Channels

Definition (Precise)

A **channel** is a typed conduit through which you can send and receive values using the channel operator `<-`. Channels are Go's primary synchronization mechanism between goroutines.

Key insight: Channels provide both **communication** (data transfer) and **synchronization** (happens-before guarantees).

Syntax

```
// Declaration
var ch chan int                 // nil channel (unusable)
ch = make(chan int)             // unbuffered channel
ch = make(chan int, 10)         // buffered channel (capacity 10)

// Send
ch <- 42                        // Send 42 to channel (blocks until received)

// Receive
value := <-ch                   // Receive from channel (blocks until sent)
value, ok := <-ch               // Receive with closed-check

// Close
close(ch)                       // Close channel (senders must NOT send after)

// Direction
func producer(ch chan<- int)    // Send-only channel
func consumer(ch <-chan int)    // Receive-only channel
```

Mental Model

Think of a channel as a **typed pipe** between goroutines:

```
Goroutine A                    Channel                    Goroutine B
─────────────                  ─────────                  ─────────────
   ch <- 42    ──────────────>  [ 42 ]  ──────────────>      value := <-ch
```

Key properties:

- **Typed:** Only values of channel's type can be sent
- **Blocking:** Send/receive operations synchronize goroutines
- **Thread-safe:** Multiple goroutines can use same channel safely

Unbuffered Channels (Default)

```
ch := make(chan int) // No buffer
```

Behavior: Send **blocks** until receive happens (and vice versa).

```
ch := make(chan int)

go func() {
    fmt.Println("Before send")
    ch <- 42                    // Blocks here until receive
    fmt.Println("After send")
}()

time.Sleep(time.Second)        // Goroutine blocked at send
value := <-ch                  // Receive unblocks send
fmt.Println("Received:", value)
```

Output:

```
Before send
(1 second pause)
Received: 42
After send
```

Synchronization Guarantee

With unbuffered channels:

- Send happens-before corresponding receive completes
- This provides **synchronization point**

```
var data string
ch := make(chan bool)

go func() {
    data = "hello"          // Write
    ch <- true              // Signal completion
}()

<-ch                        // Wait for signal
fmt.Println(data)          // Safe to read: guaranteed "hello"
```

Happens-before:

```
Write to data → Send → Receive → Read from data
```

Buffered Channels

```
ch := make(chan int, 3) // Buffer capacity 3
```

Behavior:

- Send **blocks** only when buffer is full

- Receive **blocks** only when buffer is empty

```
ch := make(chan int, 2)

ch <- 1  // Doesn't block (buffer: [1])
ch <- 2  // Doesn't block (buffer: [1, 2])
ch <- 3  // BLOCKS (buffer full)

<-ch     // Receive 1 (buffer: [2])
<-ch     // Receive 2 (buffer: [])
```

Buffer as Queue (FIFO)

```
ch := make(chan string, 3)
ch <- "first"
ch <- "second"
ch <- "third"

fmt.Println(<-ch) // "first"
fmt.Println(<-ch) // "second"
fmt.Println(<-ch) // "third"
```

Closing Channels

```
ch := make(chan int, 2)
ch <- 1
ch <- 2
close(ch)

fmt.Println(<-ch)  // 1
fmt.Println(<-ch)  // 2
fmt.Println(<-ch)  // 0 (zero value, channel closed)

v, ok := <-ch
fmt.Println(v, ok) // 0 false (ok indicates channel closed)
```

Rules:

1. Only **sender** should close channels (receiver doesn't know if more sends coming)
2. Sending to closed channel → **panic**
3. Receiving from closed channel → returns zero value immediately
4. Closing nil channel → **panic**
5. Closing already-closed channel → **panic**

Closing for Signaling Completion

```
done := make(chan struct{}) // Empty struct uses no memory
```

```
go func() {
    // Do work
    close(done) // Signal completion
}()

<-done // Wait for completion
```

Why `struct{}`: Occupies zero bytes, perfect for signaling.

Range Over Channel

```
ch := make(chan int, 5)

go func() {
    for i := 0; i < 5; i++ {
        ch <- i
    }
    close(ch) // Must close or range loops forever
}()

for value := range ch {
    fmt.Println(value) // 0, 1, 2, 3, 4
}
```

Key: `range` exits when channel closed.

Channel Directions

Send-Only Channel

```
func producer(ch chan<- int) {
    ch <- 42    // OK
    // <-ch     // Compile error: cannot receive from send-only channel
}
```

Receive-Only Channel

```
func consumer(ch <-chan int) {
    value := <-ch    // OK
    // ch <- 42      // Compile error: cannot send to receive-only channel
}
```

Why Use Directions?

