time package   standard library

Version: go1.20.1 Latest | Published: Feb 14, 2023 | License: BSD-3-Clause | Imports: 5 |
Imported by: 1,299,635

Details			
	⊘ Valid go.mod file ❓	⊘ Redistributable license ❓	⊘ Tagged version ❓
	⊘ Stable version ❓		
	Learn more		
Repository	cs.opensource.google/go/go		
Links	🛡 Report a Vulnerability		

▤ Documentation ▾

‹› Documentation

Rendered for   linux/amd64 ▾

Overview

Monotonic Clocks

Package time provides functionality for measuring and displaying time.

The calendrical calculations always assume a Gregorian calendar, with no leap seconds.

Monotonic Clocks

Operating systems provide both a "wall clock," which is subject to changes for clock synchronization, and a "monotonic clock," which is not. The general rule is that the wall clock is for telling time and the monotonic clock is for measuring time. Rather than split the API, in this package the Time returned by time.Now contains both a wall clock reading and a monotonic clock reading; later time-telling operations use the wall clock reading, but later time-measuring operations, specifically comparisons and subtractions, use the monotonic clock reading.

For example, this code always computes a positive elapsed time of approximately 20 milliseconds, even if the wall clock is changed during the operation being timed:

```
start := time.Now()
... operation that takes 20 milliseconds ...
t := time.Now()
elapsed := t.Sub(start)
```

Other idioms, such as time.Since(start), time.Until(deadline), and time.Now().Before(deadline), are similarly robust against wall clock resets.

The rest of this section gives the precise details of how operations use monotonic clocks, but understanding those details is not required to use this package.

The Time returned by time.Now contains a monotonic clock reading. If Time t has a monotonic clock reading, t.Add adds the same duration to both the wall clock and monotonic clock readings to compute the result. Because t.AddDate(y, m, d), t.Round(d), and t.Truncate(d) are wall time computations, they always strip any monotonic clock reading from their results. Because t.In, t.Local, and t.UTC are used for their effect on the interpretation of the wall time, they also strip any monotonic clock reading from their results. The canonical way to strip a monotonic clock reading is to use t = t.Round(0).

If Times t and u both contain monotonic clock readings, the operations t.After(u), t.Before(u), t.Equal(u), t.Compare(u), and t.Sub(u) are carried out using the monotonic clock readings alone, ignoring the wall clock readings. If either t or u contains no monotonic clock reading, these operations fall back to using the wall clock readings.

On some systems the monotonic clock will stop if the computer goes to sleep. On such a system, t.Sub(u) may not accurately reflect the actual time that passed between t and u.

Because the monotonic clock reading has no meaning outside the current process, the serialized forms generated by t.GobEncode, t.MarshalBinary, t.MarshalJSON, and t.MarshalText omit the monotonic clock reading, and t.Format provides no format for it. Similarly, the constructors time.Date, time.Parse, time.ParseInLocation, and time.Unix, as well as the unmarshalers t.GobDecode, t.UnmarshalBinary. t.UnmarshalJSON, and t.UnmarshalText always create times with no monotonic clock reading.

The monotonic clock reading exists only in Time values. It is not a part of Duration values or the Unix times returned by t.Unix and friends.

Note that the Go == operator compares not just the time instant but also the Location and the monotonic clock reading. See the documentation for the Time type for a discussion of equality testing for Time values.

For debugging, the result of t.String does include the monotonic clock reading if present. If t != u because of different monotonic clock readings, that difference will be visible when printing t.String() and u.String().

