Kyle Quest | golang

50 Shades of Go: Traps, Gotchas, and Common Mistakes for New Golang Devs

50 Shades of Go in Other Languages

- Chinese Translation: blog post, segmentfault (by wuYin) – needs updates
- Another Chinese Translation: blog post (by Shadowwind LEY) – needs updates
- Russian Translation: blog post (by Ilia Ozhereliev, Mail.Ru Group Blog) – needs updates

Overview

Go is a simple and fun language, but, like any other language, it has a few gotchas... Many of those gotchas are not entirely Go's fault. Some of these mistakes are natural traps if you are coming from another language. Others are due to faulty assumptions and missing details.

A lot of these gotchas may seem obvious if you took the time to learn the language reading the official spec, wiki, mailing list discussions, many great posts and presentations by Rob Pike, and the source code. Not everybody starts the same way though and that's OK. If you are new to Go the information here will save you hours debugging your code.

Total Beginner:

- Opening Brace Can't Be Placed on a Separate Line
- Unused Variables
- Unused Imports
- Short Variable Declarations Can Be Used Only Inside Functions
- Redeclaring Variables Using Short Variable Declarations
- Can't Use Short Variable Declarations to Set Field Values
- Accidental Variable Shadowing
- Can't Use "nil" to Initialize a Variable Without an Explicit Type
- Using "nil" Slices and Maps
- Map Capacity
- Strings Can't Be "nil"
- Array Function Arguments
- Unexpected Values in Slice and Array "range" Clauses
- Slices and Arrays Are One-Dimensional
- Accessing Non-Existing Map Keys
- Strings Are Immutable
- Conversions Between Strings and Byte Slices
- Strings and Index Operator
- Strings Are Not Always UTF8 Text
- String Length
- Missing Comma In Multi-Line Slice/Array/Map Literals
- log.Fatal and log.Panic Do More Than Log

- Built-in Data Structure Operations Are Not Synchronized
- Iteration Values For Strings in "range" Clauses
- Iterating Through a Map Using a "for range" Clause
- Fallthrough Behavior in "switch" Statements
- Increments and Decrements
- Bitwise NOT Operator
- Operator Precedence Differences
- Unexported Structure Fields Are Not Encoded
- App Exits With Active Goroutines
- Sending to an Unbuffered Channel Returns As Soon As the Target Receiver Is Ready
- Sending to an Closed Channel Causes a Panic
- Using "nil" Channels
- Methods with Value Receivers Can't Change the Original Value

Intermediate Beginner:

- Closing HTTP Response Body
- Closing HTTP Connections
- JSON Encoder Adds a Newline Character
- JSON Package Escapes Special HTML Characters in Keys and String Values
- Unmarshalling JSON Numbers into Interface Values
- JSON String Values Will Not Be Ok with Hex or Other non-UTF8 Escape Sequences
- Comparing Structs, Arrays, Slices, and Maps
- Recovering From a Panic
- Updating and Referencing Item Values in Slice, Array, and Map "for range" Clauses
- "Hidden" Data in Slices
- Slice Data Corruption
- "Stale" Slices
- Type Declarations and Methods
- Breaking Out of "for switch" and "for select" Code Blocks
- Iteration Variables and Closures in "for" Statements
- Deferred Function Call Argument Evaluation
- Deferred Function Call Execution
- Failed Type Assertions
- Blocked Goroutines and Resource Leaks
- Same Address for Different Zero-sized Variables
- The First Use of iota Doesn't Always Start with Zero

Advanced Beginner:

- Using Pointer Receiver Methods On Value Instances
- Updating Map Value Fields
- "nil" Interfaces and "nil" Interfaces Values
- Stack and Heap Variables
- GOMAXPROCS, Concurrency, and Parallelism
- Read and Write Operation Reordering
- Preemptive Scheduling

Cgo (aka Brave Beginner):

- Import C and Multiline Import Blocks
- No blank lines Between Import C and Cgo Comments

- Can't Call C Functions with Variable Arguments

Traps, Gotchas, and Common Mistakes

Opening Brace Can't Be Placed on a Separate Line

- level: beginner

In most other languages that use braces you get to choose where you place them. Go is different. You can thank automatic semicolon injection (without lookahead) for this behavior. Yes, Go does have semicolons :-)

Fails:

```
package main

import "fmt"

func main()
{ //error, can't have the opening brace on a separate line
    fmt.Println("hello there!")
}
```

Compile Error:

> */tmp/sandbox826898458/main.go:6: syntax error: unexpected semicolon or newline before {*

Works:

```
package main

import "fmt"

func main() {
    fmt.Println("works!")
}
```

Unused Variables

- level: beginner

If you have an unused variable your code will fail to compile. There's an exception though. You must use variables you declare inside functions, but it's OK if you have unused global variables. It's also OK to have unused function arguments.

If you assign a new value to the unused variable your code will still fail to compile. You need to use the variable value somehow to make the compiler happy.

Fails:

```
package main

var gvar int //not an error

func main() {
    var one int   //error, unused variable
    two := 2      //error, unused variable
    var three int //error, even though it's assigned 3 on the next line
    three = 3

    func(unused string) {
        fmt.Println("Unused arg. No compile error")
    }("what?")
}
```

Compile Errors:

> */tmp/sandbox473116179/main.go:6: one declared and not used /tmp/sandbox473116179/main.go:7: two declared and not used /tmp/sandbox473116179/main.go:8: three declared and not used*

Works:

```
package main

import "fmt"

func main() {
    var one int
    _ = one

    two := 2
    fmt.Println(two)

    var three int
    three = 3
    one = three

    var four int
    four = four
}
```

Another option is to comment out or remove the unused variables :-)

Unused Imports

- level: beginner

Your code will fail to compile if you import a package without using any of its exported functions, interfaces, structures, or variables.

If you really need the imported package you can use the blank identifier, `_`, as its package name to avoid this compilation failure. The blank identifier is used to import packages for their side effects.

Fails:

```
package main

import (
    "fmt"
    "log"
    "time"
)

func main() {
}
```

Compile Errors:

> */tmp/sandbox627475386/main.go:4: imported and not used: "fmt" /tmp/sandbox627475386/main.go:5: imported and not used: "log" /tmp/sandbox627475386/main.go:6: imported and not used: "time"*

Works:

```
package main

import (
    _ "fmt"
    "log"
    "time"
)

var _ = log.Println

func main() {
    _ = time.Now
}
```

Another option is to remove or comment out the unused imports :–) The `goimports` tool can help you with that.

Short Variable Declarations Can Be Used Only Inside Functions

- level: beginner

Fails:

```
package main

myvar := 1 //error

func main() {
}
```

Compile Error:

> */tmp/sandbox265716165/main.go:3: non-declaration statement outside function body*

Works:

```
package main

var myvar = 1

func main() {
}
```

Redeclaring Variables Using Short Variable Declarations

- level: beginner

You can't redeclare a variable in a standalone statement, but it is allowed in multi-variable declarations where at least one new variable is also declared.

The redeclared variable has to be in the same block or you'll end up with a shadowed variable.

Fails:

```
package main

func main() {
    one := 0
    one := 1 //error
}
```

Compile Error:

> */tmp/sandbox706333626/main.go:5: no new variables on left side of :=*

Works:

```
package main

func main() {
    one := 0
    one, two := 1,2

    one,two = two,one
}
```

Can't Use Short Variable Declarations to Set Field Values

- level: beginner

Fails:

```
package main

import (
  "fmt"
)

type info struct {
  result int
}

func work() (int,error) {
    return 13,nil
  }

func main() {
  var data info

  data.result, err := work() //error
  fmt.Printf("info: %+v\n",data)
}
```

Compile Error:

> *prog.go:18: non-name data.result on left side of :=*

Even though there's a ticket to address this gotcha it's unlikely to change because Rob Pike likes it "as is" :-)

Use temporary variables or predeclare all your variables and use the standard assignment operator.

Works:

```
package main

import (
  "fmt"
)

type info struct {
  result int
}

func work() (int,error) {
    return 13,nil
  }

func main() {
  var data info

  var err error
  data.result, err = work() //ok
  if err != nil {
    fmt.Println(err)
    return
  }

  fmt.Printf("info: %+v\n",data) //prints: info: {result:13}
}
```

Accidental Variable Shadowing

- level: beginner

The short variable declaration syntax is so convenient (especially for those coming from a dynamic language) that it's easy to treat it like a regular assignment operation. If you make this mistake in a new code block there will be no compiler error, but your app will not do what you expect.

```
package main

import "fmt"

func main() {
    x := 1
    fmt.Println(x)     //prints 1
    {
        fmt.Println(x) //prints 1
        x := 2
        fmt.Println(x) //prints 2
    }
    fmt.Println(x)     //prints 1 (bad if you need 2)
}
```

This is a very common trap even for experienced Go developers. It's easy to make and it could be hard to spot.

You can use the `vet` command to find some of these problems. By default, `vet` will not perform any shadowed variable checks. Make sure to use the `-shadow` flag: `go tool vet -shadow your_file.go`

Note that the `vet` command will not report all shadowed variables. Use `go-nyet` for more aggressive shadowed variable detection.

Can't Use "nil" to Initialize a Variable Without an Explicit Type

- level: beginner

The "nil" identifier can be used as the "zero value" for interfaces, functions, pointers, maps, slices, and channels. If you don't specify the variable type the compiler will fail to compile your code because it can't guess the type.

Fails:

```
package main

func main() {
    var x = nil //error

    _ = x
}
```

Compile Error:

> */tmp/sandbox188239583/main.go:4: use of untyped nil*

Works:

```
package main

func main() {
    var x interface{} = nil

    _ = x
}
```

Using "nil" Slices and Maps

- level: beginner

It's OK to add items to a "nil" slice, but doing the same with a map will produce a runtime panic.

