JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#41

TIMERS AND TICKERS: THE PULSE OF GO

MASTERING TIME-BASED EVENTS IN CONCURRENT SYSTEMS

Beyond time.Sleep

Why Sleep is Not Enough

time.Sleep blocks the entire goroutine, making it impossible to respond to other signals (like a cancellation) while waiting.

The Concept:

To build responsive systems, we need non-blocking ways to wait. By using Channels with Timers, we can "listen" for a timeout while simultaneously listening for work or shutdown signals.

Precision Timing with time.Timer

The One-Shot Timer

A *time.Timer* sends a signal on its channel *C* exactly once after a specified duration. Unlike a simple sleep, a Timer can be stopped or reset before it fires.

```
// Initialize a timer for 2 seconds
timer := time.NewTimer(2 * time.Second)

// This blocks until the timer's channel receives the signal
// It's a more flexible alternative to time.Sleep
expirationTime := <-timer.C
fmt.Println("Timer expired at:", expirationTime)
```

Timer in a Select

Non-blocking Waits

The true power of a Timer is realized when paired with a *select* statement. This allows your goroutine to remain responsive to multiple sources.

```go
timer := time.NewTimer(5 * time.Second)
defer timer.Stop()

select {
case <-timer.C:
    fmt.Println("Timer fired!")
case <-ctx.Done():
    fmt.Println("Operation cancelled; stopping timer.")
    if !timer.Stop() {
        <-timer.C // Drain the channel if it already fired
    }
}
```

The Heartbeat Pattern

Recurring Events with *time.Ticker*

While a Timer fires once, a *time.Ticker* fires repeatedly at a set interval. This is the foundation for background tasks, health checks, and periodic cache flushes.

```go
// Create a ticker that ticks every 500 milliseconds
ticker := time.NewTicker(500 * time.Millisecond)

// Use a loop to receive values from the ticker channel
for i := 0; i < 3; i++ {
    t := <-ticker.C
    fmt.Println("Tick at", t)
}

// Crucial: stop the ticker to release associated resources
ticker.Stop()
```

Always remember to *Stop()* your tickers. A leaked ticker will continue to attempt to send values to its channel, wasting CPU cycles and memory even if the loop using it has exited.

The Ticker Loop

Orchestrating Background Work

Use a ticker inside a for-select loop to create a robust background worker that responds to system shutdowns.

```go
ticker := time.NewTicker(1 * time.Minute)
defer ticker.Stop()

for {
    select {
    case <-ctx.Done():
        return // Graceful exit
    case t := <-ticker.C:
        // Triggered every minute
        doPeriodicCleanup(t)
    }
}
```

The Callback Alternative

Efficiency with *time.AfterFunc*

Sometimes you don't want to manage a channel at all. *time.AfterFunc* runs a specific function in its own goroutine after a delay.

The Use Case:

This is highly efficient for "fire-and-forget" delayed logic, such as logging a warning if a process takes too long without blocking the main execution path.

```go
// Log a warning if the operation isn't done in 2 seconds
timer := time.AfterFunc(2*time.Second, func() {
    log.Println("Warning: Operation is exceeding threshold")
})
defer timer.Stop()

performHeavyWork()
```

The Performance Trap

time.After() vs. *time.NewTimer()*

time.After(d) is a convenient shortcut, but it has a hidden cost in high-frequency loops.

The Pattern:
time.After creates a new *Timer* every time it's called, but you cannot stop it manually. In a tight loop, this can lead to thousands of "ghost" timers sitting in memory until they expire.

Best Practice:
For long-running loops or performance-critical paths, always use *time.NewTimer* and manually *Stop()* or *Reset()* it.

Summary:

- Use Timers for one-off delays that need to be cancellable.
- Use Tickers for recurring "heartbeat" tasks.
- Use AfterFunc for efficient, channel-free callbacks.
- Avoid time.After in long-running loops to prevent memory bloat.

Next, we scale our concurrency: Worker Pools and the Fan-in / Fan-out pattern.