Golang Estonia

Production Ready Concurrency

Golang Estonia Meetup Group

By the community, for the community

- Build a community
- Give professional content
- Share experiences
- Foster professional development
- Support interaction with experts

Get Involved

- *Join Gopher Slack #estonia*
- *Become a Speaker*
  - *Lightning talks (5-10 min)*
  - *Workshops*
  - *Talks (~20 min)*
- *Host (talk to your company)*
- *Become a Sponsor (e.g. sponsor mugs, raffles)*
- *Become a Co-Organizer*

Production Ready Concurrency

Egon Elbre
@egonelbre

Disclaimer

The following describes how to write a system
where it's hard to make mistakes and easy to test.

This occasionally comes at the
cost of performance and lines of code.

PS: There are plenty of exceptions,
I will skip over most of them.

Outline

Rules of thumb

Choosing primitives

A few custom primitives

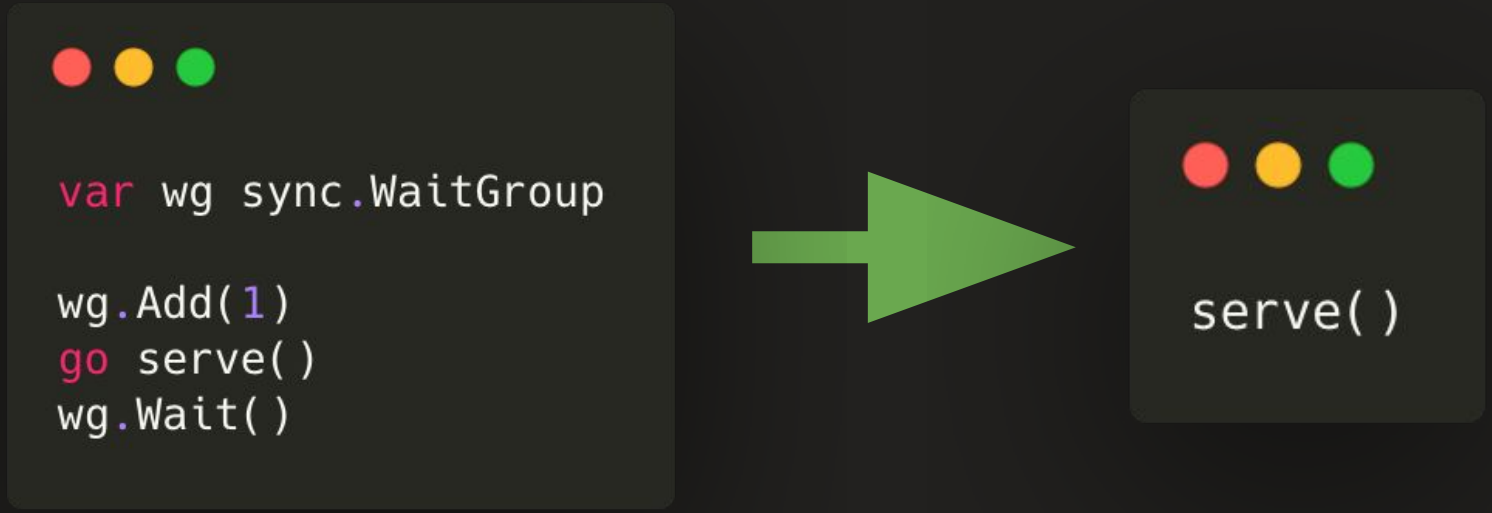
Q & A

Rules of Thumb

No global variables

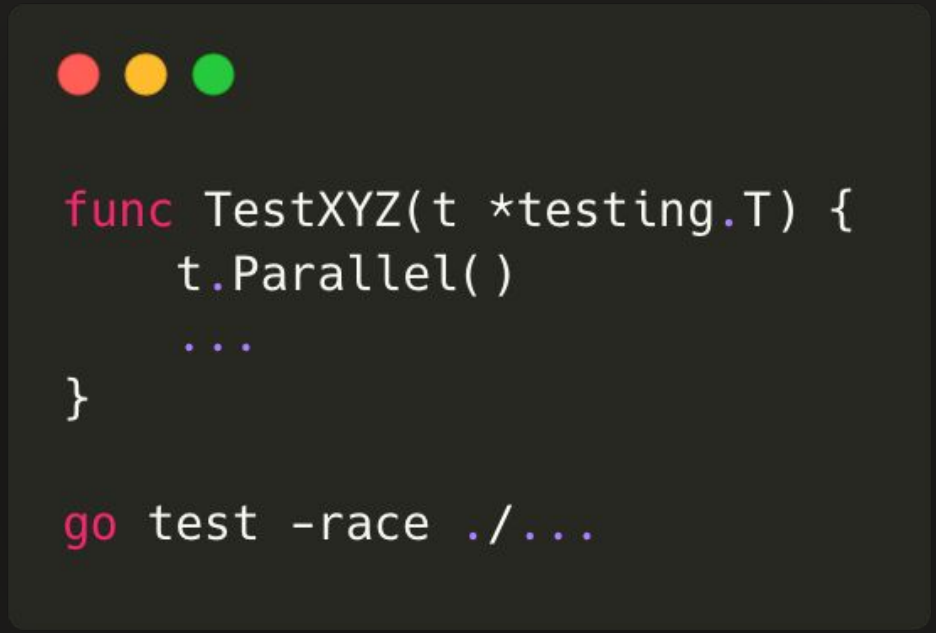
including global flags, caches, pools, logging etc. make them scoped
they all complicate testing

Use concurrency only when needed



System without concurrency is much easier to debug, test and understand.

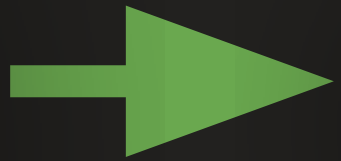
but, all your tests should use t.Parallel()



```
func TestXYZ(t *testing.T) {
    t.Parallel()
    ...
}

go test -race ./...
```

Prefer synchronous API

```
server.Start(ctx)
server.Stop()
server.Wait()
```

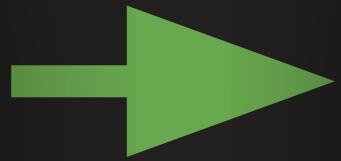


```
server.Run(ctx)
```

It's obvious how to turn sync to async, but the reverse isn't always true.

