**说说时间那些事儿**

**go的time包提供了 测量 和 显示 时间的功能，外加一个定时器**

**首先介绍一下时钟的概念。**

**Unix中定义了四种时钟：**

**1.日历事件**

该值是自1970年1月1日00:00:00以来所经过的秒数累计值。

系统基本数据类型time\_t用于保存这种时间值

**2.进程时间**

也被称为CPU时间，用于度量进程使用的中央处理器资源。进程时间以时钟滴答计算。

系统基本数据类型clock\_t保存这种时间值。

**go语言中的时钟主要使用了两个时钟：**

1.CLOCK\_REALTIME

就是wall time，即是实际的时间，用户可以自行修改

2.CLOCK\_MONOTONIC

单调时间，即从某个时间点开始到现在过去的时间，用户不可修改

那么为什么要定义两个时钟呢？

主要是用处不同，

**realtime用于告知时间，而monotonic用于测量时间**

官方文档给了下面这个例子

1. start := time.Now()
2. ... operation that takes 20 milliseconds ...
3. t := time.Now()
4. elapsed := t.Sub(start)

即使在t之前的operation中修改了wall time，最后得到的结果也还是20毫秒，

因为这里计算使用的是monotonic时间。

那么wall time和monotonic是怎么定义在数据结构中的呢？

请看下面的time.Time()基本定义

**一.go中时间的基本定义**

**1.时间与时间段**

**Time time.Time()**

1. type Time struct {
2. // wall and ext encode the wall time seconds, wall time nanoseconds,
3. // and optional monotonic clock reading in nanoseconds.
4. //
5. // From high to low bit position, wall encodes a 1-bit flag (hasMonotonic),
6. // a 33-bit seconds field, and a 30-bit wall time nanoseconds field.
7. // The nanoseconds field is in the range [0, 999999999].
8. // If the hasMonotonic bit is 0, then the 33-bit field must be zero
9. // and the full signed 64-bit wall seconds since Jan 1 year 1 is stored in ext.
10. // If the hasMonotonic bit is 1, then the 33-bit field holds a 33-bit
11. // unsigned wall seconds since Jan 1 year 1885, and ext holds a
12. // signed 64-bit monotonic clock reading, nanoseconds since process start.
13. wall uint64 //距离公元1年1月1日00:00:00UTC的秒数
14. ext int64 //纳秒
16. // loc specifies the Location that should be used to
17. // determine the minute, hour, month, day, and year
18. // that correspond to this Time.
19. // The nil location means UTC.
20. // All UTC times are represented with loc==nil, never loc==&utcLoc.
21. loc \*Location
22. }

根据注释，我们可以发现，wall字段的64位中，分成了三个部分

1.a 1-bit flag(hasMonotonic)用于判别是否含有monotonic，

**若为0，则33位的块置为0，ext保存自0001-01-01起的时间戳**

**若为1，则33位的块保存自1885-01-01起的时间戳，而monotonic time会被存在ext字段**

2.a 33-bit seconds field

3.a 30-bit wall time nanoseconds field

**注意：go语言运算符==不仅比较wall time，还会比较location和monotonic。**

**因此如果没有事先保证为所有值设置了相同的location，那么Time值不应该用于map或者数据库的key**

这可以通过使用t.UTC()或者t.Local()方法实现，通过设置t = t.Round(0)来剥离monotonic读数

**Duration time.Duration()**

持续时间.定义了以下持续时间类型.多用于时间的加减 需要传入Duration做为参数的时候

1. const (
2. Nanosecond Duration = 1
3. Microsecond = 1000 \* Nanosecond
4. Millisecond = 1000 \* Microsecond
5. Second = 1000 \* Millisecond
6. Minute = 60 \* Second
7. Hour = 60 \* Minute
8. )

**2.时区： location**

同一时刻转换成为不同的时区，就需要通过不同的location来进行

1. // A Location maps time instants to the zone in use at that time.
2. // Typically, the Location represents the collection of time offsets
3. // in use in a geographical area, such as CEST and CET for central Europe.
4. type Location struct {
5. name string
6. zone []zone
7. tx []zoneTrans
9. // Most lookups will be for the current time.
10. // To avoid the binary search through tx, keep a
11. // static one-element cache that gives the correct
12. // zone for the time when the Location was created.
13. // if cacheStart <= t < cacheEnd,
14. // lookup can return cacheZone.
15. // The units for cacheStart and cacheEnd are seconds
16. // since January 1, 1970 UTC, to match the argument
17. // to lookup.
18. cacheStart int64
19. cacheEnd int64
20. cacheZone \*zone
21. }
23. // A zone represents a single time zone such as CEST or CET.
24. type zone struct {
25. name string // abbreviated name, "CET"
26. offset int // seconds east of UTC 比如我们北京位于东八区 8\*36400 = 291200
27. isDST bool // is this zone Daylight Savings Time?
28. }
30. // A zoneTrans represents a single time zone transition.
31. type zoneTrans struct {
32. when int64 // transition time, in seconds since 1970 GMT
33. index uint8 // the index of the zone that goes into effect at that time
34. isstd, isutc bool // ignored - no idea what these mean
35. }