Index

```
        func (d Duration) Milliseconds() int64
        func (d Duration) Minutes() float64
        func (d Duration) Nanoseconds() int64
        func (d Duration) Round(m Duration) Duration
        func (d Duration) Seconds() float64
        func (d Duration) String() string
        func (d Duration) Truncate(m Duration) Duration
type Location
        func FixedZone(name string, offset int) *Location
        func LoadLocation(name string) (*Location, error)
        func LoadLocationFromTZData(name string, data []byte) (*Location, error)
        func (l *Location) String() string
type Month
        func (m Month) String() string
type ParseError
        func (e *ParseError) Error() string
type Ticker
        func NewTicker(d Duration) *Ticker
        func (t *Ticker) Reset(d Duration)
        func (t *Ticker) Stop()
type Time
        func Date(year int, month Month, day, hour, min, sec, nsec int, loc *Location) Time
        func Now() Time
        func Parse(layout, value string) (Time, error)
        func ParseInLocation(layout, value string, loc *Location) (Time, error)
        func Unix(sec int64, nsec int64) Time
        func UnixMicro(usec int64) Time
        func UnixMilli(msec int64) Time
        func (t Time) Add(d Duration) Time
        func (t Time) AddDate(years int, months int, days int) Time
        func (t Time) After(u Time) bool
        func (t Time) AppendFormat(b []byte, layout string) []byte
        func (t Time) Before(u Time) bool
        func (t Time) Clock() (hour, min, sec int)
        func (t Time) Compare(u Time) int
        func (t Time) Date() (year int, month Month, day int)
        func (t Time) Day() int
        func (t Time) Equal(u Time) bool
        func (t Time) Format(layout string) string
        func (t Time) GoString() string
        func (t *Time) GobDecode(data []byte) error
        func (t Time) GobEncode() ([]byte, error)
        func (t Time) Hour() int
        func (t Time) ISOWeek() (year, week int)
        func (t Time) In(loc *Location) Time
        func (t Time) IsDST() bool
```

func (t Time) IsZero() bool

func (t Time) Local() Time

func (t Time) Location() *Location

func (t Time) MarshalBinary() ([]byte, error)

func (t Time) MarshalJSON() ([]byte, error)

func (t Time) MarshalText() ([]byte, error)

func (t Time) Minute() int

func (t Time) Month() Month

func (t Time) Nanosecond() int

func (t Time) Round(d Duration) Time

func (t Time) Second() int

func (t Time) String() string

func (t Time) Sub(u Time) Duration

func (t Time) Truncate(d Duration) Time

func (t Time) UTC() Time

func (t Time) Unix() int64

func (t Time) UnixMicro() int64

func (t Time) UnixMilli() int64

func (t Time) UnixNano() int64

func (t *Time) UnmarshalBinary(data []byte) error

func (t *Time) UnmarshalJSON(data []byte) error

func (t *Time) UnmarshalText(data []byte) error

func (t Time) Weekday() Weekday

func (t Time) Year() int

func (t Time) YearDay() int

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

func (t Time) ZoneBounds() (start, end Time)

type Timer

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

func NewTimer(d Duration) *Timer

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

func (t *Timer) Stop() bool

type Weekday

func (d Weekday) String() string

Examples

After

Date

Duration

Duration.Hours

Duration.Microseconds

Duration.Milliseconds

Duration.Minutes

Duration.Nanoseconds

Duration.Round

Duration.Seconds

Constants

```
const (
    Layout      = "01/02 03:04:05PM '06 -0700" // The reference time, in numerical orde
    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"
    DateTime   = "2006-01-02 15:04:05"
    DateOnly   = "2006-01-02"
    TimeOnly   = "15:04:05"
)
```

These are predefined layouts for use in Time.Format and time.Parse. The reference time used in these layouts is the specific time stamp:

```
01/02 03:04:05PM '06 -0700
```

(January 2, 15:04:05, 2006, in time zone seven hours west of GMT). That value is recorded as the constant named Layout, listed below. As a Unix time, this is 1136239445. Since MST is GMT-0700, the reference would be printed by the Unix date command as:

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

It is a regrettable historic error that the date uses the American convention of putting the numerical month before the day.

The example for Time.Format demonstrates the working of the layout string in detail and is a good reference.

Note that the RFC822, RFC850, and RFC1123 formats should be applied only to local times. Applying them to UTC times will use "UTC" as the time zone abbreviation, while strictly speaking those RFCs require the use of "GMT" in that case. In general RFC1123Z should be used instead of RFC1123 for servers that insist on that format, and RFC3339 should be preferred for new protocols. RFC3339, RFC822, RFC822Z, RFC1123, and RFC1123Z are useful for formatting; when used with time.Parse they do not accept all the time formats permitted by the RFCs and they do accept time formats not formally defined. The RFC3339Nano format removes trailing zeros from the seconds field and thus may not sort correctly once formatted.

Most programs can use one of the defined constants as the layout passed to Format or Parse. The rest of this comment can be ignored unless you are creating a custom layout string.

To define your own format, write down what the reference time would look like formatted your way; see the values of constants like ANSIC, StampMicro or Kitchen for examples. The model is to demonstrate what the reference time looks like so that the Format and Parse methods can apply the same transformation to a general time value.

