# 1. 主要类型概述

time 包提供了时间的显示和计量用的功能。日历的计算采用的是公历。提供的主要类型如下：

## 1.1. Location

代表一个地区，并表示该地区所在的时区（可能多个）。Location 通常代表地理位置的偏移，比如 CEST 和 CET 表示中欧。下一节将详细讲解 Location。

## 1.2. Time

代表一个纳秒精度的时间点，是公历时间。后面会详细介绍。

## 1.3. Duration

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，也就是说如果两个时间点相差超过 290 年，会返回 290 年，也就是 minDuration(-1 << 63) 或 maxDuration(1 << 63 - 1)。

类型定义：type Duration int64。

将 Duration 类型直接输出时，因为实现了 fmt.Stringer 接口，会输出人类友好的可读形式，如：72h3m0.5s。

## 1.4. Timer 和 Ticker

这是定时器相关类型。本章最后会讨论定时器。

## 1.5. Weekday 和 Month

这两个类型的原始类型都是 int，定义它们，语义更明确，同时，实现 fmt.Stringer 接口，方便输出。

# 2. 时区

不同国家（有时甚至是同一个国家内的不同地区）使用不同的时区。对于要输入和输出时间的程序来说，必须对系统所处的时区加以考虑。Go 语言使用 Location 来表示地区相关的时区，一个 Location 可能表示多个时区。

time 包提供了 Location 的两个实例：Local 和 UTC。Local 代表当前系统本地时区；UTC 代表通用协调时间，也就是零时区。time 包默认（为显示提供时区）使用 UTC 时区。

## 2.1. Local 是如何做到表示本地时区的？

时区信息既浩繁又多变，Unix 系统以标准格式存于文件中，这些文件位于 /usr/share/zoneinfo，而本地时区可以通过 /etc/localtime 获取，这是一个符号链接，指向 /usr/share/zoneinfo 中某一个时区。比如我本地电脑指向的是：/usr/share/zoneinfo/Asia/Shanghai。

因此，在初始化 Local 时，通过读取 /etc/localtime 可以获取到系统本地时区。

当然，如果设置了环境变量 TZ，则会优先使用它。

相关代码：

tz, ok := syscall.Getenv("TZ")  
switch {  
case !ok:  
 z, err := loadLocation("localtime", []string{"/etc"})  
 if err == nil {  
 localLoc = \*z  
 localLoc.name = "Local"  
 return  
 }  
case tz != "":  
 if tz[0] == ':' {  
 tz = tz[1:]  
 }  
 if tz != "" && tz[0] == '/' {  
 if z, err := loadLocation(tz, []string{""}); err == nil {  
 localLoc = \*z  
 if tz == "/etc/localtime" {  
 localLoc.name = "Local"  
 } else {  
 localLoc.name = tz  
 }  
 return  
 }  
 } else if tz != "" && tz != "UTC" {  
 if z, err := loadLocation(tz, zoneSources); err == nil {  
 localLoc = \*z  
 return  
 }  
 }  
}  
  
// Fall back to UTC.  
localLoc.name = "UTC"

## 2.2. 获得特定时区的实例

函数 LoadLocation 可以根据名称获取特定时区的实例。函数声明如下：

func LoadLocation(name string) (\*Location, error)

如果 name 是 "" 或 "UTC"，返回 UTC；如果 name 是 "Local"，返回 Local；否则 name 应该是 IANA 时区数据库里有记录的地点名（该数据库记录了地点和对应的时区），如 "America/New\_York"。

LoadLocation 函数需要的时区数据库可能不是所有系统都提供，特别是非 Unix 系统。此时 LoadLocation 会查找环境变量 ZONEINFO 指定目录或解压该变量指定的 zip 文件（如果有该环境变量）；然后查找 Unix 系统的惯例时区数据安装位置，最后查找 $GOROOT/lib/time/zoneinfo.zip。

