Gorgo Go for Java Programmers

June 11, 2026

Contents

Chapter 12

Context and Concurrency Patterns

Chapters 10 and 11 gave you goroutines, channels, and the `sync` primitives you need to coordinate them. This chapter adds the layer that sits on top of all of that: `context.Context`, the standard way to propagate cancellation and deadlines across goroutine boundaries. It also covers the patterns you will see in real Go services — worker pools, rate limiters, fan-out with error collection, and goroutine leak detection.

context.Context

Java has no direct equivalent to `context.Context`. The closest Java analog is a combination of `Future.cancel()`, `ExecutorService.shutdownNow()`, and a hand-rolled deadline field on a request object — all bolted together differently in every codebase. Go standardizes all of this in a single interface.

```
// context.Context is defined in the standard library as:
type Context interface {
    Deadline() (deadline time.Time, ok bool) // returns the deadline, if any
    Done() <-chan struct{}                   // closed when work should be cancelled
    Err() error                              // nil, then Canceled or DeadlineExceeded
    Value(key any) any                       // returns the value associated with key, or nil
}
```

Every long-running or network-bound function in idiomatic Go accepts a `context.Context` as its first parameter. When the context is cancelled — because a deadline expired, a timeout elapsed, or the caller called a cancel function — `Done()` is closed and `Err()` returns a non-nil error. Your function is expected to notice this and return promptly.

Tip: `context.Context` is not just for HTTP handlers. Use it everywhere work can be cancelled: database queries, RPC calls, file I/O, and long computations.

The Root Contexts

Every context tree starts with one of two roots:

```
ctx := context.Background() // the default root; never cancelled, no deadline, no values
ctx := context.TODO()       // placeholder for "I haven't wired up a context yet"
```

`context.Background()` is for `main`, top-level servers, and test helpers. `context.TODO()` is a compile-time signal that you know a context should be here but have not plumbed it through yet. Treat `context.TODO()` like a `// TODO` comment — it should not survive into production code.

Cancellation, Deadlines, and Timeouts

The three constructors that add cancellation to a context are `context.WithCancel`, `context.WithDeadline`, and `context.WithTimeout`. Each returns a derived context and a cancel function. **Always call the cancel function**, even if the operation finishes before the deadline. Failing to call cancel keeps the derived context (and its timer) alive until the deadline fires or the parent is cancelled, pinning everything the context references.

```
// WithCancel returns a copy of parent whose Done channel is closed when cancel is called.
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)

// WithDeadline returns a copy of parent with a deadline set to d.
func WithDeadline(parent Context, d time.Time) (ctx Context, cancel CancelFunc)

// WithTimeout returns WithDeadline(parent, time.Now().Add(timeout)).
func WithTimeout(parent Context, timeout time.Duration) (ctx Context, cancel CancelFunc)
```

Here is a function that fetches a song's lyrics from a slow API, with a two-second timeout:

```
package main

import (
    "context"
    "fmt"
    "time"
)

// fetchLyrics simulates a slow network call; it respects ctx cancellation.
func fetchLyrics(ctx context.Context, song string) (string, error) {
    done := make(chan string, 1)
    go func() {
        time.Sleep(3 * time.Second) // simulate slow work
        done <- "I can't keep loving you the way I do"
    }()
    select {
    case lyrics := <-done:
        return lyrics, nil
    case <-ctx.Done():
        return "", ctx.Err() // context.DeadlineExceeded
    }
}

func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
    defer cancel() // always call cancel

    lyrics, err := fetchLyrics(ctx, "Gouryella")
    if err != nil {
        fmt.Println("timed out:", err) // timed out: context deadline exceeded
        return
    }
    fmt.Println(lyrics)
}
```

When the timeout fires, `ctx.Done()` is closed and `ctx.Err()` returns `context.DeadlineExceeded`. If `cancel()` is called before the timeout, `ctx.Err()` returns `context.Canceled`. Your code can distinguish the two with

errors.Is:

```
if errors.Is(ctx.Err(), context.DeadlineExceeded) {
    fmt.Println("ran out of time")
}
if errors.Is(ctx.Err(), context.Canceled) {
    fmt.Println("caller gave up")
}
```

Trap: Do not store the cancel function and call it lazily. Use `defer cancel()` immediately after the `WithTimeout` / `WithCancel` call. If you forget, the runtime cannot release resources associated with the context until the parent is cancelled or the program exits.

