Notes on Golang

wycd.net

January, 2014

Abstract

This is a collection of non-comprehensive notes by a programmer learning the Go programming language who was previously familiar with C and C++. These notes are meant to provide clarification for language features with examples. Although the Golang specs are already very clear, it could prove helpful to have additional examples of usage. If desired, please email any errors or ambiguities to `mail@wycd.net`.

```
go version go1.1.2 linux/amd64
```

Contents

1 Syntax

1.1 Variables

- All variables are implicitly initialized to their zero-value.

  `http://golang.org/ref/spec#The_zero_value`

- No implicit conversions. `T(v)` must be used to convert the value `v` to the type `T`:

```
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)
```

- These are the same:

```
var foo string = "text"
var foo = "text"
foo := "text"
```

- All types implement the empty interface `interface{}`
- Type assertion may be used `x.(T)` to verify an expression `x` of `interface{}` type is not `nil` and stores a value of type `T`.
- Type inference with decimal numbers results in `float64`.

```
var t interface{} = 1.0 /* for type assertion */
f64 := t.(float64)      /* okay */
f32 := t.(float32)      /* error */
```

- Selectors dereference pointers to structs automatically.

```
type Person struct { name string }

var (
```

```
    o = Person{name: "Fred"}
    p = &o
)

func main() {
    /* the following statements have the same effect */
    o.name = "George"
    p.name = "George"
}
```

- structs may have anonymous fields and can then satisfy methods that have receivers for those field types. The field is *promoted* and can by accessed by a selector with that type's name.

```
package main

import "fmt"

type Int int

type IntWrap struct {
        Int
}

func (i Int) test() {
        fmt.Println("test okay:", i)
}

func main() {
        a := IntWrap{42}
        a.test()
    fmt.Println("tested value:", a.Int)
}
```

- The `new(T)` allocates space for a T value and returns its address.

```
var i *int = new(int)
fmt.Println(*i)
```

- Slices (e.g. `[]int`) are wrappers around arrays containing metadata, allowing for convenient features such as (fake) dynamic sizing. They must be created with `make()`. See `http://blog.golang.org/go-slices-usage-and-internals`.

```
var s []int = make([]int, 5, 10) /* s is [0 0 0 0 0] */
len(s) /* 5 */
cap(s) /* 10 */

s = s[:cap(s)] /* s is [0 0 0 0 0 0 0 0 0 0] */
len(s) /* 10 */
cap(s) /* 10 */

s = s[2:] /* s is [0 0 0 0 0 0 0 0]                  */
          /* elements 0 and 1 are now "lost forever" */
          /* ...no such thing as s[-2:]              */
```

- Maps are created with `make()` and work like maps in any other language. They can return an optional value `ok` which is `true` if the value exists and `false` if not. If the value does not exist, the type's default `zero value` is returned as the value. In gc version 6g, maps are implemented as hashmaps: `http://golang.org/src/pkg/runtime/hashmap.c`.

```
m := make(map[string]int)
m["hello"] = 4
value := m["hello"] /* value is 4 */
value, ok := m["hello"] /* value is 4, ok is true */
value, ok = m["cheese"] /* value is 0, ok is false */
```

- Channels work like pipes. They behave like FIFO data structures and are safe to use concurrently. Data goes in direction of the `<-` arrows. They will block on send or receive by default.

```
ic := make(chan int)      /* make a channel that can pass ints */
go func() { ic <- 5 }()   /* send 5 into ic (blocking until received) */
val := <- ic              /* receive the 5 from ic into val */
```

- Channels may be created with a buffer size as a second argument to the `make()` function. The default buffer size is zero, causing channel senders to block until a channel receiver dequeues the value. With buffers, queueing without a channel receiver becomes a non-blocking operation as long as the channel is not full.

```
ic := make(chan int, 100) /* make buffered channel that can pass ints */
ic <- 5                   /* send 5 into ic (non-blocking due to buffer) */
val := <- ic              /* receive the 5 from ic into val */
```

1.2 Control Flow

- `if...else` statements may have a initializing statement, and they require braces but not parentheses.

```
if f := 1; f < 3 {
    ...
} else {
    ...
}
```

- The `range` clause of the for loop allows for iteration over slices or maps.