Know when things stop

```
go listenHTTP()
go listenGRPC()
go listenDebug()
select {}
```

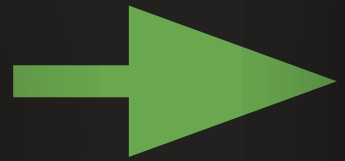


```
var endpoints errgroup.Group
endpoints.Go(listenHTTP)
endpoints.Go(listenGRPC)
endpoints.Go(listenDebug)
return endpoints.Wait()
```

At the end of your tests, all goroutines you started should have finished.

Cancellation-Aware

```
time.Sleep(time.Second)
```



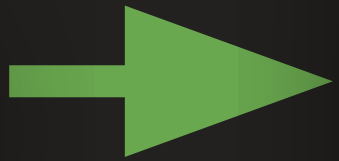
```
tick := time.NewTimer(time.Second)
defer tick.Stop()

select {
case <-tick.C:
case <-ctx.Done()
    return ctx.Err()
}
```

Cancellation handling is required to have everything shutdown fast on Ctrl-C

Cancellation-Aware

```
time.Sleep(time.Second)
```

often indicates a problem

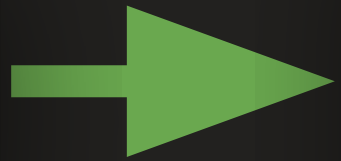


```
tick := time.NewTimer(time.Second)
defer tick.Stop()

select {
case <-tick.C:
case <-ctx.Done()
    return ctx.Err()
}
```

Context handling is required to have everything shutdown fast on Ctrl-C

Cancellation-Aware - Part 2

```
for _, f := range files {
    data, err := os.ReadFile(f)
    ...
}
```

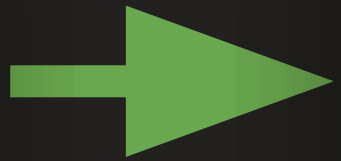


```
for _, f := range files {
    if err := ctx.Err(); err != nil {
        return err
    }
    data, err := os.ReadFile(f)
    ...
}
```

Anything taking more than 50ms should be ctx.Err aware.

Avoid worker pools

```
work := make(chan string, 10)
for k := 0; k < 10;k ++ {
    go func () {
        for _, w := range work {
            process(w)
        }
    }()
}

for _, item := range items {
    work <- item
}
```

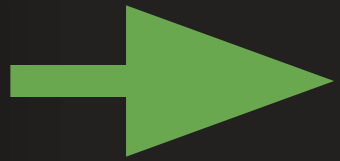


```
limiter := newLimiter(10)
for _, item := range items {
    item := item
    limiter.Go(func() {
        process(item)
    })
}
limiter.Wait()
```

Worker pools make stack traces harder to read, use more resources and are often slower.

Avoid polling state

```
lastKnown := 0
for {
    time.Sleep(time.Second)
    t.mu.Lock()
    if lastKnown != t.current {
        process(t.current)
        lastKnown = t.current
    }
    t.mu.Unlock()
}
```



This would be a whole talk

Wastes resources and makes go runtime work harder.
Also it's slower to respond to changes to the value.

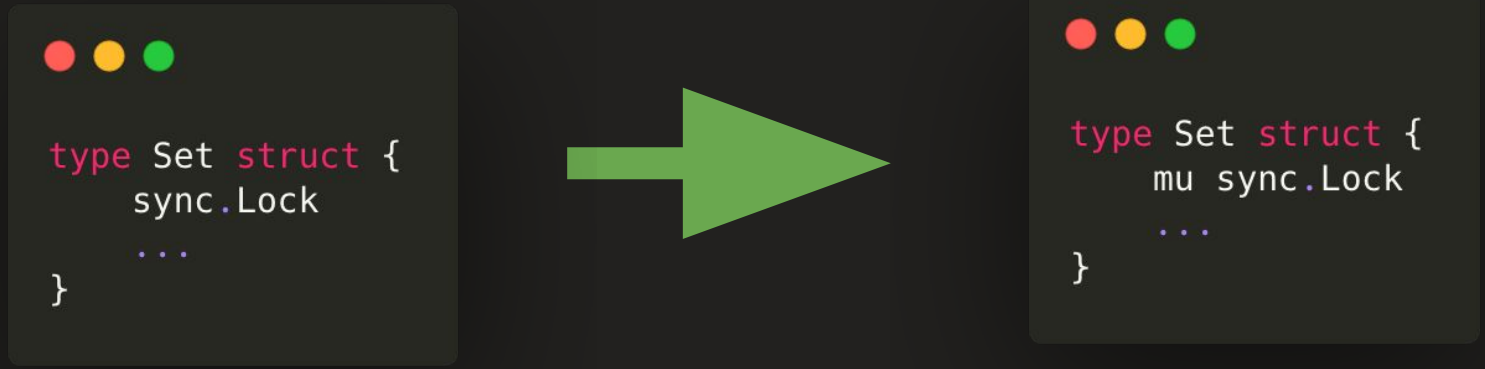
Defer post-actions (unlock, wait, close)

```
for _, item := range items {
    service.mu.Lock()
    service.process(item)
    service.mu.Unlock()
}
```

```
for _, item := range items {
    func() {
        service.mu.Lock()
        defer service.mu.Unlock()
        service.process(item)
    }()
}
```

It's clearer when a Unlock is missing.
Code modifications are less likely to introduce a bug.

Don't expose your locks



This is a quick way to get deadlocks.

