Go

Index

Introduction

Go, also known as Golang, is a programming language developed by Google in 2009. It is a statically-typed language with syntax similar to C, but with additional features such as garbage collection, type safety, and concurrency support. Go is known for its simplicity, reliability, and performance, making it a popular choice for building scalable and efficient systems. It is used by many large companies, including Google, Dropbox, and Netflix, for a variety of purposes, such as web development, distributed systems, and data analysis. Overall, Go is a versatile and powerful language that is worth considering for any project that requires high performance and reliability.

Setting Up a Development Environment

Go is a compiled language, so you'll need to install the Go compiler. You can download it from the Go website. Once you've installed it, you can run the following command to verify that it's working:

```
$ go version
go version go1.11.1 darwin/amd64
```

Hello World

Let's start with a simple "Hello World" program. Create a file called `hello.go` and add the following code:

```
package main

import "fmt"

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

Now, run the following command to compile and run the program:

```
$ go run hello.go
Hello World
```

Go commands

The `go` command is the main tool for managing Go source code. It has many subcommands, and we'll be using a few of them in this tutorial. Here's a list of the most important ones:

- `go mod` manages dependencies
- `go doc` prints documentation for a package or symbol
- `go build` compiles a bunch of Go source code files
- `go run` compiles and executes one or two files
- `go fmt` formats all the code in each file in the current directory
- `go install` compiles and installs a package
- `go get` downloads the raw source code of someone else's package
- `go test` runs any tests associated with the current project

Go packages

Go is organized into packages. A package is a collection of common Go source code files that are compiled together. For example, the `fmt` package contains code for formatting output, and the

`net/http` package contains code for making HTTP requests.

Every Go program is made up of packages, and it must start with a `package main` declaration. The `main` package tells the Go compiler that the package should compile as an executable program instead of a library.

Go imports

The `import` keyword is used to include code from other packages. The `fmt` package contains code for formatting output, so we need to import it to use the `Println` function.

```
package main

import "fmt"

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

Go Playground

You can also run Go code in the Go Playground. This is a great way to experiment with Go without having to install anything on your computer.

Init a Go module

Go modules are a new dependency management system that was introduced in Go 1.11. They are designed to make dependency management easier and more reliable. To create a new module, run the following command:

```
$ go mod init github.com/yourname/yourproject
```

This will create a `go.mod` file in the current directory. The `go.mod` file contains information about the module, including the module path and the Go version that was used to create it.

Package management

Go doesn't have a built-in package manager, but there are several third-party tools that you can use. The most popular one is go get, which is used to download and install Go packages. You can use it to install packages from the standard library, or from third-party repositories.

```
$ go get github.com/gorilla/mux
```

Variables

Variables are declared using the var keyword. The type of the variable is specified after the variable name. If the variable is being initialized, the type can be omitted and the compiler will infer the type from the value being assigned to the variable.

There are three ways to declare a variable:

- Explicitly: The type of the variable is explicitly specified.
- Implicitly: The compiler infers the type of the variable from the value being assigned to it.
- Short declaration: The := operator is used to declare and initialize a variable.

Explicitly declared variables:

```
var name string
var age int
```

Implicitly declared variables:

```
var name = "John Doe"
var age = 37
```

Short declaration:

```
name := "Brad"
age := 37
isCool := true
```

Constants

Constants are declared using the const keyword. Constants are declared like variables, but with the const keyword. Constants can be character, string, boolean, or numeric values.

```
const isCool = true
const pi = 3.14
const (
    a = 1
    b = 2
)
```

Primitives

Go has four primitive types: integers, floats, booleans, and strings.

numeric types

Go has two numeric types: integers and floats. Integers can be either signed or unsigned. Signed integers can be positive or negative, while unsigned integers can only be positive. The size of an integer is dependent on the operating system. On a 64-bit operating system, an integer is 64 bits. On a 32-bit operating system, an integer is 32 bits. The size of a float is 64 bits.

The different types of numeric types:

- int8
- int16
- int32
- int64
- uint8
- uint16
- uint32
- uint64
- float32
- float64

integers

Integers are whole numbers. They can be either signed or unsigned. Signed integers can be positive or negative, while unsigned integers can only be positive. The size of an integer is dependent on the operating system. On a 64-bit operating system, an integer is 64 bits. On a 32-bit operating system, an integer is 32 bits.

```
var age int = 37
```

floats

Floats are numbers with a decimal point.

