Channels, Timeouts, and Tickers

This lesson explains how to set the responses of goroutines with time as a controlling factor.

WE'LL COVER THE FOLLOWING

The `time` package has some interesting functionality to use in combination with channels. It contains a struct `time.Ticker`, which is an object that repeatedly sends a time value on a contained channel `C` at a specified time interval:

```
type Ticker struct {
  C <-chan Time // the channel on which the ticks are delivered.
  // contains filtered or unexported fields
  ...
}
```

A `Ticker` can be very useful when, during the execution of goroutines, something (logging a status, a printout, a calculation, and so on) has to be done periodically at a certain time interval. It is stopped with `Stop()`; use this in a `defer` statement. All this fits nicely in a select statement:

```
ticker := time.NewTicker(updateInterval)
defer ticker.Stop()
...
select {
  case u:= <- ch1:
    ...
  case v:= <- ch2:
    ...
  case <- ticker.C:

   logState(status) // e.g. call some logging function logState

  default: // no value ready to be received
    ...
}
```

The `time.Tick()` function with signature `func Tick(d Duration) <-chan Time` is useful when you only need access to the return channel and don't need to shut it down. It sends out the time on the return channel with periodicity `d`, which is a number of *nanoseconds*. It is handy to use when you have to limit the rate of processing per unit time like in the following code snippet (the function `client.Call( )` is an RPC-call, the details are not further specified here):

```
import "time"

rate_per_sec := 10
var dur Duration = 1e9 / rate_per_sec
chRate := time.Tick(dur) // a tick every 1/10th of a second
for req := range requests {
  <- chRate // rate limit our Service.Method RPC calls
  go client.Call("Service.Method", req, ...)
}
```

The net effect is that new requests are only handled at the indicated rate, which means the channel `chRate` blocks higher rates. The rate per second can be increased or decreased according to the load and / or the resources of the machine.

A `tick` type looks exactly the same as a `Ticker` type (it is constructed with `time.Tick(d Duration)`), but it sends the time only once, after a `Duration d`. There is also a function `time.After(d)` with the signature: `func After(d Duration) <-chan Time`. After `Duration d`, the current time is sent on the returned channel; this is equivalent to `NewTimer(d).C` It resembles `Tick()`, but `After()` sends the time only once. The following listing shows a concrete example, and also nicely illustrates the `default` clause in `select`:

```
package main
```

```
import (
"fmt"
"time"
)

func main() {
  tick := time.Tick(1e8)
  boom := time.After(5e8)
  for {
    select {
      case <-tick:
        fmt.Println("tick.")
      case <-boom:
        fmt.Println("BOOM!")
        return
      default:
        fmt.Println(" .")
        time.Sleep(5e7)
    }
  }
}
```

Timer goroutines

In the code above, look at **line 8**. `time.Tick` generates a tick every **1/8th** of a second, sending the current time on the return channel. At **line 9**, `time.After` generates the same, but only one time is sent after **5/8th** of a second, where `tick` and `boom` are the receiving channels.

The `select`, in the infinite for-loop starting at **line 10**, by default prints out a `.` and sleeps for a while. If there is a value on the `tick` channel **(line 12)**, it prints **tick**. When there is a value on the `boom` channel **(line 14)**, it prints **boom** and returns, stopping the program.

Simple timeout pattern

1st variant

We want to receive from a channel `ch`, but we want to wait at most 1 second for the value to arrive. Start by creating a signaling channel and launching a lambda goroutine that sleeps before sending on the channel:

```
timeout := make(chan bool, 1)
go func() {
  time.Sleep(1e9) // one second
  timeout <- true
}()
```

Then, use a `select` statement to receive from either `ch` or timeout: if nothing arrives on `ch` in the 1 s time period, the timeout case is selected and the attempt to read from `ch` is abandoned.

```
select {
  case <-ch:
  // a read from ch has occurred
  case <-timeout:
  // the read from ch has timed out
  break
}
```

2nd variant-abandoning synchronous calls

We could also use the `time.After()` function instead of a timeout-channel. This can be used in a `select` to signal a timeout or stop the execution of goroutines. When in the following code snippet, `client.Call` does not return a value to channel `ch` after `timeoutNs ns`; the timeout case is executed in the `select`:

```
ch := make(chan error, 1)
go func() { ch <- client.Call("Service.Method", args, &reply) } ()
  select {
    case resp := <-ch:
      // use resp and reply
    case <-time.After(timeoutNs):
     // call timed out
   break
}
```

Note that the buffer size of 1 is necessary to avoid deadlock of goroutines and guarantee garbage collection of the timeout channel.

3rd variant

Suppose we have a program that reads from multiple replicated databases simultaneously. The program needs only one of the answers, and it should accept the answer that arrives first. The function `Query` takes a slice of database connections and a query string. It queries each of the databases in parallel and returns the first response it receives:

```
func Query(conns []Conn, query string) Result {
  ch := make(chan Result, 1)
  for _, conn := range conns {
    go func(c Conn) {
      select {
        case ch <- c.DoQuery(query):
          default:
      }
    }(conn)
  }
return <- ch
}
```

Here, again the result channel `ch` has to be buffered; this guarantees that the first send has a place to put the value and ensures that it will always succeed. Therefore, the first value to arrive will be retrieved regardless of the order of execution.

Caching data in applications

Applications working with data coming from a database (or in general a datastore) will often cache that data in memory because retrieving a value from a database is a costly operation. When the data in the database does not change, there is no problem with costliness. However, for values that can change, we need a mechanism that periodically rereads the value in the database. The cached value, in that case, becomes invalid (it has expired), and we don't want our application to present an old value to the user. This can be solved with a goroutine and a `Ticker` object.

In the last chapter, we studied how to recover from a panic. What if we need to recover when we are using goroutines? See the next lesson to learn how to utilize the functionality of recover along with goroutines.