

1、基本介绍

time包提供了时间的显示和测量用的函数。日历的计算采用的是公历。

主要有type Weekday、Month、Location、Time、Duration、Timer、Ticker几个类型，以及一些相关的函数和方法。

2、Weekday

type Weekday

```
type Weekday int
```

Weekday代表一周的某一天。

```
const (  
    Sunday Weekday = iota  
    Monday  
    Tuesday  
    Wednesday  
    Thursday  
    Friday  
    Saturday  
)
```

0
1
2

func (Weekday) String

```
func (d Weekday) String() string
```

String返回该日（周几）的英文名（"Sunday"、"Monday"，.....）

CSDN @Golang-Study

```
3 import (  
4     "fmt"  
5     "time"  
6 )  
7  
8 func main() {  
9  
10    // 如果不赋值，默认time.Sunday  
11    var day time.Weekday = time.Thursday  
12    daystr := day.String()  
13    fmt.Println(daystr)  
14 }  
15
```

问题 1 输出 调试控制台 终端 COMMENTS

```
PS F:\tools\golang\timee> go run main.go  
Sunday  
PS F:\tools\golang\timee> go run main.go  
Thursday  
PS F:\tools\golang\timee> |
```

CSDN @Golang-Study

3、Month

type Month

```
type Month int
```

Month代表一年的某个月。

```
const (  
    January Month = 1 + iota  
    February  
    March  
    April  
    May  
    June  
    July  
    August  
    September  
    October  
    November  
    December  
)
```

Example

func (Month) String

```
func (m Month) String() string
```

String返回月份的英文名 ("January", "February",)

CSDN @Golang-Study

4、时间间隔

- type Duration
 - func ParseDuration(s string) (Duration, error)
 - func Since(t Time) Duration
 - func (d Duration) Hours() float64
 - func (d Duration) Minutes() float64
 - func (d Duration) Seconds() float64
 - func (d Duration) Nanoseconds() int64
 - func (d Duration) String() string

CSDN @Golang-Study

type Duration

```
type Duration int64
```

Duration类型代表两个时间点之间经过的时间，以纳秒为单位。可表示的最长时间段大约290年。

```
const (  
    Nanosecond Duration = 1  
    Microsecond      = 1000 * Nanosecond  
    Millisecond       = 1000 * Microsecond  
    Second           = 1000 * Millisecond  
    Minute           = 60 * Second  
    Hour             = 60 * Minute  
)
```

常用的时间段。没有定义一天或超过一天的单元，以避免夏令制的时区切换的混乱。

要将Duration类型值表示为某时间单元的个数，用除法：

```
second := time.Second  
fmt.Print(int64(second/time.Millisecond)) // prints 1000
```

要将整数个某时间单元表示为Duration类型值，用乘法：

```
seconds := 10  
fmt.Print(time.Duration(seconds)*time.Second) // prints 10s
```

CSDN @Golang-Study

4.1 ParseDuration函数

func ParseDuration

```
func ParseDuration(s string) (Duration, error)
```

ParseDuration解析一个时间段字符串。一个时间段字符串是一个序列，每个片段包含可选的正负号、十进制数、可选的小数部分和单位后缀，如"300ms"、"-1.5h"、"2h45m"。合法的单位有"ns"、"us"、"µs"、"ms"、"s"、"m"、"h"。

```
func main() {  
  
    dura, _ := time.ParseDuration("10h")  
    fmt.Println(dura) // 10h0m0s  
  
    dura, _ = time.ParseDuration("10ns")  
    fmt.Println(dura) // 10ns  
  
    dura, _ = time.ParseDuration("-10ns")  
    fmt.Println(dura) // -10ns  
  
    var err error  
    dura, err = time.ParseDuration("10uss")  
    if err != nil {  
        fmt.Println(err) // time: unknown unit "uss" in duration "10uss"  
    }  
}
```

