Practical Go: Real world advice for writing maintainable Go programs

Dave Cheney – dave@cheney.net – Version 1b5883b, 2020-01-27

Table of Contents

Preface

Hello,

Thank you for joining me today. The story of how this workshop came about needs some exposition.

A few years ago I made a tweet called "three rules for dating my code base" (https://twitter.com/davecheney/status/1013978233925033984?lang=en), it was intended to be a short list of the things you should do for a healthy Go project. Tweet sizes being what they were at the time, this tweet was followed a day later with another three things (https://twitter.com/davecheney/status/1013979669392052224), I suspect I could have continued for the rest of the year.

Related to this, in 2015 I had dabbled with an agreement to write a book for O'Reilly. The project fizzled out after a few months for two reasons:

1. O'Reilly's document preparation system was good, but not as flexible as I wanted. There were too many undefined steps between my words and the final book.
2. The unforgiving calculus of the number of hours a project like this would take meant if I failed to find the hundreds of hours necessary there would be no partial credit.

Both of these struck me as rather old world approaches to developing a book and the distinctly non-agile way that a complete manuscript was copy edited was not lost on me.

After the *three rules* tweets I began to think again about how I would like to see a book about Go written in a way that delivered continual benefit even if I were to drop out at an arbitrary point.

This workshop, the document you are reading, the articles on my blog (https://dave.cheney.net/practical-go), and the presentations I give are my answer to this problem. This document represents the best version of the material that exists today. As I present more, give more workshops, the material will grow, however, Go programmers continually reap the benefits of this material without waiting for someone to declare arbitrarily that Practical Go is *done*.

Today this is a workshop style presentation, I'm going to dispense with the usual slide deck and we'll work directly from this document which you can take away with you today.

TIP

You can find the latest version of this presentation at

http://bit.ly/gcil2020

License

Introduction

There are (probably) no correct answers

100 years ago David Hilbert was deeply unhappy with the state of mathematics. Mathematics was beset with inconsistencies and paradoxes. Seeking closure the German mathematican embarked on what became know as *Hilbert's Program*, an attempt to reduce mathematics to a set of axiomatic proofs. From those foundations, mathematics could be reconstructed. Incontravertbily, provably, correct.

Sadly for Hilbert and his compatriots, their work was undone in 1931 when Kurt Gödel published his incompleteness theorum. Gödel demonstrated that, parodoxically, any system powerful enough to formalise mathematics would be unable to resolve its own inconsistences. Yet as fundamentally flawed as mathematics is, it has permitted us to understand the universe and the space between atoms.

A few years ago I met a man who told me that the Wright Brothers built the first successful aeroplane years before the laws of hydrodynamics were fully understood. They weren't the first to fly, people had parachuted from balloons or glided from cliff tops, but they were the first to transition from ground to airborne.

This was 1903 and the theories of aerodynamics wouldn't be developed for another 60 years, yet they still flew. How did they do this? How did the brothers' succeed in flight without an understanding of the physics that underpins the principle of lift?

Wilbur and Orville didn't understand the science of why they flew, but they did learn from previous attempts. Their success was based on the experience gained from all those who had failed before them. Said another way, the Kitty Hawk flew because the Wrights were the first to *fail* to build an airplane which wouldn't fly.

My friend asserted that just because we don't understand the laws that underlie the development of software that doesn't mean they don't exist and, perhaps more importantly, the search for those theoretical underpinnings should not preclude an experimental approach.

Some argue that computer science is not a science. Alan Kay famously called it a pop culture. And this was my friends point. It's not that computer science is a pop culture, or sociology, it's that the *practice* of programming is running ahead of the *science*. We may be reasonably certain that a practice works, but we won't know *why* until later.

So, I put it to you that the period of development of the *practice* of computer programming is running ahead of the scientific foundations on which is will ultimately be based. In a few hundred years we may have a solid scientific basis for computer programming, but right now all we have is empirical evidence.

Knowing when to stick to the code and when to bend the rules probably comes down to experience, gained from observation, experimentation, and research. What's the difference between a rule and a guideline? I'd say, experience. The mastery of a topic, Malcolm Gladwell's 10,000 hour yardstick, is equal parts observation and practice. You watch someone else do the thing, while concurrently you practice doing the thing.

I love writing software. I love discovering the patterns and structure that underlies design problems, but I do not pretend to represent the truth about the right way to program. Such a thing may not be knowable, but that should not preclude us from having a discussion based on experience.

With that said, these are my experiences, and the goal today is not to be prescriptive. This is a discussion, not a lecture.

A guiding principle

If I'm going to talk about best practices for a programming language I need some framework by which to define what I mean by *best*.

Over the years that I've been thinking about this material I've tried to identify a common theme, and abstract that can summarise the tens of thousands of words I've written.

In forming this abstract many words have come and gone; simplicity, readability, clarity, productivity, but ultimately they are all synonyms for one word — m_aintainabilty_.

> *Design is the art of arranging code to work today, and be changable forever.*
>
> — *Sandi Metz*

If there was a quote that summarises the ethos of writing Pratical Go code it would be this quote from Sandi Metz.

1. Declarations

> *There are only two hard things in Computer Science: cache invalidation and naming things.*
>
> — *Phil Karlton*

> *Bad names are indicative of bad designs*
>
> — *Josh Bloch*

The first topic I'd like to discuss is *declarations*. A declaration declares an *identifier*. An identifier is a fancy word for a *name*; the name of a variable, the name of a function, the name of a method, the name of a type, the name of a package, and so on.

> *Poor naming is symptomatic of poor design.*
>
> — *Dave Cheney (https://twitter.com/davecheney/status/997150760344305665)*

Names are important. Given the limited syntax of Go, the names we choose for things in our programs have an oversized impact on the readability of our programs. Readability is a defining quality of maintainable code, thus choosing good names is crucial to the maintainability of Go code.

1.1. Choose identifiers for clarity, not brevity

> *Obvious code is important. What you can do in one line you should do in three.*
>
> — Ukiah Smith (https://twitter.com/UkiahSmith/status/1044931395112644608)

Go is not a language that optimises for clever one liners. Go is not a language which optimises for the least number of lines in a program. We're not optimising for the size of the source code on disk, nor how long it takes to type the program into an editor.

> *Good naming is like a good joke. If you have to explain it, it's not funny.*
>
> — Dave Cheney (https://twitter.com/davecheney/status/997155238929842176)

Rather, we want to optimise our code to be clear to the reader. Key to this clarity is the names we choose for identifiers in Go programs. To get technical, when I'm talking about naming, I'm talking about naming *identifiers* in Go programs. But that's a bit lengthy, so lets just call it naming from now on — you understand what I mean.

First, lets set the ground rules. Anything in Go that is an *identifier* has a name. What are the things in Go we can name?

- the name of a type, struct, or interface
- the name of a function or a method
- the name of a package
- the name of a constant
- the name of a variable, formal parameter, or return value

Go programmers care about the name of things—a lot. Naming is key to readability, hence the names of identifiers used in your program is critical. Poorly chosen names contribute to a program that is harder to comprehend thus harder to maintain.

Although not defined by `go fmt`, the canonical Go style for naming things descends from its original authors and can be identified by the following observations:

1. The greater the distance between declaration and use, the more descriptive the name.
2. The more frequently the name is referenced in a particular place, the shorter the name.

These two observations interlock to form general advice. For example, the names of local variables should be shorter than the names of the formal parameters.

Let's talk about the qualities of a good name:

A good name is concise

A good name need not be the shortest possible, but a good name should waste no space on things which are extraneous. Good names have a high signal to noise ratio.

A good name is descriptive

A good name should describe the application of a variable or constant, *not* their contents. A good name should describe the result of a function, or behaviour of a method, *not* their implementation. A good name should describe the purpose of a package, *not* its contents. The better the choice of a name, the more accurately it describes that that it identifies.

A good name should be predictable

You should be able to infer the way a symbol will be used from its name alone. This is a function of choosing descriptive names, but it also about following tradition. This is what Go programmers talk about when they say *idiomatic* (of which I shall have more to say tomorrow).

Let’s talk about each of these properties in depth.

> *If you don’t know what a thing should be called, you cannot know what it is. If you don’t know what it is, you cannot sit down and write the code.*
>
> — *Sam Gardiner[gardiner]*

NOTE

Line lengths

Go traditionally enforces no line length restriction. With that said, things considered, shorter identifiers are preferable to longer ones. If you find yourself arguing with your colleagues over wrapping a particularly long function signature, you may be able to avoid the argument by reducing the length identifiers in your program.

1.1.1. The larger the identifier’s scope, the larger it’s name

Sometimes people criticise the Go style for recommending short variable names. As Rob Pike said, "Go programmers want the *right* length identifiers". [1]

The length of the identifier should be proportional to the distance between its declaration and use.

> *The greater the distance between a name’s declaration and its uses, the longer the name should be.*
>
> — *Andrew Gerrand [1]*

Andrew Gerrand suggests that by using longer identifies to indicate to the reader things of higher importance.

From this we can draw some guidelines:

- Short variable names work well when the distance between their declaration and *last* use is short. Short functions can have short identifiers.
- Long variable names need to justify themselves; the longer they are the more value they need to provide. Lengthy bureaucratic names carry a low amount of signal compared to their weight on the page. Long functions shouldn’t have short parameter names as they will be declared a long way from where they are used.
- Don’t include the name of your type in the name of your variable. It’s needless ceremony; saying repeating the type of the variable constantly does not make it more type safe.
- Constants should describe the value they hold, *not* how that value is used.
- Prefer single words for method, interface, and package identifiers.
- Prefer single letter variables for loops and branches, single words for parameters and return values, multiple words for functions and package level declarations. Its ok to use a shorter name in a short block inside a larger function.
- Remember that the name of a package is part of the name the caller uses to to refer to it, so make use of that.
- Unless you’re embedding, the name of the field should describe its purpose, not its content.
- Globals of all kinds deserve longer identifiers than locally scoped ones.

Let's look at an example:

GO

```
type Person struct {
    Name string
    Age  int
}

// AverageAge returns the average age of people.
func AverageAge(people []Person) int {
    if len(people) == 0 {
        return 0
    }

    var count, sum int
    for _, p := range people {
        sum += p.Age
        count += 1
    }

    return sum / count
}
```

In this example, the range variable `p` is declared on line 10 and only referenced once, on the following line. `p` lives for a very short time on the page and in limited scope during the execution of the function. A reader who is interested in the effect values of `p` have on the program need only read the loop's three lines.

By comparison `people` is declared in the function parameters, is live for the body of the function, and is referenced three times over seven lines. The same is true for `sum`, and `count`, thus they justify their longer names. The reader has to scan a wider number of lines to locate them so they are given more distinctive names.

I could have chosen `s` for `sum` and `c` (or possibly `n`) for `count` but this would have reduced all the variables in the program to the same level of importance. I could have chosen `p` instead of `people` but that would have left the problem of what to call the `for … range` iteration variable and the singular `person` would look odd as the loop iteration variable which lives for little time has a longer name than the slice of values it was derived from.

TIP

Use blank lines to break up the flow of a function in the same way you use paragraphs to break up the flow of a document. In `AverageAge` we have three operations occurring in sequence. The first is the precondition, checking that we don't divide by zero if people is empty, the second is the accumulation of the sum and count, and the final is the computation of the average.

1.1.2. Context is key

Therefore, one lesson to be gleaned from this solution is that you should name methods after the concept they represent rather than how they currently behave.

— *Sandi Metz and Katrina Owen*
99 Bottles of OOP

It's important to recognise that most advice on naming is contextual. I like to say it is a guideline, not a rule.

What is the difference between two identifiers, `i`, and `index`. We cannot say conclusively that one is better than another in all situations. For example is

GO

```go
for index := 0; index < len(s); index++ {
	//
}
```

fundamentally more readable than

GO

```go
for i := 0; i < len(s); i++ {
	//
}
```

I argue it is not, because it is likely the scope of `i` , and `index` for that matter, is limited to the body of the `for` loop and the extra verbosity of the latter adds little to *comprehension* of the program. A loop nested within a loop is inherently harder to comprehend reguardless of the name of the loop induction variable.

TIP If you found yourself with so many nested loops that you exhaust your supply of `i` , `j` , and `k` variables, its probably time to break your function into smaller ones.

However, which of these functions is more readable?

GO

```go
func (s *SNMP) Fetch(oid []int, index int) (int, error)
```

versus

GO

```go
func (s *SNMP) Fetch(o []int, i int) (int, error)
```

In this example, `oid` is an abbreviation for SNMP Object ID, so shortening it to `o` would mean programmers have to translate from the common notation that they read in documentation to the shorter notation in your code. Similarly, reducing `index` to `i` obscures what `i` stands for; in SNMP messages a sub value of each OID is called an Index.

TIP Avoid mixing formal parameters of different lengths in the same declaration.

1.1.3. Use a predictable naming style

Choose variable names that won't be confused

— *Elements of programming style*
Kernighan and Plauger

Another property of a good name is it should be predictable. The reader should be able to predict the use of a name when they encounter it for the first time. When they encounter a *common* name, they should be able to assume it has not changed meanings since the last time they saw it. You could say that a good name should feel familiar.

For example, if your code passes around a database handle rather than a combination of `d *sql.DB` , `dbase *sql.DB` , `DB *sql.DB` , and `database *sql.DB` , consolidate on something like;

GO

```go
var db *sql.DB
```

and use it consistently across parameters, return values, local declarations, and potentially receivers. Doing so promotes familiarity; if you see a `db`, you know it's a `*sql.DB` and that it has either been declared locally or provided for you by the caller.

Similar advice applies to method receivers; use the same receiver name every method on that type. This makes it easier for the reader to internalise the use of the receiver across the methods of that type which may, occasionally, be defined across multiple files.

NOTE The convention for short receiver names in Go is at odds with the advice provided so far. This is just one of the choices made early on that has become the preferred style, just like the use of `CamelCase` rather than `snake_case`.

Finally, certain single letter variables have traditionally been associated with specific use cases. For example;

- `i`, `j`, and `k` are commonly the loop induction variable for simple `for` loops. [2]
- `n` is commonly associated with a counter or accumulator.
- `v` is a common shorthand for a value in a generic encoding function, `k` is commonly used for the key of a map.
- `a` and `b` are generic names for parameters comparing two variables of the same type.
- `x` and `y` are generic names for local variables created for comparision, and `s` is often used as shorthand for parameters of type `string`.
- Functions or methods that being with `Print` are traditionaly take `string`s, or things that can be converted to `string`s and print them as text.
- The `f` suffix; `Printf`, `Logf`, etc, indicate the function takes a format string and a variable number of arguments to format according to the rule.
- Functions or methods that begin with `Write` traditonally take *non* `string` values and write them out as binary data.
- Collection variables, maps, slices, and arrays, should be pluralised.

TIP Go style dictates that receivers have a single letter name, or acronyms derived from their type. You may find that the name of your receiver sometimes conflicts with name of a parameter in a method. In this case, consider making the parameter name slightly longer, and don't forget to use this new parameter name consistently.

As with the `db` example above programmers *expect* `i` to be a loop induction variable. If you ensure that `i` is *always* a loop variable, not used in other contexts outside a `for` loop. When readers encounter a variable called `i`, or `j`, they know that a loop is close by.

1.2. A variable's name should describe its contents

You should avoid naming your variables after their types for the same reason you don't name your pets "dog" and "cat". You shouldn't include the name of your type in your variable's name for the same reason.

Consider this example:

GO
```
var usersMap map[string]*User
```

What's good about this declaration? We can see that its a map, and it has something to do with the `*User` type, that's probably good. But `usersMap` *is* a map, and Go being a statically typed language won't let us accidentally use it where a different type is required. The `Map` suffix is redundant from the point of view of the compiler. Hence utility of the suffix is entirely down to whether we can prove it is of use to the reader.

Now, consider what happens if we were to declare other variables:

GO
```go
var (
        companiesMap map[string]*Company
        productsMap  map[string]*Products
)
```

Now we have three map type variables in scope, `usersMap`, `companiesMap`, and `productsMap`, all mapping strings to different types. We know they are maps; it's right there in their declaration. We also know that their map declarations prevent us from using one in place of another—the compiler will throw an error if we try to use `companiesMap` where code was expecting a `map[string]*User`. In this situation it's clear that the `Map` suffix does not improve the clarity of the code, its just extra boilerplate to type.

Removing the suffix leaves us with the more concise and equally descriptive:

GO
```go
var (
        users     map[string]*User
        companies map[string]*Company
        products  map[string]*Products
)
```

NOTE

usersMap versus userMap

If we remove the suffix denoting the type's type from its name, `usersMap` becomes `users` which is descriptive, but `userMap` would become `user`, which is misleading.

If `users` isn't descriptive enough, then `usersMap` won't be either.

My suggestion is to avoid any suffix that resembles the type of the variable. This advice also applies to function parameters. For example:

GO
```go
type Config struct {
    //
}

func WriteConfig(w io.Writer, config *Config)
```

Naming the `*Config` parameter `config` is redundant. We know its a `*Config`, it says so right there.

In this case consider `conf`, or maybe `c`, if the lifetime of the variable is short enough. If there is more that one `*Config` in scope at any one time then calling them `conf1` and `conf2` is less descriptive than calling them `original` and `updated` as the latter are less likely to be mistaken for one another.

TIP

Don't let package names steal good variable names.

The name of an imported identifier includes its package name. For example the `Context` type in the `context` package will be known as `context.Context`. This makes it impossible to use `context` as a variable or type in your package.

```go
func WriteLog(context context.Context, message string)
```

Will not compile. This is why the local declaration for `context.Context` types is traditionally `ctx`. eg.

```go
func WriteLog(ctx context.Context, message string)
```

The name of the variable should describe its contents, not the *type* of its contents.

1.3. Use a consistent declaration style

Where I live we have three levels of government; local, state and federal. It is universally accepted that this is one too many, however consensus on which level to eliminate is lacking. In much the same way, Go has at least six different ways to declare a variable:

- `x := 1`
- `var y = 2`
- `var z int = 3`
- `var a int; a = 4`
- `var b = int(5)`
- `c := int(6)`

This list does not include receivers, formal parameters and named return values. I'm sure there are more that I haven't thought of.

This is something that Go's designers recognise was probably a mistake, but its too late to change it now, and, they argue, the bigger problem is shadowing. With all these different ways of declaring a variable, how do we avoid each Go programmer choosing their own style?

In Go each variable has a purpose because each variable we declare has to be used within the same scope. Due to Go's automatic type deduction it is uncommon to declare a type *and* initialise its value; you either do one or the other, declare without initialisation, or assign without initalisation. Here is a suggestion for how to make the purpose of each declaration clear to the reader. This is the style I try to use where possible.

When declaring, but not initialising, a variable, use `var`

When declaring a variable that will be explicitly initialised later, use the `var` keyword.

GO
```go
var players int    // 0

var things []Thing // an empty slice of Things

var thing Thing    // empty Thing struct
json.Unmarshall(reader, &thing)
```

The `var` acts as a clue to say that this variable has been *deliberately* declared as the zero value of the indicated type. You should use this form only when declaring variables that you want to be explicitly initalised to the type's zero value.

This is advice consistent with the requirement to declare variables at the package level using `var` as opposed to the short declaration syntax. However, I'll argue later that you shouldn't be using package level variables at all.

