**golang包time用法详解**

在我们编程过程中，经常会用到与时间相关的各种务需求，下面来介绍 golang 中有关时间的一些基本用法,我们从 time 的几种 type 来开始介绍。

时间可分为时间点与时间段，golang 也不例外，提供了以下两种基础类型  
- 时间点(Time)  
- 时间段(Duration)

除此之外 golang 也提供了以下类型，做一些特定的业务  
- 时区(Location)  
- Ticker  
- Timer(定时器)

我们将按以上顺序来介绍 time 包的使用。

**时间点(Time)**

我们使用的所有与时间相关的业务都是基于点而延伸的，两点组成一个时间段，大多数应用也都是围绕这些点与面去做逻辑处理。

**初始化**

go 针对不同的参数类型提供了以下初始化的方式

*// func Now() Time*

fmt.Println(time.Now())

*// func Parse(layout, value string) (Time, error)*

time.Parse("2016-01-02 15:04:05", "2018-04-23 12:24:51")

*// func ParseInLocation(layout, value string, loc \*Location) (Time, error) (layout已带时区时可直接用Parse)*

time.ParseInLocation("2006-01-02 15:04:05", "2017-05-11 14:06:06", time.Local)

*// func Unix(sec int64, nsec int64) Time*

time.Unix(1e9, 0)

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

time.Date(2018, 1, 2, 15, 30, 10, 0, time.Local)

*// func (t Time) In(loc \*Location) Time 当前时间对应指定时区的时间*

loc, \_ := time.LoadLocation("America/Los\_Angeles")

fmt.Println(time.Now().In(loc))

*// func (t Time) Local() Time*

获取到时间点之后为了满足业务和设计，需要转换成我们需要的格式，也就是所谓的时间格式化。

**格式化**

**to string**

格式化为字符串我们需要使用 time.Format 方法来转换成我们想要的格式

fmt.Println(time.Now().Format("2006-01-02 15:04:05")) *// 2018-04-24 10:11:20*

fmt.Println(time.Now().Format(time.UnixDate)) *// Tue Apr 24 09:59:02 CST 2018*

Format 函数中可以指定你想使用的格式，同时 time 包中也给了一些我们常用的格式

const (

ANSIC = "Mon Jan \_2 15:04:05 2006"

UnixDate = "Mon Jan \_2 15:04:05 MST 2006"

RubyDate = "Mon Jan 02 15:04:05 -0700 2006"

RFC822 = "02 Jan 06 15:04 MST"

RFC822Z = "02 Jan 06 15:04 -0700" *// RFC822 with numeric zone*

RFC850 = "Monday, 02-Jan-06 15:04:05 MST"

RFC1123 = "Mon, 02 Jan 2006 15:04:05 MST"

RFC1123Z = "Mon, 02 Jan 2006 15:04:05 -0700" *// RFC1123 with numeric zone*

RFC3339 = "2006-01-02T15:04:05Z07:00"

RFC3339Nano = "2006-01-02T15:04:05.999999999Z07:00"

Kitchen = "3:04PM"

*// Handy time stamps.*

Stamp = "Jan \_2 15:04:05"

StampMilli = "Jan \_2 15:04:05.000"

StampMicro = "Jan \_2 15:04:05.000000"

StampNano = "Jan \_2 15:04:05.000000000"

)

**注意**: galang 中指定的**特定时间格式**为 "2006-01-02 15:04:05 -0700 MST"， 为了记忆方便，按照美式时间格式 *月日时分秒年* 外加时区 排列起来依次是 *01/02 03:04:05PM ‘06 -0700*，刚开始使用时需要注意。

**to time stamp**

func (t Time) Unix() int64

func (t Time) UnixNano() int64

fmt.Println(time.Now().Unix())

*// 获取指定日期的时间戳*

dt, \_ := time.Parse("2016-01-02 15:04:05", "2018-04-23 12:24:51")

fmt.Println(dt.Unix())

fmt.Println(time.Date(2018, 1,2,15,30,10,0, time.Local).Unix())

**其他**

time 包还提供了一些常用的方法，基本覆盖了大多数业务，从方法名就能知道代表的含义就不一一说明了。

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

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

func (t Time) Year() int

func (t Time) Month() Month

func (t Time) Day() int

func (t Time) Hour() int

func (t Time) Minute() int

func (t Time) Second() int

func (t Time) Nanosecond() int

func (t Time) YearDay() int

func (t Time) Weekday() Weekday

func (t Time) ISOWeek() (year, week int)

func (t Time) IsZero() bool

func (t Time) Local() Time

func (t Time) Location() \*Location

func (t Time) Zone() (name string, offset int)

func (t Time) Unix() int64

**时间段(Duartion)**

介绍完了时间点，我们再来介绍时间段，即 Duartion 类型， 我们业务也是很常用的类型。

*// func ParseDuration(s string) (Duration, error)*

tp, \_ := time.ParseDuration("1.5s")

fmt.Println(tp.Truncate(1000), tp.Seconds(), tp.Nanoseconds())

func (d Duration) Hours() float64

func (d Duration) Minutes() float64

func (d Duration) Seconds() float64

func (d Duration) Nanoseconds() int64

func (d Duration) Round(m Duration) Duration *// 四舍五入*

func (d Duration) Truncate(m Duration) Duration *// 向下取整*

**时区(Location)**

我们在来介绍一下时区的相关的函数

*// 默认UTC*

loc, err := time.LoadLocation("")

