How to correctly use package context

Advice and pitfalls

17 August 2017

Jack Lindamood
Software Engineer, Twitch

About the author

- Writing Go for 4 years
- Currently software engineer at Twitch
- Most backend at Twitch powered by Go
- Hundred(s) of repositories

How to correctly use context.Context in Go 1.7 (https://medium.com/@cep21/how-to-correctly-use-context-context-in-go-1-7-8f2c0fafdf39)

Talk Sections

- Why context exists
- Request cancellation
- Request scoped values

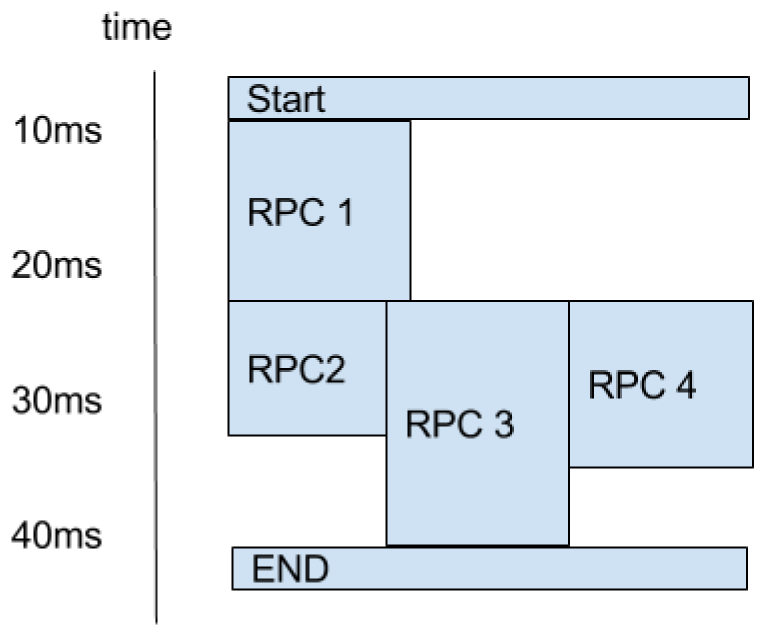
Why context exists

Problem that created context.Context

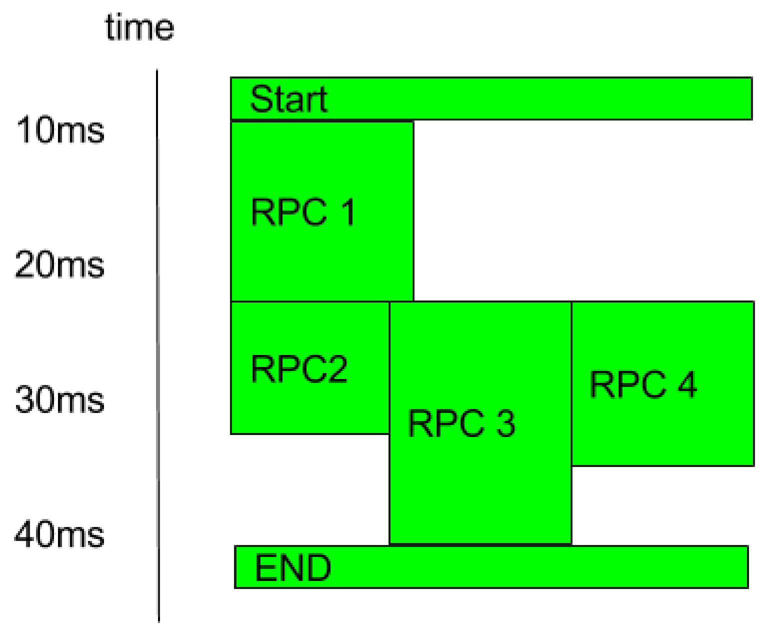
- Every long request should have a timeout
- Need to propagate that timeout across the request
- Let's say it's 3 seconds at the start of the request
- How much time is left in the middle of the request?
- Need to store that information somewhere so the middle of the request can stop

Problem expanded

- What if one request requires multiple RPC calls to resolve
- If one of those RPC calls fails, it may be worth failing the whole request

