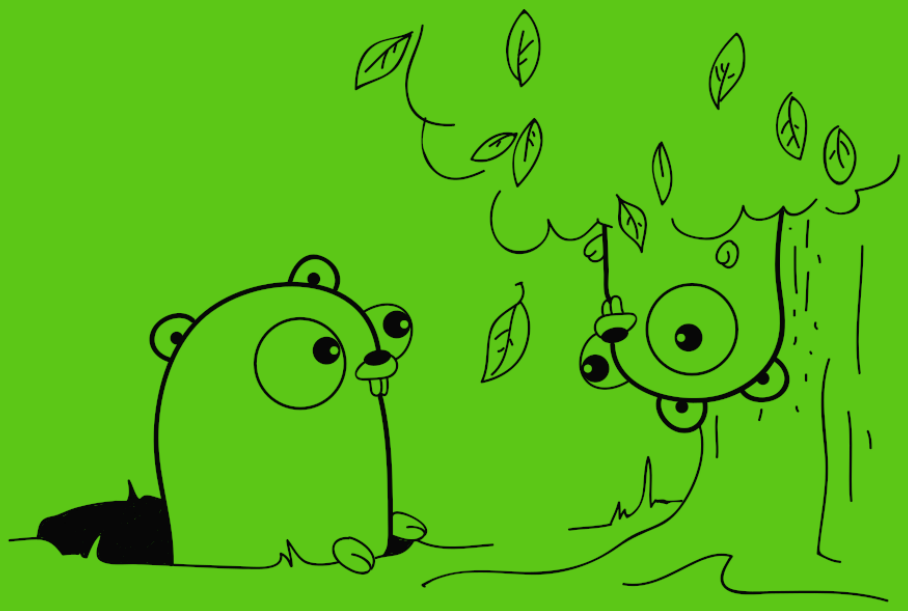
# Go Details & Tips 101

Tapir Liu

# Contents

# Go Details & Tips 101

-= v1.17.i-rev-bcf4d80-2022/02/10 =-



by Tapir Liu

# Chapter 1

# Acknowledgments

Some of the details and tips in this book are collected from the Internet, some ones are found by myself. I will try to list the source of a detail if it is possible. But I'm sorry that it is impossible task to do this for every detail.

Thanks to Olexandr Shalakhin for the permission to use one of the wonderful gopher icon designs in the cover image. And thanks to Renee French for designing the lovely gopher cartoon character.

Thanks to the authors of the following open source software and libraries, which are used in building this book:

- golang, https://go.dev/
- gomarkdown, https://github.com/gomarkdown/markdown
- goini, https://github.com/zieckey/goini
- go-epub, https://github.com/bmaupin/go-epub
- pandoc, https://pandoc.org
- calibre, https://calibre-ebook.com/
- GIMP, https://www.gimp.org

# Chapter 2

# About Go Details & Tips 101

This book collects many details and provides several tips in Go programming. The details and tips are categorized into

- syntax and semantics related
- conversions related
- comparisons related
- runtime related
- standard packages related

Most of the details are Go specific, but several of them are language independent.

Several details shown in this book might become invalid in future Go versions. But at least up to now (Go 1.17), they are valid.

## 2.1 About the author

Tapir is the author of this book. He also wrote the Go 101 book. He is planning to write some other **Go 101** series books. Please look forward to.

Tapir was ever (maybe will be again) an indie game developer. You can find his games here: tapirgames.com.

## 2.2 Feedback

Welcome to improve this book by submitting corrections to Go 101 issue list (https://github.com/go101/go101) for all kinds of mistakes, such as typos, grammar errors, wording inaccuracies, wrong explanations, description flaws, code bugs, etc.

It is also welcome to send your feedback to the Go 101 twitter account: @go100and1 (https://twitter.com/go100and1).

# Chapter 3

# Syntax and Semantics Related

## 3.1 Zero-size types/values

The size of a struct type without non-zero-size fields is zero. The size of an array type which length is zero or which element size is zero is also zero. These could be proved by the following program, which prints three zeros.

```
package main

import "unsafe"

type A [0][256]int

type S struct {
     x A
     y [1<<30]A
     z [1<<30]struct{}
}

type T [1<<30]S

func main() {
     var a A
     var s S
     var t T
     println(unsafe.Sizeof(a)) // 0
     println(unsafe.Sizeof(s)) // 0
     println(unsafe.Sizeof(t)) // 0
}
```

In Go, sizes are often denoted as `int` values. That means the largest possible length of an array is MaxInt, which value is $2^{63}$-1 on 64-bit OSes. However, the lengths of arrays with non-zero element sizes are hard limited by the official standard Go compiler and runtime.

An example:

```
var x [1<<63-1]struct{}     // okay
var y [2000000000+1]byte    // compilation error
```

```
var z = make([]byte, 1<<49) // panic: runtime error: makeslice: len out of range
```

## 3.2 How zero-size values are allocated is compiler dependent

In the current official standard Go compiler implementation (version 1.17), all local zero-size values allocated on heap share the same address. For example, the following prints `false` twice, then prints `true` twice.

```
package main

var g *[0]int
var a, b [0]int

//go:noinline
func f() *[0]int {
	return new([0]int)
}

func main() {
	// x and y are allocated on stack.
	var x, y, z, w [0]int
	// Make z and w escape to heap.
	g = &z; g = &w
	println(&b == &a)  // false
	println(&x == &y)  // false
	println(&z == &w)  // true
	println(&z == f()) // true
}
```

Please note that, the outputs of the above program depend on specific compilers. The outputs might be different for future official standard Go compiler versions.

## 3.3 Don't put a zero-size field as the final field of a struct type

In the following code, the size of the type `Tz` is larger than the type `Ty`.

```
package main

import "unsafe"

type Ty struct {
	_ [0]func()
	y int64
}

type Tz struct {
	z int64
	_ [0]func()
}
```

```
func main() {
	var y Ty
	var z Tz
	println(unsafe.Sizeof(y)) // 8
	println(unsafe.Sizeof(z)) // 16
}
```

Why the size of the type `Tz` is larger?

In the current standard Go runtime implementation, as long as a memory block is referenced by at least one alive pointer, that memory block will not be viewed as garbage and will not be collected.

All the fields of an addressable struct value can be taken addresses. If the size of the final field in a non-zero-size struct value is zero, then taking the address of the final field in the struct value will return an address which is beyond the allocated memory block for the struct value. The returned address may point to another allocated memory block which closely follows the one allocated for the non-zero-size struct value. As long as the returned address is stored in an alive pointer value, the other allocated memory block will not get garbage collected, which may cause memory leaking.

To avoid the kind of memory leak problems, the standard Go compiler will ensure that taking the address of the final field in a non-zero-size struct will never return an address which is beyond the allocated memory block for the struct. The standard Go compiler implements this by padding some bytes after the final zero-size field when needed.

So at least one byte is padded after the final (zero) field of the type `Tz`. This is why the size of the type `Tz` is larger than `Ty`.

In fact, on 64-bit OSes, 8 bytes are padded after the final (zero) field of `Tz`. To explain this, we should know two facts in the official standard compiler implementation:

1. The alignment guarantee of a struct type is the largest alignment guarantee of its fields.
2. A size of a type is always a multiple of the alignment guarantee of the type.

The first fact explains why the alignment guarantee of the type `Tz` is 8 (which is the alignment guarantee of the builtin `int64` type). The second fact explains why the size of the type `Tz` is 16.

Source: https://github.com/golang/go/issues/9401

## 3.4 Simulate `for i in 0..N` in some other languages

We could use a `for-range` loop to simulate `for i in 0..N` loops in some other languages, like the following code shows.

```
package main

const N = 8
var n = 8

func main() {
	for i := range [N]struct{}{} {
		println(i)
```

```
    }
    for i := range [N][0]int{} {
        println(i)
    }
    for i := range make([][0]int, n) {
        println(i)
    }
}
```

The steps of the first two loops must be known at compile time, whereas the last one has not this requirement. But the last one allocates a little more memory (on stack, for the slice header).

## 3.5 There are several ways to create a slice

For example, each slice in the following code is created in a different way.

```
package main

func main() {
    var s0 = make([]int, 100)
    var s1 = []int{99: 0}
    var s2 = (&[100]int{})[:]
    var s3 = new([100]int)[:]
    // 100 100 100 100
    println(len(s0), len(s1), len(s2), len(s3))
}
```

## 3.6 `for i, v = range aContainer` actually iterates a copy of `aContainer`

For example, the following program will print `123` instead of `189`.

```
package main

func main() {
    var a = [...]int{1, 2, 3}
    for i, n := range a {
        if i == 0 {
            a[1], a[2] = 8, 9
        }
        print(n)
    }
}
```

If the ranged container is a large array, then the cost of making the copy will be large.

There is an exception: if the second iteration variable in a `for-range` is omitted or ignored, then the ranged container will not get copied, because it is unnecessary to make the copy. For example, in the following two loops, the array `a` is not copied.

```
func main() {
    var a = [...]int{1, 2, 3}
```

```
	for i := range a {
		print(i)
	}
	for i, _ := range a {
		print(i)
	}
}
```

In Go, an array owns its elements, but a slice just references its elements. Values are copied shallowly in Go, copying a value will not copy the values referenced by it. So copying a slice will not copy its elements. This could be reflected in the following program. The program prints `189`.

```
package main

func main() {
	var s = []int{1, 2, 3}
	for i, n := range s {
		if i == 0 {
			s[1], s[2] = 8, 9
		}
		print(n)
	}
}
```

## 3.7 Array pointers could be used as arrays in several situations

For example, the following code compiles and runs okay.

```
package main

func main() {
	var a = [128]int{3: 789}
	var pa = &a
	// Iterate array elements without copying array.
	for i, v := range pa {
		_, _ = i, v
	}
	// Get array length and capacity.
	_, _ = len(pa), cap(pa)
	// Access array elements.
	_ = pa[3]
	pa[3] = 555
	// Derive slices from array pointers.
	var _ []int = pa[:]
}
```

Range over a nil array pointer will not panic if the second iteration variable is omitted or ignored. For example, the first two loops in the following code both print `01234`, but the last one causes a panic.

```
package main
```

```
func main() {
	var pa *[5]string

	// Prints 01234
	for i := range pa {
		print(i)
	}

	// Prints 01234
	for i, _ := range pa {
		print(i)
	}

	// Panics
	for _, v := range pa {
		_ = v
	}
}
```

## 3.8 Some function calls are evaluated at compile time

The function calls evaluated at compile time are also called as constant calls, because their evaluation results are constant values.

