よくわかるcontextの使い方

Goの標準パッケージにはcontextパッケージというものが存在します。 このパッケージは、net/httpやdatabase/sqlのような現実の事象と対応している何かが存在するようなパッケージではないため、初学者にとっては使い道がわからない、となってしまいがちです。 しかしcontextパッケージは、複数のゴールーチンを跨いだ処理を実装する際には非常に強力な力を発揮する、とても便利なパッケージなのです。 この本では、「contextとは何か？」というところから「どのように使えばいいのかわかる」ところまでたどり着けるように、Goのcontextまわりのことを解説しました。

はじめに

この本について

Goの`context`パッケージって、使い所がとてもわかりにくいですよね。

例えば`net/http`なら、httpサーバーや、http通信をする際に必要になるツールをまとめているんだな、とわかります。
また`database/sql`なら、DB接続→クエリを投げてデータを取得するためのツールが入っているんだな、と一目瞭然です。

ですが`context`と聞いても、「これは一体なんだ？」となる方がきっと大半なのではないでしょうか。

私が初めてcontextと出会ったのは、OpenAPI Generatorというもので、APIを叩くクライアントコードを自動生成させたときでした。
なぜかcontextという謎の第一引数を渡さないといけない仕様になっていて、「何だこれ知らねえ」と思ったのを覚えています。

この本では、

- かつての私と同様「context？何それ美味しいの？」「何でこんな謎なもの使わなきゃいけないの？」という方がcontextの使い所を理解できるように
- contextを何となくで使っている方が「より誤解なく・よりよく使うためにはどうすればいいのか」というところについてもわかっていただけるように

Goのcontextについての説明をまるっとまとめていきたいと思います。

本の構成

2章: contextの概要

そもそもcontextは何者なのかという定義部分を、これがあって何が嬉しいの？という動機とあわせて紹介します。

3章: Doneメソッド

contextについてくる`Done`メソッドと、`context.WithCancel()`関数の用途について説明します。

4章: キャンセルの伝播

contextを複数作った場合において、一つをキャンセルしたらそのキャンセル信号がどう伝播していくのか、というところについて詳しく説明します。

5章: Deadlineメソッドとタイムアウト

contextについてくる`Deadline`メソッドと、`context.WithDeadline`関数・`context.WithTimeout`関数の用途について説明します。

6章: Errメソッド

contextについてくる`Err`メソッドの用途について説明します。

7章: Valueメソッド

contextについてくる`Value`メソッドと、`context.WithValue()`関数の用途について説明します。

8章: Valueメソッドを有効に使うtips

contextに与えられるkeyとvalueについて、

- keyに設定できる値・設定できない値は何か
- keyの衝突を回避する方法
- contextの性質上、valueに入れるべきではない値・入れてもいい値は何か

ということについて論じます。

9章: contextの具体的な使用例

ここでは、今まで紹介したcontextの機能の中でも、

- タイムアウトを使ったキャンセル処理
- `Value`メソッドを使った値の伝播

を複合的に使った、ミニhttpサーバーもどきのコードをお見せします。

10章: パッケージへのcontext導入について

この章では、
-「contextを構造体のフィールド内に埋め込むのは良くない」という話

- `context.TODO`関数の使い所

を、既存パッケージへのcontext対応を例に出しながら説明します。

11章: contextの内部実体

ここでは、

- `context.Context`型が「インターフェース」であるということ
- このインターフェースを満たす具体型は何だ？

ということについて軽く触れておきます。

使用する環境・バージョン

- OS: macOS Catalina 10.15.7
- go version go1.17 darwin/amd64

読者に要求する前提知識

- Goの基本的な文法の読み書きができること
- 特に並行処理・ゴールーチン・チャネルについては既知として扱います。

contextの概要

この章について

この章では

- contextとは何か？
- 何ができるのか？
- どうしてそれが必要なのか？

という点について説明します。

contextの役割

`context`パッケージの概要文には、以下のように記述されています。

> Package context defines the Context type, **which carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes.**
>
> (訳): `context`パッケージで定義されている`Context`型は、**処理の締め切り・キャンセル信号・API境界やプロセス間を横断する必要のあるリクエストスコープな値を伝達させる**ことができます。
>
> 出典:pkg.go.dev - context pkg

ここに書かれているように、`Context`型の主な役割は3つです。

- 処理の締め切りを伝達
- キャンセル信号の伝播
- リクエストスコープ値の伝達

これら3つが必要になるユースケースというのがイマイチ見えてこないな、と思っている方もいるでしょう。
次に、「どのようなときにcontextが威力を発揮するのか」という点について見ていきましょう。

contextの意義

contextが役に立つのは、一つの処理が**複数のゴールーチンをまたいで**行われる場合です。

処理が複数個のゴールーチンをまたぐ例

例えばGoでhttpサーバーを立てる場合について考えてみましょう。
httpリクエストを受け取った場合、`http.HandlerFunc`関数で登録されたhttpハンドラにて、レスポンスを返す処理が行われます。

```
func main() {
    // ハンドラ関数の定義
    h1 := func(w http.ResponseWriter, _ *http.Request) {
        io.WriteString(w, "Hello from a HandleFunc #1!\n")
    }
    h2 := func(w http.ResponseWriter, _ *http.Request) {
        io.WriteString(w, "Hello from a HandleFunc #2!\n")
    }

    http.HandleFunc("/", h1) // /にきたリクエストはハンドラh1で受け付ける
    http.HandleFunc("/endpoint", h2) // /endpointにきたリクエストはハンドラh2で
受け付ける

    // サーバー起動
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

コード出典:pkg.go.dev - http.HandlerFunc#Example

このとき内部的には、`main`関数が動いているメインゴールーチンは「リクエストが来るごとに、新しいゴールーチンを`go`文で立てる」という作業に終始しており、実際にレスポンスを返すハンドラの処理については`main`関数が立てた別のゴールーチン上で行われています。

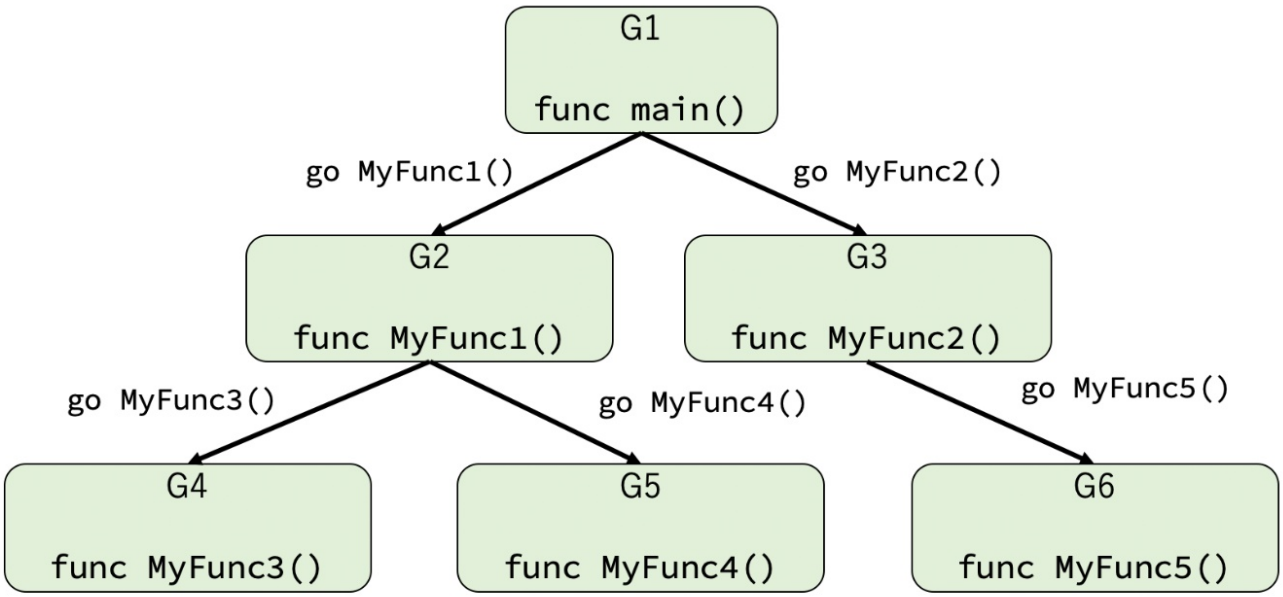
また、さらにハンドラ中で行う処理の中で、例えばDBに接続してデータを取ってきたい、そのデータ取得処理のためにまた別のゴールーチンを(場合によっては複数)立てる、という事態も往々にしてあるかと思います。

DBからのデータ取得のために複数個のゴールーチンを立てるというのは、例えば「複数個あるDBレプリカ全てにリクエストを送り、一番早くに結果が返ってきたものを採用する」といったときなどが考えられます。
Go公式ブログの"Go Concurrency Patterns: Timing out, moving on"にも、そのようなパターンについて言及されてます。

このように、

- Goのプログラマがそのことについて**意識していなくても**、ライブラリの仕様上複数のゴールーチン上に処理がまたがる
- 一つの処理を行うために、いくつものゴールーチンが**木構造的に**積み上がっていく(下図参照)

というのが決して珍しい例ではない、ということがわかっていただけると思います。



複数個ゴールーチンが絡むことによって生じる煩わしさとは

それでは、処理が複数個にゴールーチンにまたがると、どのような点が難しくなるのでしょうか。
その答えは「**情報伝達全般**」です。

基本的に、Goでは「異なるゴールーチン間での情報共有は、ロックを使ってメモリを共有するよりも、チャネルを使った伝達を使うべし」という考え方を取っています。
並行に動いている複数のゴールーチン上から、メモリ上に存在する一つのデータにそれぞれが「安全に」アクセスできることを担保するのはとても難しいからです。

> Do not communicate by sharing memory; instead, share memory by communicating.
> 出典:Effective Go

「安全に」アクセスとはどういうことか・できないとどうなるか？というところについては、拙著Zenn - Goでの並行処理を徹底解剖! 第2章をご覧ください。

困難その1 - 暗黙的に起動されるゴールーチンへの情報伝達

事前にいつどこで新規のゴールーチンが起動されるのかがわかっている場合では、新規ゴールーチン起動時に情報伝達用のチャネルを引数の一つに入れて渡していけば良いです。

```
type MyInfo int

// 情報伝達用チャネルを引数に入れる
func myFunc(ch chan MyInfo) {
    // do something
}

func main() {
    info := make(chan MyInfo)
    go myFunc(info) // 新規ゴールーチン起動時に、infoチャネルを渡していく
}
```

しかし「myFuncのような独自関数でのゴールーチンではなく、既存ライブラリ内でプログラマが意識していないところで起動されてしまうゴールーチンにどう情報伝達するのか？」というところは、プログラマ側から干渉することはできません。
そのライブラリ内で、うまくゴールーチンをまたいだ処理が確実に実装されていることを祈るしかありません。

困難その2 - 拡張性の乏しさ

また、上記のコードでは伝達する情報はMyInfo型と事前に決まっています。
しかし、追加開発で、MyInfo型以外にもMyInfo2型という新しい情報も伝達する必要が出てきた」という場合にはどうしたらいいでしょうか。

- MyInfo型の定義をinterface{}型等、様々な型に対応できるようにする
- MyFunc関数の引数に、chan MyInfo2型のチャネルを追加する

などの方法が考えられますが、前者は静的型付けの良さを完全に捨ててしまっている・受信側で元の型を判別する手段がないこと、後者は可変長に対応できないことが大きな弱点です。
このように、チャネルを使うことで伝達情報の型制約・数制約が入ってしまうことが、拡張を困難にしてしまっています。

困難その3 - 伝達制御の難しさ

また、以下のようにゴールーチンが複数起動される例に考えてみましょう。

```
func myFunc2(ch chan MyInfo) {
    // do something
    // (ただし、引数でもらったchがcloseされたら処理中断)
}

func myFunc(ch chan MyInfo) {
    // 情報伝達用のチャネルinfo1, info2, info3を
    // 何らかの手段で用意
    go myFunc2(info1)
    go myFunc2(info2)
    go myFunc2(info3)

    // do something
    // (ただし、引数でもらったchがcloseされたら処理中断)
}

func main() {
    info := make(chan MyInfo)
    go myFunc(info)

    close(info) // 別のゴールーチンで実行されているmyFuncを中断させる
}
```

main関数内にて、myFunc関数に渡したチャネルinfoをクローズすることで、myFuncが動いているゴールーチンにキャンセル信号を送信しています。

この場合、MyFunc関数の中から起動されている3つのゴールーチンmyFunc2の処理はどうなってしまうでしょうか。
これらも中断されるのか、それとも起動させたままにさせたいのか、3つとも同じ挙動をするのか、というところを正確にコントロールするには、引数として渡すチャネルを慎重に設計する必要があります。

contextによる解決

このように、「複数ゴールーチン間で安全に、そして簡単に情報伝達を行いたい」という要望は、チャネルによる伝達だけ実現しようとすると意外と難しいということがお分かりいただけたかと思います。

contextでは、ゴールーチン間での情報伝達のうち、特に需要が多い

- 処理の締め切りを伝達
- キャンセル信号の伝播
- リクエストスコープ値の伝達

の3つについて、「ゴールーチン上で起動される関数の第一引数に、context.Context型を1つ渡す」だけで簡単に実現できるようになっています。

contextの定義

それでは、context.Context型の定義を確認してみましょう。

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

出典:pkg.go.dev - context.Context

Deadline(), Done(), Err(), Value()という4つのメソッドが確認できます。

この4つのメソッドから得られる情報を使って、異なるゴールーチンからの情報を得ることができます。
contextの4つのメソッドは冪等性を持つように設計されているので、メソッドをいつ呼んでも得られる情報は同じです。

また、ゴールーチンの呼び出し側では、伝達したい情報を包含したContextを作って関数の引数に渡すことで、異なるゴールーチンと情報をシェアできるように設定します。

```
func myFunc(ctx context.Context) {
    // ctxから、メインゴールーチン側の情報を得られる
    // (例)
    // ctx.Doneからキャンセル有無の確認
    // ctx.Deadlineで締め切り時間・締め切り有無の確認
    // ctx.Errでキャンセル理由の確認
    // ctx.Valueで値の共有
```