Type safety: Prevent accidental misuse.

```
func pipeline() {
    ch := make(chan int)
```

```
    go producer(ch)  // Converted to chan<- int
    go consumer(ch)  // Converted to <-chan int
}

func producer(ch chan<- int) {
    ch <- 42
    // close(ch) // Error: cannot close send-only channel (common bug prevention)
}
```

Nil Channels (Dangerous)

```
var ch chan int // nil channel

ch <- 1  // Blocks forever (nil channel send)
<-ch     // Blocks forever (nil channel receive)
close(ch) // Panic
```

Use case: Disabling a case in select (covered in select.md).

Common Patterns

Pattern 1: Request-Response

```
type Request struct {
    Data     string
    Response chan string
}

func handleRequest(req Request) {
    result := process(req.Data)
    req.Response <- result
}

func main() {
    req := Request{
        Data:     "input",
        Response: make(chan string),
    }

    go handleRequest(req)

    result := <-req.Response
    fmt.Println(result)
}
```

Pattern 2: Fan-Out (One Producer, Multiple Consumers)

```go
func fanOut(input <-chan int, workers int) []<-chan int {
    outputs := make([]<-chan int, workers)

    for i := 0; i < workers; i++ {
        out := make(chan int)
        outputs[i] = out

        go func(ch chan int) {
            for val := range input {
                ch <- process(val)
            }
            close(ch)
        }(out)
    }

    return outputs
}
```

Pattern 3: Fan-In (Multiple Producers, One Consumer)

```go
func fanIn(channels ...<-chan int) <-chan int {
    out := make(chan int)

    var wg sync.WaitGroup
    for _, ch := range channels {
        wg.Add(1)
        go func(c <-chan int) {
            defer wg.Done()
            for val := range c {
                out <- val
            }
        }(ch)
    }

    go func() {
        wg.Wait()
        close(out)
    }()

    return out
}
```

Common Bugs

Bug 1: Forgetting to Close Channel in Range Loop

```go
// WRONG
ch := make(chan int)
```

```
go func() {
    for i := 0; i < 5; i++ {
        ch <- i
    }
    // Forgot to close(ch)
}()

for val := range ch {
    fmt.Println(val) // Deadlock after printing 0-4
}
```

Fix: Always close when done sending.

Bug 2: Closing Channel from Receiver Side

```
// WRONG
func receiver(ch <-chan int) {
    for val := range ch {
        process(val)
    }
    // close(ch) // Compile error: cannot close receive-only channel
}

// Even if ch is bidirectional:
func receiver(ch chan int) {
    for val := range ch {
        process(val)
    }
    close(ch) // Dangerous: sender might still be sending
}
```

Rule: Only sender closes.

Bug 3: Sending to Closed Channel

```
// WRONG
ch := make(chan int)
close(ch)
ch <- 1 // PANIC: send on closed channel
```

Fix: Coordinate who closes and when.

Bug 4: Deadlock from Blocking Send/Receive

```
// WRONG: Single goroutine deadlock
func main() {
    ch := make(chan int)
    ch <- 42            // Blocks forever (no receiver)
    fmt.Println(<-ch)   // Never reached
```

```
}
// fatal error: all goroutines are asleep - deadlock!
```

Fix: Use goroutines or buffered channels appropriately.

Bug 5: Channel Leaks

```
// WRONG
func leak() <-chan int {
    ch := make(chan int)
    go func() {
        ch <- compute() // If nobody receives, goroutine blocks forever
    }()
    return ch
}

result := leak()
// Forgot to receive from result → goroutine leaks
```

Fix: Always consume from channels or use context cancellation.

Channel vs Mutex: When to Use What

Use Channels When:

- **Passing ownership** of data
- **Distributing work** to multiple workers
- **Communicating results** back
- **Coordinating goroutines** at specific points

```
// Good use of channel: Passing ownership
func producer() <-chan Data {
    ch := make(chan Data)
    go func() {
        ch <- generateData() // Transfer ownership
        close(ch)
    }()
    return ch
}
```

Use Mutex When:

- **Protecting access** to shared state
- **Caching** (shared read-heavy data)
- **Short critical sections** (increment counter, update map)

```
// Good use of mutex: Protecting shared state
type Cache struct {
    mu    sync.RWMutex
    items map[string]string
```

```
}

func (c *Cache) Get(key string) string {
    c.mu.RLock()
    defer c.mu.RUnlock()
    return c.items[key]
}
```

Rob Pike's advice:

> *Share memory by communicating (channels), don't communicate by sharing memory (mutexes).*

But: This is philosophy, not law. Use the right tool.

Performance Characteristics

Operation	Unbuffered	Buffered (space available)	Buffered (full/empty)
Send	Blocks until receive	~50-100 ns	Blocks until space
Receive	Blocks until send	~50-100 ns	Blocks until data
Close	~10 ns	~10 ns	~10 ns

Mutex comparison:

- Lock/Unlock: ~20-30 ns (uncontended)
- Channels: ~50-100 ns

Takeaway: Channels are ~2-3x slower than mutexes for simple sync, but provide richer semantics.