There are two types of floats:

- float32
- float64

```
var pi float32 = 3.14
var e float64 = 2.71828
```

casting between types

Casting from int to float:

```
var x int = 10
var y float64 = float64(x)
```

Casting from float to int:

```
var x float64 = 10.5
var y int = int(x)
```

complex numbers

Complex numbers are numbers that have a real and imaginary part. The complex64 type has a float32 real and imaginary part, while the complex128 type has a float64 real and imaginary part.

```
var c complex64 = 5 + 5i
```

string type

The string type is a sequence of bytes. Strings are immutable, which means that once they are created, they cannot be changed. Strings are surrounded by double quotes.

```
var name string = "Brad"
```

strings package

The strings package contains many useful functions for working with strings.

The strings package functions:

- Contains: Returns true if the substring is within the string.
- Count: Returns the number of non-overlapping instances of the substring in the string.
- HasPrefix: Returns true if the string starts with the prefix.
- HasSuffix: Returns true if the string ends with the suffix.
- Index: Returns the index of the first instance of the substring in the string.
- Join: Concatenates the elements of a slice to create a single string.
- Repeat: Repeats a string multiple times.
- Replace: Replaces instances of the substring with another string.
- Split: Splits a string into a slice of substrings.
- ToLower: Converts a string to lowercase.
- ToUpper: Converts a string to uppercase.
- Trim: Removes leading and trailing instances of a substring.

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.Contains("test", "es"))
    fmt.Println(strings.Count("test", "t"))
    fmt.Println(strings.HasPrefix("test", "te"))
    fmt.Println(strings.HasSuffix("test", "st"))
    fmt.Println(strings.Index("test", "e"))
    fmt.Println(strings.Join([]string{"a", "b"}, "-"))
    fmt.Println(strings.Repeat("a", 5))
    fmt.Println(strings.Replace("foo", "o", "0", -1))
    fmt.Println(strings.Replace("foo", "o", "0", 1))
    fmt.Println(strings.Split("a-b-c-d-e", "-"))
    fmt.Println(strings.ToLower("TEST"))
    fmt.Println(strings.ToUpper("test"))
    fmt.Println(strings.Trim(" !!! test !!! ", "! "))
}
```

boolean type

The boolean type is a boolean value that can be either true or false.

```
var isCool bool = true
var isNew bool = false
```

boolean operators

Boolean operators are used to compare two boolean values. The result of a boolean operator is always a boolean value.

The different types of boolean operators:

- &&: Logical AND
- ||: Logical OR
- !: Logical NOT

```
fmt.Println(true && true)
fmt.Println(true && false)
fmt.Println(true || true)
fmt.Println(true || false)
fmt.Println(!true)
```

Arrays and Slices

Arrays and slices are used to store a list of items of the same type. Arrays and slices are zero-indexed, which means that the first item in the array or slice is at index 0.

Arrays

Arrays are a fixed length list of items. The length of the array is part of the type. Arrays are created using the following syntax:

```
var fruitArr [2]string
```

The above code creates an array that can hold two strings. The length of the array is part of the type, so arrays cannot be resized. Arrays are accessed using square brackets.

```
fruitArr[0] = "apple"
fruitArr[1] = "orange"
```

array length

The length of an array is the number of elements it contains. The length of an array is part of the type, so arrays cannot be resized.

```
fmt.Println(len(fruitArr))
```

array initialization

Arrays can be initialized with values.

```
fruitArr := [2]string{"apple", "orange"}
```

Slices

Slices are a dynamically-sized, flexible view into the elements of an array. Slices do not store any data, they just describe a section of an underlying array. Changing the elements of a slice modifies the corresponding elements of its underlying array. Other slices that share the same underlying array will see those changes.

Slices are created using the following syntax:

```
fruitSlice := []string{"apple", "orange", "grape"}
```

The above code creates a slice that can hold three strings. The length of the slice is not part of the type, so slices can be resized. Slices are accessed using square brackets.

```
fruitSlice[1] = "banana"
```

slice length and capacity

The length of a slice is the number of elements it contains. The capacity of a slice is the number of elements in the underlying array, counting from the first element in the slice.