```
}

func main() {
    var ctx context.Context
    ctx = (ユースケースに合わせたcontextの作成)
    go myFunc(ctx) // myFunc側に情報をシェア
}
```

次章予告

次からは、`context.Context`に含まれる4つのメソッドの詳細な説明をしていきます。

Doneメソッド

この章について

ゴールーチンリークを防ぐため、またエラー発生等の原因で別ゴールーチンでさせている処理が必要なくなった場合などは、ゴールーチン呼び出し元からのキャンセル処理というのが必要になります。
また、呼び出されたゴールーチン側からも、自分が親からキャンセルされていないかどうか、ということについて知る手段が必要です。

この章では、キャンセル処理をcontextを使ってどのように実現すればいいのか、という点について掘り下げていきます。

context導入前 - doneチャネルによるキャンセル処理

ゴールーチン間の情報伝達は、基本的にはチャネルで行えます。
キャンセル処理についても、「キャンセルならクローズされるチャネル」を導入することで実現することができます。

```
var wg sync.WaitGroup

// キャンセルされるまでnumをひたすら送信し続けるチャネルを生成
func generator(done chan struct{}, num int) <-chan int {
    out := make(chan int)
    go func() {
        defer wg.Done()

    LOOP:
        for {
            select {
            case <-done: // doneチャネルがcloseされたらbreakが実行される
                break LOOP
            case out <- num: // キャンセルされてなければnumを送信
            }
```

```
        }

        close(out)
        fmt.Println("generator closed")
    }()
    return out
}

func main() {
    done := make(chan struct{})
    gen := generator(done, 1)

    wg.Add(1)

    for i := 0; i < 5; i++ {
        fmt.Println(<-gen)
    }
    close(done) // 5回genを使ったら、doneチャネルをcloseしてキャンセルを実行

    wg.Wait()
}
```

この手法は、Go公式ブログの "Go Concurrency Patterns: Pipelines and cancellation #Explicit cancellation 節"でも触れられています。

contextのDoneメソッドを用いたキャンセル処理

上の処理は、contextを使って以下のように書き換えることができます。

```
var wg sync.WaitGroup

-func generator(done chan struct{}, num int) <-chan int {
+func generator(ctx context.Context, num int) <-chan int {
 out := make(chan int)
    go func() {
        defer wg.Done()

    LOOP:
        for {
            select {
-            case <-done:
+           case <-ctx.Done():
             break LOOP
            case out <- num:
            }
        }

        close(out)
        fmt.Println("generator closed")
    }()
```

```
    return out
}

func main() {
-    done := make(chan struct{})
- gen := generator(done, 1)
+  ctx, cancel := context.WithCancel(context.Background())
+    gen := generator(ctx, 1)

    wg.Add(1)

    for i := 0; i < 5; i++ {
        fmt.Println(<-gen)
    }
-    close(done)
+  cancel()

    wg.Wait()
}
```

キャンセルされる側の変更点

`generator`関数内での変更点は以下の通りです。

- `generator`に渡される引数が、キャンセル処理用の`done`チャネル→contextに変更
- キャンセル有無の判定根拠が、`<-done`→`<-ctx.Done()`に変更

```
// 再掲
-func generator(done chan struct{}, num int) <-chan int {
+func generator(ctx context.Context, num int) <-chan int {
 out := make(chan int)
    go func() {
        defer wg.Done()

    LOOP:
        for {
            select {
-            case <-done:
+          case <-ctx.Done():
             break LOOP
            case out <- num:
            }
        }

        close(out)
        fmt.Println("generator closed")
    }()
    return out
}
```

Doneメソッドによるキャンセル有無の確認

ここでcontextの`Done`メソッドが登場しました。
`Done`メソッドから何が得られているのか、もう一度定義を確認してみましょう。

```
type Context interface {
    Done() <-chan struct{}
    // (以下略)
}
```

出典:pkg.go.dev - context.Context

これを見ると、`Done`メソッドからは空構造体の受信専用チャネル(以下**Doneメソッドチャネル**と表記)が得られることがわかります。
contextへの書き換え前に使っていた`done`チャネルも空構造体用のチャネルでした。

2つが似ているのはある意味必然で、Doneメソッドチャネルは「呼び出し側からキャンセル処理がなされたらcloseされる」という特徴を持つのです。これで書き換え前の`done`チャネルと全く同じ役割を担うことができます。

Doneメソッドチャネルでできるのは、あくまで「呼び出し側からキャンセルされているか否かの確認」のみです。キャンセルされていることを確認できた後の、実際のキャンセル処理・後始末部分については自分で書く必要があります。

```
select {
case <-ctx.Done():
    // キャンセル処理は自分で書く
}
```

キャンセルする側の変更点

`main`関数内での変更点は以下の通りです。

- `done`チャネルの代わりに`context.Background()`, `context.WithCancel()`関数を用いてコンテキストを生成
- キャンセル処理が、`done`チャネルの明示的close→`context.WithCancel()`関数から得られた`cancel()`関数の実行に変更

```
// 再掲
func main() {
-    done := make(chan struct{})
- gen := generator(done, 1)
+  ctx, cancel := context.WithCancel(context.Background())
+    gen := generator(ctx, 1)

    wg.Add(1)

    for i := 0; i < 5; i++ {
```

```
            fmt.Println(<-gen)
        }
-       close(done)
+   cancel()

        wg.Wait()
    }
```

contextの初期化

まずは、`generator`関数に渡すためのコンテキストを作らなくてはいけません。
何もない0の状態からコンテキストを生成するためには、`context.Background()`関数を使います。

```
func Background() Context
```

出典:pkg.go.dev - context pkg

`context.Background()`関数の返り値からは、「キャンセルされない」「deadlineも持たない」「共有する値も何も持たない」状態のcontextが得られます。いわば「context初期化のための関数」です。

contextにキャンセル機能を追加

そして、`context.Background()`から得たまっさらなcontextを`context.WithCancel()`関数に渡すことで、「`Done`メソッドからキャンセル有無が判断できるcontext」と「第一返り値のコンテキストをキャンセルするための関数」を得ることができます。

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
```

出典:pkg.go.dev - context pkg

`WithCancel`関数から得られるcontextは、「引数として渡された親contextの設定を引き継いだ上で、`Done`メソッドによるキャンセル有無判定機能を追加した新たなcontext」ものになります。
第二返り値で得られた`cancel`関数を呼び出すことで、この`WithCancel`関数から得られるcontextのDoneメソッドチャネルをcloseさせることができます。

```
ctx, cancel := context.WithCancel(parentCtx)
cancel()

// cancel()の実行により、ctx.Done()で得られるチャネルがcloseされる
// ctxはparentCtxとは別物なので、parentCtxはcancel()の影響を受けない
```

まとめ

contextを使ったキャンセル処理のポイントは以下3つです。

- キャンセル処理を伝播させるためのコンテキストは`context.WithCancel()`関数で作ることができる
- `context.WithCancel()`関数から得られる`cancel`関数で、キャンセルを指示することができる
- `cancel`関数によりキャンセルされたら、contextのDoneメソッドチャネルがcloseされるので、それでキャンセル有無を判定する

```
// 使用した関数・メソッド
type Context interface {
    Done() <-chan struct{}
    // (以下略)
}
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
```

キャンセルの伝播

この章について

ここからは、

- 同じcontextを複数のゴールーチンで使いまわしたらどうなるか
- 親のcontextをキャンセルしたら、子のcontextはどうなるか

というキャンセル伝播の詳細な仕様を探っていきたいと思います。

同じcontextを使いまわした場合

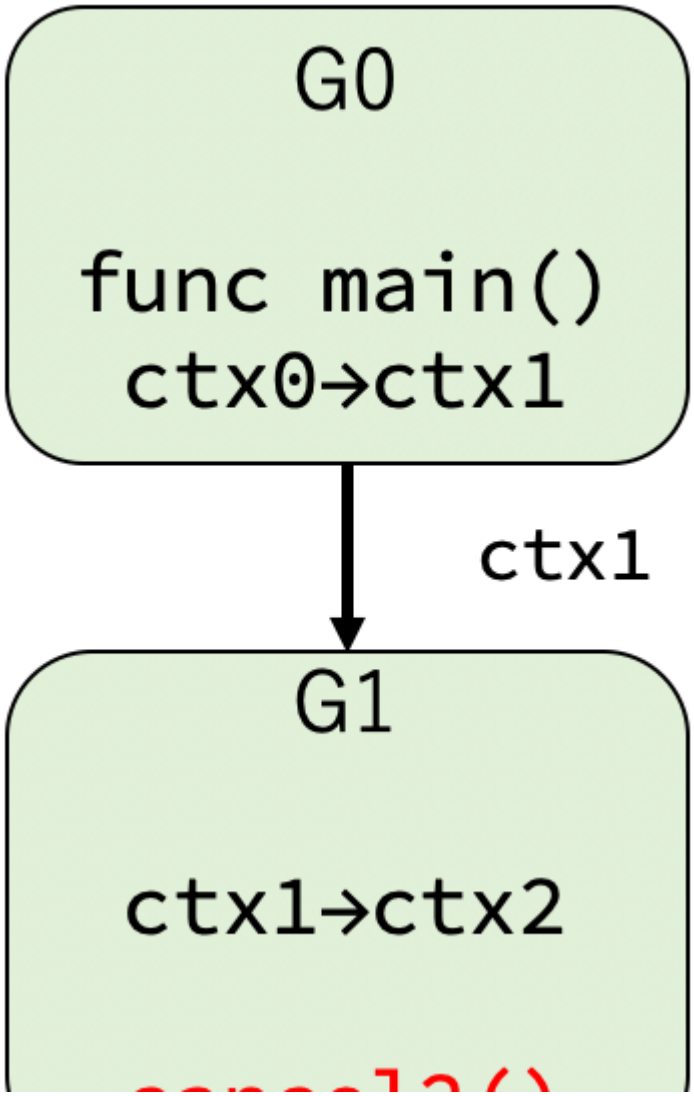
直列なゴールーチンの場合

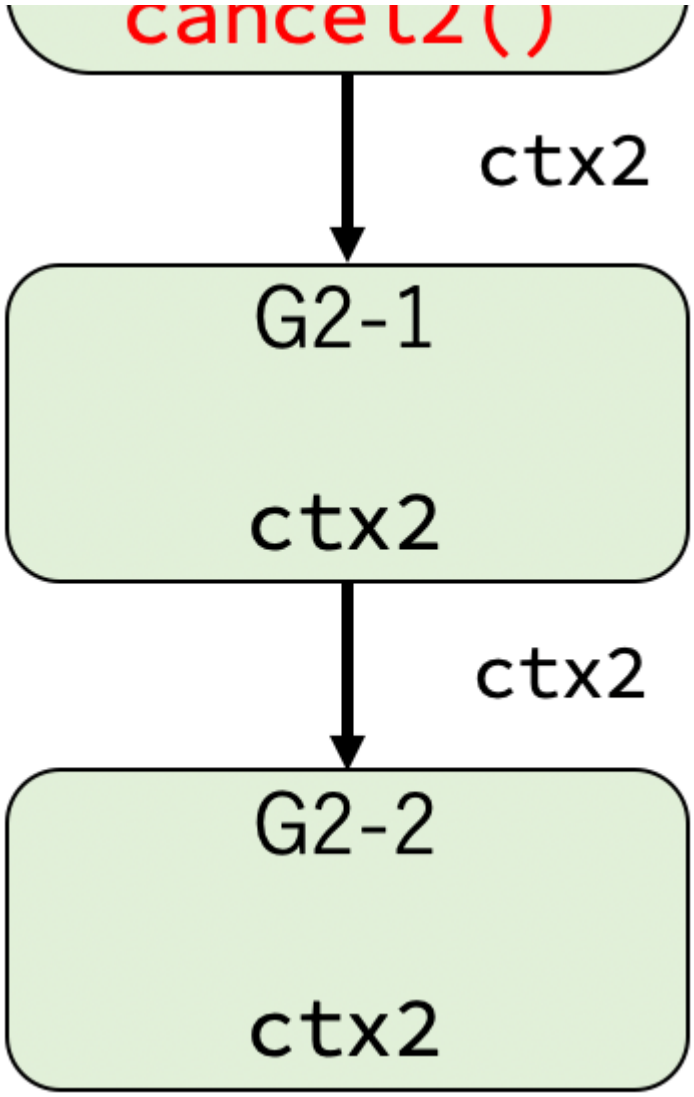
例えば、以下のようなコードを考えます。

```
func main() {
    ctx0 := context.Background()

    ctx1, _ := context.WithCancel(ctx0)
    // G1
    go func(ctx1 context.Context) {
        ctx2, cancel2 := context.WithCancel(ctx1)

        // G2-1
        go func(ctx2 context.Context) {
            // G2-2
            go func(ctx2 context.Context) {
                select {
                case <-ctx2.Done():
                    fmt.Println("G2-2 canceled")
```

```
                }
            }(ctx2)

            select {
            case <-ctx2.Done():
                fmt.Println("G2-1 canceled")
            }
        }(ctx2)

        cancel2()

        select {
        case <-ctx1.Done():
            fmt.Println("G1 canceled")
        }

    }(ctx1)

    time.Sleep(time.Second)
}
```

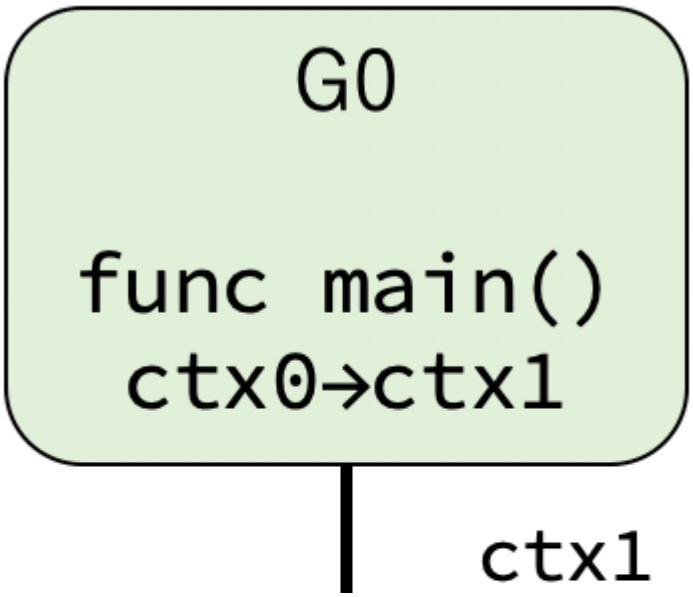
go文にて新規に立てられたゴールーチンはG1, G2-1, G2-2の3つ存在します。
それらの関係と、それぞれに引数として渡されているcontextは以下のようになっています。

