *Info, err error) error {
        fmt.Println("LogVisitor() before call function")
        err = fn(info, err)
        fmt.Println("LogVisitor() after call function")
        return err
    })
}
```

```
type NameVisitor struct {
    visitor Visitor
}

func (v NameVisitor) Visit(fn VisitorFunc) error {
    return v.visitor.Visit(func(info *Info, err error) error {
        fmt.Println("NameVisitor() before call function")
        err = fn(info, err)
        if err == nil {
            fmt.Printf("==> Name=%s, NameSpace=%s\n",
                                info.Name, info.Namespace)
        }
        fmt.Println("NameVisitor() after call function")
        return err
    })
}
```

Visitor Usage

```
func main() {
    info := Info{}
    var v Visitor = &info
    v = LogVisitor{v}
    v = NameVisitor{v}
    v = OtherThingsVisitor{v}


    loadFile := func (info *Info, err error) error {
        info.Name = "Hao Chen"
        info.Namespace = "MegaEase"
        info.OtherThings ="We are running as remote team."
        return nil
    }
    v.Visit(loadFile)
}
```

```
LogVisitor() before call function
NameVisitor() before call function
OtherThingsVisitor() before call function
==> OtherThings=We are running as remote team.
OtherThingsVisitor() after call function
==> Name=Hao Chen, NameSpace=MegaEase
NameVisitor() after call function
LogVisitor() after call function
```

Decorate Visitor

```go
type DecoratedVisitor struct {
    visitor    Visitor
    decorators []VisitorFunc
}

func NewDecoratedVisitor(v Visitor, fn ...VisitorFunc) Visitor {
    if len(fn) == 0 {
        return v
    }
    return DecoratedVisitor{v, fn}
}

// Visit implements Visitor
func (v DecoratedVisitor) Visit(fn VisitorFunc) error {
    return v.visitor.Visit(func(info *Info, err error) error {
        if err != nil {
                return err
        }
        if err := fn(info, nil); err !=nil {
                return err
        }
        for i := range v.decorators {
            if err := v.decorators[i](info, nil); err != nil {
                return err
            }
        }
        return nil
    })
}
```

```go
func NameVisitor(info *Info, err error ) error {
    fmt.Printf("Name=%s, Namespace=%s\n",
                info.Name, info.Namespace)
    return nil
}

func OtherVisitor(info *Info, err error ) error {
    fmt.Printf("Other=%s\n", info.OtherThings)
    return nil
}

func LoadFile(info *Info, err error) error {
    info.Name = "Hao Chen"
    info.Namespace = "MegaEase"
    info.OtherThings ="We are running as remote team."
    return nil
}
```

```go
func main() {
    info := Info{}
    var v Visitor = &info

    v = NewDecoratedVisitor(v, NameVisitor, OtherVisitor)

    v.Visit(LoadFile)
}
```

Pipeline

Basic Example

```
http.HandleFunc("/v2/hello", WithServerHeader(WithBasicAuth(hello)))
http.HandleFunc("/v3/hello", WithServerHeader(WithBasicAuth(WithDebugLog(hello))))
```

```
type HttpHandlerDecorator func(http.HandlerFunc) http.HandlerFunc

func Handler(h http.HandlerFunc, decors ...HttpHandlerDecorator) http.HandlerFunc {
    for i := range decors {
        d := decors[len(decors)-1-i] // iterate in reverse
        h = d(h)
    }
    return h
}
```

```
http.HandleFunc(“/v4/hello”, Handler(hello, WithServerHeader, WithBasicAuth, WithDebugLog))
```

Generic Pipeline

```go
func Pipe(fns ...interface{}) Pipeline {
    if len(fns) <= 0 {
        return empty
    }

    return func(args ...interface{}) (interface{}, error) {

        var inputs []reflect.Value
        for _, arg := range args {
            inputs = append(inputs, reflect.ValueOf(arg))
        }

        for _, fn := range fns {
            outputs := reflect.ValueOf(fn).Call(inputs)
            inputs = inputs[:0] //clean inputs
            fnType := reflect.TypeOf(fn)

            for oIdx, output := range outputs {
                if fnType.Out(oIdx).Implements(errType) {
                    if  output.IsNil() {
                        continue
                    }
                    err := fmt.Errorf("%s() failed: %w", getFunctionName(fn),
                    output.Interface().(error))
                    return nil, err
                }
                inputs = append(inputs, output)
            }
        }
        return inputs[0].Interface(), nil
    }
}
```

