JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#25

MASTERING DEFER

DEFER: LIFO AND EXECUTION TIMING

The LIFO Guarantee

Last-In, First-Out

When you stack multiple *defer* calls, Go executes them in reverse order. This is intentional: dependencies are usually cleaned up in the opposite order they were created (e.g., Close File → Unlock Mutex).

```go
func main() {
    defer fmt.Println("First")  // Executed last
    defer fmt.Println("Second") // Executed middle
    defer fmt.Println("Third")  // Executed first
}
```

```
go run main.go
Third
Second
First
```

Always place your *defer* immediately after the resource was successfully initialized. Never *defer* before checking the *err*.

Execution Timing

The "After-Return" Hook

A deferred function is called after the surrounding function executes a *return* statement, but before it actually exits to the caller.

Think of *defer* as a wrapper around the return. It allows you to inspect or modify the state of the function at the very last millisecond of its life.

The Argument Evaluation Trap

Immediate Evaluation

This trips up many: The arguments to a deferred function are evaluated immediately, not when the function actually runs.

```go
func main() {
    start := time.Now()
    // This captures the value of 'start' NOW.
    defer fmt.Printf("Duration: %v\n", time.Since(start))

    time.Sleep(1 * time.Second) // Delay for 1 second, making expected time.Since(start) to ~1s
}
```

```
go run main.go
Duration: 0s
```

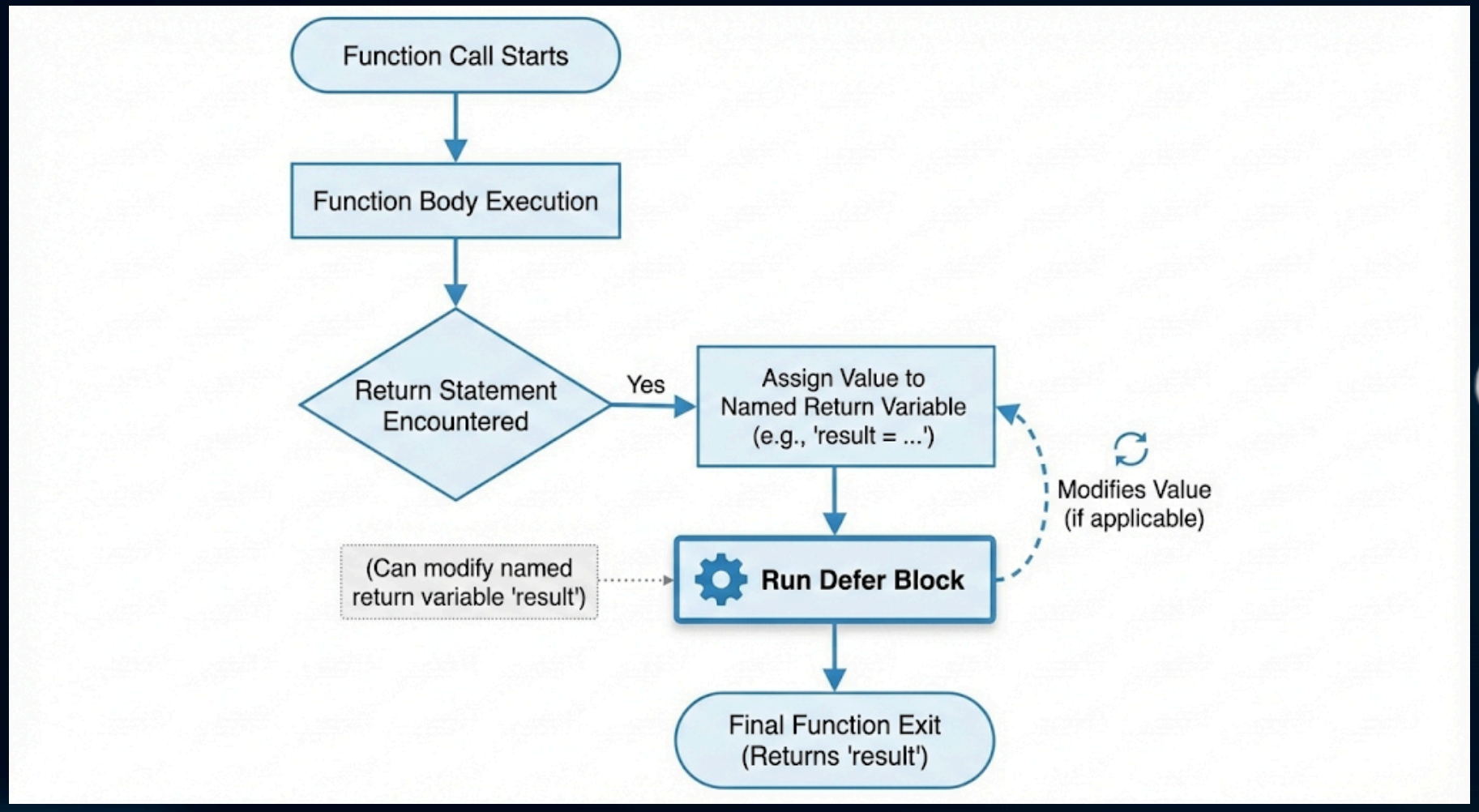
The Fix: Wrap it in an anonymous function so *time.Since* is evaluated at execution time.