```
fruitSlice := []string{"apple", "orange", "grape", "cherry", "banana"}
fmt.Printf("length of slice %d", len(fruitSlice))
fmt.Printf("capacity of slice %d", cap(fruitSlice))
```

slice bounds

Slices will panic if you try to access an element outside of its length.

```
fruitSlice := []string{"apple", "orange", "grape"}
fmt.Println(fruitSlice[1]) // orange
fmt.Println(fruitSlice[3]) // panic: runtime error: index out of range
```

slice literals

A slice literal is like an array literal without the length.

```
fruitSlice := []string{"apple", "orange", "grape"}
```

slice defaults

When slicing, you may omit the high or low bounds to use their defaults instead. The default is zero for the low bound and the length of the slice for the high bound.

```
fruitSlice := []string{}
//print length and capacity
fmt.Printf("length of slice %d", len(fruitSlice))
fmt.Printf("capacity of slice %d", cap(fruitSlice))
```

Maps and Structs

Maps

Maps are used to associate a value with a key. Maps are created using the following syntax:

```
emails := make(map[string]string)
```

The above code creates a map that can hold strings as keys and strings as values. Maps are accessed using square brackets.

```
package main

import "fmt"

func main() {
    emails := make(map[string]string)
    emails["Bob"] = "bob@email.com"
    emails["Sharon"] = "sharon@email.com"
    fmt.Println(emails)
}
```

Structs

Structs are used to group data together to form a record. Structs are created using the following syntax:

```
type Person struct {
    name string
    age int
}
```

The above code creates a struct that can hold a name and an age. Structs are accessed using a dot.

```
person1 := Person{name: "Brad", age: 37}
fmt.Println(person1.name)
```

Conditionals

Go has if, else if, and else statements.

```
if num := 9; num < 0 {
    fmt.Println(num, "is negative")
} else if num < 10 {
    fmt.Println(num, "has 1 digit")
} else {
    fmt.Println(num, "has multiple digits")
}
```

Loops

Go has for, for range, and for with condition statements.

for

The for loop is the most basic loop in Go. It has three components: init statement, condition expression, and post statement. The init statement is executed before the first iteration. The condition expression is evaluated before every iteration. If the condition expression evaluates to false, the loop terminates. The post statement is executed at the end of every iteration.

```
for i := 0; i < 10; i++ {
    fmt.Println(i)
}
```

for range

The for range loop is used to iterate over items in a collection. The for range loop returns two values: the index and a copy of the element at that index.

```
ids := []int{33, 76, 54, 23, 11, 2}

for i, id := range ids {
    fmt.Printf("%d - ID: %d", i, id)
}

// not using index

for _, id := range ids {
    fmt.Printf("ID: %d", id)
}
```

for with condition

Go does not have a while loop. You can use a for loop with a condition to simulate a while loop. The for with condition loop is used to iterate until a condition is met.

```
sum := 1
for sum < 1000 {
    sum += sum
}
fmt.Println(sum)
```

Switch

Switch is used to execute code based on multiple conditions. Switch is created using the following syntax:

```
package main

import "fmt"

func main() {
    switch num := 9; {
    case num < 0:
        fmt.Println(num, "is negative")
    case num < 10:
        fmt.Println(num, "has 1 digit")
    default:
        fmt.Println(num, "has multiple digits")
    }
}
```

The above code creates a switch that prints the number of cases. Switch can also be used to compare multiple values in a case.

Functions

Functions are used to group code together to perform a specific task. Functions are created using the following syntax:

```
package main

import "fmt"

func main() {
    fmt.Println(greet("Brad"))
}

func greet(name string) string {
    return "Hello " + name
}
```

The above code creates a function that takes a string and returns a string. Functions are called using the function name and passing in arguments.

Multiple return values

```
package main

import "fmt"

func main() {
    greet, age := greet("Brad", 37)
    fmt.Println(greet, age)
}

func greet(name string, age int) (string, int) {
    return "Hello " + name, age
}
```

Lambda functions

Lambda functions are anonymous functions that can be used inline. Lambda functions are created using the following syntax:

```
package main

import "fmt"

func main() {
    greet := func(name string) {
        fmt.Println("Hello " + name)
    }
    greet("Brad")
}
```

The above code creates a lambda function that takes an int and prints it. Lambda functions are called by adding parentheses after the function.

Defer

Defer is used to delay the execution of a function until the surrounding function returns. Defer is created using the following syntax:

