Practical Go: Real world advice for writing maintainable Go programs

Dave Cheney – dave@cheney.net – Version 3d3ca1, 2018-10-20

Table of Contents

Introduction

Hello,

My goal over the next two sessions is to give you my advice for best practices writing Go code.

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

TIP

You can find the latest version of this presentation at

https://dave.cheney.net/practical-go/presentations/qcon-china.html

1. Guiding principles

If I'm going to talk about best practices in any programming language I need some way to define what I mean by *best*. If you came to my keynote yesterday you would have seen this quote from the Go team lead, Russ Cox:

> *Software engineering is what happens to programming when you add time and other programmers.*
>
> — *Russ Cox*

Russ is making the distinction between software *programming* and software *engineering*. The former is a program you write for yourself. The latter is a product that many people will work on over time. Engineers will come and go, teams will grow and shrink over time, requirements will change, features will be added and bugs fixed. This is the nature of software engineering.

I'm possibly one of the earliest users of Go in this room, but to argue that my seniority gives my views more weight is *false*. Instead, the advice I'm going to present today is informed by what I believe to be the guiding principles underlying Go itself. They are:

1. Simplicity
2. Readability
3. Productivity

NOTE

You'll note that I didn't say *performance*, or *concurrency*. There are languages which are a bit faster than Go, but they're certainly not as simple as Go. There are languages which make concurrency their highest goal, but they are not as readable, nor as productive.

Performance and concurrency are important attributes, but not as important as *simplicity*, *readability*, and *productivity*.

1.1. Simplicity

Why should we strive for simplicity? Why is important that Go programs be simple?

We've all been in a situation where you say "I can't understand this code", yes? We've all worked on programs where you're scared to make a change because you're worried it'll break another part of the program; a part you don't understand and don't know how to fix.

This is complexity. Complexity turns reliable software in unreliable software. Complexity is what kills software projects.

Simplicity is the highest goal of Go. Whatever programs we write, we should be able to agree that they are simple.

1.2. Readability

> *Readability is essential for maintainability.*
>
> — *Mark Reinhold*
> *JVM language summit 2018*

Why is it important that Go code be readable? Why should we strive for readability?

> *Programs must be written for people to read, and only incidentally for machines to execute.*
>
> — *Hal Abelson and Gerald Sussman*
> *Structure and Interpretation of Computer Programs*

Readability is important because all software, not just Go programs, is written by humans to be read by other humans. The fact that software is also consumed by machines is secondary.

Code is read many more times than it is written. A single piece of code will, over its lifetime, be read hundreds, maybe thousands of times.

> *The most important skill for a programmer is the ability to effectively communicate ideas.*
>
> — *Gastón Jorquera* [1]

Readability is key to being able to understand what the program is doing. If you can't understand what a program is doing, how can you hope to maintain it? If software cannot be maintained, then it will be rewritten; and that could be the last time your company will invest in Go.

If you're writing a program for yourself, maybe it only has to run once, or you're the only person who'll ever see it, then do what ever works for you. But if this is a piece of software that more than one person will contribute to, or that will be used by people over a long enough time that requirements, features, or the environment it runs in changes, then your goal must be for your program to be *maintainable*.

The first step towards writing maintainable code is making sure the code is readable.

1.3. Productivity

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

The last underlying principle I want to highlight is productivity. Developer productivity is a sprawling topic but it boils down to this; how much time do you spend doing useful work verses waiting for your tools or hopelessly lost in a foreign code-base. Go programmers should feel that they can get a lot done with Go.

The joke goes that Go was designed while waiting for a C++ program to compile. Fast compilation is a key feature of Go and a key recruiting tool to attract new developers. While compilation speed remains a constant battleground, it is fair to say that compilations which take minutes in other languages, take seconds in Go. This helps Go developers feel as productive as their counterparts working in dynamic languages without the reliability issues inherent in those languages.

More fundamental to the question of developer productivity, Go programmers realise that code is written to be read and so place the act of reading code above the act of writing it. Go goes so far as to enforce, via tooling and custom, that all code be formatted in a specific style. This removes the friction of learning a project specific dialect and helps spot mistakes because they just *look* incorrect.

Go programmers don't spend days debugging inscrutable compile errors. They don't waste days with complicated build scripts or deploying code to production. And most importantly they don't spend their time trying to understand what their coworker wrote.

Productivity is what the Go team talk about when they say the language must *scale*.

2. Identifiers

The first topic we're going to discuss is *identifiers*. 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*

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 the defining quality of good code thus choosing good names is crucial to the readability of Go code.