Name your goroutines

```
labels := pprof.Labels("worker", "purge")
pprof.Do(ctx, labels,
    func(ctx context.Context) {
        ...
    })
```

Significantly helps with debugging.
See more at https://rakyll.org/profiler-labels/

Rules of Thumb: Summary

Use only when required

Prefer Synchronous API (async can be optional)

Know when things stop

Context aware code

No worker pools & polling

Defer post-actions

Don't expose your locks

Name your goroutines

Choosing your

concurrency primitives

Order of preference

1. no-concurrency
2. golang.org/x/sync, sync.Once
3. custom solution (or library)
4. sync.Mutex (special cases)

↓↓↓ *only for implementing custom solutions* ↓↓↓

5. sync.Map, sync.Pool (need a typesafe wrapper)
6. sync.WaitGroup
7. chan, go func() {
8. sync.Mutex, sync.Cond
9. atomics

Problem #1: go func() {

```
func (server *Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    ...
    go func() {
        res, err := server.db.ExecContext(r.Context(), "INSERT ...")
        ...
    }()
    ...
}

func main() {
    db, err := openDB()
    defer db.Close()

    err := server.Run(ctx)
    ...
}
```

Problem #1: go func() {

```go
func (server *Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    ...
    go func() {
        res, err := server.db.ExecContext(r.Context(), "INSERT ...")
        ...
    }()
    ...
}

func main() {
    db, err := openDB()
    defer db.Close()

    err := server.Run(ctx)
    ...
}
```

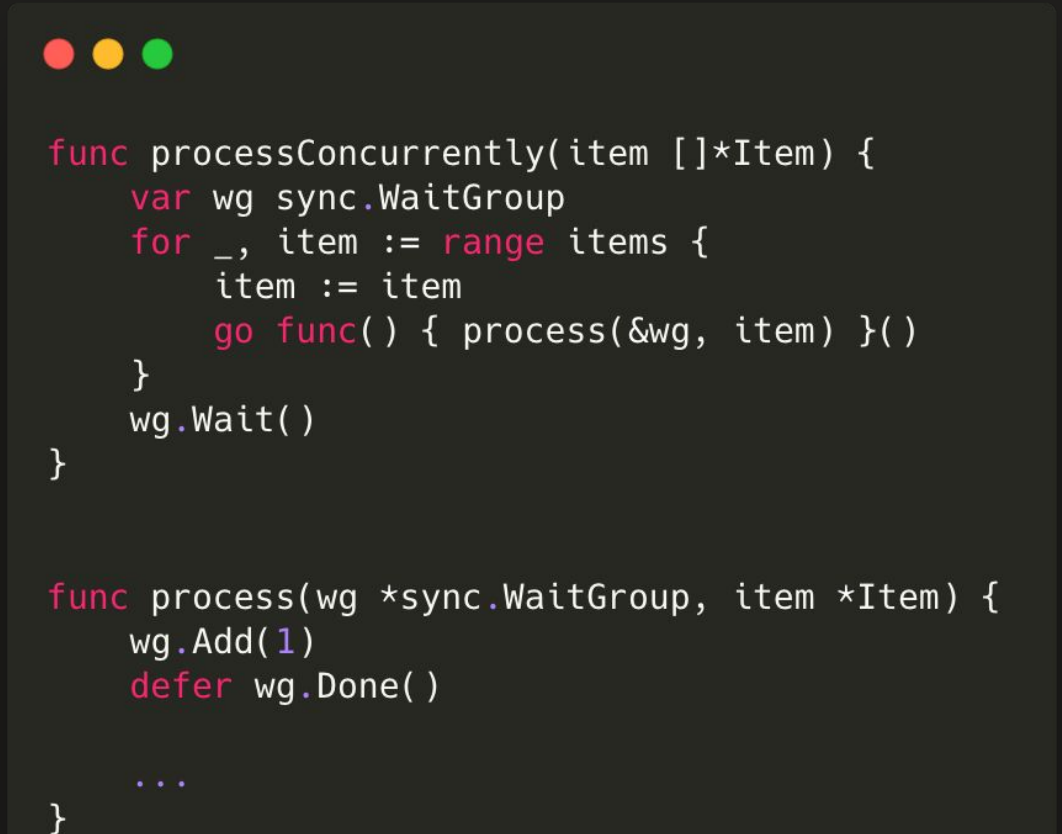
`db` might be closed here

Problem #1: go func() {

```go
func (server *Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    ...
    go func() {
        res, err := server.db.ExecContext(r.Context(), "INSERT ...")
        ...
    }()
    ...
}

func main() {
    db, err := openDB()
    defer db.Close()

    err := server.Run(ctx)
    ...
}
```

This context will get cancelled.

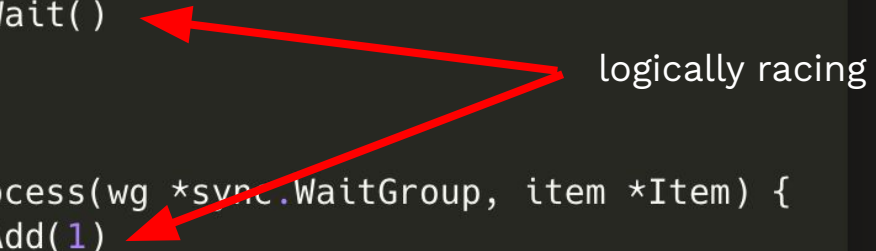
Problem #2: sync.WaitGroup



```
func processConcurrently(item []*Item) {
    var wg sync.WaitGroup
    for _, item := range items {
        item := item
        go func() { process(&wg, item) }()
    }
    wg.Wait()
}


func process(wg *sync.WaitGroup, item *Item) {
    wg.Add(1)
    defer wg.Done()

    ...
}
```

Problem #2: sync.WaitGroup

```go
func processConcurrently(item []*Item) {
    var wg sync.WaitGroup
    for _, item := range items {
        item := item
        go func() { process(&wg, item) }()
    }
    wg.Wait()
}


func process(wg *sync.WaitGroup, item *Item) {
    wg.Add(1)
    defer wg.Done()

    ...
}
```



Problem #3: sync.WaitGroup

```
var wg sync.WaitGroup
wg.Add(len(items))

for _, item := range items {
    item := item
    if filepath.Ext(item.Path) != ".go" {
        continue
    }
    go func() {
        defer wg.Done()
        process(item)
    }()
}

wg.Wait()
```

Problem #3: sync.WaitGroup

```go
var wg sync.WaitGroup
wg.Add(len(items))

for _, item := range items {
    item := item
    if filepath.Ext(item.Path) != ".go" {
        continue
    }
    go func() {
        defer wg.Done()
        process(item)
    }()
}

wg.Wait()
```

