● JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#26

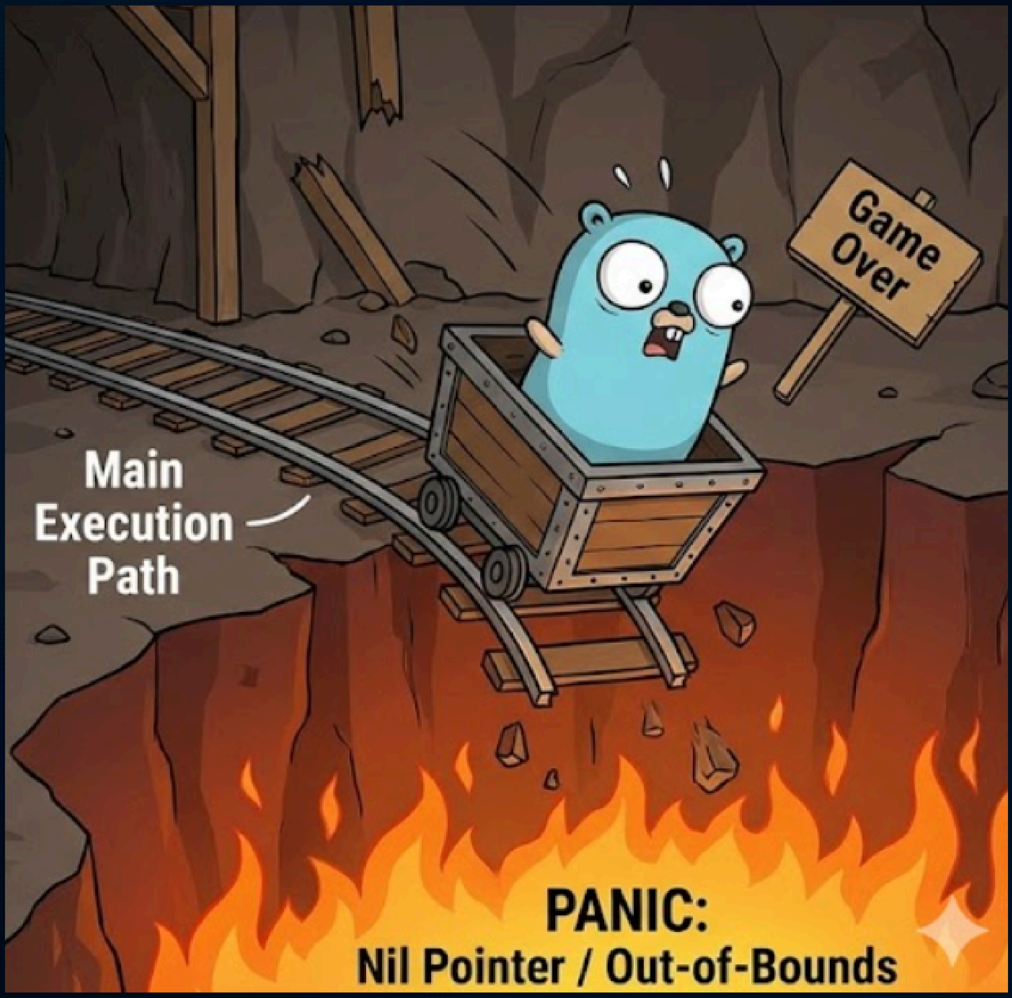
RECOVERY STRATEGIES

PANIC, RECOVER, AND MIDDLEWARE LOGGING

Not an Exception

What is Panic, Really?

Unlike *try/catch* in other languages, *panic* is for unrecoverable errors (e.g., out-of-bounds array access, nil pointer dereference).



Never use *panic* for control flow or expected errors (like "User Not Found"). Use it only when the program truly cannot continue in its current state.

Bringing it Back from the Dead

The Anatomy of *recover()*

recover() is a built-in function that regains control of a panicking goroutine.
The Crucial Catch: It only works inside a deferred function. If called elsewhere, it returns *nil* and has no effect.

```go
func SafeExecution() {
    defer func() {
        // recover() returns the interface{} passed to panic()
        if r := recover(); r != nil {
            // State is captured; the panic stops here.
            fmt.Printf("Rescued from: %v\n", r)
        }
    }()

    fmt.Println("Starting...")
    panic("unrecoverable state") // Control immediately jumps to defer
    fmt.Println("This never runs")
}
```

When you recover, the function doesn't "resume" where it left off. Execution stops at the *panic* line, the *defer* runs, and the function then returns to its caller. The caller sees the function exit normally (or with a zero-value return).

Panic and Goroutines

The "Process Killer" Trap

A *panic* in one goroutine will crash the entire application. A *defer/recover* in your *main()* function will NOT catch a panic that happens inside a separate *go func()*.

Every time you spin up a long-running goroutine, you must decide: "If this crashes, should the whole app die?", and usually, the answer is no. Every background worker needs its own internal recovery logic.

Can't try/catch in main here :)

Protecting HTTP Servers

Pattern: The Recovery Middleware

In production web servers, you use middleware to ensure a single bad request (causing a nil pointer) doesn't take down the whole API for everyone.

```go
func RecoveryMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        defer func() {
            if err := recover(); err != nil {
                // Log the stack trace and return 500
                log.Printf("PANIC: %v\n%s", err, debug.Stack())
                http.Error(w, "Internal Server Error", 500)
            }
        }()
        next.ServeHTTP(w, r)
    })
}
```

Capturing the Stack Trace

Context is Everything: *debug.Stack()*

A raw *recover()* only gives you the value passed to *panic()* (usually a string or an error). To actually fix the bug in production, you need the call stack - the sequence of function calls that led to the failure.

