Golang sync.WaitGroup: Powerful, but tricky

Last updated on September 10, 2025

Jens Neuse
2025-05-12 · 13min read

Go makes it very easy to write concurrent code. Start a few goroutines, wait for all of them to finish, and done. The `go` keyword makes this a breeze, and with the `sync.WaitGroup` primitive, synchronization can be achieved with no hassle. But is it really *that* simple? (Spoiler: Sometimes, not quite.)

At WunderGraph, we're using Golang to build a GraphQL Router that splits an incoming request into multiple smaller requests to different services. We then merge the results back together to form the final response. As you can imagine, you'd want your Router to be as efficient as possible, so you execute as many of these smaller requests in parallel as possible.

Consequently, we had to dig quite deep into the guts of writing concurrent code in Go. This post aims to share some of the lessons we learned because slapping `go` in front of everything and using `sync.WaitGroup` without fully understanding it can easily lead to bugs like deadlocks or improper cancellation.

What is Golang's sync.WaitGroup?

Let's start with a very simple example:

```
1   package main
2
3   import (
4       "fmt"
5       "sync"
6   )
7
8   func main() {
9       wg := &sync.WaitGroup{}
10      // We need to wait for one goroutine, so we Add(1) *before* starting it.
```

```
    wg.Add(1)
    go func() {
        // Defer Done() right away to ensure it's called, even if the gorouti
        defer wg.Done()
        fmt.Println("Hello, World!")
    }()
    wg.Wait() // Wait blocks until the counter is zero.
}
```

You can try out the code yourself in the Go Playground. If you remove the `wg.Wait()` call, you will see that the program exits before the goroutine is able to print the message.

Like the name suggests, `WaitGroup` is a primitive that allows us to wait for a group of goroutines to finish. However, we didn't really leverage the "group" aspect of it, so let's make our example slightly more complex and realistic.

Let's say we're fetching the availability of a product from one service and its price from another service. In GraphQL Federation, the query planner typically executes these two requests in parallel.

```
        // Always defer Done() immediately inside the goroutine.
        defer wg.Done()
        // Simulate network call to a "Products" subgraph
        fmt.Println("Fetching availability for product 1...")
        // Pretend GraphQL query: query { product(id:1) { availability } }
        time.Sleep(100 * time.Millisecond) // Simulate network latency
        product.Availability = "In Stock"
        fmt.Println("Availability fetched.")
    }()

    // Goroutine to fetch price
    wg.Add(1)
    go func() {
        // Defer Done() right away!
        defer wg.Done()
        // Simulate network call to a "Pricing" subgraph
        fmt.Println("Fetching price for product 1...")
        // Pretend GraphQL query: query { product(id:1) { price } }
        time.Sleep(150 * time.Millisecond) // Simulate network latency
        product.Price = 29.99
        fmt.Println("Price fetched.")
```

```
47          }()
48
49          // Wait for both goroutines to complete
50          fmt.Println("Router is waiting for subgraph responses...")
51          wg.Wait() // This blocks until the counter hits zero (both Done() calls
52
53          fmt.Printf("Successfully fetched data for Product ID %d: %+v\n", produc
54      }
```

Here's a link to the Go Playground to run the code yourself. Now we're doing something that resembles a real-world scenario and we're using the `WaitGroup` to wait for both goroutines to finish.

There are still a few lurking gotchas with this code (we'll get to those!), but let's first look into how the `WaitGroup` actually works under the hood.

How does sync.WaitGroup Actually Work? (The Nitty Gritty)

The `WaitGroup` implementation, although just 129 lines of code, is actually quite interesting to look at. We can learn a lot about writing concurrent code in Go and about the runtime and the Go scheduler.