All calls to the `unsafe.Sizeof`, `unsafe.Offsetof` and `unsafe.Alignof` are evaluated at compile time.

If the argument of a call to the builtin `len` and `cap` functions is a constant string, an array or a pointer to array, and the argument expression does not contain channel receives or non-constant function calls, then the call will be evaluated at compile time.

In evaluating constant calls to the just mentioned functions, only the types of involved arguments matter (except the arguments are constant strings), even if evaluating such an argument might cause a panic at run time.

For example, neither of the `f` and `g` functions in the following code will panic at run time.

```
package main

import "unsafe"

func f() {
	var v *int64 = nil
	println(unsafe.Sizeof(*v)) // 8
}

func g() {
	var t *struct {s [][16]int} = nil
	println(len(t.s[99]))  // 16
}

func main() {
	f()
```

```
	g()
}
```

On the other hand, either of the `f2` and `g2` functions will cause a panic at run time.

```
func f2() {
	var v *int64 = nil
	_ = *v
}

func g2() {
	var t *struct {s [][16]int} = nil
	_ = t.s[99]
}
```

Please note that the builtin `len` function is implicitly called in a `for-range` loop. Knowing this is the key to understand why the first two loops in the following code don't cause panics, but the last one does.

```
package main

type T struct {
	s []*[5]int
}

func main() {
	  var t *T
	  for i, _ := range t.s[99] { // not panic
	     print(i)
	  }
	  for i := range *t.s[99] { // not panic
	     print(i)
	  }
	  for i := range t.s { // panics
	     print(i)
	  }
}
```

Yes, the implicit `len(t.s[99])` and `len(*t.s[99])` calls are evaluated at compile time. Only the length (5 here) of the array value `t.s[99]` matters in the evaluations. However, the implicit call `len(t.s)` is evaluated at run time, so it causes a panic for `t` is a nil pointer.

As above mentioned, a call to the builtin `len` or `cap` function with an argument containing channel receives or non-constant function calls will not be evaluated at compile time. For example, the following code doesn't compile.

```
var c chan int
var s []byte
const X = len([1]int{<-c})    // error: len(...) is not a constant
const Y = cap([1]int{len(s)}) // error: cap(...) is not a constant
```

## 3.9 The official standard Go compiler doesn't support declaring package-level arrays with sizes larger than 2GB

For example, the following program fails to compile for an error `main.x: symbol too large (2000000001 bytes > 2000000000 bytes)`.

```
package main

var x [2000000000+1]byte

func main() {}
```

The size of a local heap-allocated array may be larger than 2GB. For example, the following program compiles okay.

```
package main

var y *[2000000000+1]byte

func main() {
	var x [2000000000+1]byte
	y = &x
}
```

Source: https://github.com/golang/go/issues/17378

## 3.10 Addressabilities of slice/array/map elements and struct fields

The following are some facts about the addressabilities of slice/array/map elements:

- Elements of a slice value are always addressable, whether or not the slice value is addressable.
- Elements of addressable array values are also addressable. Elements of unaddressable array values are also unaddressable.
- Elements of map values are always unaddressable.

Like arrays, fields of addressable struct values are also addressable. Fields of unaddressable struct values are also unaddressable.

For example, in the following code, the address taking operations in the function `foo` are all illegal, whereas the ones in the function `bar` are all legal.

```
type T struct {
	x int
}

func foo() {
	// Literals are unaddressable.
	_ = &([10]bool{}[1]) // error
	// Map elements are unaddressable.
	var mi = map[int]int{1: 0}
	_ = &(mi[1]) // error
```

```
	var ma = map[int][10]bool{2: [10]bool{}}
	_ = &(ma[2][1]) // error
	_ = &(T{}.x)    // error
}

func bar() {
	var _ = &([]int{1: 0}[1]) // okay
	// All variables are addressable.
	var a [10]bool
	_ = &(a[1]) // okay
	var t T
	_ = &(t.x) // okay
}
```

It is also illegal to derive slices from unaddressable arrays. So the following code also fails to compile.

```
var aSlice = [10]bool{}[:]
```

## 3.11 Composite literals are unaddressable, but they may be taken addresses

Composite literals include struct/array/slice/map value literals. They are unaddressable. Generally speaking, unaddressable values may not be taken addresses. However, there is an exception (a syntax sugar) made in Go: composite literals may be taken addresses.

For example, the following code compiles okay:

```
package main

type T struct {
	x int
}

func main() {
	// All the address taking operations are legal.
	_ = &T{}
	_ = &[8]byte{}
	_ = &[]byte{7: 0}
	_ = &map[int]bool{}
}
```

Please note that the precedences of the index operator `[]` and property selection operator `.` are both higher than the address-taking operator `&`. For example, both of the two lines in the following code don't compile.

```
	_ = &T{}.x // error
	_ = &[8]byte{}[1] // error
```

The reason why they fails to compile is the above code lines are equivalent to the following lines.

```
	_ = &(T{}.x) // error
	_ = &([8]byte{}[1]) // error
```

On the other hand, the following lines compile okay.

```
_ = (&T{}).x // okay
_ = (&[8]byte{})[1] // okay
```

## 3.12 One-line trick to create pointers to a non-zero bool/numeric/string values

We may take addresses of composite literals, but we may not take addresses of other literals. For example, all the code lines shown below are illegal.

```
var pb = &true
var pi = &123
var pb = &"abc"
```

In fact, we could achieve the similar effects, in one-line but more verbose forms:

```
var pb = &(&[1]bool{true})[0]
var pi = &(&[1]int{9})[0]
var ps = &(&[1]string{"Go"})[0]

// The following way is less verbose but less efficient.
var pb2 = &([]bool{true})[0]
var pi2 = &([]int{9})[0]
var ps2 = &([]string{"Go"})[0]
```

The trick is useful when filling a large struct value (often used as a configuration). For example, without using this trick, the code needs to be written as:

```
var x = true

var cfg = mypkg.Config {
    ... // many other options

    // Three possible values: nil, &false, &true.
    OptionsX: &x,

    ...
}
```

If there are many other options, the distance from the declaration of `x` to its use would be very far. This is not a big problem, but hurts code readability to some extend.

Instead, we could use the following code to avoid the far distance problem:

```
var cfg = mypkg.Config {
    ... // many other options

    // Three possible values: nil, &false, &true.
    OptionsX: &(&[1]bool{true})[0],

    ... // more options
}
```

A more verbose solution is to use an anonymous function call:

```
var cfg = mypkg.Config {
    ...
    OptionsX: func() *bool {var x = true; return &x},
    ...
}
```

Learned from this issue thread.

## 3.13 Unaddressable values are not modifiable, but map elements may be modified (in a whole)

The above sections have mentioned that map elements are unaddressable and can't be taken address. Generally, unaddressable values are also unmodifiable, but map elements may be modified, though each of the modifications must be in a whole. That means a map element can't be modified partially.

An example:

```
package main

type T struct {
    x int
}

func main() {
    var mt = map[int]T{1: T{x: 2}}
    var ma = map[int][3]bool{}
    mt[1] = T{x: 3}          // okay
    ma[1] = [3]bool{0: true} // okay

    // The two lines are viewed as partial modifications.
    mt[1].x = 3    // error
    ma[1][0] = true// error
}
```

## 3.14 The second argument of a `make` call to create a map is viewed as a hint

Each non-nil map maintains a backing array to hold its entries. The array might grow as needed along with more and more entries are put into that map.

A `make` call to create a map will allocate enough space for the created map to hold the specified number of entries (without growing the backing array). This argument is optional, its default value is compiler dependent.

The argument could be a zero, even a non-constant negative. For example, the following code runs okay (it doesn't panic).

```
var n = -99
var m = make(map[string]int, n)
```

Source: https://github.com/golang/go/issues/46909

## 3.15 Use maps to emulate sets

Go supports builtin map types, but doesn't support set types. We could use map types to emulate set types. If the element type `T` of a set type is comparable, then we could use the type `map[T]struct{}` to emulate the set type.

```
package main

type Set map[int]struct{}

func (s Set) Put(x int) {
	s[x] = struct{}{}
}

func (s Set) Has(x int) (r bool) {
	_, r = s[x]
	return
}

func (s Set) Remove(x int) {
	delete(s, x)
}

func main() {
	var s = make(Set)
	s.Put(2)
	s.Put(3)
	println(len(s))   // 2
	println(s.Has(3)) // true
	println(s.Has(5)) // false
	s.Remove(3)
	println(len(s))   // 1
	println(s.Has(3)) // false
}
```

If the element type `T` of a set type is incomparable, we could use a map type `map[*byte]T` to emulate the set type, though the functionalities of the set type is reduced much.