When declaring and initialising, use `:=`

When declaring *and* initialising a variable—that is to say we're not letting the variable be implicitly initialised to its zero value—I recommend using the short variable declaration form. For example;

GO
```go
num := rand.Int()
```

The lack of `var` prefix is a signal that this variable has been *explicitly* initalised. This makes it clear to the reader that the variable on the left hand side of the `:=` is being deliberately initialised from the expression on the right.

I've also found that by avoiding declaring the type of the variable, instead infering it from the right hand side of the assignment, this makes re-factoring easier in the future.

To explain why, Let's look at the previous example, but this time deliberately initialising each variable:

GO
```go
var players int = 0

var things []Thing = nil

var thing *Thing = new(Thing)
json.Unmarshall(reader, thing)
```

In the first and third examples, because in Go there are no automatic conversions from one type to another; the type on the left hand side of the assignment operator *must* be identical to the type on the right hand side. The compiler can infer the type of the variable being declared from the type on the right hand side, to the example can be written more concisely like this:

GO
```go
var players = 0

var things []Thing = nil

var thing = new(Thing)
json.Unmarshall(reader, thing)
```

This leaves us with explicitly initialising `players` to `0` which is redundant because `0` *is* `players'` zero value. So it's better to make it clear that we're going to use the zero value by instead writing

GO
```go
var players int
```

What about the second statement? We cannot elide the type and write

```go
var things = nil
```

Because `nil` does not have a type. [3] Instead we have a choice, do we want the zero value for a slice?

```go
var things []Thing
```

or do we want to create a slice with zero elements?

```go
var things = make([]Thing, 0)
```

If we wanted the latter then this is *not* the zero value for a slice so we should make it clear to the reader that we're making this choice by using the short declaration form:

```go
things := make([]Thing, 0)
```

Which tells the reader that we have chosen to initialise `things` explicitly.

This brings us to the third declaration,

```go
var thing = new(Thing)
```

Which is both explicitly initialising a variable and introduces the uncommon use of the `new` keyword which some Go programmer dislike. If we apply our short declaration syntax recommendation then the statement becomes

```go
thing := new(Thing)
```

Which makes it clear that `thing` is explicitly initialised to the result of the expression `new(Thing)` --a pointer to a `Thing` --but still leaves us with the unusual use of `new`. We could address this by using the *compact literal* struct initialiser form,

```go
thing := &Thing{}
```

Which does the same as `new(Thing)`, hence why some Go programmers are upset by the duplication. However this means we're explicitly initialising `thing` with a pointer to the literal `Thing{}`, which itself is the zero value for a `Thing`.

Instead we should recognise that `thing` is being declared as its zero value and use the address of operator to pass the address of `thing` to `json.Unmarshall`

```go
var thing Thing
json.Unmarshall(reader, &thing)
```

NOTE

Exceptions make the rule

Of course, with any rule of thumb, there are exceptions. For example, sometimes two variables are closely related so writing

GO
```go
var min int
max := 1000
```

would look odd. The declaration may be more readable like this

GO
```go
min, max := 0, 1000
```

However, maybe in this case `min` and `max` are really constants, and should be written as:

GO
```go
const (
    min = 0
    max = 1000
)
```

The clue is that `min` can be substituted for its zero value whereas max cannot without explicit initalisation.

TIP

Make tricky declarations obvious

When something is complicated, it should *look* complicated.

GO
```go
var length uint32 = 0x80
```

Here `length` may be being used with a library which requires a specific numeric type and is more explicit that `length` is being explicitly declared to be `uint32` than the short declaration form:

GO
```go
length := uint32(0x80)
```

In the first example I'm deliberately breaking my rule of using the `var` declaration form with an explicit initialiser. This decision to vary from my usual form is a clue to the reader that something unusual is happening.

Although, again, `length` may actually be a constant masqurading as a variable. The clue is the requirement to explicitly type the number 0x80 whereas if it were a constant it could be inferred from the calling context.

Small blocks of declarations this style may be may look mildly inconsistent, this is probably acceptable given the other advice in this chapter.

- When declaring a variable without initialisation, use the `var` syntax.
- When declaring and explicitly initialising a variable, use `:=`.

1.3.1. Compromise for consistency

Consistency is the basis of abstraction.

— *Scott meyers*

The goal of software engineering is to produce maintainable code. Therefore you will likely spend most of your career working on projects of which you are not the sole author. My advice in this situation is; follow the local style. For example, if functions in the package uses short variables throughout, do not make it inconsistent by adding one that is lengthy.

Changing styles in the middle of a file is jarring. Uniformity, even if its not your preferred approach, is more valuable for maintenance over the long run than your personal preference. The rule I try to follow is; if it fits through `go fmt` then it's usually not worth holding up a code review for.

TIP

If you want to do a renaming across a codebase, do not mix this into another change. If someone is using git bisect they don't want to wade through thousands of lines of renaming to find the code you changed as well.

This advice could also be written, "when in Rome, do as the Romans do". For example, here is a short piece of code that violates the rules of using short delcarations only for explicit declarations

GO
```go
spc := s.spanclass
size := s.elemsize
res := false
nfree := uintptr(0)
```

vs

GO
```go
var (
        spc = s.spanclass
        size = s.elemsize
        res bool
        nfree uintptr
)
```

however the overall effect is more harmonious. All four variables are declared and initalised in a block using a regular syntax, vs the inconsistent declaration and initalisation of the latter.

However there are a few places where Go programmers forgoe this advice. For example, network connections are often called `conn`:

```go
conn, err := net.Dial("tcp", "www.google.com:80")
```

because a network connection is usually live for long enough to justify a name longer than `c`. Of course, if there is more than one connection in play, rather than calling them `conn1` and `conn2`, a name that describes their respective roles is better. For example:

GO

```go
func main() {
    l, err := net.Listen("tcp", "localhost:9000")
    if err != nil {
        return err
    }
    defer l.Close()

    for {
        client, err := l.Accept()
        if err != nil {
            return err
        }
        defer client.Close()

        upstream, err := net.Dial("tcp", PROXY)
        if err != nil {
            return err
        }
        defer upstream.Close()

        go copy(client, upstream)
        copy(upstream, client)
    }
}
```

1.4. Avoid conflicts with the names of common local variables

The import statement declares a new identifier at the package level (technically the file level, but files which do not import the identifiers they need will not compile, so the distinction is mostly academic).

Consider the problems naming a package that deals with file descriptors, `fd`. `fd` would be the natural identifier for a file descriptor value returned by some hypothetical `fd.Open` function.

GO

```go
fd, err := fd.Open() // quite confusing.
```

However don't think up a convoluted package name just to retain the use of a convenient identifier.

GO

```go
Type FD uintptr
```

`FD` is a bad name for a package, you want it for the variable, it's also a bad name for a type, for the same reason.

> Don't let an import statement steal the name of a common identifier for the name of a package.

1.5. An identifier's name includes the name of its package

An identifier's name includes its package name. This means you should think about the name of your types, symbols, etc with their *qualified* name, `package.Symbol`.

A symbol's name always includes the name of it's package. A symbol's name never includes the *"full path"* of its package, so `applicationserver/v2/cache`, is just `cache`. `/apis/meta/v1` isn't the `v1` package of the `meta` package for the `api`, it's just `v1`, and potentially conflicts with all the other `v1` packages you imported.

If you find you have a lot of packages that have the same name, you place the user in the position that they're going to have to rename your imports on import. This is undesirable as the name that symbols inside the file refer to your package as, is not the same name as the package's declaration. Moving code between files is more laborious as goimports won't work for you, and now looking at the name of the package in the symbols name you're going to have to memorise the name you renamed the package too because it conflicted on import.

TIP The name of a public symbol always includes its package name. If you find yourself referring to a public symbol exclusively in the context of a single package, consider making it private.

1.5.1. Reduce repetition

```go
Ifoo foo = new(foo)
```

You may find this repetition comforting. In general gophers find it redundant. [4]

Don't name your interface type `fooInterface`, it's repetitive. The compiler knows its an interface, you don't have to continually remind it. For the same reason, don't call your interface `Ifoo`, because the `I` is shorthand for *interface* which still stutters, but adds to the cognitive load because you have to read the `I` as "interface"

A symbol's name, to its caller, includes the name of the symbols package.

```go
var buf bytes.Buffer
```

While there may be *many* Buffer implementations, in the scope of this file's imports, there is unlikely to be multiple `bytes` packages. So the name is unambigious.

Redundancy is everywhere. Here is another example. Consider these two function declarations:

```go
func lookupVirtualHost(name string)

func lookupVirtualHost(host string)
```

The former is the name parameter of a virtualhost, the latter is a nmonomic for virtualhost, which if you know it, you wouldn't need to look it up.

The name of a variable or constant is orthogonal to its type. Just as prefixes and suffixes such as

```go
type IReader interface
```

or

```go
var UsernameVar string
```

are more subtle methods of hiding a type in the name of a variable *of that type*.

1.5.2. Avoid Prefixes unless required

With the exception of its use within its own package, every public Go symbol is prefixed with the name of its package. To pick a contrived example, nothing in the the `http` package should start with `HTTP` (or a derivation thereof) because to the user of that package, everything already starts with `http.`.

Having said that, there is a growing preference for function prefixes which *modify* the operation of the function. For example, `Must` is commonly associated with a wrapper function which panics if the function it wraps is not successful.

TIP

Avoid affectations

There is no need to prefix your project with go-, just as there is no need to prefix your interface declaration with I.

1.6. Use the smallest scope possible

Scope and shadowing in Go are tightly linked. The former is considered to be a powerful tool in avoiding bugs, the latter, to the uninitiated, is a source of bugs. However the solution to both is adopting a consistent style which will highlight possible errors due to the irregularity.

The goal of scoping variables tightly is to turn the accidental use of a variable into a compile error. For example, compare:

GO

```go
if err := something(); err != nil {
    //
}
```

to

GO

```go
err := something()
if err != nil {
    //
}
```

The former restricts the scope of the binding `err` to the block from the start of the `if` statement to the closing brace. If this check was moved higher or lower in the function it will continue to compile without issue.

Compare this to the other example which requires that `err` had *not* been previously declared. If you move these four lines later in the function, it is possible that some other method *expects* `err` to be declared and will just be using assignment.

GO

```go
err = thenextfunction()
```

which will cause two compile errors.

WARNING

Be aware of shadowing.

Scoping vairables via conditional blocks is convenient, but can cause shadowing issues with named returns and nested blocks.

This is the fault of using named returns and nested blocks, but still, the author must be aware of the complications.

TIP

Declare variables close to where they are used

I suggest that the greater the distance between declaration and use, the more descriptive the name given to the declatation.

The corollary of this advice suggests that variables, and by extension functions, types, and even packages, should be arranged to avoid the creation of unnecesarily verbose names.

TIP

Use a var (…) block when declaring mixed values

GO

```
var (
    logger = stdlog.New(os.Stdout, os.Stderr, 0)
    client = newClient(*kubeconfig, *inCluster)
    ds     contour.DataSource
    g      workgroup.Group
)
```

2. Commentary

Before we move on to larger items I want to spend a little time talking about comments.

Good code has lots of comments, bad code requires lots of comments.

— *Dave Thomas and Andrew Hunt*
The Pragmatic Programmer

Comments are very important to the readability of a Go program. Each comments should do one—and only one—of three things:

1. The comment should explain *what* the thing does.
2. The comment should explain *how* the thing does what it does.
3. The comment should explain *why* the thing is the way it is.

The first form is ideal for commentary on public symbols:

GO

```
// Open opens the named file for reading.
// If successful, methods on the returned file can be used for reading.
```

The second form is ideal for commentary inside a method:

GO

```
// queue all dependant actions
var results []chan error
for _, dep := range a.Deps {
        results = append(results, execute(seen, dep))
}
```

The third form, the *why* , is unique as it does not displace the first two, but at the same time it's not a replacement for the *what*, or the *how*. The *why* style of commentary exists to explain the external factors that drove the code you read on the page. Frequently those factors rarely make sense taken out of context. The *why* style comment exists to provide that context.

GO
```
return &v2.Cluster_CommonLbConfig{
        // Disable HealthyPanicThreshold
        // See https://www.envoyproxy.io/docs/envoy/v1.9.0/intro/arch_overview/load_balancing/panic_threshold#arch-ov
        HealthyPanicThreshold: &envoy_type.Percent{
                Value: 0,
        },
}
```

In this example it may not be immediately clear what the effect of setting `HealthyPanicThreshold` to zero percent will do. The comment is needed to clarify that the value of `0` will disable the panic threshold behaviour.

Comments such as these record hard won battles for understanding deep in the business logic. When you have the opportunity to write them, be sure to include enough hints that the next reader can follow your research. Links to issues, design documents, RFCs, or specifications that provide more background are always helpful.

TIP

Comments are associative

Comments on a method or function should describe the purpose of the function and potentially the arguments, the comment should be updated when the arguments change, or the purpiose of the function changes, in which case so will its name, both of which directly follow the comment.

Comments inside a function or method should be diretly followed by the line or block they are associated with, again, when the block changes, the comments should be reviewed.

2.1. Always document public symbols

Because godoc *is* the documentation for your package, you should always add a comment for every public symbol—variable, constant, function, and method—declared in your package.

Here are two rules from the Google Style guide:

- Any public function that is not both obvious *and* short must be commented.
- Any function in a library must be commented regardless of length or complexity.

GO
```
package ioutil

// ReadAll reads from r until an error or EOF and returns the data it read.
// A successful call returns err == nil, not err == EOF. Because ReadAll is
// defined to read from src until EOF, it does not treat an EOF from Read
// as an error to be reported.
func ReadAll(r io.Reader) ([]byte, error)
```

There is one exception to this rule; you don't need to document methods that implement an interface. Specifically don't do this:

GO
```
// Read implements the io.Reader interface
func (r *FileReader) Read(buf []byte) (int, error)
```

This comment says nothing. It doesn't tell you what the method does, in fact it's worse, it tells you to go look somewhere else for the documentation. In this situation I suggest removing the comment entirely (although some linters disagree).

Here is an example from the `io` package

GO

```go
// LimitReader returns a Reader that reads from r
// but stops with EOF after n bytes.
// The underlying implementation is a *LimitedReader.
func LimitReader(r Reader, n int64) Reader { return &LimitedReader{r, n} }

// A LimitedReader reads from R but limits the amount of
// data returned to just N bytes. Each call to Read
// updates N to reflect the new amount remaining.
// Read returns EOF when N <= 0 or when the underlying R returns EOF.
type LimitedReader struct {
    R Reader // underlying reader
    N int64  // max bytes remaining
}

func (l *LimitedReader) Read(p []byte) (n int, err error) {
    if l.N <= 0 {
        return 0, EOF
    }
    if int64(len(p)) > l.N {
        p = p[0:l.N]
    }
    n, err = l.R.Read(p)
    l.N -= int64(n)
    return
}
```

Note that the `LimitedReader` declaration is directly preceded by the function that uses it, and the declaration of `LimitedReader.Read` follows the declaration of `LimitedReader` itself. Even though `LimitedReader.Read` has no documentation itself, its clear from that it is an implementation of `io.Reader`.

TIP Before you write the function, write the comment describing the function. If you find it hard to write the comment, then it's a sign that the code you're about to write could be hard to understand.

2.2. Comments on variables and constants should describe their contents

I stated earlier that the name of a variable, or a constant, should describe its purpose. When you add a comment to a variable or constant, that comment should describe the variable's *contents*, not the variable's *purpose*.

GO

```go
const RandomNumber = 6 // determined from roll of an unbiased die
```

In this example the comment describes *why* `RandomNumber` is assigned the value six, and where the six was derived from. The comment *does not* describe where `RandomNumber` will be used. This is deliberate, `RandomNumber` may be used many times by any package that references it. It is not possible to keep a record of all those uses at the site that `RandomNumber` is declared. Instead the name of the constant should be a guide the appropriate use for potential users.

Here are some more examples:

```go
package http

const (
    StatusContinue           = 100 // RFC 7231, 6.2.1
    StatusSwitchingProtocols = 101 // RFC 7231, 6.2.2
    StatusProcessing         = 102 // RFC 2518, 10.1
    StatusOK                 = 200 // RFC 7231, 6.3.1
)
```

In general use the untyped constant `100` is just the number one hundred. *In the context of HTTP* the number `100` is known as `StatusContinue`, as defined in RFC 7231, section 6.2.1. The comment included with that declaration helps the reader understand *why* `100` has special significance as a HTTP response code.

For variables *without* an initial value, the comment should describe who is responsible for initialising this variable.

```go
// sizeCalculationDisabled indicates whether it is safe
// to calculate Types' widths and alignments. See dowidth.
var sizeCalculationDisabled bool
```

This example comes deep from the bowels of the Go compiler. Here, the comment lets the reader know that the `dowidth` function is responsible for maintaining the state of `sizeCalculationDisabled`.

The fact that this advice runs contrary to previous advice that comments should not describe who uses them is a hint that `dowidth` and `sizeCalculationDisabled` are intimately entwined. The comments presence suggests a possible design weakness.

TIP

Hiding in plain sight

This is a tip from Kate Gregory. [5] Sometimes you'll find a better name for a variable hiding in a comment.

```go
// registry of SQL drivers
var registry = make(map[string]*sql.Driver)
```

The comment was added by the author because `registry` doesn't explain enough about its purpose—it's a registry, but a registry of what?

By renaming the variable to `sqlDrivers` its now clear that the purpose of this variable is to hold SQL drivers.

```go
var sqlDrivers = make(map[string]*sql.Driver)
```

Now the comment is redundant and can be removed.

This advice also applies to comments *within* a function.

```go
func Sum(v []int) int {
    // total of numbers in v
    i := 0
    for _, n := range v {
        i += n
    }
    return i
}
```

```go
func Sum(v []int) int {
    total := 0
    for _, n := range v {
        i += n
    }
    return total
}
```

2.3. Comments on functions and methods should describe their purpose

The comment on a function signature should describe what the function intends to do, not how it does it. If the name of the function is all the description it needs — even better. Similarly they should describe the inputs and outputs of a function, not be overly perscriptive of how those should be used. Rather than describe the type of the return value, the function's comment should describe the value's meaning.

The description should be sufficient to write a unit test for the documented behaviour.

TIP

Be on the lookout for conjoining words like *or*, they are smell that a function may do more than one thing, violating the single responsibility principle. The comment should explain *what* the thing does, not *how* it does it.

2.4. Don’t comment bad code, rewrite it

> *Don’t comment bad code, rewrite it.*
>
> — *Brian Kernighan*

Comments highlighting the grossness of a particular piece of code are not sufficient. If you encounter one of these comments, you should raise an issue as a reminder to refactor it later. It is okay to live with technical debt, as long as the amount of debt is known.

The tradition in the standard library is to annotate a `TODO` style comment with the username of the person who noticed it.

```go
// TODO(dfc) this is O(N^2), find a faster way to do this.
```

The username is not a promise that that person has comitted to fixing the issue, but they may be the best person to ask when the time comes to address it. Other project annotate todos with a date and or an issue number, which is a benficial tradition.

2.5. Rather than commenting a block of code, refactor it

> *Good code is its own best documentation. As you’re about to add a comment, ask yourself, 'How can I improve the code so that this comment isn’t needed?' Improve the code and then document it to make it even clearer.*
>
> — *Steve McConnell*

Functions should do one thing only. If you find yourself commenting a piece of code because it is unrelated to the rest of the function, consider extracting it into a function of its own.

