Introducing Go

A Brief Survey of The Go Programming Language

Rebecca Skinner

March 16, 2018

Rackspace Hosting

1
What is Go

History of Go

Go was announced in 2009 by google. It is a compiled, statically typed, garbage collected language in the C family. Go was designed to provide modern memory management, concurrency models, and networking in a lightweight, readable, and performant language.

Go Today

Go has been steadily gaining a foothold. Although originally designed to be a useful systems language, it has not gained large scale adoption there compared to languages like Rust or people opting to continue using C and C++. Go's largest inroads have been in devops tooling, such as Kubernetes, Terraform, and Telegraf, well as in web application backends and middleware.

2
The Basics

10,000 Foot Overview

At a very high level Go, as a language, can be described with the following features and design choices:

- Compiled
- Type-Inferred
- Garbage Collected
- Statically Typed
- Concurrent
- Opinionated
- Single-Dispatch Object Oriented

2.1
Building Go Programs

The *go* Tool

The *go* tool is the main way to build go applications. The *go* tool has subcommands for most things you'll do as a development when building applications.

```
user@host$ go build
```

Figure 1: building a go application

Common Go Tools

go get	download and install a package and it's dependencies
go build	Build a project
go test	Run unit tests
go run	Run a go source file
go generate	Generate go source files
go doc	Show documentation for a given package

Figure 2: Common go tools and their behaviors

The Go Build Environment

Go uses environment variables to define paths to it's build tools and source files. For *nix systems, the default system installation path is */usr/local/go*. If Go is installed elsewhere on the system, you must set the *GOROOT* environment variable.

Go projects for individual users must be stored in the users *GOPATH*. There is no default value for this and it must be set per user. Individual projects must be kept at *GOROOT/src/fully-qualified-package-path*. We'll talk about fully qualified package paths later, but it might be something like *github.com/rcbops/ops-fabric/utils/go-tool*.

The Development Environment

Variable	*Description*
`GOPATH`	Path to go binaries and source for the current user
`GOROOT`	System-wide path to go binaries and packages
`GOARCH`	Target architecture to use when compiling
`GOOS`	Target operating system to use when compiling

Figure 3: Environment Variables used by Go

2.2
Variables

Variables in Go

Variables in go work much like they do in other procedural and object oriented languages. Variables are mutable unless defined as constant. Go differentiates between creating and assigning variables. Creating a variable that already exists, or assigning one that doesn't, are both errors.

Scope

Variables in go are block-scoped and closed over the enclosed contexts. A variable in an inner context may shadow a variable in the outer context.

Exporting Variables

Go uses the concept of *exported* and *unexported* variables. Variables are exported or unexported at the package level. Variable names start with a capital letter are exported, and accessible outside of the current package.

```
package foo

var Exported = 12
var unexported = "foo"
```

Creating Variables

Go provides two methods for creating variables. The `var` keyword allows you to define one or more variables in a function or package. The `:=` operator allows to define and initialize a new variable inside of a function.

Default Values

Go variables have a default “zero” value. Any newly created variables not explicitly given a value default to the zero value. The default value of a struct is a struct where each field is set to it’s default value. The default value of a pointer is `nil`.

Unused Variables

The special variable _ allows you to throw away a variable. Since Go requires you to always acknowledge return values of functions, this is a useful way to explicitly a result from a computation.

Creating Variables with `var`

The var keyword can be used inside or outside of a function to create one or more variables.

```
var x int
var (
    foo   = 10
    bar   = 90.7
    baz   = "threeve!"
    buzz string
)
```

Creating Variables with :=

:= is a shortcut for creating one or more new type-inferred variables in a function. When setting multiple variables with := at least one of the variables on the left-hand side of the expression must be new. When using := inside of a block, it will prefer to create shadow variables rather than re-assign variables inherited from the enclosing context.

```
x, y := 1, 2
{
    x, z := 3, 3
    fmt.Println(x, y, z) // 3 2 3
}
fmt.Println(x, y) // 1 2
```

Assignment

A variable can be assigned any number of times during execution. Rules of assignment are:

- You can only assign a variable that exists
- You can only assign a value of a compatible type

2.3
Pointers

About Pointers

A pointer represents the machine address of a variable. A pointer represents that address in the processes virtual memory space that contains the start of the data stored in the area of memory.

