Go语言 定时器 Timer 和 Ticker

Go提供了两种定时器， 即 **一次性定时器**， **周期定时器**

- 一次性定时器：定时器只计时一次，结束便停止 Timer
- 周期定时器：定时器周期性进行计时 Ticker

Timer 定时器

Timer实际上是一种单一事件的定时器，即经过指定的时间后触发一个事件，这个事件通过其本身提供的channel进行通知。之所以叫单一事件，是因为Timer只执行一次就结束，这也是Timer与Ticker的最重要的区别之一。

Timer数据结构：

```
// Timer代表一次定时，时间到来后仅发生一个事件。
type Timer struct {
    C <-chan Time
    r runtimeTimer
}
```

Timer对外仅暴露一个channel，指定的时间到来时就往该channel中写入系统时间，也即一个事件。

创建定时器

```
变量 := time.NewTimer(持续时间)
```

持续时间：

Nanosecond Duration = 1 纳秒

Microsecond = 1000 * Nanosecond 微秒

Millisecond = 1000 * Microsecond 毫秒

Second = 1000 * Millisecond 秒

Minute = 60 * Second 分

Hour = 60 * Minute 时

创建Timer意味着把一个计时任务交给系统守护协程，该协程管理着所有的Timer，当Timer的时间到达后向Timer的管道中发送当前的时间作为事件。

time.Now() 为当前时间

示例：

```
timer1 := time.NewTimer(2 * time.Second) //创建定时器2秒
t1 := time.Now() //当前时间
fmt.Printf("t1:%v\n", t1)
t2 := <-timer1.C  //将定时器发送到通道
fmt.Printf("t2:%v\n", t2)
```

验证timer只能响应1次

```
timer2 := time.NewTimer(time.Second)
for {
    <-timer2.C
    fmt.Println("时间到")
}
```

timer实现延时的功能（使用场景）

- 通过time.Sleep实现

```
time.Sleep(持续时间) 如：time.Sleep(2 * time.Second)  //睡眠2秒
```

- 通过创建定时器

```
timer3 := time.NewTimer(2 * time.Second) //通过创建2秒定时器
<-timer3.C
fmt.Println("2秒到")
```

- 通过time.After函数实现

```
<-time.After(2 * time.Second)
fmt.Println("2秒到")
```

停止定时器

Timer创建后可以随时停止，停止计时器的方法是：

```
定时器.Stop()
```

其返回值代表定时器有没有超时：

- true: true: 定时器超时前停止，后续不会再有事件发送；
- false: 定时器超时后停止；

实际上，停止计时器意味着通知系统守护协程移除该定时器。详细的实现原理我们后面单独介绍。

示例：

```
timer4 := time.NewTimer(2 * time.Second)
go func() {
    <-timer4.C
    fmt.Println("定时器执行了")
}()
b := timer4.Stop()
if b {
    fmt.Println("timer4已经关闭")
}
```

重置定时器

已过期的定时器或者已停止的定时器，可以通过重置动作重新激活，重置方法如下：

```
定时器.Reset(持续时间)
```

返回值是bool类型。

重置的动作实质上是先停掉定时器，再启动。其返回值也是掉计时器的返回值。

需要注意的是，**重置定时器虽然可以用于修改还未超时的定时器，但正确的使用方式还是针对已过期的定时器或已被停止的定时器**，同时其返回值也不可靠，返回值存在的价值仅仅是与前面版本兼容。

实际上，重置定时器意味着通知系统守护协程移除该定时器，重新设定时间后，再把定时器交给守护协程。

示例：

```
timer5 := time.NewTimer(3 * time.Second)
timer5.Reset(1 * time.Second)
fmt.Println(time.Now())
fmt.Println(<-timer5.C)
```

After()

有时我们就是想等指定的时间，没有需求提前停止定时器，也没有需求复用该定时器，那么可以使用匿名的定时器。

```
<-time.After(time.Second * 2) // time.After函数的返回值是chan Time
```

时间间隔为2s，实际还是一个定时器，但代码变得更简洁。

示例：

```
func AfterDemo() {
    log.Println(time.Now())
    <- time.After(1 * time.Second)
    log.Println(time.Now())
}
```

延迟执行某个方法（使用场景）

有时我们希望某个方法在今后的某个时刻执行，如下代码所示：

```
func DelayFunction() {
    timer := time.NewTimer(5 * time.Second)

    select {
    case <- timer.C:
        log.Println("Delayed 5s, start to do something.")
        // do something
    }
}
```

DelayFunction()会一直等待timer的事件到来才会执行后面的方法(打印)。

AferFunc()