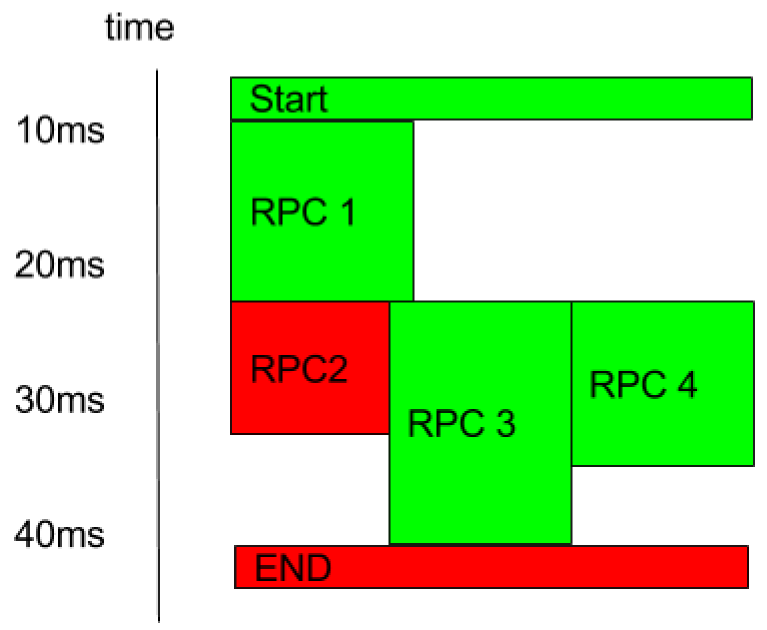

time
10ms
20ms
30ms
40ms
Start
RPC 1
RPC2
RPC 3
RPC 4
END

Good request



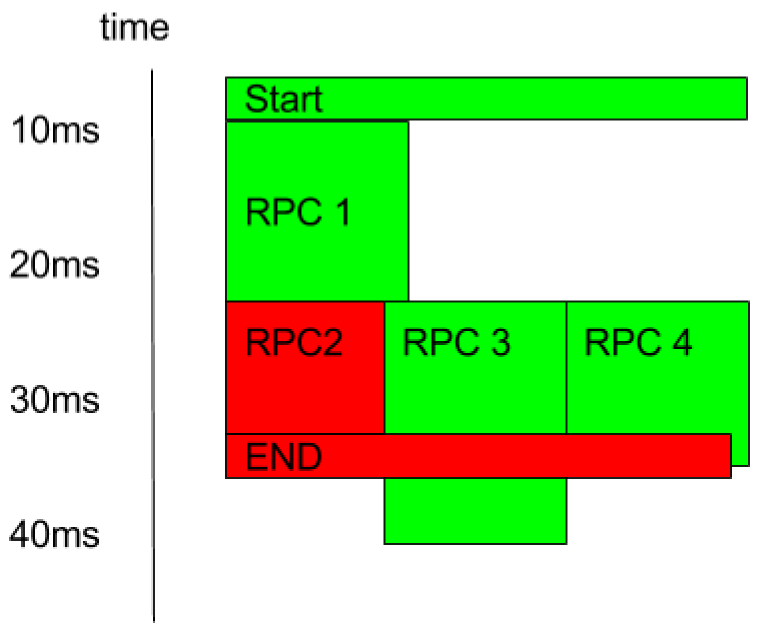
- Everything works, finishes in 40ms

Failed request



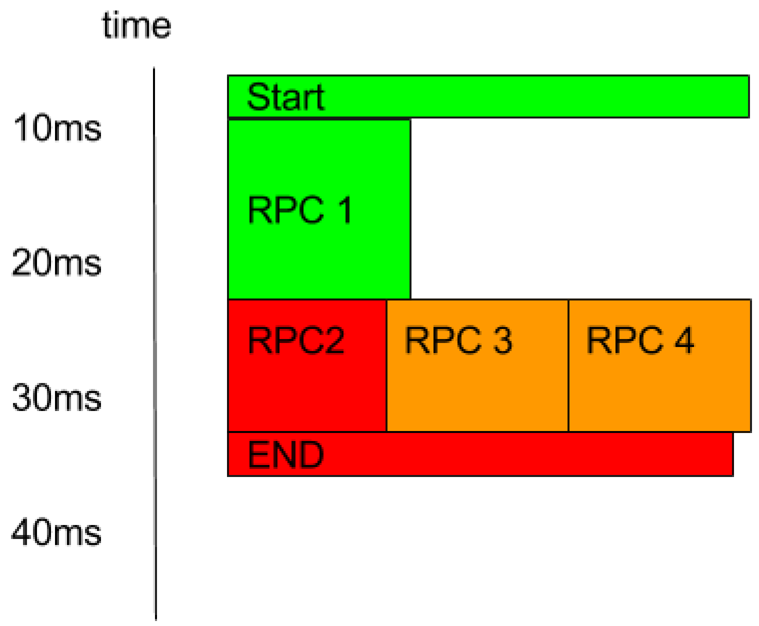
- RPC2 failed, still took 40ms

Failed request end early



- If RPC2 failed, just end early. Dangling RPC3

Failed request ideal



- If RPC2 fails, tell RPC 3 and 4 to end early

Solution

