Goroutines

Definition (Precise)

A **goroutine** is a lightweight, independently executing function managed by the Go runtime scheduler. Goroutines are Go's fundamental unit of concurrency.

Not a thread. Not a process. A goroutine.

Syntax

```
go functionCall()
```

That's it. The `go` keyword schedules a function to execute concurrently.

Mental Model

Think of a goroutine as: **"Do this work independently; I don't want to wait for it."**

```
// Sequential: Do task1, then task2
task1()
task2()

// Concurrent: Start task1 in background, continue with task2
go task1()
task2()
```

How Goroutines Work

Creation

```
func main() {
    go doWork() // Creates new goroutine

    // main continues executing immediately
    // doWork() runs concurrently
}

func doWork() {
    // This runs in its own goroutine
    fmt.Println("doing work")
}
```

What happens:

1. `go doWork()` creates a new goroutine
2. `doWork` is scheduled to run
3. **main continues immediately** without waiting

4. Scheduler decides when `doWork` actually executes

Scheduling

Goroutines are **cooperatively scheduled** with preemption:

- **Cooperative:** Goroutines yield on function calls, channel ops, blocking syscalls
- **Preemptive (Go 1.14+):** Goroutines can be preempted during long-running loops

```
// Pre Go 1.14: This would block other goroutines
for {
    // tight loop, no function calls
}

// Go 1.14+: Scheduler can preempt this
for {
    // tight loop, but scheduler intervenes
}
```

Goroutine Lifecycle

```
Created → Runnable → Running → Blocked → Runnable → ... → Dead
   ↑           ↓          ↓          ↑
go func()  Scheduled  Executing  I/O wait
```

States:

- **Runnable:** Ready to execute, waiting for CPU
- **Running:** Currently executing on a thread
- **Blocked:** Waiting for I/O, channel, mutex, etc.
- **Dead:** Function returned

Correct Examples

Example 1: Fire and Forget (With Caveat)

```
func logAsync(message string) {
    go func() {
        log.Println(message)
    }()
    // Returns immediately, logging happens in background
}

// Caveat: If main exits before goroutine runs, log may not print!
```

Example 2: Concurrent HTTP Requests

```
func fetchAll(urls []string) []string {
    results := make([]string, len(urls))
    var wg sync.WaitGroup
    for i, url := range urls {
        wg.Add(1)
        go func(index int, u string) {
            defer wg.Done()
            results[index] = fetch(u)
        }(i, url) // Pass as parameters!
    }

    wg.Wait()
    return results
}
```

Key points:

- `wg.Wait()` ensures all goroutines finish
- Parameters passed explicitly to avoid closure capture bug

Example 3: Background Worker

```
func startWorker(ctx context.Context, jobs <-chan Job) {
    go func() {
        for {
            select {
            case job := <-jobs:
                process(job)
            case <-ctx.Done():
                return // Goroutine exits cleanly
            }
        }
    }()
}
```

Key points:

- Context enables cancellation
- Goroutine exits when context cancelled

Common Bugs

Bug 1: Closure Capture in Loops

```
// WRONG
for i := 0; i < 5; i++ {
    go func() {
        fmt.Println(i) // Captures loop variable
    }()
}
// Prints: 5 5 5 5 5 (or other unpredictable values)
```

Why: All goroutines share the same `i` variable. By the time they execute, `i` is likely 5.

```go
// CORRECT 1: Pass as parameter
for i := 0; i < 5; i++ {
    go func(val int) {
        fmt.Println(val)
    }(i)
}

// CORRECT 2: Shadow variable
for i := 0; i < 5; i++ {
    i := i // Create new variable in loop scope
    go func() {
        fmt.Println(i)
    }()
}
```

Bug 2: Goroutine Leaks (Not Exiting)

```go
// WRONG
func startProcessor() {
    go func() {
        for {
            data := <-dataChannel
            process(data)
        }
    }()
    // If dataChannel is never closed and never receives,
    // this goroutine blocks forever → LEAK
}
```

Detection:

```bash
$ go test -v -timeout 10s
# If test times out, likely goroutine leak
```

Fix:

```go
// CORRECT
func startProcessor(ctx context.Context) {
    go func() {
        for {
            select {
            case data := <-dataChannel:
                process(data)
            case <-ctx.Done():
                return // Goroutine exits
            }
        }
    }()
}
```