CSDN @Golang-Study

4.2 Since函数

func Since

```
func Since(t Time) Duration
```

Since返回从t到现在经过的时间，等价于time.Now().Sub(t)。

```
func main() {  
  
    before := time.Date(2020, 20, 20, 12, 0, 0, 0, time.UTC)  
  
    dura := time.Since(before)  
  
    fmt.Println(dura) // 8904h35m53.4526059s  
}
```

CSDN @Golang-Study

4.3 相关方法

func (Duration) Hours

```
func (d Duration) Hours() float64
```

Hours将时间段表示为float64类型的小时数。

func (Duration) Minutes

```
func (d Duration) Minutes() float64
```

Hours将时间段表示为float64类型的分钟数。

func (Duration) Seconds

```
func (d Duration) Seconds() float64
```

Hours将时间段表示为float64类型的秒数。

func (Duration) Nanoseconds

```
func (d Duration) Nanoseconds() int64
```

Hours将时间段表示为int64类型的纳秒数，等价于int64(d)。

func (Duration) String

```
func (d Duration) String() string
```

返回时间段采用"72h3m0.5s"格式的字符串表示。最前面可以有符号，数字+单位为一个单元，开始部分的0值单元会被省略；如果时间段<1s，会使用"ms"、"us"、"ns"来保证第一个单元的数字不是0；如果时间段为0，会返回"0"。

```
before := time.Date(2020, 20, 20, 12, 0, 0, 0, time.UTC)

dura := time.Since(before)

fmt.Println(dura) // 8904h35m53.4526059s

fmt.Println(dura.Hours()) // 8904.675361345333
fmt.Println(dura.Minutes()) // 534280.52168072
fmt.Println(dura.Seconds()) // 3.20568313008432e+07
fmt.Println(dura.Nanoseconds()) // 32056881122408000
fmt.Println(dura.Microseconds()) // 32056911580855
fmt.Println(dura.String()) // 8904h42m13.0737279s
```

CSDN @Golang-Study

5、Time

type Time

```
type Time struct {
    // 内含隐藏或非导出字段
}
```

Time代表一个纳秒精度的时间点。

程序中应使用Time类型值来保存和传递时间，而不能用指针。就是说，表示时间的变量和字段，应为time.Time类型，而不是*time.Time类型。一个Time类型值可以被多个go程同时使用。时间点可以使用Before、After和Equal方法进行比较。Sub方法让两个时间点相减，生成一个Duration类型值（代表时间段）。Add方法给一个时间点加上一个时间段，生成一个新的Time类型时间点。

Time零值代表时间点January 1, year 1, 00:00:00.000000000 UTC。因为本时间点一般不会出现在使用中，IsZero方法提供了检验时间是否显式初始化的一个简单途径。

每一个时间都具有一个地点信息（及对应地点的时区信息），当计算时间的表示格式时，如Format、Hour和Year等方法，都会考虑该信息。Local、UTC和In方法返回一个指定时区（但指向同一时间点）的Time。修改地点/时区信息只是会改变其表示；不会修改被表示的时间点，因此也不会影响其计算。

CSDN @Golang-Study

4.1 返回值为Time的函数

```
func Date(year int, month Month, day, hour, min, sec, nsec int, loc *Location) Time
```

Date返回一个时区为loc、当地时间为：

```
year-month-day hour:min:sec + nsec nanoseconds
```

的时间点。

month、day、hour、min、sec和nsec的值可能会超出它们的正常范围，在转换前函数会自动将之规范化。如October 32被修正为November 1。

夏时制的时区切换会跳过或重复时间。如，在美国，March 13, 2011 2:15am从来不会出现，而November 6, 2011 1:15am 会出现两次。此时，时区的选择和时间是没有良好定义的。Date会返回在时区切换的两个时区其中一个时区

正确的时间，但本函数不会保证在哪一个时区正确。

如果loc为nil会panic。

Example

```
t := time.Date(2009, time.November, 10, 23, 0, 0, 0, time.UTC)
fmt.Printf("Go launched at %s\n", t.Local())
```