Works:

```
package main

func main() {
    var s []int
    s = append(s,1)
}
```

Fails:

```
package main

func main() {
    var m map[string]int
    m["one"] = 1 //error

}
```

Map Capacity

- level: beginner

You can specify the map capacity when it's created, but you can't use the `cap()` function on maps.

Fails:

```
package main

func main() {
    m := make(map[string]int,99)
    cap(m) //error
}
```

Compile Error:

> */tmp/sandbox326543983/main.go:5: invalid argument m (type map[string]int) for cap*

Strings Can't Be "nil"

- level: beginner

This is a gotcha for developers who are used to assigning "nil" identifiers to string variables.

Fails:

```
package main

func main() {
    var x string = nil //error

    if x == nil { //error
        x = "default"
    }
}
```

Compile Errors:

> /tmp/sandbox630560459/main.go:4: cannot use nil as type string in assignment
> /tmp/sandbox630560459/main.go:6: invalid operation: x == nil (mismatched types string and nil)

Works:

```
package main

func main() {
    var x string //defaults to "" (zero value)

    if x == "" {
        x = "default"
    }
}
```

Array Function Arguments

- level: beginner

If you are a C or C++ developer arrays for you are pointers. When you pass arrays to functions the functions reference the same memory location, so they can update the original data. Arrays in Go are values, so when you pass arrays to functions the functions get a copy of the original array data. This can be a problem if you are trying to update the array data.

```
package main

import "fmt"

func main() {
    x := [3]int{1,2,3}

    func(arr [3]int) {
        arr[0] = 7
        fmt.Println(arr) //prints [7 2 3]
    }(x)

    fmt.Println(x) //prints [1 2 3] (not ok if you need [7 2 3])
}
```

If you need to update the original array data use array pointer types.

```
package main

import "fmt"

func main() {
    x := [3]int{1,2,3}

    func(arr *[3]int) {
        (*arr)[0] = 7
        fmt.Println(arr) //prints &[7 2 3]
    }(&x)

    fmt.Println(x) //prints [7 2 3]
}
```

Another option is to use slices. Even though your function gets a copy of the slice variable it still references the original data.

```
package main

import "fmt"

func main() {
    x := []int{1,2,3}

    func(arr []int) {
        arr[0] = 7
        fmt.Println(arr) //prints [7 2 3]
    }(x)

    fmt.Println(x) //prints [7 2 3]
}
```

Unexpected Values in Slice and Array "range" Clauses

- level: beginner

This can happen if you are used to the "for-in" or "foreach" statements in other languages. The "range" clause in Go is different. It generates two values: the first value is the item index while the second value is the item data.

Bad:

```
package main

import "fmt"

func main() {
    x := []string{"a","b","c"}

    for v := range x {
        fmt.Println(v) //prints 0, 1, 2
    }
}
```

Good:

```
package main

import "fmt"

func main() {
    x := []string{"a","b","c"}

    for _, v := range x {
        fmt.Println(v) //prints a, b, c
    }
}
```

Slices and Arrays Are One-Dimensional

- level: beginner

It may seem like Go supports multi-dimensional arrays and slices, but it doesn't. Creating arrays of arrays or slices of slices is possible though. For numerical computation apps that rely on dynamic multi-dimensional arrays it's far from ideal in terms of performance and complexity.

You can build dynamic multi-dimensional arrays using raw one-dimensional arrays, slices of "independent" slices, and slices of "shared data" slices.

If you are using raw one-dimensional arrays you are responsible for indexing, bounds checking, and memory reallocations when the arrays need to grow.

Creating a dynamic multi-dimensional array using slices of "independent" slices is a two step process. First, you have to create the outer slice. Then, you have to allocate each inner slice. The inner slices are independent of each other. You can grow and shrink them without affecting other inner slices.

```
package main

func main() {
    x := 2
    y := 4

    table := make([][]int,x)
    for i:= range table {
        table[i] = make([]int,y)
    }
}
```

Creating a dynamic multi-dimensional array using slices of "shared data" slices is a three step process. First, you have to create the data "container" slice that will hold raw data. Then, you create the outer slice. Finally, you initialize each inner slice by reslicing the raw data slice.

```
package main

import "fmt"

func main() {
    h, w := 2, 4

    raw := make([]int,h*w)
    for i := range raw {
        raw[i] = i
    }
    fmt.Println(raw,&raw[4])
    //prints: [0 1 2 3 4 5 6 7] <ptr_addr_x>

    table := make([][]int,h)
    for i:= range table {
        table[i] = raw[i*w:i*w + w]
    }

    fmt.Println(table,&table[1][0])
    //prints: [[0 1 2 3] [4 5 6 7]] <ptr_addr_x>
}
```

There's a spec/proposal for multi-dimensional arrays and slices, but it looks like it's a low priority feature at this point in time.

Accessing Non-Existing Map Keys

- level: beginner

This is a gotcha for developers who expect to get "nil" identifiers (like it's done in other languages). The returned value will be "nil" if the "zero value" for the corresponding data type is "nil", but it'll be different for other data types. Checking for the appropriate "zero value" can be used to determine if the map record exists, but it's not always reliable (e.g., what do you do if you have a map of booleans where the "zero value" is false). The most reliable way to know if a given map record exists is to check the second value returned by the map access operation.

Bad:

```
package main

import "fmt"

func main() {
    x := map[string]string{"one":"a","two":"","three":"c"}

    if v := x["two"]; v == "" { //incorrect
        fmt.Println("no entry")
    }
}
```

Good:

```
package main

import "fmt"

func main() {
    x := map[string]string{"one":"a","two":"","three":"c"}

    if _,ok := x["two"]; !ok {
        fmt.Println("no entry")
    }
}
```

Strings Are Immutable

- level: beginner

Trying to update an individual character in a string variable using the index operator will result in a failure. Strings are read-only byte slices (with a few extra properties). If you do need to update a string then use a byte slice instead converting it to a string type when necessary.

Fails:

```
package main

import "fmt"

func main() {
    x := "text"
    x[0] = 'T'

    fmt.Println(x)
}
```

Compile Error:

> */tmp/sandbox305565531/main.go:7: cannot assign to x[0]*

Works:

```
package main

import "fmt"

func main() {
    x := "text"
    xbytes := []byte(x)
    xbytes[0] = 'T'

    fmt.Println(string(xbytes)) //prints Text
}
```

Note that this isn't really the right way to update characters in a text string because a given character could be stored in multiple bytes. If you do need to make updates to a text string convert it to a rune slice first. Even with rune slices a single character might span multiple runes, which can happen if you have characters with grave accent, for example. This complicated and ambiguous nature of "characters" is the reason why Go strings are represented as byte sequences.

Conversions Between Strings and Byte Slices

- level: beginner

When you convert a string to a byte slice (and vice versa) you get a complete copy of the orginal data. It's not like a cast operation in other languages and it's not like reslicing where the new slice variable points to the same underlying array used by the original byte slice.

Go does have a couple of optimizations for `[]byte` to `string` and `string` to `[]byte` conversions to avoid extra allocations (with more optimizations on the todo list).

The first optimization avoids extra allocations when `[]byte` keys are used to lookup entries in `map[string]` collections: `m[string(key)]` .

The second optimization avoids extra allocations in `for range` clauses where strings are converted to `[]byte` : `for i,v := range []byte(str) {...}` .

Strings and Index Operator

- level: beginner

The index operator on a string returns a byte value, not a character (like it's done in other languages).

```
package main

import "fmt"

func main() {
    x := "text"
    fmt.Println(x[0]) //print 116
    fmt.Printf("%T",x[0]) //prints uint8
}
```

If you need to access specific string "characters" (unicode code points/runes) use the `for range` clause. The official "unicode/utf8" package and the experimental utf8string package (golang.org/x/exp/utf8string) are also useful. The utf8string package includes a convenient `At()` method. Converting the string to a slice of runes is an option too.

Strings Are Not Always UTF8 Text

- level: beginner

String values are not required to be UTF8 text. They can contain arbitrary bytes. The only time strings are UTF8 is when string literals are used. Even then they can include other data using escape sequences.

To know if you have a UTF8 text string use the `ValidString()` function from the "unicode/utf8" package.

```
package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {
    data1 := "ABC"
    fmt.Println(utf8.ValidString(data1)) //prints: true

    data2 := "A\xfeC"
    fmt.Println(utf8.ValidString(data2)) //prints: false
}
```

String Length

- level: beginner

Let's say you are a python developer and you have the following piece of code:

```
data = u'♥'
print(len(data)) #prints: 1
```

When you convert it to a similar Go code snippet you might be surprised.

```
package main

import "fmt"

func main() {
    data := "♥"
    fmt.Println(len(data)) //prints: 3
}
```

The built-in `len()` function returns the number of bytes instead of the number of characters like it's done for unicode strings in Python.

