● JANUARY 2026 SERIES

FROM GO BUILD
TO GO RUN

GOLANG 2026 - NIV RAVE

#08

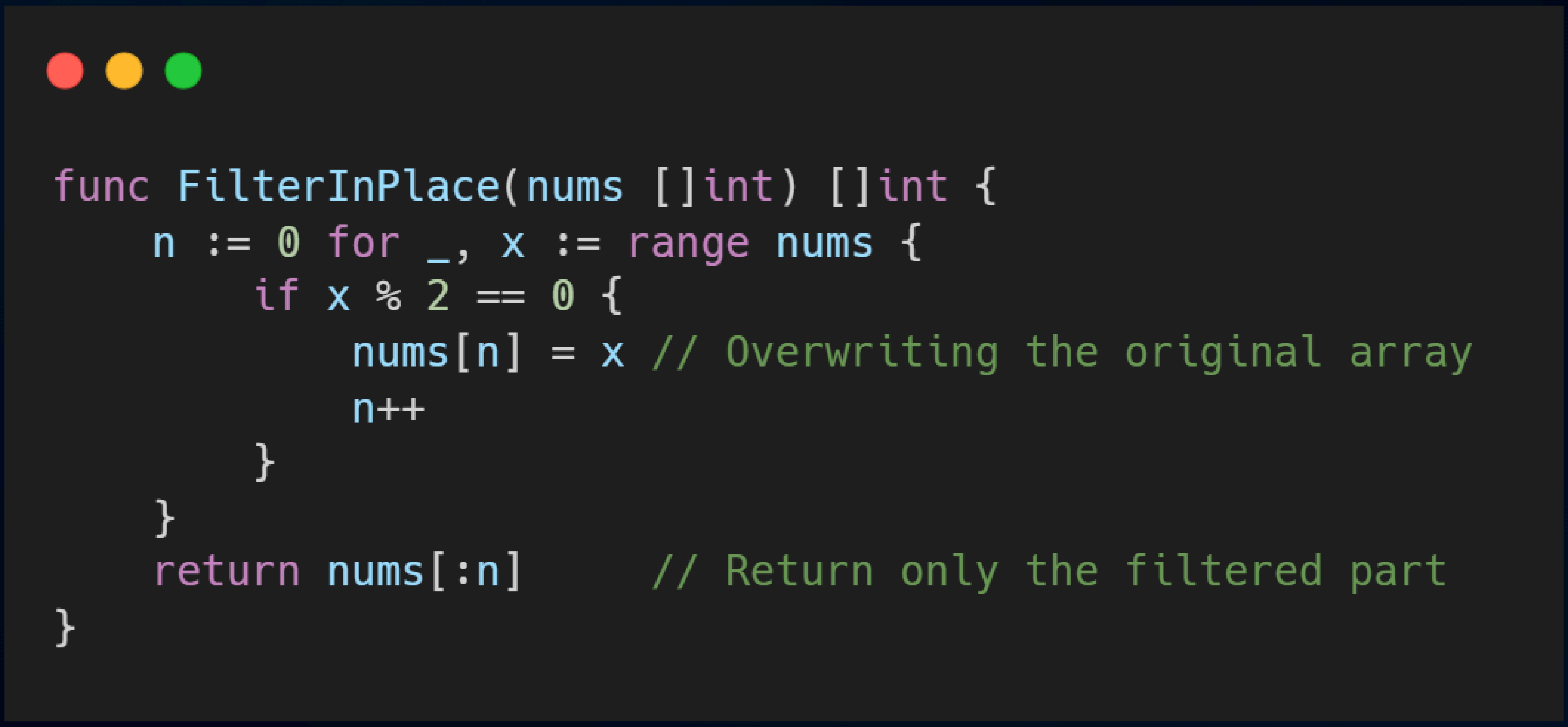
SLICE PERFORMANCE & MEMORY SAFETY

FROM DESTRUCTIVE FILTERING TO PROTECTING YOUR GARBAGE COLLECTOR

Zero-Alloc Filtering

The In-Place (Destructive) Filter

When the original input is "throwaway" data, don't allocate a new slice. Reuse the existing backing array to save memory.

