



Starting, and stopping, things

GopherCon South Africa 2025

<https://dave.cheney.net/>

Hello!



Section 1

The problem

Memory is a resource

If you obtain a resource, its on
you to release it

Memory is finite because
address space is finite

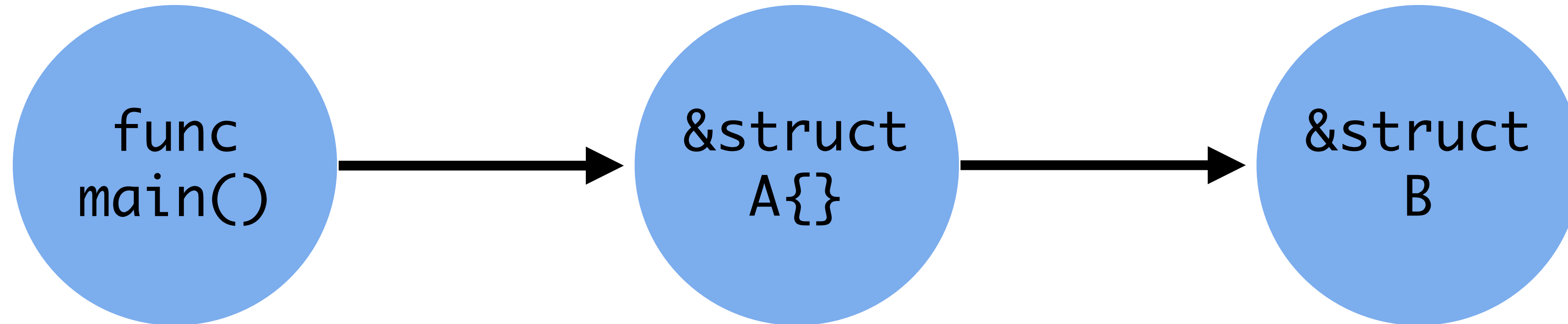
If your program allocates
memory, is it your responsibility
to release it?

Garbage collection frees the programmer from the burden of matching each allocation with a free

“Garbage collection provides the illusion of infinite memory using only finite memory.”

<https://go.dev/doc/gc-guide>

Reachability



All allocated memory must be
reachable from a root

Each goroutine is a GC root

*** Package level variables are also GC roots, but we'll ignore them for this discussion**

Goroutines are a resource,
because they are GC roots

You can close a file, or a unlock a lock, but how can you stop a goroutine?

Section 2

There's a go keyword, but no stop keyword

If a goroutine could be stopped at any time, what do we do with the resources that it's holding?

Option 1

Stopping a goroutine releases all the resources it owns



docs.oracle.com



ORACLE



Java™ Documentation

[Java Software](#)

[Java SE Downloads](#)

[Java SE 8 Documentation](#)

[Search](#)

Java Thread Primitive Deprecation

Why is `Thread.stop` deprecated?

Because it is inherently unsafe. Stopping a thread causes it to unlock all the monitors that it has locked. (The monitors are unlocked as the `ThreadDeath` exception propagates up the stack.) If any of the objects previously protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state. Such objects are said

Option 2

Ask the goroutine, politely, to stop

Stop vs stopping

```
func loop[T any](work chan T, stop chan struct{}) {  
    for {  
        select {  
            case <-stop:  
                return  
            case w := <-work:  
                // process work  
        }  
    }  
}
```

Stop, demo time

`cmd/stop-work`

* Hint: https://go.dev/ref/spec#Select_statements

There is a difference between
asking a goroutine to stop, and
knowing it has stopped

Implicit goroutine lifetime management

Stop, demo time

`cmd/http-server`

Goroutine lifetimes



Goroutine management is a
problem for middle aged Go
programs

Section 3

History

gopkg.in/tomb.v2

```
func main() {
    var t tomb.Tomb

    t.Go(func() error {
        for {
            select {
            case <-t.Dying():
                fmt.Println("shutting down because of", t.Err())
                return t.Err()
            default:
                // do something
            }
        }
    })

    t.Go(func() error {
        time.AfterFunc(2*time.Second, func() {
            t.Kill(fmt.Errorf("timeout"))
        })
        return nil
    })

    fmt.Println("waiting for goroutines to finish")
    t.Wait()
    fmt.Println("all goroutines finished")
}
```

golang.org/x/sync/errgroup

```
func main() {  
  
    // Create a new errgroup  
    var g errgroup.Group  
  
    // Add a new goroutine to the errgroup  
    g.Go(func() error {  
        // Do some work  
        return nil  
    })  
  
    // Add a new goroutine to the errgroup  
    g.Go(func() error {  
        // Do some work  
        return fmt.Errorf("oops, failed")  
    })  
  
    // Wait for all goroutines to finish  
    if err := g.Wait(); err != nil {  
        fmt.Println("At least one goroutine failed:", err)  
    }  
}
```

Use a group

Link



```
var g group.Group
{
    ctx, cancel := context.WithCancel(context.Background())
    g.Add(func() error {
        return sm.Run(ctx)
    }, func(error) {
        cancel()
    })
}
```



There is no one correct solution

(But probably quite a few *incorrect* solutions)

Managing goroutines is the *Jazz*
standard of Go programming

Section 4

A solution

```
package group // import "github.com/pkg/group"
```

```
type G struct {  
    // Has unexported fields.  
}
```

G manages the lifetime of a set of goroutines from a common context. The first goroutine in the group to return will cause the context to be canceled, terminating the remaining goroutines.

```
func New(opts ...Option) *G  
func (g *G) Add(fn func(context.Context) error)  
func (g *G) Wait() error
```

Stop, demo time

github.com/pkg/group/example_test.go

```
package main

import (
    "context"
    "fmt"
    "os"
    "time"

    "github.com/pkg/group"
)

func main() {
    if err := run(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}

func run() error {
    ctx, _ := signal.NotifyContext(context.Background(), syscall.SIGTERM, os.Interrupt)
    g := group.New(group.WithContext(ctx))
    g.Add(newHTTPService().Run)
    g.Add(newRPCService().Run)
    g.Add(newMetricsService().Run)
    return g.Wait()
}
```

```
type Service struct{}

func newService() *Service { return &Service{} }

func (svc *Service) Run(ctx context.Context) error {
    if err := svc.start(ctx); err != nil {
        return fmt.Errorf("failed to start: %w", err)
    }
    // run returns when context is done or on error
    return svc.run(ctx)
}

func (svc *Service) start(ctx context.Context) error {
    // initialise
    return nil
}

func (svc *Service) run(ctx context.Context) error {
    for {
        select {
        case <-ctx.Done():
            return ctx.Err()
        default:
            // do work
            time.Sleep(100 * time.Millisecond)
        }
    }
}
```

Conclusion

Goroutines are resources

Never start a goroutine without
knowing when it will stop

You cannot make a goroutine
stop, you have to ask politely

A goroutine's raison d'état is
work

Leave concurrency to the caller

That's it!

Go write some code that makes you proud!



Sample code

Slides