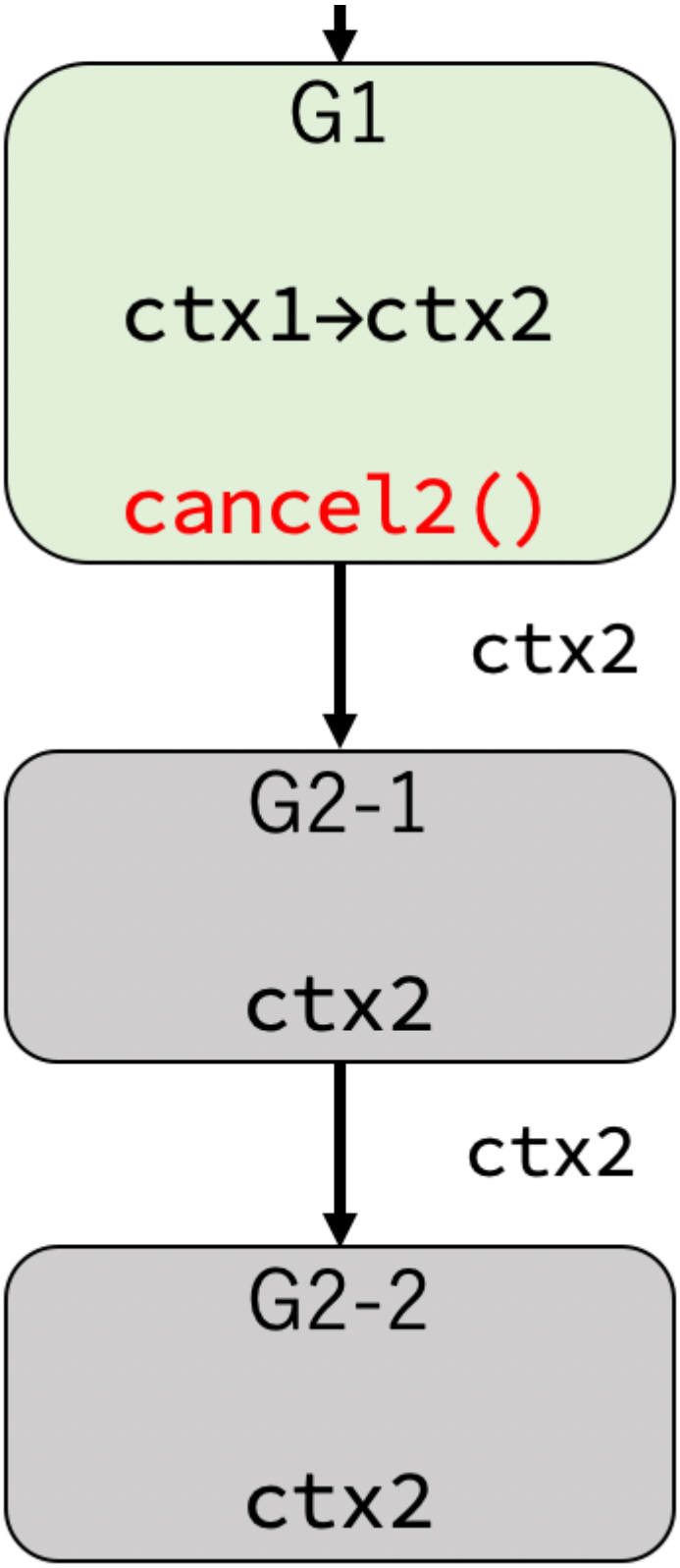




`ctx2`のキャンセルのみを実行すると、G2-1とG2-2が揃って終了し、その親であるG1は生きたままとなります。

```
$ go run main.go
G2-1 canceled
G2-2 canceled
```





width="100" loading="lazy"}

並列なゴールーチンの場合

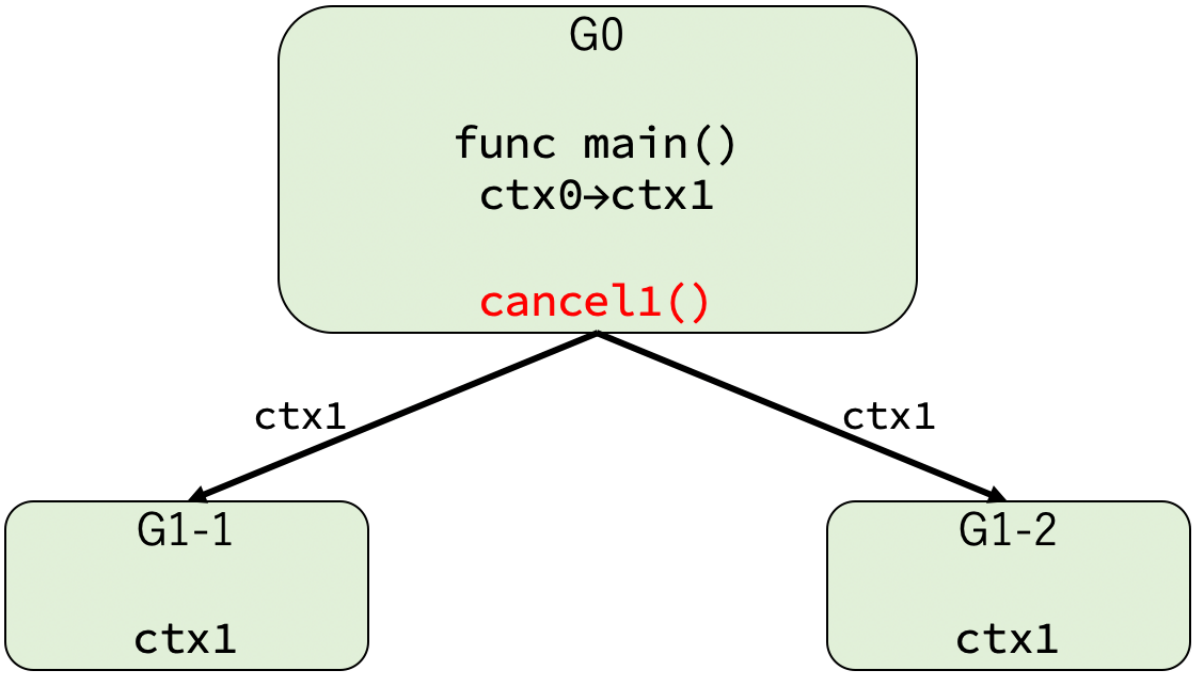
ここでの並列は、「並行処理・並列処理」の意味ではなく、直列の対義語としての並列を指します。

それでは、今度は以下のコードについて考えてみましょう。

```
func main() {
    ctx0 := context.Background()
```

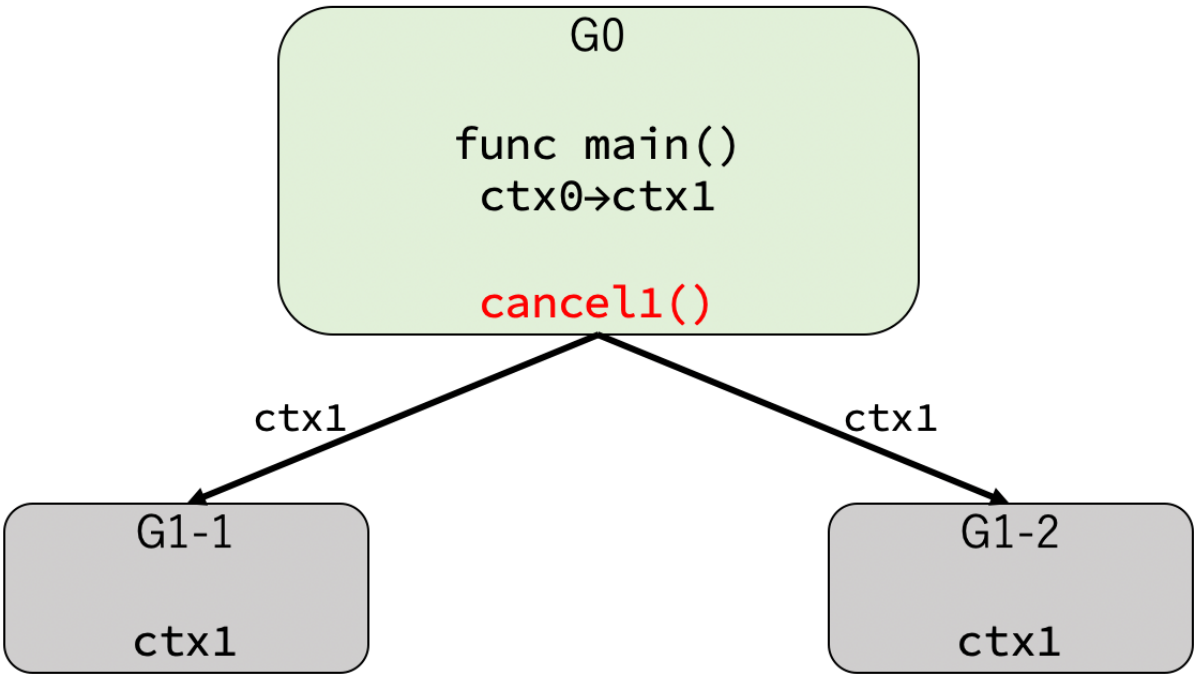
```
    ctx1, cancel1 := context.WithCancel(ctx0)
    // G1-1
    go func(ctx1 context.Context) {
        select {
        case <-ctx1.Done():
            fmt.Println("G1-1 canceled")
        }
    }(ctx1)

    // G1-2
    go func(ctx1 context.Context) {
        select {
        case <-ctx1.Done():
            fmt.Println("G1-2 canceled")
        }
    }(ctx1)

    cancel1()

    time.Sleep(time.Second)
}
```

メイン関数の中で、`go`文を二つ並列に立てて、そこに同一のcontext`ctx1`を渡しています。



ここで、`ctx1`をキャンセルすると、G1-1, G1-2ともに連動して終了します。

```
$ go run main.go
G1-1 canceled
G1-2 canceled
```



まとめ

同じcontextを複数のゴールーチンに渡した場合、それらが直列の関係であろうが並列の関係であろうが同じ挙動となります。
ゴールーチンの生死を制御するcontextが同じであるので、キャンセルタイミングも当然連動することとなります。

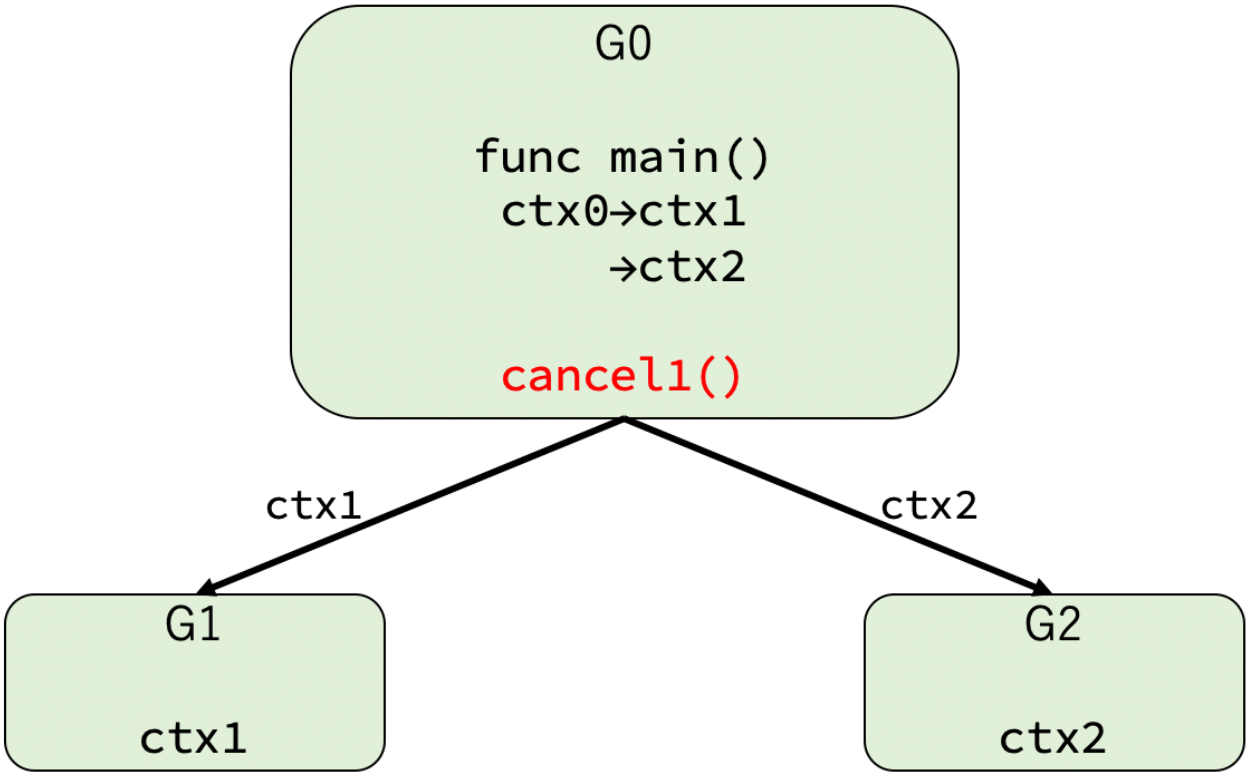
兄弟関係にあるcontextの場合

続いて、以下のようなコードを考えます。

```
func main() {
    ctx0 := context.Background()

    ctx1, cancel1 := context.WithCancel(ctx0)
    // G1
    go func(ctx1 context.Context) {
        select {
        case <-ctx1.Done():
            fmt.Println("G1 canceled")
        }
    }(ctx1)

    ctx2, _ := context.WithCancel(ctx0)
    // G2
    go func(ctx2 context.Context) {
        select {
        case <-ctx2.Done():
            fmt.Println("G2 canceled")
        }
    }(ctx2)
```