Here is a summary of the components of a layout string. Each element shows by example the formatting of an element of the reference time. Only these values are recognized. Text in the layout string that is not recognized as part of the reference time is echoed verbatim during Format and expected to appear verbatim in the input to Parse.

```
Year: "2006" "06"
Month: "Jan" "January" "01" "1"
Day of the week: "Mon" "Monday"
Day of the month: "2" "_2" "02"
Day of the year: "__2" "002"
Hour: "15" "3" "03" (PM or AM)
Minute: "4" "04"
Second: "5" "05"
AM/PM mark: "PM"
```

Numeric time zone offsets format as follows:

```
"-0700"     ±hhmm
"-07:00"    ±hh:mm
"-07"       ±hh
"-070000"   ±hhmmss
"-07:00:00" ±hh:mm:ss
```

Replacing the sign in the format with a Z triggers the ISO 8601 behavior of printing Z instead of an offset for the UTC zone. Thus:

```
"Z0700"      Z or ±hhmm
"Z07:00"     Z or ±hh:mm
"Z07"        Z or ±hh
"Z070000"    Z or ±hhmmss
"Z07:00:00"  Z or ±hh:mm:ss
```

Within the format string, the underscores in "_2" and "__2" represent spaces that may be replaced by digits if the following number has multiple digits, for compatibility with fixed-width Unix time formats. A leading zero represents a zero-padded value.

The formats __2 and 002 are space-padded and zero-padded three-character day of year; there is no unpadded day of year format.

A comma or decimal point followed by one or more zeros represents a fractional second, printed to the given number of decimal places. A comma or decimal point followed by one or more nines represents a fractional second, printed to the given number of decimal places, with trailing zeros removed. For example "15:04:05,000" or "15:04:05.000" formats or parses with millisecond precision.

Some valid layouts are invalid time values for time.Parse, due to formats such as _ for space padding and Z for zone information.

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

```
    Hour                      = 60 * Minute
)
```

Common durations. There is no definition for units of Day or larger to avoid confusion across daylight savings time zone transitions.

To count the number of units in a Duration, divide:

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

To convert an integer number of units to a Duration, multiply:

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

Variables

This section is empty.

Functions

func After

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

After waits for the duration to elapse and then sends the current time on the returned channel. It is equivalent to NewTimer(d).C. The underlying Timer is not recovered by the garbage collector until the timer fires. If efficiency is a concern, use NewTimer instead and call Timer.Stop if the timer is no longer needed.

▶ Example

func Sleep

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

Sleep pauses the current goroutine for at least the duration d. A negative or zero duration causes Sleep to return immediately.

▶ Example

func Tick

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

Tick is a convenience wrapper for NewTicker providing access to the ticking channel only. While Tick is useful for clients that have no need to shut down the Ticker, be aware that without a way to shut it down the underlying Ticker cannot be recovered by the garbage collector; it "leaks". Unlike NewTicker, Tick will return nil if d <= 0.

▶ Example

Types

type Duration

```
type Duration int64
```

A Duration represents the elapsed time between two instants as an int64 nanosecond count. The representation limits the largest representable duration to approximately 290 years.

▶ Example

func ParseDuration

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

ParseDuration parses a duration string. A duration string is a possibly signed sequence of decimal numbers, each with optional fraction and a unit suffix, such as "300ms", "-1.5h" or "2h45m". Valid time units are "ns", "us" (or "µs"), "ms", "s", "m", "h".

▶ Example

func Since

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

Since returns the time elapsed since t. It is shorthand for time.Now().Sub(t).

func Until   added in go1.8

```
func Until(t Time) Duration
```

Until returns the duration until t. It is shorthand for t.Sub(time.Now()).

func (Duration) Abs   added in go1.19

```
func (d Duration) Abs() Duration
```

Abs returns the absolute value of d. As a special case, math.MinInt64 is converted to math.MaxInt64.

func (Duration) Hours

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

Hours returns the duration as a floating point number of hours.

▶ Example

func (Duration) Microseconds   added in go1.13

```
func (d Duration) Microseconds() int64
```

Microseconds returns the duration as an integer microsecond count.

▶ Example

func (Duration) Milliseconds   added in go1.13

```
func (d Duration) Milliseconds() int64
```

Milliseconds returns the duration as an integer millisecond count.

▶ Example

func (Duration) Minutes

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

Minutes returns the duration as a floating point number of minutes.

▶ Example

func (Duration) Nanoseconds

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

Nanoseconds returns the duration as an integer nanosecond count.

