Managing Groups of Goroutines in Go

Michal Bock Follow
Aug 30, 2019 · 7 min read ★

This image doesn't really have anything to do with this article, I just like quite it...

This is a brief description of the types and patterns in go used to manage groups of concurrent goroutines that I find myself using daily. Starting with the built in wait group, working our way to error group and beyond. It vaguely follows my own path of discovering these.

Lets start some goroutines

Starting a goroutine is really simple in go, you just need to prefix a function call with the keyword `go` . So as a newcomer to the language you might expect this snippet to just print numbers from 0 to 9 in some random order. I certailnly expected that.

```go
package main

import "fmt"

func main() {
        for i := 0; i < 10; i++ {
                go fmt.Printf("This is job: %v\n", i)
        }
}
```

simple.go hosted with ❤ by GitHub view raw

Try running it in the go playground.

However, if you try running this little program, you most likely won't see any output. This is because the way go programs run. Go always executes function called `main` in the package called `main` and once this function terminates, so does the whole program. What this programs ends up doing is starting a bunch of goroutines and then exiting before they have time to execute.

If you try running it a few times you might see something being logged out, this is because one or more of the goroutines managed to execute before the main function terminated. However, the takeaway here is that there are no guarantees that every goroutine you start will be executed, so you need to take care of that yourself.

Is there something in the standard library?

The standard library in go is pretty extensive, so quite often you might find that something that solves your problem is already there. That's also true in this case, as the `sync` package has a useful type called `WaitGroup`. I find it easiest to explain how types work by using examples, so let's have a look at an updated version of the program above that uses the wait group and works as expected.

```go
package main

import (
        "fmt"
        "sync"
)

func main() {
        var wg sync.WaitGroup
        for i := 0; i < 10; i++ {
                wg.Add(1)
                go func(jobID int) {
                        defer wg.Done()
                        fmt.Printf("This is job: %v\n", jobID)
                }(i)
        }
        wg.Wait()
}
```

waitgroup.go hosted with ❤ by GitHub view raw

Try it in playground.

Wait group is essentially just a threadsafe counter that exposes the following three methods:

- `Add` increases (or decreases) the value of the counter by the specified amount.
- `Done` decreases the value of the counter by one.
- `Wait` blocks until the value of the counter is 0.

To ensure that all our goroutines have been executed before the main function terminates, we add 1 to the counter before starting each goroutine. We also call `Done` in a deferred statement, in the function being executed in the goroutine, to make sure we decrease the counter once the function terminates (no matter what the result is). Finally, we call `Wait` after starting all the goroutines to wait for them to finish.

Note that it's important to pass in `i` as an argument to the function we are calling in a goroutine, as go evaluates arguments for this function in the current goroutine and only then starts the new one. If the function used `i` directly there would be a data race between multiple goroutines, because the for loop reuses `i` in each iteration. In that case you might for example see number 10 printed 10 times if all the goroutines started after the loop finished.

What about the errors?

The examples so far were very simple, but quite idealistic as they didn't expect any errors happening. This is not a good assumption to make in the real world, so we should make sure that we handle errors appropriately in production code. Luckily there is a package `golang.org/x/sync/errgroup` that provides a wrapper around the wait group that takes care of the errors.

Before exploring the error group let's first define a job function that can randomly fail. We we will make use of it later on.

```
1  func Job(jobID int) error {
2          if rand.Intn(12) == jobID {
3                  return fmt.Errorf("job %v failed", jobID)
4          }
5
6          fmt.Printf("Job %v done.\n", jobID)
7          return nil
8  }
```

errjob.go hosted with ❤ by GitHub view raw

This function just generates a random integer between 0 and 11 using the built in `rand` package. If the generated number is equal to the job id then the job fails and returns an error, otherwise it succeeds. Now let's look at an example that uses the error group and this job function.

```
1  func main() {
2          var eg errgroup.Group
3
```

```
        for i := 0; i < 10; i++ {
                jobID := i
                eg.Go(func() error {
                        return Job(jobID)
                })
        }

        if err := eg.Wait(); err != nil {
                fmt.Println("Encountered error:", err)
        }
        fmt.Println("Successfully finished.")
}
```

errgroup.go hosted with ❤ by GitHub view raw

The whole code in playground. Running it there seems to always succeed, so maybe try running it on your machine as well.

This is very similar to how we used the wait group before, maybe even simpler. We just create the group and then instead of starting goroutines ourselves and calling `Add` and `Done` functions, before starting and when finishing respectively, we just call the `Go` function on the error group. This starts the provided function in a goroutine and then immediately returns, so we can continue executing the loop. Note that in this case we create a copy of the variable `i` called `jobId`, that's local to the current iteration of the loop, to tackle the issue of `i` being reused between iterations.