An example:

```
package main

type Set map[*byte]func()

func (s Set) Put(x func()) (remove func()) {
	key := new(byte)
	s[key] = x
	return func() {
		delete(s, key)
	}
}

func main() {
	var s = make(Set)
```

```
	remove1 := s.Put(func(){ println(111) })
	remove2 := s.Put(func(){ println(222) })
	for _, f := range s {
		f()
	}
	println(len(s))   // 2
	remove1()
	println(len(s))   // 1
	remove2()
	println(len(s))   // 0
}
```

The base type of the key (pointer) type must not be a zero-size type, otherwise the pointers created by the `new` function might be not unique.

The set implementation is convenient, but it is actually neither CPU or memory efficient. It could be viewed as a trick. The trick is learned from the Tailscale project.

## 3.16 Map entry iteration order is randomized

Go builtin maps don't maintain entry orders. So when use a `for-range` loop to iterate the entries of a map, the order of the entries is randomized (at least kind-of, depends on specific compilers).

Run the following program several times, we will find the outputs might be different.

```
package main

func main() {
	var m = map[int]int{3:3, 1:1, 2:2}
	for k, v := range m {
		print(k, v)
	}
}
```

But please note that, the print functions in the `fmt` standard package will sort the entries (by their keys) of a map when printing the map. The same happens for `json.Marshal` outputs for maps.

## 3.17 If a map entry is created during iterating the map, the entry may show up during the iteration or may be skipped

For example, the outputs of the following program are not fixed between different runs:

```
package main

var m = map[int]bool{0: true, 1: true}

func main() {
	for k, v := range m {
		m[len(m)] = true
		println(k, v)
```

```
    }
}
```

Some possible outputs:

```
$ go run main.go
0 true
1 true
2 true
3 true
$ go run main.go
0 true
1 true
$ go run main.go
0 true
1 true
2 true
$ go run main.go
1 true
2 true
3 true
4 true
5 true
6 true
7 true
0 true
```

Please note that, as mentioned above, the entry iteration order is randomized (kind of).

## 3.18 The keys in a slice or array composite literal must be constants

For examples, the following code doesn't compile:

```
var m, n = 1, 2
var s = []string{m: "Go"} // error
var a = [3]int{n: 999}    // error
```

The keys in map composite literals have no this limit.

## 3.19 The constant keys in a map/slice/array composite literal must not be duplicate

Go specification clearly specifies that the constant keys in a map/slice/array composite literal must not be duplicate.

For example, all of the following code lines fail to compile for duplicate constant keys:

```
var m = map[string]bool{"Go": true, "Go": false} // error
var s = []string{0: "Go", 0: "C"}                // error
var a = [3]int{2: 999, 2: 555}                   // error
```

Please note that non-constant duplicate keys in map literals lead to unspecified behaviors. For example, it is okay for the following code to print 1, 2 or 3. Any of these print results doesn't violate the Go specification.

```
package main

var a = 1
func main() {
     m := map[int]int{1: 1, a: 2, a: 3}
     println(m[1])
}
```

## 3.20 A compile-time assertion trick by using the fact mentioned in the last section

How to assert a constant boolean expression is true (or false) at compile time? We could make use of the fact introduced in the last section: duplicate constant keys are not allowed in a map composite literal.

For example, the following code assures that the constant boolean expression `aConstantBoolExpr` must be true. If it is not true, then the code fails to compile.

```
var _ = map[bool]int{false: 0, aConstantBoolExpr: 1}
```

For example, the following code asserts the length of a constant string is 32.

```
const S = "abcdefghijklmnopqrstuvwxyz123456"
var _ = map[bool]int{false: 0, len(S)==32: 1}
```

The map element type could be an arbitrary type in this trick.

This trick works for the official standard Go compiler, but not for gccgo (as of version 10.2.1 20210110). There is a bug in gccgo which allows duplicate constant bool keys in a map composite literal.

Source: https://twitter.com/lukechampine/status/1026695476811390976

## 3.21 More compile-time assertion tricks

For the specified assertion example shown in the last section, there are some other ways to assert a constant integer is 32:

```
var _ = [1]int{len(S)-32: 0}
var _ = [1]int{}[len(S)-32]
```

Tricks to assert a constant `N` is not smaller than another constant `M` at compile time:

```
const _ uint = N-M
type _ [N-M]int
```

Tricks to assert a constant string is not blank:

```
var _ = aStringConstant[0]
const _ = 1/len(aStringConstant)
```

Source for the last line: https://groups.google.com/g/golang-nuts/c/w1-JQMaH7c4/m/qzBFSPImBgAJ

## 3.22 The return results of a function may be modified after a `return` statement is executed

Yes, a deferred function call could modify the named return results of its containing function. For example, the following program prints `9` instead of `6`.

```
package main

func triple(n int) (r int) {
	defer func() {
		r += n
	}()

	return n + n
}

func main() {
	println(triple(3)) // 9
}
```

## 3.23 For a deferred function call, its arguments and the called function expression are evaluated when the deferred call is registered

The evaluations are not made when the deferred call is executed later, in the function exiting phase.

For example, the following program prints `1`, not `2` or `3`.

```
package main

func main() {
	var f = func (x int) {
		println(x)
	}
	var n = 1
	defer f(n)
	f = func (x int) {
		println(3)
	}
	n = 2
}
```

The following program doesn't panic. It prints `123`.

```
package main

func main() {
	var f = func () {
		println(123)
	}
	defer f()
```

```
	f = nil
}
```

The following program prints `123`, then panics.

```
package main

func main() {
	var f func () // nil
	defer f()
	println(123)
	f = func () {
	}
}
```

## 3.24 Method receiver arguments are also evaluated at the same time as other arguments

So the receiver argument of a deferred method call is also evaluated when the deferred call is registered. In a method call chain `v.M1().M2()`, the method call `v.M1()` is the receiver argument of the `M2` method call, so the call `v.M1()` will be evaluated (executed) in the deferred call `defer v.M1().M2()`.

For example, the following program prints `132`.

```
package main

type T struct{}

func (t T) M(n int) T {
	print(n)
	return t
}

func main() {
	var t T
	defer t.M(1).M(2)
	t.M(3)
}
```

The following example is more natural.

```
import "sync"

type Counter struct{
	mu sync.Mutex
	n int
}

func (c *Counter) Lock() *Counter  {
	c.mu.Lock()
	return c
}
```

```
func (c *Counter) Unlock() *Counter  {
     c.mu.Unlock()
     return c
}

func (c *Counter) Add(x int) {
     defer c.Lock().Unlock()
     c.n += x
}
```

Similar usages include `defer gl.PushMatrix().PopMatrix()` and `defer tag.Start(...).End()`.

## 3.25 If the left operand of a non-constant bit-shift expression is untyped, then its type is determined as the assumed type of the expression

If either operand of a bit-shift expression is not constant, then the bit-shift expression is a non-constant expression, which result will be evaluated at run time.

Currently (Go 1.17), untyped integers must be constants. So if a bit-shift expression is non-constant and its left operand is untyped, then its right operand must be a non-constant.

The following program prints `002`. The reason is the first two bit-shift expressions are both non-constant, so the respective untyped integer `1`s in them are deduced as values of the assume type, `byte`, so each `1 << n` is evaluated as `0` at run time (because 256 overflows byte values).

On the other hand, the third bit-shift expression is a constant expression, so it is evaluated at compile time. In fact the whole expression `(1 << N) / 128` is evaluated at compile time, as `2`.

```
package main

func main() {
     var n = 8
     var x byte = 1 << n / 128
     print(x) // 0
     var y = byte(1 << n / 128)
     print(y) // 0

     const N = 8
     var z byte = 1 << N / 128
     println(z) // 2
}
```

Why an untyped integer in such situations is not deduced as a value of its default type `int`? This could be explained by using the following example. If the untyped `1` in the following code is deduced as an `int` value instead of an `int64` value, then the bit-shift operation will return different results between 32-bit architectures (`0`) and 64-bit architectures (`0x100000000`), which may produce some silent bugs hard to detect in time.

```
var n = 32
var y = int64(1 << n)
```

The following bit-shift expressions all fail to compile, because the first three untyped integer 1s are both deduced as values of the assume type `float64` and the last one is deduced as a value of the assume type `string`, whereas floating-point and string values may not be shifted.

```
var n = 6
var x float64 = 1 << n  // error
var y = float64(1 << n) // error
var z = 1 << n + 1.0    // error
var w = string(1 << n)  // error
```

The following program prints `0 1`:

```
package main

var n = 8
// The assumed type is byte.
var x = 1 << n >> n + byte(0)
// The assumed type is int16.
var y = 1 << n >> n + int16(0)

func main() {
     println(x, y) // 0 1
}
```

Without an assumed type, the untyped left operand will be deduced as its default type. So the untyped `1` in the following code is deduced as an `int` value. The variable `x` is initialized as `0` on 32-bit architectures (overflows), but as `0x100000000` on 64-bit architectures, which should not be a surprise to a qualified Go programmer.

```
var n = 32
var x = 1 << n // an int value
```

The following code fails to compile, because the untyped `1.0` in the following code is deduced as `float64` value.

```
var n = 6
var y = 1.0 << n // error
```

## 3.26 aConstString[i] and aConstString[i:j] are non-constants even if aConstString, i and j are all constants

For example, the following two lines both fail to compile:

```
const G = "Go"[0]       // error
const Go = "Golang"[:2] // error
```

Whereas the following two lines compile okay:

```
var G = "Go"[0]
var Go = "Golang"[:2]
```

This is a design fault in Go 1.0. It is pity that, for backwards compatibility reasons, the fact is hard to change. Currently, the following program prints `4 0`, because the expression `len(s[:])` is not a constant, whereas the expression `len(s)` is.

```
package main

const s = "Go101.org" // len(s) == 9
var a byte = 1 << len(s) / 128
var b byte = 1 << len(s[:]) / 128

func main() {
	println(a, b) // 4 0
}
```

Source: https://github.com/golang/go/issues/28591

## 3.27 The type deduction rule for a binary operation which operands are both untyped

The Go specification states:

> If the untyped operands of a binary operation (other than a shift) are of different kinds, the result is of the operand's kind that appears later in this list: integer, rune, floating-point, complex.