**3.格式化：format**

1. const (
2. ANSIC = "Mon Jan \_2 15:04:05 2006"
3. UnixDate = "Mon Jan \_2 15:04:05 MST 2006"
4. RubyDate = "Mon Jan 02 15:04:05 -0700 2006"
5. RFC822 = "02 Jan 06 15:04 MST"
6. RFC822Z = "02 Jan 06 15:04 -0700" // RFC822 with numeric zone
7. RFC850 = "Monday, 02-Jan-06 15:04:05 MST"
8. RFC1123 = "Mon, 02 Jan 2006 15:04:05 MST"
9. RFC1123Z = "Mon, 02 Jan 2006 15:04:05 -0700" // RFC1123 with numeric zone
10. RFC3339 = "2006-01-02T15:04:05Z07:00"
11. RFC3339Nano = "2006-01-02T15:04:05.999999999Z07:00"
12. Kitchen = "3:04PM"
13. // Handy time stamps.
14. Stamp = "Jan \_2 15:04:05"
15. StampMilli = "Jan \_2 15:04:05.000"
16. StampMicro = "Jan \_2 15:04:05.000000"
17. StampNano = "Jan \_2 15:04:05.000000000"
18. )

**二.time库中函数解析应用**

**1.func Now**

1. func Now() Time

返回当前本地时间

先来看下源码

1. // Now returns the current local time.
2. func Now() Time {
3. sec, nsec, mono := now()
4. mono -= startNano
5. sec += *unixToInternal* - *minWall*
6. if uint64(sec)>>33 != 0 {
7. return Time{uint64(nsec), sec + *minWall*, Local}
8. }
9. return Time{*hasMonotonic* | uint64(sec)<<*nsecShift* | uint64(nsec), mono, Local}
10. }
12. *unixToInternal* int64 = (1969\*365 + 1969/4 - 1969/100 + 1969/400) \* *secondsPerDay*

这里需要说明一下， unixToInternal指的是1-1-1到1970-1-1经历的秒数

程序在存储和传递 Time 时应该是用值 ( t Time ) 而不是指针 ( t \*Time )，也就是说，Time 类型的变量和结构字段的类型应为 time.Time，而不是 \*time.Time 。除了方法 GobDecode，UnmarshalBinary，UnmarshalJSON 和 UnmarshalText 不是并发安全之外，多个 goroutine 可以同时使用 Time 值。同时，因为Time是一个值，所以就可以用来进行比较了，比如Before After Equal

**对应Time可使用的其他常用函数：**

1. // UTC returns t with the location set to UTC.
2. // 获取当前的UTC时间
3. func (t Time) UTC() Time {
4. t.setLoc(&utcLoc)
5. return t
6. }
8. // Local returns t with the location set to local time.
9. func (t Time) Local() Time {
10. t.setLoc(Local)
11. return t
12. }
14. // In returns a copy of t representing the same time instant, but
15. // with the copy's location information set to loc for display
16. // purposes.
17. //
18. // In panics if loc is nil.
19. // In（）方法仅设置loc，不会改变时间值
20. func (t Time) In(loc \*Location) Time {
21. if loc == nil {
22. panic("time: missing Location in call to Time.In")
23. }
24. t.setLoc(loc)
25. return t
26. }
28. // Location returns the time zone information associated with t.
29. func (t Time) Location() \*Location {
30. l := t.loc
31. if l == nil {
32. l = UTC
33. }
34. return l
35. }
37. // Zone computes the time zone in effect at time t, returning the abbreviated
38. // name of the zone (such as "CET") and its offset in seconds east of UTC.
39. func (t Time) Zone() (name string, offset int) {
40. name, offset, \_, \_ = t.loc.lookup(t.unixSec())
41. return
42. }
44. // Unix returns t as a Unix time, the number of seconds elapsed
45. // since January 1, 1970 UTC. The result does not depend on the
46. // location associated with t.
47. func (t Time) Unix() int64 {
48. return t.unixSec()
49. }
50. // unixSec returns the time's seconds since Jan 1 1970 (Unix time).
51. //func (t \*Time) unixSec() int64 { return t.sec() + *internalToUnix* }
53. // UnixNano returns t as a Unix time, the number of nanoseconds elapsed
54. // since January 1, 1970 UTC. The result is undefined if the Unix time
55. // in nanoseconds cannot be represented by an int64 (a date before the year
56. // 1678 or after 2262). Note that this means the result of calling UnixNano
57. // on the zero Time is undefined. The result does not depend on the
58. // location associated with t.
59. func (t Time) UnixNano() int64 {
60. return (t.unixSec())\*1e9 + int64(t.nsec())
61. }