▶ Example

func (Duration) Round   added in go1.9

```
func (d Duration) Round(m Duration) Duration
```

Round returns the result of rounding d to the nearest multiple of m. The rounding behavior for halfway values is to round away from zero. If the result exceeds the maximum (or minimum) value that can be stored in a Duration, Round returns the maximum (or minimum) duration. If m <= 0, Round returns d unchanged.

▶ Example

func (Duration) Seconds

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

Seconds returns the duration as a floating point number of seconds.

▶ Example

func (Duration) String

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

String returns a string representing the duration in the form "72h3m0.5s". Leading zero units are omitted. As a special case, durations less than one second format use a smaller unit (milli-, micro-, or nanoseconds) to ensure that the leading digit is non-zero. The zero duration formats as 0s.

▶ Example

func (Duration) Truncate   added in go1.9

```
func (d Duration) Truncate(m Duration) Duration
```

Truncate returns the result of rounding d toward zero to a multiple of m. If m <= 0, Truncate returns d unchanged.

▶ Example

type Location

```
type Location struct {
    // contains filtered or unexported fields
}
```

A Location maps time instants to the zone in use at that time. Typically, the Location represents the collection of time offsets in use in a geographical area. For many Locations the time offset varies depending on whether daylight savings time is in use at the time instant.

▶ Example

```
var Local *Location = &localLoc
```

Local represents the system's local time zone. On Unix systems, Local consults the TZ environment variable to find the time zone to use. No TZ means use the system default /etc/localtime. TZ="" means use UTC. TZ="foo" means use file foo in the system timezone directory.

```
var UTC *Location = &utcLoc
```

UTC represents Universal Coordinated Time (UTC).

func FixedZone

```
func FixedZone(name string, offset int) *Location
```

FixedZone returns a Location that always uses the given zone name and offset (seconds east of UTC).

▶ Example

func LoadLocation

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

LoadLocation returns the Location with the given name.

If the name is "" or "UTC", LoadLocation returns UTC. If the name is "Local", LoadLocation returns Local.

Otherwise, the name is taken to be a location name corresponding to a file in the IANA Time Zone database, such as "America/New_York".

LoadLocation looks for the IANA Time Zone database in the following locations in order:

- the directory or uncompressed zip file named by the ZONEINFO environment variable
- on a Unix system, the system standard installation location
- $GOROOT/lib/time/zoneinfo.zip
- the time/tzdata package, if it was imported

▶ Example

func LoadLocationFromTZData   added in go1.10

```
func LoadLocationFromTZData(name string, data []byte) (*Location, error)
```

LoadLocationFromTZData returns a Location with the given name initialized from the IANA Time Zone database-formatted data. The data should be in the format of a standard IANA time zone file (for example, the content of /etc/localtime on Unix systems).

func (*Location) String

```
func (l *Location) String() string
```

String returns a descriptive name for the time zone information, corresponding to the name argument to LoadLocation or FixedZone.

type Month

```
type Month int
```

A Month specifies a month of the year (January = 1, ...).

▶ Example

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

func (Month) String

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

String returns the English name of the month ("January", "February", ...).

type ParseError

```
type ParseError struct {
    Layout     string
    Value      string
    LayoutElem string
    ValueElem  string
    Message    string
}
```

ParseError describes a problem parsing a time string.

func (*ParseError) Error

```
func (e *ParseError) Error() string
```

Error returns the string representation of a ParseError.

type Ticker

```
type Ticker struct {
    C <-chan Time // The channel on which the ticks are delivered.
    // contains filtered or unexported fields
}
```

A Ticker holds a channel that delivers "ticks" of a clock at intervals.

func NewTicker

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

NewTicker returns a new Ticker containing a channel that will send the current time on the channel after each tick. The period of the ticks is specified by the duration argument. The ticker will adjust the time interval or drop ticks to make up for slow receivers. The duration d must be greater than zero; if not, NewTicker will panic. Stop the ticker to release associated resources.

▶ Example

func (*Ticker) Reset   added in go1.15

```
func (t *Ticker) Reset(d Duration)
```

Reset stops a ticker and resets its period to the specified duration. The next tick will arrive after the new period elapses. The duration d must be greater than zero; if not, Reset will panic.

func (*Ticker) Stop

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

Stop turns off a ticker. After Stop, no more ticks will be sent. Stop does not close the channel, to prevent a concurrent goroutine reading from the channel from seeing an erroneous "tick".

type Time

```
type Time struct {
    // contains filtered or unexported fields
}
```

A Time represents an instant in time with nanosecond precision.

Programs using times should typically store and pass them as values, not pointers. That is, time variables and struct fields should be of type time.Time, not *time.Time.