- Use object to signal when a request is over
- Includes hints on when the request is expected to end
- Channels naturally report when the request is done

Let's throw variables in there too

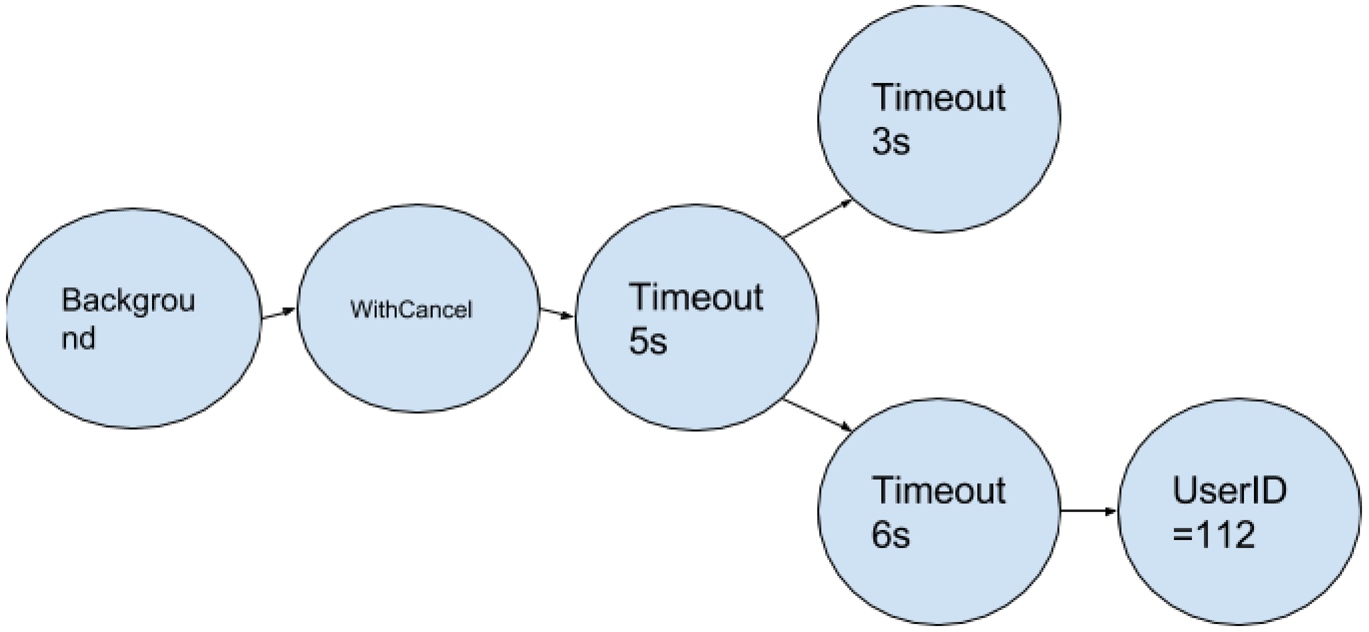
- No concept of thread/goroutine specific variables in Go
- Reasonable, since it becomes tricky when goroutines depend upon other goroutines
- Since context is threaded everywhere, throw variables on it to as a grab bag of information
- Very easy to abuse