2.1. Choose identifiers for clarity, not brevity

> *Obvious code is important. What you can do in one line you should do in three.*
>
> — *Ukiah Smith*

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.

> *Good naming is like a good joke. If you have to explain it, it's not funny.*
>
> — *Dave Cheney*

Key to this clarity is the names we choose for identifies in Go programs. Let's talk about the qualities of a good name:

- **A good name is concise.** A good name need not be the shortest it can possibly be, 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 operation. A good name should describe the purpose of a package, *not* its contents. The more accurately a name describes the thing it identifies, the better the name.
- **A good name is should be predictable.** You should be able to infer the way a name 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*.

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

2.2. Identifier length

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

Andrew Gerrand suggests that by using longer identifies for some things we indicate to the reader that they are of higher importance.

> *The greater the distance between a name's declaration and its uses, the longer the name should be.*
>
> — *Andrew Gerrand* [2]

From this we can draw some guidelines:

- Short variable names work well when the distance between their declaration and *last* use is short.
- 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.
- Don't include the name of your type in the name of your variable.
- Constants should describe the value they hold, *not* how that value is used.
- Single letter variables for loops and branches, single words for parameters and return values, multiple words for functions and package level declarations
- Single words for methods, interfaces, and packages.
- 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.

Let's look at an example to

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 on the following line. `p` lives for a very short time both on the page, and during the execution of the function. A reader who is interested in the effect values of `p` have on the program need only read two lines.

By comparison `people` is declared in the function parameters and lives for 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. 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.

2.2.1. Context is key

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

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

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

fundamentally more readable than

```
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.

However, which of these functions is more readable?

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

or

```
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 as in SNMP messages a sub value of each OID is called an Index.

> **TIP**
>
> Don't mix and match long and short formal parameters in the same declaration.

2.3. Don't name your variables for their types

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

The name of the variable should describe its contents, not the *type* of the contents. Consider this example:

```
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 scalar variable is required, so the `Map` suffix is redundant.

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

```
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, and 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 the code is 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.

My suggestion is to avoid any suffix that resembles the type of the variable.

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

This advice also applies to function parameters. For example:

```
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` will do 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.

NOTE

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.

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

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

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

2.4. Use a consistent naming style

Another property of a good name is it should be predictable. The reader should be able to understand 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.

For example, if your code passes around a database handle, make sure each time the parameter appears, it has the same name. Rather than a combination of `d *sql.DB`, `dbase *sql.DB`, `DB *sql.DB`, and `database *sql.DB`, instead consolidate on something like;

```
db *sql.DB
```

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.

Similarly for 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 in this type.

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`.

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.

Finally, certain single letter variables have traditionally been associated with loops and counting. For example, `i`, `j`, and `k` are commonly the loop induction variable for simple `for` loops. `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, and `s` is often used as shorthand for parameters of type `string`.

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.

> **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 units.

2.5. Use a consistent declaration style

Go has at least six different ways to declare a variable

- `var x int = 1`
- `var x = 1`
- `var x int; x = 1`
- `var x = int(1)`
- `x := 1`

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. With all these different ways of declaring a variable, how do we avoid each Go programmer choosing their own style?

I want to present a suggestions for how I declare variables in my programs. 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 in the function, use the `var` keyword.

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

- **When declaring *and* initialising, use `:=`.** When declaring and initialising the variable at the same time, 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. This makes it clear to the reader that the variable on the left hand side of the `:=` is being deliberately initialised.

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

```
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:

```
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 its better to make it clear that we're going to use the zero value by instead writing

```
var players int
```

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

```
var things = nil
```

Because `nil` does not have a type. [2] Instead we have a choice, do we want the zero value for a slice?

```
var things []Thing
```

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

```
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:

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

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

This brings us to the third declaration,

```
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

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

Which makes it clear that `thing` is explicitly initialised to the result of `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,

```
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 a `Thing{}`, which 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`

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

NOTE

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

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

Would be odd. The declaration may be more readable like this

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

In summary:

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

TIP

Make tricky declarations obvious.
When something is complicated, it should *look* complicated.

```
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 chosen to be `uint32` than the short declaration form:

```
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.

2.6. Be a team player

I talked about a goal of software engineering to produce readable, 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 to follow the local style.

Changing styles in the middle of a file is jarring. Uniformity, even if its not your preferred approach, is more valuable for maintenance than your personal preference. My rule of thumb is; if it fits through `gofmt` then its usually not worth holding up a code review for.

> **TIP** If you want to do a renaming across a code-base, 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.

3. Comments

Before we move on to larger items I want to spend a few minutes 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. A comments should do 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 why it is.

