Concurrency

Large programs are often made up of many smaller sub-programs. For example a web server handles requests made from web browsers and serves up HTML web pages in response. Each request is handled like a small program.

It would be ideal for programs like these to be able to run their smaller components at the same time (in the case of the web server to handle multiple requests). Making progress on more than one task simultaneously is known as concurrency. Go has rich support for concurrency using goroutines and channels.

Goroutines

A goroutine is a function that is capable of running concurrently with other functions. To create a goroutine we use the keyword `go` followed by a function invocation:

```
package main

import "fmt"

func f(n int) {
  for i := 0; i < 10; i++ {
    fmt.Println(n, ":", i)
  }
}

func main() {
  go f(0)
  var input string
  fmt.Scanln(&input)
}
```

This program consists of two goroutines. The first goroutine is implicit and is the main function itself. The second goroutine is created when we call `go f(0)`. Normally when we invoke a function our program will execute all the statements in a function and then return to the next line following the invocation. With a goroutine we return immediately to the next line and don't wait for the function to complete. This is why the call to the `Scanln` function has been included; without it the program would exit before being given the opportunity to print all the numbers.

Goroutines are lightweight and we can easily create thousands of them. We can modify our program to run 10 goroutines by doing this:

```
func main() {
  for i := 0; i < 10; i++ {
    go f(i)
  }
  var input string
  fmt.Scanln(&input)
}
```

You may have noticed that when you run this program it seems to run the goroutines in order rather than simultaneously. Let's add some delay to the function using `time.Sleep` and `rand.Intn`:

```
package main

import (
  "fmt"
  "time"
  "math/rand"
)

func f(n int) {
  for i := 0; i < 10; i++ {
    fmt.Println(n, ":", i)
    amt := time.Duration(rand.Intn(250))
    time.Sleep(time.Millisecond * amt)
  }
}

func main() {
  for i := 0; i < 10; i++ {
    go f(i)
  }
  var input string
  fmt.Scanln(&input)
}
```

`f` prints out the numbers from 0 to 10, waiting between 0 and 250 ms after each one. The goroutines should now run simultaneously.

Channels

Channels provide a way for two goroutines to communicate with one another and synchronize their execution. Here is an example program using channels:

```
package main

import (
  "fmt"
  "time"
)

func pinger(c chan string) {
  for i := 0; ; i++ {
    c <- "ping"
  }
}

func printer(c chan string) {
```

```
func printer(c chan string) {
  for {
    msg := <- c
    fmt.Println(msg)
    time.Sleep(time.Second * 1)
  }
}

func main() {
  var c chan string = make(chan string)

  go pinger(c)
  go printer(c)

  var input string
  fmt.Scanln(&input)
}
```

This program will print “ping” forever (hit enter to stop it). A channel type is represented with the keyword `chan` followed by the type of the things that are passed on the channel (in this case we are passing strings). The `<-` (left arrow) operator is used to send and receive messages on the channel. `c <- "ping"` means send `"ping"`. `msg := <- c` means receive a message and store it in `msg`. The `fmt` line could also have been written like this: `fmt.Println(<-c)` in which case we could remove the previous line.

Using a channel like this synchronizes the two goroutines. When `pinger` attempts to send a message on the channel it will wait until `printer` is ready to receive the message. (this is known as blocking) Let's add another sender to the program and see what happens. Add this function:

```
func ponger(c chan string) {
  for i := 0; ; i++ {
    c <- "pong"
  }
}
```

And modify `main`:

```
func main() {
  var c chan string = make(chan string)

  go pinger(c)
  go ponger(c)
  go printer(c)

  var input string
  fmt.Scanln(&input)
}
```

The program will now take turns printing “ping” and “pong”.

Channel Direction

We can specify a direction on a channel type thus restricting it to either sending or receiving. For example pinger's function signature can be changed to this:

```
func pinger(c chan<- string)
```

Now `c` can only be sent to. Attempting to receive from c will result in a compiler error. Similarly we can change printer to this:

```
func printer(c <-chan string)
```

A channel that doesn't have these restrictions is known as bi-directional. A bi-directional channel can be passed to a function that takes send-only or receive-only channels, but the reverse is not true.

Select

Go has a special statement called `select` which works like a `switch` but for channels:

```
func main() {
  c1 := make(chan string)
  c2 := make(chan string)

  go func() {
    for {
      c1 <- "from 1"
      time.Sleep(time.Second * 2)
    }
  }()

  go func() {
    for {
      c2 <- "from 2"
      time.Sleep(time.Second * 3)
    }
  }()

  go func() {
    for {
      select {
      case msg1 := <- c1:
        fmt.Println(msg1)
      case msg2 := <- c2:
        fmt.Println(msg2)
      }
    }
  }()

  var input string
  fmt.Scanln(&input)
}
```

This program prints “from 1” every 2 seconds and “from 2” every 3 seconds. `select` picks the first channel that is ready and receives from it (or sends to it). If more than one of the channels are ready then it randomly picks which one to receive from. If none of the channels are ready, the statement blocks until one becomes available the statement blocks until one becomes available.

The `select` statement is often used to implement a timeout:

```
select {
case msg1 := <- c1:
  fmt.Println("Message 1", msg1)
case msg2 := <- c2:
  fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
  fmt.Println("timeout")
}
```

`time.After` creates a channel and after the given duration will send the current time on it. (we weren't interested in the time so we didn't store it in a variable) We can also specify a `default` case:

```
select {
case msg1 := <- c1:
  fmt.Println("Message 1", msg1)
case msg2 := <- c2:
  fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
  fmt.Println("timeout")
default:
  fmt.Println("nothing ready")
}
```

The default case happens immediately if none of the channels are ready.

Buffered Channels

It's also possible to pass a second parameter to the make function when creating a channel:

```
c := make(chan int, 1)
```

This creates a buffered channel with a capacity of 1. Normally channels are synchronous; both sides of the channel will wait until the other side is ready. A buffered channel is asynchronous; sending or receiving a message will not wait unless the channel is already full.

Problems

- How do you specify the direction of a channel type?
- Write your own `Sleep` function using `time.After`.
- What is a buffered channel? How would you create one with a capacity of 20?

← Previous Index Next →