To get the same results in Go use the `RuneCountInString()` function from the "unicode/utf8" package.

```
package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {
    data := "♥"
    fmt.Println(utf8.RuneCountInString(data)) //prints: 1
```

Technically the `RuneCountInString()` function doesn't return the number of characters because a single character may span multiple runes.

```
package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {
    data := "e"
    fmt.Println(len(data))                    //prints: 3
    fmt.Println(utf8.RuneCountInString(data)) //prints: 2
}
```

Missing Comma In Multi-Line Slice, Array, and Map Literals

- level: beginner

Fails:

```
package main

func main() {
    x := []int{
    1,
    2 //error
    }
    _ = x
}
```

Compile Errors:

> */tmp/sandbox367520156/main.go:6: syntax error: need trailing comma before newline in composite literal*
> */tmp/sandbox367520156/main.go:8: non-declaration statement outside function body*
> */tmp/sandbox367520156/main.go:9: syntax error: unexpected }*

Works:

```
package main

func main() {
    x := []int{
    1,
    2,
    }
    x = x

    y := []int{3,4,} //no error
    y = y
}
```

You won't get a compiler error if you leave the trailing comma when you collapse the declaration to be on a single line.

log.Fatal and log.Panic Do More Than Log

- level: beginner

Logging libraries often provide different log levels. Unlike those logging libraries, the log package in Go does more than log if you call its `Fatal*()` and `Panic*()` functions. When your app calls those functions Go will also terminate your app :-)

```
package main

import "log"

func main() {
    log.Fatalln("Fatal Level: log entry") //app exits here
    log.Println("Normal Level: log entry")
}
```

Built-in Data Structure Operations Are Not Synchronized

- level: beginner

Even though Go has a number of features to support concurrency natively, concurrency safe data collections are not one them :-) It's your responsibility to ensure the data collection updates are atomic. Goroutines and channels are the recommended way to implement those atomic operations, but you can also leverage the "sync" package if it makes sense for your application.

Iteration Values For Strings in "range" Clauses

- level: beginner

The index value (the first value returned by the "range" operation) is the index of the first byte for the current "character" (unicode code point/rune) returned in the second value. It's not the index for the current "character" like it's done in other languages. Note that an actual character might be represented by multiple runes. Make sure to check out the "norm" package (golang.org/x/text/unicode/norm) if you need to work with characters.

The `for range` clauses with string variables will try to interpret the data as UTF8 text. For any byte sequences it doesn't understand it will return 0xfffd runes (aka unicode replacement characters) instead of the actual data. If you have arbitrary (non-UTF8 text) data stored in your string variables, make sure to convert them to byte slices to get all stored data as is.

```
package main

import "fmt"

func main() {
    data := "A\xfe\x02\xff\x04"
    for _,v := range data {
        fmt.Printf("%#x ",v)
    }
    //prints: 0x41 0xfffd 0x2 0xfffd 0x4 (not ok)

    fmt.Println()
    for _,v := range []byte(data) {
        fmt.Printf("%#x ",v)
    }
    //prints: 0x41 0xfe 0x2 0xff 0x4 (good)
}
```

Iterating Through a Map Using a "for range" Clause

- level: beginner

This is a gotcha if you expect the items to be in a certain order (e.g., ordered by the key value). Each map iteration will produce different results. The Go runtime tries to go an extra mile randomizing the iteration order, but it doesn't always succeed so you may get several identical map iterations. Don't be surprised to see 5 identical iterations in a row.

```
package main

import "fmt"

func main() {
    m := map[string]int{"one":1,"two":2,"three":3,"four":4}
    for k,v := range m {
        fmt.Println(k,v)
    }
}
```

And if you use the Go Playground (https://play.golang.org/) you'll always get the same results because it doesn't recompile the code unless you make a change.

Fallthrough Behavior in "switch" Statements

- level: beginner

The "case" blocks in "switch" statements break by default. This is different from other languages where the default behavior is to fall through to the next "case" block.

```
package main

import "fmt"

func main() {
    isSpace := func(ch byte) bool {
        switch(ch) {
        case ' ': //error
        case '\t':
            return true
        }
        return false
    }

    fmt.Println(isSpace('\t')) //prints true (ok)
    fmt.Println(isSpace(' '))  //prints false (not ok)
}
```

You can force the "case" blocks to fall through by using the "fallthrough" statement at the end of each "case" block. You can also rewrite your switch statement to use expression lists in the "case" blocks.

```
package main

import "fmt"

func main() {
    isSpace := func(ch byte) bool {
        switch(ch) {
        case ' ', '\t':
            return true
        }
        return false
    }

    fmt.Println(isSpace('\t')) //prints true (ok)
    fmt.Println(isSpace(' '))  //prints true (ok)
}
```

Increments and Decrements

- level: beginner

Many languages have increment and decrement operators. Unlike other languages, Go doesn't support the prefix version of the operations. You also can't use these two operators in expressions.

Fails:

```
package main

import "fmt"

func main() {
    data := []int{1,2,3}
    i := 0
    ++i //error
    fmt.Println(data[i++]) //error
}
```

Compile Errors:

> */tmp/sandbox101231828/main.go:8: syntax error: unexpected ++ /tmp/sandbox101231828/main.go:9: syntax error: unexpected ++, expecting :*

Works:

```
package main

import "fmt"

func main() {
    data := []int{1,2,3}
    i := 0
    i++
    fmt.Println(data[i])
}
```

Bitwise NOT Operator

- level: beginner

Many languages use `~` as the unary NOT operator (aka bitwise complement), but Go reuses the XOR operator (`^`) for that.

Fails:

```
package main

import "fmt"

func main() {
    fmt.Println(~2) //error
}
```

Compile Error:

> */tmp/sandbox965529189/main.go:6: the bitwise complement operator is ^*

Works:

```
package main

import "fmt"

func main() {
    var d uint8 = 2
    fmt.Printf("%08b\n",^d)
}
```

Go still uses `^` as the XOR operator, which may be confusing for some people.

If you want you can represent a unary NOT operation (e.g, `NOT 0x02`) with a binary XOR operation (e.g., `0x02 XOR 0xff`). This could explain why `^` is reused to represent unary NOT operations.

Go also has a special 'AND NOT' bitwise operator (`&^`), which adds to the NOT operator confusion. It looks like a special feature/hack to support `A AND (NOT B)` without requiring parentheses.

```
package main

import "fmt"

func main() {
    var a uint8 = 0x82
    var b uint8 = 0x02
    fmt.Printf("%08b [A]\n",a)
    fmt.Printf("%08b [B]\n",b)

    fmt.Printf("%08b (NOT B)\n",^b)
    fmt.Printf("%08b ^ %08b = %08b [B XOR 0xff]\n",b,0xff,b ^ 0xff)

    fmt.Printf("%08b ^ %08b = %08b [A XOR B]\n",a,b,a ^ b)
    fmt.Printf("%08b & %08b = %08b [A AND B]\n",a,b,a & b)
    fmt.Printf("%08b &^%08b = %08b [A 'AND NOT' B]\n",a,b,a &^ b)
    fmt.Printf("%08b&(^%08b)= %08b [A AND (NOT B)]\n",a,b,a & (^b))
}
```

Operator Precedence Differences

- level: beginner

Aside from the "bit clear" operators (`&^`) Go has a set of standard operators shared by many other languages. The operator precedence is not always the same though.

```
package main

import "fmt"

func main() {
    fmt.Printf("0x2 & 0x2 + 0x4 -> %#x\n",0x2 & 0x2 + 0x4)
    //prints: 0x2 & 0x2 + 0x4 -> 0x6
    //Go:    (0x2 & 0x2) + 0x4
    //C++:    0x2 & (0x2 + 0x4) -> 0x2

    fmt.Printf("0x2 + 0x2 << 0x1 -> %#x\n",0x2 + 0x2 << 0x1)
    //prints: 0x2 + 0x2 << 0x1 -> 0x6
    //Go:     0x2 + (0x2 << 0x1)
    //C++:   (0x2 + 0x2) << 0x1 -> 0x8

    fmt.Printf("0xf | 0x2 ^ 0x2 -> %#x\n",0xf | 0x2 ^ 0x2)
    //prints: 0xf | 0x2 ^ 0x2 -> 0xd
    //Go:    (0xf | 0x2) ^ 0x2
    //C++:    0xf | (0x2 ^ 0x2) -> 0xf
}
```

Unexported Structure Fields Are Not Encoded

- level: beginner

The struct fields starting with lowercase letters will not be (json, xml, gob, etc.) encoded, so when you decode the structure you'll end up with zero values in those unexported fields.

```
package main

import (
    "fmt"
    "encoding/json"
)

type MyData struct {
    One int
    two string
}

func main() {
    in := MyData{1,"two"}
    fmt.Printf("%#v\n",in) //prints main.MyData{One:1, two:"two"}

    encoded,_ := json.Marshal(in)
    fmt.Println(string(encoded)) //prints {"One":1}

    var out MyData
    json.Unmarshal(encoded,&out)

    fmt.Printf("%#v\n",out) //prints main.MyData{One:1, two:""}
}
```

App Exits With Active Goroutines

- level: beginner

The app will not wait for all your goroutines to complete. This is a common mistake for beginners in general. Everybody starts somewhere, so there's no shame in making rookie mistakes :-)

```
package main

import (
    "fmt"
    "time"
)

func main() {
    workerCount := 2

    for i := 0; i < workerCount; i++ {
        go doit(i)
    }
    time.Sleep(1 * time.Second)
    fmt.Println("all done!")
}

func doit(workerId int) {
    fmt.Printf("[%v] is running\n",workerId)
    time.Sleep(3 * time.Second)
    fmt.Printf("[%v] is done\n",workerId)
}
```

You'll see:

> *[0] is running*
> *[1] is running*
> *all done!*

One of the most common solutions is to use a "WaitGroup" variable. It will allow the main goroutine to wait until all worker goroutines are done. If your app has long running workers with message processing loops you'll also need a way to signal those goroutines that it's time to exit. You can send a "kill" message to each worker. Another option is to close a channel all workers are receiving from. It's a simple way to signal all goroutines at once.