context.Context implementation details

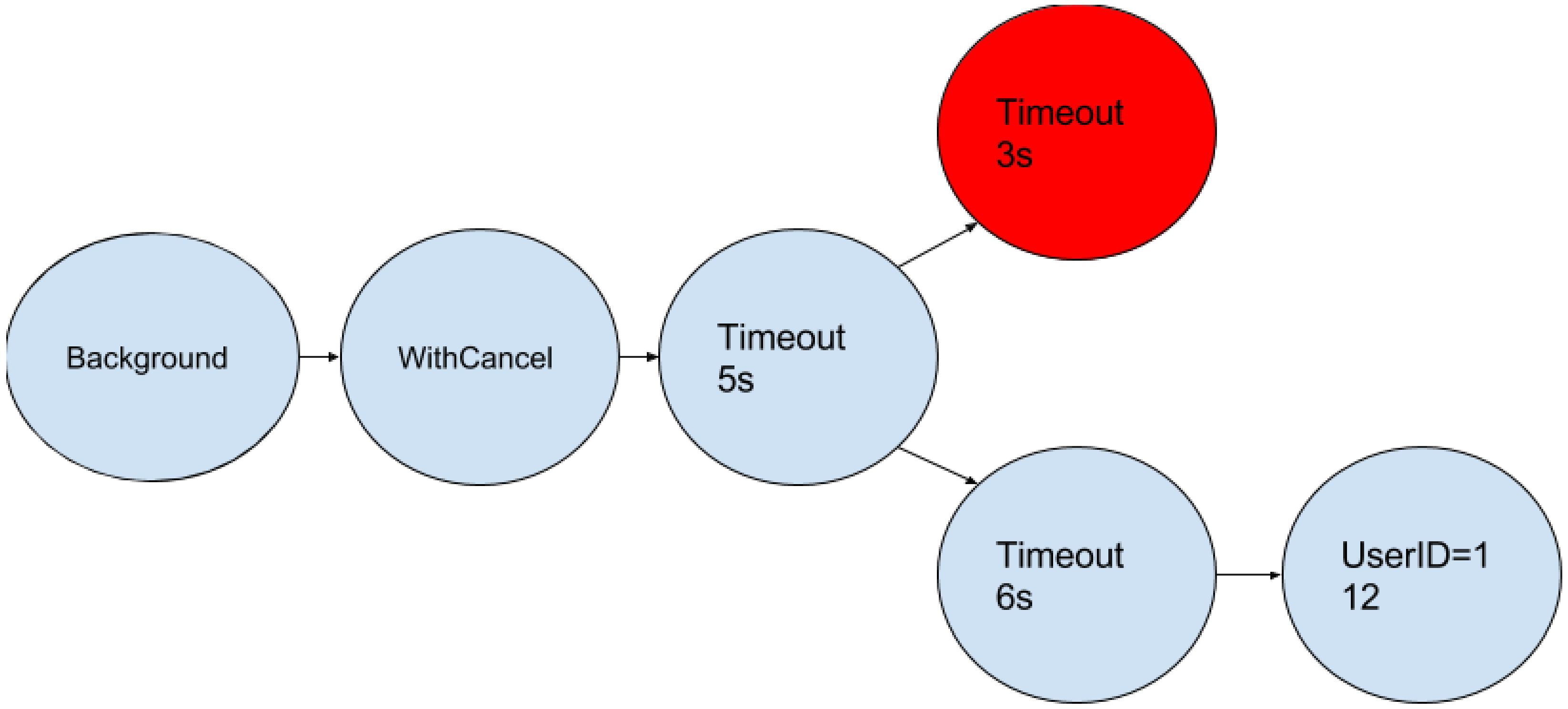
- Tree of immutable context nodes
- Cancellation of a node cancels all sub nodes
- Context values are a node
- Value lookup goes backwards up the tree