```go
// errType is the type of error interface.
var errType = reflect.TypeOf((*error)(nil)).Elem()

// Pipeline is the func type for the pipeline result.
type Pipeline func(...interface{}) (interface{}, error)

func empty(...interface{}) (interface{}, error) { return nil, nil }

func getFunctionName(i interface{}) string {
    return runtime.FuncForPC(reflect.ValueOf(i).Pointer()).Name()
}
```

```go
func TestPipeError(t *testing.T) {
    pipe := Pipe(
        func(x int) (int, error) {
            if x == 0 {
                return 0, errors.New("x should not be zero")
            }
            return x, nil
        },
        func(x int) float32 { return 100.0 / float32(x) },
        func (x float32) string { return fmt.Sprintf("%f",x) },
     )
    result, err := pipe(3)
    expect := "33.333332"
    if err != nil {
        t.Fatal(err)
    }
    if result.(string) != expect {
        t.Fatalf("pipeline failed: expect %v got %v", expect, result)
    }
}
```

Channel Pipeline

```go
func echo(nums []int) <-chan int {
    out := make(chan int)

    go func() {
        for _, n := range nums {
            out <- n
        }
        close(out)
    }()

    return out
}
```

```go
func sq(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        for n := range in {
            out <- n * n
        }
        close(out)
    }()
    return out
}
```

```go
func sum(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        var sum = 0
        for n := range in {
            sum += n
        }
        out <- sum
        close(out)
    }()
    return out
}
```

echo $nums | sq | sum

```go
var nums = []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
for n := range sum(sq(echo(nums))) {
   fmt.Println(n) // 165
}
```

Go Concurrency Patterns: Pipelines and cancellation
https://blog.golang.org/pipelines

Fan-out & Fan-in

```
func merge(cs []<-chan int) <-chan int {

    var wg sync.WaitGroup
    out := make(chan int)

    wg.Add(len(cs))
    for _, c := range cs {
        go func (c <- chan int){
            for n := range c {
                out <- n
            }
            wg.Done()
        }(c)
    }

    go func() {
        wg.Wait()
        close(out)
    }()

    return out
}
```

```
func main() {
    nums := makeRange(1,1000)
    in := gen(nums)

    const nProcess = 5
    var chans [nProcess]<-chan int
    for i := range chans {
        chans[i] = sum(in)
    }

    for n := range sum(merge(chans[:])) {
        fmt.Println(n)
    }
}
```

```
func makeRange(min, max int) []int {
    a := make([]int, max-min+1)
    for i := range a {
        a[i] = min + i
    }
    return a
}
```

Multiple functions can read from the same channel until that channel is closed; this is called *fan-out*. This provides a way to distribute work amongst a group of workers to parallelize CPU use and I/O.

A function can read from multiple inputs and proceed until all are closed by multiplexing the input channels onto a single channel that's closed when all the inputs are closed. This is called *fan-in*.

Further Readings

- **Go Concurrency Patterns** - **Rob Pike** - 2012 Google I/O
  presents the basics of Go's concurrency primitives and several ways to apply them.
  https://www.youtube.com/watch?v=f6kdp27TYZs

- **Advanced Go Concurrency Patterns** - **Rob Pike** – 2013 Google I/O
  covers more complex uses of Go's primitives, especially select.
  https://blog.golang.org/advanced-go-concurrency-patterns

- **Squinting at Power Series** - **Douglas McIlroy**'s paper
  shows how Go-like concurrency provides elegant support for complex calculations.
  https://swtch.com/~rsc/thread/squint.pdf

Rob .C Pike

Born	1956
Nationality	Canadian
Alma mater	•University of Toronto (BS) •California Institute of Technology
Occupation	Software engineer
Employer	Google
Known for	Plan 9, UTF-8, Go
Spouse(s)	Renée French
Website	herpolhode.com/rob/

Thanks to teach people to write good code!

GOPHER CHINA 2020

中国 上海 / 2020-11.21-22