Modifying Named Returns

The Power of Named Returns

defer can access and modify named return parameters. This is the only way to "reach back" and change a function's output after the logic has finished.

```go
func Save() (err error) {
    defer func() {
        if err != nil {
            log.Printf("failed to save: %v", err)
            err = fmt.Errorf("decorated error: %w", err)
        }
    }()
    return db.Execute() // If this fails, defer catches it
}
```



Performance: Stack vs. Heap

Is Defer "Slow"?

Historically, *defer* had a small overhead because it allocated on the heap. Since Go 1.14, the compiler uses "open-coded defers" for most cases, making them almost as fast as a direct function call.

Don't avoid *defer* for "performance" unless you are in a hot loop (millions of iterations). Readability and resource safety almost always outweigh the nanoseconds saved by manual cleanup.

The Defer-in-Loop Anti-Pattern

The Memory Leak Trap

Warning: *defer* does not execute at the end of a block (like a loop); it executes at the end of the function.

```go
// WARNING: This leaks file descriptors!
func ReadFiles(files []string) {
    for _, f := range files {
        res, _ := os.Open(f)
        defer res.Close() // Not called until the ENTIRE function ends
        // logic...
    }
}
```

The Fix: Wrap the loop body in an anonymous function or a separate helper function to ensure defer triggers every iteration.

```go
// THE FIX: Scope with an anonymous function
func ReadFiles(files []string) {
    for _, f := range files {
        func() {
            res, err := os.Open(f)
            if err != nil {
                return // moves to next iteration
            }
            // Defer is now tied to this anonymous func's scope
            defer res.Close()

            // Process file...
        }() // Executed immediately every loop
    }
}
```

Function Entry/Exit Tracing

A Practical Production Pattern

You can use a single *defer* line to log both the start and end of a complex operation.

```go
func trace(s string) string {
    fmt.Println("entering:", s)
    return s
}

func ComplexLogic() {
    defer fmt.Println("leaving:", trace("ComplexLogic"))
    // actual work here...
}

func main() {
    ComplexLogic()
}
```

```
go run main.go
entering: ComplexLogic
leaving: ComplexLogic
```

This pattern is invaluable for debugging synchronization issues or timing-sensitive logic in distributed systems.

GOLANG /// BA

Summary:

- Defers are LIFO.
- Arguments are evaluated at the time of the call.
- Use anonymous functions to modify named returns.
- **NEVER** use defer inside a long loop without a wrapper.

Question: Have you ever been bitten by a defer inside a loop, or a variable not updating because of immediate evaluation? Let’s share some "debugging war stories" below. 👇