The first form is ideal for commentary on public symbols:

```
// 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:

```
// 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 comment exists to provide that context.

```
return &v2.Cluster_CommonLbConfig{
    // Disable HealthyPanicThreshold
        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.

3.1. Comments on variables and constants should describe their contents not their purpose

I talked 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 variables *contents*, not the variables *purpose*.

```
const randomNumber = 6 // determined from 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. Here are some more examples:

```
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 the context of HTTP the number `100` is known as `StatusContinue`, as defined in RFC 7231, section 6.2.1.

TIP

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

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

Here the comment lets the reader know that the `dowidth` function is responsible for maintaining the state of `sizeCalculationDisabled`.

TIP

Hiding in plain sight

This is a tip from Kate Gregory. [3] Sometimes you'll find a better name for a variable hiding in a comment.

```
// 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.

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

Now the comment is redundant and can be removed.

3.2. 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

```
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:

```
// 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.

Here is an example from the `io` package

```
// 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 is going to be hard to understand.

3.2.1. 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.

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

The username is not a promise that that person has committed to fixing the issue, but they may be the best person to ask when the time comes to address it. Other projects annotate TODOs with a date or an issue number.

3.2.2. 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 be easier to comprehend, smaller functions are easier to test in isolation, and now you've isolated the orthogonal code into its own function, its name may be all the documentation required.

4. Package Design

> *Write shy code - modules that don't reveal anything unnecessary to other modules and that don't rely on other modules' implementations.*
>
> *— Dave Thomas*

Each Go package is in effect it's own small Go program. Just as the implementation of a function or method is unimportant to the caller, the implementation of the functions and methods and types that make your package's public API—its behaviour—is unimportant for 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 including the package's name, naming types, and tips for writing methods and functions.

4.1. A good package starts with its name

Writing a good Go package starts with the package's name. Think of your package's name as an elevator pitch to describe what it does using just one word.

Just as I talked about names for variables in the previous section, the name of a package is very important. The rule of thumb I follow is not, "what types should I put in this package?". Instead the question I ask "what does service does package provide?" Normally the answer to that question is not "this package provides the X type", but "this package let's you speak HTTP".

TIP	Name your package after what is *provides*, not what it *contains*.

4.1.1. Good package names should be unique.

Within your project, each package name should be unique. This advice is pretty easy to follow if the 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.

b. The package overlaps another package of a similar name. In this case either you should review your design, or consider merging the packages.

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

A common cause of poor package names is what call *utility packages*. These are packages where common 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. I believe the solution to this is to reduce the number of packages, to combine the client, server, and common code into a single package named after the function of the package.

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

An identifier's name 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.

4.3. Return early rather than nesting deeply

As Go does not use exceptions for control flow there is no requirement to deeply indent your code just to provide a top level structure for the `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. My friend Mat Ryer calls this practice 'line of sight' coding. [4]

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

```
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
}
```
GO

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,

```
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
    }
    return errors.New("bytes.Buffer: UnreadRune: previous operation was not a successful ReadRune")
}
```
GO

The body of the successful case, the most common, 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 called 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 prefer to use guard clauses and returning early on errors.

4.4. Make the zero value useful

Every variable declaration, assuming no explicit initialiser is provided, will be automatically initialised to a value that matches the contents of zeroed memory. This is the values *zero value*. The type of the value determines its zero value; for numeric types it is zero, 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 mutex'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 coded 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 declare 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 definition of a slice header.

```
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 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. The following code will output false.

