JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#20

ADVANCED PATTERNS - GENERATORS & STREAMS

PROCESSING "INFINITE" DATA WITH ZERO MEMORY OVERHEAD

The Cost of Big Collections

The Slice Problem - Memory Bloat

In Go, a slice is a "Header" (on the stack) pointing to an underlying array (on the heap). As that array grows, the problems compound.

The "Hidden" Costs:

Contiguous Allocation: To store 100,000 structs, Go needs one single, unbroken block of memory. If the heap is fragmented, finding that block becomes expensive.

The GC "Scan" Tax: The Garbage Collector must scan every pointer in your slice to see if the data it points to is still alive. A slice of 1 million pointers creates a massive "Scan Object" that increases GC latency.

The Growing Pains: When a slice is full and you append, Go allocates a new array (usually double the size) and copies everything over. This causes a sudden spike in memory usage that can trigger an OOM (Out Of Memory) crash.

The Senior Take:

Allocation Spikes: If your service uses 500MB normally but spikes to 1.5GB every time a specific report is generated, your slices are likely reallocating.

The Fragmentation Trap: Large, long-lived slices can lead to "Heap Fragmentation," where you have plenty of total memory, but no single hole big enough for a new large slice.

The Solution: Instead of "Collect all, then process," we move to "Produce one, process one." This keeps our heap usage predictable and the GC happy.

The Generator Pattern

Encapsulating State via Closures

The Generator pattern allows us to create a "lazy" data source. Instead of pre-calculating a million results and storing them in a slice (which is a Heap allocation), we return a closure that carries its own private state on the Stack.

Zero Global State: The *current* variable is completely unreachable from outside the generator. This makes it thread-safe to create multiple independent generators (they don't share memory).

Escape Analysis: Because the closure "outlives" the *IntGenerator* function, the Go compiler will likely move *current* to the Heap. However, because it's a single integer rather than a 10,000-element slice, the GC overhead is negligible.

Lazy Evaluation: We only do the work (incrementing, fetching from a DB, or parsing) at the exact moment the consumer asks for it. This is the foundation of Streaming Architecture.

```go
func IntGenerator(start, end int) func() (int, bool) {
    // 'current' is captured by the closure below.
    // As long as the generator function exists, 'current' stays alive.
    current := start

    return func() (int, bool) {
        if current > end {
            return 0, false // Sentinel value for "Done"
        }
        val := current
        current++ // State is mutated internally
        return val, true
    }
}
```

Consuming the Stream

The Streaming Loop

We use a simple *for* loop to pull data from the generator until the "ok" signal is *false*.

```go
func main() {
    // next is a closure that maintains its own state
    next := IntGenerator(1, 1000000)

    for {
        val, ok := next()
        if !ok {
            break
        }

        // Logic happens here. Memory usage stays flat.
        process(val)
    }
}
```

The Benefit: It doesn't matter if the range is 1 to 10 or 1 to 10 billion, your memory footprint remains constant because you only ever hold one integer on the stack at a time.

The "Scanner" Interface

Abstracting the Stream

Recalling our Interface lessons, we can wrap this logic in a standard interface to make it swappable.

```go
type Scanner interface {
    Next() bool
    Value() string
}

// Any data source (SQL, File, API) can implement this
func ProcessStream(s Scanner) {
    for s.Next() {
        fmt.Println(s.Value())
    }
}
```

Real-World Connection: This is exactly how *bufio.Scanner* and *sql.Rows* work. They don't give you the whole dataset; they give you a way to move through it.

Transformation Pipelines

Chaining Behavior: The Pipeline Pattern

The power of generators is Composability. You can "wrap" one generator inside another to create a processing pipeline where data flows through functions (Stack) instead of sitting in memory buffers (Heap).

```go
// 1. The Filter Stage: Only allows even numbers through
func FilterEven(next func() (int, bool)) func() (int, bool) {
    return func() (int, bool) {
        ...
    }
}

// 2. The Map Stage: Multiplies incoming numbers by 10
func Multiply(next func() (int, bool), factor int) func() (int, bool) {
    return func() (int, bool) {
        ...
    }
}

func main() {
    // Chaining the pipeline: Source -> Filter -> Map
    source := IntGenerator(1, 10)
    evens := FilterEven(source)
    pipeline := Multiply(evens, 10)

    for {
        val, ok := pipeline()
        if !ok { break }
        fmt.Println(val) // Output: 20, 40, 60, 80, 100
    }
}
```

Maintainability: You've decoupled the generation logic from the filtering and transformation logic. This is the Open/Closed Principle in action.

Large File Processing

Streaming vs. Reading

If the input size is unknown or potentially large, never use a slice. Always use a stream.

```go
// THE HEAP TAX: os.ReadFile
// Loads the entire 2GB file into a contiguous block on the heap.
data, _ := os.ReadFile("production_logs.txt")

// THE SENIOR WAY: bufio.Scanner
// Reads one line into a small, reusable buffer.
file, _ := os.Open("production_logs.txt")
defer file.Close()

scanner := bufio.NewScanner(file)
for scanner.Scan() {
    line := scanner.Text() // Only one line in memory at a time
    process(line)
}
```

The Decision Matrix

Slices vs. Generators: The Architectural Trade-off

Choose patterns because they fit the hardware and the use case. Here is how to decide between contiguous memory (Slices) and lazy evaluation (Generators).

Feature	Use Slices (Contiguous)	Use Generators (Lazy)
Data Volume	Known, small, or bounded.	Massive, infinite, or unknown.
CPU Cache	**Excellent.** High spatial locality; faster for simple math.	**Lower.** Function call overhead (indirect jumps) at every step.
Memory Pressure	High. Large slices can trigger OOM or long GC pauses.	**Minimal.** Constant memory footprint (O(1) space).
Concurrency	Easier to share (read-only) or partition.	harder. Usually bound to a single consumer goroutine.
API Complexity	Simple. []string is the "Gold Standard" of Go.	Higher. Requires custom interfaces or function signatures.
Early Exit	wasteful. You've already loaded everything into RAM.	**Optimized.** Stops processing the moment the consumer stops.

Summary:

- Slices fill the Heap; Streams flow through the Stack.
- Use Closures to track iteration state safely.
- Generators keep your memory usage predictable and your p99s low.

Tomorrow we enter Error philosophy, one of my favorite things in Go :)