Output:

```
Go launched at 2009-11-10 15:00:00 -0800 PST
```

CSDN @Golang-Study

func Now

```
func Now() Time
```

Now返回当前本地时间。

CSDN @Golang-Study

func Unix

```
func Unix(sec int64, nsec int64) Time
```

Unix创建一个本地时间，对应sec和nsec表示的Unix时间（从January 1, 1970 UTC至该时间的秒数和纳秒数）。

nsec的值在[0, 999999999]范围外是合法的。

CSDN @Golang-Study

4.2 时间戳

func (Time) Unix

```
func (t Time) Unix() int64
```

Unix将t表示为Unix时间，即从时间点January 1, 1970 UTC到时间点t所经过的时间（单位秒）。

func (Time) UnixNano

```
func (t Time) UnixNano() int64
```

UnixNano将t表示为Unix时间，即从时间点January 1, 1970 UTC到时间点t所经过的时间（单位纳秒）。如果纳秒为单位的unix时间超出了int64能表示的范围，结果是未定义的。注意这就意味着Time零值调用UnixNano方法的话，结果是未定义的。

CSDN @Golang-Study

```
8 func main() {
9
10     now := time.Now() // 获取当前时间
11     timestamp := now.Unix() // 秒级时间戳
12     milli := now.UnixMilli() // 毫秒时间戳 Go1.17+
13     micro := now.UnixMicro() // 微秒时间戳 Go1.17+
14     nano := now.UnixNano() // 纳秒时间戳
15     fmt.Println(timestamp, milli, micro, nano)
16 }
17
```

问题 1 输出 调试控制台 终端 COMMENTS

```
PS F:\tools\golang\timex> go run .\main.go
1661515590 1661515590258 1661515590258178 1661515590258178700
PS F:\tools\golang\timex>
```

CSDN @Golang-Study

4.3 时间戳转换为时间对象

```
func main() {
    // 获取北京时间所在的东八区时区对象
    secondsEastOfUTC := int((8 * time.Hour).Seconds())
    beijing := time.FixedZone("Beijing Time", secondsEastOfUTC)

    // 北京时间 2022-02-22 22:22:22.000000022 +0800 CST
    t := time.Date(2022, 02, 22, 22, 22, 22, 22, beijing)

    var (
        sec = t.Unix()
        msec = t.UnixMilli()
        usec = t.UnixMicro()
    )

    // 将秒级时间戳转为时间对象（第二个参数为不足1秒的纳秒数）
    timeObj := time.Unix(sec, 22)
    fmt.Println(timeObj) // 2022-02-22 22:22:22.000000022 +0800 CST
    timeObj = time.UnixMilli(msec) // 毫秒级时间戳转为时间对象
    fmt.Println(timeObj) // 2022-02-22 22:22:22 +0800 CST
    timeObj = time.UnixMicro(usec) // 微秒级时间戳转为时间对象
    fmt.Println(timeObj) // 2022-02-22 22:22:22 +0800 CST
}
```