Checking Cancellation Inside a Loop

Goroutines doing CPU-bound work need to poll the context rather than waiting on a channel:

```
func processTracks(ctx context.Context, tracks []string) error {
    for _, t := range tracks {
        select {
        case <-ctx.Done():
            return ctx.Err() // bail out early
        default:
        }
        // do real work with t
        fmt.Println("processing:", t)
    }
    return nil
}
```

The `select` with a `default` branch is non-blocking: it drains `Done` if it is already closed but does not block if it is still open. Chapter 10 covered `select` in detail.

context.WithValue

`context.WithValue` attaches a key-value pair to a context that child goroutines can retrieve with `ctx.Value(key)`. This is intended for **request-scoped metadata** — things like a trace ID or an authenticated user — not for passing optional function parameters.

```
func WithValue(parent Context, key, val any) Context
```

The value is retrieved by calling `Value(key)` on any derived context. If no value is found at the current level, Go walks up the context tree until it finds one or reaches the root.

Use Unexported Key Types

If you use a plain string as a key, any package in the call tree can accidentally shadow your value by using the same string key. The idiomatic fix is to define a **package-private type** for your keys. Because the type is unexported, no other package can construct a value of that type, so collisions are impossible.

```
package requestmeta

// traceKey is unexported; no other package can create a value of this type.
type traceKey struct{}

func WithTraceID(ctx context.Context, id string) context.Context {
```

```
    return context.WithValue(ctx, traceKey{}, id)
}

func TraceID(ctx context.Context) (string, bool) {
    id, ok := ctx.Value(traceKey{}).(string)
    return id, ok
}
```

Contrast this with the anti-pattern:

```
// ANTI-PATTERN: string keys can collide across packages
ctx = context.WithValue(ctx, "traceID", "abc-123")
ctx = context.WithValue(ctx, "traceID", "xyz-999") // silently shadows the first one!
```

Trap: Never use a built-in type (string, int, etc.) as a context key. Linters such as `staticcheck` (check SA1029) flag this with `should not use built-in type string as key for value` — note that plain `go vet` does *not* catch it, so do not rely on the standard tool alone here. Always define an unexported struct type for context keys.

Context as First Parameter

Go has a universal convention: if a function accepts a context, it is **always the first parameter** and it is **always named `ctx`**. [*ctx-for-context*]

```
// idiomatic
func SearchSongs(ctx context.Context, query string) ([]Song, error)

// wrong: context buried in the middle
func SearchSongs(query string, ctx context.Context) ([]Song, error)
```

This convention applies throughout the standard library, all major frameworks, and community packages. Do not put a context inside a struct (except when constructing a long-lived object like an HTTP server); pass it explicitly on each call.

Trap: Do not store a `context.Context` in a struct field and use it later. Contexts are request-scoped. A stored context will be cancelled at unpredictable times. Pass the context explicitly to every function that needs it.

The convention is so consistent that when you see a function signature like `func Foo(ctx context.Context, ...)` you immediately know it can be cancelled, timed out, and carries request metadata. Java has no equivalent signal at the call site.

errgroup — Fan-Out with Error Collection

Chapter 11 showed `sync.WaitGroup` for fan-out. `WaitGroup` works, but it cannot collect errors from goroutines. The `golang.org/x/sync/errgroup` package solves both problems: it waits for a group of goroutines to finish **and** returns the first non-nil error any of them produced. `errgroup.WithContext` returns a derived context that the group cancels as soon as any goroutine returns an error.

```
import "golang.org/x/sync/errgroup"

// Group is created with errgroup.WithContext.
// g.Go(f) launches f in a goroutine; g.Wait() blocks until all goroutines finish.
// g.Wait() returns the first non-nil error returned by any goroutine.
```

```
func WithContext(ctx context.Context) (*Group, context.Context)
func (g *Group) Go(f func() error)
func (g *Group) Wait() error
```