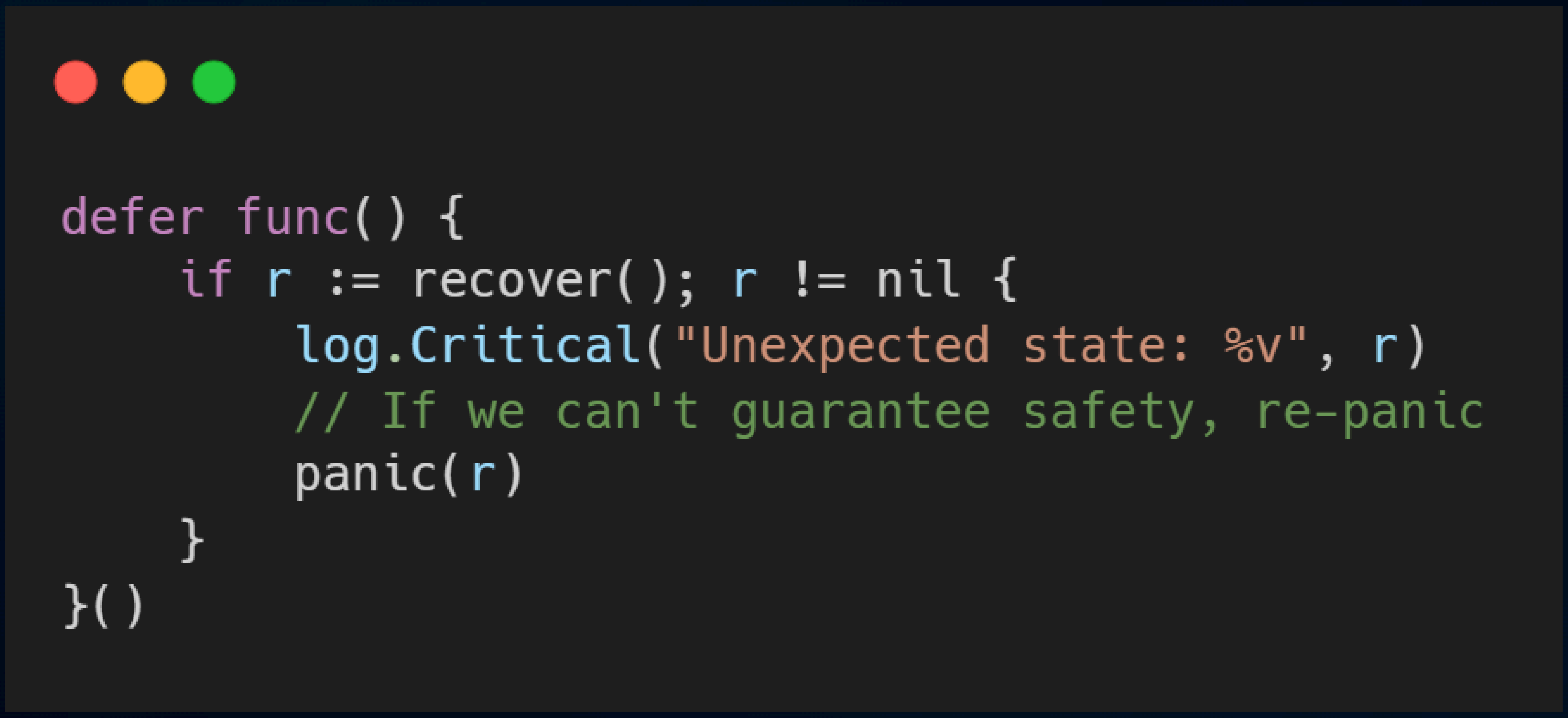
The Senior Fix: Use the *runtime/debug* package to capture the full trace. This transforms a "Something went wrong" log into a "Line 42 in auth.go failed" roadmap.

```go
import (
    "log"
    "runtime/debug"
)

func RecoverWithTrace() {
    defer func() {
        if r := recover(); r != nil {
            // debug.Stack() returns a []byte of the goroutine's stack trace
            log.Printf("CRITICAL: Panic recovered: %v\nStack Trace:\n%s", r, debug.Stack())
        }
    }()

    causePanic()
}
```

Re-Panicking

When to Let it Die

Sometimes you catch a panic, log it, and realize the state is too corrupted to continue (e.g., corrupted memory or lost DB connection).



```go
defer func() {
    if r := recover(); r != nil {
        log.Critical("Unexpected state: %v", r)
        // If we can't guarantee safety, re-panic
        panic(r)
    }
}()
```

Error vs. Panic

Design for Resilience

The Framework:

- **Errors**: Use for "Business Logic" failures (Validation, Auth, Not Found).
- **Panic**: Use for "System Integrity" failures (Developer bugs, missing required config).

Great Go code is "boring." It handles errors explicitly.
If you find yourself writing recover frequently, your architecture is likely hiding bugs rather than fixing them.

Resilience is a Choice.
Recap:

- *recover()* only lives inside defer.
- One unhandled goroutine panic kills the whole process.
- Always log the *debug.Stack()* in recovery.
- Use middleware to protect your boundaries.

Tomorrow, we move into the backbone of Go projects: Packages and Modules. How to version and manage dependencies like a pro.