In addition to being easier to comprehend, smaller functions are easier to test in isolation. Once you’ve isolated the orthogonal code into its own function, its name may be all the documentation required.

3. Style

This section deals with matters of style.

NOTE

Function or method?

For brevity when I say function I mean function or method. When I mean method, speficically, I’ll say method.

3.1. Minimize use of vertical whitespace

`gofmt` solved so many unproductive style wars, intentation, alignment, and so on, are a thing of the past, but vertical whitespace is still an open question.

This quote from the Google C++ style guide (https://google.github.io/styleguide/cppguide.html#Vertical_Whitespace) is most apt:

This is more a principle than a rule: don't use blank lines when you don't have to. In particular, don't put more than one or two blank lines between functions, resist starting functions with a blank line, don't end functions with a blank line, and be sparing with your use of blank lines. A blank line within a block of code serves like a paragraph break in prose: visually separating two thoughts.

The basic principle is: The more code that fits on one screen, the easier it is to follow and understand the control flow of the program. Use whitespace purposefully to provide separation in that flow.

Some rules of thumb to help when blank lines may be useful:

- Blank lines at the beginning or end of a function do not help readability.
- Blank lines inside a chain of if-else blocks may well help readability.
- A blank line before a comment line usually helps readability — the introduction of a new comment suggests the start of a new thought, and the blank line makes it clear that the comment goes with the following thing instead of the preceding.

Just as you begin each paragraph—each new thought—with a line break, do so with a new set of statements in a function. This allows you to understand each part of a function, each set of statements. Ideally each function only has one set of statements, so no padding is necessary.

3.2. Prefer shorter functions

The maximum length of a function is inversely proportional to the complexity and indentation level of that function.

— Linux Kernel style guide[linux]

Each function should be written in terms of a single level of abstraction. Ideally a function should do one, and only one, thing.

Naive programmers think that design means "don't make functions or classes too long". However, the real problem is writing code that mixes unrelated ideas.

— Justin Meiners[meiners2019]

This should place an upper limit on the length of a function which is beneificial because, besides longer functions being harder to read, longer functions are more likely to mix more than one idea. The required disentanglement must then be performed by the reader.

3.3. Return early rather than nesting deeply

Every time you indent you add another precondition to the programmers stack consuming one of the 7 ±2 slots in their short term memory.

Go does not use exceptions for control flow thus there is no requirement to deeply indent your code to provide a top level structure for `try` and `catch` blocks. Rather than the successful path nesting deeper and deeper to the right, Go code is written in a style where the success path continues down the screen as the function progresses. Mat Ryer calls this practice 'line of sight' coding. [6]

This is achieved by using *guard clauses*; conditional blocks which assert preconditions upon entering a function. Here is an example from the `bytes` package,

```go
func (b *Buffer) UnreadRune() error {
    if b.lastRead <= opInvalid {
        return errors.New("bytes.Buffer: UnreadRune: previous operation was not a successful ReadRune")
    }
    if b.off >= int(b.lastRead) {
        b.off -= int(b.lastRead)
    }
    b.lastRead = opInvalid
    return nil
}
```

Upon entering `UnreadRune` the state of `b.lastRead` is checked and if the previous operation was not `ReadRune` an error is returned immediately. From there the rest of the function proceeds with the assertion that `b.lastRead` is greater that `opInvalid`.

Compare this to the same function written without a guard clause,

```go
func (b *Buffer) UnreadRune() error {
    if b.lastRead > opInvalid {
        if b.off >= int(b.lastRead) {
            b.off -= int(b.lastRead)
        }
        b.lastRead = opInvalid
        return nil // success
    }
    return errors.New("bytes.Buffer: UnreadRune: previous operation was not a successful ReadRune")
}
```

The body of the successful case, the most commonly executed, is nested inside the first `if` condition and the successful exit condition, `return nil`, has to be discovered by careful matching of *closing* braces. The final line of the function now returns an error, and the reader must trace the execution of the function back to the matching *opening* brace to know when control will reach this point.

This is more error prone for the reader, and the maintenance programmer, hence why Go prefers to use guard clauses and returning early on errors.

3.4. Make the zero value useful

> *Make the zero value useful.*
>
> — *Rob Pike*
> *Go Proverb*

Every variable declaration, assuming no explicit initaliser is provided, will be automatically intialised to a value that matches the contents of zero'd memory. This is the value's *zero value*. The type of the value determines it's zero value; for numeric types it is zero, for string types it is `""`, for pointer types, `nil`, the same for slices, maps, and channels.

This property of always setting a value to a known default is important for safety and correctness of your program and can make your Go programs simpler and more compact. This is what Go programmers talk about when they say "give your structs a useful zero value".

Consider the `sync.Mutex` type. `sync.Mutex` contains two unexported integer fields, representing the variable's internal state. Thanks to the zero value those fields will be set to will be set to `0` whenever a `sync.Mutex` is declared. `sync.Mutex` has been deliberately written to take advantage of this property, making the type usable without explicit initialisation.

GO

```
type MyInt struct {
    mu  sync.Mutex
    val int
}

func main() {
    var i MyInt

    // i.mu is usable without explicit initialisation.
    i.mu.Lock()
    i.val++
    i.mu.Unlock()
}
```

Another example of a type with a useful zero value is `bytes.Buffer` You can decare a `bytes.Buffer` and start writing to it without explicit initialisation.

GO

```
func main() {
    var b bytes.Buffer
    b.WriteString("Hello, world!\n")
    io.Copy(os.Stdout, &b)
}
```

A useful property of slices is their zero value is `nil`. This makes sense if we look at the runtime's (pseudo) definition of a slice header.

GO

```
type slice struct {
        array *[...]T // pointer to the underlying array
        len   int
        cap   int
}
```

The zero value of this struct would imply `len` and `cap` have the value `0`, and `array`, the pointer to memory holding the contents of the slice's backing array, would be `nil`. This means unless you need to specify a size you don't need to explicitly `make` a slice, you can just declare it.

GO

```
func main() {
    // s := make([]string, 0)
    // s := []string{}
    var s []string

    s = append(s, "Hello")
    s = append(s, "world")
    fmt.Println(strings.Join(s, " "))
}
```

NOTE

`var s []string` is similar to the two commented lines above it, but not identical. It is possible to detect the difference between a slice value that is `nil` and a slice value that has zero length.

GO
```
func main() {
    var s1 = []string{}
    var s2 []string
    fmt.Println(reflect.DeepEqual(s1, s2)) // false
}
```

A useful, albeit surprising, property of uninitialised pointer variables—nil pointers—is you can call methods on types that have a nil value. This can be used to provide default values simply.

GO
```
type Config struct {
    path string
}

func (c *Config) Path() string {
    if c == nil {
        return "/usr/home"
    }
    return c.path
}

func main() {
    var c1 *Config
    var c2 = &Config{
        path: "/export",
    }
    fmt.Println(c1.Path(), c2.Path())
}
```

I despise configuration structs, a type who’s only purpose is to provide facts—and they must be facts—not variables, to another type. Instead, figure out how to make the original type configurable. This often means making its zero value usable.

TIP

Make the zero value for your types usable, or prohibit its construction.
Avoid constructors.

Hard to prevent a go value being created so work to make the zero value safe.

If your type has no safe zero value, ensure that nobody else can construct it unsafely.

If you have a public type with fields that cannot be zeroed, have no valid zero value, then that type should in fact be private.

3.5. Methods on a T vs methods on a *T

Without exception, everything in Go is a copy. Fundamental to the understanding of Go are the three following axioms of Go values;

1. Every variable in Go is a value.

2. Every assignment is a copy.
3. Every formal parameter and return value is a copy.

We also know that method calls are just syntactic sugar for calling a function an passing the receiver as the first parameter. So, what are the rules for how the receiver should be passed, as a value or a pointer to that value?

- Use a pointer receiver when the caller will change the receiver. This could be also written as use a pointer receiver when the method is stateful.
- The inverse is also true, use a value receiver when the receiver is immutable. One of the few std lib examples of this is the `time.Time` type.

K&D[gopl] points out that if some of your methods have pointer receivers, all your methods should have pointer receivers. The same logic applies in reverse if you really want one method to have a value receiver; all the others must follow suite.

One argument is to always declare all methods on `*T` as it avoids copying and is thus faster. However Go developers are acutely attuned to this sort of absolutist thinking and tend to reject it without further proof. After all, we pass around slice and string values, both of which are several words in length without a care, so a blanket rule that every method must be declared on a pointer receiver for performance reads like dogma.

In the end I'm left with the unhappy compromise of;

1. In general, use pointer receivers.
2. If you use a pointer receiver on one method, the rest should follow suit.

In practical terms, pointer receiver should be your go to unless you are working on a specific type that you want to exploit the properties of the copying behaviour of value receivers. Methods on values should be used sparingly, and with great consideration.

TIP *Return types with methods by reference*
Return values with methods by reference, and those without by value.

3.5.1. Avoid naming your method's receiver this, or self

Not many people know this, but method notation, i.e. `v.Method()` is actually syntactic sugar and Go also understands the de-sugared version of it: `(T).Method(v)`. Naming the receiver like any other parameter reflects that it is, in fact, just another parameter quite well.

NOTE `this` comes from smalltalk's keyword `thisContext` (which they dedicated 1/6th of the keyword space to), which begat Java's `this`. The receiver is not special in Go, it doesn't deserve special treatment

This also implies that the receiver-argument inside a method may be `nil`. This is not the case with receivers in other languages

Convention dictates that the receiver of a method be named as it were an argument, using `this` or `self` is not considered idiomatic.

3.6. Function vs methods

When should something be a method on a type vs a free function? I recommend using methods where state is retained, functions where it is not.

- If the state retained is related to this first argument, the method is placed on that type.
- If the first arg is a concrete value, and this is a public function, considering making it a method on the first value, especially if there is only one other parameter.
- If the function is private, avoid making it a method.

Public functions are the way to communicate across packages, and interfaces are the mechanism to define behaviour across packages.

Pure functions are easier to test than methods because methods live on types and are inherently impure — they contain state via the receiver. The opposite is also true, if a method never mutates its receiver, should it be a method?

TIP If you take a value who has a method you call, and that type only has one method, should it just be a function?

If you prefer a function to a method, which lets you add methods to interfaces, continue to place the receiver in the first formal parameter. As you wrap and abstract, parameters should move to the left, from the format parameters, to the receiver, to a type embedded in the receiver

Here is a rule of thumb that may guide you in deciding to use a method or a function. Methods for what they do, functions for what they return.

3.7. Avoid named return values

Named return values permit the function's author to;

- Increase separation between declaration and use. Which runs contrary to the previous suggestion, and decreases readability, especially when the function or method is long.
- Increase the risk of shadowing.
- Enable the use of naked returns.

Each of which are a net negative on the readability of the function.

- Named returned arguments introduce a discontinuity in the declaration of variables.
- Named returns move the declaration to an unexpected location.
- Named returns force you to declare all return parameters, or worse declare them `_`.

In short, named return values are a symptom of a clever piece of code which should be reviewed with suspicion. If the method is infact simple, then named returns values are playing the short game of brevity over readabilty.

Its's my opinion that names return arguments should not be used unless required to provide something that could not reasonably be done another way. For example, to modify the return arguments in a `defer` block, where it is required to name return arguments to capture them.

GO
```go
func ReadFile(name string) (output string, err error) {
    defer func() {
        if err != nil {
            err = fmt.Errorf("could not read %q: %v", name, err)
        }
    }()

    f, err := os.Open(name)
    if err != nil {
        return "", err
    }

    // ...
}
```

What is clear is that this function is complex, and named return values are part of that complexity.

All things being equal, you should aim to write simple code, not clever code. And so should avoid designs that require named return values.

> There is nothing you can do with named return values that you cannot do with a few more lines of code. Avoid them if possible.

3.8. Avoid naked returns

Naked returns combine the declaration of a return value in the function declaration with an unspecified assignment somewhere in the body of the function. Everything about the use of naked returns admits a set of actions that hides bugs, in even small functions.

Naked returns are inconsistent; they make it look like the function or method returns no values, when infact it does, as they were declared in the function signature.

Naked returns are often used inconsistently, especially in an error path where nil is returned explicitly, or the zero value of a named return value is used. Combined with early returns [link] this results in multiple, sometimes conflicting, return stamements

Use naked return consistently or not at all.

GO
```go
func (f *Filter) Open(name string) (file File, err error) {
    for _, c := range f.chain {
        file, err = c.Open(name)
        if err != nil {
            return
        }
    }
    return f.source.Open(name)
}
```

TIP If you *must* use naked returns; use only naked returns in a function — don’t mix and match.

3.9. Avoid incomplete initalisation

“Use struct literal initialization to avoid invalid intermediate state. Inline struct declarations where possible.” — Peter Bourgon[bourgon2016]

Where possible values should be completely intitalised by construction, rather than by convention. We see examples of this failure at the most basic levels, for instance when declaring a value then overriding the default zero initalisation.

```go
var userCount int // initally zero
userCount = countUsers()
```

One the first line, `userCount` is in scope, but misleadingly holds the value `0`. Only after `countUsers` has been called is `userCount` valid.

The incomplete initialsation pattern tends to show up in more complicated declarations

```go
type Virtualhost struct {
        hostname string
        routes []Route
}

vhost := &VirtualHost{
        hostname: "www.example.com",
}

for _, r := range findRoutes(vhost.hostname) {
        vhost.routes = append(vhost.routes, &Route{ ... })
}

return vhost
```

In this example `vhost` is incomplete, it has not yet had a set of routes appended too it. Compare this with

```go
type Virtualhost struct {
        hostname string
        routes []Route
}

hostname := "www.example.com"
var routes []*Route

for _, r := range findRoutes(hostname) {
        routes = append(routes, &Route{ ... })
}

return &VirtualHost{
        hostname: hostname,
        routes: routes,
}
```

In the revised version, the `routes` slices is populated fully, then assigned to the `&Virtualhost` literal, noting that this literal is never given a name so cannot appear partially initialised.

Specifically avoid public `Init` functions.

- How do you know if they've been called already?
- What happens if they are called twice? Someone might try to use them to *clean* and object from sync.Pool or otherwise recycle it.

TIP Never allow an uninitialised value to escape from your api.

3.10. Avoid finalisation

“*Go does not guarantee that a finalizer will be run in some bounded amount of time. In fact Go does not even guarantee that a finalizer will be run before the program terminates.*

— *Richard L. Hudson*

Go contains a finalisation facility that lets the programmer register a function to be run when no live references to the object remain. Finalisation's siren song of garbage collection for non memory resources can be beguiling. Inherently the idea is sound; in some programs it can be difficult to identify the owner of a resource like a file, a lock, or a socket. **Do not use it**.

At one point in the Go runtime's development there were serious discussions about making finalisation a noop; functions registered for finalisation would simply be ignored. Fortuntately cooler heads prevailed, but had they not this would *not* have been a violation of the specification or the Go 1 guarentee. Runtime finalisation does not guarentee timely execution of finalisations; that can be delayed until after the program has exited, and still be compliant.

There is only one place in the entire standard library where finalisation is used; for variables of type `*os.File`. This use is at best a historical artifact. Given the serious problems with finalisation, and near prohibition in the standard library, do not design your software to rely on timely finalisation.

Do not write programs who's correctness depends on finalisation, instead associate the lifetime of a resource to the lifetime of a goroutine.

4. Understanding `nil`

`nil` is a curious beast. There is no `nil` type, nor can you alias another type to be `nil`, it is a reserved word. `nil` can be assigned to a value, and values can be compared to `nil`, but `nil` cannot be compared to itself.

GO

```go
package main

import "fmt"

const T = nil != nil // (1)

func main() {
    fmt.Println(nil == nil) // (2)

    if nil == new(int) {
        fmt.Println("hmm") // (3)
    }
}
```

1. `nil` cannot be compared with itself for inequality
2. nor with itself for equality
3. however `nil` may appear on either side of a binary operation

Given all these restrictions, `nil` sounds out of place in the orthogonal Go world. Why would such a concept exist? The answer is, while `nil` may appear inconsistent, it makes a lot of other interactions in Go simpler.

- If you assign `nil` to a pointer the pointer will not point to anything.
- If you assign `nil` to an interface, the interface will not contain anything.
- If you assign `nil` to a slice and the slice will have zero len and zero cap and no longer point an underlying array.

`nil` 's meaning, or it's type, is fully determined by the static type of the variable it's assigned to. When you write a statement like

GO

```go
var f *os.File
if f == nil {
    // ...
}
```

The rule of expressions dictates that all the variables in the expression must have the same type. We know the type of `f`, it is a `*os.File`, therefore we know that `nil` has been coerced from an ideal constant to an expression which evaluates to the value also of type `*os.File`.

Here is a more complicated example

GO

```go
var s []string
if s == nil {
    // ...
}
```

Again, the type of `s` is known, and as there are no conversions in the expression, the type of `nil` on the other side of the comparison must be the same, `[]string`.

4.1. Be wary of `nil` and interfaces

As we saw above `nil` can be simple, or complicated, depending on how you reason about it. One area where `nil` is complicated, until you memorise the rule is the dreaded *typed nil*.

```go
func Open(path string) io.Writer {
    var f *os.File
    f, _ = os.Open(path)
    return f
}

func main() {
    f := Open("/missing")
    fmt.Println(f == nil) // (1)
}
```

1. prints false because the returned value has a *type* of `os.Writer` not `nil`.

If your method returns an *interface type*, be sure to always `return nil` explicitly. Assigning nil to a value of a concrete type and returning that will convert it to a typed `nil`

> Always return an explicit `nil`, rather than a typed value containing `nil`.

4.2. Never use nil to indicate failure

Go's inclusion of `nil` tends to upset people who come from a Java, C#, or C++ background because they are traumatised by how `nil` operated in those languages.

In other languages, especially those that don't support multiple return values, it is extremely common to return a `nil` like value a failure happens inside the method. On one hand this is eminently sensible, exceptions are overused, and for most failures they are hardly unexpected, so some mechanism of representing a failure that doesn't warrant the four alarm fire of an exception is called for. Obviously this has some major downsides. As the flow of execution is not redirected a catch block this `nil` (or `null`, not naming any names) sentinel value now represent a silent failure condition.

Fortuntaly Go does not suffer from these drawbacks. This is for two reasons main reasons:

1. Multiple return values don't require the author of a function to overload the single return value with an error state.
2. A general prohibition about passing `nil` into and out of functions.

> Never use nil to indicate a failure, only to indicate the absence of an error.

4.3. A nil receiver is a programming error

When Go programmers discover that you can call a method on `nil` receiver it generally blows their mind.

```go
type Bar struct{}

func (b *Bar) Whoa() {
    fmt.Println("whaaaaaat?")
}

func main() {
    var b *Bar // (1)
    b.Whoa()   // (2)
}
```

1. `b` is of type `*Bar` but is `nil`.
2. Prints "whaaaaaat".