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    done := make(chan struct{})
    workerCount := 2

    for i := 0; i < workerCount; i++ {
        wg.Add(1)
        go doit(i,done,wg)
    }

    close(done)
    wg.Wait()
    fmt.Println("all done!")
}

func doit(workerId int,done <-chan struct{},wg sync.WaitGroup) {
    fmt.Printf("[%v] is running\n",workerId)
    defer wg.Done()
    <- done
    fmt.Printf("[%v] is done\n",workerId)
}
```

If you run this app you'll see:

> *[0] is running*
> *[0] is done*
> *[1] is running*
> *[1] is done*

Looks like the workers are done before the main goroutine exits. Great! However, you'll also see this:

> *fatal error: all goroutines are asleep - deadlock!*

That's not so great :-) What's going on? Why is there a deadlock? The workers exited and they executed `wg.Done()` . The app should work.

The deadlock happens because each worker gets a copy of the original "WaitGroup" variable. When workers execute `wg.Done()` it has no effect on the "WaitGroup" variable in the main goroutine.

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    done := make(chan struct{})
    wq := make(chan interface{})
    workerCount := 2

    for i := 0; i < workerCount; i++ {
        wg.Add(1)
        go doit(i,wq,done,&wg)
    }

    for i := 0; i < workerCount; i++ {
        wq <- i
    }

    close(done)
    wg.Wait()
    fmt.Println("all done!")
}

func doit(workerId int, wq <-chan interface{},done <-chan struct{},wg *sync.WaitGroup) {
    fmt.Printf("[%v] is running\n",workerId)
    defer wg.Done()
    for {
        select {
        case m := <- wq:
            fmt.Printf("[%v] m => %v\n",workerId,m)
        case <- done:
            fmt.Printf("[%v] is done\n",workerId)
            return
        }
    }
}
```

Now it works as expected :-)

Sending to an Unbuffered Channel Returns As Soon As the Target Receiver Is Ready

- level: beginner

The sender will not be blocked until your message is processed by the receiver. Depending on the machine where you are running the code, the receiver goroutine may or may not have enough time to process the message before the sender continues its execution.

```
package main

import "fmt"

func main() {
    ch := make(chan string)

    go func() {
        for m := range ch {
            fmt.Println("processed:",m)
        }
    }()

    ch <- "cmd.1"
    ch <- "cmd.2" //won't be processed
}
```

Sending to an Closed Channel Causes a Panic

- level: beginner

Receiving from a closed channel is safe. The `ok` return value in a receive statement will be set to `false` indicating that no data was received. If you are receiving from a buffered channel you'll get the buffered data first and once it's empty the `ok` return value will be `false` .

Sending data to a closed channel causes a panic. It is a documented behavior, but it's not very intuitive for new Go developers who might expect the send behavior to be similar to the receive behavior.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch := make(chan int)
    for i := 0; i < 3; i++ {
        go func(idx int) {
            ch <- (idx + 1) * 2
        }(i)
    }

    //get the first result
    fmt.Println(<-ch)
    close(ch) //not ok (you still have other senders)
    //do other work
    time.Sleep(2 * time.Second)
}
```

Depending on your application the fix will be different. It might be a minor code change or it might require a change in your application design. Either way, you'll need to make sure your application doesn't try to send data to a closed channel.

The buggy example can be fixed by using a special cancellation channel to signal the remaining workers that their results are no longer neeeded.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch := make(chan int)
    done := make(chan struct{})
    for i := 0; i < 3; i++ {
        go func(idx int) {
            select {
            case ch <- (idx + 1) * 2: fmt.Println(idx,"sent result")
            case <- done: fmt.Println(idx,"exiting")
            }
        }(i)
    }

    //get first result
    fmt.Println("result:",<-ch)
    close(done)
    //do other work
    time.Sleep(3 * time.Second)
}
```

Using "nil" Channels

- level: beginner

Send and receive operations on a `nil` channel block forver. It's a well documented behavior, but it can be a surprise for new Go developers.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    var ch chan int
    for i := 0; i < 3; i++ {
        go func(idx int) {
            ch <- (idx + 1) * 2
        }(i)
    }

    //get first result
    fmt.Println("result:",<-ch)
    //do other work
    time.Sleep(2 * time.Second)
}
```

If you run the code you'll see a runtime error like this: `fatal error: all goroutines are asleep - deadlock!`

This behavior can be used as a way to dynamically enable and disable `case` blocks in a `select` statement.

```
package main

import "fmt"
import "time"

func main() {
    inch := make(chan int)
    outch := make(chan int)

    go func() {
        var in <- chan int = inch
        var out chan <- int
        var val int
        for {
            select {
            case out <- val:
                out = nil
                in = inch
            case val = <- in:
                out = outch
                in = nil
            }
        }
    }()

    go func() {
        for r := range outch {
            fmt.Println("result:",r)
        }
    }()

    time.Sleep(0)
    inch <- 1
    inch <- 2
    time.Sleep(3 * time.Second)
}
```

Methods with Value Receivers Can't Change the Original Value

- level: beginner

Method receivers are like regular function arguments. If it's declared to be a value then your function/method gets a copy of your receiver argument. This means making changes to the receiver will not affect the original value unless your receiver is a map or slice variable and you are updating the items in the collection or the fields you are updating in the receiver are pointers.

```
package main

import "fmt"

type data struct {
    num int
    key *string
    items map[string]bool
}

func (this *data) pmethod() {
    this.num = 7
}

func (this data) vmethod() {
    this.num = 8
    *this.key = "v.key"
    this.items["vmethod"] = true
}

func main() {
    key := "key.1"
    d := data{1,&key,make(map[string]bool)}

    fmt.Printf("num=%v key=%v items=%v\n",d.num,*d.key,d.items)
    //prints num=1 key=key.1 items=map[]

    d.pmethod()
    fmt.Printf("num=%v key=%v items=%v\n",d.num,*d.key,d.items)
    //prints num=7 key=key.1 items=map[]

    d.vmethod()
    fmt.Printf("num=%v key=%v items=%v\n",d.num,*d.key,d.items)
    //prints num=7 key=v.key items=map[vmethod:true]
}
```

Closing HTTP Response Body

- level: intermediate

When you make requests using the standard http library you get a http response variable. If you don't read the response body you still need to close it. Note that you must do it for empty responses too. It's very easy to forget especially for new Go developers.

Some new Go developers do try to close the response body, but they do it in the wrong place.

```
package main

import (
    "fmt"
    "net/http"
    "io/ioutil"
)

func main() {
    resp, err := http.Get("https://api.ipify.org?format=json")
    defer resp.Body.Close()//not ok
    if err != nil {
        fmt.Println(err)
        return
    }

    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(string(body))
}
```

This code works for successful requests, but if the http request fails the `resp` variable might be `nil`, which will cause a runtime panic.

The most common why to close the response body is by using a `defer` call after the http response error check.

```
package main

import (
    "fmt"
    "net/http"
    "io/ioutil"
)

func main() {
    resp, err := http.Get("https://api.ipify.org?format=json")
    if err != nil {
        fmt.Println(err)
        return
    }

    defer resp.Body.Close()//ok, most of the time :-)
    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(string(body))
}
```

Most of the time when your http request fails the `resp` variable will be `nil` and the `err` variable will be `non-nil`. However, when you get a redirection failure both variables will be `non-nil`. This means you can still end up with a leak.

You can fix this leak by adding a call to close `non-nil` response bodies in the http response error handling

block. Another option is to use one `defer` call to close response bodies for all failed and successful requests.

```
package main

import (
    "fmt"
    "net/http"
    "io/ioutil"
)

func main() {
    resp, err := http.Get("https://api.ipify.org?format=json")
    if resp != nil {
        defer resp.Body.Close()
    }

    if err != nil {
        fmt.Println(err)
        return
    }

    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(string(body))
}
```

The orignal implementation for `resp.Body.Close()` also reads and discards the remaining response body data. This ensured that the http connection could be reused for another request if the keepalive http connection behavior is enabled. The latest http client behavior is different. Now it's your responsibility to read and discard the remaining response data. If you don't do it the http connection might be closed instead of being reused. This little gotcha is supposed to be documented in Go 1.5.

If reusing the http connection is important for your application you might need to add something like this at the end of your response processing logic:

```
_, err = io.Copy(ioutil.Discard, resp.Body)
```

It will be necessary if you don't read the entire response body right away, which might happen if you are processing json API responses with code like this:

```
json.NewDecoder(resp.Body).Decode(&data)
```

Closing HTTP Connections

- level: intermediate

Some HTTP servers keep network connections open for a while (based on the HTTP 1.1 spec and the server "keep-alive" configurations). By default, the standard http library will close the network connections only when the target HTTP server asks for it. This means your app may run out of sockets/file descriptors under certain conditions.

You can ask the http library to close the connection after your request is done by setting the `Close` field in the request variable to `true` .

Another option is to add a `Connection` request header and set it to `close` . The target HTTP server should respond with a `Connection: close` header too. When the http library sees this response header it will also close the connection.

```
package main

import (
    "fmt"
    "net/http"
    "io/ioutil"
)

func main() {
    req, err := http.NewRequest("GET","http://golang.org",nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    req.Close = true
    //or do this:
    //req.Header.Add("Connection", "close")

    resp, err := http.DefaultClient.Do(req)
    if resp != nil {
        defer resp.Body.Close()
    }

    if err != nil {
        fmt.Println(err)
        return
    }

    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(len(string(body)))
}
```

You can also disable http connection reuse globally. You'll need to create a custom http transport configuration for it.

```
package main

import (
    "fmt"
    "net/http"
    "io/ioutil"
)

func main() {
    tr := &http.Transport{DisableKeepAlives: true}
    client := &http.Client{Transport: tr}

    resp, err := client.Get("http://golang.org")
    if resp != nil {
        defer resp.Body.Close()
    }

    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(resp.StatusCode)

    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(len(string(body)))
}
```

If you send a lot of requests to the same HTTP server it's ok to keep the network connection open. However, if your app sends one or two requests to many different HTTP servers in a short period of time it's a good idea to close the network connections right after your app receives the responses. Increasing the open file limit might be a good idea too. The correct solution depends on your application though.

JSON Encoder Adds a Newline Character

- level: intermediate

You are writing a test for your JSON encoding function when you discover that your tests fail because you are not getting the expected value. What happened? If you are using the JSON Encoder object then you'll get an extra newline character at the end of your encoded JSON object.

