● JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#36

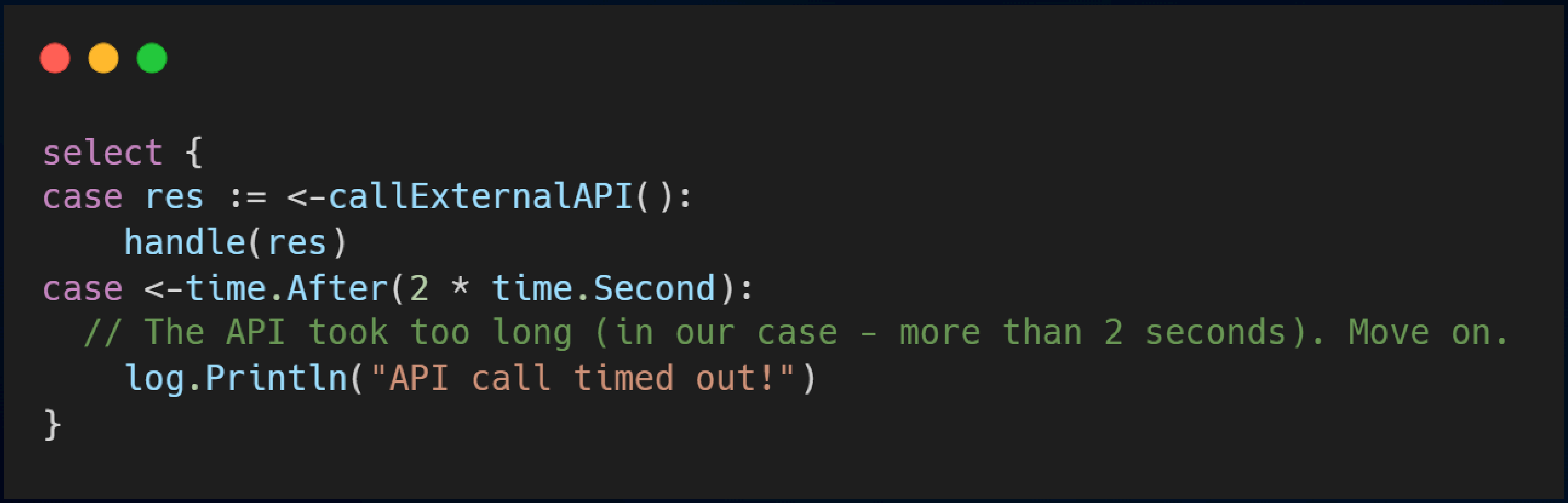
TIMEOUT PATTERN: SELECT + TIME.AFTER

PROTECTING YOUR SERVICE BOUNDARIES

The Async Alarm

The *time.After* Primitive

time.After(d) returns a channel that sends the current time after the duration *d* has passed. Combined with *select*, it creates a race between your logic and the clock.



```go
select {
case res := <-callExternalAPI():
    handle(res)
case <-time.After(2 * time.Second):
  // The API took too long (in our case - more than 2 seconds). Move on.
    log.Println("API call timed out!")
}
```

