Go Context

Sundara Senthil

Content

History

Why Context?

Context

Rules

Context APIs

Sample

Coding

History

➢ The context package originated out of Google and announced on July 2014

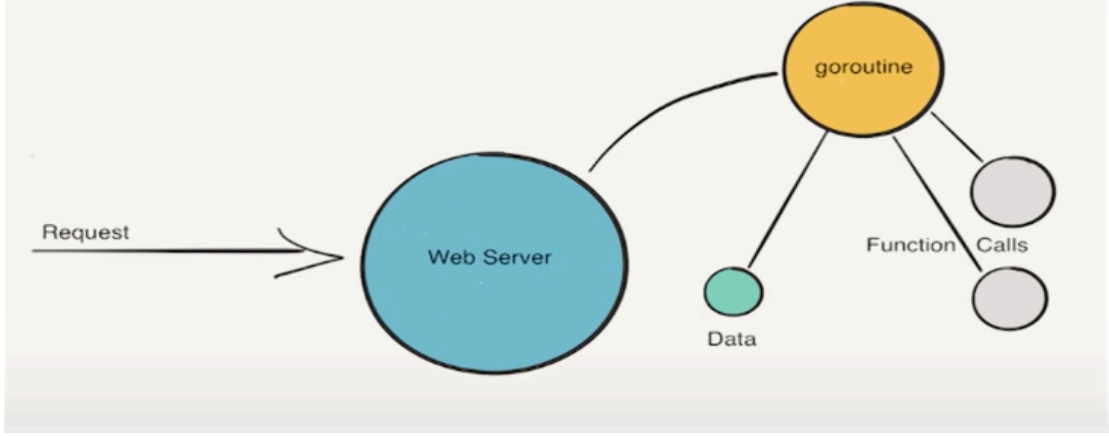
➢Go 1.7 moves the golang.org/x/net/context package into the standard library as context.

References

- https://blog.golang.org/context
- https://blog.golang.org/pipelines
- golang.org/x/net/context

Why Context?

Great introductory blog article, [Go Concurrency Patterns: Context](), by Sameer Ajmani published in 2014.



Problem :

- ❖ In Go servers, each incoming request is handled in its own goroutine.
- ❖ Request handlers often start additional goroutines to access backends such as databases and RPC services.
- ❖ The set of goroutines working on a request typically needs access to request-specific values such as the identity of the end user, authorization tokens, and the request's deadline.
- ❖ When a request is canceled or times out, all the goroutines working on that request should exit quickly so the system can reclaim any resources they are using

Solution: Context

- ❖ A context package that makes it easy by passing request-scoped values, cancelation signals, and deadlines across API boundaries
  - ➢Request-scoped values
  - ➢Cancelation Signals
  - ➢Deadlines across API boundaries

to all the goroutines involved in handling a request

Context

// A Context carries a deadline, cancelation signal, and request-scoped values across API boundaries.

```
type Context interface {

    // Done returns a channel that is closed when this Context is canceled  or times out.
    Done() <-chan struct{}

    // Err indicates why this context was canceled, after the Done channel is closed.
    Err() error

    // Deadline returns the time when this Context will be canceled, if any.
    Deadline() (deadline time.Time, ok bool)

    // Value returns the value associated with key or nil if none.
    Value(key interface{}) interface{}
}
```

Cont...

- The Done method returns a channel that acts as a cancelation signal to functions running on behalf of the Context: when the channel is closed, the functions should abandon their work and return.
- The Err method returns an error indicating why the Context was canceled. (Pipelines and Cancelation article discusses the Done channel idiom in more detail.)
- The Deadline method allows functions to determine whether they should start work at all; if too little time is left, it may not be worthwhile. Code may also use a deadline to set timeouts for I/O operations.
- Value allows a Context to carry request-scoped data. That data must be safe for simultaneous use by multiple goroutines.

Others

- A Context does *not* have a Cancel method : the function receiving a cancelation signal is usually not the one that sends the signal. So for the same reason the Done channel is receive-only.
    - In particular, when a parent operation starts goroutines for sub-operations, those sub-operations should not be able to cancel the parent. Instead, the WithCancel function (described below) provides a way to cancel a new Context value.
- Code can pass a single Context to any number of goroutines and cancel that Context to signal all of them. Hence Context is safe for simultaneous use by multiple goroutines.

Rules

✓ The Context should be the first parameter, typically named ctx:

```
func DoSomething(ctx context.Context, arg Arg) error {
        // ... use ctx ...
}
```

✓ Do not store Contexts inside a struct type; instead, pass a Context explicitly to each function that needs it.

✓ Do not pass a nil Context, even if a function permits it. Pass context. (TODO if you are unsure about which Context to use.)

✓ Use context Values only for request-scoped data that transits processes and APIs, not for passing optional parameters to functions.

✓ use context to store *values*, like strings and data structs; avoid using it to store *references*, like pointers or handles.

✓ The same Context may be passed to functions running in different goroutines

✓ Contexts are safe for simultaneous use by multiple goroutines.