可以在 Unix 系统下的 /usr/share/zoneinfo 中找到所有的名称。

## 2.3. 总结

通常，我们使用 time.Local 即可，偶尔可能会需要使用 UTC。在解析时间时，心中一定记得有时区这么回事。当你发现时间出现莫名的情况时，很可能是因为时区的问题，特别是当时间相差 8 小时时。

# 3. Time 类型详解

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，而不是指针。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 协程同时使用。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是显式初始化的一个简单途径。

每一个 Time 都具有一个地点信息（即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。

通过 == 比较 Time 时，Location 信息也会参与比较，因此 Time 不应该作为 map 的 key。

## 3.1. Time 的内部结构

type Time struct {  
 // wall and ext encode the wall time seconds, wall time nanoseconds,  
 // and optional monotonic clock reading in nanoseconds.  
 //  
 // From high to low bit position, wall encodes a 1-bit flag (hasMonotonic),  
 // a 33-bit seconds field, and a 30-bit wall time nanoseconds field.  
 // The nanoseconds field is in the range [0, 999999999].  
 // If the hasMonotonic bit is 0, then the 33-bit field must be zero  
 // and the full signed 64-bit wall seconds since Jan 1 year 1 is stored in ext.  
 // If the hasMonotonic bit is 1, then the 33-bit field holds a 33-bit  
 // unsigned wall seconds since Jan 1 year 1885, and ext holds a  
 // signed 64-bit monotonic clock reading, nanoseconds since process start.  
 wall uint64  
 ext int64  
  
 // loc specifies the Location that should be used to  
 // determine the minute, hour, month, day, and year  
 // that correspond to this Time.  
 // The nil location means UTC.  
 // All UTC times are represented with loc==nil, never loc==&utcLoc.  
 loc \*Location  
}

要讲解 time.Time 的内部结构，得先看 time.Now() 函数。

// Now returns the current local time.  
func Now() Time {  
 sec, nsec, mono := now()  
 mono -= startNano  
 sec += *unixToInternal* - *minWall* if uint64(sec)>>33 != 0 {  
 return Time{uint64(nsec), sec + *minWall*, Local}  
 }  
 return Time{*hasMonotonic* | uint64(sec)<<*nsecShift* | uint64(nsec), mono, Local}  
}

now() 的具体实现在 runtime 包中，以 linux/amd64 为例，在 sys\_linux\_amd64.s 中的 time · now，这是汇编实现的：

* 调用系统调用 clock\_gettime 获取时钟值（这是 POSIX 时钟）。其中 clockid\_t 时钟类型是 CLOCK\_REALTIME，也就是可设定的系统级实时时钟。得到的是 struct timespec 类型。（可以到纳秒）
* 如果 clock\_gettime 不存在，则使用精度差些的系统调用 gettimeofday。得到的是 struct timeval 类型。（最多到微秒）

*注意：* 这里使用了 Linux 的 vdso 特性，不了解的，可以查阅相关知识。

虽然 timespec 和 timeval 不一样，但结构类似。因为 now() 函数返回两个值：sec( 秒 ) 和 nsec( 纳秒 )，所以，time · now 的实现将这两个结构转为需要的返回值。需要注意的是，Linux 系统调用返回的 sec( 秒 ) 是 Unix 时间戳，也就是从 1970-1-1 算起的。

回到 time.Now() 的实现，现在我们得到了 sec 和 nsec，从 Time{sec + unixToInternal, nsec, Local} 这句可以看出，Time 结构的 sec 并非 Unix 时间戳，实际上，加上的 unixToInternal 是 1-1-1 到 1970-1-1 经历的秒数。也就是 Time 中的 sec 是从 1-1-1 算起的秒数，而不是 Unix 时间戳。

