CR1

// Sample program to show how to store and retrieve

// values from a context.

package main

import (

"context"

"fmt"

)

// TraceID is represents the trace id.

type TraceID string

// TraceIDKey is the type of value to use for the key. The key is

// type specific and only values of the same type will match.

type TraceIDKey int

func main() {

// Create a traceID for this request.

traceID := TraceID("f47ac10b-58cc-0372-8567-0e02b2c3d479")

// Declare a key with the value of zero of type userKey.

const traceIDKey TraceIDKey = 0

// Store the traceID value inside the context with a value of

// zero for the key type.

ctx := context.WithValue(context.Background(), traceIDKey, traceID)

// Retrieve that traceID value from the Context value bag.

if uuid, ok := ctx.Value(traceIDKey).(TraceID); ok {

fmt.Println("TraceID:", uuid)

}

// Retrieve that traceID value from the Context value bag not

// using the proper key type.

if \_, ok := ctx.Value(0).(TraceID); !ok {

fmt.Println("TraceID Not Found")

}

}

CR2

// Sample program to show how to use the WithCancel function.

package main

import (

"context"

"fmt"

"time"

)

func main() {

// Create a context that is cancellable only manually.

// The cancel function must be called regardless of the outcome.

ctx, cancel := context.WithCancel(context.Background())

defer cancel()

// Ask the goroutine to do some work for us.

go func() {

// Wait for the work to finish. If it takes too long move on.

select {

case <-time.After(100 \* time.Millisecond):

fmt.Println("moving on")

case <-ctx.Done():

fmt.Println("work complete")

}

}()

// Simulate work.

time.Sleep(50 \* time.Millisecond)

// Report the work is done.

cancel()

// Just hold the program to see the output.

time.Sleep(time.Second)

}

CR3

// Sample program to show how to use the WithDeadline function.

package main

import (

"context"

"fmt"

"time"

)

type data struct {

UserID string

}

func main() {

// Set a deadline.

deadline := time.Now().Add(150 \* time.Millisecond)

// Create a context that is both manually cancellable and will signal

// a cancel at the specified date/time.

ctx, cancel := context.WithDeadline(context.Background(), deadline)

defer cancel()

// Create a channel to received a signal that work is done.

ch := make(chan data, 1)

// Ask the goroutine to do some work for us.

go func() {

// Simulate work.

time.Sleep(200 \* time.Millisecond)

// Report the work is done.

ch <- data{"123"}

}()

// Wait for the work to finish. If it takes too long move on.

select {

case d := <-ch:

fmt.Println("work complete", d)

case <-ctx.Done():

fmt.Println("work cancelled")

}

}

CR4

// Sample program to show how to use the WithTimeout function

// of the Context package.

package main

import (

"context"

"fmt"

"time"

)

type data struct {

UserID string

}

func main() {

// Set a duration.

duration := 150 \* time.Millisecond

// Create a context that is both manually cancellable and will signal

// a cancel at the specified duration.

ctx, cancel := context.WithTimeout(context.Background(), duration)

defer cancel()

// Create a channel to received a signal that work is done.

ch := make(chan data, 1)

// Ask the goroutine to do some work for us.

go func() {

// Simulate work.

time.Sleep(50 \* time.Millisecond)

// Report the work is done.

ch <- data{"123"}

}()

// Wait for the work to finish. If it takes too long move on.

select {

case d := <-ch:

fmt.Println("work complete", d)

case <-ctx.Done():

fmt.Println("work cancelled")

}

}

CR5

// Sample program that implements a web request with a context that is

// used to timeout the request if it takes too long.

package main

import (

"context"

"io"

"log"

"net/http"

"os"

"time"

)

func main() {

// Create a new request.

req, err := http.NewRequest("GET", "https://www.ardanlabs.com/blog/post/index.xml", nil)

if err != nil {

log.Println("ERROR:", err)

return

}

// Create a context with a timeout of 50 milliseconds.

ctx, cancel := context.WithTimeout(req.Context(), 50\*time.Millisecond)

defer cancel()

// Bind the new context into the request.

req = req.WithContext(ctx)

// Make the web call and return any error. Do will handle the

// context level timeout.

resp, err := http.DefaultClient.Do(req)

if err != nil {

log.Println("ERROR:", err)

return

}

// Close the response body on the return.

defer resp.Body.Close()

// Write the response to stdout.

io.Copy(os.Stdout, resp.Body)

}

CR6

// Sample program to show when a Context is canceled, all Contexts

// derived from it are also canceled.

package main

import (

"context"

"fmt"

"sync"

)

// Need a key type.

type myKey int

// Need a key value.

const key myKey = 0

func main() {

// Create a Context that can be cancelled.

ctx, cancel := context.WithCancel(context.Background())

defer cancel()

// Use the Waitgroup for orchestration.

var wg sync.WaitGroup

wg.Add(10)

// Create ten goroutines that will derive a Context from

// the one created above.

for i := 0; i < 10; i++ {

go func(id int) {

defer wg.Done()

// Derive a new Context for this goroutine from the Context

// owned by the main function.

ctx := context.WithValue(ctx, key, id)

// Wait until the Context is cancelled.

<-ctx.Done()

fmt.Println("Cancelled:", id)

}(i)

}

// Cancel the Context and any derived Context's as well.

cancel()

wg.Wait()

}