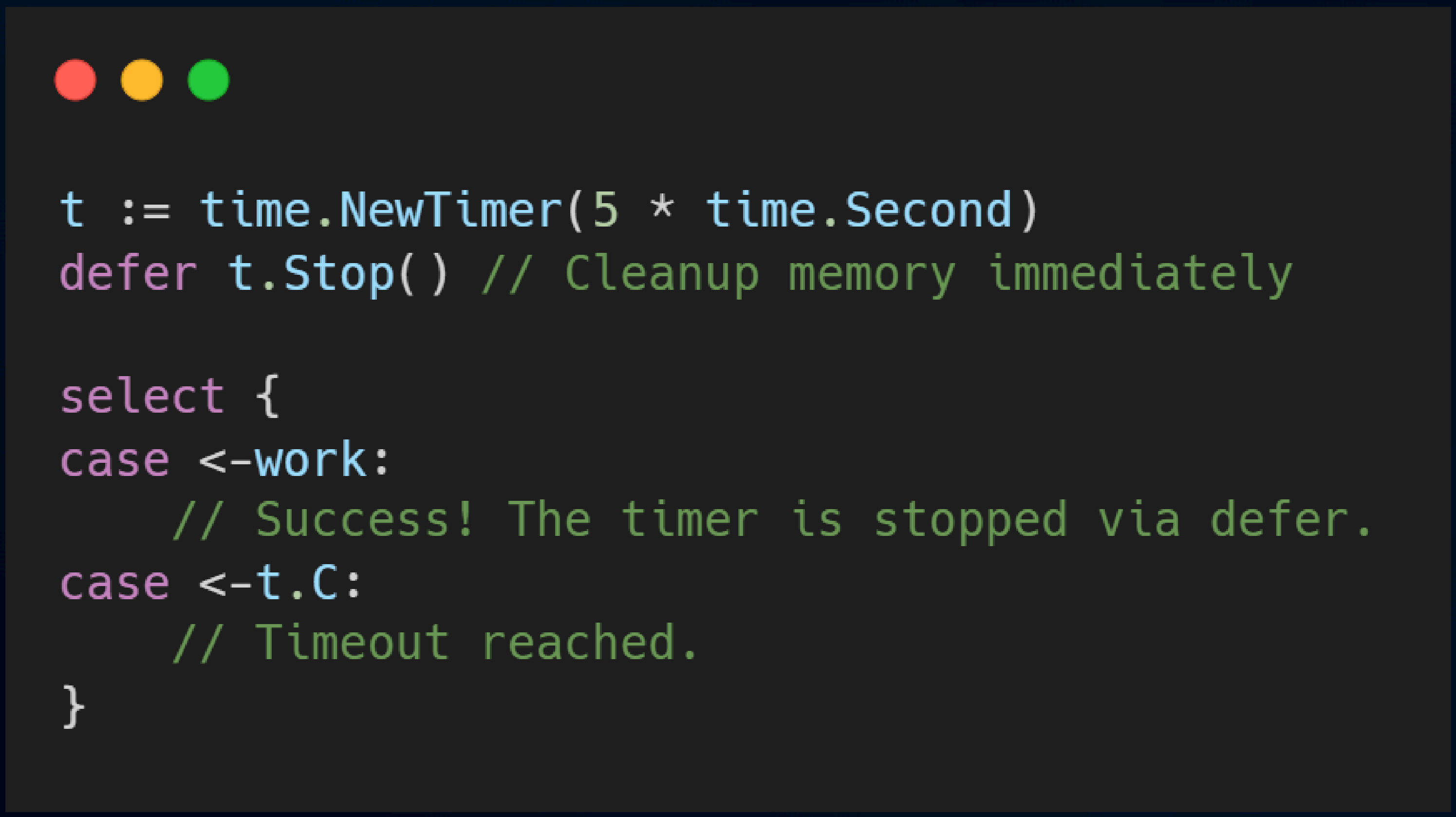
The Timer Leak Trap

Memory Leaks in Loops

Using *time.After* inside a *for* loop is a common senior-level mistake. Every call to *time.After* creates a new *time.Timer* object that stays in memory until it expires—even if the *select* case didn't use it.

The Fix:

Use *time.NewTimer* and manually *Stop()* it.



```go
t := time.NewTimer(5 * time.Second)
defer t.Stop() // Cleanup memory immediately

select {
case <-work:
    // Success! The timer is stopped via defer.
case <-t.C:
    // Timeout reached.
}
```

Timeout vs. Cancellation

Select vs. Context

While *time.After* is great for local logic, for modern Go development, you should usually prefer *context.WithTimeout*.