Tracking primitives

```
func (server *Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    ...
    err := server.pending.Go(r.Context(), func(ctx context.Context) {
        res, err := server.db.ExecContext(r.Context(), "INSERT ...")
        ...
    })
    ...
}

func (server *Server) Run(ctx context.Context) error {
    ...
    defer server.pending.Wait()
    ...
}
```

Tracking primitives

```
func (server *Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    ...
    err := server.pending.Go(r.Context(), func(ctx context.Context) {
        err := server.db.ExecContext(r.Context(), "INSERT ...")
        ...
    })
    ...
}

func (server *Server) Run(ctx context.Context) error {
    ...
    defer server.pending.Wait()
    ...
}
```

When server is shutting down
we should respond accordingly.

golang.org/x/sync/errgroup

waits all to stop on their own

```
var g errgroup.Group
g.Go(func() error {
    return public.Run(ctx)
})
g.Go(func() error {
    return private.Run(ctx)
})
err := g.Wait()
```

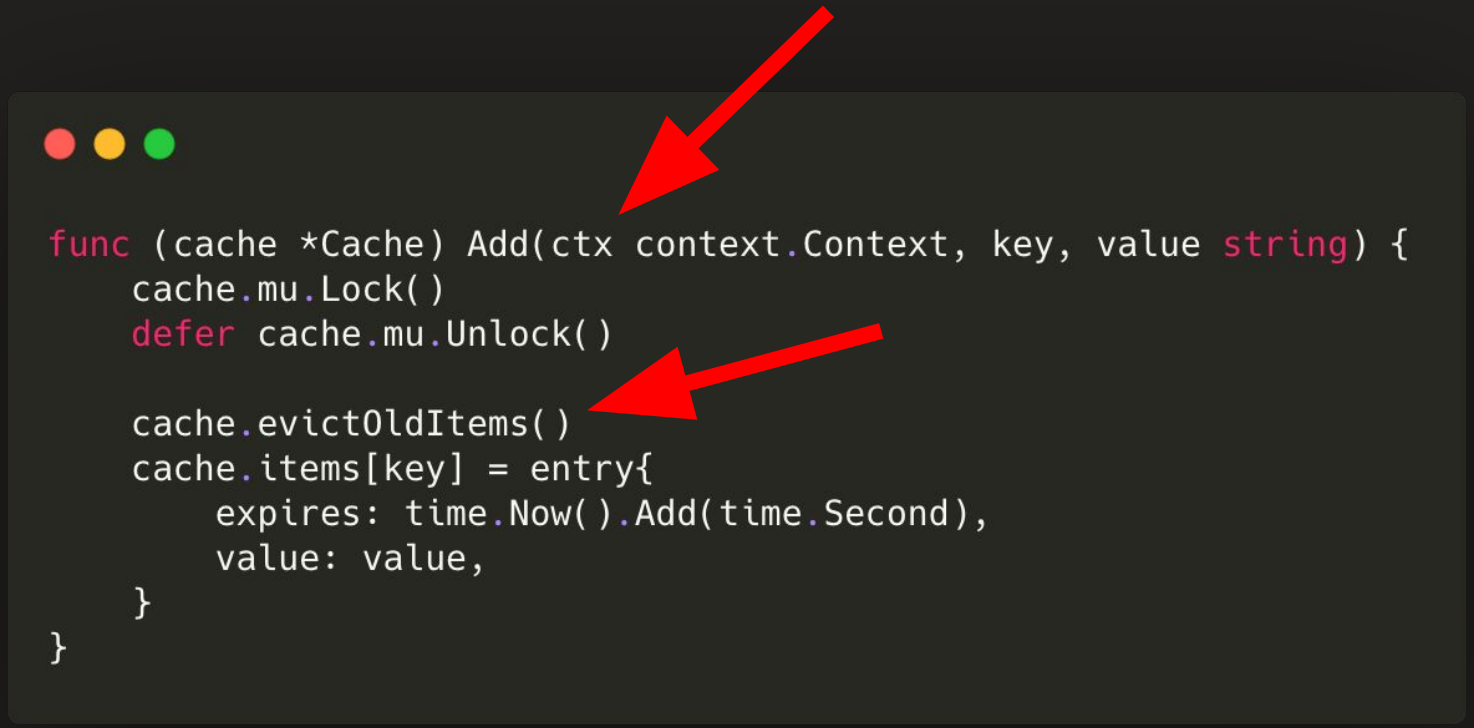
cancels ctx when one fails

```
g, ctx := errgroup.WithContext(ctx)
g.Go(func() error {
    return public.Run(ctx)
})
g.Go(func() error {
    return private.Run(ctx)
})
err := g.Wait()
```

Problem #4: sync.Mutex

```
func (cache *Cache) Add(ctx context.Context, key, value string) {
    cache.mu.Lock()
    defer cache.mu.Unlock()

    cache.evictOldItems()
    cache.items[key] = entry{
        expires: time.Now().Add(time.Second),
        value: value,
    }
}
```

Problem #4: sync.Mutex



```
func (cache *Cache) Add(ctx context.Context, key, value string) {
    cache.mu.Lock()
    defer cache.mu.Unlock()

    cache.evictOldItems()
    cache.items[key] = entry{
        expires: time.Now().Add(time.Second),
        value: value,
    }
}
```

Alternative to sync.Mutex make(chan *Item, 1)

```
func(cache *Cache) Add(ctx context.Context, key, value string) error {
    select {
    case <-ctx.Done():
        return ctx.Err()

    case state := <-cache.state:
        defer func() { cache.state <- state }()
        state.evictOldItems()

        state.items[key] = entry{
            expires: time.Now().Add(time.Second),
            value: value
        }
        return nil
    }
}
```

sync.Mutex

- only for protecting reading/writing to variables
- only in cases where code is O(N) or faster and N is small

Main issues:

- doesn't allow responding to ctx cancellation
- many nested locks is a quick way to hit lock inversions

sync.RWMutex

(similar to sync.Mutex, but worse)

- only use when sync.Mutex is **demonstrably** slower
  - sync.Mutex is almost always faster

additional issues:

- *when you have lots of readers and no writers, there's still cache contention between the readers because taking a read lock mutates the mutex (not scalable)*
- *a writer attempting to grab the lock blocks future readers from acquiring it. so long lived readers with infrequent writers causes long delays of no work*

