JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#61

THE GO MEMORY MODEL & STRUCT ALIGNMENT

OPTIMIZING GO AT THE HARDWARE LEVEL

The Alignment Guarantee

Why Padding Happens

CPUs don't read memory one byte at a time. They read in "words" (usually 8 bytes on a 64-bit system).

To make these reads efficient, the Go compiler ensures that data is "aligned"—meaning a variable's memory address must be a multiple of its size.

If your data doesn't fit perfectly into these 8-byte boundaries, the compiler inserts "Padding" (empty, wasted bytes) to push the next field to a properly aligned address.

Padding is invisible in your code, but it is very real in your RAM. In high-performance AI systems or large-scale microservices, this "invisible waste" can add up to gigabytes of unnecessary memory usage.

Visualizing Struct Padding

The Order of Fields Matters

Take a look at two structs with the exact same fields. Because of how the compiler handles alignment, the order in which you define your fields determines how much padding is added.

```go
// Poor Alignment (Size: 24 bytes)
type BadStruct struct {
    A bool    // 1 byte
    // 7 bytes of padding here!
    B float64 // 8 bytes
    C int32   // 4 bytes
    // 4 bytes of padding at the end!
}
```

```go
// Optimized Alignment (Size: 16 bytes)
type GoodStruct struct {
    B float64 // 8 bytes (Fits perfectly)
    C int32   // 4 bytes
    A bool    // 1 byte
    // 3 bytes of padding at the end
}
```

The Rule of Thumb

Sort from Largest to Smallest

You don't need to manually calculate offsets every time. The most reliable way to minimize padding in Go is a simple rule: **Define your struct fields from the largest type to the smallest type**.

By putting your 8-byte types (*float64*, *int64*, pointers) first and your 1-byte types (*bool*, *uint8*) last, the compiler can pack the smaller types together into the remaining space of the last 8-byte word, significantly reducing waste.

The Final Field Trap

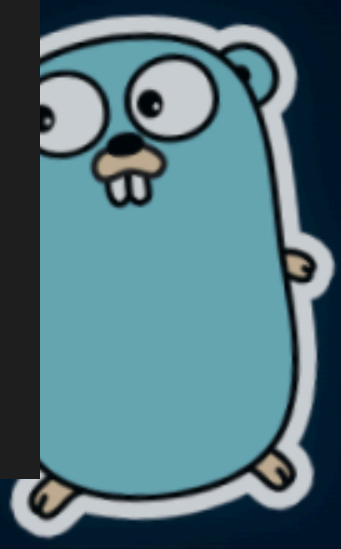
Zero-Sized Fields & False Sharing

There is a specific edge case in Go: a zero-sized field (like *struct{}* or an empty array) at the end of a struct. If a zero-sized field is the last element, the compiler will add padding to ensure it has its own unique address, preventing it from pointing "outside" the struct's memory.

The Fix:

If you use empty structs for signaling or markers, try to place them at the beginning or in the middle of your struct rather than at the very end to avoid that extra 8-byte padding penalty.