Pointer Types

A *pointer type* is a type that represents a pointer to a value of the underlying type. Pointer types in Go are represented with *. A pointer can be *nil*, or it can be assigned the value of the address of another variable. The **&** operator allows you to get the memory address of a variable. You cannot take the address of a constant. You can create a new pointer *new*.

Creating Pointers

type	pointer type	example
`string`	`*string`	`y := "foo"; x := &y`
`int`	`*int`	`x := new(int); *x = 6`

`Foo`	`*Foo`	`type Foo struct {`
		`  A int`
		`  B int`
		`}`
		`x := &Foo{B: 7}`

2.4
Arrays and Slices

Arrays and Slices

An array in go is a fixed-size set of values stored in a contiguous area of memory. A slice is a variable-sized collection of elements that uses arrays internally to manage the data. Arrays and slices in Go are 0-indexed and both share similar syntax.

Unlike C and C++, an array in Go is a value type, if you want to pass an array by reference, you need to explicitly use a pointer type.

Arrays and Slices Example

```
x := [2]int{1, 2}    // x is an array
y := []int{}         // y is an empty slice
z := make([]int, 3)  // z is {0,0,0}
```

Iterating Over Slices

It's common to iterate over elements in an array or slice. Go offers a special type of for-loop that will allow you to loop over the elements of a slice or array. The `range` keyword will return one or two values during each iteration of the loop. The first value will be the current index. The second returned value, if specified, will be the value of the array or slice at that location.

Iteration Example

```
x := []int{0, 1, 2, 3, 4}
for idx, val := range x {
  fmt.Printf("x[%d] = %d; ", idx, val)
}
```

```
x[0] = 0; x[1] = 1; x[2] = 2; x[3] = 3; x[4] = 4;
```

Common Slice Operations

Function	Example
append	`x := []int{}; x = append(x, 1};`
len	`x := []int{1,2,3}; y := len(x)`
:	`x := []int{1,2,3,4}; y := x[1:3]`

2.5
Maps

Golang Maps

Go provides maps as a fundamental data type. Maps may be keyed on any comparable values, and can have values of any Go type. The go runtime randomizes maps in order to surface bugs caused by relying on the stability of the internal hashing algorithm. The built-in map types support concurrent reads, but do not support conccurrent writes, or read/write concurrency.

Creating Maps

```
map1 := make(map[string]string, 10)
map2 := map[string]string{
    "foo":  "bar",
    "fizz": "buzz",
}
```

Accessing Elements

Elements of a map can be accessed using brackets, as with many other languages. Accessing a member of a map will return one or two values. The first value will be the value of the map at the specified key, or the zero value of the type if the element doesn't exist. The optional second return value is a bool which will be set to *true* if the value was found, and *false* if it wasn't.

Accessing Elements

```
var (
    n  int
    ok bool
    m  = map[int]int{0: 0, 2: 2, 4: 4}
)
n = m[0]      // 0
n = m[1]      // 0
n, ok = m[1]  // 0, false
n, ok = m[2]  // 2, true
```

Iterating Over Maps

The builtin *range* function allows you to iterate over keys, and optionally values, of a map. The syntax for iterating over maps is the same as it is for arrays and slices.

Iterating Over Maps

```
func main() {
    m := map[string]int{"foo": 0, "bar": 1, "baz": 2}
    for k, v := range m {
        fmt.Printf("%s => %d\n", k, v)
    }
}
```

2.6
Functions

Functions

Functions in Go start with the keyword *func*. A function make take as input zero or more values, and may return zero *or more* values. Functions in golang may return multiple values.