```
func main() {
    defer fmt.Println("World")

    fmt.Println("Hello")
}
```

The above code creates a function that prints "Hello" and "World".

Variadic functions

Variadic functions are functions that can take an indefinite number of arguments. Variadic functions are created using the following syntax:

```
package main

import "fmt"

func main() {
    fmt.Println(sum(2, 3, 4, 5, 6, 7, 8, 9, 10))
}

func sum(nums ...int) int {
    total := 0
    for _, num := range nums {
        total += num
    }
    return total
}
```

The above code creates a function that takes an indefinite number of ints and prints the sum. Variadic functions are called by passing in a slice using the ellipsis operator.

Recursion

Recursion is a function that calls itself until a condition is met. Recursion is created using the following syntax:

```
func factorial(num int) int {
    if num == 0 {
        return 1
    }
    return num * factorial(num-1)
}

func main() {
    fmt.Println(factorial(7))
}
```

The above code creates a function that takes an int and returns the factorial of that int. Recursion is handled using if statements.

Closures

Closures are functions that can access variables defined outside of the function. Closures are created using the following syntax:

```
package main

import "fmt"

func main() {
    nextEven := makeEvenGenerator()
    fmt.Println(nextEven())
    fmt.Println(nextEven())
    fmt.Println(nextEven())

    nextOdd := makeOddGenerator()
    fmt.Println(nextOdd())
    fmt.Println(nextOdd())
    fmt.Println(nextOdd())
}

func makeEvenGenerator() func() uint {
    i := uint(0)
    return func() (ret uint) {
        ret = i
        i += 2
        return
    }
}

func makeOddGenerator() func() uint {
    i := uint(1)
    return func() (ret uint) {
        ret = i
        i += 2
        return
    }
}
```

The above code creates a function that returns a function that returns an int.

Pointers

Pointer types

Pointer types are used to reference a variable's memory address. Pointer types are created using the following syntax:

```
package main

import "fmt"

func main() {
    name := "Brad"  // string
    ptr := &name  // *string
    fmt.Println(ptr, *ptr)
}
```

```
package main

import "fmt"

func main() {
    intPtr := new(int)
    *intPtr = 10
    fmt.Println(intPtr, ": ", *intPtr)

    strPtr := new(string)
    *strPtr = "Hello"
    fmt.Println(strPtr, ": ", *strPtr)

    boolPtr := new(bool)
    *boolPtr = true
    fmt.Println(boolPtr, ": ", *boolPtr)

    floatPtr := new(float64)
    *floatPtr = 3.14
    fmt.Println(floatPtr, ": ", *floatPtr)

}
```

The above code creates a function that creates pointers to different types.

```
package main

import "fmt"

func main() {
    num := 10
    fmt.Println("num:\tValue Of[", num, "]\tAddr Of[", &num, "]")

    changeMe(&num)

    fmt.Println("num:\tValue Of[", num, "]\tAddr Of[", &num, "]")
}

func changeMe(z *int) {
    fmt.Println("z:\tValue Of[", z, "]\tAddr Of[", &z, "]\tValue Points To[", *z, "]")
    *z = 24
    fmt.Println("z:\tValue Of[", z, "]\tAddr Of[", &z, "]\tValue Points To[", *z, "]")
}
```

The above code creates a function that takes a pointer to an int and changes the value of the int.

Tipical problems with pointers

The main problems with pointers are:

- broken pointers: pointers that point to a memory address that no longer exists
- memory leaks: memory that is allocated but never freed
- dangling pointers: pointers that point to a memory address that has been freed

Broken pointers are pointers that point to a memory address that no longer exists. Broken pointers are created when a pointer is set to nil.

```
package main

import "fmt"

func main() {
    index := 0
    // create a pointer to an int
    ptr := &index
    // create a pointer to a pointer to an int
    pptr := &ptr
    // print the value of the pointer to a pointer to an int
    fmt.Println("pptr:\tValue Of[", pptr, "]\tAddr Of[", &pptr, "]\tValue Points To[", *pptr,
"]\tValue Points To[", **pptr, "]")
    // remove the pointer to an int
    ptr = nil
    // print the value of the pointer to a pointer to an int
    fmt.Println("pptr:\tValue Of[", pptr, "]\tAddr Of[", &pptr, "]\tValue Points To[", *pptr,
"]\tValue Points To[", **pptr, "]")