And the default type of an integer untyped value is `int`, the default type of a rune untyped value is `rune` (a.k.a. `int32`), the default type of a floating-point untyped value is `float64`, and the default type of a complex untyped value is `complex128`.

By the rules, the following program prints `int32`, then `complex128`, and `float64` in the end.

```
package main

import "fmt"

const A = 'A' // 65
const B = 66
const C = 67 + 0i
const One = B - A // 1
const Two = C - A // 2
const Three = B / 22.0

func main() {
	fmt.Printf("%T\n", One)   // int32
	fmt.Printf("%T\n", Two)   // complex128
	fmt.Printf("%T\n", Three) // float64
}
```

The following program prints `01` (on 64-bit architectures), because the kind of the untyped constant `R` is viewed as rune (int32).

```
package main

import "fmt"

const A = '\x61' // a rune literal
const B = 0x62   // default type is int
```

```
const R = B - A  // default type is rune

var n = 32

func main() {
     if R == 1 {
          fmt.Print(R << n >> n) // 0
          fmt.Print(1 << n >> n) // 1
     }
}
```

The following program prints `2 3`.

```
package main

import "fmt"

const X = 3 / 2 * 2
const Y = 3 / 2. * 2
var x, y int = X, Y

func main() {
     fmt.Println(x, y) // 2 3
}
```

## 3.28 An untyped constant integer could overflow its default type

The two constant declarations in the following code are legal.

```
const N = 1 << 200      // default type: int
const R = 'a' + 1 << 31 // default type: rune
```

Typed values may not overflows their respective types. The following two variable and two constant declarations are all illegal.

```
const N int = 1 << 200
const R rune = 'a' + 1 << 31
var x = 1 << 200
var y = 'a' + 1 << 31
```

Whereas these following lines are all legal:

```
const N int = 1 << 200 >> 199
const R rune = 'a' + 1 << 31 - 'b'
var x = 1 << 200 >> 199
var y = 'a' + 1 << 31 - 'b'
```

## 3.29 The placement of the `default` branch (if it exists) in a `switch` code block could be arbitrary

For example, the three `switch` code blocks in the following code are all legal.

```
func foo(n int) {
	switch n {
	case 0: println("n == 0")
	case 1: println("n == 1")
	default: println("n >= 2")
	}

	switch n {
	default: println("n >= 2")
	case 0: println("n == 0")
	case 1: println("n == 1")
	}

	switch n {
	case 0: println("n == 0")
	default: println("n >= 2")
	case 1: println("n == 1")
	}
}
```

The same is for the `default` branch in a `select` code block.

## 3.30 The constant case expressions in a switch code block may be duplicate or not, depending on compilers

Currently, the official standard Go compiler and gccgo compiler both disallow duplicate constant integer case expressions. For example, the following code fails to compile.

```
	switch 123 {
	case 123:
	case 123: // error: duplicate case
	}
```

Both compilers allow duplicate constant boolean case expressions. The following code compilers okay.

```
	switch false {
	case false:
	case false: // okay
	}
```

The official standard Go compiler disallows duplicate constant string case expressions, but gccgo allows.

## 3.31 The switch expression is optional and its default value is a typed value `true` of the builtin type `bool`

For example, the following program prints `True`.

```
package main
```

```
var x, y = false, true

func main() {
     switch {
     case x: println("False")
     case y: println("True")
     }
}
```

But the following code fails to compile, because `MyBool` values, `x` and `y`, may not compare with `bool` values.

```
package main

type MyBool bool
var x, y MyBool = false, true

func main() {
     switch {
     case x: // error
     case y: // error
     }
}
```

## 3.32 Go compilers will automatically insert some semicolons in code

Let's view a small program:

```
package main

func foo() bool {
     return false
}

func main() {
     switch foo()
     {
     case false: println("False")
     case true: println("True")
     }
}
```

What is the output of the above program? Let's think for a while.

~

~

~

`False`? No, it prints `True`. Surprised? Doesn't the function `foo` always return `false`?

Yes, the function `foo` always returns `false`, but this is unrelated here. Compilers will automatically insert some semicolons for the above code as:

```
package main

func foo() bool {
     return false;
};

func main() {
     switch foo();
     {
     case false: println("False");
     case true: println("True");
     };
};
```

Now, it clearly shows that the switch expression (`true`) is omitted. The switch block is actually equivalent to:

```
     switch foo(); true
     {
     case false: println("False");
     case true: println("True");
     };
```

That is why the program prints `True`.

About detailed semicolon insertion rules, please read this article.

## 3.33 What are exactly byte slices (and rune slices)?

There are two interpretations of what are byte slices:

1. slice types which underlying types are `[]byte` are called byte slice types.
2. slice types which element type's underlying type is `byte` are called byte slice types.

The second interpretation is wider than the first one. For example, in the following code, type `Tx` and `Ty` both fit the second interpretation, but only type `Tx` fits the first interpretation.

```
type Tx []byte
type MyByte byte
type Ty []MyByte
```

It looks, currently, the official standard Go compiler (gc) adopts the first interpretation, whereas the gccgo compiler adopts the second interpretation.

In Go, a string may be converted to a byte slice, and vice versa. Whether or not the following code compiles okay depends on which interpretation is adopted. For example, in the following code, the function `foo` compiles okay by using either gc and gccgo, but the function `bar` only compiles okay by using gccgo. It looks the gc compiler sometimes adopts the second interpretation, sometimes adopts the first interpretation.

```
type Tx []byte
type MyByte byte
type Ty []MyByte

var x Tx
```

```
var y Ty
var s = "Go"

func foo() {
	x = Tx(s)
	y = Ty(s)
	s = string(x)
}

func bar() {
	s = string(y) // error (by gc)
}
```

**(Update: this conversion issue has been fixed since Go 1.18. Slice types are formally defined as the slices whose element type's underlying type is `byte`. Please read the above source issue thread for details.)**

Similarly, in the following code, the function `f` compiles okay by using either gc and gccgo, but the function `g` only compiles okay by using gccgo.

```
type Tx []byte
type MyByte byte
type Ty []MyByte

var x = make(Tx, 2)
var y = make(Ty, 2)
var s = "Go"

func f() {
	copy(x, s)
	_ = append(x, s...)
}

func g() {
	copy(y, s)          // error (for gc)
	_ = append(y, s...) // error (for gc)
}
```

The situations are the same for rune slices.

Source: https://github.com/golang/go/issues/23536

## 3.34 Iteration variables are shared between loop steps

In the following code, the two functions, `loop1` and `loop2`, are not equivalent to each other. In `loop1`, the variable `v` is shared between the three steps, whereas in `loop2`, each step declares one new variable `v`. That is why all of the three elements of the result returned by `loop1` have the same value.

```
package main

func loop1(s []int) []*int {
	r := make([]*int, len(s))
	for i, v := range s {
```

```
        r[i] = &v
    }
    return r
}

func loop2(s []int) []*int {
    r := make([]*int, len(s))
    for i := range s {
        v := s[i]
        r[i] = &v
    }
    return r
}

func printAll(s []*int) {
    for i := range s {
        print(*s[i])
    }
    println()
}

func main() {
    var s1 = []int{1, 2, 3}
    printAll( loop1(s1) ) // 333

    var s2 = []int{1, 2, 3}
    printAll( loop2(s2) ) // 123
}
```

For the same reason, the first loop in the following code prints `333`, whereas the second one prints `321`.

```
package main

func main() {
    var s = []int{1, 2, 3}

    // Prints 333
    for _, v := range s {
        defer func() {
            print(v)
        }()
    }

    // Prints 321
    for _, v := range s {
        v := v
        defer func() {
            print(v)
        }()
    }
}
```

## 3.35 `int`, `false`, `nil`, etc. are not keywords

They are predeclared identifiers, which may be shadowed by custom declared identifiers.

For example, the following weird program compiles and runs okay. It prints `false` and `123`.

```
package main

var true = false
const byte = 123
type nil interface{}
func len(nil) int {
	return byte
}

func main() {
	var s = []bool{true, true, true}
	println(s[0])   // false
	println(len(s)) // 123
}
```

## 3.36 Selector colliding

Type embedding is an important feature in Go. Through type embedding, a type could obtain the fields and methods of other types without much effort.

Sometimes, not all of the fields and methods of an embedded type are obtained by the embedding type. The reason is promoted selectors (including fields and methods) might collide with each other.

For example, in the following code, the type `B` embeds one more type (`T2`) than the type `A`. However it obtains none fields and methods. The reason is `B.T1.m` and `B.T2.m` collide with each other so that neither gets promoted. The same situation is for `B.T1.n` and `B.T2.n`.

```
package main

type T1 struct { m bool; n int }
type T2 struct { n int }
func (T2) m() {}

type A struct { T1 }
type B struct { T1; T2 }

func main() {
	var a A
	_ = a.m
	_ = a.n
	var b B
	_ = b.m // error: ambiguous selector
	_ = b.n // error: ambiguous selector
}
```

Please note that, the import path of the containing package of a non-exported selector

(either field or method) is an intrinsic property of the selector. Two unexported selectors with the same name from two different packages will not collide with each other.

For example, in the above example, if the two types `T1` and `T2` are declared in two different packages, then the type `B` will obtain 3 fields and one method.

