JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#37

MUTEX VS. RWMUTEX

PROTECTING STATE WITHOUT KILLING THROUGHPUT

The Mutual Exclusion (Mutex)

The *sync.Mutex* Contract

A standard *sync.Mutex* is binary. It is either locked or unlocked. If one goroutine holds the lock, everyone else (both readers and writers) must wait in line.

```go
type Counter struct {
    mu    sync.Mutex
    value int
}

func (c *Counter) Inc() {
    c.mu.Lock()
    defer c.mu.Unlock() // Always defer immediately
    c.value++
}
```

Standard Mutexes are incredibly fast and should be your default. They have very low overhead. Use them when your "critical section" (the code between Lock and Unlock) is extremely short.

The Reader-Writer Mutex

Scaling the Reads

In many systems, like a configuration map or a session cache, 99% of the operations are reads and only 1% are writes. A standard Mutex is wasteful here because readers don't actually interfere with each other.

The Solution:
sync.RWMutex allows unlimited concurrent readers OR one single writer.

RLock vs. Lock

Implementing the RWMutex

Use *RLock()* when you are only reading. It allows other readers to proceed. Use *Lock()* when you are modifying. It blocks everyone.

```go
type SessionCache struct {
    mu   sync.RWMutex
    data map[string]string
}

func (s *SessionCache) Get(key string) string {
    s.mu.RLock()
    defer s.mu.RUnlock()
    return s.data[key]
}

func (s *SessionCache) Set(key, val string) {
    s.mu.Lock()
    defer s.mu.Unlock()
    s.data[key] = val
}
```

The RWMutex Performance Trap

Is *RWMutex* always faster?

Counter-intuitive Fact: A *RWMutex* is more complex than a standard *Mutex*. It has to maintain a reader count and handle "writer starvation" prevention logic.

If your read operation is tiny (e.g., just returning a struct field), a standard *Mutex* is often faster because its internal overhead is lower. Only switch to *RWMutex* if your profiling shows significant contention from readers.

No Reentrancy (The Deadlock)

Recursive Locking is Fatal

Go's Mutexes are not reentrant (unlike Java's *ReentrantLock*). If a goroutine tries to *Lock()* a Mutex it already holds, it will deadlock itself forever.

```go
type Counter struct {
    mu    sync.Mutex
    value int
}

func (c *Counter) Increment() {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.value++
}

func (c *Counter) Add(n int) {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.Increment() // DEADLOCK! Increment tries to Lock mu again.
}
```

The Fix: Move the logic into an unexported helper (e.g., *increment()*) that doesn't handle locking, and have both public methods call it.

Copying Locks is Forbidden

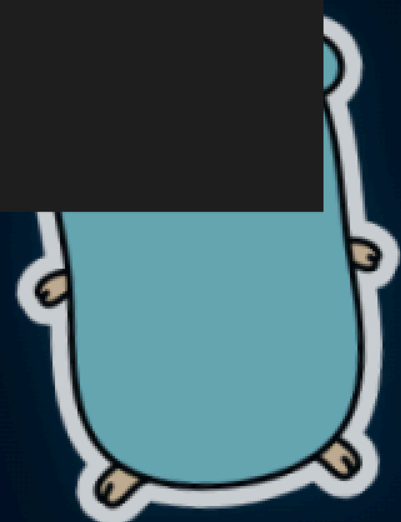
Passing the State, not the Lock

A *sync.Mutex* contains internal state (a waiter count). If you copy a struct that contains a Mutex, you copy that state.
Always pass structs with locks by pointer (*func (c *Counter)...*).

```go
type Counter struct {
    mu    sync.Mutex
    value int
}

// WRONG: Passed by value.
// Every call creates a NEW copy of the mutex. No actual protection!
func (c Counter) BadValue() int {
    c.mu.Lock()
    defer c.mu.Unlock()
    return c.value
}

// RIGHT: Pointer receiver.
// All calls share the same mutex instance.
func (c *Counter) GoodPointer() int {
    c.mu.Lock()
    defer c.mu.Unlock()
    return c.value
}
```



Lock Contention & Granularity

The "Big Lock" Anti-Pattern

Locking your entire application state with one global Mutex is a performance killer.

The Senior Move:
Fine-grained Locking. Instead of one lock for a map of 1,000,000 items, use a "Sharded Map" where 32 separate Mutexes protect different segments of the data. This drastically reduces the probability of two goroutines colliding.

Summary:

- *Mutex*: Fast, simple, default choice.
- *RWMutex*: Best for read-heavy, slow-read operations.
- No Reentrancy: Never lock twice in the same goroutine.
- Pointers only: Never copy a Mutex.

Question: Have you ever seen an RWMutex actually slow down an app compared to a regular Mutex? Why do you think that happened? 👇