Time 的最后一个字段表示地点时区信息。本章后面会专门介绍。

## 3.2. 常用函数或方法

Time 相关的函数和方法较多，有些很容易理解，不赘述，查文档即可。

### 3.2.1. 零值的判断

因为 Time 的零值是 sec 和 nsec 都是 0，表示 1 年 1 月 1 日。

Time.IsZero() 函数用于判断 Time 表示的时间是否是 0 值。

### 3.2.2. 与 Unix 时间戳的转换

相关函数或方法：

* time.Unix(sec, nsec int64) 通过 Unix 时间戳生成 time.Time 实例；
* time.Time.Unix() 得到 Unix 时间戳；
* time.Time.UnixNano() 得到 Unix 时间戳的纳秒表示；

### 3.2.3. 格式化和解析

这是实际开发中常用到的。

* time.Parse 和 time.ParseInLocation
* time.Time.Format

#### 解析

对于解析，要特别注意时区问题，否则很容易出 bug。比如：

t, \_ := time.Parse("2006-01-02 15:04:05", "2016-06-13 09:14:00")

fmt.Println(time.Now().Sub(t).Hours())

2016-06-13 09:14:00 这个时间可能是参数传递过来的。这段代码的结果跟预期的不一样。

原因是 time.Now() 的时区是 time.Local，而 time.Parse 解析出来的时区却是 time.UTC（可以通过 Time.Location() 函数知道是哪个时区）。在中国，它们相差 8 小时。

所以，一般的，我们应该总是使用 time.ParseInLocation 来解析时间，并给第三个参数传递 time.Local。

#### 为什么是 2006-01-02 15:04:05

可能你已经注意到：2006-01-02 15:04:05 这个字符串了。没错，这是固定写法，类似于其他语言中 Y-m-d H:i:s 等。为什么采用这种形式？又为什么是这个时间点而不是其他时间点？

* 官方说，使用具体的时间，比使用 Y-m-d H:i:s 更容易理解和记忆；这么一说还真是 ~
* 而选择这个时间点，也是出于好记的考虑，官方的例子：Mon Jan 2 15:04:05 MST 2006，另一种形式 01/02 03:04:05PM '06 -0700，对应是 1、2、3、4、5、6、7；常见的格式：2006-01-02 15:04:05，很好记：2006 年 1 月 2 日 3 点 4 分 5 秒 ~

*如果你是 PHPer，喜欢 PHP 的格式，可以试试 [times](https://github.com/polaris1119/times" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter04/_blank) 这个包。*

#### 格式化

时间格式化输出，使用 Format 方法，layout 参数和 Parse 的一样。Time.String() 方法使用了 2006-01-02 15:04:05.999999999 -0700 MST 这种 layout。

### 3.2.4. 实现 序列化 / 反序列化 相关接口

Time 实现了 encoding 包中的 BinaryMarshaler、BinaryUnmarshaler、TextMarshaler 和 TextUnmarshaler 接口；encoding/json 包中的 Marshaler 和 Unmarshaler 接口。

它还实现了 gob 包中的 GobEncoder 和 GobDecoder 接口。

对于文本序列化 / 反序列化，通过 Parse 和 Format 实现；对于二进制序列化，需要单独实现。

对于 json，使用的是 time.RFC3339Nano 这种格式。通常程序中不使用这种格式。解决办法是定义自己的类型。如：

type OftenTime time.Time  
func (self OftenTime) MarshalJSON() ([]byte, error) {  
 t := time.Time(self)  
 if y := t.Year(); y < 0 || y >= 10000 {  
 return nil, errors.New("Time.MarshalJSON: year outside of range [0,9999]")  
 }  
 // 注意 `"2006-01-02 15:04:05"`。因为是 JSON，双引号不能少  
 return []byte(t.Format(`"2006-01-02 15:04:05"`)), nil  
}

### 3.2.5. Round 和 Truncate 方法

比如，有这么个需求：获取当前时间整点的 Time 实例。例如，当前时间是 15:54:23，需要的是 15:00:00。我们可以这么做：

t, \_ := time.ParseInLocation("2006-01-02 15:04:05", time.Now().Format("2006-01-02 15:00:00"), time.Local)  
fmt.Println(t)