```
package main

import (
  "fmt"
  "encoding/json"
  "bytes"
)

func main() {
  data := map[string]int{"key": 1}

  var b bytes.Buffer
  json.NewEncoder(&b).Encode(data)

  raw,_ := json.Marshal(data)

  if b.String() == string(raw) {
    fmt.Println("same encoded data")
  } else {
    fmt.Printf("'%s' != '%s'\n",raw,b.String())
    //prints:
    //'{"key":1}' != '{"key":1}\n'
  }
}
```

The JSON Encoder object is designed for streaming. Streaming with JSON usually means newline delimited JSON objects and this is why the Encode method adds a newline character. This is a documented behavior, but it's commonly overlooked or forgotten.

JSON Package Escapes Special HTML Characters in Keys and String Values

- level: intermediate

This is a documented behavior, but you have to be careful reading all of the JSON package documentation to learn about it. The `SetEscapeHTML` method description talks about the default encoding behavior for the and, less than and greater than characters.

This is a very unfortunate design decision by the Go team for a number of reasons. First, you can't disable this behavior for the `json.Marshal` calls. Second, this is a badly implemented security feature because it assumes that doing HTML encoding is sufficient to protect against XSS vulnerabilities in all web applications. There are a lot of different contexts where the data can be used and each context requires its own encoding method. And finally, it's bad because it assumes that the primary use case for JSON is a web page, which breaks the configuration libraries and the REST/HTTP APIs by default.

```
package main

import (
  "fmt"
  "encoding/json"
  "bytes"
)

func main() {
  data := "x < y"

  raw,_ := json.Marshal(data)
  fmt.Println(string(raw))
  //prints: "x \u003c y" <- probably not what you expected

  var b1 bytes.Buffer
  json.NewEncoder(&b1).Encode(data)
  fmt.Println(b1.String())
  //prints: "x \u003c y" <- probably not what you expected

  var b2 bytes.Buffer
  enc := json.NewEncoder(&b2)
  enc.SetEscapeHTML(false)
  enc.Encode(data)
  fmt.Println(b2.String())
  //prints: "x < y" <- looks better
}
```

A suggestion to the Go team... Make it an opt-in.

Unmarshalling JSON Numbers into Interface Values

- level: intermediate

By default, Go treats numeric values in JSON as `float64` numbers when you decode/unmarshal JSON data into an interface. This means the following code will fail with a panic:

```
package main

import (
  "encoding/json"
  "fmt"
)

func main() {
  var data = []byte(`{"status": 200}`)

  var result map[string]interface{}
  if err := json.Unmarshal(data, &result); err != nil {
    fmt.Println("error:", err)
    return
  }

  var status = result["status"].(int) //error
  fmt.Println("status value:",status)
}
```

Runtime Panic:

> *panic: interface conversion: interface is float64, not int*

If the JSON value you are trying to decode is an integer you have serveral options.

Option one: use the float value as-is :-)

Option two: convert the float value to the integer type you need.

```
package main

import (
  "encoding/json"
  "fmt"
)

func main() {
  var data = []byte(`{"status": 200}`)

  var result map[string]interface{}
  if err := json.Unmarshal(data, &result); err != nil {
    fmt.Println("error:", err)
    return
  }

  var status = uint64(result["status"].(float64)) //ok
  fmt.Println("status value:",status)
}
```

Option three: use a `Decoder` type to unmarshal JSON and tell it to represent JSON numbers using the `Number` interface type.

```
package main

import (
  "encoding/json"
  "bytes"
  "fmt"
)

func main() {
  var data = []byte(`{"status": 200}`)

  var result map[string]interface{}
  var decoder = json.NewDecoder(bytes.NewReader(data))
  decoder.UseNumber()

  if err := decoder.Decode(&result); err != nil {
    fmt.Println("error:", err)
    return
  }

  var status,_ = result["status"].(json.Number).Int64() //ok
  fmt.Println("status value:",status)
}
```

You can use the string representation of your `Number` value to unmarshal it to a different numeric type:

```
package main

import (
  "encoding/json"
  "bytes"
  "fmt"
)

func main() {
  var data = []byte(`{"status": 200}`)

  var result map[string]interface{}
  var decoder = json.NewDecoder(bytes.NewReader(data))
  decoder.UseNumber()

  if err := decoder.Decode(&result); err != nil {
    fmt.Println("error:", err)
    return
  }

  var status uint64
  if err := json.Unmarshal([]byte(result["status"].(json.Number).String()), &status); err != nil {
    fmt.Println("error:", err)
    return
  }

  fmt.Println("status value:",status)
}
```

Option four: use a `struct` type that maps your numeric value to the numeric type you need.

```
package main

import (
  "encoding/json"
  "bytes"
  "fmt"
)

func main() {
  var data = []byte(`{"status": 200}`)

  var result struct {
    Status uint64 `json:"status"`
  }

  if err := json.NewDecoder(bytes.NewReader(data)).Decode(&result); err != nil {
    fmt.Println("error:", err)
    return
  }

  fmt.Printf("result => %+v",result)
  //prints: result => {Status:200}
}
```

Option five: use a `struct` that maps your numeric value to the `json.RawMessage` type if you need to defer the value decoding.

This option is useful if you have to perform conditional JSON field decoding where the field type or structure might change.

```
package main

import (
  "encoding/json"
  "bytes"
  "fmt"
)

func main() {
  records := [][]byte{
    []byte(`{"status": 200, "tag":"one"}`),
    []byte(`{"status":"ok", "tag":"two"}`),
  }

  for idx, record := range records {
    var result struct {
      StatusCode uint64
      StatusName string
      Status json.RawMessage `json:"status"`
      Tag string             `json:"tag"`
    }

    if err := json.NewDecoder(bytes.NewReader(record)).Decode(&result); err != nil {
      fmt.Println("error:", err)
      return
    }

    var sstatus string
    if err := json.Unmarshal(result.Status, &sstatus); err == nil {
      result.StatusName = sstatus
    }

    var nstatus uint64
    if err := json.Unmarshal(result.Status, &nstatus); err == nil {
      result.StatusCode = nstatus
    }

    fmt.Printf("[%v] result => %+v\n",idx,result)
  }
}
```

JSON String Values Will Not Be Ok with Hex or Other non-UTF8 Escape Sequences

- level: intermediate

Go expects string values to be UTF8 encoded. This means you can't have arbitrary hex escaped binary data in your JSON strings (and you also have to escape the backslash character). This is really a JSON gotcha Go inherited, but it happens often enough in Go apps that it makes sense to mention it anyways.

```
package main

import (
  "fmt"
  "encoding/json"
)

type config struct {
  Data string `json:"data"`
}

func main() {
  raw := []byte(`{"data":"\xc2"}`)
  var decoded config

  if err := json.Unmarshal(raw, &decoded); err != nil {
        fmt.Println(err)
    //prints: invalid character 'x' in string escape code
    }

}
```

The Unmarshal/Decode calls will fail if Go sees a hex escape sequence. If you do need to have a backslash in you string make sure to escape it with another backlash. If you want to use hex encoded binary data you can escape the backslash and then do your own hex escaping with the decoded data in your JSON string.

```
package main

import (
  "fmt"
  "encoding/json"
)

type config struct {
  Data string `json:"data"`
}

func main() {
  raw := []byte(`{"data":"\\xc2"}`)

  var decoded config

  json.Unmarshal(raw, &decoded)

  fmt.Printf("%#v",decoded) //prints: main.config{Data:"\\xc2"}
  //todo: do your own hex escape decoding for decoded.Data
}
```

Another option is to use the byte array/slice data type in your JSON object, but the binary data will have to be base64 encoded.

```
package main

import (
  "fmt"
  "encoding/json"
)

type config struct {
  Data []byte `json:"data"`
}

func main() {
  raw := []byte(`{"data":"wg=="}`)
  var decoded config

  if err := json.Unmarshal(raw, &decoded); err != nil {
         fmt.Println(err)
      }

  fmt.Printf("%#v",decoded) //prints: main.config{Data:[]uint8{0xc2}}
}
```

Something else to watch out for is the Unicode replacement character (U+FFFD). Go will use the replacement character instead of invalid UTF8, so the Unmarshal/Decode call will not fail, but the string value you get might not be what you expected.

Comparing Structs, Arrays, Slices, and Maps

- level: intermediate

You can use the equality operator, `==` , to compare struct variables if each structure field can be compared with the equality operator.

```
package main

import "fmt"

type data struct {
    num int
    fp float32
    complex complex64
    str string
    char rune
    yes bool
    events <-chan string
    handler interface{}
    ref *byte
    raw [10]byte
}

func main() {
    v1 := data{}
    v2 := data{}
    fmt.Println("v1 == v2:",v1 == v2) //prints: v1 == v2: true
}
```

If any of the struct fields are not comparable then using the equality operator will result in compile time errors. Note that arrays are comparable only if their data items are comparable.

```
package main

import "fmt"

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:",v1 == v2)
}
```

Go does provide a number of helper functions to compare variables that can't be compared using the comparison operators.

The most generic solution is to use the `DeepEqual()` function in the reflect package.

