JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#09

MAPS

BEYOND THE KEY-VALUE PAIR

The O(1) Mindset

The Constant Time Guarantee

Maps provide O(1) average time complexity for inserts, deletes, and lookups. While it's "Constant Time," the constant factor matters. A hash function must run for every lookup. If your keys are giant strings, that "O(1)" lookup might actually be slower than a linear search on a tiny slice.

Map Initialization

Pre-allocating for Performance

Just like slices, maps have an underlying data structure (buckets). If you know you're loading 1,000 items from a DB, pre-allocate the space to avoid expensive background re-hashes.

```go
// 1. Slow: Grows dynamically (Multiple re-allocations)
m := make(map[string]int)

// 2. Fast: Pre-allocated (Zero re-allocations)
m := make(map[string]int, 1000)
```

The "Comma OK" Idiom

Distinguishing "Zero" from "Missing"

In Go, accessing a missing key returns the zero-value of the type. Without the ok check, you cannot tell if the value was actually 0 or if the key was simply not there. Always use ok for defensive logic.

```go
val, ok := m["unknown_key"] if !ok {
    // Key truly does not exist
}
```

Maps are Reference Types

Shared State by Default

When you pass a map to a function, you are passing a pointer to the map header.

```go
func update(m map[string]int) {
    m["done"] = 1   // Calling update(myMap) modifies the original map!
}
```

This makes maps highly efficient to pass around, but dangerous for concurrent access.

The "Not Thread-Safe" Rule

The Fatal Error

Go maps do **not** support concurrent reads and writes.
If one goroutine writes while another reads, Go will throw a fatal error: concurrent map read and map write and crash the entire program.

```go
// 1. THE CRASH (Fatal Error)
m := make(map[string]int)
go func() {
    m["key"] = 1
}()
go func() {
    fmt.Println(m["key"])
}() // CRASH!

// 2. THE FIX (Thread-Safety)
type SafeMap struct {
    s sync.RWMutex
    data map[string]int
}

func (s *SafeMap) Get(key string) int {
    s.RLock() // Multiple readers allowed
    defer s.RUnlock()
    return s.data[key]
}

func (s *SafeMap) Set(key string, val int) {
    s.Lock() // Exclusive access for writing
    defer s.Unlock()
    s.data[key] = val
}
```

Use sync.Mutex or sync.RWMutex to guard map access in multi-threaded environments.

Map Keys — What is allowed?

The "Comparable" Requirement

Only types that can be compared using == can be map keys.

Allowed: strings, ints, floats, booleans, pointers, and channels.

NOT Allowed: Slices, Maps, or Functions.

Pro Tip: You can use a Struct as a key, provided all its fields are also comparable.

Deleting from a Map

The delete() Function

The Hidden Behavior: Deleting a key from a map is safe even if the key doesn't exist.

```go
m := make(map[string]int)
delete(m, "key")
```

Deleting elements does **not** shrink the map's memory usage. The map keeps the allocated buckets.

If you have a map that grew to 10GB and you delete everything, it still occupies 10GB. To free the memory, you must nil the map and let the GC handle it.

GOLANG /// BA

Recap & Tonight:

- Use make(map[T]V, hint) to avoid re-hashes.
- Always use the val, ok idiom for existence checks.
- Remember: Maps are NOT thread-safe.

Tonight we dive deeper into Map Internals: Buckets, Evacuation, and why iteration order is random.