CSDN @Golang-Study

4.4 获取Time对象的年月日时分秒等

func (Time) Year

```
func (t Time) Year() int
```

返回时间点t对应的年份。

func (Time) Month

```
func (t Time) Month() Month
```

返回时间点t对应那一年的第几个月。

func (Time) Day

```
func (t Time) Day() int
```

返回时间点t对应那一月的第几日。

func (Time) Weekday

```
func (t Time) Weekday() Weekday
```

返回时间点t对应的那一周的周几。

func (Time) Hour

```
func (t Time) Hour() int
```

返回t对应的那一天的第几小时，范围[0, 23]。

func (Time) Minute

```
func (t Time) Minute() int
```

返回t对应的那一小时的第几分钟，范围[0, 59]。

func (Time) Second

```
func (t Time) Second() int
```

返回t对应的那一分钟的第几秒，范围[0, 59]。

func (Time) Nanosecond

```
func (t Time) Nanosecond() int
```

返回t对应的那一秒内的纳秒偏移量，范围[0, 999999999]。

CSDN @Golang-Study

func (Time) Date

```
func (t Time) Date() (year int, month Month, day int)
```

返回时间点t对应的年、月、日。

func (Time) Clock

```
func (t Time) Clock() (hour, min, sec int)
```

返回t对应的那一天的时、分、秒。

CSDN @Golang-Study


```

8 func main() {
9
10     now := time.Now() // 获取当前时间
11     fmt.Printf("current time:%v\n", now)
12
13     year := now.Year()      // 年
14     month := now.Month()   // 月
15     day := now.Day()       // 日
16     hour := now.Hour()     // 小时
17     minute := now.Minute() // 分钟
18     second := now.Second()  // 秒
19     yearDay := now.YearDay() // 一年中的第几天
20     fmt.Println(year, month, day, hour, minute, second, yearDay)
21 }
22

```

问题 1 输出 调试控制台 终端 COMMENTS

```

PS F:\tools\golang\timee> go run .\main.go
current time:2022-08-26 20:13:59.9430263 +0800 CST m=+0.002119901
2022 August 26 20 13 59
PS F:\tools\golang\timee> go run .\main.go
current time:2022-08-26 20:20:48.9112255 +0800 CST m=+0.001815701
2022 August 26 20 20 48 238
PS F:\tools\golang\timee>

```

CSDN @Golang-Study

4.5 时间操作方法

func (Time) Add

```
func (t Time) Add(d Duration) Time
```

Add返回时间点t+d。

```
func main() {  
    now := time.Now()  
    later := now.Add(time.Hour) // 当前时间加1小时后的时间  
    fmt.Println(later)  
}
```

CSDN @Golang-Study

func (Time) Sub

```
func (t Time) Sub(u Time) Duration
```

返回一个时间段t-u。如果结果超出了Duration可以表示的最大值/最小值，将返回最大值/最小值。要获取时间点t-d（d为Duration），可以使用t.Add(-d）。

CSDN @Golang-Study

Equal

```
1 | func (t Time) Equal(u Time) bool
```

判断两个时间是否相同，会考虑时区的影响，因此不同时区标准的时间也可以正确比较。本方法和用t==u不同，这种方法还会比较地点和时区信息。

Before

```
1 | func (t Time) Before(u Time) bool
```

如果t代表的时间点在u之前，返回真；否则返回假。

After

```
1 | func (t Time) After(u Time) bool
```

如果t代表的时间点在u之后，返回真；否则返回假。

CSDN @Golang-Study

4.6 时间格式化成字符串

func (Time) Format

```
func (t Time) Format(layout string) string
```

Format根据layout指定的格式返回t代表的时间点的格式化文本表示。layout定义了参考时间：

```
Mon Jan 2 15:04:05 -0700 MST 2006
```

格式化后的字符串表示，它作为期望输出的例子。同样的格式规则会被用于格式化时间。

预定义的ANSIC、UnixDate、RFC3339和其他版式描述了参考时间的标准或便捷表示。要获得更多参考时间的定义和格式，参见本包的ANSIC和其他版式常量。

Example

```
// layout shows by example how the reference time should be represented.
const layout = "Jan 2, 2006 at 3:04pm (MST)"
t := time.Date(2009, time.November, 10, 15, 0, 0, 0, time.Local)
fmt.Println(t.Format(layout))
fmt.Println(t.UTC().Format(layout))
```