```
package main

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
}
```

Aside from being slow (which may or may not be a deal breaker for your application), `DeepEqual()` also has its own gotchas.

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var b1 []byte = nil
    b2 := []byte{}
    fmt.Println("b1 == b2:",reflect.DeepEqual(b1, b2)) //prints: b1 == b2: false
}
```

`DeepEqual()` doesn't consider an empty slice to be equal to a "nil" slice. This behavior is different from the behavior you get using the `bytes.Equal()` function. `bytes.Equal()` considers "nil" and empty slices to be equal.

```
package main

import (
    "fmt"
    "bytes"
)

func main() {
    var b1 []byte = nil
    b2 := []byte{}
    fmt.Println("b1 == b2:",bytes.Equal(b1, b2)) //prints: b1 == b2: true
}
```

`DeepEqual()` isn't always perfect comparing slices.

```
package main

import (
    "fmt"
    "reflect"
    "encoding/json"
)

func main() {
    var str string = "one"
    var in interface{} = "one"
    fmt.Println("str == in:",str == in,reflect.DeepEqual(str, in))
    //prints: str == in: true true

    v1 := []string{"one","two"}
    v2 := []interface{}{"one","two"}
    fmt.Println("v1 == v2:",reflect.DeepEqual(v1, v2))
    //prints: v1 == v2: false (not ok)

    data := map[string]interface{}{
        "code": 200,
        "value": []string{"one","two"},
    }
    encoded, _ := json.Marshal(data)
    var decoded map[string]interface{}
    json.Unmarshal(encoded, &decoded)
    fmt.Println("data == decoded:",reflect.DeepEqual(data, decoded))
    //prints: data == decoded: false (not ok)
}
```

If your byte slices (or strings) contain text data you might be tempted to use `ToUpper()` or `ToLower()` from the "bytes" and "strings" packages when you need to compare values in a case insensitive manner (before using `==` , `bytes.Equal()` , or `bytes.Compare()`). It will work for English text, but it will not work for text in many other languages. `strings.EqualFold()` and `bytes.EqualFold()` should be used instead.

If your byte slices contain secrets (e.g., cryptographic hashes, tokens, etc.) that need to be validated against user-provided data, don't use `reflect.DeepEqual()` , `bytes.Equal()` , or `bytes.Compare()` because those functions will make your application vulnerable to **timing attacks**. To avoid leaking the timing information use the functions from the 'crypto/subtle' package (e.g., `subtle.ConstantTimeCompare()`).

Recovering From a Panic

- level: intermediate

The `recover()` function can be used to catch/intercept a panic. Calling `recover()` will do the trick only when it's done in a deferred function.

Incorrect:

```
package main

import "fmt"

func main() {
    recover() //doesn't do anything
    panic("not good")
    recover() //won't be executed :)
    fmt.Println("ok")
}
```

Works:

```
package main

import "fmt"

func main() {
    defer func() {
        fmt.Println("recovered:",recover())
    }()

    panic("not good")
}
```

The call to `recover()` works only if it's called directly in your deferred function.

Fails:

```
package main

import "fmt"

func doRecover() {
    fmt.Println("recovered =>",recover()) //prints: recovered => <nil>
}

func main() {
    defer func() {
        doRecover() //panic is not recovered
    }()

    panic("not good")
}
```

Updating and Referencing Item Values in Slice, Array, and Map "range" Clauses

- level: intermediate

The data values generated in the "range" clause are copies of the actual collection elements. They are not references to the original items. This means that updating the values will not change the original data. It also means that taking the address of the values will not give you pointers to the original data.

```
package main

import "fmt"

func main() {
    data := []int{1,2,3}
    for _,v := range data {
        v *= 10 //original item is not changed
    }

    fmt.Println("data:",data) //prints data: [1 2 3]
}
```

If you need to update the original collection record value use the index operator to access the data.

```
package main

import "fmt"

func main() {
    data := []int{1,2,3}
    for i,_ := range data {
        data[i] *= 10
    }

    fmt.Println("data:",data) //prints data: [10 20 30]
}
```

If your collection holds pointer values then the rules are slightly different. You still need to use the index operator if you want the original record to point to another value, but you can update the data stored at the target location using the second value in the "for range" clause.

```
package main

import "fmt"

func main() {
    data := []*struct{num int} {{1},{2},{3}}

    for _,v := range data {
        v.num *= 10
    }

    fmt.Println(data[0],data[1],data[2]) //prints &{10} &{20} &{30}
}
```

"Hidden" Data in Slices

- level: intermediate

When you reslice a slice, the new slice will reference the array of the original slice. If you forget about this behavior it can lead to unexpected memory usage if your application allocates large temporary slices creating new slices from them to refer to small sections of the original data.

```
package main

import "fmt"

func get() []byte {
    raw := make([]byte,10000)
    fmt.Println(len(raw),cap(raw),&raw[0]) //prints: 10000 10000 <byte_addr_x>
    return raw[:3]
}

func main() {
    data := get()
    fmt.Println(len(data),cap(data),&data[0]) //prints: 3 10000 <byte_addr_x>
}
```

To avoid this trap make sure to copy the data you need from the temporary slice (instead of reslicing it).

```
package main

import "fmt"

func get() []byte {
    raw := make([]byte,10000)
    fmt.Println(len(raw),cap(raw),&raw[0]) //prints: 10000 10000 <byte_addr_x>
    res := make([]byte,3)
    copy(res,raw[:3])
    return res
}

func main() {
    data := get()
    fmt.Println(len(data),cap(data),&data[0]) //prints: 3 3 <byte_addr_y>
}
```

Slice Data "Corruption"

- level: intermediate

Let's say you need to rewrite a path (stored in a slice). You reslice the path to reference each directory modifying the first folder name and then you combine the names to create a new path.

```
package main

import (
    "fmt"
    "bytes"
)

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 => BBBBBBBB

    dir1 = append(dir1,"suffix"...)
    path = bytes.Join([][]byte{dir1,dir2},[]byte{'/'})

    fmt.Println("dir1 =>",string(dir1)) //prints: dir1 => AAAAsuffix
    fmt.Println("dir2 =>",string(dir2)) //prints: dir2 => uffixBBBB (not ok)

    fmt.Println("new path =>",string(path))
}
```

It didn't work as you expected. Instead of "AAAAsuffix/BBBBBBBB" you ended up with "AAAAsuffix/uffixBBBB". It happened because both directory slices referenced the same underlying array data from the original path slice. This means that the original path is also modified. Depending on your application this might be a problem too.

This problem can fixed by allocating new slices and copying the data you need. Another option is to use the full slice expression.

```
package main

import (
    "fmt"
    "bytes"
)

func main() {
    path := []byte("AAAA/BBBBBBBBB")
    sepIndex := bytes.IndexByte(path,'/')
    dir1 := path[:sepIndex:sepIndex] //full slice expression
    dir2 := path[sepIndex+1:]
    fmt.Println("dir1 =>",string(dir1)) //prints: dir1 => AAAA
    fmt.Println("dir2 =>",string(dir2)) //prints: dir2 => BBBBBBBB

    dir1 = append(dir1,"suffix"...)
    path = bytes.Join([][]byte{dir1,dir2},[]byte{'/'})

    fmt.Println("dir1 =>",string(dir1)) //prints: dir1 => AAAAsuffix
    fmt.Println("dir2 =>",string(dir2)) //prints: dir2 => BBBBBBBB (ok now)

    fmt.Println("new path =>",string(path))
}
```

The extra parameter in the full slice expression controls the capacity for the new slice. Now appending to that slice will trigger a new buffer allocation instead of overwriting the data in the second slice.

"Stale" Slices

- level: intermediate

Multiple slices can reference the same data. This can happen when you create a new slice from an existing slice, for example. If your application relies on this behavior to function properly then you'll need to worry about "stale" slices.

At some point adding data to one of the slices will result in a new array allocation when the original array can't hold any more new data. Now other slices will point to the old array (with old data).

```
import "fmt"

func main() {
    s1 := []int{1,2,3}
    fmt.Println(len(s1),cap(s1),s1) //prints 3 3 [1 2 3]

    s2 := s1[1:]
    fmt.Println(len(s2),cap(s2),s2) //prints 2 2 [2 3]

    for i := range s2 { s2[i] += 20 }

    //still referencing the same array
    fmt.Println(s1) //prints [1 22 23]
    fmt.Println(s2) //prints [22 23]

    s2 = append(s2,4)

    for i := range s2 { s2[i] += 10 }

    //s1 is now "stale"
    fmt.Println(s1) //prints [1 22 23]
    fmt.Println(s2) //prints [32 33 14]
}
```

Type Declarations and Methods

- level: intermediate

When you create a type declaration by defining a new type from an existing (non-interface) type, you don't inherit the methods defined for that existing type.

Fails:

```
package main

import "sync"

type myMutex sync.Mutex

func main() {
    var mtx myMutex
    mtx.Lock() //error
    mtx.Unlock() //error
}
```

Compile Errors:

> */tmp/sandbox106401185/main.go:9: mtx.Lock undefined (type myMutex has no field or method Lock)*
> */tmp/sandbox106401185/main.go:10: mtx.Unlock undefined (type myMutex has no field or method Unlock)*

If you do need the methods from the original type you can define a new struct type embedding the original type as an anonymous field.

Works:

```
package main

import "sync"

type myLocker struct {
    sync.Mutex
}

func main() {
    var lock myLocker
    lock.Lock() //ok
    lock.Unlock() //ok
}
```

Interface type declarations also retain their method sets.

Works:

```
package main

import "sync"

type myLocker sync.Locker

func main() {
    var lock myLocker = new(sync.Mutex)
    lock.Lock() //ok
    lock.Unlock() //ok
}
```

Breaking Out of "for switch" and "for select" Code Blocks

- level: intermediate

A "break" statement without a label only gets you out of the inner switch/select block. If using a "return" statement is not an option then defining a label for the outer loop is the next best thing.

```
package main

import "fmt"

func main() {
    loop:
        for {
            switch {
            case true:
                fmt.Println("breaking out...")
                break loop
            }
        }

    fmt.Println("out!")
}
```

A "goto" statement will do the trick too...