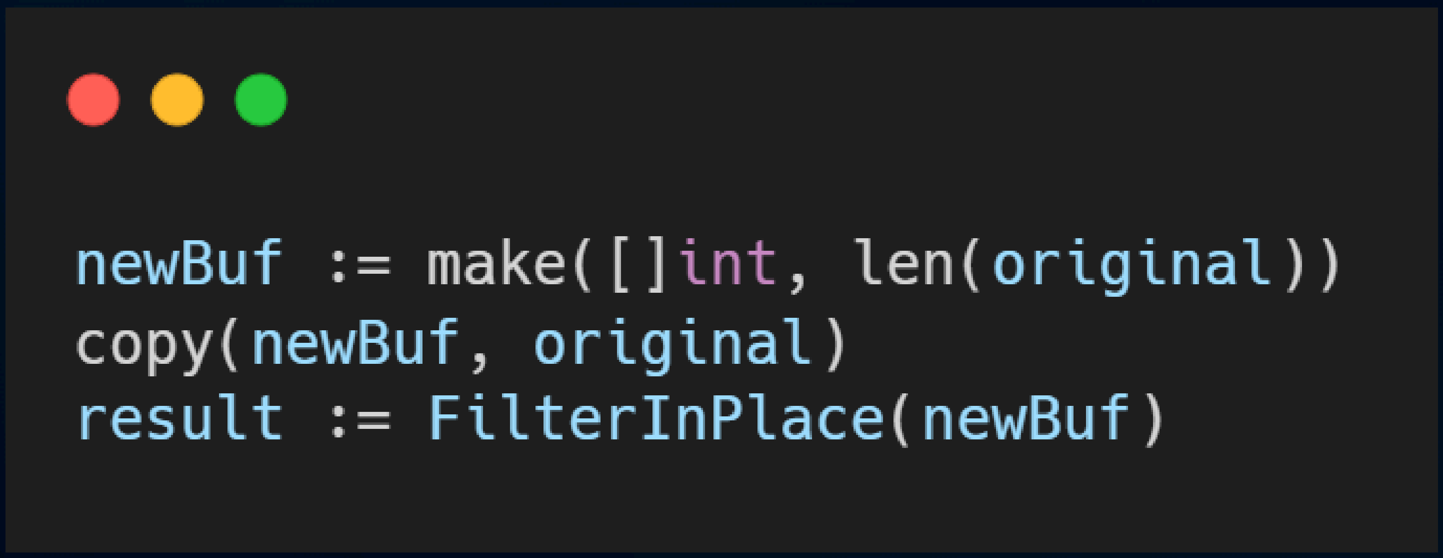


```go
func FilterInPlace(nums []int) []int {
    n := 0 for _, x := range nums {
        if x % 2 == 0 {
            nums[n] = x // Overwriting the original array
            n++
        }
    }
    return nums[:n]     // Return only the filtered part
}
```

The Copy Alternative

When Immutability Matters

If your original data must stay intact (e.g., it's being used by other goroutines), you must pay the allocation cost.



```go
newBuf := make([]int, len(original))
copy(newBuf, original)
result := FilterInPlace(newBuf)
```

This costs 1 allocation but ensures thread-safety and prevents side effects in other parts of your app.

The Append "Ghost" Bug

When Append Overwrites the Parent

If a sub-slice has extra capacity, append will overwrite the parent array's next index instead of allocating a new one.

```go
parent := []int{1, 2, 3, 4}
child := parent[:2] // len 2, cap 4
child = append(child, 99)
fmt.Println(parent) // Prints [1, 2, 99, 4] -> The "3" is GONE.
```

This is a silent logic bug that is incredibly hard to track down in complex systems.

The Defensive Fix

3-Index Slicing

The 3-index slicing syntax, known as a full slice expression, is a[low:high:max]. The primary purpose of the max index is to limit the capacity of the new slice, providing control over memory operations, especially when using the append function.

```go
array := [6]int{0, 1, 2, 3, 4, 5} // Original array: {0, 1, 2, 3, 4, 5}, cap: 6
s1 := array[1:4]                  // Standard 2-index slice, len(s1) = 3, cap(s1) = 5
s2 := array[1:4:4]                // 3-index slice, len(s1) = 3, cap(s1) = 3
// Appending to s1
s1 = append(s1, 10, 11) // Modifies the original array memory as it has capacity
fmt.Printf("s1 after append to s1: %v\n", s1)
fmt.Printf("s2 after append to s1: %v\n", s2)
mt.Printf("array after s1 append: %v\n", array) // array elements 4 and 5 were overwritten
// Appending to s2
s2 = append(s2, 20) // Forces new allocation because capacity is only 3
fmt.Printf("s1 after append to s2: %v\n", s1)
fmt.Printf("s2 after append to s2: %v\n", s2)
fmt.Printf("array after s2 append: %v\n", array) // Original array remains unchanged by s2's append
```

```
go run main.go
s1 after append to s1: [1 2 3 10 11]
s2 after append to s1: [1 2 3]
array after s1 append: [0 1 2 3 10 11]
s1 after append to s2: [1 2 3 10 11]
s2 after append to s2: [1 2 3 20]
array after s2 append: [0 1 2 3 10 11]
```

The Massive Array Leak

Holding 1GB for 10 Bytes

A slice header keeps the entire backing array reachable for the Garbage Collector.

```go
func getSmallDataFromHugeData(hugeData []byte) []byte {
    return hugeData[:10] // The 1GB backing array stays in memory!
}
```

The Fix: Use append([]byte{}, hugeData[:10]...) to force a new, tiny allocation and let the big one be freed by the GC.