This is because a method in Go is just syntactic sugar for a function who's first parameter is the receiver.

```go
func Whoa(b *Bar) {
    fmt.Println("whaaaaaat?")
}
```

Restated like this it is clear to see why passing a `nil` value for `b` is uneventful. However, a problem arises when the code attempts to access `b` or one of it's fields.

```go
type Bar struct {
    message string
}

func (b *Bar) Whoops() {
    fmt.Println(b.message) // (1)
}

func main() {
    var b *Bar
    b.Whoops() // (2)
}
```

1. panics occurs here
2. not here

Faced with this realisation Go programmers are gripped with fear that someone could call their code's methods on an accidental `nil` method. Their usual reaction is a creeping panic that they will have to pepper their code with nil checks like this protect against this scenario.

```go
func (b *Bar) Whoops() {
    if b != nil { // (1)
        fmt.Println(b.message)
    }
}
```

1. only execute the *body* of the method if `b` is not nil

The solution is to realise that the check for a `nil` receiver *before* attempting the call is in the right place.

```go
func main() {
    var b *Bar
    if b != nil {
        b.Whoops() // (1)
    }
}
```

1. only call the method if `b` is not nil

Rather than checking inside the method when it is too late, the check should be executed by the caller. But this is seen as unsatisfactory because it force the check to *every* call site, rather than in one place, the receiver.

For arguments sake let's explore the options to effectively handle a nil receiver *inside the method*. What are the options for an author to handle this situation?

Panic

Given that calling a method on a `nil` receiver, expect where the method was written to explicitly handle this behaviour (there are types in the stdlib that do this, but not many), is an unrecoverable programming error, a reasonable response would be to panic.

```go
func (b *Bar) Whoops() {
    if b == nil {
        panic("b is nil") // (1)
    }
    fmt.Println(b.message)
}
```

But given that dereferencing `b` to access the `message` field is going to panic anyway, apart from having control over the panic message, this seems to add little other than boilerplate.

Return an error to the caller

The next option to the reporting problem may be treat this coding error like any other non fatal error and return an `error` value.

```go
func (b *Bar) GetMessage() (string, error) {
    if b == nil {
        return "", errors.New("b is nil") // (1)
    }
    return message, nil
}
```

1. return a descriptive error to the caller.

This has serious implications for the caller of *any* method.

1. Every method will have to return an error. Every. Method.
2. Every caller will have to check the error *after* a call to *any* method.
3. Every interface you define will have to include an error parameter so that an implementation can report it was called on a nil receiver.
4. Every interface you implement will have to proved you with an error return parameter.

Elide execution

We saw that option above, `Whoops` would print nothing if it was called on a `nil` receiver. This is perhaps the worst choice as now the operation will silently do nothing. Imagine trying to debug a complex failure in your application because some logic did not fire because it was passed a `nil` receiver?

Given there is no reasonable way for the method executed on a `nil` receiver to protected against this, the remaining option is to simply not worry about it. After all a `nil` receiver is a symptom of a bug that happened elsewhere in your code. The most likely cause was a failure to check the error from a previous call. That is the place where you should spend your efforts, not defensively trying to code around a failure to follow proper error handling.

> Don't check for a `nil` receiver. Employ high test coverage and vet/lint tools to spot unhandled error conditions resulting in `nil` receivers.

5. Interfaces

> *People have it backwards: #golang interfaces exist for the functions that use them,* ***not*** *to describe the types that implement them.*
>
> *— https://twitter.com/jmoiron/status/532314843689132032[Jason Moiron]*

Interfaces describe behaviour, types describe data. Interfaces are the key strategy for polymorphism and information hiding in Go.

But wait, aren't interfaces types? Technically yes, but for the purpose of brevity, even though an interface is declared with the `type` keyword we'll say that *interface types* are disjoint from the set of other types.

5.1. T vs *T for interfaces

The method sets of a value of type `T` and type `*T` are disjoint.

You may convert a value of `T` to `*T` with the address of operator, `&T`. Conversely values of `*T` can be converted to values of `T` with the dereference operator.

Sadly Go retains the syntactic difficulty between the `*T` declaration and the `*T` dereference, although the former is a type, and the latter is a value of the result of the expression `*T`.

Although the method sets of T and *T are different, the compiler will help you by automatically inserting the relevant conversion operation, assuming the value is addressable.

This allows the caller to operate *as if* the methods available on T and *T are a union, almost all of the time (save where addressability is not present, see maps)

However, when a value is assigned to an interface type this illusion is broken.

```go
type Counter interface {
    Inc() int
}

type T struct {
    count int
}

func (t *T) Inc() int {
    t.count++
    return t.count

}

func main() {
    var counter Counter = T{count: 0}
    fmt.Println(counter.Inc())
}
```

Why is this so?

Deference is easy, however address of, converting a `T` into a `*T` would take the address of the *copy* of the value stored in the interfaces' value slot, not the original value before assignment.

Said another way, because everything in Go is a copy, there is no way to "wrap" an interface around an existing value, a copy must be taken, if the value is a not pointer type, then the copy of the original value placed inside an interface

If your type implements an interface and has methods on its pointer (which almost all types do), then you should always use a pointer value when assigning to the interface.

TODO:unfinshed

6. API Design

There's a great line in Strunk & White about [where] you'll find places where the rules are broken, but you'll find some compensating merit to make up for it.

— *Brian Kernighan*

All of the suggestions I've made so far are just that, suggestions. These are the way I try to write Go, but I'm not going to push them too hard in code review.

However when it comes to reviewing APIs during code review, I am less forgiving. This is because everything I've talked about so far can be fixed *without* breaking backward compatibility. They are, for the most part, implementation details.

When it comes to the public API of a package, it pays to put considerable thought into the initial design, because changing that design later is going to be disruptive for people who are already using your API. Changing your public API forces the existing user base to have to dedicate engineering resources to upgrading across your API break. The larger the break, the more likely this task will be considered *low impact*, but *high risk*, and likely to be pushed off in light of other business priorities.

Design means saying No

— Author

Go is a language designed for collaboration and composition. There is a strong delineation between what happens inside a package, privately, and what is exposed to callers of the package, publicly. What is the api of your package ? The functions, and the methods obviously, but also:

- The constants
- The symbols
- The formal params
- The returns values
- The methods on your interfaces
- The fields of your structure, including their order
- The errors, their contents and their types

If it's exported, its part of your public API.

6.1. Design APIs that are hard to misuse.

APIs should be easy to use and hard to misuse.

— Josh Bloch [2]

If you take away one thing from this section, it should be this advice from Josh Bloch. If an API is hard to use for simple things, then every invocation will look complicated. When the actual invocation of the API *is* complicated it will be less obvious and more likely to be overlooked.

Strive to make your APIs difficult to misuse *by design*.

6.2. Design APIs for their default use case.

TODO: unfinished

Take away, libraries define concrete types, helpers, free function, clients define the behaviour they want with their own interfaces.

Good api design should make the default behaviour trivial to implement and none trivial behaviour possible to implement. Each public function that takes an argument must have a obvious and defensible default behaviour.

A few years ago I gave a talk [7] about using functional options [8] to make APIs easier to use for their default case.

The gist of this talk was you should design your APIs for the common, or default, use case. Said another way, your API *should not* require the caller to provide parameters which they don't care about. More than being hard to use, you place the user in the position of *guessing* reasonable values. If they are lucky, the values they YOLO'd have no impact. That's if they are lucky.

Avoid unused parameter in default case

- If there is one unusual case, add a second constructor.
- If there are more than one, consider a functional option set.

6.2.1. Use functional options to configure complex types

TODO: unused

Unused local variables can be a source of errors and reduce readability. Unused imports slow compilation and linking, even if that unused code is removed in the final binary the cost is still paid on every build.

Language design is a trade off, as Pike and others have described Go as "a language for programming in the large". Go achieves its goals of improved readability and efficient compilation. Making unused globals an error would push more burden onto programmers, which is itself in conflict with the goal of reducing compiler bureaucracy.

Functional options let you write APIs that can grow over time. They enable the default use case to be the simplest. They provide meaningful configuration parameters. They give you access to the entire power of the language to initialize complex values.

Make types configurable during construction. If you cannot, create a new function and use function options.

6.2.2. Discourage the use of `nil` as a parameter

I opened this chapter with the suggestion that you shouldn't force the caller of your API into providing you parameters when they don't really care what those parameters mean. This is what I mean when I say *design APIs for their default use case*.

Here's an example from the `net/http` package.

NOTE I pick on the `net/http` package a lot. I don't mean to imply it, or the engineers who contributed to it, are bad. On the contrary, `net/http` has been tremendously successful and with that success has come a process of extension via accretion which makes it a great candidate for case studies.

GO
```go
package http

// ListenAndServe listens on the TCP network address addr and then calls
// Serve with handler to handle requests on incoming connections.
// Accepted connections are configured to enable TCP keep-alives.
//
// The handler is typically nil, in which case the DefaultServeMux is used.
//
// ListenAndServe always returns a non-nil error.
func ListenAndServe(addr string, handler Handler) error {
```

`ListenAndServe` takes two parameters, a TCP address to listen for incoming connections, and `http.Handler` to handle the incoming HTTP request. `Serve` allows the second parameter to be `nil`, and notes that usually the caller *will* pass `nil` indicating that they want to use `http.DefaultServeMux` as the implicit parameter.

Now the caller of `Serve` has two ways to do the same thing.

GO
```go
http.ListenAndServe("0.0.0.0:8080", nil)
http.ListenAndServe("0.0.0.0:8080", http.DefaultServeMux)
```

Both do exactly the same thing.

This `nil` behaviour is viral. The `http` package also has a `http.Serve` helper, which you can reasonably imagine that `ListenAndServe` builds upon like this

GO
```go
func ListenAndServe(addr string, handler Handler) error {
    l, err := net.Listen("tcp", addr)
    if err != nil {
        return err
    }
    defer l.Close()
    return Serve(l, handler)
}
```

Because `ListenAndServe` permits the caller to pass `nil` for the second parameter, `http.Serve` also supports this behaviour. In fact, `http.Serve` is the one that implements the "if `handler` is `nil`, use `DefaultServeMux`" logic. Accepting `nil` for one parameter may lead the caller into thinking they can pass `nil` for both parameters. However calling `Serve` like this,

GO
```go
http.Serve(nil, nil)
```

results in an ugly panic.

TIP Don't mix `nil` and non `nil`-able parameters in the same function signature.

The author of `http.ListenAndServe` was trying to make the API user's life easier in the common case, but possibly made the package harder to use safely.

There is no difference in line count between using `DefaultServeMux` explicitly, or implicitly via `nil`.

GO
```go
    const root = http.Dir("/htdocs")
    http.Handle("/", http.FileServer(root))
    http.ListenAndServe("0.0.0.0:8080", nil)
```

verses

GO
```go
    const root = http.Dir("/htdocs")
    http.Handle("/", http.FileServer(root))
    http.ListenAndServe("0.0.0.0:8080", http.DefaultServeMux)
```

and a was this confusion really worth saving one line?

GO
```go
    const root = http.Dir("/htdocs")
    mux := http.NewServeMux()
    mux.Handle("/", http.FileServer(root))
    http.ListenAndServe("0.0.0.0:8080", mux)
```

TIP Give serious consideration to how much time helper functions will save the programmer. Clear is better than concise.

TIP

Avoid public APIs with test only parameters

Avoid exposing APIs with values which only differ in test scope. Instead, use Public wrappers to hide those parameters, use test scoped helpers to set the property in test scope.

6.3. Strive for a minimal API surface

If you have a function which takes five parameters, you probably missed some.

— *Alan Perlis*

In this talk, I have presented many of the existing configuration patterns, those considered idiomatic and commonly in use today, and at every stage asked questions like: - Can this be made simpler? - Is that parameter necessary? - Does the signature of this function make it easy for it to be used safely? - Does the API contain traps or confusing misdirection that will frustrate?

Declarations provide the groundwork for a straightforward design, but it is the active elements of a Go program; the functions, the methods and it's interfaces which bear the weight of the design of a Go program.

If a function is public and does not have anything to do with the package; uses none of the packages symbols * move it away * this might be a utility function

Some people think that programming should be an art, every construct should be perfectly expressed and abstracted and beautiful, and things like simplicity, portability, and performance are unimportant details. Other people write programs to consume input and produce output, for various values of input and output. The real value of development is the work the program does, not the work the programmer does.

— *brokedown*
reddit (https://www.reddit.com/r/golang/comments/30bndg/how_do_you_respond_to_these_antigolang_pieces/cpslzwh)

6.4. Be wary of functions which take several parameters of the same type

WARNING

If you have a method which takes two parameters of the same type, there's a 50% chance the caller will reverse the args.

A good example of a simple looking, but hard to use correctly, API is one which takes two or more parameters of the same type. Let's compare two function signatures:

```go
func Max(a, b int) int
func CopyFile(to, from string) error
```

What's the difference between these two functions? Obviously one returns the maximum of two numbers, the other copies a file, but that's not the important thing.

```go
Max(8, 10) // 10
Max(10, 8) // 10
```

Max is *commutative*; the order of its parameters does not matter. The maximum of eight and ten is ten regardless of if I compare eight and ten or ten and eight.

However, this property does not hold true for `CopyFile`.

```go
CopyFile("/tmp/backup", "presentation.md")
CopyFile("presentation.md", "/tmp/backup")
```

Which one of these statements made a backup of your presentation and which one overwrite your presentation with last week's version? You can't tell without consulting the documentation. A code reviewer cannot know if you've got the order correct without consulting the documentation.

The general advice is to try to avoid this situation. Just like long parameter lists, indistinct parameter lists are a design smell.

However, a possible solution to this class of problem is to introduce a helper type which will be responsible for calling `CopyFile` correctly.

```go
type Source string

func (src Source) CopyTo(dest string) error {
    return CopyFile(dest, string(src))
}

func main() {
    var from Source = "presentation.md"
    from.CopyTo("/tmp/backup")
}
```

In this way `CopyFile` is always called correctly and, given its poor API can possibly be made private, further reducing the likelihood of misuse.

TIP APIs with multiple parameters of the same type are hard to use correctly.

6.5. Prefer var args to []T parameters

It's very common to write a function or method that takes a slice of values.

```go
func ShutdownVMs(ids []string) error
```

This is just an example I made up, but its common to a lot of code I've worked on. The problem with signatures like these is they presume that they will be called with more than one entry. However, what I have found is many times these type of functions are called with only one argument, which has to be "boxed" inside a slice just to meet the requirements of the functions signature.

Additionally, because the `ids` parameter is a slice, you can pass an empty slice or `nil` to the function and the compiler will be happy. This adds extra testing load because you *should* cover these cases in your testing.

To give an example of this class of API, recently I was refactoring a piece of logic that required me to set some extra fields if at least one of a set of parameters was non zero. The logic looked like this:

```go
if svc.MaxConnections > 0 || svc.MaxPendingRequests > 0 || svc.MaxRequests > 0 || svc.MaxRetries > 0 {
    // apply the non zero parameters
}
```

As the `if` statement was getting very long I wanted to pull the logic of the check out into its own function. This is what I came up with:

```go
// anyPostive indicates if any value is greater than zero.
func anyPositive(values ...int) bool {
    for _, v := range values {
        if v > 0 {
            return true
        }
    }
    return false
}
```

This enabled me to make the condition where the inner block will be executed clear to the reader:

```go
if anyPositive(svc.MaxConnections, svc.MaxPendingRequests, svc.MaxRequests, svc.MaxRetries) {
        // apply the non zero parameters
}
```

However there is a problem with `anyPositive`, someone could accidentally invoke it like this

```go
if anyPositive() { ... }
```

In this case `anyPositive` would return `false` because it would execute zero iterations and immediately return `false`. This isn't the worst thing in the world — that would be if `anyPositive` returned `true` when passed no arguments.

Nevertheless it would be be better if we could change the signature of `anyPositive` to enforce that the caller should pass at least one argument. We can do that by combining normal and vararg parameters like this:

```go
// anyPostive indicates if any value is greater than zero.
func anyPositive(first int, rest ...int) bool {
    if first > 0 {
        return true
    }
    for _, v := range rest {
        if v > 0 {
            return true
        }
    }
    return false
}
```

Now `anyPositive` cannot be called with less than one argument.

Go's variable arguments syntax, the elipsis, is a very powerful tool in designing functions and methods that are easier to use. If you have a functions like

```go
func DeployOne(t *Thing)
func DeployMany(t []*Thing)
```

You can combine them into one function with

```go
func Deploy(t ...*Thing)
```

Which operates the same as `DeployMany` because the type of `t` inside Deploy is `[]*Thing`.

> TIP
>
> *Deploy at least one thing*
>
> Astute readers will have noticed that `Deploy` can now be invoked without any `Thing`s to deploy. This can be solved like this
>
> ```go
> func Deploy(first *Thing, rest ...*Thing)
> ```

6.6. Prefer single method interfaces

Well designed interfaces are more likely to be small interfaces; the prevailing idiom is that interfaces contain only a single method. It follows that small interfaces lead to simpler implementations, because it is hard to do otherwise. This leads to programs comprised of simple implementations connected by small interfaces.

The opposite is also true, the larger the interface, the less abstraction it provides.

Interfaces *must* represent a generalisation of the behaviour of a set of implementations. The larger the interface, the larger the visible state of the implementation.

TODO: unfinished

6.7. Prefer streaming interfaces

Where-ever possible avoid reading data into a `[]byte` and passing it around.

Depending on the request you may end up reading megabytes of data into memory. This places huge pressure on the GC, which will increase the average latency of your application. Instead use `io.Reader` and `io.Writer` to construct processing pipelines to cap the amount of memory in use per request. For efficiency, consider implementing `io.ReaderFrom` and `io.WriterTo` if you use a lot of `io.Copy`. These interface are more efficient and avoid copying memory into a temporary buffer.

Much programming involves the movement and manipulation of data. If you are lucky this data is already in memory, in convenient types like `float64` or a `struct`. If you are unlucky, which is more often the case, you will be dealing with data in the form of raw `[]byte` slices. This last statement is especially true for Go's raison d'etre, client/server programming.

`[]byte` is always as big as it says, it's a concrete value. `io.Reader` and `io.Writer` let you work at a higher level while working with a window of data.

Also solves ownership issues, if you have an `io.Reader` value, you have little control over the ownership of the underlying data, all you can ask is for some of that data to be read into a `[]byte` slice that you provide.

TIP — Use `io.Reader` , `io.Writer` for untyped sequences of bytes. Use channels for typed sequences of values.

6.8. Functions should be named for what they return, methods should be named after the action they perform

Q\. What is the difference between a function and a method?

A\. There is no difference between a function and a method; a method is simply syntactic sugar for the receiver as the implicit first parameter to a function.

So, if there is no difference between a function and a method then when should one be used in favour of the other.

A function implicitly has no state, unless it's accessing global state. A method therefore must be used when there is state local to its instance; and you could further draw the conclusion that a function that operates on global state is a singleton jnstange of a method on a singleton instance of an unnamed type

6.9. Let callers define the interface they require

Just as it is a mistake to ask for formal parameters which go unused, asking for an interface type with methods that go uncalled is a smell.

Let's say I've been given a task to write a method that persists a Document structure to disk.

```go
type Document struct {
        // mo' state
}

// Save writes the contents of the Document to the file f.
func (d *Document) Save(f *os.File) error
```

I could specify this method, `Save` , which takes an `*os.File` as the destination to write the `Document` . But this has a few problems.

The signature of `Save` precludes the option to write the data to a network location. Assuming that in the new world of lambda functions and microservices, network storage is likely to become requirement, the signature of this function would have to change, impacting all its callers.

`Save` is also unpleasant to test, because it operates directly with files on disk. To verify its operation the test would have to read the contents of the file after being written. You would also have to ensure that `f` was written to a temporary location and always removed afterwards.