## 3.37 Each method corresponds a function which first parameter is the receiver parameter of that method

For example, in the following code,

- the type `T` has one method `M1` which corresponds a function `T.M1`.
- the type `*T` has two methods: `M1` and `M2`, which correspond functions `(*T).M1` and `(*T).M2`, respectively.

```
package main

type T struct {
	X int
}

func (t T) M1() int {
	return t.X
}

func (t *T) M2() int {
	return t.X
}

func main() {
	var t = T{X: 3}
	_ = T.M1(t)
	_ = (*T).M1(&t)
	_ = (*T).M2(&t)
}
```

## 3.38 Normalization of method selectors

Go allows simplified forms of some selectors.

For example, in the following program, `t1.M1` is a simplified form of `(*t1).M1`, and `t2.M2` is a simplified form of `(&t2).M2`. At compile time, the compiler will normalize the simplified forms to their original respective full forms.

The following program prints `0` and `9`, because the modification to `t1.X` has no effects on the evaluation result of `*t1` during evaluating `(*t1).M1`.

```
package main

type T struct {
	X int
}
```

```
func (t T) M1() int {
	return t.X
}

func (t *T) M2() int {
	return t.X
}

func main() {
	var t1 = new(T)
	var f1 = t1.M1 // <=> (*t1).M1
	t1.X = 9
	println(f1()) // 0

	var t2 T
	var f2 = t2.M2 // <=> (&t2).M2
	t2.X = 9
	println(f2()) // 9
}
```

In the following code, the function `foo` runs okay, but the function `bar` will produce a panic. The reason is `s.M` is a simplified form of `(*s.T).M`. At compile time, the compiler will normalize the simplified form to it original full form. At runtime, if `s.T` is nil, then the evaluation of `*s.T` will cause a panic. The two modifications to `s.T` have no effects on the evaluation result of `*s.T`.

```
package main

type T struct {
	X int
}

func (t T) M() int {
	return t.X
}

type S struct {
	*T
}

func foo() {
	var s = S{T: new(T)}
	var f = s.M // <=> (*s.T).M
	s.T = nil
	f()
}

func bar() {
	var s S
	var f = s.M // panic
	s.T = new(T)
	f()
```

```
}

func main() {
     foo()
     bar()
}
```

Please note that, interface method values and method values got through reflection will be expanded to the promoted method values with a delay. For example, in the following program, the modification to `s.T.X` has effects on the method values got through reflection and interface ways.

```
package main

import "reflect"

type T struct {
     X int
}

func (t T) M() int {
     return t.X
}

type S struct {
     *T
}

func main() {
     var s = S{T: new(T)}
     var f = s.M // <=> (*s.T).M
     var g = reflect.ValueOf(&s).Elem().
          MethodByName("M").
          Interface().(func() int)
     var h = interface{M() int}(s).M
     s.T.X = 3
     println( f() ) // 0
     println( g() ) // 3
     println( h() ) // 3
}
```

Source: https://github.com/golang/go/issues/47863

## 3.39 The famous := trap

Let's view a simple program.

```
package main

import "fmt"
import "strconv"

func parseInt(s string) (int, error) {
```

```
	n, err := strconv.Atoi(s)
	if err != nil {
		b, err := strconv.ParseBool(s)
		if err != nil {
			return 0, err
		}

		if b {
			n = 1
		}
	}
	return n, err
}

func main() {
	fmt.Println(parseInt("true"))
}
```

We know that the call `strconv.Atoi(s)` will return a non-nil error, but the call `strconv.ParseBool(s)` will return a nil error. Then, will the call `parseInt("true")` return a nil error, too? The answer is it will return a non-nil error.

Wait, isn't the `err` variable is re-declared in the inner code block and its value has been modified to nil before the `parseInt("true")` returns? This is a confusion many new Go programmers, including me, ever encountered when they just started using Go.

The reason why the call `parseInt("true")` returns a non-nil error is a variable declared in an inner code block is never a re-declaration of a variable declared in an outer code block. Here, the inner declared `err` variable is set (initialized) as nil. It is not a re-declaration of the outer declared `err` variable. The outer one is set (initialized) as a non-nil value, then it is never changed later.

There is the voice to remove the `... := ...` re-declaration syntax form from Go. But it looks this is a too big change for Go. Personally, I think explicitly marking the re-declared variables out is a more feasible solution.

## 3.40 The meaning of a `nil` identifier depends on specific context

In Go, the zero values of many kinds of types are represented with the predeclared `nil` identifier, including interface types and some non-interface types (pointers, slices. maps, channels, functions).

A non-interface value could be boxed into an interface value if the type of the former implements the type of the latter. Nil non-interface values are not exceptions.

An interface value boxing nothing is a nil interface value. If it is boxing a nil non-interface value, then it doesn't box nothing, so it is not nil. For example, the following program prints two `false` lines (which is another popular confusion many new Go programmers ever encountered), then prints one `true`.

```
package main

// The return result boxes a nil pointer.
```

```
func box(p *int) interface{} {
	return p
}

func main() {
	// The left nil is interpreted as a nil pointer value.
	// The right nil is interpreted as a nil interface value.
	println(box(nil) == nil) // false
	var x interface{} = nil
	var y chan int = nil
	// y is converted to interface{} before comparing.
	println(x == y) // false

	// This nil is interpreted as a nil channel value.
	println(nil == y) // true
}
```

## 3.41 Some expression evaluation orders are unspecified in Go

In Go, when evaluating the operands of an expression, assignment, or return statement, all function calls (including method calls, and channel communication operations) are evaluated in lexical left-to-right order. The relative order between a non-function operand and function operand is unspecified.

For example, the following program prints two different lines (as of Go toolchain v1.17). In the first assignment (re-declaration) statement, the expression `a` is evaluated after those function calls. But in the second assignment statement (normal variable declaration), the expression `a` is evaluated before those function calls. Neither is wrong. In fact, there is a third possibility: `3 3 6` (if the expression `a` is evaluated between those function calls).

We should not write such unprofessional code in practice.

```
package main

var a int

func f() int {
	a++
	return a
}

func g() int {
	a *= 2
	return a
}

func main() {
	{
		a = 2
		x, y, z := a, f(), g()
		println(x, y, z) // 6 3 6
```

```
	}
	{
		a = 2
		var x, y, z = a, f(), g()
		println(x, y, z) // 2 3 6
	}
}
```

The following is another unprofessional example, in which the `CreateT` call might return a `T` value which `x` field is `53` (gccgo version 10.2.1-6) or `50` (gc version 1.17).

```
package main

import (
	"errors"
	"fmt"
)

type T struct {
	x int
}

func validate(t *T) error {
	if t.x < 0 || t.x > 100 {
		return errors.New("T.x out if range")
	}
	t.x = t.x / 10 * 10
	return nil
}

func CreateT(v int) (T, error) {
	var t = T{x: v}
	return t, validate(&t)
}

func main() {
	var t, _ = CreateT(53)
	fmt.Println(t)
}
```

## 3.42 Go supports loop types

For examples, the following type declarations are all legal.

```
type S []S
type M map[int]M
type F func(F) F
type Ch chan Ch
type P *P
```

The following is an example which uses the last declared type. It compiles and runs both okay.

```
package main
```

```
func main() {
	type P *P
	var pp = new(P)
	*pp = pp
	_ = ************pp
}
```

The following program also compiles and runs both okay.

```
package main

type F func() F

func f() F {
  return f
}

func main() {
  f()()()()()()()()()
}
```

Note, the print functions in the standard `fmt` package don't work well for loop container types. For example, the following program crashes (stack overflow):

```
package main

import "fmt"

func main() {
	type S []S
	var s = make(S, 1)
	s[0] = s
	_ = s[0][0][0][0][0][0][0]
	fmt.Println(s) // panic
}
```

## 3.43 Almost any code element could be declared as the blank identifier _

For example, the following code is legal.

```
const _ = 123
var _ = false
type _ string
func _() {
_: // a label
	return
}
type T struct{
	_ []int
}
func (T) _() {}
```

Package name and interface method names may not be blank identifiers.

## 3.44 Copy slice elements without using the builtin copy function

Since Go 1.17, there is a new way to copy slice elements if the number of the copied elements is known at coding time. The following example shows this way.

```
package main

const N = 128
var x = []int{N-1: 789}

func main() {
     var y = make([]int, N)
     *(*[N]int)(y) = *(*[N]int)(x) // <=> copy(y, x)
     println(y[N-1]) // 789
}
```

# Chapter 4

# Conversions Related

## 4.1 If the underlying type of a defined type is an undefined type, then values of one of the defined types may be implicitly converted to the underlying type, and vice versa

In the following code, the underlying types of the two declared defined types (`Bytes` and `MyBytes`) are both `[]byte`. Values of one of the two defined types may be converted to the other one, but the conversions must be explicit. However, values of two defined types may be implicitly converted to their underlying type, `[]byte`, and vice versa. Because their underlying type is an undefined type.

```
package main

type Bytes []byte
type MyBytes []byte

func f(bs []byte) {}
func g(bs Bytes) {}
func h(bs MyBytes) {}

func main() {
     var x []byte
     var y Bytes
     var z MyBytes

     f(y)
     f(z)

     g(x)
     g(z) // error: cannot use z (type MyBytes) as Bytes
     g(Bytes(z))

     h(x)
     h(y) // error: cannot use y (type Bytes) as MyBytes
```

```
    h(MyBytes(y))
}
```

## 4.2 Values of two different defined pointer types may be indirectly converted to each other's type if the base types of the two types shares the same underlying type

Generally, the values of two pointer types may not be converted to each other if the underlying types of the two pointer types are different. For example, the 4 conversions in the following code are all illegal.

```
package main

type MyInt int
type IntPtr *int     // underlying type is *int
type MyIntPtr *MyInt // underlying type is *MyInt

func main() {
    var x IntPtr
    var y MyIntPtr
    x = IntPtr(y)       // error
    y = MyIntPtr(x)     // error
    var _ = (*int)(y)   // error
    var _ = (*MyInt)(x) // error
}
```

Although the above 4 conversions may not achieved directly, they may be achieved indirectly. This benefits from the fact that the following conversions are legal.

```
package main

type MyInt int

func main() {
    var x *int
    var y *MyInt
    x = (*int)(y)   // okay
    y = (*MyInt)(x) // okay
}
```

The reason why the above two conversions are legal is values of two non-defined (unnamed) pointers types may be converted to each other's type if the base types of the two types shares the same underlying type. In the above example, the base types of the types of `x` and `y` are `int` and `MyInt`, which share the same underlying type `int`, so `x` and `y` may be converted to each other's type.