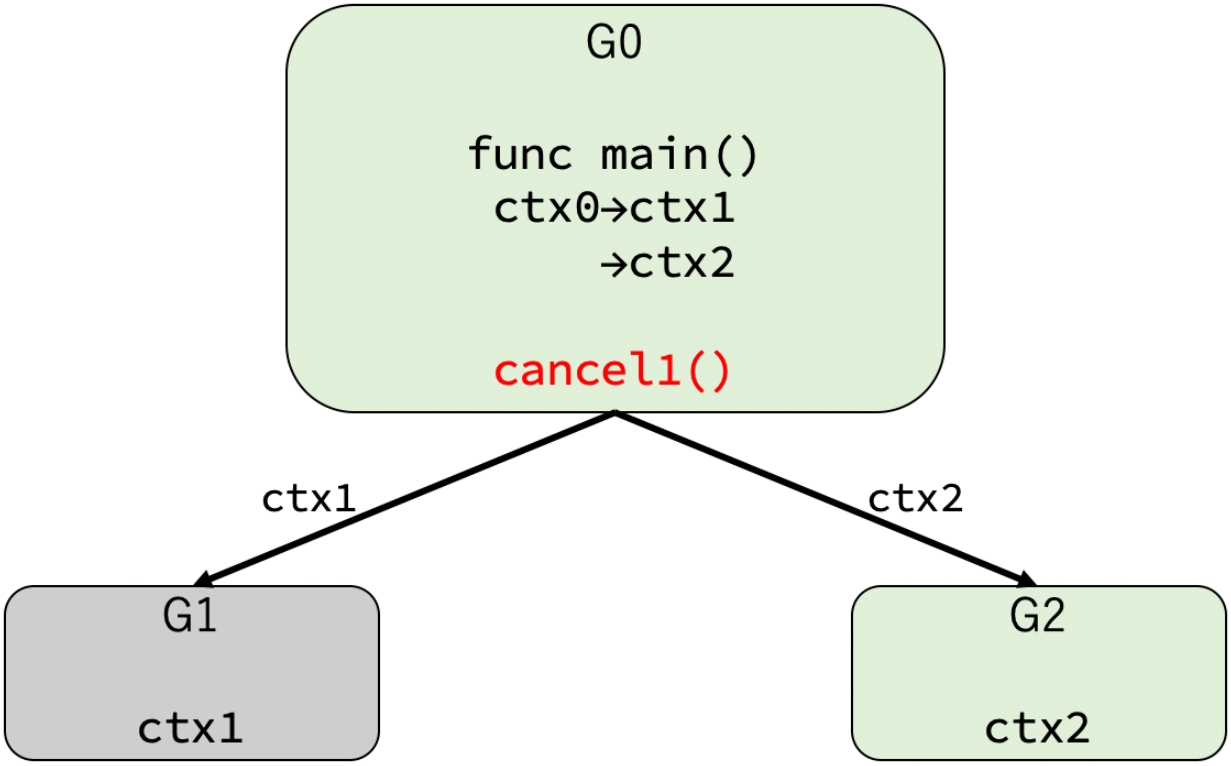
```
    cancel1()

    time.Sleep(time.Second)
}
```

メイン関数の中でgo文を二つ並列に立てて、ゴールーチンG1,G2を立てています。
そしてそれぞれには、ctx0を親にして作ったcontextctx1,ctx2を渡しています。



ここで、ctx1をキャンセルすると、G1のみが終了し、G2はその影響を受けることなく生きていることが確認できます。

```
$ go run main.go
G1 canceled
```



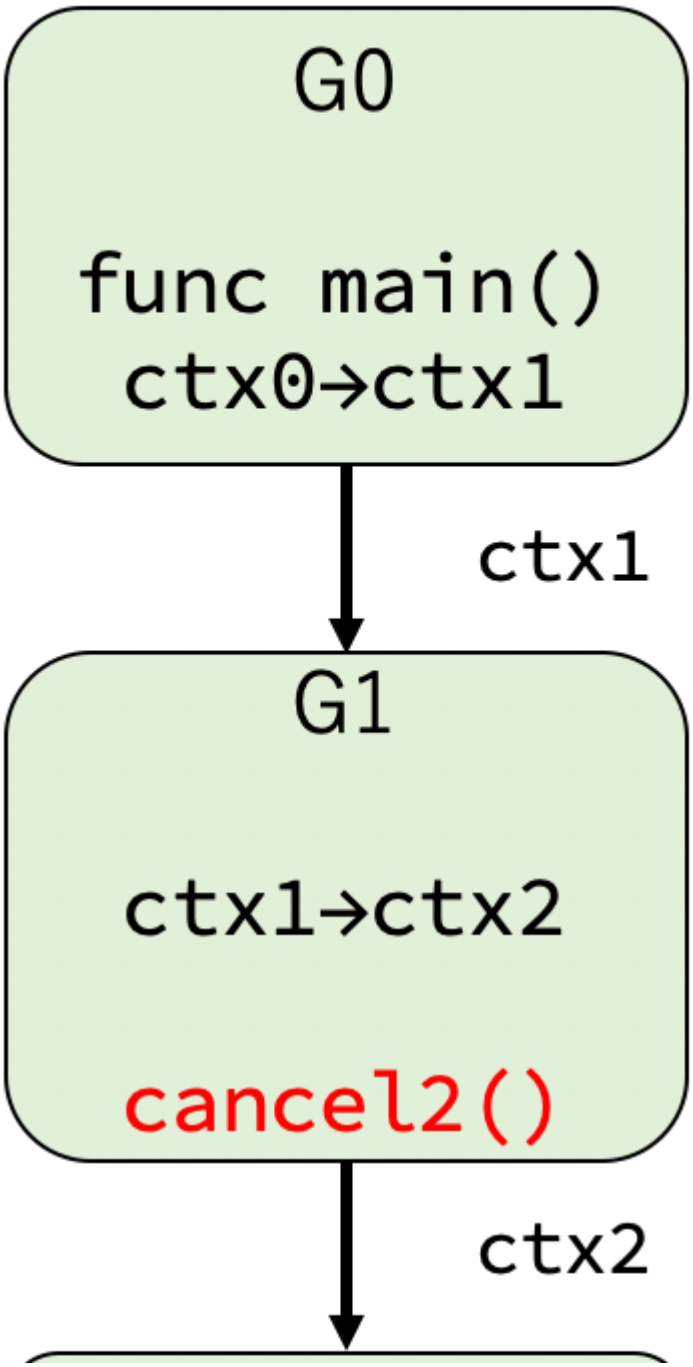
親子関係にあるcontextの場合

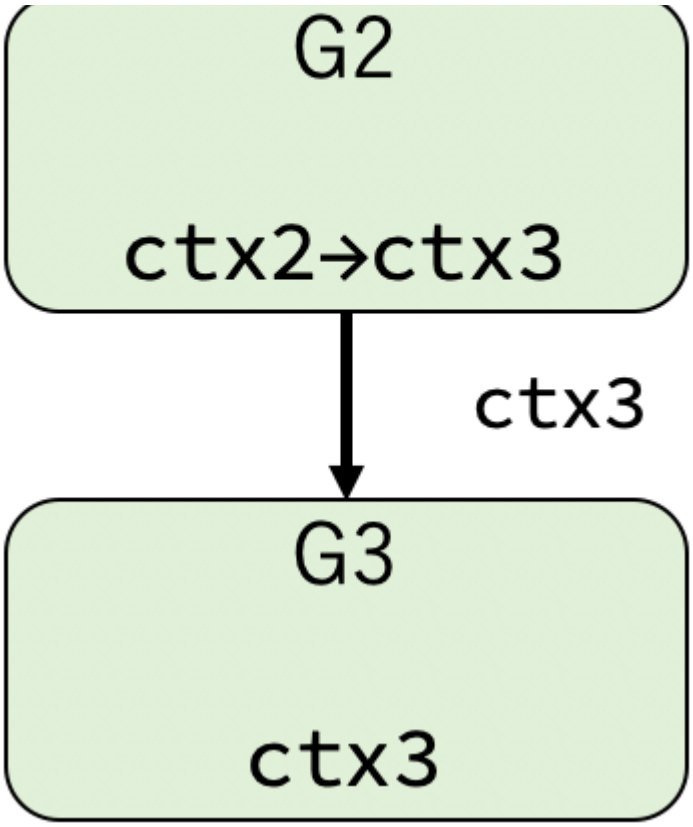
以下のようなコードを考えます。

```
func main() {
    ctx0 := context.Background()

    ctx1, _ := context.WithCancel(ctx0)
    // G1
    go func(ctx1 context.Context) {
        ctx2, cancel2 := context.WithCancel(ctx1)

        // G2
        go func(ctx2 context.Context) {
            ctx3, _ := context.WithCancel(ctx2)

            // G3
            go func(ctx3 context.Context) {
                select {
                case <-ctx3.Done():
                    fmt.Println("G3 canceled")
                }
            }(ctx3)

            select {
            case <-ctx2.Done():
                fmt.Println("G2 canceled")
            }
```

```
        }(ctx2)

        cancel2()

        select {
        case <-ctx1.Done():
            fmt.Println("G1 canceled")
        }

    }(ctx1)

    time.Sleep(time.Second)
}
```

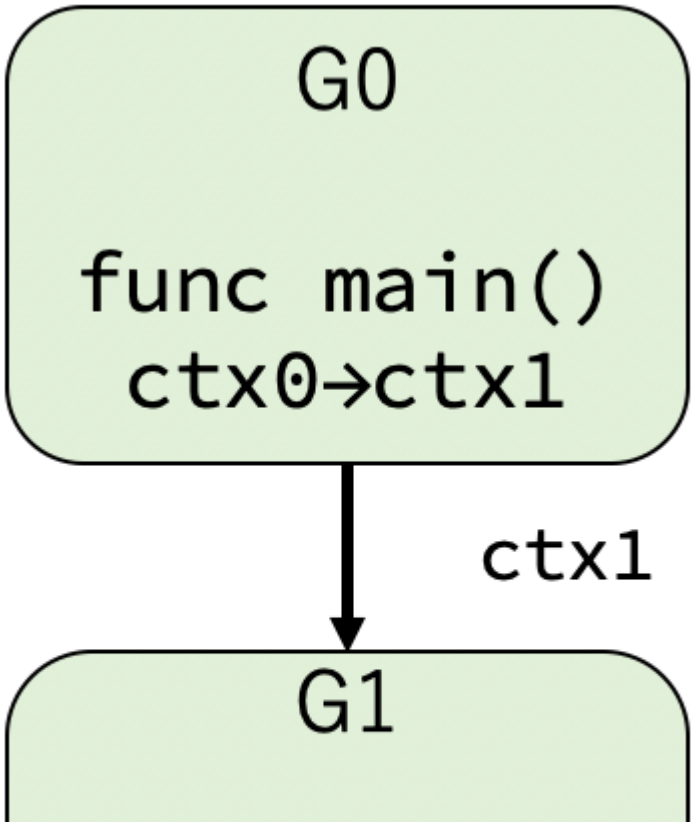
go文にて新規に立てられたゴールーチンはG1, G2, G3の3つ存在します。
それらの関係と、それぞれに引数として渡されているcontextは以下のようになっています。

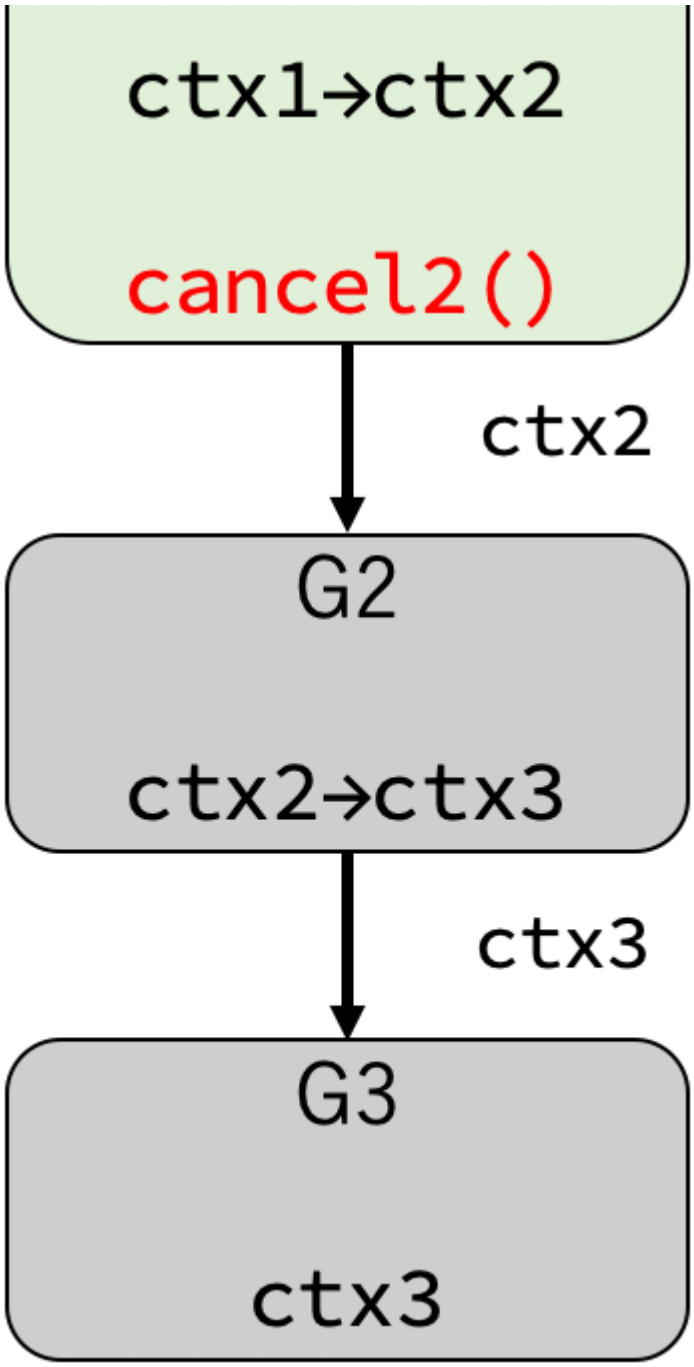




`ctx2`のキャンセルのみを実行すると、`ctx2`ともつG2と、その子である`ctx3`を持つG3が揃って終了します。
一方、`ctx2`の親である`ctx1`を持つG1は生きたままとなります。

```
$ go run main.go
G2 canceled
G3 canceled
```





これで、「親contextがキャンセルされたら、子のcontextにまで波及する」ということが確認できました。

(おまけ)子から親のキャンセル

「親から子へのキャンセル(=`ctx2`→`ctx3`)」は確認できましたが、「子から親へのキャンセル(`ctx2`→`ctx1`)」は行われませんでした。

このような設計になっていることについて、Go公式ブログ - Go Concurrency Patterns: Contextでは以下のように述べられています。

> **A Context does not have a Cancel method for the same reason the Done channel is receive-only**: the function receiving a cancelation signal is usually not the one that sends the signal.
> In particular, when a parent operation starts goroutines for sub-operations, those sub-operations should not be able to cancel the parent.