Example context chain

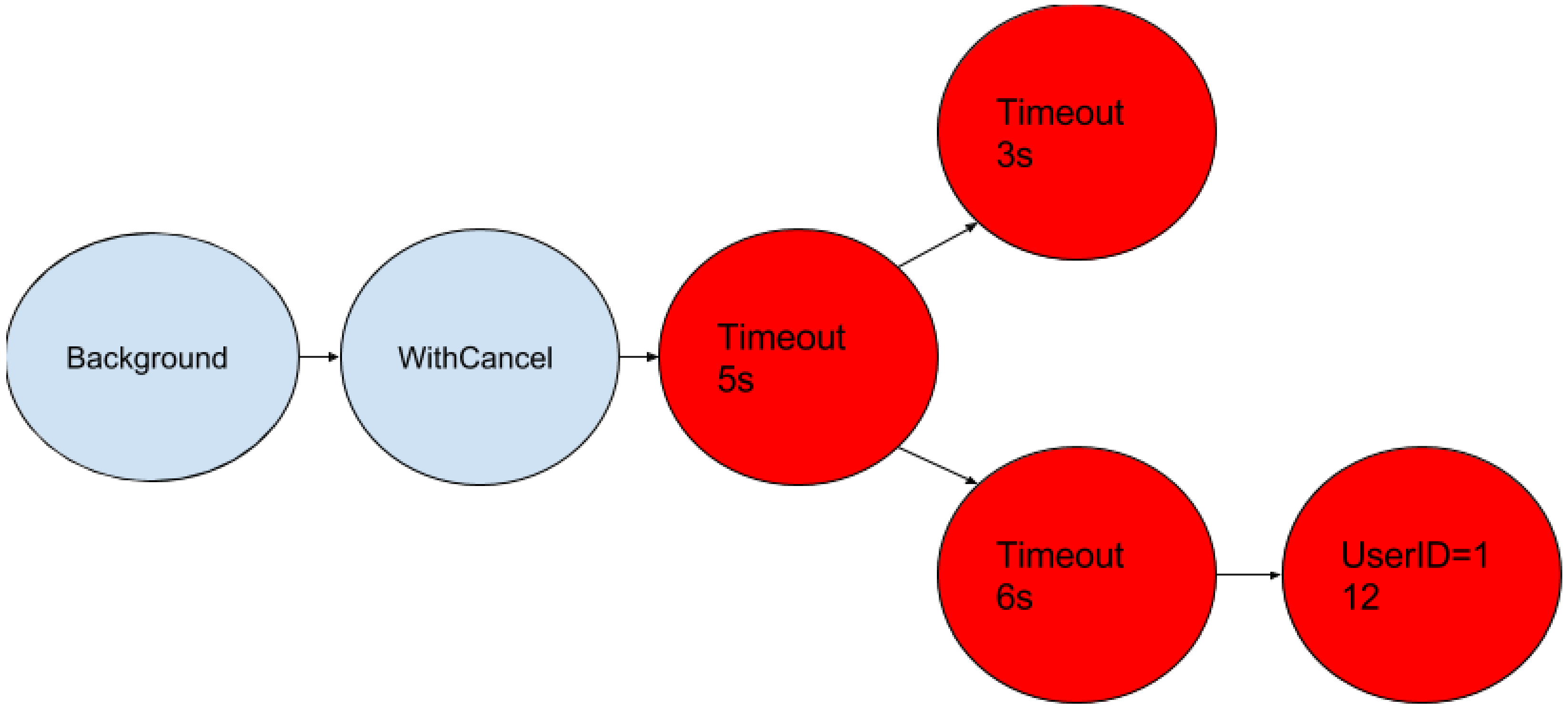
```
package main

func tree() {
    ctx1 := context.Background()
    ctx2, _ := context.WithCancel(ctx1)
    ctx3, _ := context.WithTimeout(ctx2, time.Second*5)
    ctx4, _ := context.WithTimeout(ctx3, time.Second*3)
    ctx5, _ := context.WithTimeout(ctx3, time.Second*6)
    ctx6 := context.WithValue(ctx5, "UserID", 12)
}
```



Context tree at 3 sec



Context tree at 5 sec



context.Context API

- 3 functions on when to end your sub-request
- 1 function on request scoped variables

