Concurrent Programming in Go

http://www.golang.org

Raghu Prabhakar
raghu@cs.ucla.edu

Rohit Kumar
rohitkk@cs.ucla.edu

June 3, 2011

1 Abstract

The Go programming language was released by Google in November 2009. The language was designed and implemented by the creators of Unix: Rob Pike and Ken Thompson. It's a general purpose programming language intended for systems programming. This manual gives a brief overview of the language's support for concurrency.

2 Introduction

Go was designed as a general purpose system's programming language. It borrows from C (pointers, goto), Newsqueak (channels) [2]. It has garbage collection, is statically typed and has compilers on Windows, Linux and MacOS X.

The primitives for concurrency present in Go are similar to those in CSP [6]. Goroutines are used to spawn new threads of execution and channels are used to communicate between goroutines. Although this model of concurrency is the recommended one, the programmer can choose to use shared memory and locks. The following sections talk about Go's concurrency primitives and their usage.

3 Goroutines

Goroutines are functions executing in a separate thread. To start a goroutine, you prefix the function call with the `go` keyword.

```
go map_block(start, end)
```

The above statement will start the `map_block` function as a goroutine in a separate thread. This is an asynchronous call and the control will not wait for the execution of `map_block` to finish before executing the next statement[4]. When the goroutine finishes it will exit silently. The goroutine shares the same memory as other goroutines and the main thread of execution. Multiple goroutines might be multiplexed on the same system thread.

By default the Go runtime will only use one processor to schedule goroutines. To use more than one processor, one should call the `runtime.GOMAXPROCS` function. For example:

```
import ("runtime")
func main() {
        runtime.GOMAXPROCS(4)
}
```

The above code instructs the runtime to use 4 processors.

4 Channels

Channels are the main form of syncrhonization provided by Go. They can be used to send and receive values between goroutines. Channels are typed.

```
1    ch := make(chan int)
2    go func() {
3            v:= <-ch)
4    }()
5    ch <- 23
```

On Line 1 we create a new channel using `make`. The default channels are unbuffered and will block on both send and receive. We then spawn a new goroutine which receives a value from the channel (Line 3). Finally we send a the number `23` through the channel (Line 5).

To send a value through a channel one uses the `<-` operator with the channel on the left hand side (Line 5). To receive a value, place the channel on the right hand side of the `<-` operator.

The order of send and receive is important. If we had done `ch<-23` before Line 2. The program would block and never execute the `go` statement. As an unbuffered channel blocks both on a send and a receive [1].

Channels can also be used to wait for goroutines to finish. For example:

```
1     ch := make(chan bool)
2     for i:=0; i<n; i++ {
3             go func() {
4             //do something
5                     ch <- true
6             }()
7     }
8     for i:=0; i<n; i++ {
9             <-ch
10    }
```

In the above code snippet we spawn an anonymous goroutine `n` times (Line 3). Each one of these goroutines sends a value `true` on the channel `ch` after it has finished its task (Line 5). In the main goroutine we wait for all child goroutines to finish by receiving the values on the channels and discarding them (Line 9). We know that after Line 10, all the spawned goroutines have finished.

There are other patterns of using channels, many of which are illustrated in [1].

5 WaitGroup

Waitgroups are a better construct to syncrhonize the completion of goroutines. They are present in the `sync` package. The same code listed in the previous section can be rewritten using a `WaitGroup`.

```
1     var wg sync.WaitGroup
2     for i:=0; i<n; i++ {
3             wg.Add(1)
4             go func() {
5             //do something
6                     wg.Done()
7             }()
8     }
9     wg.Wait()
10    }
```

Here we see that the main goroutine calls add to set the number of goroutines to wait for (Line 3). When each goroutine has finished execution, it calls the `Done` method (Line 6) on the `WaitGroup`. The main routine waits for the all the child goroutines to finish by calling `Wait` (Line 9) [5].

6 Select

The `select` statement is used to choose a send or receive from a set of channels. The statement is structured like a `switch` statement, with each case being either a send to or a receive from a channel. Each of these cases is evaluated from the top to bottom. Out of all the send/receive expressions which can proceed one is selected and executed. See [4] for examples.

7 Locks

Within the `sync` package, which comes with the standard Go distribution, there are two types of locks: Mutex and Reader Writer Lock. The recommended way of synchronizing is channels. The locks provided here are used to build higher level synchronization mechanisms. See [5] for more details.

8 Once

The `Once` structure can be used to execute a particular function only one time. For example:

```
1     var once sync.Once
2     for i:=0; i<n; i++ {
```

```
3               go func() {
4                       //do something
5                       once.Do(cleanup)
6               }()
9       }
```

In the above code although multiple goroutines will reach Line 5, only one of them will get to execute the `cleanup` function.

9 Conclusion

The Go programming language has been around for almost 2 years. Although it was experimental when it was released it has since been deployed in production environments [7]. Given that it's general purpose, imperative, easy to learn and provides native concurrency primitives, we believe that it will be adopted by many more developers in the future.

References

[1] The Go Authors. Effective go. http://golang.org/doc/effective_go.html, 2011.

[2] The Go Authors. Go faq. http://golang.org/doc/go_faq.html#ancestors, 2011.

[3] The Go Authors. The go memory model. http://golang.org/doc/go_mem.html, 2011.

[4] The Go Authors. The go programming language specification. http://golang.org/doc/go_spec.html, 2011.

[5] The Go Authors. Package sync. http://golang.org/pkg/sync/, 2011.

[6] Andrew Gerrand. Share memory by communicating. http://blog.golang.org/2010/07/share-memory-by-communicating.html, 2010.

[7] Blake Mizerany Keith Rarick. Go at heroku. http://blog.golang.org/2011/04/go-at-heroku.html, 2011.