go ctx & err

mutaguchi

アジェンダ

- テーマの選定理由
- Context
- Error
- おまけ

テーマの選定理由

(作図する)

- 聞いてくれている方にも
- 自身にとっても学びになる

Context

ネットワーク I/O が発生する箇所では必ず使う

```
ctx := context.Background()
client, err := spanner.NewClient(ctx, "projects/foo/instances/bar/databases/zoo")
```

Context とは何なのか？

- Go 製サーバーは各リクエストを独自ゴルーチンで処理する
- 一連のゴルーチンはリクエスト固有値にアクセスする必要ある
  - エンドユーザ ID／承認トークン／リクエスト期限など
- リクエストがキャンセル (デッドラインやタイムアウトを含む) した場合、動作しているすべてのゴルーチンはすぐに終了し、システムが使用中のリソースを再利用できるようにする必要がある

これらの解決手段として context パッケージが用意されている

cf. https://go.dev/blog/context

Context の責務

- キャンセルの伝達
- リクエスト固有値の伝達

cf. https://pkg.go.dev/context#pkg-overview

Context の責務

デッドラインもタイムアウトもキャンセルのラッパー関数なので、本スライドでは一括してキャンセル処理として扱う

```
func WithDeadline(parent Context, d time.Time) (Context, CancelFunc) {
        ( ... 略 ... )
        if cur, ok := parent.Deadline(); ok && cur.Before(d) {
                // The current deadline is already sooner than the new one.
                return WithCancel(parent)
        }
        ( ... 略 ... )
}
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc) {
        return WithDeadline(parent, time.Now().Add(timeout))
}
```

Context の責務ではない事

- 関数のオプショナル引数 (python でいうところのキーワード引数) ではない
- WithValue をオプショナル引数用途で使うと関数の実行に必要なシグネチャが分からなくなる

cf. https://pkg.go.dev/context#pkg-overview

Context の責務ではない事

- オプショナル引数が必要なら FOP (Functional Option Pattern) を検討する

```
options := []option.ClientOption{
        option.WithCredentialsFile("PATH_TO_CREDENTIALS_FILE"),
}
client, err := spanner.NewClient(ctx, dbName, options...)
```

FOP も乱用するとシグネチャが分からなくなるので、必須パラメータを FOP にしたりしない事

Context のインターフェース

```
type Context interface {
        // Done は、この Context がキャンセルされるか、タイムアウトしたときに閉じられるチャネルを返す
        Done() <-chan struct{}
        // Err は、Done チャンネルが閉じた後、このコンテキストがキャンセルされた理由を示す
        Err() error
        // Deadline は、この Context がキャンセルされる時刻（がもしあれば）を返す
        Deadline() (deadline time.Time, ok bool)
        // Value は、key に関連する値を返し、無い場合は nil を返す
        Value(key any) any
}
```

キャンセルされるとチャンネルを使った伝達が行われる

Context でキャンセルされた場合の処理フロー

- Goのcontext.Contextで学ぶ有向グラフと実装
  - 伝達は親から子へのみ行われる
    - 親がキャンセルされた → 子もキャンセルされる
    - 子がキャンセルされた → 親はキャンセルされない

Context の伝達に使われるチャンネル

紐解こうとすると `c10k問題` というネットワークプログラミング史における課題を知る必要がある

- 世界で最も普及した Apache サーバーが抱える問題

Context の伝達に使われるチャンネル

- クライアントが 10,000 台を超えるとプロセス数の上限に達する
  - Apache は 1 リクエスト 1 プロセス (Apache 方式)
    - 32bit Linux ではプロセス数上限が 32,767 であるため、それ以上リクエストを捌けなくなる
- コンテキストスイッチのコストが増大
  - 1 CPU が複数プロセスを並行処理するため、それまでの処理内容を保存して新しい処理の内容を復元すること
  - Apache 方式ではリクエスト増＝プロセス増であるため、コンテキストスイッチのコストが無視できなくなる

Context の伝達に使われるチャンネル

これらの問題はシングルプロセス・マルチスレッドにすればかなり軽減されるらしいが、それでもファイルディスクリプタ上限の問題が残る

cf. https://udon-yuya.hatenablog.com/entry/2020/09/03/233227

並行プログラミング

c10k 問題に限らず、有限リソースであるプロセス・スレッドを活用するため、モダンなプログラミング言語ではランタイムに吸収する仕組みが入っており、Go では goroutine と channel による `CSP ( Communicating Sequential Process ) モデル` が採用されている