Output:

```
Nov 10, 2009 at 3:00pm (PST)
Nov 10, 2009 at 11:00pm (UTC)
```

CSDN @Golang-Study

```
func main() {
    now := time.Now()
    // 格式化的模板为 2006-01-02 15:04:05

    // 24小时制
    fmt.Println(now.Format("2006-01-02 15:04:05.000 Mon Jan")) // 2022-08-26 21:10:34.510 Fri Aug
    // 12小时制
    fmt.Println(now.Format("2006-01-02 03:04:05.000 PM Mon Jan")) // 2022-08-26 09:10:34.510 PM Fri Aug

    // 小数点后写0，因为有3个0所以格式化输出的结果也保留3位小数
    fmt.Println(now.Format("2006/01/02 15:04:05.000")) // 2022/08/26 21:10:34.510
    // 小数点后写9，会省略末尾可能出现的0
    fmt.Println(now.Format("2006/01/02 15:04:05.999")) // 2022/08/26 21:10:34.51

    // 只格式化时分秒部分
    fmt.Println(now.Format("15:04:05")) // 21:10:34
    // 只格式化日期部分
    fmt.Println(now.Format("2006.01.02")) // 2022.08.26
}
```

CSDN @Golang-Study

4.7 字符串转时间对象

对于从文本的时间表示中解析出时间对象，time包中提供了 `time.Parse` 和 `time.ParseInLocation` 两个函数。

其中 `time.Parse` 在解析时不需要额外指定时区信息。

```
1 // parseDemo 指定时区解析时间
2 func parseDemo() {
3     // 在有时区指示符的情况下，time.Parse 返回UTC时间
4     timeObj, err := time.Parse("2006/01/02 15:04:05", "2022/10/05 11:25:20")
5     if err != nil {
6         fmt.Println(err)
7         return
8     }
9     fmt.Println(timeObj) // 2022-10-05 11:25:20 +0000 UTC
10
11    // 在有时区指示符的情况下，time.Parse 返回对应时区的时间表示
12    // RFC3339      = "2006-01-02T15:04:05Z07:00"
13    timeObj, err = time.Parse(time.RFC3339, "2022-10-05T11:25:20+08:00")
14    if err != nil {
15        fmt.Println(err)
16        return
17    }
18    fmt.Println(timeObj) // 2022-10-05 11:25:20 +0800 CST
19 }
```

CSDN @Golang-Study

`time.ParseInLocation` 函数需要在解析时额外指定时区信息。

```
1 // parseDemo 解析时间
2 func parseDemo() {
3     now := time.Now()
4     fmt.Println(now)
5     // 加载时区
6     loc, err := time.LoadLocation("Asia/Shanghai")
7     if err != nil {
8         fmt.Println(err)
9         return
10    }
11    // 按照指定时区和指定格式解析字符串时间
12    timeObj, err := time.ParseInLocation("2006/01/02 15:04:05", "2022/10/05 11:25:20", loc)
13    if err != nil {
14        fmt.Println(err)
15        return
16    }
17    fmt.Println(timeObj)
18    fmt.Println(timeObj.Sub(now))
19 }
```

CSDN @Golang-Study

6、Ticker计时器

type Ticker

```
type Ticker struct {  
    C <-chan Time // 周期性传递时间信息的通道  
    // 内含隐藏或非导出字段  
}
```

Ticker保管一个通道，并每隔一段时间向其传递"tick"。

func NewTicker

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

NewTicker返回一个新的Ticker，该Ticker包含一个通道字段，并会每隔时间段d就向该通道发送当时的时间。它会调整时间间隔或者丢弃tick信息以适应反应慢的接收者。如果d<=0会panic。关闭该Ticker可以释放相关资源。

func (*Ticker) Stop

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

Stop关闭一个Ticker。在关闭后，将不会发送更多的tick信息。Stop不会关闭通道t.C，以避免从该通道的读取不正确的成功。

CSDN @Golang-Study

```
func main() {  
    // 计时器的使用  
    ticker := time.NewTicker(time.Second * 5)  
  
    for {  
        select {  
        case <-ticker.C: // ticker每隔5秒就会往ticker的信道C中写入当时的时间，这里可以取出，如果取到了就说明有一次计时到了  
            fmt.Println("上报服务器数据...")  
        default:  
            fmt.Println("服务器运行中...")  
            time.Sleep(time.Second * 2)  
        }  
    }  
}
```