}
```

Pointer to a struct

```
package main

import "fmt"

type Person struct {
    name string
    age  int
}
func main() {
    person := Person{name: "Brad", age: 37}
    fmt.Println(person)

    changeMe(&person)

    fmt.Println(person)
}
func changeMe(p *Person) {
    p.name = "Brad Traversy"
    p.age = 38
}
```

The above code creates a function that takes a pointer to a Person and changes the name and age.

Interfaces

Interfaces are used to define a set of methods that a type must implement. Interfaces are created using the following syntax:

```
package main

import (
    "fmt"
    "math"
)

type Shape interface {
    area() float64
}

type Circle struct {
    x, y, radius float64
}

func (c *Circle) area() float64 {
    return math.Pi * c.radius * c.radius
}

type Rectangle struct {
    width, height float64
}

func (r *Rectangle) area() float64 {
    return r.width * r.height
}

func getArea(s Shape) float64 {
    return s.area()
}

func main() {
    circle := Circle{x: 0, y: 0, radius: 5}
    rectangle := Rectangle{width: 10, height: 5}

    fmt.Printf("Circle area: %f\n", getArea(&circle))
    fmt.Printf("Rectangle area: %f\n", getArea(&rectangle))
}
```

The above code creates an interface that defines a method called area that returns a float64. Interfaces are implemented by types.

Heritance

Heritance is used to define a type that inherits properties from another type. Heritance is created using the following syntax:

```
package main

import "fmt"

type Person struct {
    name string
    age int
}

type Student struct {
    Person
    grade int
}

func main() {
    s1 := Student{Person{"Bob", 20}, 5}
    fmt.Println(s1.name)
}
```

The above code creates a type called Student that inherits from the Person type. Heritance is used to access properties.

Error handling

Error handling is used to handle errors that occur during runtime. Errors are created using the following syntax:

```
package main

import (
    "fmt"
    "log"
    "os"
)

func main() {
    f, err := os.Open("test.txt")
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(f)
}
```

The above code creates an error that is assigned to the err variable.

Throw error

Throw error is used to create a custom error. Throw error is created using the following syntax:

```
package main
import (
    "errors"
    "fmt"
)
func sqrt(f float64) (float64, error) {
    if f < 0 {
        return 0, errors.New("Math: negative number passed to Sqrt")
    }
    return 42, nil
    // implementation
}
func main() {
    _, err := sqrt(-10.23)
    if err != nil {
        fmt.Println("ERROR CATCHED: ", err)
    }
}
```

The above code creates a function that returns an error if the number passed in is negative.

Panic

Panic is used to create a custom error and stop the program. Panic is created using the following syntax:

```
package main

import "math"

func sqrt(f float64) float64 {
    if f < 0 {
        panic("Math: negative number passed to Sqrt")
    }
    return math.Sqrt(f)
}

func main() {
    sqrt(-10.23)
}
```

The above code creates a function that panics if the number passed in is negative. Panic is handled using if statements.

Recover

Recover is used to recover from a panic. Recover is created using the following syntax:

```
package main

import (
    "fmt"
    "math"
)
func sqrt(f float64) float64 {
    if f < 0 {
        panic("Math: negative number passed to Sqrt")
    }
    return math.Sqrt(f)
}
func main() {
    defer func() {
        str := recover()
        fmt.Println(str)
    }()
    sqrt(-10.23)
}
```

Go Modules

Go modules are used to manage dependencies. Go modules are created using the following command:

```
go mod init github.com/bradtraversy/go_crash_course
```

The above command creates a go.mod file in the current directory. Go modules are imported using the following syntax:

```
import "github.com/bradtraversy/go_crash_course/quote"

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

To print the modules used in a project, use the following command:

```
go list -m all
```

Go Routines

Go routines are used to run code concurrently. Go routines are created using the following syntax:

```
go funcName()
```

The above code creates a go routine that runs the funcName function. Go routines are used to run code concurrently.

```
func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

Channels

Channels are used to communicate between go routines. Channels are created using the following syntax:

```
ch := make(chan string)
```

The above code creates a channel that can hold strings. Channels are used to send and receive data.