前面我们例子中讲到延迟一个方法的调用，实际上通过AfterFunc可以更简洁。AfterFunc的原型为：

```
func AfterFunc(d Duration, f func()) *Timer
```

示例：

```
func AfterFuncDemo() {
    log.Println("AfterFuncDemo start: ", time.Now())
    time.AfterFunc(1 * time.Second, func() {
        log.Println("AfterFuncDemo end: ", time.Now())
    })

    time.Sleep(2 * time.Second) // 等待协程退出
}
```

AfterFuncDemo()中先打印一个时间，然后使用AfterFunc启动一个定器，并指定定时器结束时执行一个方法打印结束时间。

与上面的例子所不同的是，`time.AfterFunc()`**是异步执行的**，所以需要在函数最后sleep等待指定的协程退出，否则可能函数结束时协程还未执行。

Ticker 定时器

Ticker是周期性定时器，即周期性的触发一个事件，通过Ticker本身提供的管道将事件传递出去。

Ticker的数据结构与Timer完全一样：

```
type Ticker struct {
    C <- chan Time
    r runtimeTimer
}
```

Ticker对外仅暴露一个channel，指定的时间到来时就往该channel中写入系统时间，也即一个事件。

在创建Ticker时会指定一个时间，作为事件触发的周期。这也是Ticker与Timer的最主要的区别。

另外，ticker的英文原意是钟表的"滴哒"声，钟表周期性的产生"滴哒"声，也即周期性的产生事件。

创建定时器

使用NewTicker()方法就可以创建一个周期性定时器，函数原型如下

```
func NewTicker(d Duration) *Ticker
```

其中参数`d`即为定时器时间触发的周期。

示例：

```
ticker := time.NewTicker(1 * time.Second)
```

停止定时器

使用定时器对外暴露的 Stop 方法就可以停掉一个周期性定时器， 函数原型如下

```
func (t *Ticker)Stop()
```

需要注意的是， 该方法会停止计时， 意味著不会向定时器的管道中写入事件，但管道并不会被关闭。管道在使用完成后，生命周期结束后会自动释放。

Ticker在使用完后务必要释放，否则会产生资源泄露，进而会持续消耗CPU资源，最后会把CPU耗尽。

示例：

```
ticker := time.NewTicker(1 * time.Second)
defer ticker.Stop()
```

简单接口

部分场景下，启动一个定时器并且永远不会停止， 比如定时轮询任务， 此时可以使用一个简单的Tick函数来获取定时器的管道， 函数原型如下：

```
func Tick(d Duration) <-chan Time
```

这个函数内部实际还是创建一个Ticker，但并不会返回出来，所以没有手段来停止该Ticker。所以，一定要考虑具体的使用场景。

示例：

```
for tick := range time.Tick(3 * time.Second) {
    fmt.Println(tick)
}
```

使用场景

简单定时任务

有时，我们希望定时执行一个任务，这时就可以使用ticker来完成。
下面代码演示，每隔1s记录一次日志：

```
// TickerDemo 用于演示ticker基础用法
func TickerDemo() {
    ticker := time.NewTicker(1 * time.Second)
    defer ticker.Stop()

    for range ticker.C {
        log.Println("Ticker tick.")
    }
}
```

`for range ticker.C`会持续从管道中获取事件，收到事件后打印一行日志，如果管道中没有数据会阻塞等待事件，由于ticker会周期性的向管道中写入事件，所以上述程序会周期性的打印日志。

定时聚合任务

有时，我们希望把一些任务打包进行批量处理。比如，公交车发车场景：

- 公交车每隔5分钟发一班，不管是否已坐满乘客；
- 已经坐满乘客情况下，不足五分钟也发车

```
// TickerLaunch用于演示ticker聚合任务用法
func TickerLaunch() {
    ticker := time.NewTicker(5 * time.Minute)
    maxPassenger := 30                    // 每车最大装载人数
    passengers := make([]string, 0, maxPassenger)

    for {
        passenger := GetNewPassenger() // 获取一个新乘客
        if passenger != "" {
            passengers = append(passengers, passenger)
        } else {
            time.Sleep(1 * time.Second)
        }

        select {
        case <- ticker.C:                // 时间到，发车
            Launch(passengers)
            passengers = []string{}
        default:
            if len(passengers) >= maxPassenger {  // 时间没到，车已座满，发车
                Launch(passengers)
                passengers = []string{}
            }
        }
    }
}
```

上面代码中for循环负责接待乘客上车，并决定是否要发车。每当乘客上车，select语句会先判断ticker.C中是否有数据，有数据则代表发车时间已到，如果没有数据，则判断车是否已坐满，坐满后仍然发车。