GOLANG

A PREPRINT

Jiacheng Yang
Department of Computer Science
The University of Arizona
jiachengyang@email.arizona

Wenkai Zheng
Department of Computer Science
The University of Arizona
wenkaizheng@email.arizona

April 14, 2020

ABSTRACT

abstract Golang

***K*eywords** Go · Golang

1 Introduction

Introduction XXX

2 History

Golang was designed by Rob Pike, Ken Thompson, and Robert Griesemer in order to achieve fast execution speed and high efficiency of development. In comparisons to popular compiled languages like C, C++, and Java, programs written in Golang are only slightly slower than the same programs written in C and C++ but faster than the ones written in Java, while it needs much less time to compile source code. Another feature that Golang offers is the ability to easily writing concurrency programs. The keyword "go" enables programmers to create a go-routine, which is similar to but costs less than a thread in Java or C++.

3 Control Structures

Golang has five kinds of control structures: if-else, switch, for-loop, labels and defer. If-else, switch, for-loop, and labels work very similar in C. The keyword, defer, is designed for error handling and resource cleaning up.

3.1 Control Structures: if-else

If-else in Golang works just like in C with a minor difference that is variables can be defined in the same line of condition statement as the code below. This feature is useful when the condition statement and the body of if-else both need the same value returned by another function. Programmers can just declare a variable to receive it and use it in both places.

```
if x:= somefunction(); x%2 == 0 {
    fmt.Print(x)
    fmt.Println(" is even")
} else {
    fmt.Print(x)
    fmt.Println(" is odd")
}
```

This feature also benefits error handling. For example, when handling errors, since errors in Golang are returned by functions, callers can declare variables to receive the return values and then check errors in the condition statement. After that, errors can be handled in the body of if-else. This usage of if-else avoids declaring variables in the name-space outside of the if-else.

```
reader := bufio.NewReader(os.Stdin)
if x, err := reader.ReadString('\n'); err==nil {
    fmt.Println("read: "+ x)
} else {
    fmt.Println("fail")
}
```

3.2 Control Structures: switch

Switch in Golang is also slightly different from switch in C. One difference is that programmers no longer need to explicitly use break to break from cases; instead, programmers need to use the keyword, fallthrough, to explicitly execute the case below the current one. For example, if x in the code below is 2 or 3, the program will still print: "x is a prime less than 6."

```
switch x {
case 2:
    fallthrough
case 3:
    fallthrough
case 5:
    fmt.Println("x is a prime less than 6.")
}
```

Another way to accomplish the logic of the code above is to provide expected values in a single case as the example below.

```
switch x {
case 2, 3, 5, 7:
    fmt.Println("x is a prime less than 10")
default:
    fmt.Println("x is not a prime less than 10")
}
```

If a variable for comparison is not provided, switch in Golang then works just like multiple if-else. Each case is evaluated from top to bottom, and the one evaluated to true will be executed.

```
switch {
case variable%2 == 0:
    fmt.Println("variable is divisible by 2")
case variable%3 == 0:
    fmt.Println("variable is divisible by 3")
case variable%5 == 0:
    fmt.Println("variable is divisible by 3")
default:
    fmt.Println("variable is not divisible by 2, 3 or 5")
}
```

3.3 Control Structures: for-loop

In Golang, iterations are implemented only by for-loops. This means there are no while and do-while loops. However, they can be easily and elegantly simulated by for-loops. A simple for-loop can be implemented as in C.

```
for i := 0; i < 10; i++ {
    sum += i
}
```

When a for-loop is not given initialization and change of variable, it works like a while loop, and programmers do not have to provide extra semicolons as the code below.

```
for i < 10 {
    sum += i
    i++
}
```

For-loop in Golang can also work as for-loop in python. Programmers need to use the range keyword in a for-loop as the code below. Range keyword goes over an array or a string and assigns the index to the first variable and the value to the second variable defined in the for-loop. This feature allows programmers to easily iterate over an array or a string.

```
for i, char := range "abcdef"{
    fmt.Printf("Index %d is %s\n",i,string(char))
}
```

3.4 Control Structures: label and goto keyword

Like C, Golang supports labels and goto keyword. Programmers can define a label by a name followed by a colon, and then it can be used by goto keyword. For instance, the code below will skip printing 2 and only print 1 and 3.

```
    fmt.Println("1")
    goto A
    fmt.Println("2")
A:
    fmt.Println("3")
```

Moreover, a loop can be labeled in order to break or continue a specific loop. For example, the code below will only print a = 0, a = 1, and then i = 2.

```
Loop1:
    for a:=0;a < 10; a++ {
        fmt.Println("a = "+ strconv.Itoa(a))
        for i := 1; i < 4; i++ {
            fmt.Println("i = "+ strconv.Itoa(i))
            if i%2 == 0 {
                break Loop1
            }
        }
}
```

3.5 Control Structures: defer

Golang has a unique keyword, defer. As mentioned at the beginning of this section, defer is designed to deal with error handling and resource cleaning up. Programmers can defer a call of a function immediately after the return of the current function. For example, the code snippet below will first print zero and then one.

```
func foo() {
    defer func() { fmt.Println("one") }()
    fmt.Println("zero")
}
```

