Context cancellation: Stop wasting resources on aborted requests

Published Thu 23 Mar, 2023

GO/GOLANG CONTEXT

If you open your browser and load a web page, but press the “stop loading” or “X” button before it is fully loaded, the browser will abort any open requests.

But what happens if such a http request already reached your Go application and it is being served by a handler?

Well.. if you’re not explicitly supporting **cancellation**, the handler will happily continue executing until it is time to write a response to the client. Once enough data is written, the server will attempt to write that data to the underlying connection, which will fail with an error like this:

```
write tcp 127.0.0.1:8080->127.0.0.1:60710: write: broken pipe
```

Any work (long running calculations, database queries, http requests to other systems, etc.) done before that response is written is likely wasted, since the client will never see the response.

This browser/http example is one situation in which it is useful to allow “cancellation” of an operation, but you will run into it almost any time you have two systems communicating.

The Context package

To support “cancellation”, the Go standard library provides the **context package**.

> *Package context defines the Context type, which carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes.*

The `Context` type is an *interface* defining a handful of methods:

```
package context

type Context interface {
    Deadline() (deadline time.Time, ok bool)
    Done() <-chan struct{}
    Err() error
    Value(key any) any
}
```

In this post we will focus on the cancellation functionality provided by this interface.

So how does that work and what is meant by “cancellation signals”? Let’s find out by making an existing function cancellable.

Making a function cancellable

Let’s look at an example and make a cancellable function.

The following program defines and executes a function called `work`, which represents a longer running task:

- It prints `"start"`.
- Then, it prints a number of steps with a 1 second pause in between.
- And finally it prints `"done"`.

```
package main

import (
    "fmt"
    "time"
)
```

We want to make the `work` function *cancellable* by allowing us to abort it any point during the for-loop.

First we add the context as a parameter. By convention any function or method that accepts a context should accept it as the first parameter. The code now looks like this:

```
package main

import (
    "context"
    "fmt"
    "time"
)

// work now accepts a context as a first parameter.
func work(ctx context.Context, nr int) {
    fmt.Println("start")

    ticker := time.NewTicker(1 * time.Second)
    for i := 0; i < nr; i++ {
        <-ticker.C
        fmt.Printf("step: %d\n", i)
    }

    ticker.Stop()
    fmt.Println("done")
```

Apart from passing a context, this doesn't do anything.

As a next step we will send a "cancellation signal" by using `context.WithCancel` and calling the returned `cancel` function.

```
// ...

func main() {
    // create a new context derived from context.Background(),
    // this new context can be canceled by calling cancel().
    ctx, cancel := context.WithCancel(context.Background())

    // cancel the context concurrently.
    go func() {
        fmt.Println("cancelling...")
        cancel()
        // NOTE: it is not recommended to run goroutines like this
        // without `main` waiting for it to finish. For this demo it
        // won't cause an issue because we know that `work` will take
        // ~5 seconds to run. But don't do this in real code.
    }()

    // run work with the new context.
    work(ctx, 5)
}
```

If you run the program, you will get something like the following output. Note that `cancelling...` can potentially be in a different position, because the goroutine is not always scheduled to run at the same time.

```
start
cancelling...
step: 0
step: 1
step: 2
step: 3
```

As you can see, even though we call `cancel()`, the `work` method itself is not canceled yet. This is because it is not yet listening to the "cancellation signal".

As you might have seen earlier, the `context.Context` interface has a `Done()` **method**:

> *Done returns a channel that's closed when work done on behalf of this context should be canceled. Done may return nil if this context can never be canceled. Successive calls to Done return the same value. The close of the Done channel may happen asynchronously, after the cancel function returns.*

If we wait for the channel returned by `Done()` to be closed, we get our "cancellation signal".

The channel returned by `Done()` is only used to *wait for a close*, you will never receive a value on it.

This means you don't need to differentiate between the "received a value because it was send succesfully" and "received a zero value because the channel was closed" cases.

You can just use the `<-` receive operator directly (instead of `val, ok <-` and then checking `ok`).

Let's use this in the `work` function using a `select` statement:

```
// ...

func work(ctx context.Context, nr int) {
    fmt.Println("start")

    ticker := time.NewTicker(1 * time.Second)
    for i := 0; i < nr; i++ {
        select {
        case <-ctx.Done():
            // channel was closed (Received a "cancellation signal")
```

If you run the program now, you get the following:

```
start
cancelling...
canceled!
```

As you can see the function was succesfully canceled.