Problem #5: chan

```
const workerCount = 100

var wg sync.WaitGroup
workQueue := make(chan *Item)

for i := 0; i < workerCount; i++ {
    wg.Add(1)
    go func() {
        defer wg.Done()
        for item := range workQueue {
            process(item)
        }
    }()
}

err := db.IterateItems(ctx, func(item *Item) {
    workQueue <- item
})

wg.Wait()
```

Problem #5: chan

```go
const workerCount = 100

var wg sync.WaitGroup
workQueue := make(chan *Item)

for i := 0; i < workerCount; i++ {
    wg.Add(1)
    go func() {
        defer wg.Done()
        for item := range workQueue {
            process(item)
        }
    }()
}

err := db.IterateItems(ctx, func(item *Item) {
    workQueue <- item
})

wg.Wait()
```

workQueue is not closed

Order of preference

1. no-concurrency
2. golang.org/x/sync, sync.Once
3. custom solution (or library)
4. sync.Mutex (special cases)

↓↓↓ *only for implementing custom solutions* ↓↓↓

5. sync.Map, sync.Pool (need a typesafe wrapper)
6. sync.WaitGroup
7. chan, go func() {
8. sync.Mutex, sync.Cond
9. atomics

My rule of thumb:
use these only in concurrency primitives

- make(chan X, N)
- go func() {
- sync.WaitGroup

they are error-prone, hard to test, difficult to review

** but they are also not "the-end-of-the-world bad"*

note: using `select` is fine

A few custom primitives

There are many variations.

You need to figure out what is
best for your codebase.

Batch processing a slice

Batch processing a slice

```go
func Parallel(n int, batchSize int, process func(low, high int)) {
    var wg sync.WaitGroup
    defer wg.Wait()

    if batchSize <= 0 {
        batchSize = 1
    }

    for low := 0; low < n; low += batchSize {
        low, high := low, low + batchSize
        if high > n {
            high = n
        }

        wg.Add(1)
        go func() {
            defer wg.Done()
            process(low, high)
        }()
    }
}
```

Batch processing a slice

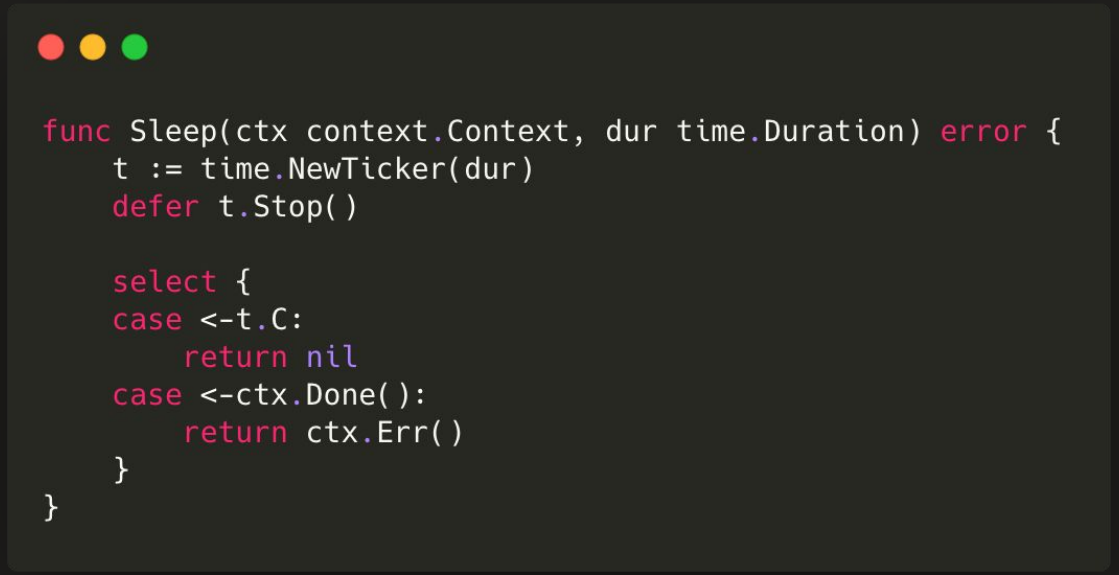
```go
func Parallel(n int, batchSize int, process func(low, high int)) {
    var wg sync.WaitGroup
    defer wg.Wait()

    if batchSize <= 0 {
        batchSize = 1
    }

    for low := 0; low < n; low += batchSize {
        low, high := low, low + batchSize
        if high > n {
            high = n
        }

        wg.Add(1)
        go func() {
            defer wg.Done()
            process(low, high)
        }()
    }
}
```

this example is probably not faster than single core

```go
var mu sync.Mutex
total := 0

Parallel(len(items), 256, func(low, high int) {
    price := 0
    for _, item := range items[low:high] {
        price += item.Price
    }

    mu.Lock()
    defer mu.Unlock()
    total += price
})
```

Sleeping

Sleeping



```go
func Sleep(ctx context.Context, dur time.Duration) error {
    t := time.NewTicker(dur)
    defer t.Stop()

    select {
    case <-t.C:
        return nil
    case <-ctx.Done():
        return ctx.Err()
    }
}
```

Sleeping

```go
func Sleep(ctx context.Context, dur time.Duration) error {
    t := time.NewTicker(dur)
    defer t.Stop()

    select {
    case <-t.C:
        return nil
    case <-ctx.Done():
        return ctx.Err()
    }
}
```

```go
if err := Sleep(ctx, time.Second); err != nil {
    return err
}
```