Bug 3: Racing on Shared State

```go
// WRONG
var counter int

for i := 0; i < 100; i++ {
    go func() {
        counter++ // DATA RACE
    }()
}
```

Run with race detector:

```
$ go run -race main.go
WARNING: DATA RACE
Write at 0x... by goroutine X:
Read at 0x... by goroutine Y:
```

Fix:

```go
// CORRECT 1: Use sync.Mutex
var counter int
var mu sync.Mutex

for i := 0; i < 100; i++ {
    go func() {
        mu.Lock()
        counter++
        mu.Unlock()
    }()
}

// CORRECT 2: Use atomic
var counter int64

for i := 0; i < 100; i++ {
    go func() {
        atomic.AddInt64(&counter, 1)
    }()
}

// CORRECT 3: Use a channel
counterCh := make(chan int)
go func() {
    count := 0
    for range counterCh {
        count++
    }
}()

for i := 0; i < 100; i++ {
    go func() {
        counterCh <- 1
    }()
}
```

Bug 4: Exiting Main Before Goroutines Finish

```
// WRONG
func main() {
    go doWork()
    // main exits immediately, doWork may not run!
}
```

Fix:

```
// CORRECT
func main() {
    var wg sync.WaitGroup
    wg.Add(1)

    go func() {
        defer wg.Done()
        doWork()
    }()

    wg.Wait() // Wait for goroutine
}
```

Performance Characteristics

Metric	Value	Comparison
Initial stack size	2 KB	OS thread: 1-2 MB
Creation time	~100-200 ns	OS thread: ~1-2 μs
Context switch	~50-100 ns	OS thread: ~1-2 μs
Max concurrent	Millions	OS threads: Thousands

Takeaway: Goroutines are cheap, but not free.

When to Use Goroutines

✅ **Use goroutines for:**

- Concurrent I/O operations (network, disk, databases)

- Background tasks (logging, metrics, cleanup)
- Servers handling multiple clients
- Processing independent units of work
- Event-driven systems

❌ **Avoid goroutines for:**

- Single operations (unnecessary overhead)
- Tightly coupled sequential steps
- When debugging is more valuable than concurrency
- When you can't guarantee they'll exit (leak risk)

Goroutine Anti-Patterns

Anti-Pattern 1: Unbounded Goroutine Creation

```
// BAD: Creates goroutine per request, no limit
func handler(w http.ResponseWriter, r *http.Request) {
    go processRequest(r) // Unbounded!
}
```

Problem: During traffic spike, millions of goroutines created → scheduler thrashing.

Fix: Use worker pool (covered in patterns section).

Anti-Pattern 2: Fire-and-Forget Without Observability

```
// BAD: No way to know if goroutine succeeded
go doImportantWork()
```

Problem: If `doImportantWork` panics or fails, you'll never know.

Fix:

```
// BETTER
go func() {
    defer func() {
        if r := recover(); r != nil {
            log.Printf("Panic in doImportantWork: %v", r)
            metrics.Inc("goroutine_panics")
        }
    }()
    doImportantWork()
}()
```

Anti-Pattern 3: Starting Goroutines in Constructors

```
// BAD
func NewService() *Service {
    s := &Service{}
```

```go
    go s.backgroundTask() // Leaks if service is never used
    return s
}
```

Problem: Goroutine starts immediately, no way to stop it.

Fix:

```go
// BETTER
func NewService() *Service {
    return &Service{}
}

func (s *Service) Start(ctx context.Context) {
    go s.backgroundTask(ctx) // Started explicitly with context
}
```

Real-World Failure: Goroutine Leak at Scale

Company: Major payment processor (2021)

What happened:
Memory usage grew steadily over days, eventually OOM crashing the service.

Root cause:

```go
func processPayment(payment Payment) {
    go func() {
        result := externalAPI.charge(payment) // Can block forever if API hangs
        saveResult(result)
    }()
}
```

Each payment spawned a goroutine. If `externalAPI.charge` hung (network timeout, slow API), the goroutine blocked forever.

