JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#33

THE ANATOMY OF CHANNELS

SYNCHRONIZATION VS. BUFFERING

Unbuffered Channels

Zero Capacity, Total Sync

An unbuffered channel (*make(chan int)*) has no storage. A "send" operation blocks until a "receiver" is ready, and vice versa.

This is a guaranteed hand-off. When the sender unblocks, they know with 100% certainty that the receiver has the data.

```go
ch := make(chan string)

go func() {
    ch <- "data" // Blocks until main() is ready
    fmt.Println("Sent!")
}()

fmt.Println(<-ch) // Blocks until goroutine sends
```

Buffered Channels

Asynchronous Communication

A buffered channel (*make(chan int, 10)*) has a fixed-size internal ring buffer. The sender only blocks when the buffer is full; the receiver only blocks when the buffer is empty.

```go
// Create a channel with a buffer size of 2
ch := make(chan string, 2)

// These happen immediately; no receiver needed yet
ch <- "message 1"
ch <- "message 2"

fmt.Println("Buffer is full, but sender is NOT blocked")

// This third send would block until a receiver takes an item
// ch <- "message 3"

fmt.Println(<-ch) // Takes "message 1", buffer now has 1 slot free
```

The Trade-off: With unbuffered channels, you have synchronization. With buffered channels, you have decoupling. Use buffers when you want to handle "bursty" traffic without slowing down the producer's main loop.

Choosing the Buffer Size

The "Buffer Sizing" Trap

You shouldn't just "guess" a buffer size. Use:
Size 0 (Unbuffered): Use for strong synchronization.
Size 1: Use for "signal" channels or single-item handoffs.
Size N: Use to smooth out "bursty" traffic.

If you need a buffer larger than 100, you are likely trying to solve a bottleneck by hiding it in memory.
A full buffer just moves the blocking problem elsewhere.

Restricting Behavior

Channel Directionality (Type Safety)

You can define if a function is allowed to only send to or only receive from a channel. This prevents logic bugs where a consumer accidentally tries to close a channel.

```go
// Only allowed to SEND to the channel
func produce(ch chan<- int) {
    ch <- 42
}

// Only allowed to RECEIVE from the channel
func consume(ch <-chan int) {
    fmt.Println(<-ch)
}
```

Closing Channels Safely

The "Close" Protocol

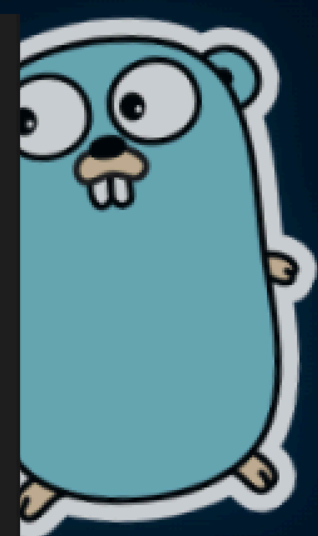
Only the sender should close a channel. Sending to a closed channel causes a panic. Closing a channel that is already closed causes a panic.

```go
func producer(ch chan<- int) {
    defer close(ch) // Always ensure closure to unblock receivers
    for i := 0; i < 5; i++ {
        ch <- i
    }
}

func consumer(ch <-chan int) {
    // The 'range' loop automatically exits when the channel is closed
    // AND all existing values have been consumed.
    for val := range ch {
        fmt.Printf("Received: %d\n", val)
    }
    fmt.Println("Channel closed, cleaning up...")
}
```

The "Comma Ok" Idiom: If you aren't using a range loop, you must manually check if the channel was closed to avoid processing "garbage" zero-values.

```go
val, ok := <-ch
if !ok {
    // ok is false if the channel is closed and empty
    return
}
// process val
```



The "Signal" Pattern

Beyond Data: The Signal Channel

Channels aren't just for moving data; they are for controlling state. A *chan struct{}* is the most efficient way to broadcast a "Stop" or "Ready" signal to 1,000 goroutines at once with zero memory overhead.

Why *struct{}*? It occupies 0 bytes of memory. Closing the channel acts as a broadcast, unblocking every listener simultaneously.

```go
// Use a 'done' channel to coordinate shutdown
done := make(chan struct{})

// In 100 different goroutines:
go func() {
    for {
        select {
        case <-done:
            return // All 100 goroutines exit immediately when 'done' is closed
        default:
            doWork()
        }
    }
}()

// To shut everything down:
close(done)
```

In modern Go, we usually use *context.Context* for this, but under the hood, *ctx.Done()* is exactly this - a receive-only channel that closes when the context is cancelled.

The Synchronization Overhead

Channel Performance: The Hidden Cost

Channels are safe because they use Mutexes internally. This means every send/receive operation involves a lock.

The Trade-off:
For massive data streams (millions of small objects), channels can be slower than a shared slice protected by a *sync.Mutex* or *atomic* values.

The Optimization:
"Batching" Instead of sending 1,000 integers one by one, send a single slice of 1,000 integers. This reduces the number of lock acquisitions from 1000 down to 1

Senior Rule:
Use channels for orchestration and ownership transfer. Use shared memory (with proper locking) or batching for high-frequency data throughput.

Orchestration over Communication.
Recap:

- **Unbuffered**: Instant synchronization; high safety.
- **Buffered**: Decouples producer/consumer; smooths bursts.
- **Signal Channels**: Use *chan struct{}* for 0-cost broadcasts.
- **Batching**: Crucial for high-performance channel pipelines.

Question: Have you ever used a "signal channel" to coordinate workers, or do you strictly stick to *context.Context*?
What are the pros and cons of each in your experience?