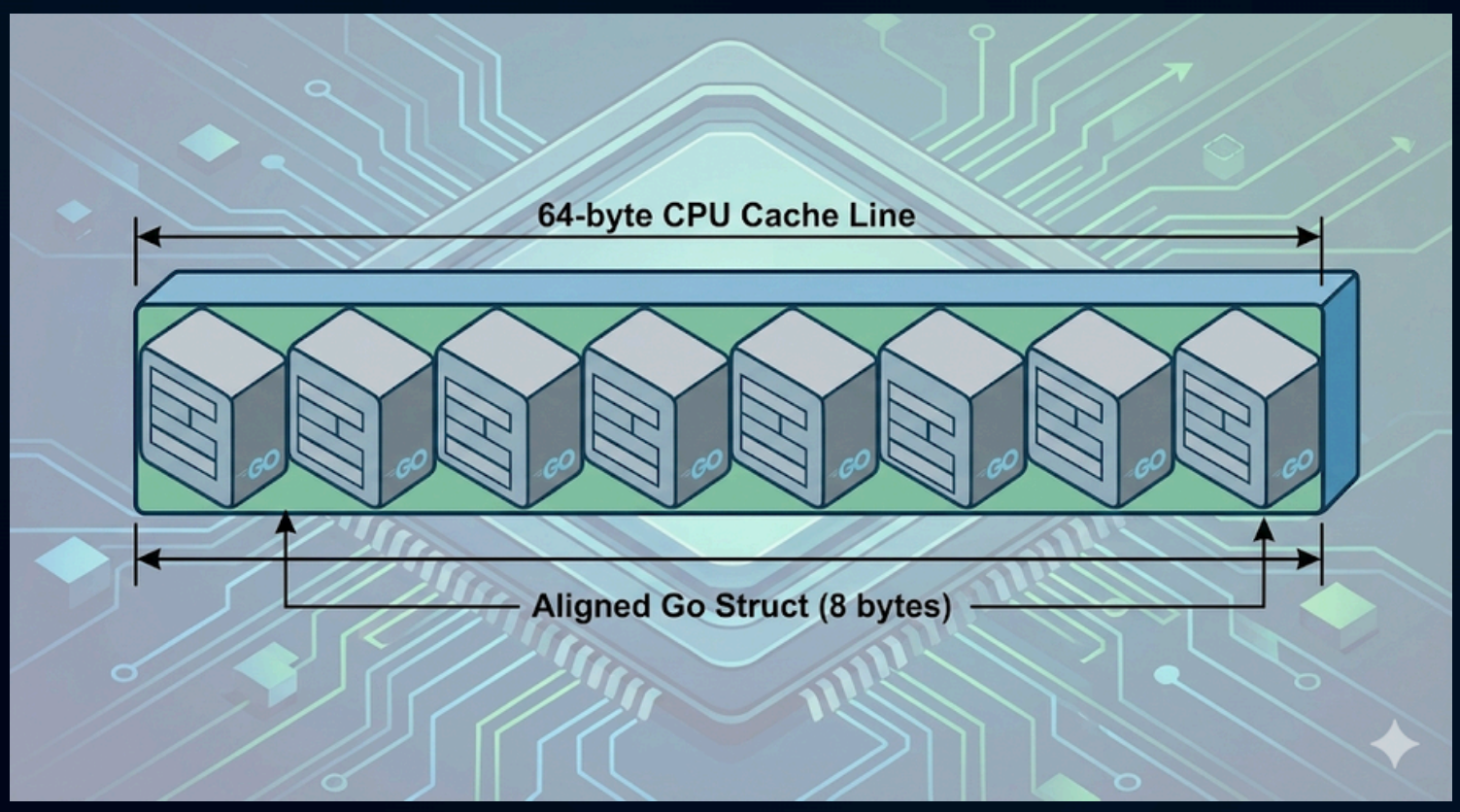
```go
// Sub-optimal: Last field adds 8 bytes of padding
type LastField struct {
    A int64
    B struct{} // Total size: 16 bytes
}

// Optimal: First field keeps it at 8 bytes
type FirstField struct {
    B struct{} // Size: 0 bytes (plus alignment)
    A int64    // Total size: 8 bytes
}

func main() {
    l := LastField{}
    f := FirstField{}

    fmt.Printf("LastField size: %d\n", unsafe.Sizeof(l))  // 16
    fmt.Printf("FirstField size: %d\n", unsafe.Sizeof(f)) // 8
}
```



Staying in the L1 Cache

Cache Locality: The Real Speed Limit

Memory alignment isn't just about saving bytes; it’s about Cache Locality. When the CPU fetches a variable, it fetches an entire "Cache Line" (usually 64 bytes). If your struct is small and well-packed, multiple instances of it can fit into a single cache line. This means the CPU can process your data without jumping back to the slow Main RAM. A Senior Go Engineer optimizes for the cache because it is often 100x faster than traditional memory access.



Measuring the Invisible

Checking Your Work with *unsafe*

How do you know if your optimization worked? Go provides the *unsafe* package (which we generally avoid in production logic) to inspect the memory layout of our types during development or profiling.

```go
import "unsafe"

func main() {
    b := BadStruct{}
    g := GoodStruct{}

    fmt.Printf("Bad size: %d\n", unsafe.Sizeof(b))   // Output: 24
    fmt.Printf("Good size: %d\n", unsafe.Sizeof(g))  // Output: 16

    // Check the offset of specific fields
    fmt.Printf("Offset of B: %d\n", unsafe.Offsetof(b.B)) // Output: 8
}
```

When Should You Optimize?

The Performance vs. Readability Balance

As a Senior, you must be pragmatic. Don't go back and re-order every struct in your codebase just for the sake of it.

Optimize when:

- You have a slice of millions of structs.
- You are building a high-frequency trading system or an AI data processor.
- Your pprof heap profile shows significant memory pressure.

For standard web handlers, prioritize readability and grouping related fields together. Only reach for memory alignment when the data proves you have a bottleneck.

The Machine-First Mindset. Recap:

- **Padding**: The compiler adds empty bytes to align data to word boundaries.
- **Ordering**: Always sort fields from largest to smallest for minimum waste.
- **Cache**: Small, packed structs stay in the CPU cache and run faster.
- **Tooling**: Use unsafe.Sizeof to verify your memory gains.

Tonight, we close the series with the final post: The Path to Senior Go Engineer (The Wrap).