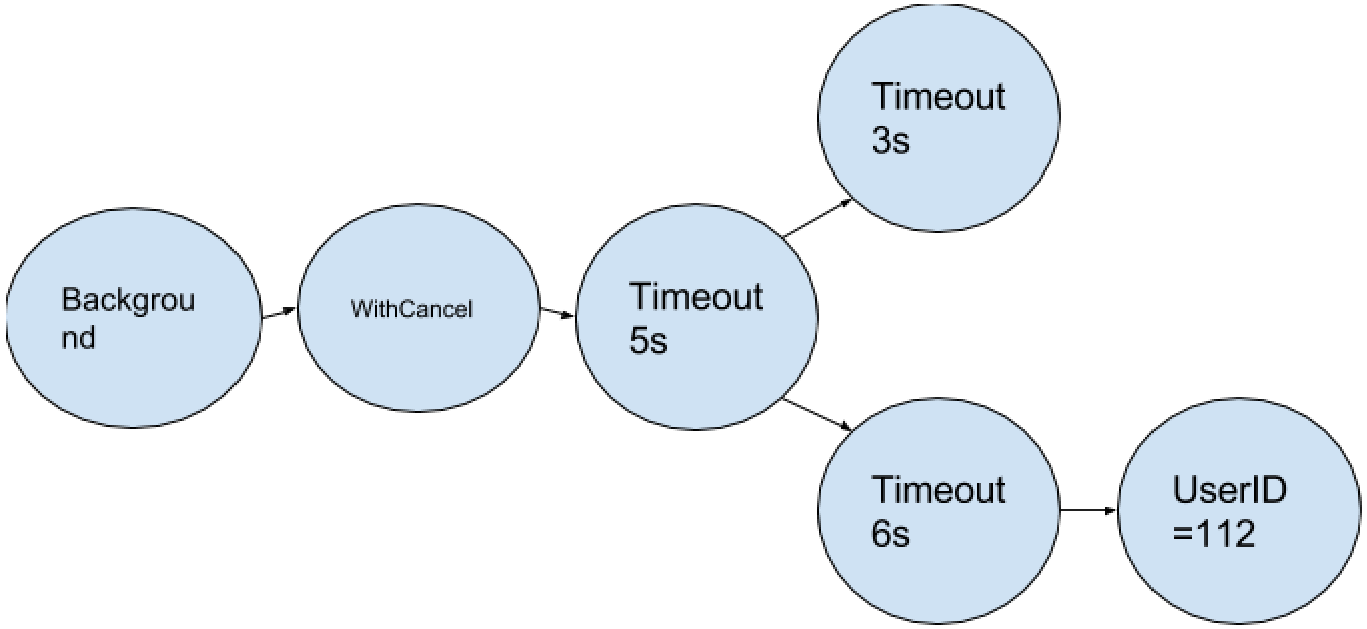
```
type Context interface {

    Deadline() (deadline time.Time, ok bool)
    Done() <-chan struct{}
    Err() error

    Value(key interface{}) interface{}
}
```

When should you use context.Context?

- Every RPC call should have an ability to time out
- This is just reasonable API design
- Not just *timeout*, since you also need to be able to end an RPC call early if the result is no longer needed.
- context.Context is the Go standard solution
- Any function that can block or take a long time to finish should have a context.Context

How to create a context

- Use `context.Background()` at the beginning of an RPC request
- If you don't have a context, and need to call a context function, use `context.TODO()`
- Give sub requests their own sub context if they need other timeouts



How to integrate context.Context

- As the first variable of a function call

```
func (d* Dialer) DialContext(ctx context.Context, network, address string) (Conn, error)
```

- As an optional value on a request struct

```
func (r *Request) WithContext(ctx context.Context) *Request
```

- The variable name should probably be `ctx`

Where to put a context

- Think of a context flowing through your program, like water in a river
- Ideally, context exists only on the call stack
- Do not store the context in a struct
- Only exception is when the struct is a **request** struct (`http.Request`)
- Request structs should end with a request
- Context instances should be unreferenced when the RPC call is finished
- Context dies when the request dies

Context package caveats

- Create pattern of closing contexts
- Especially timeout contexts
- If a context GCs, and isn't canceled, you probably did something wrong

```
ctx, cancel := context.WithTimeout(parentCtx, time.Second)
// Uses time.AfterFunc
// Will not garbage collect before timer expires
defer cancel()
// Good pattern to defer cancel() after creation
```