Moreover `*os.File` defines a lot of methods which are not relevant to `Save` , like reading directories and checking to see if a path is a symlink. It would be useful if the signature of `Save` could describe only the parts of `*os.File` that were relevant.

```go
// Save writes the contents of d to the supplied ReadWriterCloser.
func (d *Document) Save(rwc io.ReadWriteCloser) error
```

Using `io.ReadWriteCloser` we can apply the interface segregation principle to redefine `Save` to take an interface that describes more general file shaped things. With this change, any type that implements the `io.ReadWriteCloser` interface can be substituted for the previous `*os.File`. This makes `Save` both broader in its application, and clarifies to the caller of `Save` which methods of the `*os.File` type are relevant to its operation. As the author of `Save` I no longer have the option to call those unrelated methods on `*os.File` as it is hidden behind the `io.ReadWriteCloser` interface. But we can take the interface segregation principle a bit further.

Firstly, it is unlikely that if `Save` follows the single responsibility principle, it will read the file it just wrote to verify its contents—that should be responsibility of another piece of code.

GO
```go
// Save writes the contents of d to the supplied WriteCloser.
func (d *Document) Save(wc io.WriteCloser) error
```

We can narrow the specification for the interface we pass to Save to just writing and closing.

Secondly, by providing `Save` with a mechanism to close its stream, which we inherited in this desire to make it still look like a file, this raises the question of under what circumstances will `wc` be closed. Possibly `Save` will call `Close` unconditionally, or perhaps `Close` will be called in the case of success. Neither of these is a good option. Unconditionally closing `wc` after the call to `Save` precludes the caller from writing additional data after the document is written. Conditionally closing the `WriteCloser` — it doesn't matter if its on success, or failure—means the caller must grow intricate knowledge of the operation of `Save`.

GO
```go
// Save writes the contents of d to the supplied Writer.
func (d *Document) Save(w io.Writer) error
```

A better solution would be to redefine `Save` to take only an `io.Writer`, stripping it completely of the responsibility to do anything but write data to a stream.

By applying the interface segregation principle to our `Save` function, the results has simultaneously been a function which is the most specific in terms of its requirements—it only needs a thing that is writable—and the most general in its function, we can now use `Save` to save our data to anything which implements `io.Writer`.

As a side effect it is clear that the name of the method is no longer accurate. A better name may be

GO
```go
func (d *Document) WriteTo(w io.Writer) error
```

> Interfaces declare the behaviour the caller requires not the behaviour the type will provide. Let callers define an interface that describes the behaviour they expect. The interface belongs to them, the consumer, not you.

6.10. Prefer types rather than names for interface methods

When declaring an interface the variable name is not needed — that's a property of the type that implements the interface. Thus

GO
```go
type Runnable interface {
        Run(context.Context, string, int)
}
```

Is valid, but not very useful descriptive.

However there is a clue for how we can make this interface declaration more expressive. Part of it is aleady there, can you see it. `Run` takes three parameters, a `context.Context`, a `string`, and an `int`. What does the `string` value denote, maybe its a name of the thing being run, maybe its a operation too run? What does the `int` valye denote, maybe its an identifier for this job, maybe its a trace ID, maybe its a timeout?

But the first parameter, `context.Context` is unequivocal. It its a context, more importnatly it is *the* context, not just a generic context like thing.

Perhaps the take away is for interface declarations, the avoid primative types (those declared in the universe block).

TIP

The types of a parameter, not its name should describe what a function does

> *"If users must read the code of a method in order to use it, there is no abstraction: all of the complexity of the method is exposed."*
>
> — *John Ousterhout*
> *A Philosophy of software design*

Remember, methods exist to implement interfaces.

7. Package Design

> *Before software can be reusable it first has to be usable.*
>
> — *Ralph Johnson (https://twitter.com/codewisdom/status/1118493566479810571?s=12)*

I want to open this discussion with an observation based on Johnson's. I don't want to take a position go code reuse, but I want to use the notion of *usable* for my central thesis that *each package should provide one thing*.

Packages are foundation of every Go program. This is a profoundly different approach to modular software design to many contemporary languages; some of which share the ideas of modules, but these are mixed up with notions of private, protected, friend, access rules. You can declare methods, only on types you declare in your package, not ones you've imported elsewhere even though those types are by necessity public.

We see this in the compilation model. In C you ask questions like, if I change this header file, which source files include it and need rebuilding? In Go the unit of compilation is the package, so we ask, which packages does this package depend on?

The notion of Go's packages are one of it's strongest examples of orthogonality.

The rule of thumb I follow is not, "what types should I put in this package", instead I turn it around and ask "what does this package provide?" Normally the answer to that question will not be "this package provides the X type", but "this package let's you speak http".

> *Write shy code - modules that don't reveal anything unnecessary to other modules and that don't rely on other modules' implementations.*
>
> — *Dave Thomas (https://twitter.com/codewisdom/status/1045305561317888000?s=12)*

In his book, *Test Driven Design and Development*, Kent Beck describes the idea of an indivisible, atomic, unit of software.

In the physical world atoms are composed of quarks, mesons, bosons, and gluons. We cannot observe them directly, only infer them from their *behaviour*--mass, charge, gravitational attraction. In the software world, if a unit is composed of smaller subatomic particles, as a user—a caller of that software—we are unable to directly observe the imlementation details of the unit. Instead we rely on the *behaviour* of a unit.

The size of a unit of software differs by language.

- In C the unit is a function, as C offers little else.
- In Java, the unit of software is commonly incorrectly believed to be the class.
- In Go, the unit of software is not the function, or the type, or the method, but instead the *package*.

Just as the implementation of a function or method is unimportant to the caller, the implementation of the functions, methods and types that comprise your package's public API—its behaviour—is unimportant for the caller. The public API of a package describes *what* it does not *how* it does it. Moreover, when designed well, the *implementation* of your package is obscured from the caller

A good Go package should strive to have a low degree of source level coupling such that, as the project grows, changes to one package do not cascade across the code-base. These stop-the-world refactorings place a hard limit on the rate of change in a code base and thus the productivity of the members working in that code-base.

In this section we'll talk about designing a package around its behaviour as exposed via its public API.

7.1. A good package starts with its name

If the goal of a well designed Go package is to provide a set of related behaviours, writing a good package starts with choosing a good name. Think of your package's name as an elevator pitch to describe what it does, using just one word.

Each Go package is in effect it's own small Go program. You have access to all the symbols in this package, public and private (let's ignore external tests for the moment), and can use the all the features of the language on all parts of this package.

TIP Prototype packages in `internal/`. If you find some code that should be spun off into a lib, use the internal feature to prototype it.

But, when you combine those pieces, from the perspective of one module / function / package / piece, you want to take about the other *as abstractly as possible*.

A pkg's name should be a one word elevator pitch *for_what_it_provides* (not what it contains) https://twitter.com/davecheney/status/793306691370557440

A package should confirm to SRP, it should have a single purpose and a single responsiblity

https://blog.golang.org/package-names .

Models and other package names . They re fine, but not very reusable. Imagine the task of merging two projects, both with their own models directories. It should be clear that models is just a taxonomy, a place to put data, not reuse behavior

Initially it appeared attractive for every major piece of functinality to have a `testing` sub package

```
database/testing
client/testing
```

This package could contain mocks or helper functions.

The problem is most functional tests will need to import the mocks for more than one of these areas, causing a name clash on `testing`

As you have to rename the imports anyway,

```
import dbtesting "database/testing"
```

Why not start with that initially

```
package dbtesting
```

cf, no utils packages, cf, io/ioutils

A package's name should describe the function of the package, not the contents. `http` is a good package name, it describes that this package provides something to do with HTTP.

```go
package utils
```

is not as good name because it does not describe what this package provides. Providing "utility" functions is of no value to the reader, especially if every package author were to follow this trend.

```go
package client
```

is not a good name for two reasons

NOTE While the import statement referrs to

1. it suggests that the package is a client, but for what ?
2. even if imported as import "http/client" that

TIP Name your packages for what they provide, not what they contain

TIP *Prefer plurals for utility packages*
Utility package names should be plural, they provide helpers to work with things of a specific type.

7.1.1. Prefer lower case package names and import paths

The Go spefication does not define the meaning of the `import` declaration's argument, its *import path*. Tools like `go get` utilise this to allow it to fetch source directly from GitHub.

In practice, however, the import path is directly correlated to the name of a directory on disk that contains the source of the Go package being imported.

TIP

A package's name and it's import path are not required to be the same. However, this is very confusing. Don't do it.

Not all file systems handle casing the same. Apple's OSX treats the names of files in a case insensitive manner, you may refer to a file by any. This is known as case preserving case insensitive

Microsoft's Windows NTFS file system can record files who's name differ only in case. However, the operating system has great difficulty in

Case sensitivity issues are still a signficant cross platform issue and will limit the usefulness of you package by others. Package names and import paths must be lower case only.

TIP

Avoid renaming packages

A package's name should match its import path. Package names should be unique within your repository.

Never use dot imports

they were also a mistake Dot imports, import . "somepackage", obscure the origin of the symbols in somepackage making them Design package names well so users will not feel the need to rename them or use dot imports to obscure their name.

They are different, but should be treated the same as a courtesy to the user of the package, The entire history of trained and teaching about go tells programmers that importing x makes x symbols available in that file. Its just good manners,

Dot imports create a unique syntax used only in your file. Hurt readability, you don't know what is declared in your package and what is imported. Can create bizarre hard to debug issues where multiple symbols from unrelated packages appear in the same scope.

http://talks.golang.org/2014/organizeio.slide

Imports can be renamed, but this should not be your normal mode of operation. If you have named your packages to describe their function

Package layout

If your package contains a single source file, consider naming that file after the name of your package Corollary, if your package contains a single source file, don't add a `doc.go` just to hold package documentation. There is not call for that ceremony. Avoid doc.go,as an anti pattern I prefer starting with a single file with this name of the packages, place package doc and common symbols in that file, add additional files and split out from there after considering rule x and y (contitinal compilation) = Code should be factored so that changes don't ripple throughout the codebase If a package has to always import two other packages to be useful, consider combining those packages. Do not name your test helpers package, testing

Just as I talked earlier about nameing variables, the name of a package is very important. I start by asking myself questions like, "what is the purpose of this package" or "what does service does package provide?". Hopefully the answer to that question is "this package let's you speak HTTP", not "this package provides the X type", otherwise its time to go back to the drawing board.

TIP | Name your package for what it *provides*, not what it *contains*.

7.2. Good package names should be unique.

Within your project, each package name should be unique. This should pretty easy to if you've followed the previous advice that a package's name should derive from its purpose. If you find you have two packages which need the same name, it is likely either;

a. The name of the package is too generic-- `client`, `worker`, `shared`, etc.

b. The package overlaps another package of a similar name. In this case either you should review your design, or consider merging the packages, or renaming the conflicting packages to make their purpose more specific. Consider the `io/ioutil` and `net/http/httputil` packages as weak supporting evidence.

Provide something, cannot provide multiple things, ie http not https or http servers.

7.3. Avoid package names like `base`, `common`, or `util`

A common cause of poor package names is what I call *utility packages*. These are packages where helpers and utility code congeals over time. As these packages contain an assortment of unrelated functions, their utility is hard to describe in terms of what the package provides. This often leads to the package's name being derived from what the package *contains*-- utilities.

Package names like `utils` or `helpers` are commonly found in larger projects which have developed deep package hierarchies and want to share helper functions without encountering import loops. By extracting utility functions to new package the import loop is broken, but because the package stems from a design problem in the project its name doesn't reflect its purpose, only its function of breaking the import cycle.

My recommendation to improve the name of `utils` or `helpers` packages is to analyse where they are called and if possible move the relevant functions into their caller's package. Even if this involves duplicating some helper code this is better than introducing an import dependency between two packages.

> *[A little] duplication is far cheaper than the wrong abstraction.*
>
> — *Sandy Metz*

In the case where utility functions are used in many places prefer multiple packages, each focused on a single aspect, to a single monolithic package.

TIP | Use plurals for naming utility packages. For example the `strings` for string handling utilities.

Packages with names like `base` or `common` are often found when functionality common to two or more implementations, or common types for a client and server, has been refactored into a separate package. Their names also represent design holdovers from languages like Java and C++ where the relationship between packages followed similar rules to those of inheretence. I believe the solution to packeges like `base` or `common` is to reduce the number of packages, combine the client, server, and common code into a single package named after the behaviour delivered from the previously fractured packages.

For example, the `net/http` package does not have `client` and `server` sub packages, instead it has a `client.go` and `server.go` file, each holding their respective types, and a `transport.go` file for the common message transport code.

TIP

A public identifier includes its package name.

It's important to remember that the name of an identifier includes the name of its package.

- The `Get` function from the `net/http` package becomes `http.Get` when referenced by another package.
- The `Reader` type from the `strings` package becomes `strings.Reader` when imported into other packages.
- The `Error` interface from the `net` package is clearly related to network errors.

7.4. Avoid package level state

> *We claim that the non-1ocal variable is a major contributing factor in programs which are difficult to understand.*
>
> — *W. Wulf and Mary Shaw*
> *Global Variable Considered Harmful*
> *(http://www.informatik.uni-bremen.de/agbkb/lehre/programmiersprachen/artikel/wulf-shaw-global-variables-harmful.pdf)*

The key to writing maintainable programs is that they should be loosely coupled. A change to one package should have a low probability of affecting another.

In Go we can declare variables at the block, function, or method scope, and also at the package scope. When the variable is public—the identifier starts with a capital letter—then its scope is effectively global to the entire program. Any package may observe the type and contents of that variable *at any time*.

Mutable global state introduces tight coupling between independent parts of your program as global variables become an invisible parameter to every function in your program! Any function that relies on a global variable can be broken if that variable's type changes. Any function that relies on the state of a global variable can be broken if another part of the program changes that variable.

WARNING

Underscore, the side effect operator

Aliasing an imported package to , *known as a _side effect import*, serves only one purpose; to run `init` functions that affect global state.

- Without package level variables you don't need `func init()` either.
- Without package without package level variables, side effect imports would be almost meaningless

Specifically, while six possible declarations exist at the package level, consider restricting your programs to `package`, `import`, `const`, `type`, and `func`. By avoiding package level `var` declarations, you remove the opportunity for package global state to leak into your program.

NOTE Due to the limits of `const` declarations, some package level variables are unavoidable. In these cases, they are best treated as *effective* constant.

If you follow this advice then the use of the explicit `func init` becomes less useful, or at least less stateful.

> *You provide loggers as dependencies to components that need them: in constructors, or during struct initialization. Never, ever, as globals.*
>
> — *Peter Bourgon (https://twitter.com/peterbourgon/status/872077945355067392)*

If you want to reduce the coupling a global variable creates,

1. Move the relevant variables as fields on structs that need them.
2. Use interfaces to reduce the coupling between the behaviour and the implementation of that behaviour.

Avoid the use of global state, specifically avoid global side effects orchestrated by package level variables.

TODO: unfinished

7.5. Avoid leaking internal state

Avoid getter type methods because they break encapsulation. In the case where they are unavoidable—perhaps because of a poorly designed interface—be wary that you do not leak a reference to an internal data structure which can be retained and mutated long after the original getter was accessed.

For example, many would recognise the dangers with the following function

```go
func (u *User) FirstName() *string {
    return u.first
}
```

By retaining a reference to `u.first`, the caller can both observe the state of `u.first` without going through `FirstName` and can, if it wishes, mutate `u.first`. The difficulties with this design are obvious.

Somehwhat less obvious are accessors like this

```go
func (u *User) Siblings() []*User {
    return u.siblings
}
```

Slices, as we know, are value types, however the first field in a slice is a pointer to its *backing array*.

While the value returned from `Siblings` is a *copy* of `u.siblings`, its backing array pointes to the `u.siblings` array—the share the same backing array.

This means, assuming `u.siblings` is not being appended too, the *copy* returned from `Siblings` can be used to access and mutate `u’s `siblings` slice. In the case that other code appends to `u.siblings` at some point the various copies returned from `Siblings` may point to *different* backing arrays.This is perhaps, although this might be seem to be splitting hairs, a worse outcome. The situation has evovled from a simple data race, to a non deterministic data race.

The solution, I belive is twofold

1. Return a slice pointing to a copy of the backing array; an entirely indepdenant slice

```
func (u *User) Siblings() []*Users {
    return append([]*Users{}, u.siblings...)
}
```

Which may present a performance issue depending on the size, frequency, and use of `Siblings`.. Refactor the code to reflect the use of the *result* of `Siblings`.

8. Project Structure

TIP Prototype packages in `internal/`. If you find some code that should be spun off into a lib, use the internal feature to prototype it.

Go code expects to be placed in a directory hierarchy, from which it determines each package's absolute import path.

The go get convention requires that a subdirectory of this path be checked in, strippibg the information required to reconstruct the absolute import path of a package.

Prefer fewer, larger packages, a packages name is a one word elevator pitch for what it provides (does), not what it contains.

You want to compose your programs out of small composible pieces — java shop politics

But, when you combine those pieces, from the perspective of one module / function / package / piece, you want to take about the other *as abstractly as possible*.

TIP *Repository layout*
Every package should contain some source code. Exception, the cmd/ directory pattern.

My take away from watching Blake's presentation is dependencies are not free, just as every line of code you write is a liability, so too is every package you import. They both have to pay their way. Specifically testing frameworks, sure, it may appear annoying to write an if statement when you can imagine a nice little assert function provided by a library. But what has happened is you have gone from every single go programmer who understands if statements at an organic level to a smaller set of go programmers who have experience with the specific testing framework you have chosen (insert xkcd reference about standards) So while you may want

```go
t.Assert(s != "")
```

Someone else will think

```go
t.NotEmpty(s)
```

Is clearly superior. The bottom line is these are all little dialects, little languages built in go, that don't pay their way over just writing the damn check and knowing that every single living breathing go programmer will be able to understand it, first time, every time. And that is what I took from Blake's talk, that adding dependencies for trivial functions can, and sometimes does not, repay the cost of the complexity they introduce.

> *My major concern here is the Principle of One Right Place - there should be One Right Place to look for any nontrivial piece of code, and One Right Place to make a likely maintenance change.*
>
> — *Pj Plauger*
> *programming on purpose*

Let's talk about combining packages together into a project. Commonly this will be a single git repository. In the future Go developers will use the terms *module* and *project* interchangeably.

Just like a package, each project should have a clear purpose. If your project is a library, it should provide one thing, say XML parsing, or logging. You should avoid combining multiple purposes into a single project, this will help avoid the dreaded `common` library.

TIP: In my experience, the `common` repo ends up tightly coupled to its biggest consumer and that makes it hard to back-port fixes without upgrading both common and consumer in lock step, bringing in a lot of unrelated changes and API breakage along the way.

- Package, just some code
- Main package, something you deliver
- Library a set of packages that some one else depends on
- Module ???
- Project, a set of main packages that includes one or more

8.1. Eschew elaborate package hierarchies

Go eschewed type hierarchy, and that is generally considered to be a good thing, don't make the mistake or replacing that with an elaborate package hierarchy. That will lead to too many internal types being made public. If you have two packages that are always imported together, maybe combine them. Alternatively, if you have one package that is never imported directly, only via a third, maybe roll it up.

`cmd` has a long legacy, going all the way back to the arrangement of the 1972 unix source code.