After 1 week at 10,000 requests/sec:

- ~6 billion goroutines leaked
- ~12 TB memory consumed
- Application crashed

Fix:

```go
func processPayment(ctx context.Context, payment Payment) {
    go func() {
        ctx, cancel := context.WithTimeout(ctx, 10*time.Second)
        defer cancel()

        resultCh := make(chan Result, 1)
        go func() {
            result := externalAPI.charge(payment)
            resultCh <- result
        }()

        select {
        case result := <-resultCh:
            saveResult(result)
        case <-ctx.Done():
            log.Printf("Payment %v timed out", payment.ID)
        }
    }()
}
```

Lessons:

1. Always use timeouts with external dependencies
2. Context cancellation is mandatory
3. Monitor goroutine count in production: `runtime.NumGoroutine()`

Goroutine Panics

```go
func main() {
    go func() {
        panic("something went wrong") // Panics in goroutine
    }()

    time.Sleep(time.Second)
    fmt.Println("main continues") // Not reached!
}
// Output: panic: something went wrong
// Program crashes
```

Key insight: A panic in a goroutine crashes the entire program unless recovered.

Recovery:

```go
func main() {
    go func() {
        defer func() {
            if r := recover(); r != nil {
                log.Printf("Recovered from panic: %v", r)
            }
        }()
        panic("something went wrong")
    }()

    time.Sleep(time.Second)
    fmt.Println("main continues") // This runs!
}
```

Debugging Goroutines

Get Goroutine Count

```
fmt.Println("Goroutines:", runtime.NumGoroutine())
```

Get Goroutine Dump

```
# Send SIGQUIT to running program
kill -QUIT <pid>

# Or in code:
pprof.Lookup("goroutine").WriteTo(os.Stdout, 1)
```

Output:

```
goroutine 1 [running]:
main.main()
    /path/to/main.go:10 +0x39

goroutine 17 [chan receive]:
main.worker()
    /path/to/main.go:25 +0x4f
```

Interview Traps

Trap 1: "Goroutines run in parallel"

Incomplete. Goroutines are concurrent; parallelism depends on `GOMAXPROCS` and available cores.

Correct answer:
"Goroutines enable concurrent execution. Whether they run in parallel depends on GOMAXPROCS and the number of available CPU cores. On a single core, goroutines are concurrent but not parallel."

Trap 2: "Goroutines are threads"

Wrong. Goroutines multiplex onto threads.

Correct answer:
"Goroutines are lightweight, user-space constructs managed by the Go runtime. Multiple goroutines multiplex onto a smaller number of OS threads (M:N scheduling). They have much smaller stacks (2KB vs 1-2MB) and cheaper context switches."

Trap 3: "This loop is fine because I used `go`"

```
for i := 0; i < 10; i++ {
    go func() { fmt.Println(i) }()
}
```

Wrong. Classic closure capture bug.

Correct answer:
"This has a closure capture bug. All goroutines share the loop variable `i`. By the time they execute, `i` is likely 10. Fix by passing `i` as a parameter or shadowing it in the loop body."

Trap 4: "I don't need to wait for goroutines"

Wrong. If main exits, all goroutines die.

Correct answer:
"If the main goroutine exits, the program terminates immediately, killing all other goroutines. I must use sync.WaitGroup, channels, or other synchronization to ensure main waits for goroutines to complete."

Key Takeaways

1. **Goroutines are cheap but not free** (~2KB + scheduling cost)
2. **`go` schedules execution, doesn't guarantee when**
3. **Closure capture is a common bug** (pass parameters)
4. **Goroutine leaks are silent and deadly** (monitor with `runtime.NumGoroutine()`)
5. **Panics in goroutines crash the program** (use recover in goroutines)
6. **Context cancellation is mandatory** for production goroutines
7. **Main exiting kills all goroutines** (use WaitGroup or similar)

What You Should Be Thinking Now

- "How do I coordinate goroutines?"
- "How do I safely share data between goroutines?"
- "What are channels and when should I use them?"
- "How do I prevent goroutine leaks systematically?"

Next: channels.md - We'll learn Go's primary synchronization mechanism.

Exercises (Do These Before Moving On)

1. Write a program that starts 10 goroutines, each printing its index. Fix the closure capture bug.

2. Create a goroutine leak intentionally. Use `runtime.NumGoroutine()` to detect it.

3. Add a panic to a goroutine and recover from it. Verify the program doesn't crash.

4. Benchmark goroutine creation time:

```go
func BenchmarkGoroutineCreation(b *testing.B) {
    for i := 0; i < b.N; i++ {
        go func() {}()
    }
}
```

Don't continue until you can explain: "Why does `go func() { fmt.Println(i) }()` in a loop print the wrong values?"