> (訳):**contextが自発的な`Cancel`メソッドを持たないのは、doneチャネルがレシーブオンリーであるのと同じ理由です。**キャンセル信号を受信した関数が、そのままその信号を別の関数に送ることになるわけではないのです。
> 特に、親となる関数が子関数の実行場としてゴールーチンを起動した場合、その子関数側から親関数をキャンセルするようなことはやるべきではありません。
>
> 出典:Go公式ブログ - Go Concurrency Patterns: Context

Deadlineメソッドとタイムアウト

この章について

`context.WithCancel`関数を使って作られたcontextは、`cancel()`関数を呼ぶことで手動でキャンセル処理を行いました。
しかし、「一定時間後に自動的にタイムアウトされるようにしたい」という場合があるでしょう。

contextには、指定したDeadlineに達したら自動的にDoneメソッドチャネルをcloseする機能を組み込むことができます。
本章ではそれについて詳しく見ていきましょう。

context導入前 - doneチャネルを用いる場合のキャンセル処理

contextを用いずにユーザーが定義した`done`チャネルによってキャンセル信号を伝播させる場合は、一定時間経過後のタイムアウトは`time.After`関数から得られるチャネルを明示的に使う必要があります。

```
var wg sync.WaitGroup

// キャンセルされるまでnumをひたすら送信し続けるチャネルを生成
func generator(done chan struct{}, num int) <-chan int {
    out := make(chan int)
    go func() {
        defer wg.Done()

    LOOP:
        for {
            select {
            case <-done: // doneチャネルがcloseされたらbreakが実行される
                break LOOP
            // case out <- num: これが時間がかかっているという想定
            }
        }

        close(out)
        fmt.Println("generator closed")
    }()
    return out
```

```
}

func main() {
    // doneチャネルがcloseされたらキャンセル
    done := make(chan struct{})
    gen := generator(done, 1)
    deadlineChan := time.After(time.Second)

    wg.Add(1)

LOOP:
    for i := 0; i < 5; i++ {
        select {
        case result := <-gen: // genから値を受信できた場合
            fmt.Println(result)
        case <-deadlineChan: // 1秒間受信できなかったらタイムアウト
            fmt.Println("timeout")
            break LOOP
        }
    }
    close(done)

    wg.Wait()
}
```

`time.After`を使ったタイムアウトについての詳細は、拙著Zenn - Goでの並行処理を徹底解剖! 第5章をご覧ください。

contextを使った実装

上の処理は、contextを使って以下のように書き換えることができます。

```
var wg sync.WaitGroup

-func generator(done chan struct{}, num int) <-chan int {
+func generator(ctx context.Context, num int) <-chan int {
 out := make(chan int)

    go func() {
        defer wg.Done()

    LOOP:
        for {
            select {
-            case <-done:
+           case <-ctx.Done():
             break LOOP
            // case out <- num: これが時間がかかっているという想定
            }
        }
```

```
          close(out)
          fmt.Println("generator closed")
      }()
      return out
  }

  func main() {
  -    done := make(chan struct{})
  - gen := generator(done, 1)
  - deadlineChan := time.After(time.Second)
  +  ctx, cancel := context.WithDeadline(context.Background(),
  time.Now().Add(time.Second))
  +    gen := generator(ctx, 1)

      wg.Add(1)

  LOOP:
      for i := 0; i < 5; i++ {
          select {
  -        case result := <-gen:
  -          fmt.Println(result)
  -      case <-deadlineChan: // 1秒間selectできなかったら
  -          fmt.Println("timeout")
  -          break LOOP

  +      case result, ok := <-gen:
  +              if ok {
  +                  fmt.Println(result)
  +              } else {
  +                  fmt.Println("timeout")
  +                  break LOOP
  +              }
       }
      }
  -    close(done)
  +  cancel()

      wg.Wait()
  }
```

キャンセルされる側の変更点

`generator`関数内での変更点は以下の通りです。

- `generator`に渡される引数が、キャンセル処理用の`done`チャネル→contextに変更
- キャンセル有無の判定根拠が、`<-done`→`<-ctx.Done()`に変更

この変更については、前章の「`Done`メソッドによるキャンセル有無判定」と内容は変わりありません。

明示的なキャンセル処理から一定時間経過後の自動タイムアウトへの変更によって生じる差異は、キャンセルする側で生成するcontextに現れます。

キャンセルする側の変更点

`main`関数内での変更点は以下の通りです。

- `done`チャネルの代わりに`context.Background()`, `context.WithDeadline()`関数を用いてコンテキストを生成
- `select`文中でのタイムアウト有無の判定方法
- キャンセル処理が、`done`チャネルの明示的close→`context.WithDeadline()`関数から得られた`cancel()`関数の実行に変更

```
// 再掲
func main() {
-    done := make(chan struct{})
- gen := generator(done, 1)
- deadlineChan := time.After(time.Second)
+  ctx, cancel := context.WithDeadline(context.Background(),
time.Now().Add(time.Second))
+    gen := generator(ctx, 1)

    wg.Add(1)

LOOP:
    for i := 0; i < 5; i++ {
        select {
-        case result := <-gen:
-         fmt.Println(result)
-     case <-deadlineChan: // 1秒間selectできなかったら
-         fmt.Println("timeout")
-         break LOOP

+     case result, ok := <-gen:
+            if ok {
+                fmt.Println(result)
+            } else {
+                fmt.Println("timeout")
+                break LOOP
+            }
     }
    }
-    close(done)
+  cancel()

    wg.Wait()
}
```

自動タイムアウト機能の追加

`WithDeadline`関数

`context.WithDeadline`関数を使うことで、指定された**時刻**に自動的にDoneメソッドチャネルがcloseされるcontextを作成することができます。

```
func WithDeadline(parent Context, d time.Time) (Context, CancelFunc)
```

出典:pkg.go.dev - context pkg

`WithDeadline`関数から得られるcontextは、「引数として渡された親contextの設定を引き継いだ上で、Doneメソッドチャネルが第二引数で指定した時刻に自動closeされる新たなcontext」ものになります。
また、タイムアウト時間前にキャンセル処理を行いたいという場合は、第二返り値で得られた`cancel`関数を呼び出すことでもDoneメソッドチャネルを手動でcloseさせることができます。

```
ctx, cancel := context.WithDeadline(parentCtx,
time.Now().Add(time.Second))
// このctxは、時刻time.Now().Add(time.Second)に自動キャンセルされる

cancel()
// 明示的にcancelさせることも可能

// ctxはparentCtxとは別物なので、parentCtxはcancel()の影響を受けない
```

`WithTimeout`関数

自動タイムアウトするタイミングを、時刻ではなく**時間**で指定したい場合は、`context.WithTimeout`関数を使います。

```
func WithTimeout(parent Context, timeout time.Duration) (Context,
CancelFunc)
```

出典:pkg.go.dev - context pkg

そのため、`WithDeadline`関数を用いたcontext生成は`WithTimeout`関数を使って書き換えることもできます。
例えば、以下の2つはどちらも「1秒後にタイムアウトさせるcontext」を生成します。

```
// 第二引数に時刻 = time.Timeを指定
ctx, cancel := context.WithDeadline(context.Background(),
time.Now().Add(time.Second))

// 第二引数に時間 = time.Durationを指定
ctx, cancel := context.WithTimeout(context.Background(), time.Second)
```

タイムアウト有無の判定

contextによる自動タイムアウトの導入によって、`main`関数内でタイムアウトしたか否かを判定するロジックが変わっています。

```
// 再掲
-deadlineChan := time.After(time.Second)
select {
-case result := <-gen:
- fmt.Println(result)
-case <-deadlineChan: // 1秒間selectできなかったら
- fmt.Println("timeout")
- break LOOP

+case result, ok := <-gen:
+    if ok {
+        fmt.Println(result)
+    } else {
+        fmt.Println("timeout")
+        break LOOP
+    }
}
```

変更前では「一定時間経っても返答が得られないかどうか」は、呼び出し側である`main`関数中で、`case`文と`time.After`関数を組み合わせる形で判定する必要がありました。

しかし、変更後はタイムアウトした場合、`gen`チャネルを得るために呼び出された側である`generator`関数中で`gen`チャネルのclose処理まで行われるようになります。
そのため、タイムアウトかどうかを判定するためには、「`gen`チャネルからの受信が、チャネルcloseによるものなのか否か(=`ok`のbool値に対応)」を見るだけで実現できるようになりました。

明示的なキャンセル処理の変更

context導入によって、明示的なキャンセル指示の方法が「`done`チャネルの明示的close→`cancel`関数の実行」に変わっています。

```
// 再掲
-close(done)
+cancel()
```

`WithDeadline`関数・`WithTimeout`関数による自動タイムアウトが行われると、Doneメソッドチャネルが自動的にcloseされます。
それでは、タイムアウトされた後に`cancel`関数を呼び出すといったいどうなるのでしょうか。
closedなチャネルをcloseしようとするとpanicになりますが、そうなってしまうのでしょうか。

正解は「**panicにならず、正常に処理が進む**」です。
context生成時に得られる`cancel`関数は、「すでにDoneメソッドチャネルがcloseされているときに呼ばれたら、何もしない」というような制御がきちんと行われています。そのためpanicに陥ることはありません。

そのため、ドキュメントでは「タイムアウト設定をしていた場合にも、明示的に`cancel`を呼ぶべき」という記述があります。

> **Even though ctx will be expired, it is good practice to call its cancellation function in any case.**
> Failure to do so may keep the context and its parent alive longer than necessary.
>
> (訳)**`ctx`がタイムアウト済みであっても、明示的に`cancel`を呼び出すべきでしょう。**
> そうでなければ、コンテキストやその親contextが不必要にメモリ上に残ったままになる可能性があります(contextリーク)。
>
> 出典:pkg.go.dev - context pkg #example-WithDeadline

Deadlineメソッドによるタイムアウト有無・時刻の確認

さて、あるcontextにタイムアウトが設定されているかどうか確認したい、ということもあるでしょう。
そのような場合には`Deadline`メソッドを使います。

contextの`Deadline`メソッドの定義を確認してみましょう。

```
type Context interface {
    Deadline() (deadline time.Time, ok bool)
    // (以下略)
}
```

出典:pkg.go.dev - context.Context

第二返り値のbool値を確認することで、「そのcontextにタイムアウトが設定されているか」を判定することができます。
設定されていれば`true`、されていなければ`false`です。
また、設定されている場合には、第一返り値にはタイムアウト時刻が格納されています。

```
ctx := context.Background()
fmt.Println(ctx.Deadline()) // 0001-01-01 00:00:00 +0000 UTC false

fmt.Println(time.Now()) // 2021-08-22 20:03:53.352015 +0900 JST
m=+0.000228979
ctx, _ = context.WithTimeout(ctx, 2*time.Second)
fmt.Println(ctx.Deadline()) // 2021-08-22 20:03:55.352177 +0900 JST
m=+2.000391584 true
```

まとめ

contextでタイムアウトを行う場合のポイントは以下4つです。

- 自動タイムアウトさせるためのcontextは、`WithDeadline`関数・`WithTimeout`関数で作れる
- タイムアウトが設定されているcontextは、指定時刻にDoneメソッドチャネルがcloseされる

- WithDeadline関数・WithTimeout関数それぞれから得られるcancel関数で、タイムアウト前後にもキャンセルを明示的に指示することができる
- そのcontextのタイムアウト時刻・そもそもタイムアウトが設定されているかどうかはDeadlineメソッドで確認できる

```
// 使用した関数・メソッド
type Context interface {
    Deadline() (deadline time.Time, ok bool)
    // (以下略)
}
func WithDeadline(parent Context, d time.Time) (Context, CancelFunc)
func WithTimeout(parent Context, timeout time.Duration) (Context,
CancelFunc)
```

Errメソッド

この章について

この章では、contextに含まれているErrメソッドの概要・使いどころについて説明します。

キャンセルか、タイムアウトか

キャンセルされる側の関数では、Doneメソッドチャネルでキャンセルを認識した段階で後処理の実行に移ることが多いと思います。
しかし、「明示的なキャンセルとタイムアウトによるキャンセルで、後処理を変えたい」という場合、現状のDoneメソッドではそのどちらなのかを判断する術がありません。

```
func generator(ctx context.Context, num int) <-chan int {
    out := make(chan int)

    go func() {
        defer wg.Done()

    LOOP:
        for {
            select {
            case <-ctx.Done():
                // タイムアウトで止まったのか？
                // それともキャンセルされて止まったのか？
                // Doneメソッドだけでは判定不可
                break LOOP
            case out <- num:
            }
        }
```

```
        close(out)
        fmt.Println("generator closed")
    }()
    return out
}
```

contextパッケージに存在する2種類のエラー変数

contextパッケージには、2種類のエラーが定義されています。

```
var Canceled = errors.New("context canceled")
var DeadlineExceeded error = deadlineExceededError{}
```

出典:pkg.go.dev context-variables

一つが`Canceled`で、contextが明示的にキャンセルされたときに使用されます。
もう一つが`DeadlineExceeded`で、タイムアウトで自動キャンセルされた場合に使用されます。

また`DeadlineExceeded`には`Timeout`メソッドと`Temporary`メソッドがついており、`net.Error`インターフェースも追加で満たすようになっています。

```
// deadlineExceededError型の定義
type deadlineExceededError struct{}

func (deadlineExceededError) Error() string   { return "context deadline
exceeded" }
func (deadlineExceededError) Timeout() bool   { return true }
func (deadlineExceededError) Temporary() bool { return true }
```

出典:context/context.go

```
// net.Errorインターフェース
type Error interface {
    error
    Timeout() bool   // Is the error a timeout?
    Temporary() bool // Is the error temporary?
}
```

出典:pkg.go.dev - net pkg

Errメソッド

contextのErrメソッドからは、

- contextがキャンセルされていない場合: nil
- contextが明示的にキャンセルされていた場合: Canceledエラー
- contextがタイムアウトしていた場合: DeadlineExceededエラー

が得られるようになっています。

```
type Context interface {
    Err() error
    // (以下略)
}
```

出典:pkg.go.dev - context.Context

そのため、前述した「明示的なキャンセルとタイムアウトによるキャンセルで、後処理を変えたい」という場合は、以下のように実現することができます。

```
select {
case <-ctx.Done():
    if err := ctx.Err(); errors.Is(err, context.Canceled) {
        // キャンセルされていた場合
        fmt.Println("canceled")
    } else if errors.Is(err, context.DeadlineExceeded) {
        // タイムアウトだった場合
        fmt.Println("deadline")
    }
}
```

Valueメソッド

この章について

この章では、contextを使った「値の伝達」について説明します。

context未使用の場合 - 関数の引数での実装

今まで使用してきたgeneratorに、以下のような機能を追加してみましょう。

- ユーザーID、認証トークン、トレースIDも渡す
- generatorは、終了時にこれらの値をログとして出力する

まず一つ考えられる例としては、これらの値を伝達できるように、generator関数の引数を3つ追加するという方法です。

```
var wg sync.WaitGroup

func generator(ctx context.Context, num int, userID int, authToken string,
traceID int) <-chan int {
    out := make(chan int)
    go func() {
        defer wg.Done()

    LOOP:
        for {
            select {
            case <-ctx.Done():
                break LOOP
            case out <- num:
            }
        }

        close(out)
        fmt.Println("log: ", userID, authToken, traceID) // log:  2
xxxxxxxx 3
        fmt.Println("generator closed")
    }()
    return out
}

func main() {
    ctx, cancel := context.WithCancel(context.Background())
    gen := generator(ctx, 1, 2, "xxxxxxxx", 3)

    wg.Add(1)

    for i := 0; i < 5; i++ {
        fmt.Println(<-gen)
    }
    cancel()

    wg.Wait()
}
```

