Summary

1. What is golang?

The goals of the language and its accompanyng tools were to be expressive, efficient in both compilation and execution, and effective in writing reliable and robust programs.

It borrows and adapts good ideas from many other languages, while avoiding features that gave led to complexity and unreliable code. It's facilities for concurrency are new and efficient, and its approach to data abstraction and object-oriented programming is unusually flexible. It has automatic memory management or garbage collection.

1.1. Go project - the motivation

The go project was borne of frustration with several software system at Google that were suffering from an explosion of complexity. As a recent high-level language, Go has the benefit of hindsight, and the basics are done well: it has garbage collection, a package system, first class functions, lexical scope, a system call interface, and immutable strings in which text is generally encoded in UTF-8. But it has comparatively few features and is unlikely to add more.

2. Hello world in Go

After download the language, you need to enable dependency tracking for your code by creating a `go.mod` file, to achieve that run `go mod init` giving it the name of the module your code will be in. The name is the module's module path.

2.1. Code example - tutor

```go
// hello-world.go

package main
import "fmt"

func main(){
  fmt.Println("Hello world")
}
```

- Declare a main package (a package is a way to group functions, and it's made up of all the files in the same directory).
- Import the popular fmt package, which contains functions for formatting text, including printing to the console. This package is one of the standard library packages you got when you installed Go.
- Implement a main function to print a message to the console. A main function executes by default when you run the main package.
- Just run `go run hello-world.go` on your terminal

2.2. How to import external library?

```go
package main
import "fmt"

import "rsc.io/quote"

func main(){
  fmt.Println(quote.Go())
}
```

- After you add your new **package**, you need to run `go mod tidy`, this will download the download module as a requirement, as well as a go.sum file for use in authenticating the module.

```
$ go mod tidy
go: finding module for package rsc.io/quote
go: found rsc.io/quote in rsc.io/quote v1.5.2
```

3. Create a Go Module

Go code is grouped into packages, and packages are grouped into modules. Your module specifies dependencies needed to run your code, including the Go version and the set of other modules it requires.

Go code is grouped into packages, and packages are grouped into modules. Your module specifies dependencies needed to run your code, including the Go ersion and the set of other modules it requires.

```go
package greetings

import "fmt"

func Hello(name string) string {
  message := fmt.Sprintf("Hi, %v. Welcome!", name)
  return message
}
```

- In Go, a function whose name starts with a capital letter can be called by a function not in the same package. **This is known in Go as an exported name**.

3.1. Importing this module

After create this folder, like `example/greetings` we'll create a `example/greetings-caller`.

1. `mkdir greetings-caller`
2. `go mod init example/greetings-caller`, after that, create `greetings-caller/main.go`:

```go
package main

import (
  "example/greetings"
  "fmt"
)

func main() {
  message := greetings.Hello("Guilherme")

  fmt.Println(message)
}
```

If you try to run this, you got an error.

Before run `go mod tidy` to add our package, we need to edit the `go.mod` file, because we dont publish our package yet.

```
❯ go mod edit -replace example/greetings=../
greetings
```

- Now our `example/greetings-caller/go.mod` should be like this:

```
replace example/greetings => ../greetings
```

3.2. Return and handle an error

```go
// greetings.go
package greetings

import (
  "fmt"
  "errors"
)

func Hello(name string) (string, error) {
  if name == "" {
    return "", errors.New("empty name")
  }

  message := fmt.Sprintf(
    "Hi, %v. Welcome!", name
  )
  return message, nil
}
```

```go
// main.go
package main

import(
  "fmt"
  "log"
  "example/greetings"
)

func main(){
  message, err := greetings.Hello("")

  if err != nil {
    log.Fatal(err)
  }
  fmt.Println(message)
}
```

- That's common error handling in Go: return an error as a value so the caller can check for it.

4. Slices and capacity in Golang

You'll use a Go slice. **A slice is like an array, except that its size changes dynamiccaly as you add and remove items**. The slice is one of Go's most useful types.

Capacity is the total allocated memory slots avaliable in a slice, while **length** is how many elements are currently begin used.

```
slice := []int{1, 2, 3}

fmt.Println("Length:", len(slice))   // 3 - elements currently used
fmt.Println("Capacity:", cap(slice)) // 3 - total available space
```

To create a slice, you have two main ways:

```
list := []string{"A", "B"} // Normal way - ["A", "B"]

list2 := make([]string, 2) // using make() - ["", ""]
```

When you `append()` and exceed capacity, Go automatically **allocates new memory**, **copies existing elements** to new location and **updates the slice header** with the new pointer, length and capacity.

```
list := make([]string, 0, 2)  // len=0, cap=2
list = append(list, "A")      // len=1, cap=2
list = append(list, "B")      // len=2, cap=2
list = append(list, "C")      // len=3, cap=4  ← automatic doubling!
```

When you defined a slice size, is more efficient in go.

```
// Without pre-allocated slice capacity

package main
import(
  "fmt"
  "time"
)

func main() {
  start := time.Now()
  items := []int{}

  for i:=0; i< 1000000; i++ {
    items = append(items, i)
  }

  elapsed := time.Since(start)

  fmt.Println("Time taken:", elapsed)

  // Time taken: 9.826222ms
}
```

```
// With pre-allocated slice capacity

package main
import (
  "fmt"
  "time"
)

func main() {
  start := time.Now()
  items := make([]int, 0, 1000000)

  for i := 0; i < 1000000; i++ {
    items = append(items, i)
  }

  elapsed := time.Since(start)

  fmt.Println("Time taken:", elapsed)

  //Time taken: 1.827185ms
}
```

In the first code, Go needs to copy all existing elements into the new array, let the old array **become garbage**. Each step copies everything, perform **large memcopies** plus **allocations** and **GC pressure**. This shows up as CPU time and cache misses.