Benefiting from the just mentioned fact, values of `IntPtr` and `MyIntPtr` may be also converted to each other's type, though such conversions must be indirectly, as shown in the following code.

```
package main
```

```
type MyInt int
type IntPtr *int
type MyIntPtr *MyInt

func main() {
     var x IntPtr
     var y MyIntPtr
     x = IntPtr((*int)((*MyInt)(y)))      // okay
     y = MyIntPtr(((*MyInt))((*int)(x))) // okay
     var _ = (*int)((*MyInt)(y))          // okay
     var _ = (*MyInt)((*int)(x))          // okay
}
```

## 4.3 Values of a defined bidirectional channel type may not be converted to a defined unidirectional channel type with the same element type directly, but may indirectly

An example:

```
package main

func main() {
     type C chan string
     type Cw chan<- string
     type Cr <-chan string

     var c C
     var w Cw
     var r Cr

     // The following two lines fail to compile.
     // w = Cw(c) // error
     // r = Cr(c) // error

     // This line compiles okay.
     _ = (chan string)(c)

     // The two lines also compile okay.
     w = Cw((chan string)(c)) // indirectly
     r = Cr((chan string)(c)) // indirectly

     _, _ = w, r
}
```

Such conversions are rarely used in practice, but knowing more is not a bad thing, right?

## 4.4 The capacity of the result of a conversion from string to byte slice is unspecified

The implementation of the `addPrefixes` function in the following code is unprofessional.

```
package main

func addPrefixes(prefixStr string, bss [][]byte) {
	var prefix = []byte(prefixStr)
	println(len(prefix), cap(prefix))
	for i, bs := range bss {
		bss[i] = append(prefix, bs...)
	}
}

func main() {
	var bss = [][]byte {
		[]byte("Java"),
		[]byte("C++"),
		[]byte("Go"),
		[]byte("C"),
	}
	addPrefixes("> ", bss)
	println(string(bss[0])) // > Co+a
	println(string(bss[1])) // > Co+
	println(string(bss[2])) // > Co
	println(string(bss[3])) // > C
}
```

The outputs of the above program:

```
2 8
> Co+a
> Co+
> Co
> C
```

The outputs are not what we expect. Why? Because the capacity of the result of the conversion `[]byte("> ")` is 8, which is actually compiler dependent. In the end, all of the elements of `bss` share some leading bytes with the conversion result. Each `append` call overwrite some bytes in the conversion result.

To fix the problem, we should clip the conversion result, so that the elements of `bss` doesn't share bytes. The fixed `addPrefixes` function implementation:

```
func addPrefixes(prefixStr string, bss [][]byte) {
	var prefix = []byte(prefixStr)
	prefix = prefix[:len(prefix):len(prefix)] // clip it
	for i, bs := range bss {
		bss[i] = append(prefix, bs...)
	}
}
```

Then the outputs will become as expected:

> Java
> C++
> Go
> C

# Chapter 5

# Comparisons Related

## 5.1 Compare two slices which lengths are equal and known at coding time

In Go, slices are incomparable. But if the elements of two slices are comparable, then there is a way to compare the two slices (since Go version 1.17). Assume the elements of the two slices are identical and their lengths are equal and known at coding time, then we could use the following way to compare the two slices.

```
package main

func main() {
	var x = []int{1, 2, 3, 4, 5}
	var y = []int{1, 2, 3, 4, 5}
	var z = []int{1, 2, 3, 4, 9}

	// The following two lines fail to compile.
	// _ = x == y
	// _ = x == z

	// The two lines compile okay.
	println(*(*[5]int)(x) == *(*[5]int)(y)) // true
	println(*(*[5]int)(x) == *(*[5]int)(z)) // false
}
```

## 5.2 More ways to compare byte slices

The above introduced way works for slices with any comparable element types. It certainly could be used to compare byte slices (which lengths are equal and known at coding time). Meanwhile, there are two other ways to compare byte slices `x` and `y`, even if the lengths of the two byte slices are not known at compile time.

- The first way: `bytes.Compare(x, y) == 0`.
- The second way: `string(x) == string(y)`. Due to an optimization made by the official standard Go compiler, no underlying bytes will be duplicated in this way. In fact, the `bytes.Equal` function uses this way to do the comparison.

The two ways have no requirements on the lengths of the two operand byte slices.

Go 1.18 introduces custom generics, so there might be a `slices.Compare` function in future Go versions to compare slices with any comparable element types.

## 5.3 Comparing two interface values produces a panic if the dynamic type of the two operands are identical and the identical type is an incomparable type

For example, the following program prints three `false`, then panics.

```
package main

func main() {
	var x interface{} = []int{1, 2}
	var y interface{} = map[string]int{}
	var z interface{} = func() {}

	// The lines all print false.
	println(x == y)
	println(x == z)
	println(x == nil)

	// Each of these line could produce a panic.
	println(x == x)
	println(y == y)
	println(z == z)
}
```

## 5.4 How to make a struct type incomparable

It is easy, just put an incomparable field in the struct type. For example, the following struct types are all incomparable.

```
type T1 struct {
	_ func()
	x int
}

type T2 struct {
	_ []int
	y bool
}

type T3 struct {
	_ map[int]bool
	z string
}
```

Lest the `_` fields waste memory, their types should be zero-size types. For example, the size of the type `Ty` is smaller than the type `Tx` in the following code.

```
package main

import "unsafe"

type Tx struct {
	_ func()
	x int64
}

type Ty struct {
	_ [0]func()
	y int64
}

func main() {
	var x Tx
	var y Ty
	println(unsafe.Sizeof(x)) // 16
	println(unsafe.Sizeof(y)) // 8
}
```

Please try to avoid putting a zero-size field as the final field of a struct type.

## 5.5 Array values are compared element by element

When comparing two array values, their elements will be compared one by one. Once two corresponding elements are found unequal, the whole comparison stops and a false result is resulted. The whole comparison might also stop for a panic produced when comparing two interfaces.

For example, the first comparison in the following code results false, but the second one causes a panic.

```
package main

type T [2]interface{}

func main() {
	var a = T{1, func(){}}
	var b = T{2, func(){}}
	println(a == b) // false

	var c = T{2, func(){}}
	var d = T{2, func(){}}
	println(c == d) // panics
}
```

## 5.6 Struct values are compared field by field

Similarly, when comparing two struct values, their fields will be compared one by one. Once two corresponding fields are found unequal, the whole comparison stops and a false result

is resulted. The whole comparison might also stop for a panic produced in comparing two interfaces.

For example, the first comparison in the following code results false, but the second one causes a panic.

```
package main

type T struct {
	x interface{}
	y interface{}
}

func main() {
	var a = T{x: 1, y: func(){}}
	var b = T{x: 2, y: func(){}}
	println(a == b) // false

	var c = T{x: 2, y: func(){}}
	var d = T{x: 2, y: func(){}}
	println(c == d) // panics
}
```

## 5.7 The _ fields in struct comparisons are ignored

For example, the following program prints `true`.

```
package main

type T struct {
	_ int
	x string
}

func main() {
	var x = T{123, "Go"}
	var y = T{789, "Go"}
	println(x == y) // true
}
```

But please note that, as shown in a previous section, if a struct type contains a `_` field of an incomparable type, then the struct type is also incomparable.

## 5.8 NaN != NaN, Inf == Inf

In floating-point computations, there are some cases in which the computation results might be infinities (Inf) or not-a-number (NaN). For example, in the following code, a `+Inf` and a `NaN` values are produced (yes, an `Inf` value times zero results a `NaN` value).

Every two `+Inf` (or `-Inf`) values are equal to each other, but every two `NaN` values are not equal.

```
package main
```

```
var a = 0.0
var x = 1 / a // +Inf
var y = x * a // NaN

func main() {
     println(x, y)   // +Inf NaN
     println(x == x) // true
     println(y == y) // false
}
```

As `NaN` values are not equal to each other, it is always a vain to loop up an entry from a map by using a `NaN` key, which could be proved from the following code.

```
package main

var a = 0.0
var x = 1 / a // +Inf
var y = x * a // NaN

func main() {
     var m = map[float64]int{}
     m[y] = 123
     m[y] = 456
     m[y] = 789
     q, ok := m[y]
     println(q, ok, len(m)) // 0 false 3
}
```

In fact, comparing a `NaN` value with any value will result `false`:

```
package main

var a = 0.0
var y = 1 / a * a // NaN

func main() {
     println(y < y)  // false
     println(y == y) // false
     println(y > y)  // false

     println(y < a)  // false
     println(y == a) // false
     println(y > a)  // false
}
```

So it looks putting an entry with a `NaN` key into a map is like putting the entry into black hole, though `NaN` keys could be retrieved from a `for-range` loop:

```
package main

var a = 0.0
var y = 1 / a * a // NaN

func main() {
     var m = map[float64]int{}
```

```
	m[y] = 1
	m[y] = 2
	m[y] = 3
	for k, v := range m {
		println(k, v)
	}
}
```

The (possible) outputs of the above program:

```
NaN 1
NaN 2
NaN 3
```

A map entry which key is `NaN` is unable to be deleted for its containing map.