GO

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

A surprising, but useful, 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())
}
```

4.5. Avoid package level state

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 package that does not directly depend on the first.

There are two excellent ways to achieve loose coupling in Go

1. Use interfaces to describe the behaviour your functions or methods require.
2. Avoid the use of global state.

In Go we can declare variables at the function or method scope, and also at the package scope. When the variable is public, given a identifier starting 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.

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.

5. Project Structure

Let's talk about combining packages together into a project. Commonly this will be a single git repository, but in the future Go developers will use *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 package, 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.

If your project is an application, like your web application, Kubernetes controller, and so on, then you might have one or more `main` packages inside your project. For example, the Kubernetes controller I work on has a single `cmd/contour` package which serves as both the server deployed to a Kubernetes cluster, and a client for debugging purposes.

5.1. 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.

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

In Go we have only two access modifiers, public and private, 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?

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

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

Coming from Java?

If you're coming from a Java or C# background, consider this rule of thumb. - A Java package is equivalent to a single `.go` source file. - A Go package is equivalent to a whole Maven module or .NET assembly.

5.1.1. Arrange code into 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 consider consolidating `.go` file?

Here are the rules of thumb I use:

- Start each package with one `.go` file. Give that file the same name as the name of the folder. eg. `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
- 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.

TIP

Prefer nouns for source file names.

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 does not affect compilation time.

5.1.2. Prefer internal tests to external tests

The `go` tool supports writing your `testing` package tests in two places. Assuming your package is called `http2`, you can write a `http2_test.go` file and use the `package http2` declaration. Doing so will compile the code in `http2_test.go` *as if* it were part of the `http2` package. This is known colloquially as an *internal* test.

The `go` tool also supports a special package declaration, ending in `*test*`, *ie.,* `package http_test`. *This allows your test files to live alongside your code in the same package, however when those tests are compiled they are not part of your package's code, they live in their own package. This allows you to write your tests as if you were another package calling into your code. This is known as an _external* test.

I recommend using internal tests when writing unit tests for your package. This allows you to test each function or method directly, avoiding the bureaucracy of external testing.

However, you *should* place your `Example` test functions in an external test file. This ensures that when viewed in godoc, the examples have the appropriate package prefix and can be easily copy pasted.

TIP

Avoid elaborate package hierarchies, resist the desire to apply taxonomy

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

If you find you have created intermediate directories in your project which contain no `.go` files, you may have failed to follow this advice.

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

If your project contains multiple packages you may 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 place code which is public to your project, but private to other projects.

To create such a package, place it in 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 package doing the import 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. [5]

5.2. Keep package main small as small as possible

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, *including tests*.

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`, thus you should aim to move as much of your business logic out of your main function and ideally out of your main package.

TIP

main should parse flags, open connections to databases, loggers, and such, then hand off execution to a high level object.

6. API Design

The last piece of design advice I'm going to give today I feel is the most important.

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

6.1. Design APIs that are hard to misuse.

> *APIs should be easy to use and hard to misuse.*

— Josh Bloch [3]

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

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

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:

```
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.

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

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

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

```
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.

One possible solution to this 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—this can be asserted with a unit test—and 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.

6.2. Design APIs for their default use case

A few years ago I gave a talk [6] about using functional options [7] to make APIs easier to use for their default case.

The gist of this talk was you should design your APIs for the common use case. Sad another way, your API *should not* require the caller to provide parameters which they don't care about.

6.2.1. 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

```
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.

```
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

```
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,

```
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`.

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

verses

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

and a was this confusion really worth saving one line?

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

> **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 who only differ in test scope. Instead, use Public wrappers to hide those parameters, use test scoped helpers to set the property in test scope.

6.2.2. Prefer var args to []T parameters

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

```
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:

```
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:

```
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

```
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.

6.3. Let functions define the behaviour they requires

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

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

I could specify this function, 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 network storage is likely to become requirement later, 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. So, to verify its operation, the test would have to read the contents of the file after being written.

And I would have to ensure that `f` was written to a temporary location and always removed afterwards.

`*os.File` also 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 the `Save` function could describe only the parts of `*os.File` that were relevant.

What can we do ?

```
// Save writes the contents of doc to the supplied
// ReadWriterCloser.
func Save(rwc io.ReadWriteCloser, doc *Document) 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.

And 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.

```
// Save writes the contents of doc to the supplied
// WriteCloser.
func Save(wc io.WriteCloser, doc *Document) error
```

So 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.

This presents a problem for the caller of `Save` as it may want to write additional data to the stream after the document is written.

```
// Save writes the contents of doc to the supplied
// Writer.
func Save(w io.Writer, doc *Document) 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.

7. Error handling

I've given several presentations about error handling [8] and written a lot about error handling on my blog. I also spoke a lot about error handling in yesterday's session so I won't repeat what I've said.

- https://dave.cheney.net/2014/12/24/inspecting-errors
- https://dave.cheney.net/2016/04/07/constant-errors

Instead I want to cover two other areas related to error handling.

7.1. Eliminate error handling by eliminating errors

If you were in my presentation yesterday I talked about the draft proposals for improving error handling. But do you know what is better than an improved syntax for handling errors? Not needing to handle errors at all.

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

This section draws inspiration from John Ousterhout's recently book, A philosophy of Software Design [9]. One of the chapters in that book is called "Define Errors Out of Existence". We're going to try to apply this advice to Go.

7.1.1. 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 *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
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.

7.1.2. WriteResponse

My second example is inspired from the *Errors are values* blog post [10].

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

```
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.

7.2. Only handle an error once

Lastly, I want to mention that you should only handle errors once. Handling an error means inspecting the error value, and making a *single* decision.

```
// 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

```
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
```
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.