Here is a fan-out that fetches three song titles concurrently and cancels all of them if any fails:

```
package main

import (
    "context"
    "fmt"
    "time"

    "golang.org/x/sync/errgroup"
)

// fetchTitle simulates fetching a song title; slow returns an error.
func fetchTitle(ctx context.Context, id int) (string, error) {
    titles := []string{"Gouryella", "Flaming June", "Saltwater"}
    select {
    case <-time.After(time.Duration(id+1) * 100 * time.Millisecond):
        return titles[id%len(titles)], nil
    case <-ctx.Done():
        return "", ctx.Err()
    }
}

func main() {
    ctx := context.Background()
    g, ctx := errgroup.WithContext(ctx)

    results := make([]string, 3)
    for i := 0; i < 3; i++ {
        i := i // capture loop variable; unnecessary on Go 1.22+ (per-iteration scope)
        g.Go(func() error {
            title, err := fetchTitle(ctx, i)
            if err != nil {
                return err
            }
            results[i] = title
            return nil
        })
    }

    if err := g.Wait(); err != nil {
        fmt.Println("error:", err)
        return
    }
    for _, t := range results {
        fmt.Println(t)
    }
}
```

When any goroutine in the group returns a non-nil error, `errgroup` cancels the shared context. Goroutines that are still running will see `ctx.Done()` closed and should return promptly. `g.Wait()` blocks until all

goroutines have returned, then returns the first error.

Tip: `errgroup` is the idiomatic replacement for `sync.WaitGroup` whenever goroutines can fail. If none of them can fail, `sync.WaitGroup` is fine.

Goroutine Leak Detection

A goroutine leak is a goroutine that starts but never exits. Leaks accumulate over the lifetime of a server: each request might leave one goroutine behind, and after thousands of requests you have thousands of idle goroutines consuming memory and scheduler time. Each leaked goroutine also keeps every value it has closed over alive, preventing GC from reclaiming that memory. [*leaked-goroutine-grows-memory*]

The most common cause is a goroutine blocked on a channel send or receive with no path to exit:

```
// BUG: this goroutine leaks if the caller stops listening on results.
func streamHits(results chan<- string) {
    for {
        results <- "Gamemaster" // blocks forever if nobody reads
    }
}
```

Every goroutine must have a clear exit path. [*goroutine-must-exit*] The idiomatic exits are:

- the function returns naturally,
- a `done` channel is closed,
- the context passed in is cancelled.

The `go.uber.org/goleak` package detects leaked goroutines in tests:

```
import (
    "testing"
    "go.uber.org/goleak"
)

func TestMain(m *testing.M) {
    goleak.VerifyTestMain(m) // fails the test suite if any goroutine leaks
}

func TestNoLeak(t *testing.T) {
    defer goleak.VerifyNone(t) // fails this test if a goroutine leaks

    ctx, cancel := context.WithCancel(context.Background())

    done := make(chan struct{})
    go func() {
        defer close(done)
        select {
        case <-ctx.Done(): // goroutine exits when context is cancelled
        }
    }()

    cancel() // signal the goroutine to exit
    <-done   // wait for it to finish before VerifyNone runs
}
```

`goleak.VerifyTestMain(m)` runs `m.Run()` itself, checks for leaked goroutines after the suite completes, and then calls `os.Exit` with the test exit code. Because it calls `os.Exit` for you, it must be the only thing your `TestMain` does — do not call `m.Run()` or `os.Exit` yourself, or you will exit twice and skip the leak check. `goleak.VerifyNone(t)` checks for leaks at the end of a single test function.

Tip: Run `goleak.VerifyTestMain` in every package that uses goroutines. It is cheap, catches real bugs, and forces you to wire up context cancellation correctly.

Trap: A goroutine that loops forever without a `ctx.Done()` or `done` channel check is always a potential leak. Even if your current tests do not expose it, a future caller that cancels the operation will leave it running. Keep goroutine lifetimes simple enough that the exit paths are obvious at a glance. [***obvious-goroutine-lifetimes***]

GOMAXPROCS

Go's scheduler multiplexes goroutines onto OS threads. `GOMAXPROCS` controls how many OS threads the scheduler uses simultaneously. By default it is set to the number of logical CPUs available to the process (i.e., `runtime.NumCPU()`).