## 5.9 Some details in using the `reflect.DeepEqual` function

A call to the `reflect.DeepEqual` function always return `false` if the types of its two arguments are different.

When using the `reflect.DeepEqual` function to compare two different pointer values (of the same type), the values referenced by them are compared instead (still using the `reflect.DeepEqual` function to do the deeper comparison).

If both of the two arguments of a `reflect.DeepEqual` function call are in cyclic reference chains, then, to avoid infinite looping, the call might return `true`. An example:

```
package main

import "reflect"

type Node struct{peer *Node}

func main() {
	var x, y, z Node
	x.peer = &x // form a cyclic reference chain
	y.peer = &z // form a cyclic reference chain
	z.peer = &y
	println(reflect.DeepEqual(&x, &y)) // true
}
```

When using the `reflect.DeepEqual` function to compare two function values, the return result is `true` only if the two functions share the identical type and they are both nil. For example, the following program prints `true` then `false`.

```
package main

import "reflect"

func main() {
	var x, y func()
	println(reflect.DeepEqual(x, y)) // true
	var z = func() {}
```

```
	println(reflect.DeepEqual(z, z)) // false
}
```

When using the `reflect.DeepEqual` function to compare two slice values (of the same type and with the same length), generally, their elements will be compared one by one. However, if their corresponding first elements have the same address, then `true` is returned without comparing their elements, even if their elements are self-unequal values (for example, non-nil functions and NaNs).

For example, the following program also prints `true` then `false`.

```
package main

import "reflect"

func main() {
	var f = func() {}
	var a = [2]func(){f, f}
	var x = a[:]
	var y = a[:]
	var z = []func(){f, f}
	println(reflect.DeepEqual(x, y)) // true
	println(reflect.DeepEqual(x, z)) // false
}
```

Similarly, if two map values are referencing the same underlying hashtable, the result is also `true` if they are compared with the `reflect.DeepEqual` function, even if the hashtable contains self-unequal values.

```
package main

import (
	"math"
	"reflect"
)

func main() {
	nan := math.NaN()
	println(reflect.DeepEqual(nan, nan)) // false

	m1 := map[int]float64{1: nan}
	m2 := map[int]float64{1: nan}
	m3 := m1

	println(reflect.DeepEqual(m1, m1)) // true
	println(reflect.DeepEqual(m1, m2)) // false
	println(reflect.DeepEqual(m3, m3)) // true
}
```

## 5.10 The return results of the `bytes.Equal` and `reflect.DeepEqual` functions might be different

The `reflect.DeepEqual` function thinks a nil slice and a blank slice are not equal. However, the `bytes.Equal` function thinks a nil byte slice and a blank byte slice are equal. This could be proved from the following program.

```
package main

import (
	"bytes"
	"reflect"
)

func main() {
	var x = []byte{}
	var y []byte
	println(bytes.Equal(x, y))       // true
	println(reflect.DeepEqual(x, y)) // false
}
```

## 5.11 A type alias embedding bug

Go 1.9 introduced custom type alias declarations. However, a bug has also been introduced since then. Up to now (Go 1.17), this bug has not been fixed yet.

The bug could be exposed by the following program. It should print `false`, but it prints `true` now.

```
package main

type Int = int

type A = struct{ int }
type B = struct{ Int }

func main() {
	var x, y interface{} = A{}, B{}
	println(x == y) // true
}
```

Source: https://github.com/golang/go/issues/24721

[Update]: this bug has been fixed in Go 1.18.

# Chapter 6

# Runtime Related

## 6.1 In the official standard compiler implementation, the backing array of a map never shrinks

The official standard runtime maintains a backing array for a map to hold the entries of the map. With more and more entries are put into the map, the backing array will grow and grow, but it will never shrink. That means if a map even contained millions of entries, then after these entries are all deleted from the map, the backing array is still capable of holding millions of entries without growing its backing array.

Then how to release the memory occupied by the backing array of a map? Just set the map value as nil, or create a new map and assign the new map to it.

## 6.2 64-bit word alignment problem

64-bit atomic operations on a 64-bit integer require the address of the 64-bit integer must be 8-byte aligned in memory. On 64-bit architectures, 64-bit integers are always 8-byte aligned, so the requirement is always satisfied on 64-bit architectures. This is not always true on 32-bit architectures.

The docs of the `sync/atomic` standard package states that a qualified Go compiler should make sure that the first (64-bit) word (think it as an `int64` or `uint64` integer) in a variable or in an allocated struct, array, or slice can be relied upon to be 64-bit aligned. What does the word `allocated` mean? We can think an allocated value as a declared variable, a value returned the built-in `make` function, or the value referenced by a value returned by the built-in `new` function.

In the following example, the first `AddX` method call is safe, because `t.x` is always 8-byte aligned, even on 32-bit architectures. However, the second `AddX` method call is not safe on 32-bit architectures. It might cause a panic, because `s.t.x` is not guaranteed to be 8-byte aligned.

```
package main

import "sync/atomic"

type T struct {
```

```
	x uint64
}

func (t *T) AddX(dx uint64) {
	atomic.AddUint64(&t.x, dx)
}

type S struct {
	y int32
	t T
}

func main() {
	var t T
	t.AddX(1) // safe, even on 32-bit architectures

	var s S
	s.t.AddX(1) // might panic on 32-bit architectures
}
```

One fact we should be aware of is that the official Go compilers (gc and gccgo) guarantee that 32-bit and 64-bit words are always 4-byte aligned on any architectures. In fact, the ever implementation of the `sync.WaitGroup` type relied upon this fact.

The `sync.WaitGroup` type needs two fields. Normally, it should be defined as

```
type WaitGroup struct {
   state uint64
   sema  uint32
}
```

Here, the `state` need to participate 64-bit atomic operations. However, on 32-bit architectures, its address is not guaranteed to be 8-byte aligned. So instead, the `sync.WaitGroup` type was ever defined as

```
type WaitGroup struct {
	state1 [3]uint32
}
```

At runtime, the `state1` field of a `sync.WaitGroup` value might 4-byte aligned or 8-byte aligned. If it is 8-byte aligned, the combination of the first two elements of the `state1` field is viewed as the original `state` field and the third element is viewed as the original `sema` field; otherwise, the combination of the last two elements of the `state1` field is viewed as the original `state` field and the first element is viewed as the original `sema` field.

## 6.3 Let `go vet` detect copying values of a type

The official `go vet` command will warn on copying values of types which values should not be copied, such as the types in the `sync` standard package. We call such types as noCopy types here.

Currently, there is no special syntax for this purpose. The `go vet` command determines whether or not a type `T` is a noCopy type by checking whether or not the pointer type `*T` has a `Lock()` method and an `Unlock()` method.

For example, The `go vet` command will report a warning for the assignment in the following code.

```
package main

type T struct{}

func (*T) Lock() {}
func (*T) Unlock() {}

func main() {
	var t T
	_ = t // warning: assignment copies lock value to _
}
```

A struct type with a noCopy field (embedding or not) or an array type with noCopy elements is also a noCopy type. For example:

```
package main

type T struct{}

func (*T) Lock() {}
func (*T) Unlock() {}

type S struct {
	t T
}

func main() {
	var s S
	_ = s // warning: assignment copies lock value to _

	var a [8]T
	_ = a // warning: assignment copies lock value to _
}
```

## 6.4 Values of some types in the standard packages should not be copied

Besides the types in the `sync` standard package, values of some other types in the standard packages should not be copied too, such as `bytes.Buffer` and `strings.Builder`.

Generally, if a value is referencing some other values, and these referenced values should not be referenced by multiple values, then the referencing value should not be copied.

## 6.5 Some zero values might contain non-zero bytes in memory

An example:

```
package main

import (
	"fmt"
	"reflect"
	u "unsafe"
)

var s = "abc"[0:0]

func main() {
	header := (*reflect.StringHeader)(u.Pointer(&s))
	if s == "" {
		fmt.Printf("%#v\n", *header)
	}
}
```

The `reflect.StringHeader` type represents the internal structure of the `string` type.

Run the program, the outputs are like:

```
reflect.StringHeader{Data:0x4957ec, Len:0}
```

From the outputs, we could find that the `Data` field of the zero string `s` are non-zero, which doesn't prevent the runtime from thinking the string `s` is a zero value. In fact, the zero length is sufficient to indicate the string `s` is a blank string.

## 6.6 The address of a value might change at run time

In the official standard Go runtime implementation, the stack of a goroutine will grow or shrink as needed at run time. The address of a value allocated on a stack will change when the stack size changes.

For example, the following program very probably prints two different addresses.

```
package main

func f(i int) byte {
	type T int // avoid being inlined
	var a [1 << 12]byte
	return a[i]
}

func main() {
	var x int
	println(&x)
	f(100) // make stack grow
	println(&x)
}
```

## 6.7 The official standard Go runtime behaves badly when system memory is exhausted

When system memory is exhausted and memory swapping is involved, the Go runtime often doesn't crash program but exhausts almost all CPU resources so that the OS UI is often totally not responsive. An hard restart is often needed to escape such awkward situations.

For example, sometimes, during the phase of debugging a program, if we accidentally write a piece of code like the following shows, then the OS might hang when running a program which uses the piece of code.

*(Warning: please save your works if you would like to run this program on you machine!)*

```
package main

var s = "1234567890"

func condition() bool {
	return true // simplified for demo purpose
}

func main() {
	for condition() {
		s += s
		println(len(s))
	}
}
```

It is a good idea to limit the number of loop steps to a reasonable number in debugging.

```
func main() {
	for range [10]struct{}{} {
		if condition() {
			break
		}

		s += s
		println(len(s))
	}
}
```

## 6.8 Currently, a `runtime.Goexit` call unexpectedly cancels the already happened panics

In other words, a `runtime.Goexit` call acts as a `recover` call. This should be a bug in the current official standard Go compiler (version 1.17).

