Context-aware Reader

You can find all the code here

This chapter demonstrates how to test-drive a context aware `io.Reader` as written by Mat Ryer and David Hernandez in The Pace Dev Blog.

Context aware reader?

First of all, a quick primer on `io.Reader`.

If you've read other chapters in this book you will have ran into `io.Reader` when we've opened files, encoded JSON and various other common tasks. It's a simple abstraction over reading data from *something*

```
type Reader interface {
	Read(p []byte) (n int, err error)
}
```

By using `io.Reader` you can gain a lot of re-use from the standard library, it's a very commonly used abstraction (along with its counterpart `io.Writer`)

Context aware?

In a previous chapter we discussed how we can use `context` to provide cancellation. This is especially useful if you're performing tasks which may be computationally expensive and you want to be able to stop them.

When you're using an `io.Reader` you have no guarantees over speed, it could take 1 nanosecond or hundreds of hours. You might find it useful to be able to cancel these kind of tasks in your own application and that's what Mat and David wrote about.

They combined two simple abstractions (`context.Context` and `io.Reader`) to solve this problem.

Let's try and TDD some functionality so that we can wrap an `io.Reader` so it can be cancelled.

Testing this poses an interesting challenge. Normally when using an `io.Reader` you're usually supplying it to some other function and you don't really concern yourself with the details; such as `json.NewDecoder` or `ioutil.ReadAll`.

What we want to demonstrate is something like

> Given an `io.Reader` with "ABCDEF", when I send a cancel signal half-way through I when I try to continue to read I get nothing else so all I get is "ABC"

Let's look at the interface again.

```go
type Reader interface {
        Read(p []byte) (n int, err error)
}
```

The `Reader`'s `Read` method will read the contents it has into a `[]byte` that we supply.

So rather than reading everything, we could:

- Supply a fixed-size byte array that doesnt fit all the contents
- Send a cancel signal
- Try and read again and this should return an error with 0 bytes read

For now, let's just write a "happy path" test where there is no cancellation, just so we can get familiar with the problem without having to write any production code yet.

```go
func TestContextAwareReader(t *testing.T) {
        t.Run("lets just see how a normal reader works", func(t *testing.T) {
                rdr := strings.NewReader("123456")
                got := make([]byte, 3)
                _, err := rdr.Read(got)

                if err != nil {
                        t.Fatal(err)
                }

                assertBufferHas(t, got, "123")

                _, err = rdr.Read(got)
```

```go

		if err != nil {
			t.Fatal(err)
		}

		assertBufferHas(t, got, "456")
	})
}

func assertBufferHas(t testing.TB, buf []byte, want string) {
	t.Helper()
	got := string(buf)
	if got != want {
		t.Errorf("got %q, want %q", got, want)
	}
}
```

- Make an `io.Reader` from a string with some data
- A byte array to read into which is smaller than the contents of the reader
- Call read, check the contents, repeat.

From this we can imagine sending some kind of cancel signal before the second read to change behaviour.

Now we've seen how it works we'll TDD the rest of the functionality.

Write the test first

We want to be able to compose an `io.Reader` with a `context.Context`.

With TDD it's best to start with imagining your desired API and write a test for it.

From there let the compiler and failing test output can guide us to a solution

```go
t.Run("behaves like a normal reader", func(t *testing.T) {
	rdr := NewCancellableReader(strings.NewReader("123456"))
	got := make([]byte, 3)
	_, err := rdr.Read(got)
	if err != nil {
		t.Fatal(err)
	}

	assertBufferHas(t, got, "123")

	_, err = rdr.Read(got)

	if err != nil {
		t.Fatal(err)
	}

	assertBufferHas(t, got, "456")
})
```

Try to run the test

```
./cancel_readers_test.go:12:10: undefined: NewCancellableReader
```

Write the minimal amount of code for the test to run and check the failing test output

We'll need to define this function and it should return an `io.Reader`

```go
func NewCancellableReader(rdr io.Reader) io.Reader {
	return nil
}
```

If you try and run it

```
=== RUN   TestCancelReaders
=== RUN   TestCancelReaders/behaves_like_a_normal_reader
```

```
panic: runtime error: invalid memory address or nil pointer dereference [recovered]
        panic: runtime error: invalid memory address or nil pointer dereference

[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x10f8fb5]
```

As expected

Write enough code to make it pass

For now, we'll just return the `io.Reader` we pass in

```go
func NewCancellableReader(rdr io.Reader) io.Reader {
	return rdr
}
```

The test should now pass.