Let's take a look at the `WaitGroup` struct:

```
1   // A WaitGroup waits for a collection of goroutines to finish.
2   // The main goroutine calls Add to set the number of goroutines to wait for.
3   // Then each of the goroutines runs and calls Done when finished. At the same
4   // time, Wait can be used to block until all goroutines have finished.
5   //
6   // A WaitGroup must not be copied after first use.
7   type WaitGroup struct {
8       noCopy noCopy // Used by vet tool to check for copying
9
10      // 64-bit value: high 32 bits are counter, low 32 bits are waiter count.
11      // access atomically, holds the state plus waiter count.
12      state atomic.Uint64
13      // Semaphore for waiters to sleep on.
```

```
14        sema uint32
15    }
```

The first thing that stands out is the `noCopy` field. This isn't data; it's a clever trick. If you try to copy a `WaitGroup` after its first use, the Go `vet` tool will yell at you. Why? Because copying it would mean the counter and waiters wouldn't be shared correctly, leading to chaos. Think of it like trying to photocopy a shared to-do list – everyone ends up with different versions!

The second thing is the `state` field, an `atomic.Uint64`. This is where the magic happens. Instead of using separate variables (and a mutex!) for the goroutine counter (how many `Done()` calls are still needed) and the waiter counter (how many goroutines are blocked on `Wait()`), it packs them both into one 64-bit integer. The high 32 bits track the main counter, and the low 32 bits track the waiters. This atomic variable allows multiple goroutines to update and read the state safely and efficiently without needing locks, in most cases. Pretty neat, huh?

Finally, the `sema` field is a semaphore used internally by the Go runtime. When a goroutine calls `Wait()` and the counter isn't zero, it essentially tells the runtime, "Okay, put me to sleep on this semaphore (`runtime_SemacquireWaitGroup`)." When the counter *does* hit zero (because the last `Done()` was called), the runtime is signaled to wake up *all* the goroutines sleeping on that semaphore (`runtime_Semrelease`). This `sema` field is key to understanding potential issues, as we'll see later.

In a nutshell, the `WaitGroup` lifecycle is:

1. Call `Add(n)` *before* starting your goroutines to tell the `WaitGroup` how many `Done()` calls to expect. A common pattern is `wg.Add(1)` right before each `go` statement.
2. Inside each goroutine, call `defer wg.Done()` *immediately* to ensure the counter is decremented when the goroutine finishes, no matter what.
3. Call `Wait()` where you need to block until all `n` goroutines have called `Done()`.

Now that we've peeked behind the curtain, let's talk about where things can go wrong.

Pitfalls of using the sync.WaitGroup in Golang

One of the most important lessons I've learned about Go is that you should always understand the lifecycle of a goroutine. Launching them is easy with `go`, but you *must* think about how they will eventually end. This is especially critical when using `WaitGroup`.

Deadlock due to improper counter management

Let's look at a common mistake. What if we forget to call `Done()` under certain conditions?

```go
package main

import (
    "fmt"
    "net/http"
    "sync"
)

func main() {
    wg := &sync.WaitGroup{}
    // Intend to wait for one goroutine
    wg.Add(1)
    go func() {
        // PROBLEM: Defer is missing!
        req, err := http.NewRequest("GET", "https://api.example.com/data",
        if err != nil {
            fmt.Println("Error creating request:", err)
            // We return early, wg.Done() is never called!
            return
        }
        resp, err := http.DefaultClient.Do(req)
        if err != nil {
            fmt.Println("Error sending request:", err)
            // We return early again, wg.Done() is never called!

            return
```

```
        }
        defer resp.Body.Close()
        // Only call Done() on the happy path
        wg.Done() // <<< If errors happen, this line is skipped!
    }()
```

In case of an error during request creation or sending, the `Done()` method is skipped. The `WaitGroup` counter never reaches zero, and our main goroutine calling `wg.Wait()` blocks indefinitely. Classic deadlock!

If you run similar code locally, or sometimes in the Go Playground, you'll eventually get this dreaded output:

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

goroutine 1 [sync.WaitGroup.Wait]:
sync.runtime_SemacquireWaitGroup(0xc0000121f0?)
    /usr/local/go-faketime/src/runtime/sema.go:110 +0x25
sync.(*WaitGroup).Wait(0x0?)
    /usr/local/go-faketime/src/sync/waitgroup.go:118 +0x48
main.main()
    /tmp/sandbox3310007909/prog.go:31 +0x6f
```

The fix is simple but crucial: use `defer wg.Done()` *at the very beginning* of the goroutine.