8.2. Consider fewer, larger packages

One of the things I tend to pick up in code review for programmers who are transitioning from other languages to Go is they tend to overuse packages. To be fair, mastery of Go is effectively the art of moderation in the use of all Go's features, but package overuse seems to be one of the most common misstep. I suspect driven by programmers' innate drive to categorise and neatly organise the things they see.

Go does not provide elaborate ways of establishing visibility. Go lacks Java's `public`, `protected`, `private`, and implicit `default` access modifiers. There is no equivalent of C++'s notion of a `friend` classes.

In Go we have only two access modifiers, public and private, the former indicated by the capitalisation of the first letter of the identifier. If an identifier is public, it's name starts with a capital letter, that identifier can be referenced by *any* other Go package.

NOTE You may hear people say *exported* and *not exported* as synonyms for public and private.

Given the limited controls available to control access to a package's symbols, what practices should Go programmers follow to avoid creating over-complicated package hierarchies?

The advice I find myself repeating is to prefer fewer, larger packages. Your default position should be to not create a new package. That will lead to too many types being made public creating a wide, shallow, API surface for your package..

The sections below explores this suggestion in more detail.

TIP Every package, with the exception of `cmd/` and `internal/`, should contain some source code.

NOTE Possibly because of the early use of a `pkg/` directory to hold package—and the corresponding `cmd/` directory to hold commands (`package main`) this practice of putting your packages in an empty `pkg/` directory has spread to other Go projects. This practice was never a recommendation, just a result of the original `Makefile` based build system.

In September 2014, the stdlib moved away from storing package code in an otherwise empty `pkg/` directory, and you should follow their lead. Other than a superficial symetary with `cmd/` putting packages in a `pkg/` directory is needless boilerplate and distracts from the potentially more useful `internal/` directory.

TIP *Coming from Java?*

One file per type is mandated by Java. Go isn't Java, this is not required. If you're coming from a Java or C# background, consider this guideline.

- A Java package is equivalent to a single `.go` source file.
- A Go package is equivalent to a whole Maven module or .NET assembly.

TIP *Repository design*

Every package should contain some source code. Exception, the cmd/ directory pattern.

Consider fewer source code files per package where practical.

8.3. Arrange files by import statements

If you're arranging your packages by what they provide to callers, should you do the same for files within a Go package? How do you know when you should break up a `.go` file into multiple ones? How do you know when you've gone to far and should instead consolidate several `.go` files together?

Here are the guidelines I use:

- Start each package with one `.go` file. Give that file the same name as the name of the folder. For example the source for `package http` should be placed in a file called `http.go` in a directory named `http`.
- As your package grows you may decide to split apart the various *responsibilities* into different files. eg, `messages.go` contains the `Request and Response` types, `client.go` contains the `Client` type, `server.go` contains the `Server` type.
- If you find your files have similar `import` declarations, consider combining them. Alternatively, identify the differences between the import sets and move those types/functions/methods into their own file.
- Different files should be responsible for different areas of the package. `messages.go` may be responsible for marshalling of HTTP requests and responses on and off the network, `http.go` may contain the low level network handling logic, `client.go` and `server.go` implement the HTTP business logic of request construction or routing, and so on.

NOTE

The Go compiler compiles each package in parallel. Within a package the compiler compiles each *function* (methods are just fancy functions in Go) in parallel. Changing the layout of your code within a package should not affect compilation time.

TIP

Avoid elaborate package hierarchies

With one exception, which we'll talk about next, the hierarchical directory structure a Go project has no meaning to the `go` tool. For example, the `net/http` package is *not* a child or sub-package of the `net` package.

Go eschewed elaborate type hierarchy. This is generally considered to be a good thing. Don't make the mistake or replacing that with an elaborate package hierarchy. If you find you have created intermediate directories in your project which contain no `.go` files, you may have fallen afoul of the desire to create a taxonomy of your source code.

My rule of thumb for splitting up a large file is to use imports as a guideline. That is, all the things that import network stuff go in one file, all the thing importing strings/regex and parsing go in another and so on.

This tends to also make testing more straight forward. If you have a file, say, `conn.go`, that deals with network type stuff, then its counterpart `conn_test.go` should deal only with testing the network functionality of this package. If you didn't you've got parsing or business logic tests in that file as well, that's a sign that your design isn't right, or that you carved your file along the wrong boundary.

Obviously some judgement needs to be applied as packages like `fmt` and `errors` tend to be used everywhere. Also, don't take this to the extreme and split too much, otherwise you'll replace the problem of a large file with the problem of constantly searching a directory of files. If your package starts to look like Java, with one function or type per file then you've gone to far and it's time to consolidate. Again, imports help guide you here, if you find two files with identical imports and related functionality, then try meeting them.

TIP

Prefer nouns for source file names. They are containers for source code after all. But, ensure those nouns describe the *purpose*, not the *contents*. It is not necessary to place each type or function in its own file, especially when that file's name becomes a repetition of its singular contents.

How do we know when files should be sperate?::When they have different import sets. How should we know when files should be combined?::When they have the same import sets. How should we know when do have more than one file?::When it gets too long, when it expresses too many unrelated ideas.

8.4. Use `internal` packages to reduce your public API surface

If your project contains multiple packages you may find you have some exported functions which are intended to be used by other packages in your project, but are not intended to be part of your project's public API. If you find yourself in this situation the `go` tool recognises a special folder name—not package name-- `internal/` which can be used to indicate code which is public to your project, but private to others.

To create an internal package, place it within a directory named `internal/` or in a sub-directory of a directory named `internal/`. When the `go` command sees an import of a package with `internal` in its path, it verifies that the importing package is within the tree rooted at the *parent* of the `internal` directory.

For example, a package `…/a/b/c/internal/d/e/f` can be imported only by code in the directory tree rooted at `…/a/b/c`. It cannot be imported by code in `…/a/b/g` or in any other repository. [9]

8.5. Keep package main as small as possible

If you're designing an application—one that will will span multiple packages—your `main` function, and `main` package should do as little as possible. This is because `main.main` acts as a singleton; there can only be one `main` function in a program.

Main packages are harder to test. Instead extract all the functionality to other packages, possibly `internal/`, and leave main for - flag parsing - constructing the top level object - managing the top level lifetime of components

Because `main.main` is a singleton there are a lot of assumptions built into the things that `main.main` will call, that they will only be called during `main.main` or `main.init`, and only called *once*. This makes it hard to write tests for code written in `main.main`. Main packages often invoke singletons, parse command line flags, expect files to be on disk in a certain place, and never expect to be executed concurrently. You can't even reference `main.main` from a test.

Thus you should aim to extract as much of your business logic out of your main function and ideally out of your main package. `func main()` should parse flags, open connections to databases, loggers, and such, then hand off execution to a high level object.

TIP

One command per package
Commands should do flag handling, and be prepared to instantiate the service or command. This let's you build binaries that perform one job, or many jobs, or even all jobs specifically if you use the argv trick

8.6. Resist the desire to apply taxonomy

Every package should have a unique name within your project, within the code that you are writing. This means that when multiple packages are imported into your package, they will not conflict.

As your project grows, you will naturally have to add more packages (but please consider the other topics in this chapter)

don't do

```go
import (
    "github.com/foo/datamodel/banana"
    "github.com/foo/controllers/banana"
)
```

This package organisation highlights a fundamental issue with the design. This also speaks to writing larger packages and writing less packages.

Go packages are not for creating taxonomies, `net/http` is not a type of `net` thing. Go packages are for avoid namespace collisions.

> **TIP** Avoid empty packages. An empty package is a sign you developing a package hierarchy. An exception to this rule is the use of grouping main packages into a `cmd/` directory.

As an aside, I find that many programmers have a strong desire to aptly taxonomies to their work, they want to find *differences* between two things so they can be separated and classified. I think to write successful Go, you should look for similarities and combine things with a similar purpose into a package. For example, net/http contains both http client and server facilities, because they both relate to the use of http. The prevailing style in other languages would have these placed in separate packages, one for client, one for server, and a third for things which are common to both. I think this is unnecessary.

9. Error handling

Error handling is important for reliable programs. Error handling is as important as the rest of your code. Error handling is as important as checking a loop index for the exit condition, or checking the result of a shift operation, or testing the result of a multiplication is within the expected bounds, that’s how fundamental error handling is to Go. And, just like shifting or comparisons or multiplication, error handling is a first class responsibility of all Go programmers. So important that Go makes it a first class citizen. Because, you have to plan for failure.

When you write to a network, assume the other side never gets the request. When you write to a channel, assume the other side never picks up the write.

The `error` interface is the key to Go’s composable error handling story. If the error is `nil`, the call worked.

> **TIP** *Never use any other variables without checking the error*
> You only need to check the error value if you care about the result. Said another way, why check if there is nothing you can do?

At a higher level, I think there are two kinds of schools of error handling, programmers who think they can fix the error and continue on, and programmers who recognise that errors happen and to give up safe in the knowledge that some other piece of code will retry the operation. Basically it’s a question of do you write software that expects almost everything to work, or software that expects almost nothing to work. I put myself in the second category, because I believe that is the best way to write robust software.

If you need confirmation, you must confirm. So you don’t focus on the happy path.

The `error` interface is the key to Go’s composable error handling story. If the error is `nil`, the call worked.

TIP

Try / catch are not sufficient

Today, the common concensus is checked exceptions were a mistake. As Bruce Eckel said on his closing keynote at GeeCON, Prague, no other language after Java has engaged in using checked exceptions, and even Java 8 does no longer embrace them in the new Streams API (which can actually be a bit of a pain, when your lambdas use IO or JDBC).

- Exceptions obfuscate control flow just as badly as return codes.
- Reliable software cannot be written with unchecked exceptions, yet checked exceptions have never been repeated by any other language other than java.

Programmers can get so distracted by the `if err != nil { return err }` dance that they missing the *most* important part of error handleding happens just before `return err`, when you recover and clean up.

9.1. Errors are just values

> *Errors are just values.*
>
> — *Rob Pike*
> *Go Proverbs (https://go-proverbs.github.io/)*

This statement is almost universal in the Go programmer's phrase book, but what do Go programmers mean when they say "errors are just values", and what does this technique imply? By way of explanation, consider the counter example of `panic` and `recover`, often mistaken for exceptions.

`panic` and `recover`, two keywords added to the language for a single purpose. `recover` can only be used for one purpose; to access a value previously passed to `panic`. If that wasn't enough `recover's use case is so specific, it can only be used inside a `defer` block. You cannot use `recover` for any other purpose, it can only be used in concert with `panic`.

This pair of features sit by themselves in a corner of the language. How's that for non orthogonal?

By contrast, error values are not limited to the rarefied semantics of `panic` and `recover`.

9.1.1. Errors should be opaque

> *With a sufficient number of users of an API, it does not matter what you promise in the contract, all observable behaviours of your system will be depended on by somebody.*
>
> *— Hyram's Law [10]*
>
> — *Joe Tsai*
> *GopherCon 2017*

Programmers will rely on whatever behaviour, guaranteed or not, they observe from your API. Simply put, the more observable state your API returns, the larger the yoke of backwards compatibility you are implicitly committing to.

To the caller, the type and contents of an error value, if not `nil`, should be considered opaque. To do otherwise introduces brittle coupling between the function and its caller.

The exception to this rule are are sentinel values like `io.EOF`. These are however, the exception to the rule, not a pattern to be emulated.

TIP

Almost all errors terminate processing. Almost all errors are opaque. Design APIs that take advantage of these properties.

TIP

Return values are opaque until you've checked the error value

Distinguish between error and failure. An error is opaque, a failure is the higher level abstraction that the request succeeded but its operation failed. Don't mix errors at different levels.

NOTE

Issue 12866 (https://github.com/golang/go/issues/12866) is an example of an unintended consequence of overspecifying the error value returned.

9.1.2. Avoid overloaded errors

Some errors are not actually errors

Must check error value before assuming the state of any other return values. `io.Reader` is the odd man out but still follows the rule. Reads exception is necessary to avoid forcing state down to the underlying reader, but still unorthodox

NOTE

`io.EOF` isn't actually an error, so it shouldn't be wrapped. It's just a convenient way to indicate the non-error state of end of file.

Just as error values themselves are opaque to the caller, until the error value itself has been checked, the caller must not assume anything about the state of any other return values. The methodology I follow is; if a function can return an error, you cannot make any assumptions about the state of any other values returned until you check the error. If it was found that the error was set (ie, not nil), then the state of those other values is unknown.

9.2. Assert errors for behaviour, not type

The common contract for functions which return a value of the interface type `error`, is the caller should not presume anything about the state of the other values returned from that call without first checking the error. In the majority of cases, error values returned from functions should be opaque to the caller. That is to say, a test that error is `nil` indicates if the call succeeded or failed, and that's all there is to it.

The methodology I follow is; if a function can return an error, you cannot make any assumptions about the state of any other values returned until you check the error. If it was found that the error was set (ie, not `nil`), then the state of those other values is *unknown*.

A small number of cases, require that the caller investigate the nature of the error to decide if it is reasonable to retry the operation. A common request for package authors is to return errors of a known public type, so the caller can type assert and inspect them. I believe this practice leads to a number of undesirable outcomes:

1. Public error types increase the surface area of the package's API.
2. New implementations must only return types specified in the interface's declaration, even if they are a poor fit. This also introduces coupling. My implementation must import the package that declares the specific error type required.

3. The error type cannot be changed or deprecated after introduction without breaking compatibility, making for a brittle API.

You should feel no more comfortable asserting an error is a particular type than they would be asserting the `string` returned from `Error()` matches a particular pattern.

Assert that error value implements a particular behaviour. Don't assert an error value is a specific type, but rather assert that the value implements a particular behaviour.

Instead I present a suggestion that permits package authors and consumers to communicate about their intention, without having to overly couple their implementation to the caller. This suggestion fits the *has a* [behaviour] nature of Go's implicit interfaces, rather than the *is a* [subtype of] nature of inheritance based languages. Consider this example:

```go
func isTimeout(err error) bool {
        type timeout interface {
                Timeout() bool
        }
        te, ok := err.(timeout)
        return ok && te.Timeout()
}
```

The caller can use `isTimeout` to determine if the error is related to a timeout, and if so confirm if the error was timeout related, all without knowing anything about the type, or the original source of the `error` value.

Gift wrapping errors, usually by libraries that annotate the error path, is enabled by this method; providing that the wrapped error types also implement the interfaces of the error they wrap. This may seem like a generally intractable problem, but in practice there are relatively few interface methods that are in common use, so `Timeout() bool` and `Temporary() bool` cover a large set of use cases.

For package authors, if your package generates errors of a temporary nature, ensure you return error types that implement the respective interface methods. If you wrap error values on the way out, ensure that your wrappers respect the interface(s) that the underlying error value implemented.

For package users, *if* you need to inspect an error—and hopefully this should be infrequent—declare and assert an interface to assert the behaviour you expect, not the error's type. Don't ask package authors for public error types; instead ask that they make their types conform to common interfaces as appropriate.

TIP Don't assert an error value is a specific type, but rather assert that the value implements a particular behaviour.

9.2.1. Decouple error handling from type

Does the code that calls your function care about the type or value of the error returned? Would returning different errors alter the logic of the calling code? If so then you have a tight coupling between caller and your code, and your error types or values are part of the contract for that function or method.

My recommendation, is to write the call in such a way that it doesn't need to know the specifics of any error value returned, they are implementation details, instead write your caller to simply. If there is an error, execute any cleanup, and return the error to the caller.

9.3. Don't panic

> *Don't panic.*
>
> — *Rob Pike*
> *Go Proverb*

The simple rule of thumb is panic should only be used in truely exceptional cases, which, as their name suggests are rare. To paraphrase the words of Dash Parr, if everything is exceptional, the nothing is.

Go's error handling strategy is via the `error` interface and returning `error` values. Go does have `panic`, which is a by-product of the counterpart in the runtime's internal `throw` function. There are few cases of using `recover` that I know of, and all of those are used to simulate non local transfer of control *not* exception handling. Using recover has all the problems of sensing errors by type, with the added complication that the set of types returned is unbounded.

While it is true that any Go function can call `panic`, any Go procedure can fail due to out of memory, the program can be killed by a process manager, or the serve can simply fail. Always write your programs to assume failure, not success. Avoid `panic` and eschew `recover`, they're not the tool you are looking for.

Panic will, during unwinding the stack, execute any deferred statements. However just as a panic in one goroutine cannot be recovered in another, a panic in one goroutine will not allow defer statements in other goroutines to exit. For a goroutine spawned by a library to panic the entire program is selfish.

9.3.1. Avoid selfish panics

If a function or method returns an error value, there is no call for a `panic`. `panic` must be truly the last resort; exiting on impossible conditions, or in scenarios where the applications truly cannot recover. Panicing in a library must be the absolute last resort. Not only does it have direct impact on the reliability of the program your code is embedded into, but engenders a belief that your library is hard to work with, or itself unreliable.

Panic will, during unwinding the stack, execute any deferred statements. However just as a panic in one goroutine cannot be recovered in another, a panic in one goroutine will not allow defer statements in other goroutines to exit. For a goroutine spawned by a library to panic the entire program is selfish and *must* be avoided.

TIP

The common party line is panics, if used, should not leak beyond the API boundary. I would strengthen this statement by simply saying, *do not use panic in library code*.

9.3.2. Avoid log.Fatal

The `log` package provides two ways to exit your program, `log.Fatal` and `log.Panic`. These are effectively the same as panic, and the same rules for panic should apply. They were a mistake and should not have been added. The convenience of being able to log and crash the program in one line, not two, created a misleading precident.

NOTE

Faults vs Failures

- a fault is something you recovered from, you handled it
- a failure is something that cannot handle and has propogated to the caller

9.4. Eliminate error handling by eliminating errors

As part of the Go 2 design goals, improvements (although few can agree on what improve means) to error handling are highlighted as a candidate. This surfaced in 2018 with the `check` / `handle` proposal, which was later refined into the highly contentious `try` proposal. But do you know what is better than an improved syntax for handling errors? Not needing to handle errors at all.

This section draws inspiration from John Ousterhout's book, A philosophy of Software Design [11]. One of the chapters in that book is called "Define Errors Out of Existence".

One of Osterhout's examples was the operation of UNIX's `write(2)` API, specifically that if an error occurs during writing, you can perhaps ignore it as the error will surface during `close(2)` on that file. This example plays a little fast and loose with POSIX, but illustrates the idea; done well, error handling can be repetitive, thus, where possible, use the emergent properties of the problem to avoid error handling where possible.

NOTE I'm not saying "remove your error handling". What I am suggesting is; change your code so you do not have so many errors to handle.

We'll see an example of Osterhout's write/close solution a little later as I try to apply his advice to Go.

9.4.1. Counting lines

Let's write a function to count the number of lines in a file.

GO
```
func CountLines(r io.Reader) (int, error) {
    var (
        br    = bufio.NewReader(r)
        lines int
        err   error
    )

    for {
        _, err = br.ReadString('\n')
        lines++
        if err != nil {
            break
        }
    }

    if err != io.EOF {
        return 0, err
    }
    return lines, nil
}
```

Because we're following our advice from previous sections, `CountLines` takes an `io.Reader`, not a `*os.File`; its the job of the caller to provide the `io.Reader` who's contents we want to count.

We construct a `bufio.Reader`, and then sit in a loop calling the `ReadString` method, incrementing a counter until we reach the end of the file, then we return the number of lines read.