Retrying with backoff

Retrying with backoff

```
retry := NewRetry(10, time.Second/10, time.Second)
for retry.Next(ctx) {

}
if err := retry.Err(); err != nil {
    return err
}
```

```
), time.Second/10,
:.Context) error {
    ...
})
```

implementation is an exercise for the listener

Retrying with backoff

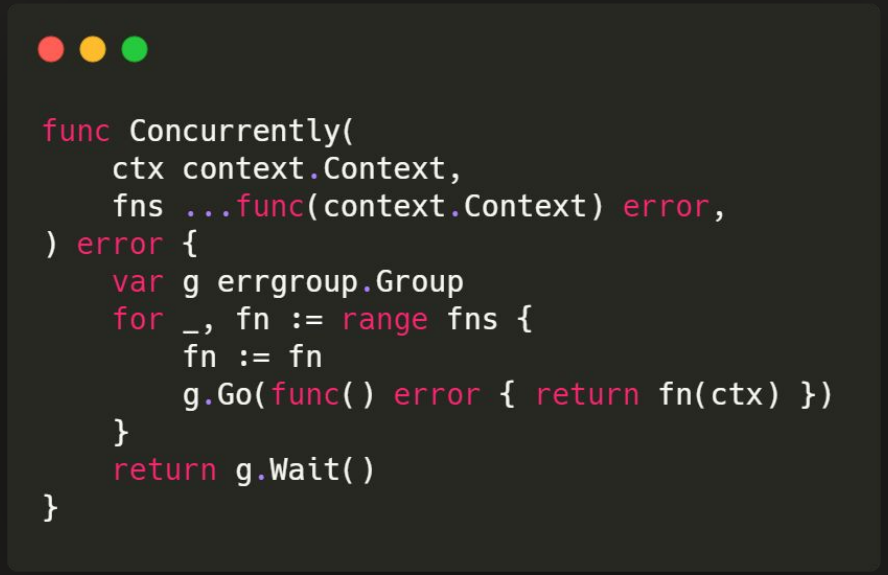
```
retry := NewRetry(10, time.Second/10, time.Second)
for retry.Next(ctx) {

}
if err := retry.Err(); err != nil {
    return err
}
```

```
err := Retry(ctx, 10, time.Second/10,
    func(ctx context.Context) error {
        ...
    })
```

implementation is an exercise for the listener

Running few things concurrently

Running few things concurrently



```go
func Concurrently(
    ctx context.Context,
    fns ...func(context.Context) error,
) error {
    var g errgroup.Group
    for _, fn := range fns {
        fn := fn
        g.Go(func() error { return fn(ctx) })
    }
    return g.Wait()
}
```

Running few things concurrently

```
func Concurrently(
    ctx context.Con
    fns ...func(con
) error {
    var g errgroup.
    for _, fn := ra
        fn := fn
        g.Go(func()
    }
    return g.Wait()
}
```

```
err := Concurrently(ctx,
    func(ctx context.Context) error {
        // ...
    },
    func(ctx context.Context) error {
        // ...
    },
    func(ctx context.Context) error {
        // ...
    },
)
```

Running few things concurrently

```
func Concurrently(
    ctx context.Cor
    fns ...func(cor
) error {
    var g errgroup.
    for _, fn := ra
        fn := fn
        g.Go(func()
    }
    return g.Wait()
}
```

```
err := Concurrently(ctx,
    func(ctx context.Context) error {
        // ...
    },
    func(ctx context.Context) error {
        // ...
    },
    func(ctx context.Context) error {
        // ...
    },
)
```

what should happen
when only one fails?

Running few things concurrently

```
func Concurrently(
    ctx context.Cor
    fns ...func(cor
) error {
    var g errgroup.
    for _, fn := ra
        fn := fn
        g.Go(func()
    }
    return g.Wait()
}
```

```
err := Concurrently(ctx,
    func(ctx context.Context) error {
        // ...
    },
    func(ctx context.Context) error {
        // ...
    },
    func(ctx context.Context) error {
        // ...
    },
)
```

what should happen
when only one fails?

what should happen
when one or more panics?

Waiting for a thing

Waiting for a thing

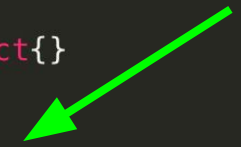
```
type Fence struct {
    create  sync.Once
    release sync.Once
    wait    chan struct{}
}

func (f *Fence) init()  {
    f.create.Do(func() {
        f.wait = make(chan struct{})
    })
}
```

```
func (f *Fence) Release() {
    f.init()
    f.release.Do(func(){
        close(f.wait)
    })
}

func (f *Fence) Released() chan struct{} {
    f.init()
    return f.wait
}

func (f *Fence) Wait(ctx context.Context) error {
    f.init()
    select {
    case <-f.Released():
        return nil
    case <-ctx.Done():
        return ctx.Err()
    }
}
```

Waiting for a thing

```go
type Fence struct {
    create  sync.Once
    release sync.Once
    wait    chan struct{}
}

func (f *Fence) init()  {
    f.create.Do(func() {
        f.wait = make(chan struct{})
    })
}
```

doesn't require a constructor



```go
func (f *Fence) Release() {
    f.init()
    f.release.Do(func(){
        close(f.wait)
    })
}

func (f *Fence) Released() chan struct{} {
    f.init()
    return f.wait
}

func (f *Fence) Wait(ctx context.Context) error {
    f.init()
    select {
    case <-f.Released():
        return nil
    case <-ctx.Done():
        return ctx.Err()
    }
}
```

Waiting for a thing

```
type Fence struct {
    create  sync.Once
    release sync.Once
    wait    chan struct{}
}

func (f *Fence) init()  {
    f.create.Do(func() {
        f.wait = make(chan struct{})
    })
}
```

```
func (f *Fence) Release() {
    f.init()
    f.release.Do(func(){
        close(f.wait)
    })
}

func (f *Fence) Released() chan struct{} {
    f.init()
    return f.wait
}

func (f *Fence) Wait(ctx context.Context) error {
    f.init()
    select {
    case <-f.Released():
        return nil
    case <-ctx.Done():
        return ctx.Err()
    }
}
```

in some cases you could want a value to passed along

Waiting for a thing

```go
type Fence struct {
    create  sync.Once
    release sync.Once
    wait    chan struct{}
}

func (f *Fence) init()  {
    f.create.Do(func() {
        f.wait = make(chan struct{})
    })
}
```

```go
func (f *Fence) Release() {
    f.init()
    f.release.Do(func(){
        close(f.wait)
    })
}

func (f *Fence) Released() chan struct{} {
    f.init()
    return f.wait
}

func (f *Fence) Wait(ctx context.Context) error {
    f.init()
    select {
    case <-f.Released():
        return nil
    case <-ctx.Done():
        return ctx.Err()
    }
}
```

optionally expose the channel

Waiting for a thing

```
var loaded Fence
var data map[string]int

err := Concurrently(ctx,
    func(ctx context.Context) error {
        defer loaded.Release()

        data, err := getData(ctx, url)
        if err != nil {
            return err
        }
        return nil
    },
```