You can read or change it at runtime:

```
import "runtime"

n := runtime.GOMAXPROCS(0) // 0 means "don't change it; just return the current value"
fmt.Println("GOMAXPROCS:", n)

runtime.GOMAXPROCS(4) // limit to 4 OS threads
```

You can also set it via an environment variable before starting the process:

```
GOMAXPROCS=4 ./myserver
```

In container environments (Docker, Kubernetes), this used to be a famous trap: through Go 1.24, the default `GOMAXPROCS` read the host CPU count, not the container CPU limit, so a container limited to 2 vCPUs on a 32-core host started with `GOMAXPROCS=32`. Since Go 1.25 the runtime is container-aware on Linux: the default considers the cgroup CPU bandwidth limit and even updates periodically if the limit changes. The historical workaround, `go.uber.org/automaxprocs`, is only needed for programs built with Go 1.24 or earlier (or with `GODEBUG=containermaxprocs=0`):

```
import _ "go.uber.org/automaxprocs" // pre-1.25: sets GOMAXPROCS from cgroup quota
```

Tip: On Go 1.25+ the default `GOMAXPROCS` is correct on bare metal *and* in containers. Reach for `automaxprocs` or an explicit `GOMAXPROCS` in your deployment manifest only when you must support older toolchains.

Worker Pool

Spawning one goroutine per task is fine when tasks are cheap and bounded, but unbounded goroutine creation can exhaust memory or overwhelm a downstream service. A **worker pool** fixes the number of concurrent goroutines: a fixed set of `N` workers pull tasks off a shared jobs channel and push outcomes onto a results channel. This is the Go answer to Java's `ExecutorService` with a fixed thread pool, but built from channels and a `sync.WaitGroup` instead of a framework.

The pattern has three moving parts: the producer sends jobs and closes the jobs channel, the workers range over jobs until it is closed, and a closer goroutine waits for all workers to finish and then closes the results channel so the consumer's `range` terminates.

```
package main

import (
    "fmt"
    "sort"
    "sync"
)

// worker pulls jobs until the jobs channel is closed, then returns.
func worker(id int, jobs <-chan int, results chan<- string, wg *sync.WaitGroup) {
    defer wg.Done()
    for n := range jobs { // exits when jobs is closed and drained
        results <- fmt.Sprintf("worker %d squared %d = %d", id, n, n*n)
    }
}

func main() {
    const numWorkers = 3
    jobs := make(chan int)
    results := make(chan string)

    var wg sync.WaitGroup
    for id := 1; id <= numWorkers; id++ {
        wg.Add(1)
        go worker(id, jobs, results, &wg)
    }

    // producer: send all jobs, then close so workers can exit.
    go func() {
        for n := 1; n <= 7; n++ {
            jobs <- n
        }
        close(jobs)
    }()

    // closer: once every worker has returned, close results.
    go func() {
        wg.Wait()
        close(results)
    }()

    var out []string
    for r := range results { // drains until results is closed
        out = append(out, r)
    }
    sort.Strings(out) // worker order is nondeterministic; sort for a stable display
    for _, r := range out {
        fmt.Println(r)
    }
}
```

The key invariant: the producer closes `jobs`, which lets each worker's `range` loop return; the `WaitGroup` tracks those returns; and the closer goroutine closes `results` only after the last worker is done. Closing `results` is what lets `main`'s `range` terminate. If you forget to close `results`, `main` blocks forever on the final receive — a classic deadlock.

Trap: Do not close `results` from inside a worker. With `N` workers you would close it `N` times, and closing an already-closed channel panics. Close it exactly once, from a goroutine that waits on the `WaitGroup`.

Wut: The worker order in the output is nondeterministic — whichever worker the scheduler wakes first grabs the next job. With three workers and seven instant tasks, even the split is nondeterministic — one eager worker may grab five of the seven jobs. Only when every task takes the same non-trivial time does the distribution settle near 3/2/2.

Rate Limiting

Sometimes the problem is not too few workers but too many requests. **Rate limiting** caps how often an operation runs — protecting a downstream API, a database, or your own service from a thundering herd. Go does not need a library for the common cases; a ticker or a buffered channel is enough.