A Time value can be used by multiple goroutines simultaneously except that the methods GobDecode, UnmarshalBinary, UnmarshalJSON and UnmarshalText are not concurrency-safe.

Time instants can be compared using the Before, After, and Equal methods. The Sub method subtracts two instants, producing a Duration. The Add method adds a Time and a Duration, producing a Time.

The zero value of type Time is January 1, year 1, 00:00:00.000000000 UTC. As this time is unlikely to come up in practice, the IsZero method gives a simple way of detecting a time that has not been initialized explicitly.

Each Time has associated with it a Location, consulted when computing the presentation form of the time, such as in the Format, Hour, and Year methods. The methods Local, UTC, and In return a Time with a specific location. Changing the location in this way changes only the presentation; it does not change the instant in time being denoted and therefore does not affect the computations described in earlier paragraphs.

Representations of a Time value saved by the GobEncode, MarshalBinary, MarshalJSON, and MarshalText methods store the Time.Location's offset, but not the location name. They therefore lose information about Daylight Saving Time.

In addition to the required "wall clock" reading, a Time may contain an optional reading of the current process's monotonic clock, to provide additional precision for comparison or subtraction. See the "Monotonic Clocks" section in the package documentation for details.

Note that the Go == operator compares not just the time instant but also the Location and the monotonic clock reading. Therefore, Time values should not be used as map or database keys without first guaranteeing that the identical Location has been set for all values, which can be achieved through use of the UTC or Local method, and that the monotonic clock reading has been stripped by setting t = t.Round(0). In general, prefer t.Equal(u) to t == u, since t.Equal uses the most accurate comparison available and correctly handles the case when only one of its arguments has a monotonic clock reading.

func Date

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

Date returns the Time corresponding to

```
yyyy-mm-dd hh:mm:ss + nsec nanoseconds
```

in the appropriate zone for that time in the given location.

The month, day, hour, min, sec, and nsec values may be outside their usual ranges and will be normalized during the conversion. For example, October 32 converts to November 1.

A daylight savings time transition skips or repeats times. For example, in the United States, March 13, 2011 2:15am never occurred, while November 6, 2011 1:15am occurred twice. In such cases, the choice

of time zone, and therefore the time, is not well-defined. Date returns a time that is correct in one of the two zones involved in the transition, but it does not guarantee which.

Date panics if loc is nil.

▶ Example

func Now

```
func Now() Time
```

Now returns the current local time.

func Parse

```
func Parse(layout, value string) (Time, error)
```

Parse parses a formatted string and returns the time value it represents. See the documentation for the constant called Layout to see how to represent the format. The second argument must be parseable using the format string (layout) provided as the first argument.

The example for Time.Format demonstrates the working of the layout string in detail and is a good reference.

When parsing (only), the input may contain a fractional second field immediately after the seconds field, even if the layout does not signify its presence. In that case either a comma or a decimal point followed by a maximal series of digits is parsed as a fractional second. Fractional seconds are truncated to nanosecond precision.

Elements omitted from the layout are assumed to be zero or, when zero is impossible, one, so parsing "3:04pm" returns the time corresponding to Jan 1, year 0, 15:04:00 UTC (note that because the year is 0, this time is before the zero Time). Years must be in the range 0000..9999. The day of the week is checked for syntax but it is otherwise ignored.

For layouts specifying the two-digit year 06, a value NN >= 69 will be treated as 19NN and a value NN < 69 will be treated as 20NN.

The remainder of this comment describes the handling of time zones.

In the absence of a time zone indicator, Parse returns a time in UTC.

When parsing a time with a zone offset like -0700, if the offset corresponds to a time zone used by the current location (Local), then Parse uses that location and zone in the returned time. Otherwise it records the time as being in a fabricated location with time fixed at the given zone offset.

When parsing a time with a zone abbreviation like MST, if the zone abbreviation has a defined offset in the current location, then that offset is used. The zone abbreviation "UTC" is recognized as UTC regardless of location. If the zone abbreviation is unknown, Parse records the time as being in a fabricated location with the given zone abbreviation and a zero offset. This choice means that such a

time can be parsed and reformatted with the same layout losslessly, but the exact instant used in the representation will differ by the actual zone offset. To avoid such problems, prefer time layouts that use a numeric zone offset, or use ParseInLocation.