It is common in go for functions that may fail to return both a value and an error.

Function Example

```
func CanFail(in int) (int, error) {
  if in < 10 {
    return 0, errors.New("out of range")
  }
  return (in + 1), nil
}
```

First Class Functions

Functions are first-class types in Go. You can assign a function to a variable, use it as an argument to a function, or return one from a function.

Function Example

```
func getFunc(s string) (func(int) bool, error) {
  even := func(i int) bool { return 0 == i%2 }
  odd  := func(i int) bool { return !even(i) }
  if s == "even" {
    return even, nil
  } else if s == "odd" {
    return odd, nil
  }
  return nil, errors.New("invalid function name")
}
func main() {
  if f, err := getFunc("even"); err == nil {
    fmt.Println(f(3))
  }
  if f, err := getFunc("steven"); err == nil {
    fmt.Println(f(3))
  }
```

2.7
Packages

Packages

Packages are the basic unit of modularity in go applications. Executables should have a package called *main* at the top level of your project directory.

Packages contain one or more files, should be named after the directory that contains the files.

Package Paths

For packages that are not relative to the current working directory, packages are given by their full path relative to *GOROOT/src* or *PROJECTROOT/vendor*. Typically, the path reflects the URL you would use to *go get* a package, so for example a github project hosted at `https://github.com/rebeccaskinner/converge` containing a package, *resource* would be stored on disk at *GOROOT/src/github.com/rebeccaskinner/converge/resource* and the import path would be *"github.com/rebeccaskinner/converge/resource"*

3
Structs and Interfaces

Structs

A struct is a named structured data type. The fields in a struct may be exported or unexported. The exported fields are accessible from outside of package where the struct is defined. Within the package all fields of a struct are accessible. Structs are regular values that can be created like any other type of variable. You can set specific fields of a struct using key/value syntax.

```
type Example struct {
  Foo int
  Bar string
}

e := Example{Foo: 0, Bar: "bar"}
```

Empty Structs

Empty structs do not take up any memory at runtime. This allows them to be used in a way analagous to symbols in languages like lisp and ruby. Empty structs are written `struct{}`. It's common to use empty structs with maps when you are only concerned with presence in a set.

Embedded Structs

Structs may be embedded into one another. When structs are embedded, they inherit the embedded structs fields. The struct itself may be accessed by it's type name. To created an embedded struct, add it into the containing struct unnamed.

```
type Embedded struct {
    Val int
}
type Example struct {
    Embedded
    Foo int
    Bar string
}
func main() {
    e := Example{Embedded: Embedded{Val: 0}}
    fmt.Println(e.Val)
}
```

Receivers

Receivers are functions that are attached to a struct. This is Go's solution to object-oriented programming. Receivers may be attached to structs, or pointers to structs, where they are called *pointer receivers*. Functions attached to structs may be called on a specific instance of a struct with *.MethodName()*, this should look famillliar if you've used languages like C++, Java, or Javascript.

Receivers

```
type Example struct {
  Foo int
  Bar string
}
func (e *Example) String() string {
return fmt.Sprintf("%s: %d", e.Bar, e.Foo)
}
func main() {
  e := &Example{Foo: 0, Bar: "bar"}
  fmt.Println(e.String())
}
```

Receivers On Embedded Structs

The receivers attached to embedded structs are accessible from instances of the embedding function. If both the embedding and embedded structures have receivers for the same function, the embedding structures version will be called.

```
type Embedded struct{}
func (e *Embedded) EmbeddedFunc() string {
return "foo"
}
type Example struct{ Embedded }
func main() {
        e := &Example{}
        fmt.PrintIn(e.EmbeddedFunc())
}
```

Interfaces

An interface defines a set of functions. A structure implements an interface if it has receivers of the correct name and type for each of the functions listed by the interface. It's important to note that interface fulfillment is automatic. You may define an interface in one package that is automatically fulfilled by some type imported from some other package. This is very useful for mocking for unit tests, creating factories, or otherwise proxying types.