Range Expression	Literal Expression	1st value	2nd value
array or slice	a [n]E, *[n]E, or []E	index i int	a[i] E
string	s string type	index i int	r rune
map	m map[K]V	key k K	m[k] V
channel	c chan E, <-chan E	element e E	

More information and above table source here: `http://golang.org/ref/spec#For_statements`

```
shifts := []int{1, 2, 4, 8, 16, 32}
for i, v := range shifts { /* i is optional */
    fmt.Printf("1 << %d = %d\n", i, v)
}
```

- The `switch` statement automatically terminates at the end of a case, but may fallthrough to subsequent cases using the `fallthrough` keyword. Case statements evaluate expressions if not given a variable to match against.

```
v := 5
switch { /* if...else style-usage */
case v%2 == 0:
    fmt.Println("even")
default:
    fmt.Println("odd")
}

c := ' '
switch c { /* match-style usage */
case ' ', '\t', '\n', '\r':
    fmt.Println("space")
default:
    fmt.Println("not a space")

}
```

- The `select` statement is structured like the switch statement, but has the main functionality of triggering depending on what `channel`s are available.

```
var c1, c2 chan int
select {
case c1 <- (<-c2):
        fmt.Println("popped a value from c2 into c1")
        break
```

```
default:
        fmt.Println("either c2 or c1 not ready")
}
```

1.3 Functions

- These are the same:

```
func add(x int, y int) int {...}
func add(x, y int) int {...}
```

- Functions can return multiple values:

```
func beginningMiddleEnd(x float64) (float64, float64, float64) {
    return 0.0, (x / 2), x
}

func main() {
    b, m, e := beginningMiddleEnd(12)
}
```

  Note that these return values do not form a first-class object, so they may not be accessed as members (i.e. `f()[0]`).

- Named return variables allow for implicit return values.

```
func polarToCartesian(r, theta float64) (x, y float64) {
    x = r * math.Cos(theta)
    y = r * math.Sin(theta)
    return
}
```

- Functions may be assigned to variables because they are values.

```
var inc func(int) int = func(x int) int {
    return x + 1
}
```

- A function may be a closure and reference variables outside of its body.

```
func accumulator() func(int) []int {
    s := make([]int, 0)
    return func(x int) []int {
        s = append(s, x)
        return s
    }
}

func main() {
    var acc func(int) []int = accumulator()
    acc(4)
    acc(3)
    acc(2)
    acc(1)
    acc(0) /* expression is [4 3 2 1 0] */
}
```

- goroutines allow for concurrency. It is used as `go f()` to run `f()` as a parallel goroutine. Channels are commonly used to control goroutines, but Go also provides locks and wait groups in standard `sync` package. However, there are currently not provided methods similar to `join` for threads. The function call may be a closure:

```
package main

import "fmt"

func main() {
    done := make(chan bool)
    go func() { fmt.Println("test"); done <- true }()
```

```
    <-done
}
```

1.4 Methods and Interfaces

- Methods are "glued" on to struct types.

```
type Rectangle struct {
    W, H float64
}

func (r *Rectangle) Area() float64 {
    return r.W * r.H
}

func main() {
    var r *Rectangle = &Rectangle{8, 10}
    var area float64 = r.Area()
}
```

- In the above function definition for `Area()`, the text `(r *Rectangle)` represents a *receiver* for the type `*Rectangle`. Depending on if the type is `*Rectangle` or `Rectangle`, the method receives a pointer to the original or a new copy, respectively.

- Interfaces specify methods that must be implemented to be considered that interface. "No explicit declaration of intent" is required, and a type implements an interface just by having all the required methods.

```
type Shape interface {
    Area() float64
}

type Rectangle struct {
    W, H float64
}
```

```
func (r *Rectangle) Area() float64 {
    return r.W * r.H
}

func main() {
    var s Shape = &Rectangle{5, 10} /* this is fine */
}
```

- In the above example, a method with the receiver `(r Rectangle)` would have also sufficed because of method sets. A pointer type `*T` has methods with receivers of both `*T` and `T` in its method set.

  `http://golang.org/ref/spec#Method_sets`

2 Packages

2.1 `fmt`

- `fmt` printing methods can print values of primitives and try to use reflection to call the `String()` routine of an object. They will otherwise try to print the object's address in `&{0xdeadbeef}` form. See `printArg()` in `fmt/print.go` for details.

- `fmt.Println(arg0, arg1, ...)` automatically adds a space between arguments when printing.

2.2 `errors`

- The built-in interface type `error` only has one method `Error() string`. Anything satisfying this implements the `error` interface.