CSDN @Golang-Study

7、Timer定时器

type Timer

```
type Timer struct {  
    C <-chan Time  
    // 内含隐藏或非导出字段  
}
```

Timer类型代表单次时间事件。当Timer到期时，当时的时间会被发送给C，除非Timer是被AfterFunc函数创建的。

func NewTimer

```
func NewTimer(d Duration) *Timer
```

NewTimer创建一个Timer，它会在最少过去时间段d后到期，向其自身的C字段发送当时的时间。

func AfterFunc

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

AfterFunc另起一个go程等待时间段d过去，然后调用f。它返回一个Timer，可以通过调用其Stop方法来取消等待和对f的调用。

func (*Timer) Reset

```
func (t *Timer) Reset(d Duration) bool
```

Reset使t重新开始计时，（本方法返回后再）等待时间段d过去后到期。如果调用时t还在等待中会返回真；如果t已经到期或者被停止了会返回假。

func (*Timer) Stop

```
func (t *Timer) Stop() bool
```

Stop停止Timer的执行。如果停止了t会返回真；如果t已经被停止或者过期了会返回假。Stop不会关闭通道t.C，以避免从该通道的读取不正确的成功。

CSDN @Golang-Study

```
func main() {  
    // 计时器的使用  
    ticker := time.NewTicker(time.Second * 5)  
  
    // 定时器的使用  
    timer := time.NewTimer(time.Second * 3)  
  
    for {  
        select {  
        case <-ticker.C: // ticker每隔5秒就会往ticker的信道C中写入当时的时间，这里可以取出，如果取到了就说明有一次计时到了  
            fmt.Println("上报服务器数据...计时器一般用于实现周期性的任务")  
        case <-timer.C:  
            fmt.Println("定时器timer只会执行一次...定时器一般用于延迟任务")  
        default:  
            fmt.Println("服务器运行中...")  
            time.Sleep(time.Second * 2)  
        }  
    }  
}
```

CSDN @Golang-Study

8、Sleep睡眠函数

func Sleep

```
func Sleep(d Duration)
```

Sleep阻塞当前go程至少d代表的时间段。d<=0时，Sleep会立刻返回。

Example

```
time.Sleep(100 * time.Millisecond) 睡眠100毫秒
```

CSDN @Golang-Study

9、After函数

func After

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

After会在另一线程经过时间段d后向返回值发送当时的时间。等价于NewTimer(d).C。CSDN @Golang-Study

```
func main() {  
    // 计时器的使用  
    ticker := time.NewTicker(time.Second * 5)  
  
    // 定时器的使用  
    timer := time.NewTimer(time.Second * 3)  
  
    // After函数类似于NewTimer  
    timeout := time.After(time.Second * 8)  
    for {  
        select {  
        case <-ticker.C: // ticker每隔5秒就会往ticker的信道C中写入当时的时间，这里可以取出，如果取到了就说明有一次计时到了  
            fmt.Println(a...: "上报服务器数据...计时器一般用于实现周期性的任务")  
        case <-timer.C:  
            fmt.Println(a...: "定时器timer只会执行一次...定时器一般用于延迟任务")  
        case <-(timeout):  
            fmt.Println(a...: "After函数.....")  
        default:  
            fmt.Println(a...: "服务器运行中...")  
            time.Sleep(time.Second * 2)  
        }  
    }  
}
```

