● JANUARY 2026 SERIES

FROM GO BUILD
TO GO RUN

GOLANG 2026 - NIV RAVE

#07

ARRAYS VS. SLICES

WHY WE (ALMOST) NEVER USE ARRAYS

The Array - Fixed & Rigid

The Fixed-Size Value Type

Arrays are allocated on the stack (usually).
Their size is a part of the type, set at compile-time and cannot change.

```go
func main() {
    var arr [5]int  // Size is part of the TYPE
    printSome(arr)  // Error: cannot use arr (variable of type [5]int) as [10]int value in argument to printSome
}

func printSome([10]int){
    fmt.Println("Got int array of size 10")
}
```

Passing a large array to a function copies every element. Passing a slice only copies the 24-byte header.

The Hidden Performance Trap

Arrays are Value Types

In Go, arrays are values, not pointers.
Warning: Passing a massive array to a function copies every single element onto the stack.

```go
a := [3]int{1, 2, 3}
b := a                // This creates a FULL COPY of the data
b[0] = 99             // Usually, expecting b[0]=a[0]

fmt.Println(a[0])     // Still prints 1
```

Use arrays when memory layout must be constant (for example [32]byte for a SHA256 hash).

The Slice - The Dynamic Window

The Anatomy of a Slice

A slice is actually a tiny, 24-byte struct (on 64-bit systems) made of 3 fields:
1. Pointer: Address of the first element in the backing array.
2. Len (Length): Number of elements currently in the slice.
3. Cap (Capacity): Total number of elements the backing array can hold.

```go
s := []int{}        // Create an "empty" slice

fmt.Println(len(s)) // 0
fmt.Println(cap(s)) // 0
fmt.Println(s)      // []
```

Understanding how Go manages slice memory is the difference between a clean backend and a memory-leaking nightmare.

Initializing with make

Pre-allocating with make()

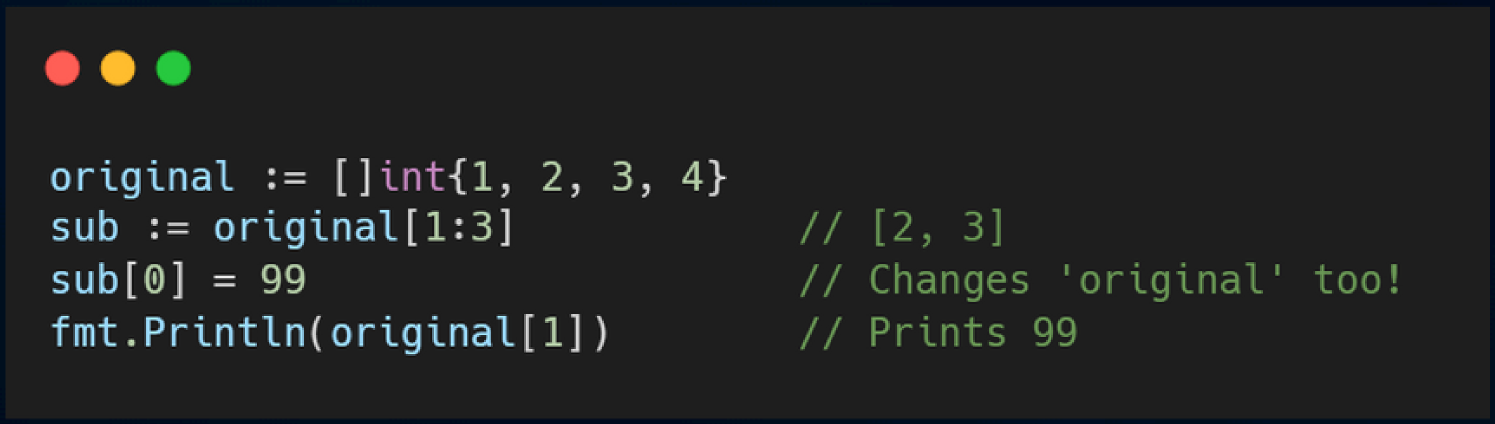
Using make with a capacity prevents the "Growth Penalty" (re-allocation). If you know you'll need 1000 items, allocate them upfront to avoid multiple background copies.

```go
// Pre-allocate using func make([]T, len, cap)

s := make([]int, 5, 10)

fmt.Println(len(s)) // 5
fmt.Println(cap(s)) // 10
fmt.Println(s)      // [0 0 0 0 0]
```

The "Parent" Relationship

Multiple Slices, One Array - The Shared Memory Trap

When you reslice an existing slice, you aren't copying data.
You are creating a new header pointing to the same memory



