Programming in Go

Matt Holiday
Christmas 2020

Context

Cancellation and timeouts

The `Context` package offers a common method to cancel requests

- explicit cancellation
- implicit cancellation based on a timeout or deadline

A context may also carry request-specific values, such as a trace ID

Many network or database requests, for example, take a context for cancellation

Cancellation and timeouts

A context offers two controls:

- a channel that closes when the cancellation occurs
- an error that's readable once the channel closes

The error value tells you whether the request was cancelled or timed out

We often use the channel from `Done()` in a `select` block

Cancellation and timeouts

Contexts form an **immutable** tree structure
(goroutine-safe; changes to a context do not affect its ancestors)

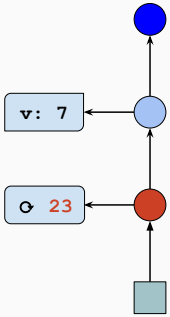
Cancellation or timeout applies to the current context and its **subtree**

Ditto for a value

A subtree may be created with a shorter timeout (but not longer)

Context as a tree structure

It’s a tree of **immutable** nodes which can be extended




```
ctx := context.Background()

ctx = context.WithValue(ctx, "v", 7)

ctx, canc := context.WithTimeout(ctx, t)

req, _ := http.NewRequest(method, url, nil)
req = req.WithContext(ctx)
resp, err := http.DefaultClient.Do(req)
```

Context example

The `Context` value should always be the first parameter

```
// First runs a set of queries and returns the result from the
// the first to respond, canceling the others.
func First(ctx context.Context, urls []string) (*Result, error) {
    c := make(chanResult, len(urls))        // buffered to avoid orphans
    ctx, cancel := context.WithCancel(ctx)

    defer cancel()  // cancel the other queries when we're done

    search := func(url string) {
        c <- runQuery(ctx, url)
    }

    . . .
```

Context example

Don't call `Err()` until the channel from `Done()` closes

```
    . . .

    for _, url := range urls {
        go search(url)
    }

    select {
    case r := <- c:
        return &r, nil
    case <-ctx.Done():
        return nil, ctx.Err()
    }
}
```

Values

Context values should be data specific to a request, such as:

- a trace ID or start time (for latency calculation)
- security or authorization data

Avoid using the context to carry “optional” parameters

Use a package-specific, private context key type (not string) to avoid collisions

Value example

```
type contextKey int

const TraceKey contextKey = 1

// AddTrace is HTTP middleware to insert a trace ID into the request.
func AddTrace(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        ctx := r.Context()

        if traceID := r.Header.Get("X-Cloud-Trace-Context"); traceID != "" {
            ctx = context.WithValue(ctx, TraceKey, traceID)
        }

        next.ServeHTTP(w, r.WithContext(ctx))
    }
}
```

Value example

```
type contextKey int

const TraceKey contextKey = 1

// ContextLog makes a log with the trace ID as a prefix.
func ContextLog(ctx context.Context, f string, args ...interface{}) {
    // reflection -- to be discussed

    traceID, ok := ctx.Value(TraceKey).(string)

    if ok && traceID != "" {
        f = traceID + ": " + f
    }

    log.Printf(f, args...)
}
```

Parallel get with timeout

```
func main() {
    results := make(chan result)   // channel for results
    list := []string{"https://amazon.com",  . . .}
    ctx, cancel := context.WithTimeout(context.Background, 3*time.Second)
    defer cancel()
    for _, url := range list {
        go get(ctx, url, results) // start a CSP process
    }
    for range list {              // read from the channel
        r := <-results
        if r.err != nil {
            log.Printf("%-20s %s\n", r.url, r.latency)
        } else {
            log.Printf("%-20s %s\n", r.url, r.err)
        }
    }
}
```

Parallel get with timeout

```
func get(ctx context.Context, url string, ch chan<- result) {
    start := time.Now()
    req, err := http.NewRequestWithContext(ctx, http.MethodGet, url, nil)

    if err != nil {
        ch <- result{url, err, 0}      // error response
        return
    }

    if resp, err := http.DefaultClient.Do(req); err != nil {
        ch <- result{url, err, 0}      // error response
    } else {
        t := time.Since(start).Round(time.Millisecond)
        ch <- result{url, nil, t}      // normal response
        resp.Body.Close()
    }
}
```

Server with variable delay

```
package main

import ("flag"; "fmt"; "log"; "net/http"; "time")

var delay int

func handler(w http.ResponseWriter, r *http.Request) {
    time.Sleep(time.Duration(delay) * time.Second)
    fmt.Fprintf(w, "hello")
}

func main() {
    flag.IntVar(&delay, "delay", 60, "delay all responses")
    flag.Parse()
    http.HandleFunc("/", handler)
    log.Fatal(http.ListenAndServe(":8081", nil))
}
```