Iteration Variables and Closures in "for" Statements

- level: intermediate

This is the most common gotcha in Go. The iteration variables in `for` statements are reused in each iteration. This means that each closure (aka function literal) created in your `for` loop will reference the same variable (and they'll get that variable's value at the time those goroutines start executing).

Incorrect:

```
package main

import (
    "fmt"
    "time"
)

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

    for _,v := range data {
        go func() {
            fmt.Println(v)
        }()
    }

    time.Sleep(3 * time.Second)
    //goroutines print: three, three, three
}
```

The easiest solution (that doesn't require any changes to the goroutine) is to save the current iteration variable value in a local variable inside the `for` loop block.

Works:

```
package main

import (
    "fmt"
    "time"
)

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

    for _,v := range data {
        vcopy := v //
        go func() {
            fmt.Println(vcopy)
        }()
    }

    time.Sleep(3 * time.Second)
    //goroutines print: one, two, three
}
```

Another solution is to pass the current iteration variable as a parameter to the anonymous goroutine.

Works:

```
package main

import (
    "fmt"
    "time"
)

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

    for _,v := range data {
        go func(in string) {
            fmt.Println(in)
        }(v)
    }

    time.Sleep(3 * time.Second)
    //goroutines print: one, two, three
}
```

Here's a slightly more complicated version of the trap.

Incorrect:

```
package main

import (
    "fmt"
    "time"
)

type field struct {
    name string
}

func (p *field) print() {
    fmt.Println(p.name)
}

func main() {
    data := []field{{"one"},{"two"},{"three"}}

    for _,v := range data {
        go v.print()
    }

    time.Sleep(3 * time.Second)
    //goroutines print: three, three, three
}
```

Works:

```
package main

import (
    "fmt"
    "time"
)

type field struct {
    name string
}

func (p *field) print() {
    fmt.Println(p.name)
}

func main() {
    data := []field{{"one"},{"two"},{"three"}}

    for _,v := range data {
        v := v
        go v.print()
    }

    time.Sleep(3 * time.Second)
    //goroutines print: one, two, three
}
```

What do you think you'll see when you run this code (and why)?

```
package main

import (
    "fmt"
    "time"
)

type field struct {
    name string
}

func (p *field) print() {
    fmt.Println(p.name)
}

func main() {
    data := []*field{{"one"},{"two"},{"three"}}

    for _,v := range data {
        go v.print()
    }

    time.Sleep(3 * time.Second)
}
```

Deferred Function Call Argument Evaluation

- level: intermediate

Arguments for a deferred function call are evaluated when the `defer` statement is evaluated (not when the function is actually executing). The same rules apply when you defer a method call. The structure value is also saved along with the explicit method parameters and the closed variables.

```
package main

import "fmt"

func main() {
    var i int = 1

    defer fmt.Println("result =>",func() int { return i * 2 }())
    i++
    //prints: result => 2 (not ok if you expected 4)
}
```

If you have pointer parameters it is possible to change the values they point to because only the pointer is saved when the `defer` statement is evaluated.

```
package main

import (
  "fmt"
)

func main() {
  i := 1
  defer func (in *int) { fmt.Println("result =>", *in) }(&i)

  i = 2
  //prints: result => 2
}
```

Deferred Function Call Execution

- level: intermediate

The deferred calls are executed at the end of the containing function (and in reverse order) and not at the end of the containing code block. It's an easy mistake to make for new Go developers confusing the deferred code execution rules with the variable scoping rules. It can become a problem if you have a long running function with a `for` loop that tries to `defer` resource cleanup calls in each iteration.

```
package main

import (
    "fmt"
    "os"
    "path/filepath"
)

func main() {
    if len(os.Args) != 2 {
        os.Exit(-1)
    }

    start, err := os.Stat(os.Args[1])
    if err != nil || !start.IsDir(){
        os.Exit(-1)
    }

    var targets []string
    filepath.Walk(os.Args[1], func(fpath string, fi os.FileInfo, err error) error {
        if err != nil {
            return err
        }

        if !fi.Mode().IsRegular() {
            return nil
        }

        targets = append(targets,fpath)
        return nil
    })

    for _,target := range targets {
        f, err := os.Open(target)
        if err != nil {
            fmt.Println("bad target:",target,"error:",err) //prints error: too many open files
            break
        }
        defer f.Close() //will not be closed at the end of this code block
        //do something with the file...
    }
}
```

One way to solve the problem is by wrapping the code block in a function.

```
package main

import (
    "fmt"
    "os"
    "path/filepath"
)

func main() {
    if len(os.Args) != 2 {
        os.Exit(-1)
    }

    start, err := os.Stat(os.Args[1])
    if err != nil || !start.IsDir(){
        os.Exit(-1)
    }

    var targets []string
    filepath.Walk(os.Args[1], func(fpath string, fi os.FileInfo, err error) error {
        if err != nil {
            return err
        }

        if !fi.Mode().IsRegular() {
            return nil
        }

        targets = append(targets,fpath)
        return nil
    })

    for _,target := range targets {
        func() {
            f, err := os.Open(target)
            if err != nil {
                fmt.Println("bad target:",target,"error:",err)
                return
            }
            defer f.Close() //ok
            //do something with the file...
        }()
    }
}
```

Another option is to get rid of the `defer` statement :-)

Failed Type Assertions

- level: intermediate

Failed type assertions return the "zero value" for the target type used in the assertion statement. This can lead to unexpected behavior when it's mixed with variable shadowing.

Incorrect:

```
package main

import "fmt"

func main() {
    var data interface{} = "great"

    if data, ok := data.(int); ok {
        fmt.Println("[is an int] value =>",data)
    } else {
        fmt.Println("[not an int] value =>",data)
        //prints: [not an int] value => 0 (not "great")
    }
}
```

Works:

```
package main

import "fmt"

func main() {
    var data interface{} = "great"

    if res, ok := data.(int); ok {
        fmt.Println("[is an int] value =>",res)
    } else {
        fmt.Println("[not an int] value =>",data)
        //prints: [not an int] value => great (as expected)
    }
}
```

Blocked Goroutines and Resource Leaks

- level: intermediate

Rob Pike talked about a number of fundamental concurrency patterns in his"Go Concurrency Patterns" presentation at Google I/O in 2012. Fetching the first result from a number of targets is one of them.

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result)
    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```

The function starts a goroutines for each search replica. Each goroutine sends its search result to the result channel. The first value from the result channel is returned.

What about the results from the other goroutines? What about the goroutines themselves?

The result channel in the `First()` function is unbuffered. This means that only the first goroutine returns. All other goroutines are stuck trying to send their results. This means if you have more than one replica each call will leak resources.

To avoid the leaks you need to make sure all goroutines exit. One potential solution is to use a buffered result channel big enough to hold all results.

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result,len(replicas))
    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```

Another potential solution is to use a `select` statement with a `default` case and a buffered result channel that can hold one value. The `default` case ensures that the goroutines don't get stuck even when the result channel can't receive messages.

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result,1)
    searchReplica := func(i int) {
        select {
        case c <- replicas[i](query):
        default:
        }
    }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```

You can also use a special cancellation channel to interrupt the workers.

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result)
    done := make(chan struct{})
    defer close(done)
    searchReplica := func(i int) {
        select {
        case c <- replicas[i](query):
        case <- done:
        }
    }
    for i := range replicas {
        go searchReplica(i)
    }

    return <-c
}
```

Why did the presentation contain these bugs? Rob Pike simply didn't want to comlicate the slides. It makes sense, but it can be a problem for new Go developers who would use the code as is without thinking that it might have problems.

Same Address for Different Zero-sized Variables

- level: intermediate

If you have two different variables shouldn't they have different addresses? Well, it's not the case with Go :-

) If you have zero-sized variables they might share the exact same address in memory.

```
package main

import (
  "fmt"
)

type data struct {
}

func main() {
  a := &data{}
  b := &data{}

  if a == b {
    fmt.Printf("same address - a=%p b=%p\n",a,b)
    //prints: same address - a=0x1953e4 b=0x1953e4
  }
}
```

The First Use of iota Doesn't Always Start with Zero

- level: intermediate

It may seem like the `iota` identifyer is like an increment operator. You start a new constant declaration and the first time you use `iota` you get zero, the second time you use it you get one and so on. It's not always the case though.

```
package main

import (
  "fmt"
)

const (
  azero = iota
  aone  = iota
)

const (
  info  = "processing"
  bzero = iota
  bone  = iota
)

func main() {
  fmt.Println(azero,aone) //prints: 0 1
  fmt.Println(bzero,bone) //prints: 1 2
}
```

The `iota` is really an index operator for the current line in the constant declaration block, so if the first use of `iota` is not the first line in the constant declaration block the initial value will not be zero.

Using Pointer Receiver Methods On Value Instances

- level: advanced

It's OK to call a pointer receiver method on a value as long as the value is addressable. In other words, you

don't need to have a value receiver version of the method in some cases.

Not every variable is addressable though. Map elements are not addressable. Variables referenced through interfaces are also not addressable.

```
package main

import "fmt"

type data struct {
    name string
}

func (p *data) print() {
    fmt.Println("name:",p.name)
}

type printer interface {
    print()
}

func main() {
    d1 := data{"one"}
    d1.print() //ok

    var in printer = data{"two"} //error
    in.print()

    m := map[string]data {"x":data{"three"}}
    m["x"].print() //error
}
```

Compile Errors:

> */tmp/sandbox017696142/main.go:21: cannot use data literal (type data) as type printer in assignment: data does not implement printer (print method has pointer receiver)*
> */tmp/sandbox017696142/main.go:25: cannot call pointer method on m["x"]*
> */tmp/sandbox017696142/main.go:25: cannot take the address of m["x"]*

Updating Map Value Fields

- level: advanced

If you have a map of struct values you can't update individual struct fields.

Fails:

```
package main

type data struct {
    name string
}

func main() {
    m := map[string]data {"x":{"one"}}
    m["x"].name = "two" //error
}
```

Compile Error:

> */tmp/sandbox380452744/main.go:9: cannot assign to m["x"].name*

It doesn't work because map elements are not addressable.

What can be extra confusing for new Go devs is the fact that slice elements are addressable.