I know, I know, this seems silly and pedantic but before charging in to the fancy work it is important that we have *some* verification that we haven't broken the "normal" behaviour of an `io.Reader` and this test will give us confidence as we move forward.

Write the test first

Next we need to try and cancel.

```go
t.Run("stops reading when cancelled", func(t *testing.T) {
	ctx, cancel := context.WithCancel(context.Background())
	rdr := NewCancellableReader(ctx, strings.NewReader("123456"))
	got := make([]byte, 3)
	_, err := rdr.Read(got)

	if err != nil {
		t.Fatal(err)
	}

	assertBufferHas(t, got, "123")
	cancel()


	n, err := rdr.Read(got)

	if err == nil {
		t.Error("expected an error after cancellation but didnt get one")
	}

	if n > 0 {
		t.Errorf("expected 0 bytes to be read after cancellation but %d were
	}
})
```

We can more or less copy the first test but now we're:

- Creating a `context.Context` with cancellation so we can `cancel` after the first read
- For our code to work we'll need to pass `ctx` to our function
- We then assert that post- `cancel` nothing was read

Try to run the test

```
./cancel_readers_test.go:33:30: too many arguments in call to NewCancellableReader
	have (context.Context, *strings.Reader)
	want (io.Reader)
```

Write the minimal amount of code for the test to run and check the failing test output

The compiler is telling us what to do; update our signature to accept a context

```go
func NewCancellableReader(ctx context.Context, rdr io.Reader) io.Reader {
	return rdr
}
```

(You'll need to update the first test to pass in `context.Background` too)

You should now see a very clear failing test output

```
=== RUN   TestCancelReaders
=== RUN   TestCancelReaders/stops_reading_when_cancelled
--- FAIL: TestCancelReaders (0.00s)
    --- FAIL: TestCancelReaders/stops_reading_when_cancelled (0.00s)
        cancel_readers_test.go:48: expected an error but didnt get one
        cancel_readers_test.go:52: expected 0 bytes to be read after cancellation but
```

Write enough code to make it pass

At this point, it's copy and paste from the original post by Mat and David but we'll still take it slowly and iteratively.

We know we need to have a type that encapsulates the `io.Reader` that we read from and the `context.Context` so let's create that and try and return it from our function instead of the original `io.Reader`

```go
func NewCancellableReader(ctx context.Context, rdr io.Reader) io.Reader {
	return &readerCtx{
		ctx:      ctx,
		delegate: rdr,
	}
}

type readerCtx struct {
	ctx      context.Context
	delegate io.Reader
}
```

As I have stressed many times in this book, go slowly and let the compiler help you

```
./cancel_readers_test.go:60:3: cannot use &readerCtx literal (type *readerCtx) as typ
*readerCtx does not implement io.Reader (missing Read method)
```

The abstraction feels right, but it doesn't implement the interface we need (`io.Reader`) so let's add the method.

```go
func (r *readerCtx) Read(p []byte) (n int, err error) {
	panic("implement me")
}
```

Run the tests and they should *compile* but panic. This is still progress.

Let's make the first test pass by just *delegating* the call to our underlying `io.Reader`

```go
func (r readerCtx) Read(p []byte) (n int, err error) {
	return r.delegate.Read(p)
}
```

At this point we have our happy path test passing again and it feels like we have our stuff abstracted nicely

To make our second test pass we need to check the `context.Context` to see if it has been cancelled.

```go
func (r readerCtx) Read(p []byte) (n int, err error) {
	if err := r.ctx.Err(); err != nil {
		return 0, err
	}
	return r.delegate.Read(p)
}
```

All tests should now pass. You'll notice how we return the error from the `context.Context`. This allows callers of the code to inspect the various reasons cancellation has occurred and this is covered more in the original post.

Wrapping up

- Small interfaces are good and are easily composed
- When you're trying to augment one thing (e.g `io.Reader`) with another you usually want to reach for the delegation pattern

> In software engineering, the delegation pattern is an object-oriented design pattern that allows object composition to achieve the same code reuse as inheritance.

- An easy way to start this kind of work is to wrap your delegate and write a test that asserts it behaves how the delegate normally does before you start composing other parts to change behaviour. This will help you to keep things working correctly as you code toward your goal