When there are multiple defer's, it follows the rule of FILO (first in last out). So, the code snippet below will print two after one.

```
func deferPrintln() {
    defer func() { fmt.Println("two") }()
    defer func() { fmt.Println("one") }()
}
```

Because a deferred function will always be executed regardless of errors, programmers can close resources, such as IO-stream, in a deferred function as the code below.

```
func writeTo(fileName string) {
    f, _ := os.Create(fileName)
    defer func() {f.Close()}()
    //write file
}
```

4 Data Types

Golang is a statically typed language and is almost strongly typed with a package called unsafe that helps programmers to escape from the type system. The built-in types are int, float, rune, byte, string, and pointers. It also has arrays and maps that rescue programmers from building their own.

4.1 Data Types: discrete types

In Golang, int, float, rune, byte are discrete types, and int and float are actually two groups of types. Int consists of int, uint, int8, uint8, int16, uint16, int32, uint32, int64, and uint64. Similarly, float consists of float32 and float64.
The type, rune, is similar to char in other languages; however, rune is designed to store Unicode as Golang tends to be a language for international usages. For example, the code below shows a rune stores a Japanese Hiragana letter - は and prints its Unicode number.

```
var p8 rune = は
fmt.Printf("%#U %+q %c\n", p8, p8, p8)
```

Byte represents 8 bits as widely accepted in the computer science world, and it works just as in other language like Python.
All of the types introduced in this sub-section has an associated pointer type that stores the address of the value. However, there is not a void pointer that allows programmers to step around the type system, and the compiler forbids casting a pointer to a different type. By this prohibition, Golang avoids being a weakly-typed language like C.

4.2 Data Types: composite types

String, array, and map are built-in composite types. Strings are actually an array of rune, yet it is tricky to manually iterated over or indexed because strings can contain languages other than English, and they take more indexes than English letters. For example, the string in the code below contains English letters, Chinese letters, and Japanese letters. The index of g in the string is 2 as it is the third letter; however, the index of 日 is 13, and the index of 本 is 16, which means 日 takes 3 indexes to hold it. Thus, it may be a problem to index and slice a string. However, it is fine to use a for-range loop to iterate over a string since for-range was designed to treat strings specially in order to solve the problem.

```
s := "English中文日本語は"
for i, c := range s {
    fmt.Printf("%d , %c\n", i, c)
}
```

Array is another built-in composite type. Arrays can hold other built-in types, and, as many other languages, arrays can be indexed and sliced. A slice of an array is based-on the original array. Thus, a modification on a slice is also applied to the original array. A slice can also be constructed by the make function as the code below, and programmers can specify the length and the capacity of the slice.

```
aSlice := make([]string, 5)
aSlice[0] = ""
```

Map is just like diction in python, but the types of keys and values are set in declaration. A map also has to be constructed by the make function as the code below.

```
m := make(map[rune]string)
m['0'] = "zero"
```

```
m['1'] = "one"
m['2'] = "two"
delete(m, '0')
```

4.3 Data Types: structures and interfaces

Structures and interfaces are types that programmers can define. A structure can be simply defined as the code below.

```
type Student struct {
    Age   int
    Name  string
    Grade int
}
```

Additionally, a structure can be constructed by giving the values as the code below, and if a value is not given, Golang will assign the default value of the type.

```
derrick := Student{22, "Derrick", 100}
frank := Student{Age: 19, Name: "Frank", Grade: 99}
underwood := Student{Age: 21, Name: "Underwood"}
```

Golang uses name equivalence for structures, so the code below is illegal and produces a compile time error.

```
type student2 Student
s := Student{Age: 22, Name: "Derrick"}
var s2 student2 = s
```

Structures can also binds with methods. Normally, a function in Golang is defined as below.

```
func getName(s Student) string {
    return s.Name
}
getName(s)
```

However, there is another way to define it.

```
func (s Student) getName()string {
    return s.Name
}
s.getName()
```

By defining it in this way, it can be called as a JAVA method. Interface is also a type in Golang. Unlike structures, interfaces only contains signatures of methods as the code below.

```
type adder interface {
    add() int
}
type pair struct {
    First  int
    Second int
}
func (a pair) add() int {
    return a.First + a.Second
}
```

An instance of a interface has to have the corresponding methods. Thus, instances of interfaces are usually structures. In the code above, a instance of pair can be a instance of adder.

4.4 Data Types: polymorphism

Golang is criticized for being lack of support of parametric polymorphism. Although interfaces provide a solution to treat different structures that have same methods associated with, this solution cannot work with primitive types since

they do not have associated methods. Thus, programmers usually have to write a function multiple times to support for different primitive types.
Inclusion polymorphism on structures can only be done by embedding, which means a structure cannot extend another. Also, interfaces cannot extend and embed each other, but, as long as, two interfaces define exactly the same methods, any instance of them can be converted without any problem.

5 Subprograms

Subprograms xxx

6 Summary

Summary xxx

References

References xxx