```
func say(s string, ch chan string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        ch <- s
    }
}

func main() {
    ch := make(chan string)
    go say("world", ch)
    for i := 0; i < 5; i++ {
        msg := <-ch
        fmt.Println(msg)
    }
}
```

The above code creates a channel that can hold strings. The say function sends a string to the channel every 100 milliseconds. The main function receives a string from the channel every 100 milliseconds.

Concurrency

Concurrency is the ability to run multiple tasks at the same time. Concurrency is achieved using go routines and channels.

Mutex

Mutex is used to provide safe access to data across multiple go routines. Mutex is created using the following syntax:

```
var m sync.Mutex
```

The above code creates a mutex. Mutex is used to lock and unlock data.

```
var count int
var m sync.Mutex

func increment() {
    for i := 0; i < 20; i++ {
        m.Lock()
        count++
        fmt.Println("Incrementing", count)
        m.Unlock()
    }
}

func decrement() {
    for i := 0; i < 20; i++ {
        m.Lock()
        count--
        fmt.Println("Decrementing", count)
        m.Unlock()
    }
}

func main() {
    go increment()
    go decrement()
    time.Sleep(100 * time.Millisecond)
}
```

The above code creates a mutex that is used to lock and unlock the count variable. The increment function increments the count variable 20 times. The decrement function decrements the count variable 20 times. The main function runs the increment and decrement functions concurrently.

Mutex types

Mutex types are used to provide safe access to data across multiple go routines. Mutex types are:

- sync.Mutex: Provides mutual exclusion
- sync.RWMutex: Provides mutual exclusion for readers and writers

Concurrency vs Parallelism

Concurrency is the ability to run multiple tasks at the same time. Parallelism is the ability to run multiple tasks at the same time on different cores.

Generics in Go

Generics are used to create reusable code. Generics are created using the following syntax:

```
func add[T any](a, b T) T {
    return a + b
}

func main() {
    fmt.Println(add(1, 2))
    fmt.Println(add("Hello ", "World"))
}
```

The above code creates a function that adds two numbers or two strings. The add function is generic and can be used with any type.

RxGo

RxGo is a reactive programming library for Go. Reactive programming is a programming paradigm that allows you to create asynchronous and event-based programs.

To install RxGo, use the following command:

```
go get github.com/reactivex/rxgo/v2
```

Observables use the following concepts: * Observable: An observable is a stream of data. * Observer: An observer is a function that receives data from an observable. * Subscription: A subscription is a function that is called when an observable is completed or an error occurs.

RxGo is created using the following syntax:

```
package main

import (
    "fmt"
    "time"

    "github.com/reactivex/rxgo/v2"
)

func main() {
    ch := make(chan rxgo.Item)
    go func() {
        ch <- rxgo.Of(1, 2, 3, 4, 5)
        close(ch)
    }()
    observable := rxgo.FromChannel(ch)
    observable.
        Map(func(i interface{}) interface{} {
            return i.(int) * 2
        }).
        Filter(func(i interface{}) bool {
            return i.(int) > 5
        }).
        Subscribe(context.Background(),
            func(i interface{}) {
                fmt.Println(i)
            },
            func(err error) {
                fmt.Println(err)
            },
            func() {
                fmt.Println("Completed")
            })
    time.Sleep(1 * time.Second)
}
```

The above code creates an observable that emits numbers from 1 to 5. The observable is then mapped to double the numbers. The observable is then filtered to only emit numbers greater than 5. The observable is then subscribed to and the numbers are printed.

The main methods used in RxGo are:

- Just: Creates an observable from a single value.
- From: Creates an observable from an array.
- Map: Maps the data emitted by an observable.
- Filter: Filters the data emitted by an observable.
- Range: Creates an observable that emits a range of numbers.
- Repeat: Creates an observable that emits the same data multiple times.
- Reduce: Reduces the data emitted by an observable.
-

Next: Emits data from an observable.

- Subscribe: Subscribes to an observable.

Subscribe

The subscribe method is used to subscribe to an observable. The subscribe method takes the following parameters:

- context: The context is used to cancel the subscription.
- onNext: The onNext function is called when the observable emits data.
- onError: The onError function is called when the observable emits an error.
- onComplete: The onComplete function is called when the observable is completed.

```
observable.Subscribe(context.Background(),
    func(i interface{}) {
        fmt.Println(i)
    },
    func(err error) {
        fmt.Println(err)
    },
    func() {
        fmt.Println("Completed")
    })
```

Resources

- Go Documentation
- Effective Go
- Go by Example
- Go Tour
- Awesome Go