实际上，time 包给我们提供了专门的方法，功能更强大，性能也更好，这就是 Round 和 Trunate，它们区别，一个是取最接近的，一个是向下取整。

使用示例：

t, \_ := time.ParseInLocation("2006-01-02 15:04:05", "2016-06-13 15:34:39", time.Local)// 整点（向下取整）  
fmt.Println(t.Truncate(1 \* time.*Hour*))// 整点（最接近），输出：2016-06-13 15:00:00 +0800 CST  
fmt.Println(t.Round(1 \* time.*Hour*)) //输出：2016-06-13 16:00:00 +0800 CST  
// 整分（向下取整）  
fmt.Println(t.Truncate(1 \* time.*Minute*))// 整分（最接近）输出：2016-06-13 15:34:00 +0800 CST  
fmt.Println(t.Round(1 \* time.*Minute*)) //输出：2016-06-13 15:35:00 +0800 CST  
  
t2, \_ := time.ParseInLocation("2006-01-02 15:04:05", t.Format("2006-01-02 15:00:00"), time.Local)  
fmt.Println(t2) //输出：2016-06-13 15:00:00 +0800 CST

# 4. 定时器

定时器是进程规划自己在未来某一时刻接获通知的一种机制。本节介绍两种定时器：Timer（到达指定时间触发且只触发一次）和 Ticker（间隔特定时间触发）。

## 4.1. Timer

### 4.1.1. 内部实现源码分析

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 (channel)，除非 Timer 是被 AfterFunc 函数创建的。

注意：Timer 的实例必须通过 NewTimer 或 AfterFunc 获得。

类型定义如下：

// The Timer type represents a single event.  
// When the Timer expires, the current time will be sent on C,  
// unless the Timer was created by AfterFunc.  
// A Timer must be created with NewTimer or AfterFunc.  
type Timer struct {  
 C <-chan Time  
 r runtimeTimer  
}

C 已经解释了，我们看看 runtimeTimer。它定义在 sleep.go 文件中，必须和 runtime 包中 time.go 文件中的 timer 必须保持一致：

// Package time knows the layout of this structure.  
// If this struct changes, adjust ../time/sleep.go:/runtimeTimer.  
type timer struct {  
 // If this timer is on a heap, which P's heap it is on.  
 // puintptr rather than \*p to match uintptr in the versions  
 // of this struct defined in other packages.  
 pp puintptr  
  
 // Timer wakes up at when, and then at when+period, ... (period > 0 only)  
 // each time calling f(arg, now) in the timer goroutine, so f must be  
 // a well-behaved function and not block.  
 //  
 // when must be positive on an active timer.  
 when int64  
 period int64  
 f func(any, uintptr)  
 arg any  
 seq uintptr  
  
 // What to set the when field to in timerModifiedXX status.  
 nextwhen int64  
  
 // The status field holds one of the values below.  
 status uint32  
}

我们通过 NewTimer() 来看这些字段都怎么赋值，是什么用途。

// NewTimer creates a new Timer that will send  
// the current time on its channel after at least duration d.  
func NewTimer(d Duration) \*Timer {  
 c := make(chan Time, 1)  
 t := &Timer{  
 C: c,  
 r: runtimeTimer{  
 when: when(d),  
 f: sendTime,  
 arg: c,  
 },  
 }  
 startTimer(&t.r)  
 return t  
}

在 when 表示的时间到时，会往 Timer.C 中发送当前时间。when 表示的时间是纳秒时间，正常通过 runtimeNano() + int64(d) 赋值。跟上一节中讲到的 now() 类似，runtimeNano() 也在 runtime 中实现（runtime · nanotime）：