8秒后执行，且只执行一次

CSDN @Golang-Study

10、Tick函数

func Tick

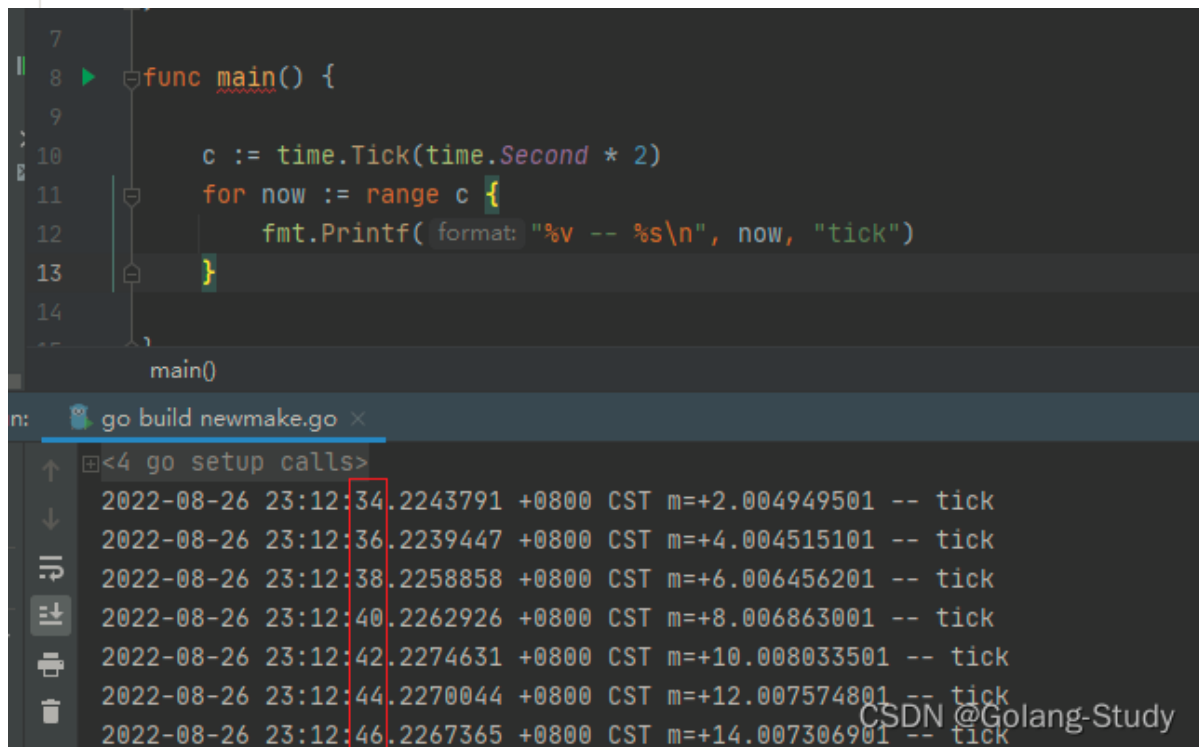
```
func Tick(d Duration) <-chan Time
```

Tick是NewTicker的封装，只提供对Ticker的通道的访问。如果不需要关闭Ticker，本函数就很方便。

Example

```
c := time.Tick(1 * time.Minute)
for now := range c {
    fmt.Printf("%v %s\n", now, statusUpdate())
}
```

CSDN @Golang-Study



```
7
8 func main() {
9
10     c := time.Tick(time.Second * 2)
11     for now := range c {
12         fmt.Printf("format: \"%v -- %s\n\", now, \"tick\")
13     }
14
15     main()
16 }
```

Terminal Output:

```
<4 go setup calls>
2022-08-26 23:12:34.2243791 +0800 CST m=+2.004949501 -- tick
2022-08-26 23:12:36.2239447 +0800 CST m=+4.004515101 -- tick
2022-08-26 23:12:38.2258858 +0800 CST m=+6.006456201 -- tick
2022-08-26 23:12:40.2262926 +0800 CST m=+8.006863001 -- tick
2022-08-26 23:12:42.2274631 +0800 CST m=+10.008033501 -- tick
2022-08-26 23:12:44.2270044 +0800 CST m=+12.007574801 -- tick
2022-08-26 23:12:46.2267365 +0800 CST m=+14.007306901 -- tick
```

CSDN @Golang-Study