As before after starting the goroutines we just wait for them to finish afterwards. In this case however, the `Wait` function returns an error. This is the first error encountered, so if more than one job fails, we would miss all but the first error. That's usually fine, as in most cases since we don' t want anything to fail.

What about the context?

In go it is a common practice to control the execution of functions using the `Context` interface from the built in `context` package. This interface has a `Done` function that returns a channel that is closed once the context is cancelled, what ussualy happens when the caller is no longer interested in the result. The functions that take context as an argument should therefore check the done channel to avoid doing any unnecessary work.

The `errgroup` package provides an alternative way to create an error group using context by calling the `WithContext` function. This function takes a context as an argument and returns an error group along with a context that is a child of the provided one (this means that whenever the parent context is cancelled, the child one is cancelled as well).

The context returned by the `WithContext` function is also cancelled once any of the functions started in the error group terminates with an error. This will allow us to stop the execution of the other jobs in the group as soon as we encounter the first error. (Note that the error group context is also cancelled once the first call to `Wait` returns as well, so the context can't be reused.)

So let's adapt our job now to use the context.

```go
func JobWithCtx(ctx context.Context, jobID int) error {
        select {
        case <-ctx.Done():
                fmt.Printf("context cancelled job %v terminting\n", jobID)
                return nil
        case <-time.After(time.Second * time.Duration(rand.Intn(3))):
        }
        if rand.Intn(12) == jobID {
                fmt.Printf("Job %v failed.\n", jobID)
                return fmt.Errorf("job %v failed", jobID)
        }

        fmt.Printf("Job %v done.\n", jobID)
        return nil
}
```

ctxjob.go hosted with ❤️ by GitHub view raw

Here we added the context as the first argument. Note that it's a convention in go to always have context as the first argument whenever a function uses it. At the top of the function we added a select statement that reads from the `Done` channel of the context and the channel returned by a call to `time.After`. The current time is sent on the time channel after a specified duration passes since its creation. This essentially performs a random sleep that can be interrupted by context cancelation what will cause the function to terminate. The rest of the function is same as before.

Now all we need to do in our main function to make use of the context is to create error group using the `WithContext` method and pass the returned context to all the jobs we start. So the code will look like this.

```go
func main() {
        eg, ctx := errgroup.WithContext(context.Background())

        for i := 0; i < 10; i++ {
                jobID := i
                eg.Go(func() error {
                        return JobWithCtx(ctx, jobID)
                })
        }

        if err := eg.Wait(); err != nil {
                fmt.Println("Encountered error:", err)
        }
        fmt.Println("Successfully finished.")
}
```

errgroupwithctx.go hosted with ❤️ by GitHub view raw

Full example in the go playground.

One more thing we can do is to try to ensure clean termination when the program receives an interrupt signal. To do that I like to use the context created by the following

function instead of the one returned by `context.Background()` that is never cancelled. This new context is cancelled as soon as the program receives an interrupt signal, what causes all its child contexts, including the error group one, to be cancelled as well, thus ensuring clean termination.

```go
func NewCtx() context.Context {
        ctx, cancel := context.WithCancel(context.Background())
        go func() {
                sCh := make(chan os.Signal, 1)
                signal.Notify(sCh, syscall.SIGINT, syscall.SIGTERM)
                <-sCh
                cancel()
        }()
        return ctx
}
```

newctx.go hosted with ❤ by GitHub view raw

With setup like this we should be able to deal with any error we encounter and shut down cleanly on interrupts. If you are not 100% sure about all of this setup, then try it on your own and have a look at the great documentation for all the types and functions mentioned here.

What if error group is not exactly what I need?

Error group is great, but sometimes we might need slightly different behaviour. For example we might be interested in returning a slice containing all errors we encounter instead of just the first one. The error group doesn't support such a behaviour, so you might be tempted to look at its source code and maybe try to adapt it to do that. That would be the right thing to do since the code is super simple only consisting of 66 lines of code including big docstrings. Swapping `sync.Once` for a mutex and a single error for a slice of errors will give us the behaviour we want.

If you want alternatives that are readily available, then you can try ones we built in the `github.com/uw-labs/sync` package. It contains two sub-packages one called `rungroup` with a type that behaves exactly like error group, except that it enforces the use of context and it cancels the underlying context as soon as any function started in the group terminates. This is useful for running pipelines that need all components running in order to function.

The other sub-package is called `gogroup` and it behaves like rungroup, except that a call to `Wait` returns as soon as the first function started in the group terminates. This is useful for implementing gRPC server with bidirectional streaming, as it allows the handler function to return when the server encounters an error even if there is a goroutine still waiting for messages from the client (examples can be found in this project).

I think this is all I have to say for now, please share your own adaptations of the error group in the comments. I would be interested to see them.