* 调用系统调用 clock\_gettime 获取时钟值（这是 POSIX 时钟）。其中 clockid\_t 时钟类型是 CLOCK\_MONOTONIC，也就是不可设定的恒定态时钟。具体的是什么时间，SUSv3 规定始于未予规范的过去某一点，Linux 上，始于系统启动。
* 如果 clock\_gettime 不存在，则使用精度差些的系统调用 gettimeofday。

f 参数的值是 sendTime，定时器时间到时，会调用 f，并将 arg 和 seq 传给 f。

因为 Timer 是一次性的，所以 period 保留默认值 0。

定时器的具体实现逻辑，都在 runtime 中的 time.go 中，它的实现，没有采用经典 Unix 间隔定时器 setitimer 系统调用，也没有 采用 POSIX 间隔式定时器（相关系统调用：timer\_create、timer\_settime 和 timer\_delete），而是通过四叉树堆 (heep) 实现的（runtimeTimer 结构中的 i 字段，表示在堆中的索引）。通过构建一个最小堆，保证最快拿到到期了的定时器执行。定时器的执行，在专门的 goroutine 中进行的：go timerproc()。有兴趣的同学，可以阅读 runtime/time.go 的源码。

### 4.1.2. Timer 相关函数或方法的使用

****通过**time.After**模拟超时：****

c := make(chan int)  
go func() {  
 // time.Sleep(1 \* time.Second)  
 time.Sleep(3 \* time.*Second*)  
 <-c  
}()  
select {case c <- 1:  
 fmt.Println("channel...")case <-time.After(2 \* time.*Second*):  
 close(c)  
 fmt.Println("timeout...")  
}

**time.Stop**停止定时器 或**time.Reset**重置定时器****

start := time.Now()  
timer := time.AfterFunc(2\*time.*Second*, func() {  
 fmt.Println("after func callback, elaspe:", time.Now().Sub(start))  
})  
  
time.Sleep(1 \* time.*Second*)   
// time.Sleep(3\*time.Second)  
//Reset 在 Timer 还未触发时返回 true；触发了或 Stop 了，返回 false  
if timer.Reset(3 \* time.*Second*) {  
 fmt.Println("timer has not trigger!")  
} else {  
 fmt.Println("timer had expired or stop!")  
}  
  
time.Sleep(10 \* time.*Second*)  
// output:  
//timer has not trigger!  
//after func callback, elaspe: 4.00026461s

如果定时器还未触发，Stop 会将其移除，并返回 true；否则返回 false；后续再对该 Timer 调用 Stop，直接返回 false。

Reset 会先调用 stopTimer 再调用 startTimer，类似于废弃之前的定时器，重新启动一个定时器。返回值和 Stop 一样。

### 4.1.3. Sleep 的内部实现

查看 runtime/time.go 文件中的 timeSleep 可知，Sleep 的是通过 Timer 实现的，把当前 goroutine 作为 arg 参数（getg())。

## 4.2. Ticker 相关函数或方法的使用

Ticker 和 Timer 类似，区别是：Ticker 中的 runtimeTimer 字段的 period 字段会赋值为 NewTicker(d Duration) 中的 d，表示每间隔 d 纳秒，定时器就会触发一次。

除非程序终止前定时器一直需要触发，否则，不需要时应该调用 Ticker.Stop 来释放相关资源。

如果程序终止前需要定时器一直触发，可以使用更简单方便的 time.Tick 函数，因为 Ticker 实例隐藏起来了，因此，该函数启动的定时器无法停止。

## 4.3. 定时器的实际应用

在实际开发中，定时器用的较多的会是 Timer，如模拟超时，而需要类似 Tiker 的功能时，可以使用实现了 cron spec 的库 [cron](https://github.com/robfig/cron" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter04/_blank)，感兴趣的可以参考文章：[《Go 语言版 crontab》](http://blog.studygolang.com/2014/02/go_crontab/" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter04/_blank)。