The simplest throttle uses `time.NewTicker`, which sends the current time on its channel at a fixed interval. Receiving from the ticker channel before each operation paces the loop to one operation per tick.

```
// NewTicker returns a Ticker that sends the time on its C channel every d.
func NewTicker(d Duration) *Ticker
func (t *Ticker) Stop() // stops the ticker; the channel is not closed
```

```
package main

import (
    "fmt"
    "time"
)

func main() {
    requests := []string{"As It Was", "Vampire", "Anti-Hero", "Houdini"}

    limiter := time.NewTicker(200 * time.Millisecond)
    defer limiter.Stop() // release the ticker's resources

    for _, req := range requests {
        <-limiter.C // wait for the next tick before proceeding
        fmt.Println("serving", req)
    }
}
```

This serves at most one request every 200 ms. There is also `time.Tick`, a convenience wrapper that returns just the channel — it has no `Stop`, which used to make it a leak machine.

Tip: Before Go 1.23, every `time.Tick` call leaked its ticker: there is no `Stop`, and unreferenced tickers were never garbage collected. Since Go 1.23 the GC reclaims unreferenced tickers, so `time.Tick` is safe; `time.NewTicker` plus `defer t.Stop()` is still the explicit, version-proof habit and releases the timer immediately instead of waiting for the GC.

A plain ticker throttles to a steady rate but allows no bursts. When you want to permit a short burst and then settle to a steady rate, use a **token bucket**: a buffered channel pre-filled with tokens, refilled on a ticker. Each operation takes a token (blocking if the bucket is empty); a background goroutine drops a token in on every tick.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    const burst = 3
    tokens := make(chan struct{}, burst)
    for range burst {
        tokens <- struct{}{} // start full: allow an initial burst of 3
    }

    refill := time.NewTicker(200 * time.Millisecond)
    defer refill.Stop()
    go func() {
        for range refill.C {
            select {
            case tokens <- struct{}{}: // add a token if there is room
            default:                   // bucket full; drop this token
            }
        }
    }()

    for i := 1; i <= 5; i++ {
        <-tokens // take a token, blocking if none are available
        fmt.Println("request", i, "at", time.Now().Format("15:04:05.000"))
    }
}
```

The first three requests fire immediately (draining the initial burst), then the remaining two are paced at 200 ms each as the refiller tops the bucket back up. The `select` with a `default` in the refiller is what enforces the cap: if the bucket is already full, the new token is discarded rather than blocking.

Try It

Type this in and run it. It threads a request ID through a context with `WithValue`, wraps that context with a `WithTimeout`, and plays a short playlist where each track respects cancellation. Two tracks finish before the deadline; the third runs out of time.

```
package main

import (
    "context"
    "errors"
    "fmt"
    "time"
)
```

```go
type reqIDKey struct{}

func withReqID(ctx context.Context, id string) context.Context {
    return context.WithValue(ctx, reqIDKey{}, id)
}

func reqID(ctx context.Context) string {
    if id, ok := ctx.Value(reqIDKey{}).(string); ok {
        return id
    }
    return "unknown"
}

func play(ctx context.Context, track string) error {
    select {
    case <-time.After(150 * time.Millisecond):
        fmt.Printf("[%s] played %q\n", reqID(ctx), track)
        return nil
    case <-ctx.Done():
        return fmt.Errorf("[%s] %q aborted: %w", reqID(ctx), track, ctx.Err())
    }
}

func main() {
    ctx := withReqID(context.Background(), "req-2026")
    ctx, cancel := context.WithTimeout(ctx, 400*time.Millisecond)
    defer cancel()