Interfaces

```
type Thinger interface {
  DoTheThing(int) string
}
func ShowTheThing(t Thinger) {
  fmt.Println(t.DoTheThing(4))
}
type Example struct{}
func (e *Example) DoTheThing(i int) string { return fmt.
func main() {
  ShowTheThing(&Example{})
}
```

A Bit More On Interfaces

Like structs, interfaces may be nested. It is idiomatic in Go to create many small interfaces with one or two functions, and compose them into larger interfaces.

Idiomatically, Go functions should accept interfaces as input, and return concrete types as output.

The Empty Interface

The empty interface is represented in Go as *interface*{}. The empty interface can represent any type, and is often used in place of generics when trying to create libraries with containers.

Casting Interfaces

One common challenge when working with Go applications is the need to go from an interface. Type casting in go is not limited to interfaces, but it shown here since this is by far the most common use-case.

A type cast returns two values, the result of the cast, and a bool indicating whether the types were compatible. To cast a value, call `value.(newtype)`

Casting Example

```
type Thinger interface {
        DoTheThing(int) string
}
func asThinger(t Thinger) Thinger { return t }
type Example struct{}
func (e *Example) ExampleFunc() {}
func (e *Example) DoTheThing(i int) string { return fmt.
func main() {
        thinger := asThinger(&Example{})
        example, ok := thinger.(*Example)
        if !ok {
                return
        }
        example.ExampleFunc()
}
```

4
Concurrency

Concurrency

Go is built around a message passing concurrency system with support for lightweight threads. Go uses channels for passing messages between concurrent routines.

Channels

A channel is a way of passing messages from within or between go routines. Channels are first class values and can be created, passed into and returned from functions, stored in maps, and compared.

A channel may be designated for input, output, or both, and has a size and a type. Input channels may only be written to, output channels may only be read from. The only values that can be written to, or read from, a channel are given by it's type.

The size of a channel represents the total number of items it can buffer before new writes will block.

Creating a Channel

Channels are created with *make* and are designated with the keyword *chan*. Arrows are used to indicate whether a channel is read-only, write-only, or read-write.

Channel Types

`chan int`	a read/write channel containing integers
`chan<- string`	a write-only channel containing strings
`<-chan (chan<- int)`	a read-only channel that returns write-only channels containing integers

Channel Compatability

Channels will automatically downcast to channels of more limited permissions. In other words, you may use a read-write channel where a read-only or write-only channel is expected, but not vice-versa.

Channel IO

The same arrows used to describe a channel as read-only or write-only are also used to read from or write to a channel.

```
ch := make(chan int, 1)
ch <- 1 // write the value 1 to the channel
val := <-ch // read the value 1 from the channel
```

Go Routines

Go routines are lightweight asynchronous threads that run a specific function. Go routines will run until the function they are executing returns, or the application terminates. Execution of go-routines is nondeterministic. Goroutines are spawned with they keyword *go*

```
go func(){ fmt.Println("hello")}()
```

Defer

Defer is a useful way of handling something that will occur when a function exits. Defer will cause a funciton to be executed after the final statement in the containing function, but before the value is returned to the caller. Defer can be used to free up resources or to signal to another go-routine that it has exited.

```
defer func(){ fmt.Println("exiting") }()
```

Select

Select allows the user to take an action based on selecting from some group of blocking calls. Select statements are most often used to manage control flow by coordinating actions based on input from channels being written to from various go routines.

Select Example

```
func main() {
  ch := make(chan int, 1)
  done := make(chan struct{}, 1)
  go func() {
    for idx := 0; idx < 10; idx++ {ch <- idx}
    done <- struct{}{}
  }()
  for {
    select {
    case val := <-ch:
      fmt.Println(val)
    case <-done:
      return
    }
  }
}
```

5
Questions?