この方法は簡単ですが、これから「さらに別の値も追加で`generator`に渡したくなった」という場合に困ってしまいます。その度に関数の引数を一つずつ追加していくのは骨が折れますね。
つまり、関数の引数を利用する方法は拡張性という観点で難があるのです。

contextを使用した値の伝達

上の処理は、contextの力を最大限使えば、以下のように書き直すことができます。

```
-func generator(ctx context.Context, num int, userID int, authToken
string, traceID int) <-chan int {
```

```
 +func generator(ctx context.Context, num int) <-chan int {
  out := make(chan int)
     go func() {
         defer wg.Done()

     LOOP:
         for {
             select {
             case <-ctx.Done():
                 break LOOP
             case out <- num:
             }
         }

         close(out)
 +     userID, authToken, traceID := ctx.Value("userID").(int),
 ctx.Value("authToken").(string), ctx.Value("traceID").(int)
      fmt.Println("log: ", userID, authToken, traceID)
         fmt.Println("generator closed")
     }()
     return out
 }

 func main() {
     ctx, cancel := context.WithCancel(context.Background())
 -    gen := generator(ctx, 1, 2, "xxxxxxxx", 3)

 + ctx = context.WithValue(ctx, "userID", 2)
 +    ctx = context.WithValue(ctx, "authToken", "xxxxxxxx")
 +    ctx = context.WithValue(ctx, "traceID", 3)
 +    gen := generator(ctx, 1)

     wg.Add(1)

     for i := 0; i < 5; i++ {
         fmt.Println(<-gen)
     }
     cancel()

     wg.Wait()
 }
```

キャンセルする側の変更点

`main`関数内での変更点は「`generator`関数に渡したい値を、関数の引数としてではなく、contextに付加している」というところです。

WithValue関数による、contextへの値付加

`WithCancel`関数や`WithTimeout`関数を用いて、contextにキャンセル機能・タイムアウト機能を追加できたように、`WithValue`関数を使うことで、contextに値を追加することができます。

```
func WithValue(parent Context, key, val interface{}) Context
```

出典:pkg.go.dev - context pkg

`WithValue`関数から得られるcontextは、引数`key`をkeyに、引数`val`値をvalueとして内部に持つようになります。

```
ctx = context.WithValue(parentCtx, "userID", 2)
// ctx内部に、keyが"userID", valueが2のデータが入る
```

キャンセルされる側の変更点

`generator`関数側での変更点は、「関数の引数→contextの中へと移動した値を、`Value`メソッドを使って抽出する作業が入った」というところです。

Valueメソッドによるcontext中の値抽出

まずは、contextにおける`Value`メソッドの定義を見てみましょう。

```
type Context interface {
    Value(key interface{}) interface{}
    // (以下略)
}
```

出典:pkg.go.dev - context.Context

引数にkeyを指定することで、それに対応するvalueを**インターフェースの形で**取り出すことができます。

```
ctx := context.WithValue(parentCtx, "userID", 2)

interfaceValue := ctx.Value("userID") // keyが"userID"であるvalueを取り出す
intValue, ok := interfaceValue.(int)  // interface{}をint型にアサーション
```

まとめ & 次章予告

contextで値を付加・取得する際には、

- 付加: `WithValue`関数
- 取得: `Value`メソッド

を利用します。

```
// 使用した関数・メソッド
type Context interface {
    Value(key interface{}) interface{}
    // (以下略)
}
func WithValue(parent Context, key, val interface{}) Context
```

しかし、それぞれの引数・返り値を見ていただければわかる通り、keyとvalueはcontextを介した時点で全て`interface{}`型になってしまいます。
また、contextに値が入っているのかどうかパッと見て判断する方法がないため、これは見方を変えると「引数となりうる値を、contextで隠蔽している」という捉え方もできます。

それゆえにcontextへの値付加を効果的に使うのは、これらの懸念点をうまく解決できるようなノウハウが必要となります。
次章では、contextの値をうまく使うための方法について詳しく掘り下げていきます。

Valueメソッドを有効に使うtips

この章について

前章でも説明した通り、contextへの値付加というのは

- keyとvalueはcontextを介した時点で全て`interface{}`型になる
- 見方を変えると「引数となりうる値を、contextで隠蔽している」という捉え方にもなる

という点で、扱い方が難しい概念です。

この章では、「contextのvalueを、危うさなしに使うにはどういう設計にしたらいいか」ということについて考察していきたいと思います。

contextに与えるkeyの設定

keyに設定できる値

> The provided key must be comparable.
> (訳) keyに使用する値は比較可能なものでなくてはなりません。
>
> 出典: pkg.go.dev - context.WithValue

これはよくよく考えてもらえば当たり前のことをいってるな、ということがわかると思います。
contextの`Value(key)`メソッドにて「引数に与えたkeyを内部に持つvalueがないかな」という作業をすることを想像すると、「引数とcontextが持っているkeyは**等しいかどうか(=比較可能かどうか)**」ということが決定できないといけないのです。

比較可能(comparable)な値の定義については、Goの言語仕様書に明確に定義されています。

- bool値は比較可能であり、`true`同士と`false`同士が等しいと判定される

- 整数値(int, int64など), 浮動小数点値(float32, float64)は比較可能
- 複素数値は比較可能であり、2つの複素数の実部と虚部が共に等しい場合に等しいと判定される
- 文字列値は比較可能
- ポインタ値は比較可能であり、「どちらも同じ変数を指している場合」と「どちらも`nil`である場合」に等しいと判定される
- チャネル値は比較可能であり、「どちらも同様の`make`文から作られている場合」と「どちらも`nil`である場合」に等しいと判定される
- インターフェース値は比較可能であり、「どちらも同じdynamic type・等しいdynamic valueを持つ場合」と「どちらも`nil`である場合」に等しいと判定される
- 非インターフェース型の型`X`の値`x`と、インターフェース型`T`の値`t`は、「型`X`が比較可能でありかつインターフェース`T`を実装している場合」に比較可能であり、「`t`のdynamic typeとdynamic valueがそれぞれ`X`と`x`であった場合」に等しいと判定される
- 構造体型はすべてのフィールドが比較可能である場合にそれ自身も比較可能となり、それぞれの対応するnon-blankなフィールドの値が等しい場合に2つの構造体値が等しいと判定される
- 配列型は、その配列の基底型が比較可能である場合にそれ自身も比較可能となり、全ての配列要素が等しい場合に2つの配列値は等しいと判定される

逆に、スライス、マップ、関数値などは比較可能ではない(not comparable)ため、contextのkeyとして使うことはできません。

dynamic type/valueとは何か？
変数定義時に明確に型宣言されていない場合において、コンパイル時にそれに適した型・値であるdynamic type/valueが与えられます。

```
// staticなtype・valueの例
var x interface{}  // x is nil and has static type interface{}
var v *T           // v has value nil, static type *T

// dynamicなtype・valueの例
x = 42             // x has value 42 and dynamic type int
x = v              // x has value (*T)(nil) and dynamic type *T
```

コード出典:Go言語仕様書#Variables

keyの衝突

contextに与えるkeyについて、注意深く設計していないと「keyの衝突」が起こる可能性があります。

悪い例

状況設定

`hoge`と`fuga`2つのパッケージにて、同じkeyでcontextに値を付加する関数`SetValue`を用意しました。

```
// hoge
func SetValue(ctx context.Context) context.Context {
    return context.WithValue(ctx, "a", "b") // hoge pkgの中で("a", "b")という
```

```
key-valueを追加
}

// fuga
func SetValue(ctx context.Context) context.Context {
    return context.WithValue(ctx, "a", "c") // fuga pkgの中で("a", "c")という
key-valueを追加
}
```

そして、`main`関数内で作ったcontextに、`hoge.SetValue`→`fuga.SetValue`の順番で値を付加していきます。

```
import (
    "bad/fuga"
    "bad/hoge"
    "context"
)

func main() {
    ctx := context.Background()

    ctx = hoge.SetValue(ctx)
    ctx = fuga.SetValue(ctx)

    hoge.GetValueFromHoge(ctx) // hoge.SetValueでセットしたkey"a"に対する
Value(="b")を見たい
    fuga.GetValueFromFuga(ctx) // fuga.SetValueでセットしたkey"a"に対する
Value(="c")を見たい
}
```

値を付加した後に、それぞれの`GetValueFromXXX`関数で実際にどんなvalueが格納されているのか確認しています。

```
func GetValueFromHoge(ctx context.Context) {
    val, ok := ctx.Value("a").(string)
    fmt.Println(val, ok)
}

func GetValueFromFuga(ctx context.Context) {
    val, ok := ctx.Value("a").(string)
    fmt.Println(val, ok)
}
```

結果

これを実行すると、以下のようになります。

```
$ go run main.go
c true  // hoge.GetValueFromHoge(ctx)からの出力
c true  // fuga.GetValueFromFuga(ctx)からの出力
```

hogeパッケージの中でcontextに値"b"を付加していたのに、hoge.GetValueFromHoge関数で確認できたvalueは"c"でした。
これは、hogeとfugaで同じkey"a"を利用してしまったため、key"a"に対応するvalueは、後からSetしたfugaの方の"c"が使用されてしまうのです。

解決策: パッケージごとに独自の非公開key型を導入

このようなkeyの衝突を避けるために、Goでは「keyとして使用するための独自のkey型」を導入するという手段を公式で推奨しています。
context.WithValue関数の公式ドキュメントにも、以下のような記述があります。

> The provided key should not be of type string or any other built-in type to avoid collisions between packages using context.
> **Users of WithValue should define their own types for keys.**
>
> (訳)異なるパッケージ間でcontextを共有したときのkey衝突を避けるために、keyにセットする値にstring型のようなビルトインな型を使うべきではありません。
> その代わり、**ユーザーはkeyには独自型を定義して使うべき**です。
>
> 出典: pkg.go.dev - context.WithValue

コード改修

hoge,fugaパッケージの中身を、それぞれ以下のように改修します。

```
+// hoge

+type ctxKey int
+
+const (
+    a ctxKey = iota
+)

func SetValue(ctx context.Context) context.Context {
-    return context.WithValue(ctx, "a", "b")
+  return context.WithValue(ctx, a, "b")
}

func GetValueFromHoge(ctx context.Context) {
-    val, ok := ctx.Value("a").(string)
+  val, ok := ctx.Value(a).(string)
 fmt.Println(val, ok)
}
```

```
+// fuga

+type ctxKey int
+
+const (
+    a ctxKey = iota
+)

func SetValue(ctx context.Context) context.Context {
-    return context.WithValue(ctx, "a", "c")
+  return context.WithValue(ctx, a, "c")
}

func GetValueFromFuga(ctx context.Context) {
-    val, ok := ctx.Value("a").(string)
+  val, ok := ctx.Value(a).(string)
 fmt.Println(val, ok)
}
```

`hoge`,`fuga`パッケージ共に`ctxKey`型という非公開型を導入し、それぞれ`ctxKey`型の定数`a`をkeyとしてcontextに値を付与しています。

この改修を終えた後に、先ほどと同じ`main`関数を実行したらどうなるでしょうか。

結果

```
$ go run main.go
b true  // hoge.GetValueFromHoge(ctx)からの出力
c true  // fuga.GetValueFromFuga(ctx)からの出力
```

無事衝突することなく、`hoge.GetValueFromHoge`関数からは`hoge`パッケージで付加されたvalue`"b"`が、`fuga.GetValueFromFuga`関数からは`fuga`パッケージで付加されたvalue`"c"`が確認できました。

これは、contextに付与された値のkeyがそれぞれ

- `hoge`パッケージ内: `hoge.ctxKey`型の定数`a = itoa`
- `fuga`パッケージ内: `fuga.ctxKey`型の定数`a = itoa`

であるからです。
各パッケージ内で独自の型を作ったことにより、`hoge`と`fuga`パッケージ双方`iota`で同じ見た目の値になったとしても、型が異なるので違う値扱いになり衝突しなくなるのです。
また、独自型を非公開にすれば、keyの衝突を避けるためには「`hoge`パッケージ内で同じkeyを使ってないか」「`fuga`パッケージ内で同じkeyを使っていないか」というところのみ気にすればいいので、contextが断然扱いやすくなります。

また、同じパッケージ内でのkey衝突に関しても、「keyをまとめて非公開型の定数で用意してから、全て`iota`で値をセット」という方法をとれば簡単に回避可能です。

go-staticcheckという静的解析ツールでは、独自非公開型を定義せずにビルトイン型(`int`や`string`のように、Goに元からある型)をkeyにしている`context.WithValue`関数を見つけると、
`should not use built-in type xxxx as key for value; define your own type to avoid collisions (SA1029)`
という警告が出るようになっています。

(210829追記)

syumaiさん(@__syumai)から以下のようなコメントいただきました！
ありがとうございます！

https://twitter.com/__syumai/status/1431640657311846408

```
// ある一定の型の定数でkeyを区別する
type ctxKet int
const (
    a ctxKey = iota
    b
)

↓

// そもそもkeyが違えば型も変えてしまう
type ctxKeyA struct{}
type ctxKeyB struct{}
```

`int`型の`iota`にするよりも、空構造体`struct{}`を採用することで、メモリアロケーションを抑えることができるというメリットがあります。

valueとして与えてもいいデータ・与えるべきでないデータ

「contextの値として付加するべき値はどのようなものがふさわしいか？」というのは、Goコミュニティの中で盛んに議論されてきたトピックです。
数々の人が様々な使い方をして、その結果経験則として分かったことを一言でいうならば、

> **Use context Values only for request-scoped data that transits processes and APIs**, not for passing optional parameters to functions.
>
> (訳)contextのvalueは、関数のoptionalなパラメータを渡すためにではなく、**プロセスやAPI間を渡り歩くリクエストスコープなデータを伝播するために使うべき**である。
>
> 出典: pkg.go.dev - context

これについて、もっと深く具体例を出しながら論じていきましょう。

valueとして与えるべきではないデータ

関数の引数

関数の引数となるべきものを、contextの値として付加するべきではありません。
「関数の引数とは何か？」ということをはっきりさせておくと、ここでは「その関数の**挙動**を決定づける因子」としておきましょう。