Real-World Failure: Goroutine Leak via Channel

Company: E-commerce platform (2019)

What happened:
Service memory grew from 500MB to 50GB over 2 weeks, eventually crashing.

Root cause:

```
func processOrder(order Order) {
    resultCh := make(chan Result)

    go func() {
        result := expensiveComputation(order)
        resultCh <- result // Blocks forever if nobody receives
    }()

    // BUG: Sometimes function returns early on validation error
    if !order.IsValid() {
        return // resultCh never consumed → goroutine leaks
    }

    result := <-resultCh
```

```
    saveResult(result)
}
```

After 2 weeks at 1000 requests/sec:

- ~1 billion leaked goroutines
- ~2TB memory consumed (each goroutine + result)
- Service OOM crashes

Fix 1: Always consume channel

```
defer func() {
    // Drain channel even if returning early
    select {
    case <-resultCh:
    case <-time.After(time.Second):
    }
}()
```

Fix 2: Use buffered channel (better)

```
resultCh := make(chan Result, 1) // Buffer prevents blocking

go func() {
    result := expensiveComputation(order)
    resultCh <- result // Never blocks (buffer size 1)
}()

if !order.IsValid() {
    return // goroutine completes, no leak
}

result := <-resultCh
```

Lesson: Unbuffered channels between goroutines require both ends to coordinate. Buffered channels (size 1) for result returns prevent leaks.

Interview Traps

Trap 1: "Channels are queues"

Incomplete. Channels can queue (if buffered), but they're primarily a **synchronization mechanism**.

Correct answer:
"Channels provide both communication and synchronization. Unbuffered channels synchronize sender and receiver (blocking until both ready). Buffered channels have queue semantics but still provide happens-before guarantees between sends and receives."

Trap 2: "Buffered channels make everything faster"

Wrong. Buffers reduce blocking but don't eliminate synchronization cost.

Correct answer:
"Buffered channels reduce blocking when sender/receiver run at different speeds, improving throughput. But they don't eliminate channel overhead (~50-100ns) and can hide coordination issues. Buffer size should match your concurrency model (e.g., number of workers)."

Trap 3: "I should close channels in defer"

Usually wrong. Closing should happen when all sends are done, which may not align with function exit.

Correct answer:
"Channels should be closed by the sender when no more values will be sent. This might be in defer if the sender is the function itself, but often it's after a loop or when workers finish. Closing in defer everywhere can lead to closing too early or closing channels you don't own."

Trap 4: "Channels always prevent races"

Wrong. Channels synchronize communication, but shared state outside channels can still race.

Correct answer:
"Channels synchronize their own send/receive operations and provide happens-before guarantees between those operations. But if goroutines share state outside channel communication (e.g., global variables), races can still occur. Proper channel usage transfers ownership of data, avoiding shared state."

Trap 5: "This code is safe because I use channels"

```
var result int
done := make(chan bool)

go func() {
    result = compute() // Write
    done <- true
}()

<-done
fmt.Println(result) // Read
```

Correct: This IS safe. Why?

Answer:
"This is safe because the channel creates a happens-before relationship: the write to `result` happens-before the send, which happens-before the receive, which happens-before the read. The channel provides memory synchronization, making the write visible to the reader."

Key Takeaways

1. **Channels synchronize AND communicate**
2. **Unbuffered channels = synchronous rendezvous**
3. **Buffered channels = asynchronous up to capacity**
4. **Only sender should close** (receiver doesn't know when to close)
5. **Closed channel returns zero value** (use `, ok` to check)
6. **Nil channel blocks forever** (use for disabling select cases)
7. **Channel directions** provide type safety

8. **Channels are ~2-3x slower than mutexes** but more expressive
9. **Channel leaks cause goroutine leaks** (always consume or use buffered)
10. **Use select for non-blocking/timeout operations** (next doc)

What You Should Be Thinking Now

- "When do I use buffered vs unbuffered?"
- "How do I handle multiple channels?"
- "How do I implement timeouts with channels?"
- "What's the select statement for?"

Next: buffered-vs-unbuffered.md - Deep dive into choosing the right channel type.

Exercises (Do These Before Moving On)

1. Write a program where two goroutines use an unbuffered channel. Observe blocking behavior.
2. Create a buffered channel and demonstrate it blocks only when full.
3. Write code that intentionally panics by sending to a closed channel. Verify the panic.
4. Implement fan-in: merge 3 channels into 1.
5. Create a channel leak. Use `runtime.NumGoroutine()` to detect it.
6. Explain: Why does `for val := range ch` require closing the channel?

Don't continue until you can explain: "When should I use a buffered channel of size 1 for result returns?"