**2.func Parse**

1. func Parse(layout, value string) (Time, error)

Parse解析一个格式化的时间字符串并返回它代表的时间。

layout：定义输入的时间格式，比如(time.RFC3339或者"2006-01-02 15:04:05")

**如果没有指定时区，则会认为这是一个UTC时间，时区是UTC**

value：value的时间格式需要与layout格式保持一致

**2.1 mysql对应的时间处理**

MySQL驱动解析时间的前提是连接字符串加了parseTime和loc, 如果parseTime为false, 会把mysql的date类型变成[]byte/string自行处理, parseTime为true才处理时间, loc指定MySQL中存储时间数据的时区, 如果没有指定loc, 用UTC. 序列化和反序列化均使用连接字符串中的设定的loc, SQL语句中的time.Time类型的参数的时区信息如果和loc不同, 则会调用t.In(loc)方法转时区.

解析连接字符串的代码位于parseDSNParams函数https://github.com/go-sql-driver/mysql/blob/master/dsn.go#L526-L550

解析SQL语句中time.Time类型的参数的代码位于mysqlConn.interpolateParams方法https://github.com/go-sql-driver/mysql/blob/master/connection.go#L227-L271

从MySQL数据流中解析时间的代码位于textRows.readRow方法https://github.com/go-sql-driver/mysql/blob/master/packets.go#L772-L791

**注意：只要MySQL连接字符串设置了parseTime=true, 就会解析时间, 不管是用string还是time.Time接收.**

**3.Timer(定时器）**

**3.1 基本定义**

1. type Timer struct {
2. C <-chan Time
3. // 内含隐藏或非导出字段
4. }

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。

**3.2 func NewTimer**

1. 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。

Example:

1. fmt.Println(time.Now())
3. timer := time.NewTimer(time.Second \* 2)
5. <-timer.C
7. fmt.Println(time.Now())
9. // 输出:
10. // 2018-06-04 12:55:32.426676958 +0800 CST m=+0.000332587
11. // 2018-06-04 12:55:34.42745008 +0800 CST m=+2.001045690

//此处m=+2.001045690即为monotonic时间

**3.3 func AfterFunc**

1. func AfterFunc(d Duration, f func()) \*Timer

AfterFunc 另起一个 go 协程等待时间段 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。

Example:

1. wait := sync.WaitGroup{}
2. fmt.Println("start", time.Now())
4. wait.Add(1)
6. timer := time.AfterFunc(time.Second \* 3, func() {
7. fmt.Println("get timer", time.Now())
8. wait.Done()
9. })
11. time.Sleep(time.Second)
12. fmt.Println("sleep", time.Now())
14. timer.Reset(time.Second \* 2)
16. wait.Wait()
18. // 输出:
19. // start 2018-06-04 13:11:25.478294853 +0800 CST m=+0.000332881
20. // sleep 2018-06-04 13:11:26.480826469 +0800 CST m=+1.002156500
21. // get timer 2018-06-04 13:11:28.483455973 +0800 CST m=+3.003496118

**3.4 func (t \*Timer)Reset**

1. 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。

**3.5 func (t \*Timer)Stop**

1. 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。

**4.Ticker(断续器）**

1. type Ticker struct {
2. C <-chan Time // 周期性传递时间信息的通道
3. // 内含隐藏或非导出字段
4. }

Ticker 保管一个通道，并每隔一段时间向其传递”tick”。

**4.1 func NewTicker**

1. 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会触发panic。关闭该 Ticker 可以释放相关资源。

Example:

1. fmt.Println("start", time.Now())
3. ticker := time.NewTicker(time.Second)
5. go func() {
6. for tick := range ticker.C {
7. fmt.Println("tick at", tick)
8. }
9. }()
11. time.Sleep(time.Second \* 5)
12. ticker.Stop()
14. fmt.Println("stoped", time.Now())
16. // 输出:
17. // start 2018-06-04 13:21:20.443700752 +0800 CST m=+0.000376994
18. // tick at 2018-06-04 13:21:21.448276294 +0800 CST m=+1.004922401
19. // tick at 2018-06-04 13:21:22.44666211 +0800 CST m=+2.003278267
20. // tick at 2018-06-04 13:21:23.446749266 +0800 CST m=+3.003335423
21. // tick at 2018-06-04 13:21:24.445154097 +0800 CST m=+4.001710303
22. // stoped 2018-06-04 13:21:25.445239727 +0800 CST m=+5.001765933

**5.时间戳相关**

时间戳定义：通过距离UTC时区1970-01-01 00:00:00的秒数，即我们先把时间换算成，UTC时间，然后计算两个时间相差的秒数。

我国一般使用的是北京时间，假设时间为T，那么对应的时间戳为：

时间戳 =（T + 8个小时 - 1970-01-01 00:00:00 ）秒

在go里面，通过time.Now()拿到的时间，默认的时区为本地时区；

通过time.Parse(layout, value string) (Time, error)方法拿到的时间，时区为UTC对应的时区。

通过time.ParseInLocation(layout, value string, loc \*Location) (Time, error)方法拿到的时间，是调用该方法传递的loc对应的时区。

**5.1 获取时间戳**

1. //获取当前时间戳
2. time.Now().Unix()
4. //获取时间字符串对应的时间戳
5. var strTime = "2018-03-11 15:35:03"
6. timeObj ,\_ := time.Parse("2006-01-02 15:04:05", strTime)//获取时间对象，获取的时间对应的时区是UTC
7. //timeObj2, \_ := time.ParseInLocation("2006-01-02 15:04:05", strTime, time.Local) //获取时间对象，获取的时间对应的时区是本地时间
8. timeObj.Unix() //获取对应的时间戳 int64类型

**附：时间相关标准**

**1.UTC**

1. 协调世界时（英语：Coordinated Universal Time，法语：Temps Universel Coordonné，简称UTC）是最主要的世界时间标准，其以原子时秒长为基础，在时刻上尽量接近于格林尼治标准时间。中华民国采用CNS 7648的《资料元及交换格式–资讯交换–日期及时间的表示法》（与ISO 8601类似）称之为世界协调时间。中华人民共和国采用ISO 8601:2000的国家标准GB/T 7408-2005《数据元和交换格式 信息交换日期和时间表示法》中亦称之为协调世界时。
2. 协调世界时是世界上调节时钟和时间的主要时间标准，它与0度经线的平太阳时相差不超过1秒，并不遵守夏令时。协调世界时是最接近格林威治标准时间(GMT)的几个替代时间系统之一。对于大多数用途来说，UTC时间被认为能与GMT时间互换，但GMT时间已不再被科学界所确定。
3. 人们认为最准确的计算应该是使用‘原子震荡周期’所计算的物理时钟了 (Atomic Clock, 也被称为原子钟)，这也被定义为标准时间 (International Atomic Time)。而我们常常看见的UTC（Universal Time Coordinated，世界协调时间）就是利用这种 Atomic Clock 为基准所定义出来的正确时间。UTC 标准时间是以 GMT 这个时区为主，所以本地时间与 UTC 时间的时差就是本地时间与 GMT 时间的时差。
4. UTC + 时区差 = 本地时间
5. example：
6. UTC + 8h = 北京时间

**2.ISO 8601**

1. 计算某一天在一年的第几周会用到此标准
2. 国际标准ISO 8601，是国际标准化组织的日期和时间的表示方法，全称为《数据存储和交换形式·信息交换·日期和时间的表示方法》。目前是第三版“ISO8601:2004”以替代第一版“ISO8601:1988”与第二版“ISO8601:2000”。

**3.UNIX时间**

1. UNIX时间，或称POSIX时间是UNIX或类UNIX系统使用的时间表示方式：从协调世界时1970年1月1日0时0分0秒起至现在的总秒数，不考虑闰秒。 在多数Unix系统上Unix时间可以通过date+%s指令来检查。

**4.GMT**

1. 格林威治标准时间指位于英国伦敦郊区的皇家格林尼治天文台的标准时间，因为本初子午线被定义在通过那里的经线(UTC与GMT时间基本相同)。

**5.CST**

1. 中国标准时间
2. CST = UTC + 8 = GMT + 8