At least that's the code we want to write, but instead this function is made more complicated by error handling. For example, there is this strange construction,

GO
```
        _, err = br.ReadString('\n')
        lines++
        if err != nil {
            break
        }
```

We increment the count of lines *before* checking the error—that looks odd.

The reason we have to write it this way is `ReadString` will return an error if it encounters and end-of-file before hitting a newline character. This can happen if there is no final newline in the file.

To try to fix this, we rearrange the logic to increment the line count, then see if we need to exit the loop.

NOTE	this logic still isn't perfect, can you spot the bug?

But we're not done checking errors yet. `ReadString` will return `io.EOF` when it hits the end of the file. This is expected, `ReadString` needs some way of saying *stop, there is nothing more to read*. So before we return the error to the caller of `CountLine`, we need to check if the error was *not* `io.EOF`, and in that case propagate it up, otherwise we return `nil` to say that everything worked fine.

I think this is a good example of Russ Cox's observation that error handling can obscure the operation of the function. Let's look at an improved version.

GO
```go
func CountLines(r io.Reader) (int, error) {
    sc := bufio.NewScanner(r)
    lines := 0

    for sc.Scan() {
        lines++
    }
    return lines, sc.Err()
}
```

This improved version switches from using `bufio.Reader` to `bufio.Scanner`.

Under the hood `bufio.Scanner` uses `bufio.Reader`, but it adds a nice layer of abstraction which helps remove the error handling with obscured the operation of `CountLines`.

NOTE	`bufio.Scanner` can scan for any pattern, but by default it looks for newlines.

The method, `sc.Scan()` returns `true` if the scanner *has* matched a line of text and *has not* encountered an error. So, the body of our `for` loop will be called only when there is a line of text in the scanner's buffer. This means our revised `CountLines` correctly handles the case where there is no trailing newline, and also handles the case where the file was empty.

Secondly, as `sc.Scan` returns `false` once an error is encountered, our `for` loop will exit when the end-of-file is reached or an error is encountered. The `bufio.Scanner` type memoises the first error it encountered and we can recover that error once we've exited the loop using the `sc.Err()` method.

Lastly, `sc.Err()` takes care of handling `io.EOF` and will convert it to a `nil` if the end of file was reached without encountering another error.

TIP	When you find yourself faced with overbearing error handling, try to extract some of the operations into a helper type.

9.4.2. WriteResponse

My second example is inspired from the *Errors are values* blog post by Rob Pike.[12]

Earlier in this presentation We've seen examples dealing with opening, writing and closing files. The error handling is present, but not overwhelming as the operations can be encapsulated in helpers like `ioutil.ReadFile` and `ioutil.WriteFile`. However when dealing with low level network protocols it becomes necessary to build the response directly using I/O primitives the error handling can become repetitive. Consider this fragment of a HTTP server which is constructing the HTTP response.

GO
```go
type Header struct {
    Key, Value string
}

type Status struct {
    Code   int
    Reason string
}

func WriteResponse(w io.Writer, st Status, headers []Header, body io.Reader) error {
    _, err := fmt.Fprintf(w, "HTTP/1.1 %d %s\r\n", st.Code, st.Reason)
    if err != nil {
        return err
    }

    for _, h := range headers {
        _, err := fmt.Fprintf(w, "%s: %s\r\n", h.Key, h.Value)
        if err != nil {
            return err
        }
    }

    if _, err := fmt.Fprint(w, "\r\n"); err != nil {
        return err
    }

    _, err = io.Copy(w, body)
    return err
}
```

First we construct the status line using `fmt.Fprintf`, and check the error. Then for each header we write the header key and value, checking the error each time. Lastly we terminate the header section with an additional `\r\n`, check the error, and copy the response body to the client. Finally, although we don't need to check the error from `io.Copy`, we need to translate it from the two return value form that `io.Copy` returns into the single return value that `WriteResponse` returns.

That's a lot of repetitive work. But we can make it easier on ourselves by introducing a small wrapper type, `errWriter`.

`errWriter` fulfils the `io.Writer` contract so it can be used to wrap an existing `io.Writer`. `errWriter` passes writes through to its underlying writer until an error is detected. From that point on, it discards any writes and returns the previous error.

GO

```
type errWriter struct {
    io.Writer
    err error
}

func (e *errWriter) Write(buf []byte) (int, error) {
    if e.err != nil {
        return 0, e.err
    }
    var n int
    n, e.err = e.Writer.Write(buf)
    return n, nil
}

func WriteResponse(w io.Writer, st Status, headers []Header, body io.Reader) error {
    ew := &errWriter{Writer: w}
    fmt.Fprintf(ew, "HTTP/1.1 %d %s\r\n", st.Code, st.Reason)

    for _, h := range headers {
        fmt.Fprintf(ew, "%s: %s\r\n", h.Key, h.Value)
    }

    fmt.Fprint(ew, "\r\n")
    io.Copy(ew, body)
    return ew.err
}
```

Applying `errWriter` to `WriteResponse` dramatically improves the clarity of the code. Each of the operations no longer needs to bracket itself with an error check. Reporting the error is moved to the end of the function by inspecting the `ew.err` field, avoiding the annoying translation from `io.Copy's return values.

9.5. Only handle an error once

To close this chapter, I want to recommend that your code should only handle errors once.

- Handling an error means inspecting the error value, and making a *single* decision.
- Handling an error may include retrying the operation, or an alternative path.
- Handling an error may include logging the error, in which case the error is handled and should not be returned to the caller. Specifically, this error is logged a `INFO` level and is by defintion not an error—you just handled it. If you cannot handle the error, and have exhausted all possible alternatives, then return the error to the caller and let them deal with it.

GO

```
// WriteAll writes the contents of buf to the supplied writer.
func WriteAll(w io.Writer, buf []byte) {
        w.Write(buf)
}
```

If you make less than one decision, you're ignoring the error. As we see here, the error from `w.WriteAll` is being discarded.

But making *more than one* decision in response to a single error is also problematic. The following is code that I come across frequently.

GO
```go
func WriteAll(w io.Writer, buf []byte) error {
    _, err := w.Write(buf)
    if err != nil {
        log.Println("unable to write:", err) // annotated error goes to log file
        return err                           // unannotated error returned to caller
    }
    return nil
}
```

In this example if an error occurs during `w.Write`, a line will be written to a log file, noting the file and line that the error occurred, and the error is also returned to the caller, who possibly will log it, and return it, all the way back up to the top of the program.

The caller is probably doing the same

GO
```go
func WriteConfig(w io.Writer, conf *Config) error {
    buf, err := json.Marshal(conf)
    if err != nil {
        log.Printf("could not marshal config: %v", err)
        return err
    }
    if err := WriteAll(w, buf); err != nil {
        log.Println("could not write config: %v", err)
        return err
    }
    return nil
}
```

So you get a stack of duplicate lines in your log file,

```
unable to write: io.EOF
could not write config: io.EOF
```

but at the top of the program you get the original error without any context.

GO
```go
err := WriteConfig(f, &conf)
fmt.Println(err) // io.EOF
```

I want to dig into this a little further because I don’t see the problems with logging *and* returning as just a matter of personal preference.

GO
```go
func WriteConfig(w io.Writer, conf *Config) error {
    buf, err := json.Marshal(conf)
    if err != nil {
        log.Printf("could not marshal config: %v", err)
        // oops, forgot to return
    }
    if err := WriteAll(w, buf); err != nil {
        log.Println("could not write config: %v", err)
        return err
    }
    return nil
}
```

The problem I see a lot is programmers forgetting to return from an error. As we talked about earlier, Go style is to use guard clauses, checking preconditions as the function progresses and returning early.

In this example the author checked the error, logged it, but *forgot* to return. This has caused a subtle bug.

The contract for error handling in Go says that you cannot make any assumptions about the contents of other return values in the presence of an error. As the JSON marshalling failed, the contents of `buf` are unknown, maybe it contains nothing, but worse it could contain a half written JSON fragment.

Because the programmer forgot to return after checking and logging the error, the corrupt buffer will be passed to `WriteAll`, which will probably succeed and so the config file will be written incorrectly. However the function will return just fine, and the only indication that a problem happened will be a single log line complaining about marshalling JSON, *not* a failure to write the config.

Either handle an error, or return it—never do both.

10. Concurrency

> *Concurrency is about dealing with a lot of things at once. Parallelism is about doing a lot of things at once.*
>
> — *Rob Pike*

Concurrency is not parallism. Concurrency describes a methodology for composing applications from indepdenat processes. These processes may execute asynchonusly, or in parallel, or sequentially, but that is not important, and not a definitaion of *concurrency*.

Often Go is chosen for a project because of its concurrency features. The Go team have gone to great lengths to make concurrency in Go cheap (in terms of hardware resources) and performant, however it is possible to use Go's concurrency features to write code which is neither performant or reliable. With the time I have left I want to leave you with some advice for avoid some of the pitfalls that come with Go's concurrency features.

Go features first class support for concurrency with channels, and the `select` and `go` statements. If you've learnt Go formally from a book or training course, you might have noticed that the concurrency section is always one of the last you'll cover. This workshop is no different, I have chosen to cover concurrency last, as if it is somehow additional to the regular the skills a Go programmer should master.

There is a dichotomy here; Go's headline feature is our simple, lightweight concurrency model. As a product, our language almost sells itself on this feature alone. On the other hand, there is a narrative that concurrency isn't actually that easy to use, otherwise authors wouldn't make it the last chapter in their book.

10.1. Channels

Channels are a signature feature of the Go programming language. Channels provide a powerful way to reason about the flow of data from one goroutine to another without the use of locks or critical sections.

I want to talk about two important properties of channels that make them useful for controlling not just data flow within your program, but the flow of control as well.

10.1.1. Channel ownership

Channels, by their nature as *the* mechanism to communicate across goroutines are somewhat unique in that they the only runtime provided mutable shared values.

Because channel value is shared by multiple goroutines, its useful to ask the question, who, if anyone owns the channel. Owernship is important because while multiple goroutines can send on a channel, only one can close it.

Closing a channel is a *level triggered* signal to all other goroutines. A channel can only be closed once, further attempts will panic. This seems incongrious with other things that exhibit closeable behavior, like files, database connections, and so on. Those are resources, a channel is not. While you should always close a file or network connection, there is no requirement to close a channel — a channel will be garbage collected, just like any other variable, when no references to it exist. Furthermore, a send on a closed channel will panic.

TIP You do not need to close a channel for it to be garbage collected, that will happen once every reference to your channel has been discarded.

From this we can derrive some rules of channel ownership

- Only the owner of a channel should close it
- Only the owner of a channel may write to it.
- If a channel has more than one writer, then none of them are the owner, so none may close the channel.

When constructing programs that communicate via channels, the channel's state, open or closed, is an important property.

10.1.2. Familiarise yourself with basics of channels

Most new Go programmers quickly grasp the idea of a channel as a queue of values and are comfortable with the notion that channel operations may block when full or empty. However there are for less well known, but more fundamental properties of channels:

1. Sending to a nil channel blocks forever.
2. Receiving from a nil channel blocks forever.
3. Sending to a closed channel panics.
4. Receiving from a closed channel returns the zero value immediately.

A send to a nil channel blocks forever

The first case which is a little surprising to newcomers is a send on a `nil` channel, a channel value that has not been initalised or has been set to `nil`, blocks forever.

For example

```go
func main() {
    var c chan string
    c <- "let's get started" // (1)
}
```

1. deadlock

This example program will deadlock on line 5 because the zero value for an uninitialised channel is `nil`. You cannot send to a channel that has not been initialised.

A receive from a nil channel blocks forever

Similarly receiving from a nil channel blocks the receiver forever.

GO
```go
func main() {
    var c chan string
    fmt.Println(<-c) // (1)
}
```

1. deadlock

So why does this happen? Here is one possible explanation:

- The size of a channel's buffer is not part of its type declaration, so it must be part of the channel's value.
- If the channel is not initialised then its buffer size will be zero.
- If the size of the channel's buffer is zero, then the channel is unbuffered.
- If the channel is unbuffered, then a send will block until another goroutine is ready to receive (https://golang.org/ref/spec#Send_statements).
- If the channel is `nil` then the sender and receiver have no reference to each other; they are both blocked waiting on independent channels and will never unblock.

This might not seem important, but is a useful property when you want to use the closed channel idiom to wait for multiple channels to close. For example

GO
```go
// WaitMany waits for a and b to close.
func WaitMany(a, b chan bool) {
    var aclosed, bclosed bool
    for !aclosed || !bclosed {
        select {
        case <-a:
            aclosed = true
        case <-b:
            bclosed = true
        }
    }
}
```

`WaitMany` looks like a good way to wait for channels a and b to close, but it has a problem. Let's say that channel a is closed first, then it will always be ready to receive. Because `bclosed` is still false the program can enter an infinite loop, preventing the channel b from ever being closed.

A safe way to solve the problem is to leverage the blocking properties of a `nil` channel and rewrite the program like this:

GO

```go
package main

import (
    "fmt"
    "time"
)

func WaitMany(a, b chan bool) {
    for a != nil || b != nil {
        select {
        case <-a:
            a = nil
        case <-b:
            b = nil
        }
    }
}

func main() {
    a, b := make(chan bool), make(chan bool)
    t0 := time.Now()
    go func() {
        close(a)
        close(b)
    }()
    WaitMany(a, b)
    fmt.Printf("waited %v for WaitMany\n", time.Since(t0))
}
```

In the rewritten `WaitMany` we set the reference to `a` or `b` to `nil` as soon as they have received a value. When a `nil` channel is part of a select statement, it is effectively ignored, so niling a removes it from selection, leaving only b which blocks until it is closed, exiting the loop without spinning.

A send to a closed channel panics

The following program will likely panic as the first goroutine to reach 10 will close the channel before its siblings have time to finish sending their values.

GO

```go
func main() {
        var c = make(chan int, 100)
        for i := 0; i < 10; i++ {
                go func() {
                        for j := 0; j < 10; j++ {
                                c <- j
                        }
                        close(c)
                }()
        }
        for i := range c {
                fmt.Println(i)
        }
}
```

So why isn’t there a version of `close` that lets you check if a channel is closed? Something like this:

GO

```go
if !isClosed(c) {
        // c isn't closed, send the value
        c <- v
}
```

But this function would have an inherent race. Someone may close the channel after we checked `isClosed(c)` but *before* the code gets to `c ← v`. This race is unavoidable in a parallel program.

One way to think about how this is possible is to imagine that goroutines work in different universes. They cannot observe each other unless they communicate. Because they cannot observe each other except for these communication points, time moves differently for each goroutine (given we cannot prove the opposite; time moves at the same rate for each goroutine, we must admit that it is *possible*) hence you cannot make statements like "a small amount of time" when talking about the interactions of different goroutines. There is no *happens before* relationship with goroutines unless they explicitly communicate.

This is not just a theoretical bun fight, it is easily demonstrable that the operating system thread backing any goroutine may be rescheduled at any time by the operating system. A different thread hosting a different goroutine can move ahead, relative to the sleeping thread, in time easily able to execute the channel close operation before the original thread is revived to attempt to send on the now closed channel.

If you need to ensure that only one goroutine closes a channel you must create a point of coordination *before* the `close` operation.

GO

```go
func main() {
    var c = make(chan int, 100)
    var mu sync.Mutex
    var closed bool
    for i := 0; i < 10; i++ {
        go func() {
            for j := 0; j < 10; j++ {
                c <- j
            }
            mu.Lock()
            if !closed {
                close(c)
                closed = true
            }
            mu.Unlock()
        }()
    }
    for i := range c {
        fmt.Println(i)
    }
}
```

TIP Solutions for dealing with this fan in problem are discussed in https://blog.golang.org/pipelines

A receive from a closed channel returns the zero value immediately

Once a channel has been closed, you cannot send a value on this channel, but you can still receive from the channel. Once a channel is closed *and* all values drained from its buffer, the channel will always return zero values immediately.

In this example we create a channel with a buffer of two, fill the buffer, then close it.

GO

```go
func main() {
    c := make(chan int, 3)
    c <- 1
    c <- 2
    c <- 3
    close(c)
    for i := 0; i < 4; i++ {
        fmt.Printf("%d ", <-c) // prints 1 2 3 0
    }
}
```

Running the program shows we retrieve the first three values we sent on the channel, then on our forth attempt the channel gives us the channel's zero value.

The correct solution to this problem is to use a `for range` loop.

GO

```go
for v := range c {
                // do something with v
}

for v, ok := <- c; ok ; v, ok = <- c {
                // do something with v
}
```

These two statements are equivalent in function, and demonstrate what for range is doing under the hood.

Being able to detect if your channel is closed is a useful property, it is used in the range over channel idiom to exit the loop once a channel has been drained.

GO

```go
func main() {
    ch := make(chan bool, 2)
    ch <- true
    ch <- true
    close(ch)

    for v := range ch {
        fmt.Println(v) // called twice
    }
}
```

but really comes into its own when combined with select. Let's start with this example

GO

```go
func main() {
    finish := make(chan bool)
    var done sync.WaitGroup
    done.Add(1)
    go func() {
        select {
        case <-time.After(1 * time.Hour):
        case <-finish:
        }
        done.Done()
    }()
    t0 := time.Now()
    finish <- true // send the close signal
    done.Wait()    // wait for the goroutine to stop
    fmt.Printf("Waited %v for goroutine to stop\n", time.Since(t0))
}
```

Running the program, on my system, gives a low wait duration, hence it is clear that the goroutine does not wait the full hour before calling `done.Done()`.

```
Waited 129.607us for goroutine to stop
```

But there are a few problems with this program. The first is the finish channel is not buffered, so the send to finish may block if the receiver forgot to add finish to their select statement. You could solve that problem by wrapping the send in a select block to make it non blocking, or making the finish channel buffered. However what if you had many goroutines listening on the finish channel, you would need to track this and remember to send the correct number of times to the finish channel. This might get tricky if you aren't in control of creating these goroutines; they may be being created in another part of your program, perhaps in response to incoming requests over the network.

A nice solution to this problem is to leverage the property that a closed channel is always ready to receive. Using this property we can rewrite the program, now including 100 goroutines, without having to keep track of the number of goroutines spawned, or correctly size the finish channel

GO

```go
package main

import (
	"fmt"
	"sync"
	"time"
)

func main() {
	const n = 100
	finish := make(chan bool)
	var done sync.WaitGroup
	for i := 0; i < n; i++ {
		done.Add(1)
		go func() {
			select {
			case <-time.After(1 * time.Hour):
			case <-finish:
			}
			done.Done()
		}()
	}
	t0 := time.Now()
	close(finish)    // closing finish makes it ready to receive
	done.Wait()      // wait for all goroutines to stop
	fmt.Printf("Waited %v for %d goroutines to stop\n", time.Since(t0), n)
}
```

On my system, this returns

```
Waited 231.385us for 100 goroutines to stop
```

So what is going on here? As soon as the finish channel is closed, it becomes ready to receive. As all the goroutines are waiting to receive either from their `time.After` channel, or finish, the select statement is now complete and the goroutines exits after calling `done.Done()`` to deincrement the WaitGroup counter. This powerful idiom allows you to use a channel to send a signal to an unknown number of goroutines, without having to know anything about them, or worrying about deadlock.