▶ Example

func ParseInLocation   added in go1.1

```
func ParseInLocation(layout, value string, loc *Location) (Time, error)
```

ParseInLocation is like Parse but differs in two important ways. First, in the absence of time zone information, Parse interprets a time as UTC; ParseInLocation interprets the time as in the given location. Second, when given a zone offset or abbreviation, Parse tries to match it against the Local location; ParseInLocation uses the given location.

▶ Example

func Unix

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

Unix returns the local Time corresponding to the given Unix time, sec seconds and nsec nanoseconds since January 1, 1970 UTC. It is valid to pass nsec outside the range [0, 999999999]. Not all sec values have a corresponding time value. One such value is 1<<63-1 (the largest int64 value).

▶ Example

func UnixMicro   added in go1.17

```
func UnixMicro(usec int64) Time
```

UnixMicro returns the local Time corresponding to the given Unix time, usec microseconds since January 1, 1970 UTC.

▶ Example

func UnixMilli   added in go1.17

```
func UnixMilli(msec int64) Time
```

UnixMilli returns the local Time corresponding to the given Unix time, msec milliseconds since January 1, 1970 UTC.

▶ Example

func (Time) Add

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

Add returns the time t+d.

▶ Example

func (Time) AddDate

```
func (t Time) AddDate(years int, months int, days int) Time
```

AddDate returns the time corresponding to adding the given number of years, months, and days to t. For example, AddDate(-1, 2, 3) applied to January 1, 2011 returns March 4, 2010.

AddDate normalizes its result in the same way that Date does, so, for example, adding one month to October 31 yields December 1, the normalized form for November 31.

▶ Example

func (Time) After

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

After reports whether the time instant t is after u.

▶ Example

func (Time) AppendFormat   added in go1.5

```
func (t Time) AppendFormat(b []byte, layout string) []byte
```

AppendFormat is like Format but appends the textual representation to b and returns the extended buffer.

▶ Example

func (Time) Before

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

Before reports whether the time instant t is before u.

▶ Example

func (Time) Clock

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

Clock returns the hour, minute, and second within the day specified by t.

func (Time) Compare

```
func (t Time) Compare(u Time) int
```

Compare compares the time instant t with u. If t is before u, it returns -1; if t is after u, it returns +1; if they're the same, it returns 0.

func (Time) Date

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

Date returns the year, month, and day in which t occurs.

▶ Example

func (Time) Day

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

Day returns the day of the month specified by t.

▶ Example

func (Time) Equal

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

Equal reports whether t and u represent the same time instant. Two times can be equal even if they are in different locations. For example, 6:00 +0200 and 4:00 UTC are Equal. See the documentation on the Time type for the pitfalls of using == with Time values; most code should use Equal instead.

▶ Example

func (Time) Format

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

Format returns a textual representation of the time value formatted according to the layout defined by the argument. See the documentation for the constant called Layout to see how to represent the layout

format.

The executable example for Time.Format demonstrates the working of the layout string in detail and is a good reference.

▶ Example

▶ Example (Pad)

func (Time) GoString

```
func (t Time) GoString() string
```

GoString implements fmt.GoStringer and formats t to be printed in Go source code.

▶ Example

func (*Time) GobDecode

```
func (t *Time) GobDecode(data []byte) error
```

GobDecode implements the gob.GobDecoder interface.

func (Time) GobEncode

```
func (t Time) GobEncode() ([]byte, error)
```

GobEncode implements the gob.GobEncoder interface.

func (Time) Hour

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

Hour returns the hour within the day specified by t, in the range [0, 23].

func (Time) ISOWeek

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

ISOWeek returns the ISO 8601 year and week number in which t occurs. Week ranges from 1 to 53. Jan 01 to Jan 03 of year n might belong to week 52 or 53 of year n-1, and Dec 29 to Dec 31 might belong to week 1 of year n+1.

func (Time) In

```
func (t Time) In(loc *Location) Time
```

In returns a copy of t representing the same time instant, but with the copy's location information set to loc for display purposes.

