Part 5: Constants

31 MARCH 2017

This is tutorial number 5 in Golang tutorial series.

Definition

The term constant is used in Go to denote fixed values such 5, -89, "I love Go", 67.89 and so on.

Consider the following code,

```
var a int = 50
var b string = "I love Go"
```

In the above code *a* and *b* are assigned to constants *50* and *I love Go* respectively. The keyword **const** is used to denote constants such as *50* and *I love Go*. Even though we do not explicitly use the keyword const anywhere in the above code, internally they are constants in Go.

Constants as the name indicate cannot be reassigned again to any other value and hence the below program will not work and it will throw error **cannot assign to a**.

```
package main

func main() {
    const a = 55 //allowed
    a = 89 //reassignment not allowed
}
```

The value of a constant should be known at compile time. Hence it cannot be assigned to a value returned by a function call since the function call takes place at run time.

```
package main

import (
    "fmt"
    "math"
)

func main() {
    fmt.Println("Hello, playground")
    var a = math.Sqrt(4)//allowed
    const b = math.Sqrt(4)//not allowed
}
```

In the above program, a is a variable and hence it can be assigned to the result of the function `math.Sqrt(4)` (We will discuss functions in more detail in a separate tutorial). But b is a constant and the value of b needs to be know at compile time. But the function `math.Sqrt(4)` will be evaluated only during run time and hence `const b = math.Sqrt(4)` throws **error main.go:11: const initializer math.Sqrt(4) is not a constant.**

String Constants

String constants are the simplest constants to start with to help understand the concepts of constants better.

Any value enclosed between double quotes is a string constant in Go. For example strings like "Hello World" or "Sam" are all constants in Go.

What type does a string constant belong to? The answer is they are **untyped**.

A string constant like "Hello World" does not have any type.

```
const hello = "Hello World"
```

In the above case we have assigned "Hello World" to a named constant **hello**. Now does the constant *hello* have a type? The answer is No. The constant still doesn't have a type.

How does the following code which assigns a variable `name` to an untyped constant `Sam` work?

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello, playground")
    var name = "Sam"
    fmt.Printf("type %T value %v", name, name)

}
```

The answer is untyped constants have a default type associated with them and they supply it if and only if a line of code demands it. In the statement `var name = "Sam"`, `name` needs a type and it gets it from the default type of the string constant "Sam" which is a *string*.

Is there a way to create a typed constant? The answer is yes. The following code creates a typed constant.

```
const typedhello string = "Hello World"
```

***typedhello* in the above code is a constant of type string.**

Go is a strongly typed language. Mixing types during assignment is not allowed. Let's see what this means by the help of a program.

```
package main

func main() {
        var defaultName = "Sam" //allowed
        type myString string
        var customName myString = "Sam" //allowed
        customName = defaultName //not allowed

}
```

In the above code, we first create a variable *defaultName* and assign it to the constant *Sam*. **The default type of the constant Sam is string, so after the assignment defaultName is of type String.**

In the next line we create a new type myString which is an alias of string.

Then we create a variable customName of type myString and assign it to the constant *Sam*. Since the constant *Sam* is untyped it can be assigned to any string variable. Hence this assignment is allowed and customName gets the type myString.

Now we have a variable defaultName of type string and another variable customName of type myString. Even though we know that myString is an alias of string, Go's strong typing policy disallows variables of one type to be assigned to another. **Hence the assignment customName = defaultName is not allowed and the compiler throws an error *main.go:10: cannot use defaultName (type string) as type myString in assignment***

Boolean Constants

Boolean constants are no different from string constants. They are two untyped constants **true** and **false**. The same rules for string constants apply to booleans so we will not repeat them here. Heres the code to explain boolean constants.

```
package main

func main() {
    const trueConst = true
    type myBool bool
    var defaultBool = trueConst //allowed
    var customBool myBool = trueConst //allowed
    defaultBool = customBool //not allowed
}
```

The above program is self explanatory.

Numeric Constants

Numeric constants include integers, floats and complex constants. There are some subtleties in numeric constants.

Lets look at some examples to make things clear.

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello, playground")
    const a = 5
    var intVar int = a
    var int32Var int32 = a
    var float64Var float64 = a
    var complex64Var complex64 = a
    fmt.Println("intVar",intVar, "\nint32Var", int32Var
}
```

In the above program the const a is untyped and has a value 5. **You may be wondering what is the default type of a and if it does have one, how do we then assign it to variables of different types**. The answer lies in the syntax of a. The following program will make things more clear.

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello, playground")
    var i = 5
    var f = 5.6
    var c = 5 + 6i
    fmt.Printf("i's type %T, f's type %T, c's type %T",

}
```

In the above program, the type of each variable is determined by the syntax of the numeric constant. **5** is an integer by syntax, **5.6** is a float and **5 + 6i** is a complex number by syntax. When the above program is run, it prints `i's type int, f's type float64, c's type complex128`

Now I hope it will be clear how the below program

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello, playground")
    const a = 5
    var intVar int = a
    var int32Var int32 = a
    var float64Var float64 = a
    var complex64Var complex64 = a
    fmt.Println("intVar",intVar, "\nint32Var", int32Var
}
```

worked. In this program, the value of a is 5 and the syntax of a is generic (it can represent a float, integer or even a complex number with no imaginary part) and hence it is possible to be assigned to any compatible type. The default type of these kind of constants can be thought of as being generated on the fly depending on the context. `var intVar int = a` requires a to be int so it becomes an int constant. `var complex64Var complex64 = a` requires a to be a complex number and hence it becomes a complex constant. Pretty neat :).

Numeric Expressions

Numeric constants are free to be mixed and matched in expressions and a type is needed only when they are assigned to variables or used in any place in code which demands a type.

```
package main

import (
    "fmt"
)

func main() {
    var a = 5.9/8
    fmt.Printf("a's type %T value %v",a, a)
}
```

In the above program 5.9 is a float by syntax and 8 is a integer by syntax. Still 5.9/8 is allowed as both are numeric constants. The result of the division is 0.7375 which is a float and hence variable *a* is of type float. The output of the program is `a's type float64 value 0.7375`.

Thats it for constants. Please shared your valuable feedback and comments.

Next tutorial - Functions

Naveen Ramanathan

iOS developer at dietco.de and Golang enthusiast.