    tracks := []string{"As It Was", "Vampire", "Anti-Hero"}
    for _, t := range tracks {
        if err := play(ctx, t); err != nil {
            fmt.Println(err)
            if errors.Is(err, context.DeadlineExceeded) {
                break
            }
        }
    }
}
```

The deterministic output is the first two tracks playing, then `"Anti-Hero" aborted: context deadline exceeded`.

Try these modifications:

- Bump the timeout to `600 * time.Millisecond` and confirm all three tracks play.
- Replace `WithTimeout` with `WithCancel` and call `cancel()` from a separate goroutine after 250 ms; watch `ctx.Err()` switch from `DeadlineExceeded` to `Canceled`.
- Fetch the request ID with a plain string key (`ctx.Value("req")`) and observe that it returns `nil` — the key type matters, not just the underlying value.

Key Points

- `context.Context` carries cancellation, deadlines, and request-scoped values across goroutine boundaries; Java has no direct equivalent.

- The three context constructors are `WithCancel` (manual cancel), `WithDeadline` (absolute time), and `WithTimeout` (relative duration); always `defer cancel()`.
- Use unexported struct types as context keys to prevent collisions; never use strings or other built-in types as keys.
- The universal convention is `func Foo(ctx context.Context, ...)` — context is always the first parameter, always named `ctx`, never stored in a struct.
- `golang.org/x/sync/errgroup` is the idiomatic fan-out tool when goroutines can fail; it collects the first error and cancels the shared context.
- A worker pool fixes concurrency at `N` goroutines ranging over a shared jobs channel; the producer closes `jobs`, a `WaitGroup` tracks the workers, and a closer goroutine closes `results` exactly once after they finish.
- Rate limiting needs no library: pace a loop with `time.NewTicker` for a steady rate, or use a buffered channel as a token bucket to allow a burst before settling; `defer t.Stop()` to release the timer promptly.
- Every goroutine must have an exit path; use `go.uber.org/goleak` in tests to catch leaks automatically.
- `GOMAXPROCS` defaults to the number of logical CPUs, and since Go 1.25 it also respects container cgroup CPU limits; `go.uber.org/automaxprocs` is only needed on older toolchains.

Exercises

1. **Think about it:** In Java, cancelling an in-flight operation typically means calling `Future.cancel(true)` or interrupting a thread via `Thread.interrupt()`. Describe how Go's `context.Context` model differs from Java's thread-interrupt approach. What are the advantages of passing a context explicitly rather than relying on a thread-level interrupt mechanism? Consider what happens when a Java thread is blocked in a third-party library that does not handle `InterruptedException`, compared to how a Go function using a context-aware library would behave.

2. **What does this print?**

```go
package main

import (
    "context"
    "fmt"
    "time"
)

func work(ctx context.Context, label string) {
    select {
    case <-time.After(500 * time.Millisecond):
        fmt.Println(label, "done")
    case <-ctx.Done():
        fmt.Println(label, "cancelled:", ctx.Err())
    }
}

func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 200*time.Millisecond)
    defer cancel()

    go work(ctx, "Flaming June")
    go work(ctx, "Saltwater")
    time.Sleep(400 * time.Millisecond)
    fmt.Println("main done")
}
```

3. **Calculation:** You run a worker pool with `workers = 3` and feed it a slice of 7 tasks. Each task takes exactly 100 ms. Assuming no overhead and perfect parallelism, how many milliseconds does the pool take to complete all 7 tasks? Show your work: how many rounds of 3 concurrent workers are needed and what does each round contribute?

4. **Where is the bug?**

```go
package main

import (
    "context"
    "fmt"
    "time"
)

func fetchData(url string) <-chan string {
    ch := make(chan string)
    go func() {
        time.Sleep(2 * time.Second)
        ch <- "result for " + url
    }()
    return ch
}

func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 500*time.Millisecond)
    defer cancel()

    ch := fetchData("https://example.com/songs")
    select {
    case result := <-ch:
        fmt.Println(result)
    case <-ctx.Done():
        fmt.Println("timed out")
    }
}
```

5. **Write a program:** Implement a function `fanOutFetch(ctx context.Context, songs []string) ([]string, error)` that uses `errgroup` to fetch all song titles concurrently. Simulate each fetch with a `time.Sleep` of a random duration between 50 and 150 ms (use `math/rand`). If any fetch takes longer than 300 ms total (enforced by a timeout on the context passed to `fanOutFetch`), the entire operation should be cancelled and an error returned. Print either all results in order or the cancellation error.

6. **What does this print?**

```go
package main

import (
    "context"
    "fmt"
)

type ctxKey string

func main() {
    const userKey ctxKey = "user"
```

```
    ctx := context.Background()
    ctx = context.WithValue(ctx, userKey, "ana")
    ctx = context.WithValue(ctx, ctxKey("user"), "beto")

    fmt.Println(ctx.Value(userKey))
    fmt.Println(ctx.Value("user"))
}
```