```
package main

import "fmt"

type data struct {
    name string
}

func main() {
    s := []data {{"one"}}
    s[0].name = "two" //ok
    fmt.Println(s)    //prints: [{two}]
}
```

Note that a while ago it was possible to update map element fields in one of the Go compilers (gccgo), but that behavior was quickly fixed :–) It was also considered as a potential feature for Go 1.3. It wasn't important enough to support at that point in time, so it's still on the todo list.

The first work around is to use a temporary variable.

```
package main

import "fmt"

type data struct {
    name string
}

func main() {
    m := map[string]data {"x":{"one"}}
    r := m["x"]
    r.name = "two"
    m["x"] = r
    fmt.Printf("%v",m) //prints: map[x:{two}]
}
```

Another workaround is to use a map of pointers.

```
package main

import "fmt"

type data struct {
    name string
}

func main() {
    m := map[string]*data {"x":{"one"}}
    m["x"].name = "two" //ok
    fmt.Println(m["x"]) //prints: &{two}
}
```

By the way, what happens when you run this code?

```
package main

type data struct {
    name string
}

func main() {
    m := map[string]*data {"x":{"one"}}
    m["z"].name = "what?" //???
}
```

"nil" Interfaces and "nil" Interfaces Values

- level: advanced

This is the second most common gotcha in Go because interfaces are not pointers even though they may look like pointers. Interface variables will be "nil" only when their type and value fields are "nil".

The interface type and value fields are populated based on the type and value of the variable used to create the corresponding interface variable. This can lead to unexpected behavior when you are trying to check if an interface variable equals to "nil".

```
package main

import "fmt"

func main() {
    var data *byte
    var in interface{}

    fmt.Println(data,data == nil) //prints: <nil> true
    fmt.Println(in,in == nil)     //prints: <nil> true

    in = data
    fmt.Println(in,in == nil)     //prints: <nil> false
    //'data' is 'nil', but 'in' is not 'nil'
}
```

Watch out for this trap when you have a function that returns interfaces.

Incorrect:

```
package main

import "fmt"

func main() {
    doit := func(arg int) interface{} {
        var result *struct{} = nil

        if(arg > 0) {
            result = &struct{}{}
        }

        return result
    }

    if res := doit(-1); res != nil {
        fmt.Println("good result:",res) //prints: good result: <nil>
        //'res' is not 'nil', but its value is 'nil'
    }
}
```

Works:

```
package main

import "fmt"

func main() {
    doit := func(arg int) interface{} {
        var result *struct{} = nil

        if(arg > 0) {
            result = &struct{}{}
        } else {
            return nil //return an explicit 'nil'
        }

        return result
    }

    if res := doit(-1); res != nil {
        fmt.Println("good result:",res)
    } else {
        fmt.Println("bad result (res is nil)") //here as expected
    }
}
```

Stack and Heap Variables

- level: advanced

You don't always know if your variable is allocated on the stack or heap. In C++ creating variables using the `new` operator always means that you have a heap variable. In Go the compiler decides where the variable will be allocated even if the `new()` or `make()` functions are used. The compiler picks the location to store the variable based on its size and the result of "escape analysis". This also means that it's ok to return references to local variables, which is not ok in other languages like C or C++.

If you need to know where your variables are allocated pass the "-m" gc flag to "go build" or "go run" (e.g., `go run -gcflags -m app.go`).

GOMAXPROCS, Concurrency, and Parallelism

- level: advanced

Go 1.4 and below uses only one execution context / OS thread. This means that only one goroutine can execute at any given time. Starting with 1.5 Go sets the number of execution contexts to the number of logical CPU cores returned by `runtime.NumCPU()` . That number may or may not match the total number of logical CPU cores on your system depending on the CPU affinity settings of your process. You can adjust this number by changing the `GOMAXPROCS` environment variable or by calling the `runtime.GOMAXPROCS()` function.

There's a common misconception that `GOMAXPROCS` represents the number of CPUs Go will use to run goroutines. The `runtime.GOMAXPROCS()` function documentation adds more to the confusion. The `GOMAXPROCS` variable description (https://golang.org/pkg/runtime/) does a better job talking about OS threads.

You can set `GOMAXPROCS` to more than the number of your CPUs. As of 1.10 there's no longer a limit for GOMAXPROCS. The max value for `GOMAXPROCS` used to be 256 and it was later increased to 1024 in 1.9.

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Println(runtime.GOMAXPROCS(-1)) //prints: X (1 on play.golang.org)
    fmt.Println(runtime.NumCPU())       //prints: X (1 on play.golang.org)
    runtime.GOMAXPROCS(20)
    fmt.Println(runtime.GOMAXPROCS(-1)) //prints: 20
    runtime.GOMAXPROCS(300)
    fmt.Println(runtime.GOMAXPROCS(-1)) //prints: 256
}
```

Read and Write Operation Reordering

- level: advanced

Go may reorder some operations, but it ensures that the overall behavior in the goroutine where it happens doesn't change. However, it doesn't guarantee the order of execution across multiple goroutines.

```
package main

import (
    "runtime"
    "time"
)

var _ = runtime.GOMAXPROCS(3)

var a, b int

func u1() {
    a = 1
    b = 2
}

func u2() {
    a = 3
    b = 4
}

func p() {
    println(a)
    println(b)
}

func main() {
    go u1()
    go u2()
    go p()
    time.Sleep(1 * time.Second)
}
```

If you run this code a few times you might see these `a` and `b` variable combinations:

> *1*
> *2*
>
> *3*
> *4*
>
> *0*
> *2*
>
> *0*
> *0*
>
> *1*
> *4*

The most interesting combination for `a` and `b` is "02". It shows that `b` was updated before `a` .

If you need to preserve the order of read and write operations across multiple goroutines you'll need to use channels or the appropriate constructs from the "sync" package.

Preemptive Scheduling

- level: advanced

It's possible to have a rogue goroutine that prevents other goroutines from running. It can happen if you

have a `for` loop that doesn't allow the scheduler to run.

```
package main

import "fmt"

func main() {
    done := false

    go func(){
        done = true
    }()

    for !done {
    }
    fmt.Println("done!")
}
```

The `for` loop doesn't have to be empty. It'll be a problem as long as it contains code that doesn't trigger the scheduler execution.

The scheduler will run after GC, "go" statements, blocking channel operations, blocking system calls, and lock operations. It may also run when a non-inlined function is called.

```
package main

import "fmt"

func main() {
    done := false

    go func(){
        done = true
    }()

    for !done {
        fmt.Println("not done!") //not inlined
    }
    fmt.Println("done!")
}
```

To find out if the function you call in the `for` loop is inlined pass the "-m" gc flag to "go build" or "go run" (e.g., `go build -gcflags -m`).

Another option is to invoke the scheduler explicitly. You can do it with the `Gosched()` function from the "runtime" package.

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    done := false

    go func(){
        done = true
    }()

    for !done {
        runtime.Gosched()
    }
    fmt.Println("done!")
}
```

Note that the code above contains a race condition. It was done intentionally to show the sheduling gotcha.

Import C and Multiline Import Blocks

- level: Cgo

You need to import the "C" package to use Cgo. You can do that with a single line `import` or you can do it with an `import` block.

```
package main

/*
#include <stdlib.h>
*/
import (
  "C"
)

import (
  "unsafe"
)

func main() {
  cs := C.CString("my go string")
  C.free(unsafe.Pointer(cs))
}
```

If you are using the `import` block format you can't import other packages in the same block.

```
package main

/*
#include <stdlib.h>
*/
import (
  "C"
  "unsafe"
)

func main() {
  cs := C.CString("my go string")
  C.free(unsafe.Pointer(cs))
}
```

Compile Error:

> *./main.go:13:2: could not determine kind of name for C.free*

No blank lines Between Import C and Cgo Comments

- level: Cgo

One of the first gotchas with Cgo is the location of the cgo comments above the `import "C"` statement.

```
package main

/*
#include <stdlib.h>
*/

import "C"

import (
  "unsafe"
)

func main() {
  cs := C.CString("my go string")
  C.free(unsafe.Pointer(cs))
}
```

Compile Error:

> *./main.go:15:2: could not determine kind of name for C.free*

Make sure you don't have any blank lines above `import "C"` statement.

Can't Call C Functions with Variable Arguments

- level: Cgo

You can't call C functions with variable arguments directly.

```
package main

/*
#include <stdio.h>
#include <stdlib.h>
*/
import "C"

import (
  "unsafe"
)

func main() {
  cstr := C.CString("go")
  C.printf("%s\n",cstr) //not ok
  C.free(unsafe.Pointer(cstr))
}
```

Compile Error:

> *./main.go:15:2: unexpected type: ...*

You have to wrap your variadic C functions in functions with a known number of parameters.

```
package main

/*
#include <stdio.h>
#include <stdlib.h>

void out(char* in) {
  printf("%s\n", in);
}
*/
import "C"

import (
  "unsafe"
)

func main() {
  cstr := C.CString("go")
  C.out(cstr) //ok
  C.free(unsafe.Pointer(cstr))
}
```

Comments and Discussions

Old Reddit discussion.

Latest Hacker News discussion.

Thank you for your feedback and suggestions!

Kyle Quest
Builder, breaker, and defender. Cloud, security, and Big Data.
Twitter LinkedIn