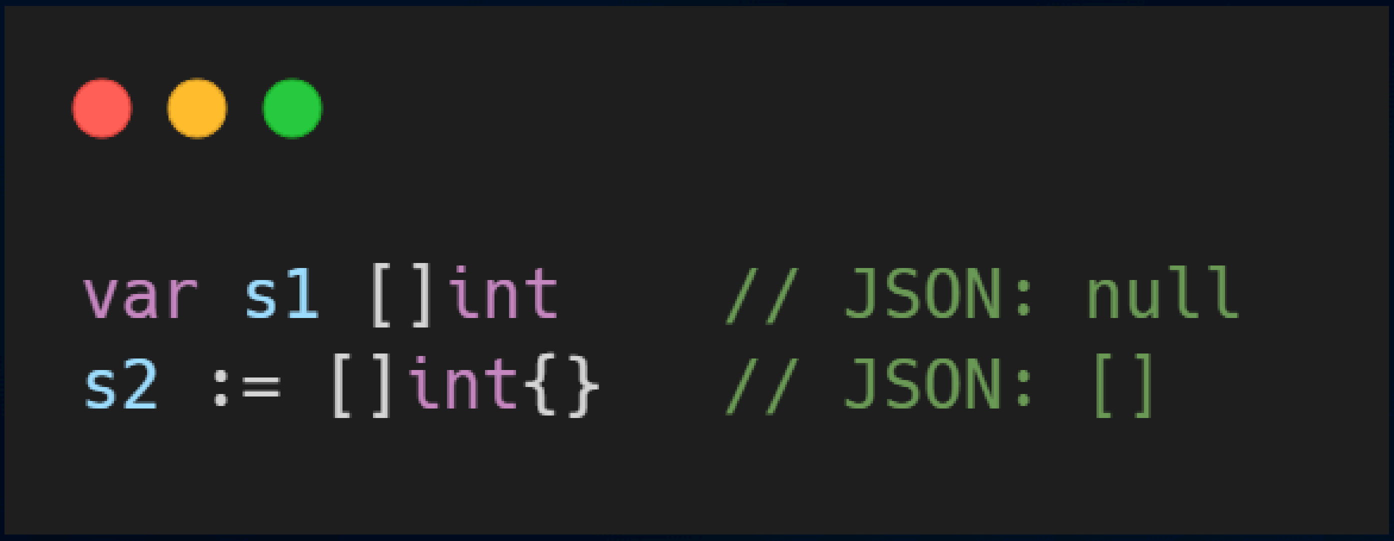
```go
append([]byte{}, hugeData[:10]...)
```

Breaking the Frontend

Nil vs. Empty (The API Contract)

How you declare a slice changes your API's JSON output.

- If your frontend uses .map() on the result, returning null (nil slice) might cause a crash. Use []T{} for empty results to be safe.



```go
var s1 []int     // JSON: null
s2 := []int{}    // JSON: []
```

Helping the Garbage Collector

GC Pressure: Values vs. Pointers

The memory layout of your slice determines how hard the GC has to work.

```
// 1. SLICE OF VALUES:
[]MyStruct // One contiguous block of memory. The GC scans it as a single object. High "Locality of Reference" = CPU
Cache friendly.

// 2. SLICE OF POINTERS:
[]*MyStruct // A list of memory addresses. The GC must "dereference" and scan every single pointer. If you have 1
million pointers, that's 1 million extra GC stops.
```

The Senior Decision:

- **GC Latency:** For large caches (100k+ items), Slices of Values are significantly faster. They reduce "Mark" phase time and keep "Stop the World" pauses short.
- **Stack vs. Heap:** Slices of values often allow data to stay "closer" together in memory, reducing the overhead of the Garbage Collector entirely.

The "When to use Pointers" Exception:

Use pointers if your structs are massive (>256 bytes) and the cost of copying them during appends/passes outweighs the GC scanning cost, or if you need to share mutable state across multiple slices.

Recap:

- Reuse arrays for speed (Filter In-Place).
- Use copy() or 3-index slicing for safety.
- Favor slices of values to keep the GC happy.
-

Do you prefer the speed of In-Place filtering or the safety of Copying? Let's talk about where you draw the line!