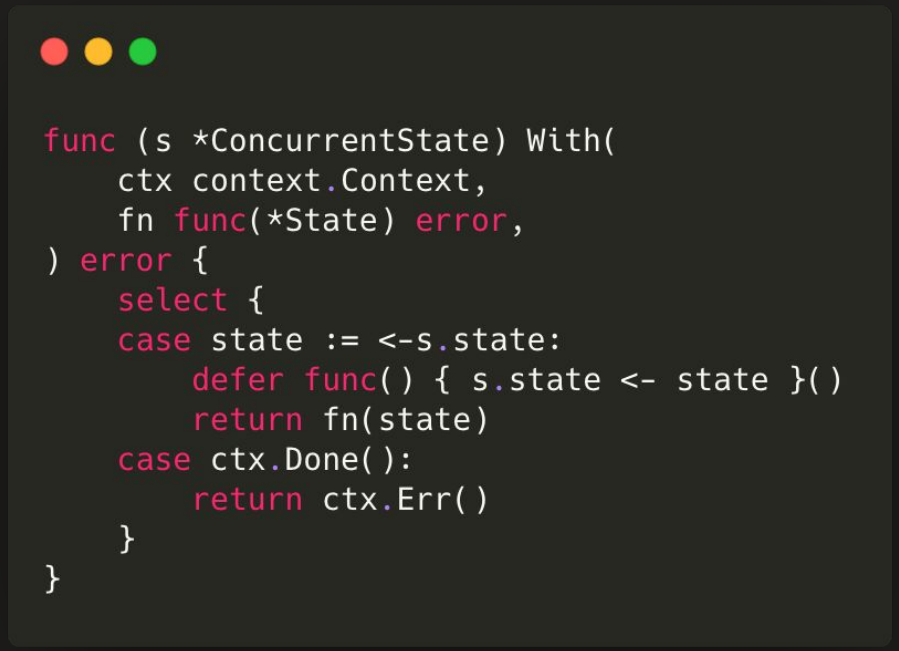
Waiting for a thing

```
var loaded Fence
var data map[string]int

err := Concurrently(ctx,
    func(ctx context.Context) error {
        defer loaded.Release()

        data, err := getData(ctx, url)
        if err != nil {
            return err
        }
        return nil
    },
```

```
    func(ctx context.Context) error {
        if err := loaded.Wait(ctx); err != nil {
            return err
        }
        return saveToCache(data)
    },
    func(ctx context.Context) error {
        if err := loaded.Wait(ctx); err != nil {
            return err
        }
        return processData(data)
    })
```

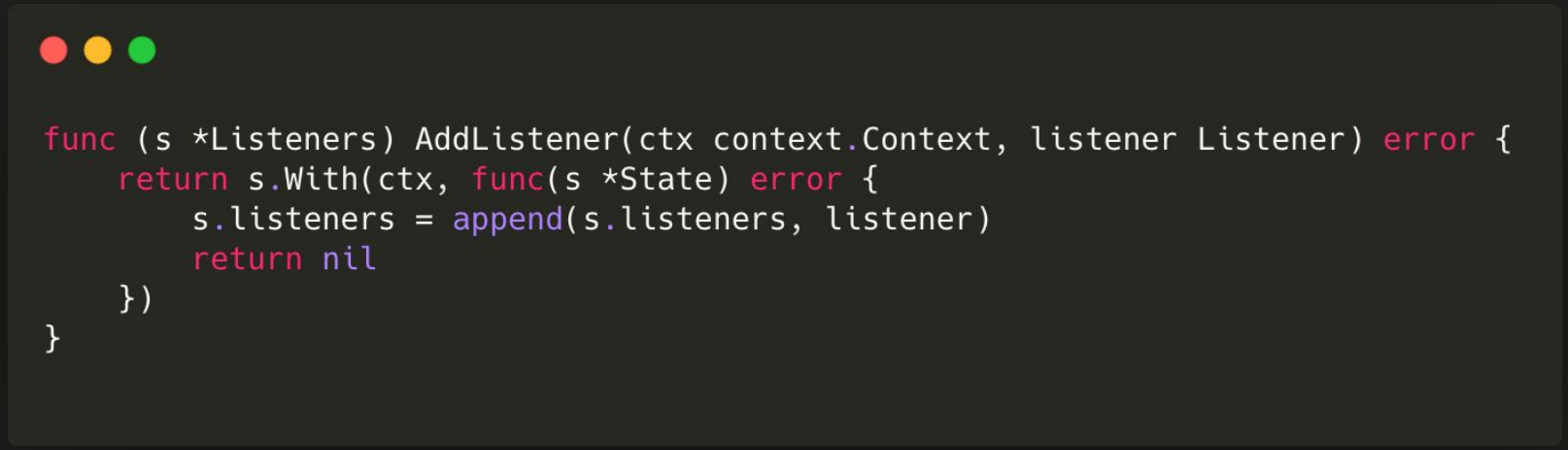
Protecting State

Protecting State



```go
func (s *ConcurrentState) With(
    ctx context.Context,
    fn func(*State) error,
) error {
    select {
    case state := <-s.state:
        defer func() { s.state <- state }()
        return fn(state)
    case ctx.Done():
        return ctx.Err()
    }
}
```

Protecting State



```go
func (s *Listeners) AddListener(ctx context.Context, listener Listener) error {
    return s.With(ctx, func(s *State) error {
        s.listeners = append(s.listeners, listener)
        return nil
    })
}
```

Limiter

Limiter

```go
type Limiter struct {
    limit   chan struct{}
    working sync.WaitGroup
}

func NewLimiter(n int) *Limiter {
    return &Limiter{limit: make(chan struct{}, n)}
}
```