Sample Done

```
func someHandler() {
    ctx, cancel := context.WithCancel(context.Background())
    go doStuff(ctx)
    // ...some work happens...
    if someCondition {
        cancel()
    }
}

func doStuff(ctx context.Context) {
    // ...Doing some work
    select {
    case <-ctx.Done():
        fmt.Println("Stop work!")
        return
    }
}
```

Cont... Err and Deadline

```
func someHandler() {
        ctx, cancel := context.WithDeadline(context.Background(), time.Now().Add(5 * time.Second))
        go doStuff(ctx)
        // if deadline expires before work completes Done() channel is trigger
        cancel()
}

func doStuff(ctx context.Context) {
        if deadline, ok := ctx.Deadline(); ok {
                if time.Now().After(deadline) {
                        return ctx.Err()
                }
        }
        // ... do actual work...
}
```

Cont… Value

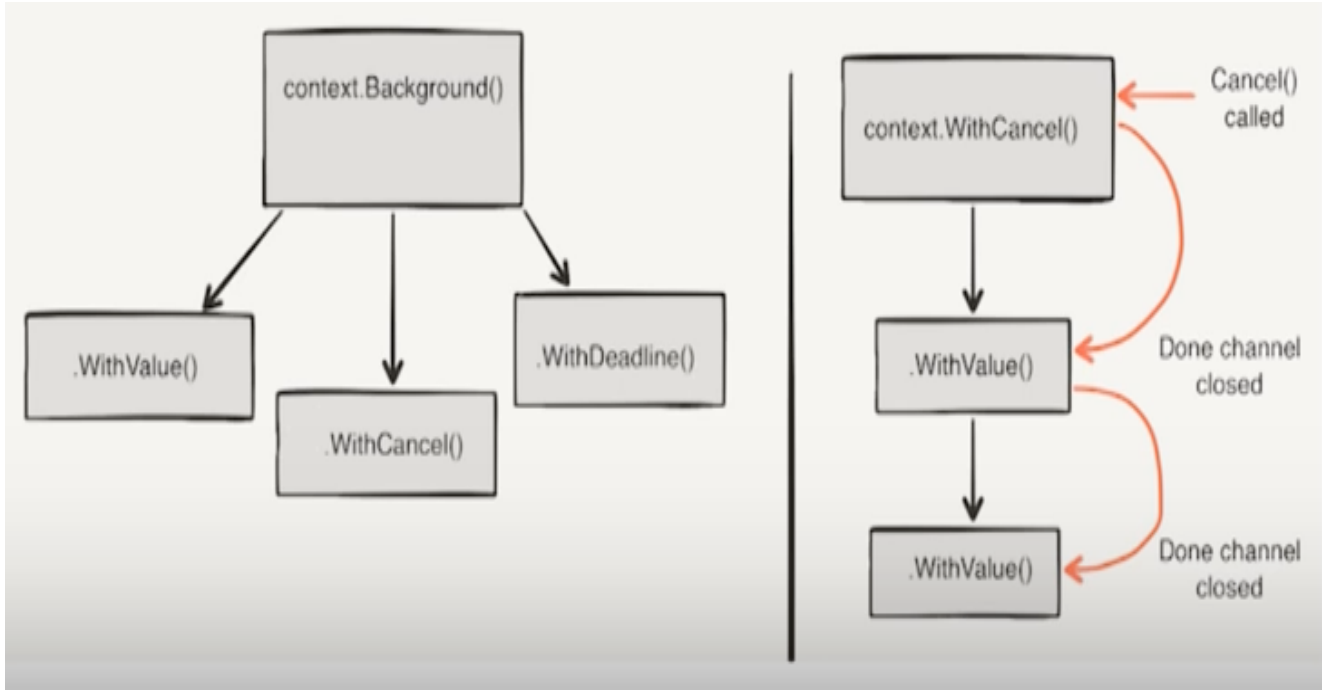
- Provides a way to load request scoped data

```
var string value = “SomeValue”
ctx := context.WithValue(context.Backgroud(),key,value)
val := ctx.Value(key).(string)
```

A few notes on Value

- ❑ Context Value is completely type. Unsafe.
- ❑ So can’t be checked at compile time
- ❑ Its map[interface{}]interface{}
- ❑ Example: extracted data from headers, cookies, userID tied with auth information, etc

Derived Context



- Derived context, drives new context value from existing context
- Context derivation from like a Tree
- When a context is cancelled, all its derived contexts will also be cancelled
- Provides a mechanism to manage the lifecycle of dependent functions within a request scoped operation

Cont...

- func Background() Context
  - Typically the top level context for incoming requests

- func TODO() Context
  - Placeholder. When its unclear, what context to use or pass. TODO never send nil context.

- func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
  - Return a copy of parent context with Done Channel
  - Done Channel is closed when the context function is called or the parent context Done Channel is closed

- func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
  - Take Time param, with deadline adjusted to be no later than the time parameter
  - Done Channel is closed, when deadline expires, when cancelled function is called or the parent context Done Channel is closed(Which ever comes first)

- func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
  - Return context with deadline set to current time.
  - Code should call Cancel as soon as operation running this context complete

- func WithValue(parent Context, key interface{}, val interface{}) Context
  - return a copy of the parent in which the value for the specified key is set to val

Coding

Thank You