**1、定时器（time.NewTimer）**

Go语言的定时器实质是单向通道，time.Timer结构体类型中有一个time.Time类型的单向chan，源码（src/time/time.go）如下

type Timer struct {

C <-chan Time

r runtimeTimer

}

初始化 Timer 方法为NewTimer  
示例

package main

import (

"fmt"

"time"

)

func main() {

t := time.NewTimer(time.Second \* 2)

defer t.Stop()

for {

<-t.C

fmt.Println("timer running...")

// 需要重置Reset 使 t 重新开始计时

t.Reset(time.Second \* 2)

}

}

输出  
timer running…  
timer running…  
timer running…  
timer running…  
这里使用NewTimer定时器需要t.Reset重置计数时间才能接着执行。如果注释 t.Reset(time.Second \* 2)会导致通道堵塞，报fatal error: all goroutines are asleep - deadlock!错误。  
同时需要注意 defer t.Stop()在这里并不会停止定时器。这是因为Stop会停止Timer，停止后，Timer不会再被发送，但是Stop不会关闭通道，防止读取通道发生错误。  
如果想停止定时器，只能让go程序自动结束。  
示例  
package main

import (  
“fmt”  
“time”  
)

func main() {

t := time.NewTimer(time.Second \* 2)

ch := make(chan bool)

go func(t \*time.Timer) {

defer t.Stop()

for {

select {

case <-t.C:

fmt.Println("timer running....")

// 需要重置Reset 使 t 重新开始计时

t.Reset(time.Second \* 2)

case stop := <-ch:

if stop {

fmt.Println("timer Stop")

return

}

}

}

}(t)

time.Sleep(10 \* time.Second)

ch <- true

close(ch)

time.Sleep(1 \* time.Second)

**2、定时器（NewTicker）**

package main

import (

"fmt"

"time"

)

func main() {

t := time.NewTicker(time.Second\*2)

defer t.Stop()

for {

<- t.C

fmt.Println("Ticker running...")

}

}

结果  
Ticker running…  
Ticker running…  
Ticker running…  
ticker只要定义完成后，不需要其他操作就可以定时执行。  
这里的defer t.Stop()和上面示例相似，也不会停止定时器，解决办法一样。

package main

import (

"time"

"fmt"

)

func main() {

ticker := time.NewTicker(2 \* time.Second)

ch := make(chan bool)

go func(ticker \*time.Ticker) {

defer ticker.Stop()

for {

select {

case <-ticker.C:

fmt.Println("Ticker running...")

case stop := <-ch:

if stop {

fmt.Println("Ticker Stop")

return

}

}

}

}(ticker)

time.Sleep(10 \* time.Second)

ch <- true

close(ch)

}

**3、time.After**

time.After()表示多长时间长的时候后返回一条time.Time类型的通道消息。但是在取出channel内容之前不阻塞，后续程序可以继续执行。

先看源码（src/time/sleep.go）

func After(d Duration) <-chan Time {

return NewTimer(d).C

}

通过源码我们发现它返回的是一个NewTimer(d).C，其底层是用NewTimer实现的，所以如果考虑到效率低，可以直接自己调用NewTimer。

示例1

package main

import (

"fmt"

"time"

)

func main() {

t := time.After(time.Second \* 3)

fmt.Printf("t type=%T\n", t)

//阻塞3秒

fmt.Println("t=", <-t)

}

运行结果

t type=<-chan time.Time  
t= 2019-05-23 09:58:59.5103274 +0800 CST m=+3.008172101

先打印第一行，3s后打印第二行

基于time.After()特性可以配合select实现计时器

示例2

package main

import (

"fmt"

"time"

)

func main() {

ch1 := make(chan int, 1)

ch1 <- 1

for {

select {

case e1 := <-ch1:

//如果ch1通道成功读取数据，则执行该case处理语句

fmt.Printf("1th case is selected. e1=%v\n", e1)

case <-time.After(time.Second\*2):

fmt.Println("Timed out")

}

}

}

1th case is selected. e1=1  
Timed out  
Timed out  
Timed out  
Timed out

select语句阻塞等待最先返回数据的channel`,如ch1通道成功读取数据，则先输出1th case is selected. e1=1，之后每隔2s输出 Timed out。