*// 服务器设定的时区，一般为CST*

loc, err := time.LoadLocation("Local")

*// 美国洛杉矶PDT*

loc, err := time.LoadLocation("America/Los\_Angeles")

*// 获取指定时区的时间点*

local, \_ := time.LoadLocation("America/Los\_Angeles")

fmt.Println(time.Date(2018,1,1,12,0,0,0, local))

可以在 *$GOROOT/lib/time/zoneinfo.zip* 文件下看到所有时区。

**时间运算**

好了，基础的类型我们介绍完，现在开始时间运算相关的函数，也是日常业务中我们大量应用的。

*// func Sleep(d Duration) 休眠多少时间，休眠时处于阻塞状态，后续程序无法执行*

time.Sleep(time.Duration(10) \* time.Second)

*// func After(d Duration) <-chan Time 非阻塞,可用于延迟*

time.After(time.Duration(10) \* time.Second)

*// func Since(t Time) Duration 两个时间点的间隔*

start := time.Now()

fmt.Println(time.Since(start)) *// 等价于 Now().Sub(t)， 可用来计算一段业务的消耗时间*

func Until(t Time) Duration *// 等价于 t.Sub(Now())，t与当前时间的间隔*

*// func (t Time) Add(d Duration) Time*

fmt.Println(dt.Add(time.Duration(10) \* time.Second)) *// 加*

func (t Time) Sub(u Time) Duration *// 减*

*// func (t Time) AddDate(years int, months int, days int) Time*

fmt.Println(dt.AddDate(1, 1, 1))

*// func (t Time) Before(u Time) bool*

*// func (t Time) After(u Time) bool*

*// func (t Time) Equal(u Time) bool 比较时间点时尽量使用Equal函数*

我们大概就介绍完了多数涉及时间点与时间段的函数，接下面我们通过一些使用场景来做一些演示。

**使用场景**

**日期时间差**

dt1 := time.Date(2018, 1, 10, 0, 0, 1, 100, time.Local)

dt2 := time.Date(2018, 1, 9, 23, 59, 22, 100, time.Local)

*// 不用关注时区，go会转换成时间戳进行计算*

fmt.Println(dt1.Sub(dt2))

**基于当前时间的前后运算**

now := time.Now()

*// 一年零一个月一天之后*

fmt.Println(now.Date(1,1,1))

*// 一段时间之后*

fmt.Println(now.Add(time.Duration(10)\*time.Minute))

*// 计算两个时间点的相差天数*

dt1 = time.Date(dt1.Year(), dt1.Month(), dt1.Day(), 0, 0, 0, 0, time.Local)

dt2 = time.Date(dt2.Year(), dt2.Month(), dt2.Day(), 0, 0, 0, 0, time.Local)

fmt.Println(int(math.Ceil(dt1.Sub(dt2).Hours() / 24)))

**时区转换**

*// time.Local 用来表示当前服务器时区*

*// 自定义地区时间*

secondsEastOfUTC := int((8 \* time.Hour).Seconds())

beijing := time.FixedZone("Beijing Time", secondsEastOfUTC)

fmt.Println(time.Date(2018,1,2,0,0,0,0, beijing)) *// 2018-01-02 00:00:00 +0800 Beijing Time*

*// 当前时间转为指定时区时间*

fmt.Println(time.Now().In(beijing))

*// 指定时间转换成指定时区对应的时间*

dt, err := time.ParseInLocation("2006-01-02 15:04:05", "2017-05-11 14:06:06", time.Local)

*// 当前时间在零时区年月日 时分秒 时区*

year, mon, day := time.Now().UTC().Date() *// 2018 April 24*

hour, min, sec := time.Now().UTC().Clock() *// 3 47 15*

zone, \_ := time.Now().UTC().Zone() *// UTC*

**比较两个时间点**

dt := time.Date(2018, 1, 10, 0, 0, 1, 100, time.Local)

fmt.Println(time.Now().After(dt)) *// true*

fmt.Println(time.Now().Before(dt)) *// false*

*// 是否相等 判断两个时间点是否相等时推荐使用 Equal 函数*

fmt.Println(dt.Equal(time.Now()))

**设置执行时间**

通过time.After 函数与 select 结合使用可用于处理程序超时设定

select {

case m := <- c:

*// do something*

case <- time.After(time.Duration(1)\*time.Second):

fmt.Println("time out")

}

**Ticker类型**

Ticker 类型包含一个 channel，有时我们会遇到每隔一段时间执行的业务(比如设置心跳时间等)，就可以用它来处理，这是一个*重复的过程*

*// 无法取消*

tick := time.Tick(1 \* time.Minute)

for \_ = range tick {

*// do something*

}

*// 可通过调用ticker.Stop取消*

ticker := time.NewTicker(1 \* time.Minute)

for \_ = range tick {

*// do something*

}

**Timer类型**

Timer 类型用来代表一个单独的事件，当设置的时间过期后，发送当前的时间到 channel, 我们可以通过以下两种方式来创建

func AfterFunc(d Duration, f func()) \*Timer *// 指定一段时间后指定的函数*

func NewTimer(d Duration) \*Timer

以上两函数都可以使用 Reset, 这个有个需要注意的地方是使用 Reset 时需要确保 t.C 通道被释放时才能调用，以防止发生资源竞争的问题，可通过以下方式解决

if !t.Stop() {

<-t.C

}

t.Reset(d)