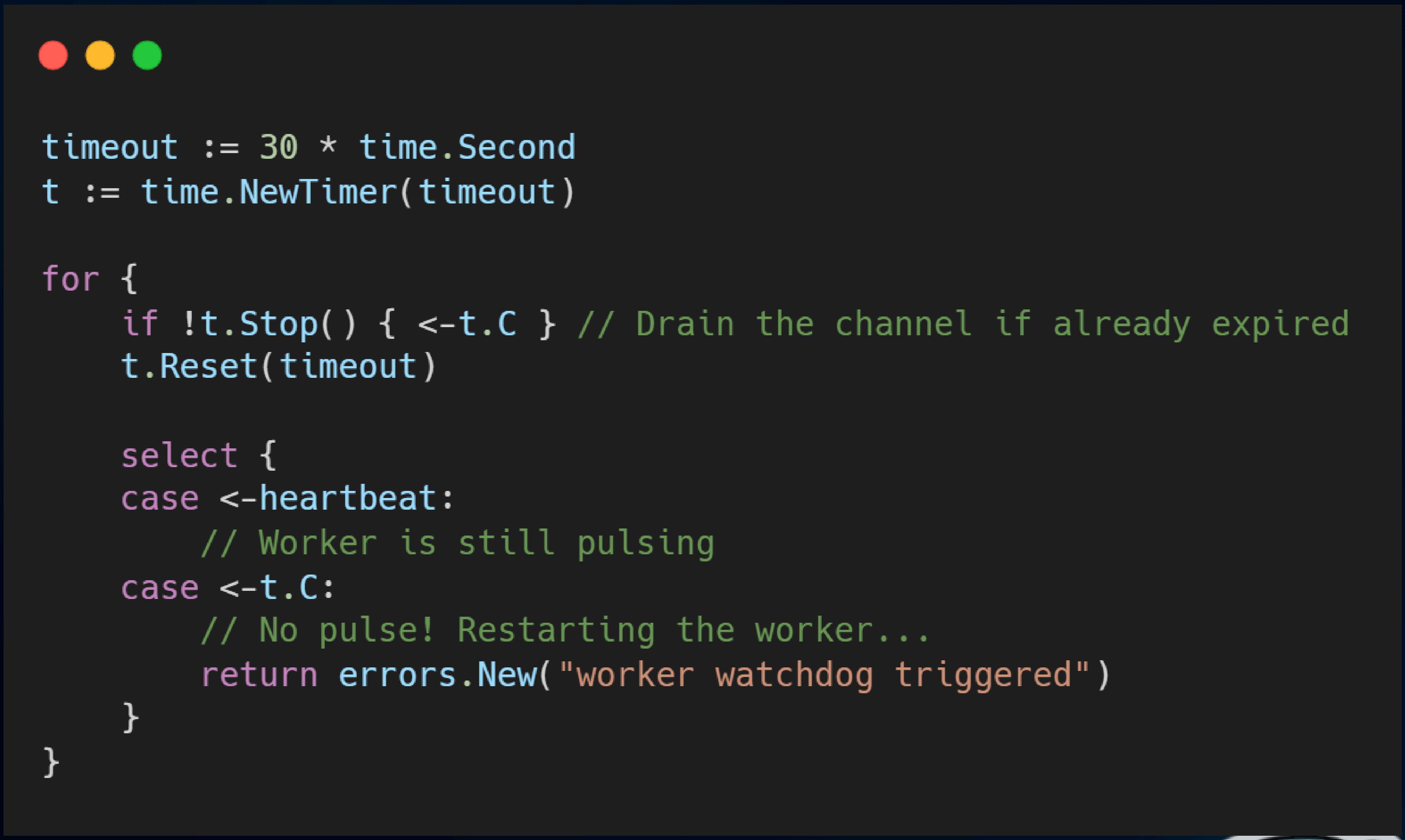
The Difference: *context* propagates the timeout down the call stack to other functions and libraries (like *database/sql*). *time.After* only protects the current *select* block.

Senior Rule: Use *select + time.After* for local coordination; use *Context* for request lifecycles.

Heartbeats & Watchdogs

Keeping a Worker Alive

You can use *select* to monitor if a background worker is still healthy by resetting a timer every time you receive a "heartbeat" signal.