```
package main

import (
    "fmt"
    "net/http"
    "sync"
)

func main() {
    wg := &sync.WaitGroup{}
    wg.Add(1) // Add before go
    go func() {
        defer wg.Done() // Defer immediately! Now it *always* runs on exit.
```

```
        req, err := http.NewRequest("GET", "https://api.example.com/data",
        if err != nil {
            fmt.Println("Error creating request:", err)
            return // Done() will still run thanks to defer
        }
        resp, err := http.DefaultClient.Do(req)
        if err != nil {
            fmt.Println("Error sending request:", err)
            return // Done() will still run
        }
        defer resp.Body.Close()
        // No need for wg.Done() here anymore
        fmt.Println("Request successful (in theory)!")
    }()
    wg.Wait()
    fmt.Println("Main goroutine finished waiting.")
```

Okay, deadlock avoided. But wait, there's more! What about timeouts and cancellation?

Improper context cancellation when using WaitGroup

Our HTTP request example still has a subtle but dangerous problem: we're not passing a `context.Context`. The `http.DefaultClient` might hang forever waiting for a response if the network is slow or the server is unresponsive. If the HTTP call blocks, `wg.Done()` never runs, and `wg.Wait()` blocks forever. Back to deadlock city!

So how can we add a timeout to `wg.Wait()` itself? You might see this pattern suggested online or by an LLM:

```
package main

import (
    "fmt"
    "net/http"
    "sync"
    "time"
)

func main() {
```

```
11      wg := &sync.WaitGroup{}
12      wg.Add(1)
13      go func() {
14          defer wg.Done() // Done is deferred correctly...
15          // ...BUT this HTTP call has no timeout / context! It could block f
16          req, err := http.NewRequest("GET", "http://httpbin.org/delay/10", n
17          if err != nil {
18              fmt.Println("Error creating request:", err)
19              return
20          }
21          fmt.Println("Sending request...")
22          resp, err := http.DefaultClient.Do(req) // This blocks...
23          if err != nil {
24              fmt.Println("Error sending request:", err)
25              return
26          }
27          defer resp.Body.Close()
28          fmt.Println("Request finished.") // ...potentially never reaching h
29      }()
```

This *seems* like it solves the main goroutine blocking forever. We use a `select` statement with a timeout. If `wg.Wait()` finishes within 5 seconds, the `done` channel is closed and we proceed. If 5 seconds pass, the `time.After` case triggers and we print a timeout message. Problem solved?

No! This is a trap! We've introduced a potential **goroutine leak**.

Think about what happens in the timeout case:

1. The `select` statement in `main` proceeds because `time.After` fires.
2. The goroutine running `wg.Wait()` is *still blocked* waiting for the counter to hit zero. Remember `runtime_SemacquireWaitGroup`? It's stuck there.
3. The original worker goroutine (doing the HTTP request) might *also* still be blocked, waiting indefinitely on the network call because we didn't give it a context with a deadline or cancellation signal. `defer wg.Done()` hasn't run yet!

So, `main` continues, but we've potentially left *two* goroutines hanging around, consuming resources, unable to finish. They are leaked. This is bad, especially in long-running server applications. Like I said earlier: always reason about how your goroutines *end*!

The `WaitGroup` itself doesn't support cancellation. `wg.Wait()` is fundamentally a blocking call until the counter reaches zero. The *real* fix is to make sure the *work* being done inside the goroutines can be cancelled.

Let's bring `context.Context` to the rescue:

```
1   package main
2
3   import (
4       "context"
5       "fmt"
6       "net/http"
7       "sync"
8       "time"
9   )
10
11  func main() {
12      wg := &sync.WaitGroup{}
13
14      // Create a context that will be cancelled after 3 seconds
15      ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
16      defer cancel() // Good practice to call cancel, though timeout does it
17
18      wg.Add(1)
19      go func() {
20          defer wg.Done() // Ensure Done is called
21
22          // Create the request *with the context*. This is key!
23          req, err := http.NewRequestWithContext(ctx, "GET", "http://httpbin.
24          if err != nil {
25              // Note: Error checking context errors might be needed in real
26              fmt.Println("Error creating request:", err)
27              return
28          }
29
```

You can run the code in Go Playground. We now pass a context with a timeout to `http.NewRequestWithContext`, and the `http.Client` respects this context. If the request takes longer than the context's deadline, `Do` will return an error (usually `context.DeadlineExceeded`). Crucially, the goroutine *unblocks* and proceeds to its end, executing the deferred `wg.Done()`.

Now, `wg.Wait()` is safe to call directly. We don't need the leaky `done` channel hack anymore. If all worker goroutines properly handle context cancellation, `wg.Wait()` will eventually return.

To be extra sure you're not leaking goroutines in your tests, you can use packages like `goleak`. Highly recommended!

So, context is vital. But what if we also need to handle errors from our goroutines and potentially cancel *other* running tasks if one fails? `WaitGroup` doesn't help there. Enter its more sophisticated cousin...

Alternatives to sync.WaitGroup: When to use `errgroup.Group`

So far, we've focused on just *waiting* for tasks to finish. We haven't really considered what happens if one of those tasks fails or if we want to collect errors. `WaitGroup` isn't concerned with errors.

This is fine if partial success is okay. Maybe you're firing off notifications and it's not critical if one fails.

Still, in scenarios like our GraphQL Federation example, failure is not an option. If fetching the price fails, the entire product response is incomplete and likely useless. We need to know about the error, and maybe we shouldn't even bother waiting for the availability call to finish if the price call already failed.

This is where `errgroup.Group` shines. It's designed specifically for running a group of tasks where you care about errors and want coordinated cancellation. It lives in the `golang.org/x/sync/errgroup` package.

Let's refactor our product example using `errgroup.Group`:

```
1   package main
2
```

```
import (
    "context"
    "errors"
    "fmt"
    "net/http" // Assuming we might use this context for real calls
    "time"

    // Import the errgroup package
    "golang.org/x/sync/errgroup"
)

// Product model remains the same
type Product struct {
    ID           int
    Availability string
    Price        float64
    // Use pointers or sync.Mutex for concurrent writes if needed,
    // but simple fields are okay for this demo if reads happen after Wait(
}

// Simulate fetching availability
func fetchAvailability(ctx context.Context, product *Product) error {
    // Simulate network call that respects context
    select {
    case <-time.After(100 * time.Millisecond):
        fmt.Println("Fetching availability for product 1...")
        product.Availability = "In Stock"
```

(You'll need `go get golang.org/x/sync/errgroup` to run this locally.)

Notice the key differences:

1. We use `errgroup.WithContext(context.Background())` to create the group and a derived context.
2. We use `eg.Go(func() error { ... })` to launch goroutines. `errgroup` manages the waiting internally (no manual `Add`/`Done`).
3. Our worker functions (`fetchAvailability`, `fetchPrice`) now accept and *respect* the `context.Context`. This is crucial for cancellation.
4. `eg.Wait()` blocks until all `eg.Go` functions complete. It returns the *first* non-nil error encountered.
5. The magic: If any function passed to `eg.Go` returns a non-nil error, `errgroup` automatically cancels the `ctx` it created. Any *other* running goroutines

launched via `eg.Go` that are properly checking `ctx.Done()` (like ours using `select`) will receive the cancellation signal and can exit early. This prevents wasted work.

This behavior is exactly what we often need for scenarios like API gateways or data fetching: fail fast, clean up, and report the first problem.

TL;DR: The WaitGroup & errgroup Cheat Sheet (For the Impatient Gopher)

Okay, that was a lot. Here's the quick version:

Use `sync.WaitGroup` when:

- You just need to wait for several independent goroutines to finish.
- You don't care too much if some of them error (or they handle errors internally).
- Partial success is acceptable.
- **Remember:** Call `wg.Add(1)` *before* `go`, `defer wg.Done()` *inside*, and ensure goroutines handle `context.Context` if they do blocking I/O to prevent leaks when using `wg.Wait()`.

Use `golang.org/x/sync/errgroup` when:

- You need to run several goroutines and get the **first error** that occurs.
- You want other goroutines to be **cancelled** automatically if one fails.
- Your goroutines perform work that should respect cancellation (e.g., network calls, long computations).
- **Remember:** Use `eg, ctx := errgroup.WithContext(...)`, launch with `eg.Go(func() error { ... })`, pass `ctx` into your work functions and check `ctx.Done()`, and check the `err` returned by `eg.Wait()`.

General Go Concurrency Wisdom:

- Always think about the **entire lifecycle** of your goroutines (how do they start, how do they *stop*?).
- Use `context.Context` religiously for cancellation and deadlines in any blocking or long-running operation.
- `defer wg.Done()` is your friend for `WaitGroup`.
- Test for goroutine leaks using tools like `goleak`.

A Go 1.25 Proposal might make WaitGroup more ergonomic

There's a **proposal** for Go 1.25 to make the use of `WaitGroup` more ergonomic and less error-prone.

Here's what the usage would look like:

```
1   var wg sync.WaitGroup
2   wg.Go(func() {
3       // perform task
4   })
5   wg.Wait()
```

The benefits of this proposal are:

- **Simplified Syntax:** Reduces the need for separate Add and go statements.
- **Reduced Errors:** Minimizes the risk of forgetting to call Done or calling Add after the goroutine has started.
- **Improved Readability:** Makes the code more concise and easier to understand.

Conclusion

When I first encountered `WaitGroup`, like many, I wondered, "Why doesn't `Wait()` just take a context?" It seemed like an obvious omission. But diving deeper, you realize `WaitGroup` is intentionally simple, a low-level primitive focused purely on counting. Its simplicity is its strength, but also its limitation. `errgroup` builds upon these ideas to provide the richer error handling and cancellation logic needed for more complex concurrent patterns.

The big takeaway? Go makes concurrency *accessible*, but not necessarily *easy* to get right. The primitives are powerful, but subtle bugs like deadlocks and goroutine leaks are easy to introduce if you don't understand the underlying mechanics (like context propagation and how `Wait` blocks). There's no magic linter that catches all these concurrency nuances; it requires careful design, testing (including leak detection!), and code review.

I hope this deeper dive into the world of `WaitGroup` and `errgroup` helps you navigate the sometimes-choppy waters of Go concurrency with a bit more confidence! May your goroutines always finish, and your channels never deadlock (unless you want them to, which is weird, but okay).

If you're exploring performance techniques in Go, you might also find this **deep dive into sync.Pool** useful. It covers the trade-offs of object reuse and when pooling can backfire.

You can follow me on **X** for more ramblings on Go, APIs, and the joy of building things.

And hey, if wrangling concurrent network requests and building high-performance API infrastructure sounds like your kind of fun, check out our open positions at WunderGraph! We're always looking for Gophers who enjoy a good challenge. Find us **here**.

Frequently Asked Questions (FAQ)

What is `sync.WaitGroup` used for in Go?

What are common mistakes with `sync.WaitGroup`?

Can you cancel a `WaitGroup` with a timeout?

When should I use `errgroup.Group` instead?

How do I avoid goroutine leaks with `WaitGroup`?

What's the proposed improvement to `WaitGroup` in Go 1.25?

Jens Neuse
CEO & Co-Founder at WunderGraph

Jens Neuse is the CEO and one of the co-founders of WunderGraph, where he builds scalable API infrastructure with a focus on federation and AI-native workflows. Formerly an engineer at Tyk Technologies, he created graphql-go-tools, now widely used in the open source community. Jens designed the original WunderGraph SDK and led its evolution into Cosmo, an open-source federation platform adopted by global enterprises. He writes about systems design, organizational structure, and how Conway's Law shapes API architecture.

0 reactions

0 comments

Write Preview Aa

Sign in to comment

Styling with Markdown is supported Sign in with GitHub

Previous
← When to Migrate from Cosmo OSS

Next
Why AI Needs A Common Language →

Support
Blog
Customers
Jobs

GraphQL API Gateway
Open Previews

LEGAL

Privacy Policy
Trust Center
Website Terms of Use
Cosmo Managed Service Terms
Cookie Policy
Cookie Preferences

SOC 2 Type II Certified

All services are online

Cookie Policy Cookie Preferences

RSS Atom JSON