例えば、以下のようなコードを考えます。

```
func doSomething(ctx context.Context) {
    isOdd(ctx) // ctxに入っているkey=numに対応する値が、奇数かどうかを判定する関数
}

func main() {
    ctx := context.Background()
    ctx = prepareContext1()
    ctx = prepareContext2()
    ctx = prepareContext3()

    doSomething(ctx)
}
```

これには問題点があります。

- コメントがないと「`isOdd`関数は、contextの『`num`』というkeyの偶奇を見ているんだな」という情報がわからない
- `doSomething`関数の引数として渡されているcontextが、いつどこで`key=num`の値を付加されているのかが非常に分かりにくい
- contextにどのような値が入っているのかがわからないので、`isOdd`関数の結果がどうなるのか予想が非常につきにくい

簡単にいうと、`isOdd`関数の**挙動**を決めるための引数がcontextの中に**隠蔽**されてしまっているため、非常に見通しがつきにくいコードになってしまっているのです。

それでは、`isOdd`関数の挙動を決める「判定対象の数値」を、`isOdd`関数の引数にしたらどうなるでしょうか。

```
func doSomething(ctx context.Context, num int) {
    isOdd(num) // numが奇数かどうか判定する関数
}

func main() {
    ctx := context.Background()
    ctx = prepareContext1()
    ctx = prepareContext2()
    ctx = prepareContext3()

    num := 1

    doSomething(ctx, num)
}
```

こうすることで、

- `isOdd`関数が見ているのは、引数の`num`のみだということが明確
- 「`doSomething`関数内で呼ばれている`isOdd`関数の挙動を決定するのは、`main`関数内で定義されている変数`num`である」ということが明確
- コードの実行結果が、`num=1`であるため奇数判定されるだろうという予測が容易に立つ

という点で非常に良くなりました。

繰り返しますが、「関数の挙動を決める変数」というのは、引数の形で渡すべきです。contextの中に埋め込む形で隠蔽するべきではありません。

type-unsafeになったら困るもの

再び先ほどの`isOdd`関数の例を挙げてみましょう。

contextを使った`isOdd`関数の実装は以下のようになっていました。

```
const num ctxKey = 0

func isOdd(ctx context.Context) {
    num, ok := ctx.Value(num).(int) // 型アサーション
    if ok {
        if num%2 == 1 {
            fmt.Println("odd number")
        } else {
            fmt.Println("not odd number")
        }
    }
}

func doSomething(ctx context.Context) {
    isOdd(ctx) // ctxに入っているkey=numに対応する値が、奇数かどうかを判定する関数
}
```

`isOdd`関数の中で、contextから得られるkey=numの値が、`int`型に本当になるのかどうかを確認するアサーション作業が入っているのがわかるかと思います。
これは、「contextに渡した時点で、keyとvalueは`interface{}`型になってしまう」ゆえに起こる現象です。

```
// WithValueで渡した時点でkeyもvalueもinterface{}型になり、元の型情報は失われてしまう
func WithValue(parent Context, key, val interface{}) Context

// 当然、取り出す時も型情報が失われたinterface{}型となる
type Context interface {
    Value(key interface{}) interface{}
}
```

isOdd関数の引数に判定対象numを入れてしまう形ならば、型アサーションを排除することができます。
これは、関数の引数としてなら、変数numの元の型であるintを保全することができるからです。

```
func isOdd(ctx context.Context, num int) {
    // 型アサーションなし
    if num%2 == 1 {
        fmt.Println("odd number")
    } else {
        fmt.Println("not odd number")
    }
}

func doSomething(ctx context.Context) {
    isOdd(ctx, 1) // 第二引数を、奇数かどうかを判定する関数
}
```

contextに渡した値は、interface{}型となって型情報が失われるということを意識するべきです。
そのため、type-unsafeになったら困る値をcontextに渡すべきではありません。

可変な値

今度は先ほどのisOdd関数を、以下のように使ってみましょう。

```
func doSomethingSpecial(ctx context.Context) context.Context {
    return context.WithValue(ctx, num, 2)
}

func main() {
    ctx := context.Background()
    ctx = context.WithValue(ctx, num, 1)

    isOdd(ctx) // odd

    ctx = doSomethingSpecial(ctx)

    isOdd(ctx) // ???
}
```

main関数内で与えたcontextの値は当初1だったので、isOdd関数の結果は「奇数」判定されるでしょう。
しかし、その後にdoSomethingSpecialという全然スペシャルではない関数の実行が挟まれています。
そのため、isOdd(ctx)という呼び出しの字面は同じでも、2回目のisOdd関数の結果が1回目のそれと同じになるかどうか、というのが一目ではわからなくなってしまいました。

これも先ほど述べた内容ではあるのですが、contextの中に値を付与するというのは下手したら「context中に変数を隠蔽する」ということにもなりかねます。
そのため、「contextの中には何が入っているのか？」の見通しを良くするために、contextに渡す値というのは不変値が望ましいでしょう。

ゴールーチンセーフではない値

そもそも、contextは「複数のゴールーチン間で情報伝達をするための仕組み」でした。
そのため、contextに渡すvalueというのも、異なる並行関数で扱われることを想定して、ゴールーチンセーフなものにする必要があります。

> The same Context **may be passed to functions running in different goroutines**
>
> (訳)同一のcontextは、**異なるゴールーチン上で動いている関数に渡される可能性があります。**
>
> 出典: pkg.go.dev - context

ゴールーチンセーフでない値の例として、スライスが挙げられます。
例えば以下のようにゴールーチンを10個立てて、それらの中で個別にあるスライス`slice`に要素を一つずつ追加していったとしても、最終的な`len(slice)`の値が`10`になるとは限りません。
これは、スライスがゴールーチンセーフではなく、`append`の際の排他処理が取れていないからです。

```
func main() {
    var wg sync.WaitGroup
    wg.Add(10)

    slice := make([]int, 0)
    for i := 0; i < 10; i++ {
        go func(i int) {
            defer wg.Done()
            slice = append(slice, i)
        }(i)
    }

    wg.Wait()
    fmt.Println(len(slice)) // 10になるとは限らない
}
```

繰り返しますが、contextにゴールーチンセーフでない値を渡すべきではありません。
その部分を担保するのは、Goの言語仕様ではなくGoを利用するプログラマ側の責任です。

valueに与えるのがふさわしい値

それでは逆に、「contextに渡してやった方がいい値」というのはなんでしょうか。

渡すべきではない値の条件を全て避けようとすると、条件は以下のようになります。

1. 関数の挙動を変えうる引数となり得ない
2. type-unsafeを許容できる
3. 不変値
4. ゴールーチンセーフ

そして、contextというのは本来「異なるゴールーチン上で情報伝達するための機構」なのです。
これらの条件を鑑みると、自ずと使用用途は限られます。
それは「**リクエストスコープ**な値」であることです。

リクエストスコープとは？

リクエストスコープとは、「一つのリクエストが処理されている間に共有される」という性質のことです。
例を挙げると、

- ヘッダから抜き出したユーザーID
- 認証トークン
- トレースのためにサーバー側でつける処理ID
- etc...

です。これらの値は、一つのリクエストの間に変わることがなく、リクエストを捌くために使われる複数のゴールーチン間で共有されるべき値です。

contextの具体的な使用例

この章について

この章では、今まで紹介したcontextの機能をフルで使ったコードを書いてみたいと思います。

作るもの

今回は「httpサーバーもどき」を作ろうと思います。
要件は以下の通りです。

<機能要件>

- `go run main.go`で起動
- 起動後に標準入力された値「`path, token`」を基に、しかるべきハンドラにルーティング
- ルーティング後のハンドラにて、DBからのデータ取得→レスポンス作成の処理を行い、そのレスポンスの内容を標準出力に書き込む

<非機能要件>

- DBからのデータ取得が、2秒以内に終了しなければタイムアウトさせる

作成

エントリポイント(main)

まずは`go run main.go`でサーバーを起動させるエンドポイントである`main.go`を作っていきます。

```
package main

func main() {
    srv := server.DefaultServer
```

```
    srv.ListenAndServe()
}
```

やっていることは、

1. 自分で定義・設計したserver.DefaultServerを取得
2. サーバーを起動

です。ここではまだcontextが絡む様子は見当たりません。

サーバー(server)

それでは、エントリポイント中で起動しているサーバーの中身を見てみましょう。

リクエストの読み取り

```
package server

type MyServer struct {
    router map[string]handlers.MyHandleFunc
}

func (srv *MyServer) ListenAndServe() {
    for {
        var path, token string
        fmt.Scan(&path)
        fmt.Scan(&token)

        ctx := session.SetSessionID(context.Background())
        go srv.Request(ctx, path, token)
    }
}
```

ListenAndServeメソッドでは、for無限ループを回すことによって起動中にリクエストを受け取り続けます。
リクエストを受け取る処理は、以下のように実装されています。

1. 標準入力からリクエスト内容(パス、トークン)を読み込む
2. contextを作成し、それにトレースのための内部IDをつける
3. 別ゴールーチンを起動し、リクエストを処理させる

リクエストを処理させているのは、Requestメソッドです。次にこれの中身を見ていきましょう。

ルーティング

Requestメソッドの中身は、

1. ハンドラに渡すリクエスト構造体を作り、
2. リクエストスコープな値をcontextに詰めて
3. ルーティングする

というものです。

```
package server

func (srv *MyServer) Request(ctx context.Context, path string, token
string) {
    // リクエストオブジェクト作成
    var req handlers.MyRequest
    req.SetPath(path)

    // (key:authToken <=> value:token)をcontextに入れる
    ctx = auth.SetAuthToken(ctx, token)

    // ルーティング操作
    if handler, ok := srv.router[req.GetPath()]; ok {
        handler(ctx, req)
    } else {
        handlers.NotFoundHandler(ctx, req)
    }
}
```

最終的に、「ルーティング先が見つかったらhandlerを、見つからなければNotFoundHandlerを呼び出す」という操作に行きついています。
次に、呼び出されるハンドラの中の一つを見てみましょう。

ハンドラ(handlers)

handlersパッケージ内では、ハンドラを表す独自型としてMyHandleFuncというものを定義しました。
この型を満たす変数の一つとして、ハンドラGetGreetingを定義しました。

そしてその中で、

- トークン検証
- DBリクエスト(タイムアウトあり)
- レスポンス返却

までの処理を行わせています。

```
package handlers

type MyHandleFunc func(context.Context, MyRequest)

var GetGreeting MyHandleFunc = func(ctx context.Context, req MyRequest) {
    var res MyResponse

    // トークンからユーザー検証→ダメなら即return
    userID, err := auth.VerifyAuthToken(ctx)
    if err != nil {
        res = MyResponse{Code: 403, Err: err}
        fmt.Println(res)
```

```
        return
    }

    // DBリクエストをいつタイムアウトさせるかcontext経由で設定
    dbReqCtx, cancel := context.WithTimeout(ctx, 2*time.Second)

    //DBからデータ取得
    rcvChan := db.DefaultDB.Search(dbReqCtx, userID)
    data, ok := <-rcvChan
    cancel()

    // DBリクエストがタイムアウトしていたら408で返す
    if !ok {
        res = MyResponse{Code: 408, Err: errors.New("DB request timeout")}
        fmt.Println(res)
        return
    }

    // レスポンスの作成
    res = MyResponse{
        Code: 200,
        Body: fmt.Sprintf("From path %s, Hello! your ID is %d\ndata → %s",
req.path, userID, data),
    }

    // レスポンス内容を標準出力(=本物ならnet.Conn)に書き込み
    fmt.Println(res)
}
```

リクエストスコープな値の共有(session, auth)

この「httpサーバーもどき」で登場したリクエストスコープ値は2つありました。

- トレースのための内部ID(sesssion)
- 認証トークン(auth)

これらをcontext中に格納したり、逆にcontext中から読み出したりする関数を、別パッケージの形で提供しました。

```
package session

type ctxKey int

const (
    sessionID ctxKey = iota
)

var sequence int = 1

func SetSessionID(ctx context.Context) context.Context {
    idCtx := context.WithValue(ctx, sessionID, sequence)
    sequence += 1
```

```
    return idCtx
}

func GetSessionID(ctx context.Context) int {
    id := ctx.Value(sessionID).(int)
    return id
}
```

```
package auth

type ctxKey int

const (
    authToken ctxKey = iota
)

func SetAuthToken(ctx context.Context, token string) context.Context {
    return context.WithValue(ctx, authToken, token)
}

func getAuthToken(ctx context.Context) (string, error) {
    if token, ok := ctx.Value(authToken).(string); ok {
        return token, nil
    }
    return "", errors.New("cannot find auth token")
}

func VerifyAuthToken(ctx context.Context) (int, error) {
    // token取得
    token, err := getAuthToken(ctx)
    if err != nil {
        return 0, err
    }

    // token検証作業→userID取得
    userID := len(token)
    if userID < 3 {
        return 0, errors.New("forbidden")
    }

    return userID, nil
}
```

これらをわざわざ別パッケージに切り出したのは、利便性向上のためです。

例えば、今回は`auth`パッケージの中に入れた認証トークン周りの機能(=`SetAuthToken`,`VerifyAuthToken`関数)を`handlers`パッケージの中に入れてしまったとしましょう。
そして、そのトークン認証機能を、`handlers`とは別の`db`パッケージでも使いたい、という風になったとしましょう。

すると、

- `handlers` ← この中の認証トークン周りの機能を`db`パッケージで使いたい
- `db` ← この中のDBデータ取得機能を`handlers`パッケージで使いたい

という循環参照を引き起こしてしまう可能性があるのです。
そのため、「パッケージを超えてたくさんの場所で使いたい！」というcontext Valueは別パッケージに切り出すのが便利でしょう。

パッケージへのcontext導入について

この章について

さて、ここまでcontextで何ができるのか・どう便利なのかというところを見てきました。
そこで、「自分のパッケージにもcontextを入れたい」と思う方もいるかもしれません。

ここからは、「パッケージにcontextを導入する」にはどのようにしたらいいか、について考えていきたいと思います。

既存パッケージへのcontext導入

状況設定

例えば、すでに`mypkg`パッケージのv1として`MyFunc`関数があり、それを`main`関数内で呼び出しているとしましょう。

```
// mypkg pkg

type MyType sometype

func MyFunc(arg MyType) {
    // doSomething
}
```

```
// main pkg

func main() {
    // argを準備
    mypkg.MyFunc(arg)
}
```

この状況で、新たに「`MyFunc`関数にcontextを渡すようにしたい」という改修を考えます。

mypkg内の改修

NG例: contextを構造体の中に入れる

よくいわれるNG例は、「MyTypeの型定義を改修して、contextを内部に持つ構造体型にする」というものです。

```
-type MyType sometype
+type MyType struct {
+    sometype
+    ctx context.Context
+}

func MyFunc(arg MyType) {
    // doSomething
}
```