```
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
```
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 1/2 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.

7.2.1. Adding context to errors

The bug occurred because the author was trying to add *context* to the error message. They were trying to leave themselves a breadcrumb to point them back to the source of the error.

Let's look at another way to do the same thing using `fmt.Errorf`.

GO

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

func WriteAll(w io.Writer, buf []byte) error {
    _, err := w.Write(buf)
    if err != nil {
        return fmt.Errorf("write failed: %v", err)
    }
    return nil
}
```

By combining the annotation of the error with returning onto one line there it is harder to forget to return an error and avoid continuing accidentally.

If an I/O error occurs writing the file, the `error's `Error()` method will report something like this;

```
could not write config: write failed: input/output error
```

7.2.2. Wrapping errors with github.com/pkg/errors

The `fmt.Errorf` pattern works well for annotating the error *message*, but it does so at the cost of obscuring the *type* of the original error. I've argued that treating errors as opaque values is important to producing software which is *loosely coupled*, so the face that the type of the original error should not matter if the only thing you do with an error value is

1. Check that it is not `nil`.
2. Print or log it.

However there are some cases, I believe they are infrequent, where you do need to recover the original error. In that case you can use something like my errors package to annotate errors like this

GO

```
func ReadFile(path string) ([]byte, error) {
    f, err := os.Open(path)
    if err != nil {
        return nil, errors.Wrap(err, "open failed")
    }
    defer f.Close()

    buf, err := ioutil.ReadAll(f)
    if err != nil {
        return nil, errors.Wrap(err, "read failed")
    }
    return buf, nil
}

func ReadConfig() ([]byte, error) {
    home := os.Getenv("HOME")
    config, err := ReadFile(filepath.Join(home, ".settings.xml"))
    return config, errors.WithMessage(err, "could not read config")
}

func main() {
    _, err := ReadConfig()
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
}
```

Now the error reported will be the nice K&D [11] style error,

```
could not read config: open failed: open /Users/dfc/.settings.xml: no such file or directory
```

and the error value retains a reference to the original cause.

GO

```
func main() {
    _, err := ReadConfig()
    if err != nil {
        fmt.Printf("original error: %T %v\n", errors.Cause(err), errors.Cause(err))
        fmt.Printf("stack trace:\n%+v\n", err)
        os.Exit(1)
    }
}
```

Thus you can recover the original error and print a stack trace;

```
original error: *os.PathError open /Users/dfc/.settings.xml: no such file or directory
stack trace:
open /Users/dfc/.settings.xml: no such file or directory
open failed
main.ReadFile
        /Users/dfc/devel/practical-go/src/errors/readfile2.go:16
main.ReadConfig
        /Users/dfc/devel/practical-go/src/errors/readfile2.go:29
main.main
        /Users/dfc/devel/practical-go/src/errors/readfile2.go:35
runtime.main
        /Users/dfc/go/src/runtime/proc.go:201
runtime.goexit
        /Users/dfc/go/src/runtime/asm_amd64.s:1333
could not read config
```

Using the `errors` package gives you the ability to add context to error values, in a way that is inspectable by both a human and a machine. If you came to my presentation yesterday you'll know that wrapping is moving into the standard library in an upcoming Go release.

8. 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 performent 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 on 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 and we wouldn't look back on our formative efforts with regret.

This section discusses some pitfalls of naive usage of Go's concurrency features.

8.1. 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

```
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

```
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** 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

```
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 the key to success.

8.2. Leave concurrency to the caller

What is the difference between these two APIs?

```
// 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
```

Firstly, the obvious differences; 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

Its 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.

```
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.

8.3. Never start a goroutine without when it will stop.

The 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

```
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

```
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

```
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.

1. https://gaston.life/books/effective-programming/
2. https://talks.golang.org/2014/names.slide#4
3. https://www.infoq.com/articles/API-Design-Joshua-Bloch

1. https://www.lysator.liu.se/c/pikestyle.html
2. https://speakerdeck.com/campoy/understanding-nil
3. https://www.youtube.com/watch?v=Ic2y6w8lMPA
4. https://medium.com/@matryer/line-of-sight-in-code-186dd7cdea88
5. https://golang.org/doc/go1.4#internalpackages
6. https://dave.cheney.net/2014/10/17/functional-options-for-friendly-apis
7. https://commandcenter.blogspot.com/2014/01/self-referential-functions-and-design.html
8. https://dave.cheney.net/2016/04/27/dont-just-check-errors-handle-them-gracefully
9. https://www.amazon.com/Philosophy-Software-Design-John-Ousterhout/dp/1732102201
10. https://blog.golang.org/errors-are-values
11. http://www.gopl.io/

Version 3d3ca1
Last updated 2018-10-20 19:30:54 +1100