Request cancellation

When to cancel a context early

- When you don't care about spawned logic
- golang.org/x/sync/errgroup as example
- errgroup uses context to create an RPC cancellation abstraction
- Great deep dive into ideal context usage

Example usage (golang.org/x/sync/errgroup)

```
type Group struct {
    cancel func()

    wg sync.WaitGroup

    errOnce sync.Once
    err     error
}

// Create a group and a context to use with that group
func WithContext(ctx context.Context) (*Group, context.Context) {
    ctx, cancel := context.WithCancel(ctx)
    return &Group{cancel: cancel}, ctx
}
```

- Return a context that **Group** can cancel early

Example usage (golang.org/x/sync/errgroup)

```
func (g *Group) Go(f func() error) {
    g.wg.Add(1)
    go func() {
        defer g.wg.Done()
        if err := f(); err != nil {
            g.errOnce.Do(func() {
                g.err = err
                if g.cancel != nil {
                    g.cancel()
                }
            })
        }
    }()
}
```

- Execute a function concurrently

Example usage (golang.org/x/sync/errgroup)

```
func (g *Group) Go(f func() error) {
    g.wg.Add(1)
    go func() {
        defer g.wg.Done()
        if err := f(); err != nil {
            g.errOnce.Do(func() {
                g.err = err
                if g.cancel != nil {
                    g.cancel()
                }
            })
        }
    }()
}
```

- If that function fails

Example usage (golang.org/x/sync/errgroup)

```
func (g *Group) Go(f func() error) {
    g.wg.Add(1)
    go func() {
        defer g.wg.Done()
        if err := f(); err != nil {
            g.errOnce.Do(func() {
                g.err = err
                if g.cancel != nil {
                    g.cancel()
                }
            })
        }
    }()
}
```

- Cancel the returned context once with `sync.Once`

Example usage (golang.org/x/sync/errgroup)

```go
// Create a group and a context to use with that group
func WithContext(ctx context.Context) (*Group, context.Context) {
    // ...
}

// Go executes f in another goroutine
func (g *Group) Go(f func() error) {
    // ...
}

// Wait for all Go functions to finish
func (g *Group) Wait() error {
    g.wg.Wait()
    if g.cancel != nil {
        g.cancel()
    }
    return g.err
}
```

- Notice cleanup `cancel()` after Wait

Example usage (errgroup)

```go
package main

func DoTwoRequestsAtOnce(ctx context.Context) error {
    eg, egCtx := errgroup.WithContext(ctx)
    var resp1, resp2 *http.Response
    f := func(loc string, respIn **http.Response) func() error {
        return func() error {
            reqCtx, cancel := context.WithTimeout(egCtx, time.Second)
            defer cancel()
            req, _ := http.NewRequest("GET", loc, nil) // TODO: Check this error
            var err error
            *resp, err = http.DefaultClient.Do(req.WithContext(reqCtx))
            if err == nil && (*respIn).StatusCode > 500 {
                return errors.New("unexpected!")
            }
            return err
        }
    }

    eg.Go(f("http://localhost:8080/fast_request", &resp1)) // <--- Run two requests
    eg.Go(f("http://localhost:8080/slow_request", &resp2))

    return eg.Wait() // TODO: Actually do something with resp1 and resp2
}
```

Request scoped values

context.Value, the API duct tape

- Creates `Value` nodes in the context chain

```
package context

func WithValue(parent Context, key, val interface{}) Context {
    // .. some key/val validation redacted
    return &valueCtx{parent, key, val}
}

type valueCtx struct {
    Context
    key, val interface{}
}

func (c *valueCtx) Value(key interface{}) interface{} {
    if c.key == key {
        return c.val
    }
    return c.Context.Value(key)
}
```

Scope your keyspace

- Private type
- Instance of private type
- Get/Set that uses private instance

```
package reqinfo

type privateCtxType string

var (
    reqID = privateCtxType("req-id")
)

func GetRequestID(ctx context.Context) (int, bool) {
    id, exists := ctx.Value(reqID).(int)
    return id, exists
}

func WithRequestID(ctx context.Context, reqid int) context.Context {
    return context.WithValue(ctx, reqID, reqid)
}
```

Context.Value should be immutable

- context.Context is designed immutable
- Keep this with context.Value
- Do not store a value that, if changed, the change is seen by other contexts
- Remember this when you use context.Value

What to put in context.Value

- Everything about a context should be request scoped
- Includes context.Value
- What is a request scoped value?
- Derived from request data and goes away when the request is over

What things are clearly not request scoped

- Objects made outside the request and not changed with the request
- Database connection
- But what if you put the user ID on the connection?
- Global logger
- But what if you put the user ID on the logger?