For example, the following program should crash for the unrecovered panic in the `worker` goroutine, but it exits normally. Future versions of the official standard Go compiler might change the behavior.

```
package main
```

```
import "runtime"

func wroker(c chan int) {
	defer close(c)
	defer runtime.Goexit()

	// ... do work load

	panic("bye")
}

func main() {
	c := make(chan int)
	go wroker(c)
	<-c
}
```

Source: https://github.com/golang/go/issues/35378

## 6.9 There might be multiple panics coexisting in a goroutine

Two coexisting panics must stay at two different function call depths, and a newer panic must stay at a deeper function call. When a deeper panic spreads to a shallower function call and there is another panic staying there, then the deeper panic will replace the shallower panic.

For example, for the following program, at a time when it is running, there will be two active panics coexisting. At a later time, the second panic replaces the first one and is recovered finally.

```
package main

func main() {
	defer func() {
		println("Panic", recover().(int), "is recovered.")
	}()
	defer println("Now, panic 2 replaces panic 1.")
	defer func() {
		defer println("Now, 2 panics coexist.")
		panic(2)
	}()
	defer println("Only one panic exists now.")
	panic(1)
}
```

The outputs of the above program:

```
Only one panic exists now.
Now, 2 panics coexist.
Now, panic 2 replaces panic 1.
Panic 2 is recovered.
```

## 6.10 The current Go specification (version 1.17) doesn't explain the panic/recover mechanism very well

By the current specification, the line marked as "no-op" in the following code should recover the panic 1, but it doesn't actually. The reason is only the latest produced panic could be recovered.

```
package main

import "fmt"

func main() {
     defer func() {
          fmt.Print(recover())
     }()
     defer func() {
          defer func() {
               fmt.Print(recover())
          }()
          defer recover() // no-op
          panic(2)
     }()
     panic(1)
}
```

The above program prints `21`. If we change the "no-op" line to a non-deferred call, then `2<nil>` will be printed instead.

# Chapter 7

# Standard Packages Related

## 7.1 Use `%w` format verb in `fmt.Errorf` calls to build error chains

When using the `fmt.Errorf` function to wrap a deeper error, it is recommended to use the `%w` verb instead of `%s`, to avoid losing information of the wrapped error.

For example, in the following code, the `Bar` implementation is preferred to the `Foo` implementation, because the caller could judge whether or not the returned error is caused by the specified error (here it is `ErrNotImpl`).

```
package main

import (
     "errors"
     "fmt"
)

var ErrNotImpl = errors.New("not implemented yet")

func doSomething() error {
     return ErrNotImpl
}

func Foo() error {
     if err := doSomething(); err != nil {
          return fmt.Errorf("Foo: %s", err)
     }
     return nil
}

func Bar() error {
     if err := doSomething(); err != nil {
          return fmt.Errorf("Bar: %w", err)
     }
     return nil
}
```

```
func main() {
    println(errors.Is(Foo(), ErrNotImpl)) // false
    println(errors.Is(Bar(), ErrNotImpl)) // true
}
```

In user code, we should try to use the `errors.Is` function instead of using direct comparisons to judge the cause of an error.

## 7.2 Small differences between `fmt.Println`, `fmt.Print` and `print` functions

The `fmt.Println` function (and `println`) will output a space between any two adjacent arguments. The `fmt.Print` function will only do this between two adjacent arguments which are both not strings. The `print` function never output spaces between arguments.

This could be proved by the following code.

```
package main

import "fmt"

func main() {
    // 123 789 abc xyz
    println(123, 789, "abc", "xyz")
    // 123 789 abc xyz
    fmt.Println(123, 789, "abc", "xyz")
    // 123 789abcxyz
    fmt.Print(123, 789, "abc", "xyz")
    println()
    // 123789abcxyz
    print(123, 789, "abc", "xyz")
    println()
}
```

## 7.3 The `reflect.Type/Value.NumMethod` methods will count unexported methods for interfaces

For non-interface types and values, the `reflect.Type.NumMethod` and `reflect.Value.NumMethod` methods don't count unexported methods. But for interface types and values, they count.

This could be proved by the following code.

```
package main

import "reflect"

type I interface {
    m()
    M()
}
```

```
type T struct {}
func (T) m() {}
func (T) M() {}

func main() {
     var t T
     var i I = t
     var vt = reflect.ValueOf(t)
     var vi = reflect.ValueOf(&i).Elem()
     println(vt.NumMethod()) // 1
     println(vi.NumMethod()) // 2
}
```

## 7.4 Values of two slices may not be converted to each other's type if the element types of the two slices are different, but there is a hole to this rule

For example, the two conversions in the following code are both illegal.

```
package main

type MyByte byte
var x []MyByte
var y []byte

func main() {
     x = []MyByte(y) // error
     y = []byte(x)   // error
}
```

There is a hole to this rule. If the underlying type of the element type of a slice is `byte` (such as the `MyByte` type shown in the above example), then we could use the `reflect.Value.Bytes` methods to convert the slice to `[]byte`. For example:

```
package main

import "reflect"

type MyByte byte
Value.Bytes
func main() {
     var x = make([]MyByte, 128)
     var y []byte
     y = reflect.ValueOf(x).Bytes()
     y[127] = 123
     println(x[127]) // 123
}
```

Source: https://github.com/golang/go/issues/24746

## 7.5 Don't misuse the `TrimLeft` function as `TrimPrefix` in the `strings` and `bytes` standard packages

The second parameter of the `TrimLeft` function is a cutset, any leading Unicode code points in the first parameter contained in the cutset will be removed, which is quite different from the `TrimPrefix` function.

The following program shows the differences.

```
package main

import "strings"

func main() {
	var hw = "DoDoDo!"
	println(strings.TrimLeft(hw, "Do"))   // !
	println(strings.TrimPrefix(hw, "Do")) // DoDo!
}
```

The same situation is for the `TrimRight` and `TrimSuffix` functions.

## 7.6 The `json.Unmarshal` function accepts case-insensitive object key matches

For example, the following program prints `bar`, instead of `foo`.

```
package main

import (
	"encoding/json"
	"fmt"
)

	"encoding/json"
	HTML string `json:"HTML"`
}

var s = `{"HTML": "foo", "html": "bar"}`

func main() {
	var t T
	if err := json.Unmarshal([]byte(s), &t); err != nil {
		fmt.Println(err)
		return
	}
	fmt.Println(t.HTML) // bar
}
```

The docs of the `json.Unmarshal` function states "preferring an exact match but also accepting a case-insensitive match". So personally, I think this is a bug in the `json.Unmarshal` function implementation, but the Go core team don't think so.

## 7.7 The spaces in struct tag key-value pairs will not be trimmed

For example, the following program will print `{" foo":""}`. The misspelt `omitempty` option for the `Foo` field is different from `omitempty`, and the tag key of the `Foo` field is `" foo"`, instead of `"foo"`.

```
package main

import (
	"encoding/json"
	"fmt"
)

type T struct {
	Foo string `json:" foo, omitempty"`
	Bar string `json:"bar,omitempty"`
}

func main() {
	var t T
	var s, _ = json.Marshal(t)
	fmt.Printf("%s", s) // {" foo":""}
}
```

## 7.8 How to try to run a custom `init` function as early as possible?

Assume the module of the project is `x.y/app`, add an `x.y/app/internal/init` package, and put an `init` function in the `init` package, then import the `init` package in the `main` package. The `init` package will be loaded after some core packages (such as `runtime` standard package), but before other packages.

## 7.9 How to resolve cyclic package dependency problem?

Go doesn't support cyclic package dependencies. If package `foo` imports package `bar`, then package `bar` may not import package `foo`.

Sometimes, we might encounter the situation that two packages do need to use the exported identifiers from each other. How should we handle such situations? There are two ways to solve this problem.

One way is to merge the two packages into one bigger package, so that the cyclic dependency problem will go. This way will always work.

The other way is to split the two packages into more smaller ones to remove the cyclic dependency relations. Sometimes, this is impossible to achieve.

## 7.10 Deferred calls will not be executed after the `os.Exit` function is called

For example, the deferred call `cleanup()` in the following code is totally useless.

```
func run() {
     defer cleanup()

     if err := doSomething(); err != nil {
          log.Println(err)
          os.Exit(1)
     }

     os.Exit(0)
}
```

Please note that the `log.Fatal` function calls the `os.Exit` function, so deferred calls will also not get executed after the `log.Fatal` function is called.

## 7.11 How to let the `main` function return an exit code?

No way to do this. Go syntax doesn't support this. However, we can use the following way to simulate a main function which returns an exit code.

```
import "os"

func main() {
    os.Exit(realMain())
}

func realMain() int {
    ... // do something, return non-zero on errors

    return 0
}
```

## 7.12 Try not to export errors as variables

We should try to avoid exporting variables (in particular `error` values) from the packages we are maintaining. The standard packages contain many exported `error` variable values, which is actually a bad practice. Personally, I recommend to use the following way to declare `error` values.

```
package foo

type errType int

const (
     ErrA errType = iota
     ErrB
     ErrC
     errCount
```

```
)

func (e errType) Error() string {
     if e < 0 || e >= errCount {
          panic("invalid error number")
     }
     return errDescriptions[e]
}

var _ = [1]int{}[len(errDescriptions) - int(errCount)]

var errDescriptions = [...]string {
     ErrA: "error A",
     ErrB: "error B",
     ErrC: "error C",
}
```

By using this way, users of the `foo` package couldn't modify the declared error values.