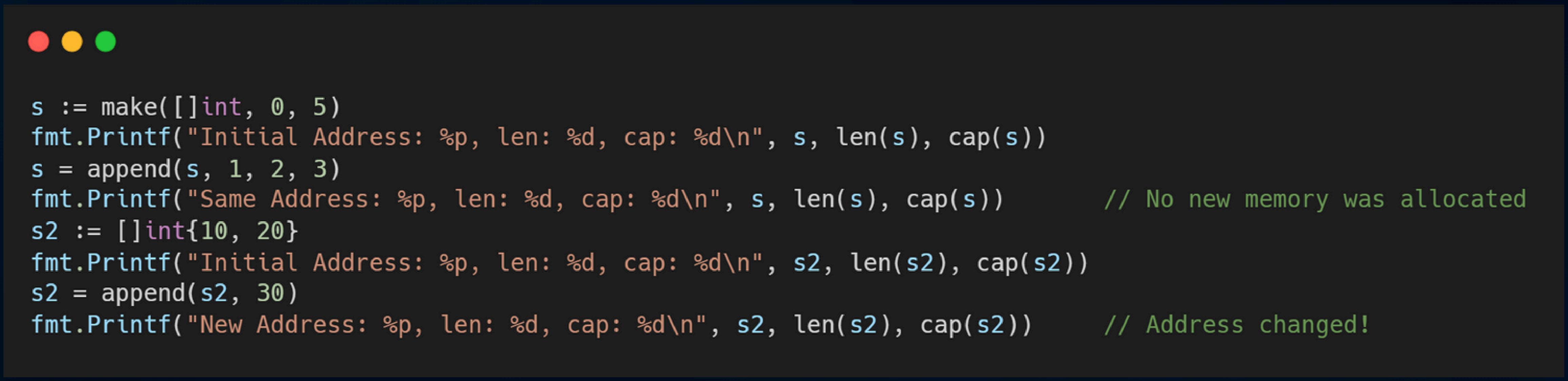
```go
original := []int{1, 2, 3, 4}
sub := original[1:3]           // [2, 3]
sub[0] = 99                    // Changes 'original' too!
fmt.Println(original[1])       // Prints 99
```

This is highly efficient for performance but dangerous for state management. If you need a completely independent list, you must use copy(), slices.Clone() or in complex scenarios - when the slice contains pointers, complex structs, etc. , using a manual implementation or a third-party library are required.

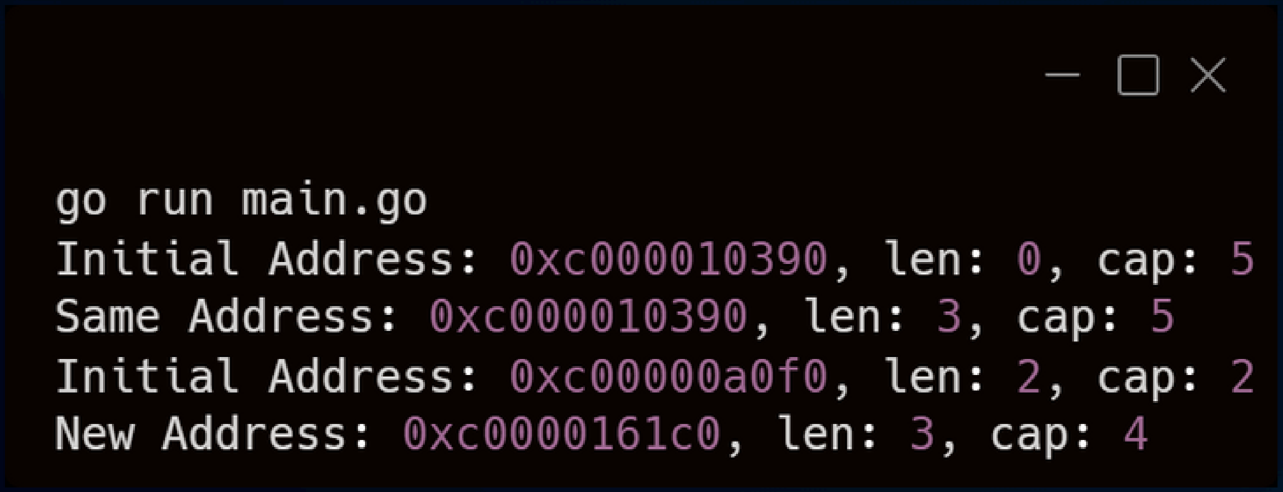
Slice Growth & Re-allocation

What happens during append()?

If len == cap and you append a new item:
Go allocates a NEW, larger backing array → It copies the old data to the new array → It updates the slice header's pointer.



```go
s := make([]int, 0, 5)
fmt.Printf("Initial Address: %p, len: %d, cap: %d\n", s, len(s), cap(s))
s = append(s, 1, 2, 3)
fmt.Printf("Same Address: %p, len: %d, cap: %d\n", s, len(s), cap(s))       // No new memory was allocated
s2 := []int{10, 20}
fmt.Printf("Initial Address: %p, len: %d, cap: %d\n", s2, len(s2), cap(s2))
s2 = append(s2, 30)
fmt.Printf("New Address: %p, len: %d, cap: %d\n", s2, len(s2), cap(s2))     // Address changed!
```



```
go run main.go
Initial Address: 0xc000010390, len: 0, cap: 5
Same Address: 0xc000010390, len: 3, cap: 5
Initial Address: 0xc00000a0f0, len: 2, cap: 2
New Address: 0xc0000161c0, len: 3, cap: 4
```

Declaration Cheat Sheet

Know Your Allocations

```
// 1. THE ARRAY (Value Type). Allocated on the stack. Fixed size.
var a [5]int
b := [...]int{1, 2, 3}

// 2. THE NIL SLICE (No Allocation). Best for: Default return values. Fields: ptr=nil, len=0, cap=0
var s []int

// 3. THE EMPTY SLICE (Allocated). Best for: Initializing JSON arrays to []. Fields: ptr=non-nil, len=0, cap=0
s := []int{}

// 4. THE PRE-ALLOCATED SLICE (Efficiency). Best for: Loops where count is known.
s := make([]int, 0, 100)
```

The JSON Trap: A nil slice (Example 2) marshals to null in JSON. An empty slice (Example 3) marshals to []. Use Example 3 when your frontend expects an array, not a null.

The Performance Win: Example 4 is your go-to for production. It allocates the backing array once, saving the CPU from expensive "grow-and-copy" operations during appends.

Recap & Tonight:

Consider when to use array/slice, Pre-allocate when possible, Watch your appends (Always capture the return of append), Watch out for shared-memory leaks.

Tonight we dive deeper into some advanced slice stuff.