iOSの通信処理

黒田 光

 # 概要

iOSにおけるHTTP通信の実行方法について説明する。
通信はURLSessionクラスを使用し、非同期処理の記述にはSwift Concurrencyを用いる。
非同期処理・・・メインスレッドから一時的に抜ける処理。

URLSession

URLSessionクラスを用いることで通信を行うことができる。このクラスでは、URLで示されるエンドポイントからデータをダウンロードしたり、アップロードするためのAPIを提供する。ダウンロードは、data関数を用いる。
data関数の通信結果はData型になっており、それを任意の型に変換し、アプリ上で表示を行う。

data関数

```
data(for: URLRequest, delegate: (URLSessionTaskDelegate)?) -> (Data, URLResponse)
```

指定されたリクエストに基づき、URLの内容をダウンロードし、非同期で値を返す。
(クロージャを用いたコールバック版, Publisherを返すCombine版があるが、今回は最も簡潔に書くことができるSwift Concurrency版を使用する)
コールバック・・・非同期処理の完了後に実行する処理
Publisher・・・時間の経過とともに値を送信する型

Swift Concurrency

Swift Concurrencyは、Swift5.5から登場した非同期処理・並列処理を簡潔に書くための機能である。従来はクロージャによるコールバックを用いていたが、Concurrencyの登場により非同期処理を同期的に記述できるようになった。

実装例

```
func fetchData() async -> Data? {
    let request = URLRequest(url: URL(string: "https://...")!)
    do {
        let (data, response) = try await URLSession.shared.data(for: request)
        return data
    } catch { return nil }
}

Task {
    let data = await fetchData()
}
```

async・・・停止させることができる関数の定義で使用する
await・・・停止する可能性がある関数の呼び出しで使用する
Task { }・・・同期的な関数の中でawaitするときに用いる

 # 画像を取得する
URLにdata関数でGETリクエストを送れば画像をData型として取得できる。
Data型からUIImage型への変換はUIImageのイニシャライザを用いる。

```
let image = UIImage(data: Data)
```

 # JSONを取得する
画像と同様にJSONを返すAPIにGETリクエストを送れば、JSONをData型として取得できる。
JSONは辞書型に変換できるが、Swift上で扱いづらいため、別の型に変換を行う。この変換作業のことをデコードと呼ぶ。

JSONデコード
まず、返ってくるJSONの構造に合わせて、Decodableに準拠したstructを定義する。
次にJSONDecoderを用いて、Data型からstructに変換を行う。

実装例

```
// JSON
{
  "id": "5445d5b3-7ae1-c833-469d-25a86ac33fc8",
  "items": [
    { "name": "item1", "score": 50 },
    { "name": "item2", "score": 80 },
    { "name": "item3", "score": 90 }
  ]
}

// 変換先の型
struct JSONResponse: Decodable {
    var id: String
    var items: Item

    struct Item: Decodable {
         var name: String
         var score: Int
    }
}

// デコードする
let response = try! JSONDecoder().decode(JSONResponse.self, from: data)
print(response.id) // 5445d5b3-7ae1-c833-469d-25a86ac33fc8
print(response.items[0].name) // item1
```

JSONが階層化している場合、別のstructを定義し、型に指定することで表現する。

JSONのキー名とstruct側の変数名が違っている場合や、struct側の変数がJSON側に存在しない場合はデコードに失敗するので注意が必要である。ただし、struct側の変数にはオプショナル型を使用することができ、その変数については、JSONに対象となる値が存在しなくても良い。

 # エラーハンドリング
アプリの挙動として、処理中にエラーが発生した場合はユーザーに伝達することが重要である。
(ここで言うエラーとは、通信の失敗やサーバーの不具合により適切な値が返って来ないなどの復帰可能な状態を指す。)
処理すべきエラーケースとしては、以下のようなものが挙げられる。
- 通信に失敗
- 通信に成功したがリクエストが拒否された
- リクエスト先が存在しない
- JSONのデコードに失敗

エラーハンドリングに際し、純粋なError型はエラーの内容が判別しづらいため、自前のError型を定義する。

実装例
```
enum CustomError: Error {
    case network
    case parse
}

func fetchData() throws {
  throw CustomError.network
}

do {
    try fetchData()
} catch {
    // 実際はエラー内容に応じたメッセージを画面に表示する
    print(error as! CustomError) // network
}
```

エラーをキャッチした場合、Error型から自前で定義した型に変換し、その内容に応じた内容を画面に表示することになる。また、enumでエラーを定義しておけば、switch文で全ケースを分岐させることが可能で扱いやすくなる。エラー内容の表示に関しては、UIAlertControllerやプレースホルダーで表示させると良い。

 # まとめ
- 通信はURLSessionクラスで行う
- 通信で取得した値はData型である
- Dataを適切な値・型に変換してアプリ上で表示を行う

 # リソース
https://github.com/1rsrx/ZemiCamp