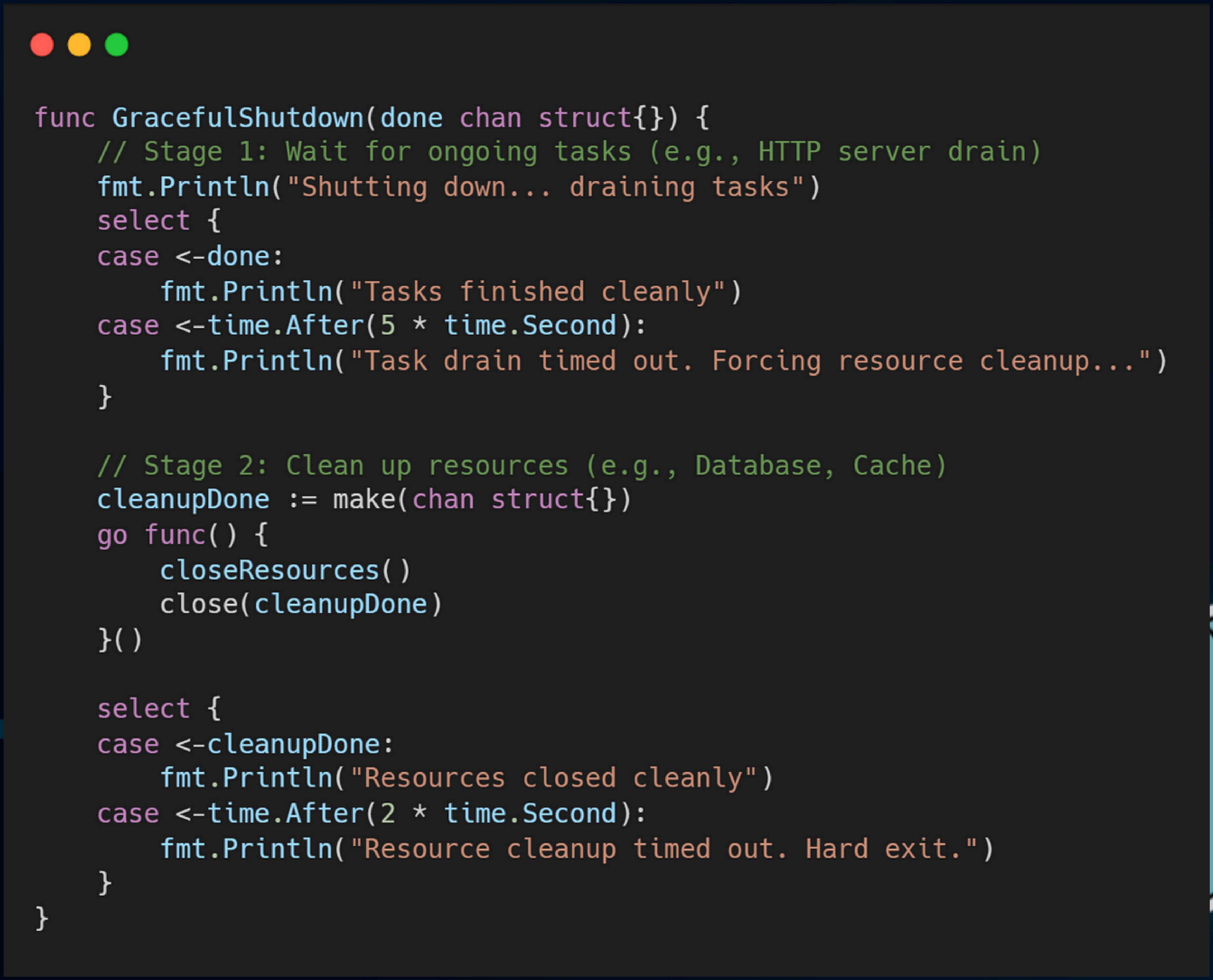
```go
timeout := 30 * time.Second
t := time.NewTimer(timeout)

for {
    if !t.Stop() { <-t.C } // Drain the channel if already expired
    t.Reset(timeout)

    select {
    case <-heartbeat:
        // Worker is still pulsing
    case <-t.C:
        // No pulse! Restarting the worker...
        return errors.New("worker watchdog triggered")
    }
}
```

Multi-Stage Timeouts

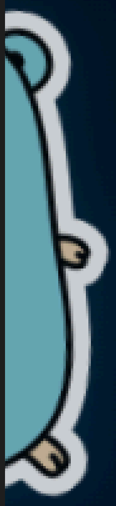
The "Grace Period" Pattern

During shutdown, you might want to give a service *x* seconds to finish current tasks, then *y* more seconds to close DB connections and other resources, then force-exit.

The Strategy:
Nesting select statements with different timers allows for "Graceful Degradation."