What's the problem with context.Value?

- Unfortunately, almost everything is derived from the request
- Why bother having function parameters.
- Just accept a context?
- Think about it: what isn't derived from the request?

```
// Who needs other function parameters?
func Add(ctx context.Context) int {
    return ctx.Value("first").(int) + ctx.Value("second").(int)
}
```

Why I dislike context.Value

- Function parameters clearly tell you what a function needs

```
func IsAdminUser(ctx context.Context) bool {
  userID := GetUserID(ctx)
  return authSingleton.IsAdmin(userID)
}
```

What if we changed the function signature?

```
func IsAdminUser(ctx context.Context) bool
func IsAdminUser(ctx context.Context, userID string, authenticator auth.Service) bool
```

- What does this function signature tell us?
- This function can end early
- This function takes a user ID
- This function uses an authenticator on the userID
- What do I need to change to test this function?
- stub out authenticator
- modify the userID
- All of this from the signature

Which function is easier to refactor?

- If it takes just a context, I need to make sure the userID is there wherever I use it
- If it takes what it needs, then I know what to modify

So what is ok to put in context.Value?

- context.Value should inform, not control
- Should never be required input for documented results
- If your function can't behave correctly because of what `context.Value` has, you're obscuring your API too heavily

What things usually don't control

- Request ID
- Often given to each RPC request.
- The logic of the request is almost never gated on what the ID is
- Logging
- The logger itself is not request scoped, so should not sit on the context
- Logging decoration can be request scoped, so can sit on the context
- User ID (if used only for logging)
- Incoming request ID
- The non request scoped logger, can use the context to decorate logs

Things that clearly control

- Database connection
- Controls logic very heavily
- Explicitly call it out as a parameter
- Authentication
- Obviously controls logic
- Very important to how a function works
- Call it out explicitly

Creative debugging with context.Value

- net/http/httptrace

Go 1.7 httptrace and context debug patterns (https://medium.com/@cep21/go-1-7-httptrace-and-context-debug-patterns-608ae887224a)

```
package main

func trace() {
    trace := &httptrace.ClientTrace{
        GotConn: func(connInfo httptrace.GotConnInfo) {
            fmt.Println("Got Conn")
        },
        ConnectStart: func(network, addr string) {
            fmt.Println("Dial start")
        },
        ConnectDone: func(network, addr string, err error) {
            fmt.Println("Dial done")
        },
    }
    req = req.WithContext(httptrace.WithClientTrace(req.Context(), trace))
    c.Do(req)
}
```

How net/http uses httptrace

- If a trace is attached, execute trace callbacks
- Inform, not control

```
package http

func (req *Request) write(w io.Writer, usingProxy bool, extraHeaders Header, waitForContinue func() bool) (er
    // ...
    trace := httptrace.ContextClientTrace(req.Context())
    // ...
    if trace != nil && trace.WroteHeaders != nil {
        trace.WroteHeaders()
    }
}
```

Dodgy dependency injections (github.com/golang/oauth2)

- They use `ctx.Value` to locate dependencies
- Recommend not doing this

```
package main

import "github.com/golang/oauth2"

func oauth() {
    c := &http.Client{Transport: &mockTransport{}}
    ctx := context.WithValue(context.Background(), oauth2.HTTPClient, c)
    conf := &oauth2.Config{ /* ... */ }
    conf.Exchange(ctx, "code")
}
```

Reasons people abuse context.Value

- Middleware abstractions
- Deep callstacks
- Spaghetti designs
- context.Value doesn't make your API cleaner, it makes it more obscured

Summary of context.Value

- Great for debugging information
- Required context.Value parts obscure your API
- Just try not to use it
- Be explicit if possible

Thank you

Jack Lindamood
Software Engineer, Twitch
cep221@gmail.com (mailto:cep221@gmail.com)