Cancelling after some time

Instead of using a call to `cancel()` to cancel a context, you can also use timeouts to automatically cancel a context after a period of time. This can be done using the `context.WithTimeout` function, it's like `context.WithCancel` but also takes a timeout value.

The example below modifies `main` to cancel after `2.5` seconds.

```
// ...

func main() {
    // create a new context derived from context.Background(),
    // this new context will time out after 2.5 seconds.
    ctx, cancel := context.WithTimeout(context.Background(), 2500*time.Millisecc
    // It's good practice to defer the cancel, so any underlying resources
    // can be released if `work` would return before the timeout finishes.
    defer cancel()

    // run work with the new context
```

Running this program you should get the following output:

```
start
step: 0
step: 1
canceled!
```

As you can see the context is canceled before we output `step: 2` at around the third second.

Checking for cancellation

In both the earlier examples we did not care if `work` actually finished succesfully. In real programs, you often want to know if a function or method completed succesfully or failed due to an error or cancellation.

Luckily, `Context` can help with checking for timeouts and cancellations by providing you with specific errors.

`Context` has an `Err()` method that returns an error once the channel returned by `Done()` has been closed. The error returned is either:

- `context.Canceled` : when the context is canceled.
- `context.DeadlineExceeded` : when the context has timed out.

Generally you want to return the error provided by `Err()` to the caller of your function or method. So that the code that created, configured and passed in the `Context` can also check if that actually happened.

Let's modify our timeout example to handle the `context.DeadlineExceeded` error in a custom way.

```
package main
```

In the above example, we check that `work` has timed out by identifying the error it returned using `errors.Is(...)`. Similarly, checking for

`context.Canceled` would allow you to check if it was canceled.

Note this symmetry: Since the `main` function was responsible for creating the context and configuring the timeout, it is also responsible for handling the consequences of that timeout.

Getting http handlers cancelled

Let's go back to the situation described in the intro: cancelling http handlers.

Using context inside http handlers is pretty straightforward, we don't need to explicitly create one, nor do we need to set up cancellation manually.

Each incoming http request will automatically have a fresh context assigned. This context can be retrieved using the `Context()` **method** on the `http.Request`.

> *For incoming server requests, the context is canceled when the client's connection closes, the request is canceled (with HTTP/2), or when the ServeHTTP method returns.*

As the documentation says, this context will automatically be cancelled:

- Once the connection is closed.
- If the request is cancelled when using HTTP/2.
- Or the handler has finished (that is, the `ServeHTTP` method returns).

Let's try this out by calling `work` from a handler, modify `main` like this:

```
package main

import (
    "context"
    "fmt"
    "log"
    "net/http"
    "time"
)
```

If run this program and visit `http://localhost:8080` in your browser you can see that the page takes about 5 seconds to load.

If you reload and abort the request by pressing the “X” button in your browser, you will something like this:

```
start
step: 0
step: 1
2023/03/23 11:17:33 work error: context canceled
```

Note: You might actually see two requests, some browsers will send a request for a `/favicon.ico` as well.

Cancelling outgoing http requests

Next to receiving contexts and handling cancellations, you might also want to cancel functions and methods that you call.

A lot of packages support passing in a `context.Context`, below is an example from the `net/http` package.

To create a http request that can be cancelled you should use `http.NewRequestWithContext`, the resulting request can then be send by a http client.

As stated in the **documentation**, the context you provide applies during the entire request-response cycle:

> *For an outgoing client request, the context controls the entire lifetime of a request and its response: obtaining a connection, sending the request, and reading the response headers and body.*

So let's try to cancel an outgoing client request.

```
package main

import (
    "context"
    "log"
    "net/http"
    "time"
)

func main() {
    // create a new http client
    client := &http.Client{}

    // create a context that times out after 5 milliseconds
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Millisecond)
    defer cancel()

    // create a request using this context.
    req, err := http.NewRequestWithContext(ctx, http.MethodGet, "https://example
    if err != nil {
        log.Fatal(err)
    }

    // actually send the request
    _, err = client.Do(req)
    if err != nil {
        log.Fatal(err)
    }
}
```

Unless the request somehow finishes in 5 milliseconds, running this program will result in a failure due to the cancellation of the context:

```
2023/03/23 12:10:55 Get "https://example.com": context deadline exceeded
```

Closing

Hopefully this post has given you a decent understanding how context and cancellation can make your applications use less resources and make it more responsive. If you have any questions or comments feel free to reach out to me.

If you want to read more posts like this, sign up for my newsletter below and receive the latest posts in your inbox.

Hello! I'm the Willem behind willem.dev

I created this blog to help newcomers to Go and/or web development. I hope it brings you some value :)

Thanks for reading!