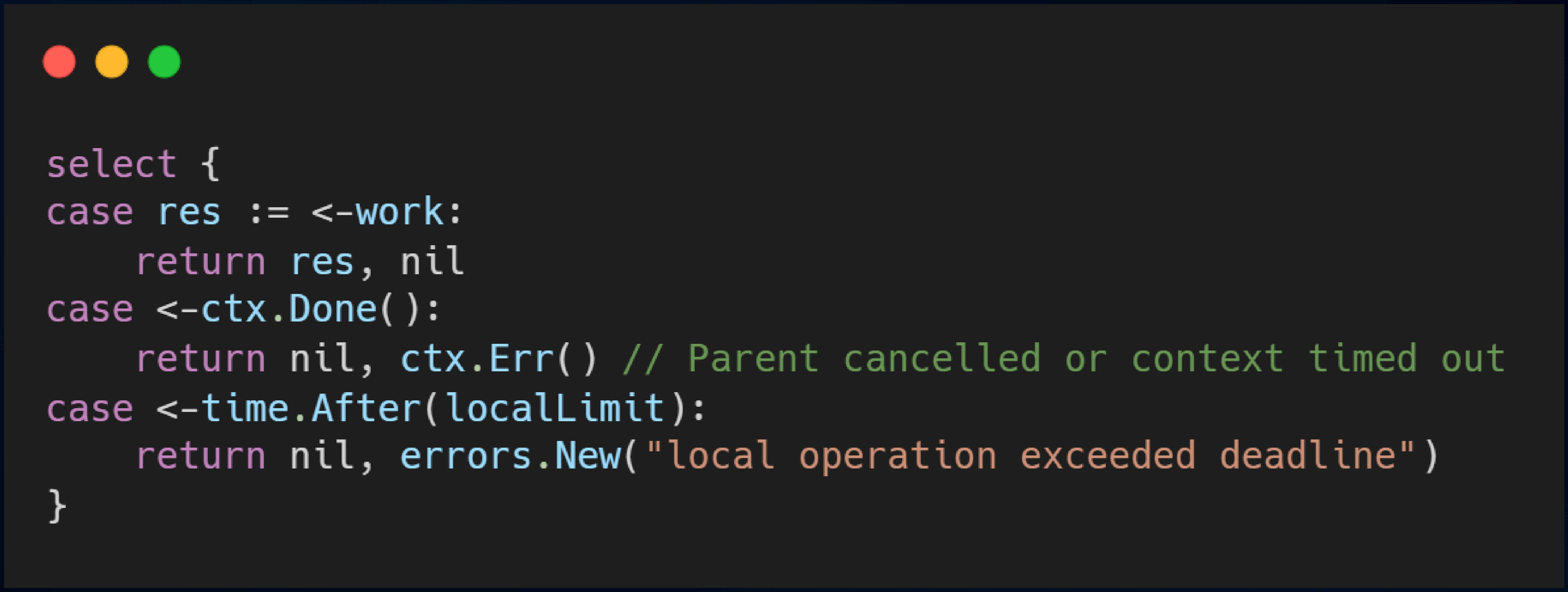
```go
func GracefulShutdown(done chan struct{}) {
    // Stage 1: Wait for ongoing tasks (e.g., HTTP server drain)
    fmt.Println("Shutting down... draining tasks")
    select {
    case <-done:
        fmt.Println("Tasks finished cleanly")
    case <-time.After(5 * time.Second):
        fmt.Println("Task drain timed out. Forcing resource cleanup...")
    }

    // Stage 2: Clean up resources (e.g., Database, Cache)
    cleanupDone := make(chan struct{})
    go func() {
        closeResources()
        close(cleanupDone)
    }()

    select {
    case <-cleanupDone:
        fmt.Println("Resources closed cleanly")
    case <-time.After(2 * time.Second):
        fmt.Println("Resource cleanup timed out. Hard exit.")
    }
}
```



The "First One Out" Logic

Combining *done* and *timeout*

Real-world workers often need to listen for the actual result, a user cancellation, and a hard timeout simultaneously.



```go
select {
case res := <-work:
    return res, nil
case <-ctx.Done():
    return nil, ctx.Err() // Parent cancelled or context timed out
case <-time.After(localLimit):
    return nil, errors.New("local operation exceeded deadline")
}
```

The Standard: No Naked Calls

Production Rule: Deadlines Everywhere

A "Naked Call" is any channel receive or I/O operation without a fallback.

The Rule:
Every *<-chan* from a source you don't control must be wrapped in a *select* with a timeout. This is the difference between a service that slows down under load and one that completely locks up.

Defend Concurrency!
Recap:

- *time.After* is a one-shot timeout for local blocks.
- Use *time.NewTimer* in loops to avoid memory bloat.
- Use *Context* for cross-boundary timeout propagation.
- Watchdogs are the best way to monitor long-running background tasks.

Tomorrow morning we pivot to shared memory. When is a channel the wrong tool? We dive into Mutex vs. RWMutex.