In panics if loc is nil.

func (Time) IsDST

```
func (t Time) IsDST() bool
```

IsDST reports whether the time in the configured location is in Daylight Savings Time.

func (Time) IsZero

```
func (t Time) IsZero() bool
```

IsZero reports whether t represents the zero time instant, January 1, year 1, 00:00:00 UTC.

func (Time) Local

```
func (t Time) Local() Time
```

Local returns t with the location set to local time.

func (Time) Location

```
func (t Time) Location() *Location
```

Location returns the time zone information associated with t.

func (Time) MarshalBinary

```
func (t Time) MarshalBinary() ([]byte, error)
```

MarshalBinary implements the encoding.BinaryMarshaler interface.

func (Time) MarshalJSON

```
func (t Time) MarshalJSON() ([]byte, error)
```

MarshalJSON implements the json.Marshaler interface. The time is a quoted string in the RFC 3339 format with sub-second precision. If the timestamp cannot be represented as valid RFC 3339 (e.g., the year is out of range), then an error is reported.

func (Time) MarshalText

```
func (t Time) MarshalText() ([]byte, error)
```

MarshalText implements the encoding.TextMarshaler interface. The time is formatted in RFC 3339 format with sub-second precision. If the timestamp cannot be represented as valid RFC 3339 (e.g., the year is out of range), then an error is reported.

func (Time) Minute

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

Minute returns the minute offset within the hour specified by t, in the range [0, 59].

func (Time) Month

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

Month returns the month of the year specified by t.

func (Time) Nanosecond

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

Nanosecond returns the nanosecond offset within the second specified by t, in the range [0, 999999999].

func (Time) Round   added in go1.1

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

Round returns the result of rounding t to the nearest multiple of d (since the zero time). The rounding behavior for halfway values is to round up. If d <= 0, Round returns t stripped of any monotonic clock reading but otherwise unchanged.

Round operates on the time as an absolute duration since the zero time; it does not operate on the presentation form of the time. Thus, Round(Hour) may return a time with a non-zero minute, depending on the time's Location.

▶ Example

func (Time) Second

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

Second returns the second offset within the minute specified by t, in the range [0, 59].

func (Time) String

```
func (t Time) String() string
```

String returns the time formatted using the format string

```
"2006-01-02 15:04:05.999999999 -0700 MST"
```

If the time has a monotonic clock reading, the returned string includes a final field "m=±<value>", where value is the monotonic clock reading formatted as a decimal number of seconds.

The returned string is meant for debugging; for a stable serialized representation, use t.MarshalText, t.MarshalBinary, or t.Format with an explicit format string.

▶ Example

func (Time) Sub

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

Sub returns the duration t-u. If the result exceeds the maximum (or minimum) value that can be stored in a Duration, the maximum (or minimum) duration will be returned. To compute t-d for a duration d, use t.Add(-d).

▶ Example

func (Time) Truncate   added in go1.1

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

Truncate returns the result of rounding t down to a multiple of d (since the zero time). If d <= 0, Truncate returns t stripped of any monotonic clock reading but otherwise unchanged.

Truncate operates on the time as an absolute duration since the zero time; it does not operate on the presentation form of the time. Thus, Truncate(Hour) may return a time with a non-zero minute, depending on the time's Location.

▶ Example

func (Time) UTC

```
func (t Time) UTC() Time
```

UTC returns t with the location set to UTC.

func (Time) Unix

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

Unix returns t as a Unix time, the number of seconds elapsed since January 1, 1970 UTC. The result does not depend on the location associated with t. Unix-like operating systems often record time as a 32-

bit count of seconds, but since the method here returns a 64-bit value it is valid for billions of years into the past or future.

▶ Example

func (Time) UnixMicro added in go1.17

```
func (t Time) UnixMicro() int64
```

UnixMicro returns t as a Unix time, the number of microseconds elapsed since January 1, 1970 UTC. The result is undefined if the Unix time in microseconds cannot be represented by an int64 (a date before year -290307 or after year 294246). The result does not depend on the location associated with t.

func (Time) UnixMilli added in go1.17

```
func (t Time) UnixMilli() int64
```

UnixMilli returns t as a Unix time, the number of milliseconds elapsed since January 1, 1970 UTC. The result is undefined if the Unix time in milliseconds cannot be represented by an int64 (a date more than 292 million years before or after 1970). The result does not depend on the location associated with t.

func (Time) UnixNano

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

UnixNano returns t as a Unix time, the number of nanoseconds elapsed since January 1, 1970 UTC. The result is undefined if the Unix time in nanoseconds cannot be represented by an int64 (a date before the year 1678 or after 2262). Note that this means the result of calling UnixNano on the zero Time is undefined. The result does not depend on the location associated with t.

func (*Time) UnmarshalBinary added in go1.2

```
func (t *Time) UnmarshalBinary(data []byte) error
```

UnmarshalBinary implements the encoding.BinaryUnmarshaler interface.

func (*Time) UnmarshalJSON

```
func (t *Time) UnmarshalJSON(data []byte) error
```