5. Primitive Types and declarations

- Go assigns a default **zero value** to any variable that is decalred but not assinged a value.
- On a 32-bit CPU, int is a 32-bit signed integer like an int32. On most 64-bit CPUs, int is a 64-bit sgined integer, just like an int64. Because int ins't consistent from platform to platform, it is a compile time error to assign, compare, or perform mathematical operations between an int and an int32 or int64 without a type conversion.
- The third special name is uint. It follows the same rules as int, only it is unsigned (the values are always 0 or positive).
- Go also has bit-manipulatio nopeartor for integers. You can bit shift left and right with << and >>, or do bit masks with & (logical AND), | (logincal OR), ^ (logical XOR), and &^ (logical AND NOT). Justi like the arithmetic operators, you can allso combine all of the logical operatiors with = to modify a variable: &=, |=, ^=, &^=, <<= and >>=.
- A floating point number cannot represent a decial value exactly. Do not use them to represente money or any other value that must have and exact decimal representation.
- Strings in Go are immutable, you can reassign the value of a string variable, but you cannot change the value of the string that is assgined to it.
- Go doens't allow truthiness. In fact, no other type can be converted to a bool, implicitly or explicitly. If you want to convert from another data type to boolean. you must use one of the comparison operators. For example, to check if variable x is equal to 0, the code would be x == 0. If you want to check if strings **s** is empty, use `s == ""`.

5.1. "var" vs " := "

- You can declare multiple variables at once with var, and they can be of the same type:

```
var x, y int = 10, 20

var (
  x = 10
  y = 20
)
```

- Go also supports a short declaration format. When you are within a function, you can use the := operator to replace a var declaration that uses type inference

```
 x, y := 10, "Hello"
```

- There is one limitation on :=. If you are declaring a variable at package level, you must use var because := is not legal ouside of functions.

5.2. Naming variables and constants

- Go uses the case of the first letter in the name of a package-level declaration to determine if the item is accessible outside the package. For the common naming variables, just use the camel case pattern.

6. Composite Types

6.1. Arrays

You can declare arrays with multiples ways:

```
var x [3]int

var x [3]int{1,2,3}

x := [3]int{1,2,3}
```

Earlier I said that arrays in Go are rarely used explicitly. This is because they come with an unusual limitation: Go considers the *size* of the array to be part of the *type* of the array. This makes an array that's declared to be `[3]int` a differente type from an array that's decalred to be `[4]int`. This also means that you cannot use a variable to specify the size of an array, because types must be resolved at compile time, not at runtime.

- What's more, *you can't use a type conversion to directly convert arrays of different sizes to identical types*. Because you can't convert arrays of differente sizes into each other, you can't write a function that works with arrays of any size and you can't assign arrays of different sies to the same variable

Because of these restrictions, don't use arrays unless you know the exact length you need ahead of time.

6.2. Slices

What makes slices so useful is that you can grow slsices as needed. This is because the length of a slice is *not* part of its type. This removes the biggest limitations of arrays and allows you to write a single function that processes slices of any size.

```
var x = []int{1,2,3}
```

- Using [...] makes an array. Using [] makes a slice.
- If you want to compare two arrays, you need to use `slices` library:

```
package main

import (
  "fmt"
  "slices"
)

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

  fmt.Println(slices.Equal(x, y))
}

// You cannot compare two arrays with different type
```

You can spread arrays to add inside another:

```

func main() {
  x := []int{1, 2, 3}
  y := []int{4,5,6}
  x = append(x, y ... );

  fmt.Println(x)
}
// Output: [1 2 3 4 5 6]
```

- Go is *a call-by-value* language. Every time you pass a parameter to a function, Go makes a copy of the value that's passed in. Passing a slice to the append function actually passes a copy of the slice to the function. The function adds the values to the copy of the slice and returns the copy. You then assign the returned slice back to the variable in the calling function.

6.2.1. Capacity

Every slice also has a capacity, which is the number of consecutive memory locations reserved. This can be larger than the length. Each time you append to a slice, one or more values is added to the end of the slice. Each value added increases the length by one. When the length reaches the capacity, there's no more room to put values.

If you try to add additional values when the length equals the capacity, the append function uses the Go runtime to allocate a new backing array for the slice with a larger capacity. The values in the original backing array are copied to the new one, the new values are added to the end of the new backing array, and the slice is updated to refer to the new backing array. Finally, the updated slice is returned.

6.2.2. The Go runtime

The Go runtime provides services like *memoryh allocation and garbage collection, concurrency support, networking, and implementations of built-in types and functions.*

The Go runtime is compiled into every Go binary. This is different from languages that use a virtual machine. The drawback of including the runtime in the binary is that even the simples Go program produces a binary that's about **2MB**.

6.2.3. make

It allows you to specify the type, length, and, optionally, the capacity. Let's take a look:

```go
x := make([]int, 5);
```

The next code, show how to create a slice with a length of 5 and a capacity of 10:

```go
x := make([]int, 5, 10);
```

To append new values:

```go
func main() {
  x := make([]int, 5, 10)

  x = append(x, 5, 6, 7, 8, 4)

  fmt.Println(x)

  // Output: [0 0 0 0 0 5 6 7 8 4]
}
```

- `append` always increases the length of a slice! If you have specified a slice's length using `make`, be sure that you mean to append to it before you do so, or you might end up with a bunch of surprise zero values at the beggining of your slice.

6.2.4. Slicing slices