Before moving on to the next topic, I want to mention a final simplification that is preferred by many Go programmers. If you look at the sample program above, you'll note that we never send a value on the finish channel, and the receiver always discards any value received. Because of this it is quite common to see the program written like this:

GO

```go
package main

import (
        "fmt"
        "sync"
        "time"
)

func main() {
        finish := make(chan struct{})
        var done sync.WaitGroup
        done.Add(1)
        go func() {
                select {
                case <-time.After(1 * time.Hour):
                case <-finish:
                }
                done.Done()
        }()
        t0 := time.Now()
        close(finish)
        done.Wait()
        fmt.Printf("Waited %v for goroutine to stop\n", time.Since(t0))
}
```

As the behaviour of the close(finish) relies on signalling the close of the channel, not the value sent or received, declaring finish to be of type chan struct{} says that the channel contains no value; we’re only interested in its closed property.

GO

```go
func main() {
    c := make(chan int, 3)
    c <- 1
    c <- 2
    c <- 3
    close(c)
    for i := 0; i < 4; i++ {
        fmt.Printf("%d ", <-c) // prints 1 2 3 0
    }
}
```

NOTE

When consuming values from a channel until it closes, the better solution is to use a `for range` style loop.

GO

```go
for v := range c {
    // do something with v
}
```

Which is just syntactic sugar over the more verbose

GO

```go
for v, ok := <- c; ok ; v, ok = <- c {
        // do something with v
}
```

These two statements are equivalent in function, and demonstrate what for range is doing under the hood.

When sending or receiving on a channel, consider what happens if the other party never receives the message?

- What happens if sending this value blocks ?
- What happens if a value is never received from this channel ?

10.1.3. Prefer channels with a size of zero or one

When dealing with an unknown producer or consumer choose a buffer size of zero or one.

A buffer size of zero is ideal for coordination. A buffer size of one is idea to permit the sender to deposit the value without blocking and move on.

Usually to exit, the one buffer size is usually used with a single producer and a multiplexing consumer.

A buffer size greater than one is useful in the case where you know that exact number of values that will be deposited in the channel *before* it is drained. The common case is multiple workers operating in parallel, and a coordinator waiting on that result.

The most reasonable channels sizes are usually zero and one. Most other sizes are *guesses*. When you guess incorrectly, the program is unreliable.

10.1.4. Closed channels as a semaphore

Closing a channel is an operation taken by the sender to tell the receiver that there is no more data coming on the channel. Clearly sending a value on a channel that has been closed is an error. By analogy, closing a channel that has been closed—trying to send an additional piece of information on that channel, namely that there is no more data coming—is an error.

— Rob

(I think when I argued that the caller should be able to close a channel)

10.1.5. Channels vs Mutexes

Channels should be used to synchronise between goroutines by sending data, ie the receiver waits for the sender to reach a point in its execution where it generates a value. In the case of an unbuffered channel, the sender also waits for the receiver to reach a point where it can consume the value. Mutexes have a different role, they exist to prevent multiple goroutines concurrently mutating a value. Most of the time the goroutines do not want to synchronise, but merely avoid corrupting a shared value.

TIP Make channel send the last line of your function: ensure data safety

10.2. Goroutines

10.2.1. The ownership of a resource belongs to a goroutine

Each resource (what's a resource?) should have one and only one owner. The owner is responsible for freeing the resource. It may not have created it, it may not have been the last to use it, but that specific goroutine is responsible for freeing it.

This is one of the rare cases we're i'm breaking my prohibition on absolutist terms; a goroutine is responsible for a resource. If that is not true, then the resource cannot be safely freed. Perhaps this is ok in your design, but from experience, its not a pattern to reach for.

If you buy my line that a goroutine owns a resource then the sure fire way to free a resource is to stop that goroutine. Perhaps there are other actions which free the resource within the lifetime of the goroutine, but in the limit, if a goroutine owns a resource, and the goroutine has finished, one of two things are true

1. The resource has been free'd
2. The resource can never be free'd.

A channel is a resource, so's a goroutine.

10.2.2. Keep yourself busy or do the work yourself

What is the problem with this program?

```go
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, GopherCon SG")
    })
    go func() {
        if err := http.ListenAndServe(":8080", nil); err != nil {
            log.Fatal(err)
        }
    }()

    for {
    }
}
```

The program does what we intended, it serves a simple web server. However it also does something else at the same time, it wastes CPU in an infinite loop. This is because the `for{}` on the last line of `main` is going to block the main goroutine because it doesn't do any IO, wait on a lock, send or receive on a channel, or otherwise communicate with the scheduler.

As the Go runtime is mostly cooperatively scheduled, this program is going to spin fruitlessly on a single CPU, and may eventually end up live-locked.

How could we fix this? Here's one suggestion.

GO
```go
package main

import (
    "fmt"
    "log"
    "net/http"
    "runtime"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, GopherCon SG")
    })
    go func() {
        if err := http.ListenAndServe(":8080", nil); err != nil {
            log.Fatal(err)
        }
    }()

    for {
        runtime.Gosched()
    }
}
```

This might look silly, but it's a common common solution I see in the wild. It's symptomatic of not understanding the underlying problem.

Now, if you're a little more experienced with go, you might instead write something like this.

GO
```go
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, GopherCon SG")
    })
    go func() {
        if err := http.ListenAndServe(":8080", nil); err != nil {
            log.Fatal(err)
        }
    }()

    select {}
}
```

An empty select statement will block forever. This is a useful property because now we're not spinning a whole CPU just to call `runtime.GoSched()`. However, we're only treating the symptom, not the cause.

I want to present to you another solution, one which has hopefully already occurred to you. Rather than run `http.ListenAndServe` in a goroutine, leaving us with the problem of what to do with the main goroutine, simply run `http.ListenAndServe` on the main goroutine itself.

TIP

The act of sharing creates action.

A goroutine waiting on a select may as well not be there. It cannot be observed, except for very gross measures like a counter. The sender knows nothing about the state of the reciever and vice versa until they decide to communicate.

TIP

If the `main.main` function of a Go program returns then the Go program will unconditionally exit no matter what other goroutines started by the program over time are doing.

GO

```go
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, GopherCon SG")
    })
    if err := http.ListenAndServe(":8080", nil); err != nil {
        log.Fatal(err)
    }
}
```

So this is my first piece of advice: if your goroutine cannot make progress until it gets the result from another, oftentimes it is simpler to just do the work yourself rather than to delegate it.

This often eliminates a lot of state tracking and channel manipulation required to plumb a result back from a goroutine to its initiator.

TIP

Many Go programmers overuse goroutines, especially when they are starting out. As with all things in life, moderation is the key to success.

Don’t wait for something a goroutine can do itself. Don’t pass work to a goroutine and wait for it to finish; do the work yourself.

10.2.3. Be mindful of the lifetime of your goroutines

Most goroutine live for a short period of time, or effectively forever, that is, until the end of your program.

Goroutines and channels are not free. Their cost may be o(1), but it is not zero, and their use must be balanced against the o(1) cost

Goroutines hold state, references to data structure, sparce operating system resources like network sockets, open files, and possibly other resources like semaphores designed to limit parallel use of a shard resource, disk io or database connections are a good example. Lastly goroutines directly own an amount of stack memory, possibly a little, potentially a lot, depending on your program. It is these resources we are concerning ourselves with.

Cleaning up resource allocations is not difficult, Go's ubiquitous defer statement ties resource deallocation to the calling frame ; exit the function, cleanup automatically.

But the question remains, how do we ensure that the goroutine exits it's function, triggering defer ?

#golang top tip: if your package provides a New style constructor for types that start goroutines in the background, you need to provide some way to stop those goroutines.

never start a goroutine without knowing how it will finish (who's on the other end of that channel)

No facility for a goroutine join operation. Don't just signal goroutines to exit. Wait until they exit, even if the exit signal is executed with no latency - no lines between ← done; and return - but there may be defer statements that have to execute

A goroutine is responsible for operating on a resource.

Ideally it's creates them but it must be responsible for releasing them

This ties the lifetime of the resources to the lifetime of the goroutine

This i is especially important to control the lifetime of the goroutine so that it's associated resources may me freed

If you cannot identify the lifetime of a resource in terms of a goroutine, that is a design issue. This resource has a static lifetime; once its initialised it can never be safely free'd because its ownership is never clear.

It's not just about the memory they pin, to be useful a goroutine has to do something, and that means it almost always hold reference to, or ownership of a resource. A lock, a network connection, a buffer with data, the sending end of a channel. While that goroutine is alive, the lock is held, the network connection remains open, the buffer retained and the receivers of the channel will continue to wait for more data.

The simplest way to free those resources is to tye them to the lifetime of the goroutine. When the goroutine exits, you know the resource has been freed.

Cancellation is about goroutines because each resource has one owner, and that is a goroutines.

Cancellation is about channels, because those are the mechanisms for communicating between goroutines.

Perhaps fitting for the final topic in this presentation, we're going to talk about stopping.

A previous example showed using a goroutine when one wasn't really necessary. But one of the driving reasons for using Go is the first class concurrency features the language offers. Indeed there are many instances where you want to exploit the parallelism available in your hardware. To do so, you must use goroutines.

This simple application serves http traffic on two different ports, port 8080 for application traffic and port 8001 for access to the `/debug/pprof` endpoint.

GO
```go
package main

import (
    "fmt"
    "net/http"
    _ "net/http/pprof"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(resp http.ResponseWriter, req *http.Request) {
        fmt.Fprintln(resp, "Hello, QCon!")
    })
    go http.ListenAndServe("127.0.0.1:8001", http.DefaultServeMux) // debug
    http.ListenAndServe("0.0.0.0:8080", mux)                       // app traffic
}
```

Although this program isn't very complicated, it represents the basis of a real application.

There are a few problems with the application as it stands which will reveal themselves as the application grows, so lets address a few of them now.

GO
```go
func serveApp() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(resp http.ResponseWriter, req *http.Request) {
        fmt.Fprintln(resp, "Hello, QCon!")
    })
    http.ListenAndServe("0.0.0.0:8080", mux)
}

func serveDebug() {
    http.ListenAndServe("127.0.0.1:8001", http.DefaultServeMux)
}

func main() {
    go serveDebug()
    serveApp()
}
```

By breaking the `serveApp` and `serveDebug` handlers out into their own functions we've decoupled them from `main.main`. We've also followed the advice from above and make sure that `serveApp` and `serveDebug` leave their concurrency to the caller.

But there are some operability problems with this program. If `serveApp` returns then `main.main` will return causing the program to shutdown and be restarted by whatever process manager you're using.

TIP

Just as functions in Go leave concurrency to the caller, applications should leave the job of monitoring their status and restarting them if they fail to the program that invoked them. Do not make your applications responsible for restarting themselves, this is a procedure best handled from outside the application.

However, `serveDebug` is run in a separate goroutine and if it returns just that goroutine will exit while the rest of the program continues on. Your operations staff will not be happy to find that they cannot get the statistics out of your application when they want too because the `/debug` handler stopped working a long time ago.

What we want to ensure is that if *any* of the goroutines responsible for serving this application stop, we shut down the application.

```go
func serveApp() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(resp http.ResponseWriter, req *http.Request) {
        fmt.Fprintln(resp, "Hello, QCon!")
    })
    if err := http.ListenAndServe("0.0.0.0:8080", mux); err != nil {
        log.Fatal(err)
    }
}

func serveDebug() {
    if err := http.ListenAndServe("127.0.0.1:8001", http.DefaultServeMux); err != nil {
        log.Fatal(err)
    }
}

func main() {
    go serveDebug()
    go serveApp()
    select {}
}
```

Now `serverApp` and `serveDebug` check the error returned from `ListenAndServe` and call `log.Fatal` if required. Because both handlers are running in goroutines, we park the main goroutine in a `select{}`.

This approach has a number of problems:

1. If `ListenAndServer` returns with a `nil` error, `log.Fatal` won't be called and the HTTP service on that port will shut down without stopping the application.
2. `log.Fatal` calls `os.Exit` which will unconditionally exit the program; defers won't be called, other goroutines won't be notified to shut down, the program will just stop. This makes it difficult to write tests for those functions.

TIP Only use `log.Fatal` from `main.main` or `init` functions.

What we'd really like is to pass any error that occurs back to the originator of the goroutine so that it can know *why* the goroutine stopped, can shut down the process cleanly.

GO

```go
func serveApp() error {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(resp http.ResponseWriter, req *http.Request) {
        fmt.Fprintln(resp, "Hello, QCon!")
    })
    return http.ListenAndServe("0.0.0.0:8080", mux)
}

func serveDebug() error {
    return http.ListenAndServe("127.0.0.1:8001", http.DefaultServeMux)
}

func main() {
    done := make(chan error, 2)
    go func() {
        done <- serveDebug()
    }()
    go func() {
        done <- serveApp()
    }()

    for i := 0; i < cap(done); i++ {
        if err := <-done; err != nil {
            fmt.Println("error: %v", err)
        }
    }
}
```

We can use a channel to collect the return status of the goroutine. The size of the channel is equal to the number of goroutines we want to manage so that sending to the `done` channel will not block, as this will block the shutdown the of goroutine, causing it to leak.

As there is no way to safely close the `done` channel we cannot use the `for range` idiom to loop of the channel until all goroutines have reported in, instead we loop for as many goroutines we started, which is equal to the capacity of the channel.

Now we have a way to wait for each goroutine to exit cleanly and log any error they encounter. All that is needed is a way to forward the shutdown signal from the first goroutine that exits to the others.

It turns out that asking a `http.Server` to shut down is a little involved, so I've spun that logic out into a helper function. The `serve` helper takes an address and `http.Handler`, similar to `http.ListenAndServe`, and also a `stop` channel which we use to trigger the `Shutdown` method.

```go
func serve(addr string, handler http.Handler, stop <-chan struct{}) error {
    s := http.Server{
        Addr:    addr,
        Handler: handler,
    }

    go func() {
        <-stop // wait for stop signal
        s.Shutdown(context.Background())
    }()

    return s.ListenAndServe()
}

func serveApp(stop <-chan struct{}) error {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(resp http.ResponseWriter, req *http.Request) {
        fmt.Fprintln(resp, "Hello, QCon!")
    })
    return serve("0.0.0.0:8080", mux, stop)
}

func serveDebug(stop <-chan struct{}) error {
    return serve("127.0.0.1:8001", http.DefaultServeMux, stop)
}

func main() {
    done := make(chan error, 2)
    stop := make(chan struct{})
    go func() {
        done <- serveDebug(stop)
    }()
    go func() {
        done <- serveApp(stop)
    }()

    var stopped bool
    for i := 0; i < cap(done); i++ {
        if err := <-done; err != nil {
            fmt.Println("error: %v", err)
        }
        if !stopped {
            stopped = true
            close(stop)
        }
    }
}
```

Now, each time we receive a value on the `done` channel, we close the `stop` channel which causes all the goroutines waiting on that channel to shut down their `http.Server`. This in turn will cause all the remaining `ListenAndServe` goroutines to return. Once all the goroutines we started have stopped, `main.main` returns and the process stops cleanly.

TIP Writing this logic yourself is repetitive and subtle. Consider something like this package, `https://github.com/heptio/workgroup` which will do most of the work for you.

10.2.4. Goroutines operate asynchronously

Not only do goroutines operate at their own rate and scheduled at unknown points. But also time passes independently for each goroutine, the contract for `time.Sleep` does not guarantee a maximum sleep duration, only a minimum.

TIP

Avoid `runtime.SetFinaliser`. Finalisers are not guarenteed to run, a design which free's resources based on the operation of the garbage collector is inherently unpredictable.

10.3. Leave concurrency to the caller

What is the difference between these two APIs?

GO

```go
// ListDirectory returns the contents of dir.
func ListDirectory(dir string) ([]string, error)

// ListDirectory returns a channel over which
// directory entries will be published. When the list
// of entries is exhausted, the channel will be closed.
func ListDirectory(dir string) chan string
```

The obvious differences are the first example reads a directory into a slice then returns the whole slice, or an error if something went wrong. This happens synchronously, the caller of `ListDirectory` blocks until all directory entries have been read. Depending on how large the directory, this could take a long time, and could potentially allocate a lot of memory building up the slide of directory entry names.

Lets look at the second example. This is a little more Go like, `ListDirectory` returns a channel over which directory entries will be passed. When the channel is closed, that is your indication that there are no more directory entries. As the population of the channel happens *after* `ListDirectory` returns, `ListDirectory` is probably starting a goroutine to populate the channel.

NOTE

It's not necessary for the second version to actually use a Go routine; it could allocate a channel sufficient to hold all the directory entries without blocking, fill the channel, close it, then return the channel to the caller. But this is unlikely, as this would have the same problems with consuming a large amount of memory to buffer all the results in a channel.

The channel version of `ListDirectory` has two further problems:

- By using a closed channel as the signal that there are no more items to process there is no way for `ListDirectory` to tell the caller that the set of items returned over the channel is incomplete because an error was encountered partway through. There is no way for the caller to tell the difference between an *empty directory* and an *error* to read from the directory entirely. Both result in a channel returned from `ListDirectory` which appears to be closed immediately.
- The caller *must* continue to read from the channel until it is closed because that is the only way the caller can know that the goroutine which was started to fill the channel has stopped. This is a serious limitation on the use of `ListDirectory`, the caller has to spend time reading from the channel even though it may have received the answer it wanted. It is probably more efficient in terms of memory usage for medium to large directories, but this method is no faster than the original slice based method.

The solution to the problems of both implementations is to use a callback, a function that is called in the context of each directory entry as it is executed.

GO

```go
func ListDirectory(dir string, fn func(string))
```

Not surprisingly this is how the `filepath.WalkDir` function works.

TIP

If your function starts a goroutine you must provide the caller with a way to explicitly stop that goroutine. It is often easier to leave decision to execute a function asynchronously to the caller of that function.

TIP

Always block

The api of your code should present a synchronous world view. If the implementation leverages concurrency in the implementation, idealy that should not be observable to the caller. If your code takes a callback, make sure it's called synchronously. If not, you must manage the memory fence.

If your code is running concurrently it should be very explicit about this fact. Concurrency is not something which can be managed *transparently*.

Let the caller choose a paralise the execution of your library or function. If your program uses concurrency it should do so transparently, don't leak goroutines.

---

1. https://talks.golang.org/2014/names.slide#4
2. https://www.infoq.com/articles/API-Design-Joshua-Bloch
1. https://www.lysator.liu.se/c/pikestyle.html
2. This practice can be traced back to Fortran where variables beginning with the letter `I` through `N` (the first two letters in the word *integer*) were automatically declared to be integers. A variable beginning with some other letter was a real (floating point).
3. https://speakerdeck.com/campoy/understanding-nil
4. Colloqually repetition is known as *stuttering*. I appreciate the sentiment this implies, but feel it's unkind to those who suffer from this speach impediment (myself included), so I prefer instead to say *repetition*.
5. https://www.youtube.com/watch?v=Ic2y6w8lMPA
6. https://medium.com/@matryer/line-of-sight-in-code-186dd7cdea88
7. https://dave.cheney.net/2014/10/17/functional-options-for-friendly-apis
8. https://commandcenter.blogspot.com/2014/01/self-referential-functions-and-design.html
9. https://golang.org/doc/go1.4#internalpackages
10. http://www.hyrumslaw.com/
11. https://www.amazon.com/Philosophy-Software-Design-John-Ousterhout/dp/1732102201
12. https://blog.golang.org/errors-are-values

Version 1b5883b
Last updated 2020-01-27 10:04:01 +0200