これがどうしてダメなのか、ということについて考えてみます。

contextのスコープが分かりにくい

例えばもしも、MyFunc関数の中でまた新たに別の関数AnotherFuncを呼んでいたらどうなるでしょうか。

```
func MyFunc(arg MyType) {
    // doSomething
    AnotherFunc(arg) // 別の関数を呼ぶ
}
```

よく見るとAnotherFuncの引数にargが使われています。
このarg構造体の中にはcontextが埋め込まれていました。そのため、AnotherFunc関数の中でもcontextが使える状態になります。
ですが「AnotherFunc関数の中でもcontextが使える」というのが、一目見ただけではわかりませんよね。

このように、contextを構造体の中に埋め込んで隠蔽してしまうと、「あるcontextがどこからどこまで使われているのか？」ということが特定しにくくなるのです。

contextの切り替えが難しい

また、MyType型にメソッドがあった場合には別のデメリットが発生します。

```
type MyType struct {
    sometype
    ctx context.Context
}

// メソッド1
func (*m MyType)MyMethod1() {
    // doSomething
}
```

```
// メソッド2
func (*m MyType)MyMethod2() {
    // doSomething
}
```

この場合に「メソッド1とメソッド2で違うcontextを渡したい」というときには、レシーバーである`MyType`型を別に作り直す必要が出てきます。
それはちょっと面倒ですよね。

OK例: MyFuncの第一引数にcontextを追加

これらの不便さを解消するには、contextは関数・メソッドの引数として明示的に渡す方法を取るべきです。

```
type MyType sometype

-func MyFunc(arg MyType) {
+func MyFunc(ctx context.Context, arg MyType)
 // doSomething
}
```

実際contextを関数の第一引数にする形では、contextのスコープ・切り替えの面でどうなるのかについてみてみましょう。

contextのスコープ

まずは、「`MyFunc`関数の中で別の関数`AnotherFunc`を呼んでいる」というパターンです。

```
func MyFunc(ctx context.Context, arg MyType) {
    AnotherFunc(arg)
    // or
    AnotherFunc(ctx, arg)
}
```

前者の呼び出し方なら「`AnotherFunc`内ではcontextは使っていない」、後者ならば「`AnotherFunc`でもcontextの内容が使われる」ということが簡単にわかります。

このような明示的なcontextの受け渡しは、contextのスコープをわかりやすくする効果があるのです。

contextの切り分け

また、`MyType`にメソッドが複数あった場合についてはどうでしょうか。

```
type MyType sometype

// メソッド1
func (*m MyType)MyMethod1(ctx context.Context) {
```

```
    // doSomething
}

// メソッド2
func (*m MyType)MyMethod2(ctx context.Context) {
    // doSomething
}
```

このように、contextをメソッドの引数として渡すようにすれば、「メソッド1とメソッド2で別のcontextを使わせたい」という場合では、引数に別のcontextを渡せばいいだけなので簡単です。
レシーバーである`MyType`を作り直すという手間は発生しません。

まとめ

> **Do not store Contexts inside a struct type; instead, pass a Context explicitly to each function that needs it.**
> The Context should be the first parameter, typically named ctx.
>
> (訳)**contextは構造体のフィールド内に持たせるのではなく、それを必要としている関数の引数として明示的に渡すべきです。**
> その場合、contextは`ctx`という名前の第一引数にするべきです。
>
> 出典:pkg.go.dev - context

mainパッケージ内の改修

さて、`MyFunc`関数の第一引数がcontextになったことで、`main`関数側での`MyFunc`呼び出し方も変更する必要があります。
`mypkg`パッケージ内でのcontext対応が終わっており、問題なく使える状態になっているなら、以下のように普通に`context.Background`で大元のcontextを作ればOKです。

```
func main() {
    ctx := context.Background()
    // argを準備
    mypkg.MyFunc(ctx, arg)
}
```

しかし、「`MyFunc`の第一引数がcontextにはなっているけれども、context対応が本当に終わっているか分からないなあ」というときにはどうしたらいいでしょうか。

NG例: nilを渡す

やってはいけないのは、「使われるかわからないcontextのところにはnilを入れておこう」というものです。

```
func main() {
    // argを準備
    mypkg.MyFunc(nil, arg)
}
```

これは中身がnilであるcontextのメソッドが万が一呼ばれてしまった場合、ランタイムパニックが起こってしまうからです。

```
var ctx context.Context

func main() {
    ctx = nil
    fmt.Println(ctx.Deadline())
}
```

```
$ go run main.go
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x488fe9]

goroutine 1 [running]:
main.main()
    /tmp/sandbox74431567/prog.go:12 +0x49
```

OK例: TODOを渡す

「`MyFunc`の第一引数がcontextにはなっているけれども、context対応が本当に終わっているか分からない」という場合に使うべきものが、contextパッケージ内には用意されています。
それが`context.TODO`です。

```
func main() {
+ ctx := context.TODO()
 // argを準備
-    mypkg.MyFunc(nil, arg)
+  mypkg.MyFunc(ctx, arg)
}
```

`TODO`は`Background`のように空のcontextを返す関数です。

```
func TODO() Context
```

出典:pkg.go.dev - context.TODO

> TODO returns a non-nil, empty Context.
> **Code should use context.TODO when it's unclear which Context to use or it is not yet available (because the surrounding function has not yet been extended to accept a Context parameter).**

> (訳)`TODO`はnilではない空contextを返します。
> **どのcontextを渡していいか定かではない場合や、その周辺の関数がcontext引数を受け付ける拡張が済んでおらず、まだcontextを渡せないという場合にはこの`TODO`を使うべきです。**
>
> 出典:pkg.go.dev - context.TODO

この`TODO`は「context対応中に、仮で使うためのcontext」という意図で作られているので、実際に本番環境に載せるときには残っているべきではありません。
本番デプロイ前には、然るべき機能を持つ別のcontextにすべて差し替えましょう。

標準パッケージにおけるcontext導入状況

さて、これで既存パッケージにcontextを導入する際には「contextを構造体フィールドに入れるのではなく、関数の第一引数として明示的に渡すべき」という原則を知りました。

contextパッケージがGoに導入されたのはバージョン1.7からです。
そのため、それ以前からあった標準パッケージはcontext対応を何かしらの形で行っています。

ここからは、二つの標準パッケージがどうcontextに対応させたのか、という具体例を見ていきましょう。

database/sqlの場合

`database/sql`パッケージは、まさに「contextを関数の第一引数の形で明示的に渡す」という方法を使ってcontext対応を行いました。

```
type DB
    func (db *DB) Exec(query string, args ...interface{}) (Result, error)
    func (db *DB) ExecContext(ctx context.Context, query string, args
...interface{}) (Result, error)

    func (db *DB) Ping() error
    func (db *DB) PingContext(ctx context.Context) error

    func (db *DB) Prepare(query string) (*Stmt, error)
    func (db *DB) PrepareContext(ctx context.Context, query string)
(*Stmt, error)

    func (db *DB) Query(query string, args ...interface{}) (*Rows, error)
    func (db *DB) QueryContext(ctx context.Context, query string, args
...interface{}) (*Rows, error)

    func (db *DB) QueryRow(query string, args ...interface{}) *Row
    func (db *DB) QueryRowContext(ctx context.Context, query string, args
...interface{}) *Row
```

出典:pkg.go.dev - database/sql

context導入以前に書かれたコードの後方互換性を保つために古いcontextなしの関数`Xxxx`も残しつつも、context対応した`XxxxContext`関数を新たに作ったのです。

net/httpの場合

`net/http`パッケージは、あえて「構造体の中にcontextを持たせる」というアンチパターンを採用しました。

例えば`http.Request`型の中には、非公開ではありますがctxフィールドが確認できます。

```
type Request struct {
    ctx context.Context
    // (以下略)
}
```

出典:net/http/request.go

なぜそのようなことをしたのでしょうか。実はこれも後方互換性の担保のためなのです。

`net/http`の中に、引数・返り値何らかの形で`Request`型が含まれている関数・メソッドの数は、公開されているものだけでも数十にのぼります。`http`パッケージ内部のみで使われている非公開関数・メソッドまで含めるとその数はかなりのものになるのは想像に難くないでしょう。

そのため、それらをすべて「contextを第一引数に持つように」改修するのは非現実的でした。
`database/sql`のように「後方互換性のために古い関数`Xxx`を残した上で、新しく`XxxContext`を作る」というのをやるのなら、それはもう新しく`httpcontext`というパッケージを作るようなものでしょう。並大抵の労力ではできません。

「非公開フィールドとしてcontextを追加する」という方法ならば、後方互換性を保ったcontext対応が比較的簡単に行えます。
そのため、`net/http`パッケージではあえてこのアンチパターンが採用されたのです。

Go公式ブログ - Contexts and structsでは`net/http`の例を取り上げて、「これが構造体の中にcontextを入れて許される唯一の例外パターンである」と述べています。

contextの内部実体

この章について

ここからは、ここまであえて言及してこなかった「context**インターフェース**」について触れていきたいと思います。

context「インターフェース」？

`context.Context`型の定義をよくよく見てみると、実はインターフェースじゃないか、というところに気付いていただけるかと思います。

```
type Context interface {
    Deadline() (deadline time.Time, ok bool)
```

```
    Done() <-chan struct{}
    Err() error
    Value(key interface{}) interface{}
}
```

出典:pkg.go.dev - context.Context

インターフェースということは、これを満たす具体型があるはずです。
ここからはその「contextになりうる具体型」を探しにいきましょう。

具体型一覧

contextパッケージの中には、`context.Context`インターフェースを満たす具体型が4つ存在します。

context.emptyCtx型

まず一つが、`context.emptyCtx`型です。

```
// An emptyCtx is never canceled, has no values, and has no deadline. It
is not
// struct{}, since vars of this type must have distinct addresses.
type emptyCtx int
```

出典:context/context.go

これは`context.Background`や`context.TODO`を呼んだときにできる空インターフェースを表現するために作られたものです。
キャンセルすることもできず、値やデッドラインを持ちません。

context.cancelCtx型

`context.cancelCtx`型は、内部にdoneチャネルをもち、キャンセル伝播を行うことができるcontextを表します。
また、`err`フィールドの中には、contextの`Err`メソッドで取得できるキャンセル理由のエラーが格納されます。

```
// A cancelCtx can be canceled. When canceled, it also cancels any
children
// that implement canceler.
type cancelCtx struct {
    Context

    mu       sync.Mutex            // protects following fields
    done     atomic.Value          // of chan struct{}, created lazily,
closed by first cancel call
    children map[canceler]struct{} // set to nil by the first cancel call
    err      error                 // set to non-nil by the first cancel
call
}
```

出典:context/context.go

context.timerCtx型

`context.timerCtx`は内部に`cancelCtx`を持った上で、タイムアウトのカウントをするためのタイマーも持ち合わせています。

```
// A timerCtx carries a timer and a deadline. It embeds a cancelCtx to
// implement Done and Err. It implements cancel by stopping its timer then
// delegating to cancelCtx.cancel.
type timerCtx struct {
    cancelCtx
    timer *time.Timer // Under cancelCtx.mu.

    deadline time.Time
}
```

出典:context/context.go

context.valueCtx型

`context.valueCtx`は、内部にkey-valueセットを持っています。
key,valフィールドにセットされた内容＋`valueCtx`内部に持っているContextが持っているkey-valueのセットが、`Value`メソッドで取ってこれる内容です。

```
// A valueCtx carries a key-value pair. It implements Value for that key
and
// delegates all other calls to the embedded Context.
type valueCtx struct {
    Context
    key, val interface{}
}
```

出典:context/context.go

具体型をまとめるインターフェースのメリット

このように、contextの機能である

- キャンセル伝播
- タイムアウト実装
- 値の伝達

は、実は全部違う型で実装されているのです。

これらの違う型をすべて「インターフェース」としてまとめて扱うために、contextはインターフェースとして公開されているのです。

```
// インターフェースがなかったら
func MyFuncWithCancel(ctx context.CancelCtx) // キャンセル機能があるcontextを受け取る場合
func MyFuncWithTimeout(ctx context.TimerCtx) // タイムアウト機能があるcontextを受け取る場合
func MyFuncWithValue(ctx context.ValueCtx) // 値伝達機能があるcontextを受け取る場合

↓

// インターフェースがあると
func MyFunc(ctx context.Context) // これで済む
```

おわりに

おわりに

というわけで、contextに関連する事項をまとめて紹介してきましたが、いかがでしたでしょうか。
contextは、`database/sql`や`net/http`のように現実の事象と対応している何かが存在するパッケージではないので、イマイチその存在意義や使い方がわかりにくいと思います。

そういう方々に対して、contextのわかりやすいユースケースや、使用の際の注意点なんかを伝えられていれば書いてよかったなと思います。

コメントによる編集リクエスト・情報提供等大歓迎です。
連絡先: 作者Twitter @saki_engineer

参考文献

書籍

書籍 Go言語による並行処理

https://learning.oreilly.com/library/view/go/9784873118468/

Goを書く人にはお馴染みの並行処理本です。
4.12節がまるまる`context`パッケージについての内容で、advancedな具体例をもとにcontextの有用性について記述しています。

書籍 Software Design 2021年1月号

https://gihyo.jp/magazine/SD/archive/2021/202101

Go特集の第4章の内容がcontextでした。
こちらについては、本記事ではあまり突っ込まなかった「キャンセル処理を行った後に、コンテキスト木がどのように変化するのか」などというcontextパッケージ内部の実装に関する話についても重点的に触れられています。

ハンズオン

ハンズオン 分かるゴールーチンとチャネル

https://github.com/gohandson/goroutine-ja

tenntennさんが作成されたハンズオンです。
STEP6にて、実際に`context.WithCancel`を使ってcontextを作ってキャンセル伝播させる、というところを体験することができます。

The Go Blog

Go Concurrency Patterns: Context

https://blog.golang.org/context

てっとり早くcontext4メソッドについて知りたいなら、このブログを読むのが一番早いです。
記事の後半部分ではGoogle検索エンジンもどきの実装を例に出して、contextが実際にどう使われるのかということをわかりやすく説明しています。

Contexts and structs

https://blog.golang.org/context-and-structs

「contextを構造体フィールドに入れるのではなく、関数の第一引数として明示的に渡すべき」ということに関して、1記事丸々使って論じています。

Go Concurrency Patterns: Pipelines and cancellation

https://blog.golang.org/pipelines

この記事の中にはcontextは登場しませんが、`Done`メソッドにおける「`chan struct{}`を使ってキャンセル伝播する」という方法の元ネタがここで登場しています。