UnmarshalJSON implements the json.Unmarshaler interface. The time must be a quoted string in the RFC 3339 format.

func (*Time) UnmarshalText added in go1.2

```
func (t *Time) UnmarshalText(data []byte) error
```

UnmarshalText implements the encoding.TextUnmarshaler interface. The time must be in the RFC 3339 format.

func (Time) Weekday

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

Weekday returns the day of the week specified by t.

func (Time) Year

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

Year returns the year in which t occurs.

func (Time) YearDay   added in go1.1

```
func (t Time) YearDay() int
```

YearDay returns the day of the year specified by t, in the range [1,365] for non-leap years, and [1,366] in leap years.

func (Time) Zone

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

Zone computes the time zone in effect at time t, returning the abbreviated name of the zone (such as "CET") and its offset in seconds east of UTC.

func (Time) ZoneBounds   added in go1.19

```
func (t Time) ZoneBounds() (start, end Time)
```

ZoneBounds returns the bounds of the time zone in effect at time t. The zone begins at start and the next zone begins at end. If the zone begins at the beginning of time, start will be returned as a zero Time. If the zone goes on forever, end will be returned as a zero Time. The Location of the returned times will be the same as t.

type Timer

```
type Timer struct {
    C <-chan Time
    // contains filtered or unexported fields
}
```

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.

func AfterFunc

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

AfterFunc waits for the duration to elapse and then calls f in its own goroutine. It returns a Timer that can be used to cancel the call using its Stop method.

func NewTimer

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

NewTimer creates a new Timer that will send the current time on its channel after at least duration d.

func (*Timer) Reset   added in go1.1

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

Reset changes the timer to expire after duration d. It returns true if the timer had been active, false if the timer had expired or been stopped.

For a Timer created with NewTimer, Reset should be invoked only on stopped or expired timers with drained channels.

If a program has already received a value from t.C, the timer is known to have expired and the channel drained, so t.Reset can be used directly. If a program has not yet received a value from t.C, however, the timer must be stopped and—if Stop reports that the timer expired before being stopped—the channel explicitly drained:

```
if !t.Stop() {
    <-t.C
}
t.Reset(d)
```

This should not be done concurrent to other receives from the Timer's channel.

Note that it is not possible to use Reset's return value correctly, as there is a race condition between draining the channel and the new timer expiring. Reset should always be invoked on stopped or expired channels, as described above. The return value exists to preserve compatibility with existing programs.

For a Timer created with AfterFunc(d, f), Reset either reschedules when f will run, in which case Reset returns true, or schedules f to run again, in which case it returns false. When Reset returns false, Reset neither waits for the prior f to complete before returning nor does it guarantee that the subsequent goroutine running f does not run concurrently with the prior one. If the caller needs to know whether the prior execution of f is completed, it must coordinate with f explicitly.

func (*Timer) Stop

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

Stop prevents the Timer from firing. It returns true if the call stops the timer, false if the timer has already expired or been stopped. Stop does not close the channel, to prevent a read from the channel succeeding incorrectly.

To ensure the channel is empty after a call to Stop, check the return value and drain the channel. For example, assuming the program has not received from t.C already:

```
if !t.Stop() {
    <-t.C
}
```

This cannot be done concurrent to other receives from the Timer's channel or other calls to the Timer's Stop method.

For a timer created with AfterFunc(d, f), if t.Stop returns false, then the timer has already expired and the function f has been started in its own goroutine; Stop does not wait for f to complete before returning. If the caller needs to know whether f is completed, it must coordinate with f explicitly.

type **Weekday**

```
type Weekday int
```

A Weekday specifies a day of the week (Sunday = 0, ...).

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

func (Weekday) **String**

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

String returns the English name of the day ("Sunday", "Monday", ...).

📄 Source Files

View all 🔗

format.go
format_rfc3339.go
sleep.go
sys_unix.go

tick.go
time.go
zoneinfo.go
zoneinfo_goroot.go

zoneinfo_read.go
zoneinfo_unix.go

📁 Directories

tzdata

Package tzdata provides an embedded copy of the timezone database.

Why Go

Use Cases

Case Studies

Get Started

Playground

Tour

Stack Overflow

Help

Packages

Standard Library

About Go Packages

About

Download

Blog

Issue Tracker

Release Notes

Brand Guidelines

Code of Conduct

Connect

Twitter

GitHub

Slack

r/golang

Meetup

Golang Weekly

Copyright

Terms of Service

Privacy Policy

Report an Issue

Google