Limiter

```
type Limiter struct {
    limit   chan struct{}
    working sync.WaitGrou
}

func NewLimiter(n int) *L
    return &Limiter{limit
}
```

```
func (lim *Limiter) Go(ctx context.Context, fn func()) bool {
    if ctx.Err() != nil {
        return false
    }

    select {
    case limiter.limit <- struct{}{}:
    case <-ctx.Done():
        return false
    }

    limiter.working.Add(1)
    go func() {
        defer func() {
            <-limiter.limit
            limiter.working.Done()
        }()

        fn()
    }()

    return true
}

func (lim *Limiter) Wait() {
    limiter.working.Wait()
}
```

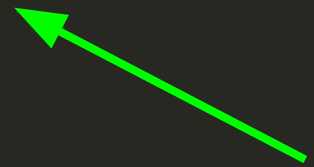
Async processes in a server

Async processes in a server

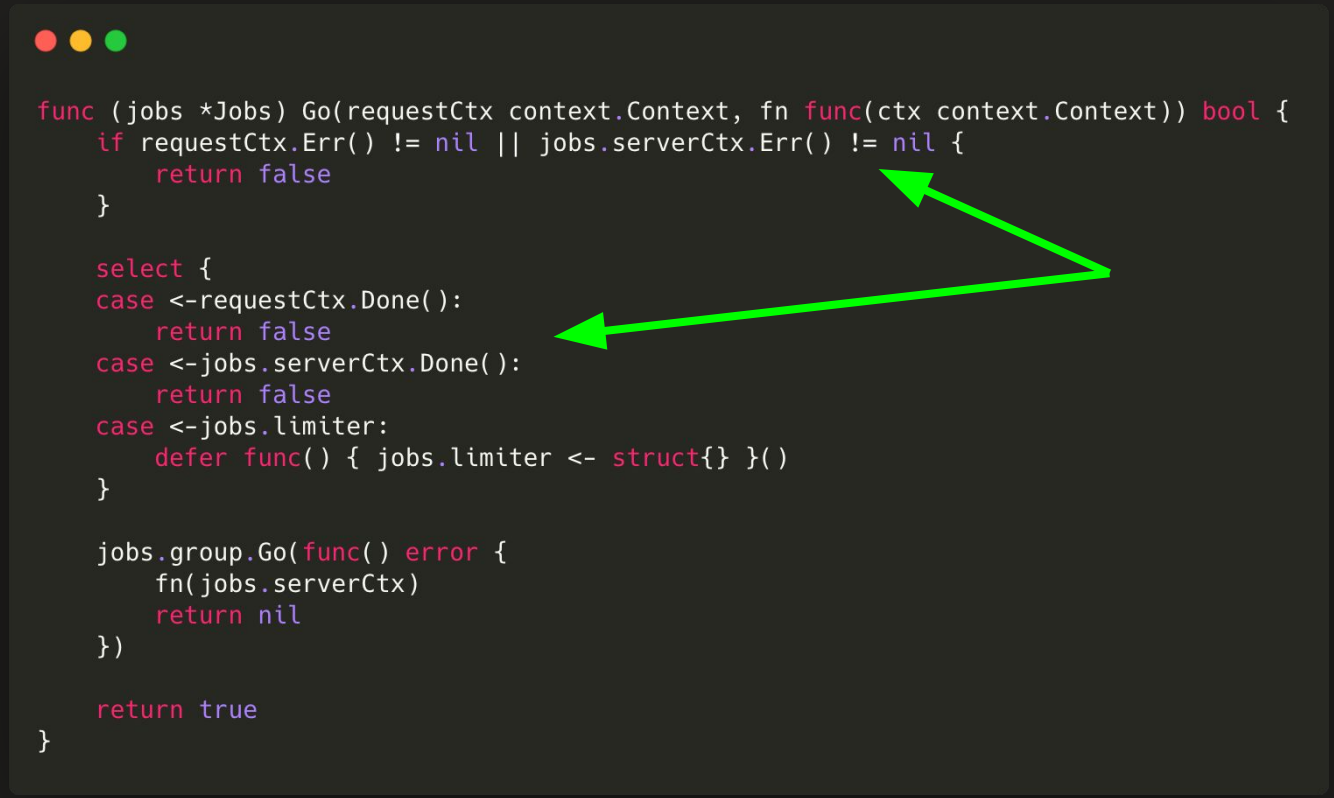
```go
func (server *Server) Run(ctx context.Context) error {
    server.pending = NewJobs(ctx)
    defer server.pending.Wait()

    // ... start listening on server ...
}


func (server *Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // ...
    started := server.pending.Go(r.Context(),
        func(ctx context.Context) {
            ... := server.db.ExecContext(ctx, ...)
        })
    if !started {
        // ... handle server shutting down or request canceld.
    }
    // ...
}
```

Async processes in a server

```go
type Jobs struct {
    serverCtx context.Context
    group     errgroup.Group
}

func NewJobs(serverCtx context.Context) *Jobs {
    return &Jobs{ serverCtx: serverCtx }
}

func (jobs *Jobs) Wait() { _ = jobs.group.Wait() }

func (jobs *Jobs) Go(requestCtx context.Context, fn func(ctx context.Context)) bool {
    if requestCtx.Err() != nil || jobs.serverCtx.Err() != nil {
        return false
    }

    jobs.group.Go(func() error {
        fn(jobs.serverCtx)
        return nil
    })

    return true
}
```



Async processes in a server (with limiter)



```
func (jobs *Jobs) Go(requestCtx context.Context, fn func(ctx context.Context)) bool {
    if requestCtx.Err() != nil || jobs.serverCtx.Err() != nil {
        return false
    }

    select {
    case <-requestCtx.Done():
        return false
    case <-jobs.serverCtx.Done():
        return false
    case <-jobs.limiter:
        defer func() { jobs.limiter <- struct{} }()
    }

    jobs.group.Go(func() error {
        fn(jobs.serverCtx)
        return nil
    })

    return true
}
```

Additional resources:

- Bryan C. Mills - Rethinking Classical Concurrency Patterns
  - https://www.youtube.com/watch?v=5zXAHh5tJqQ
  - https://drive.google.com/file/d/1nPdvhB0PutEJzdCq5ms6UI58dp50fcAN/view
- Allen B. Downey - Little Book of Semaphores
  - https://greenteapress.com/wp/semaphores/
- Understanding Real-World Concurrency Bugs in Go
  - https://songlh.github.io/paper/go-study.pdf
- **#bestpractices** in Gophers Slack

Thank You