わいわいswiftc #35

夢が広がる！コード生成でどこでもSwift

Twitter @iceman5499

既存コード生成技術の紹介

- ステンシルを書いてテンプレート出力する系
  - SwiftGen/SwiftGen
  - krzysztofzablocki/Sourcery
- 単一の高度な機能を提供する系
  - uber/mockolo
  - uber/needle

SwiftGen以外は全てSwiftSyntaxを用いている

SwiftSyntaxの課題

多くのコード生成ライブラリはSwiftSyntaxを利用しているが、Xcodeとバージョンを揃えて使う必要があって地味に大変

→ BetaなXcodeを使用しているなどで利用できない

→ Xcodeから実行する際に環境変数の指定が必要

使いやすさの問題

- stencilファイルの難しさ（SwiftGen, Sourcery）
  - 独自文法を勉強するのが大変
  - できない表現があったりして、代替案を頑張って模索する
    - regexが使えないなど: https://github.com/SwiftGen/StencilSwiftKit/pull/123
  - 魔術的なコードになりやすい

使いやすさの問題

- 自由度の課題
  - ライブラリが提供する表現力の範囲でしかコード生成できない
  - オプションで切り替えられる範囲にも限度がある

→ ライブラリが想定する使い方の範囲で強く効果を発揮する
→ 自分のプロジェクトのほうをライブラリの思想に合わせて設計する必要がある

BinarySwiftSyntax & SwiftTypeReader

- BinarySwiftSyntax
  - ローカルのXcode依存を回避
- SwiftTypeReader
  - コード生成器を自作しやすくする

作例紹介

CodableToTypeScript

CodableToTypeScript

- SwiftのCodableな型をTypeScriptの型に変換する

例1: シンプルなCodable

```
public struct Foo: Codable {
    public var bar: URL?
    public var baz: [String]
}
```

→

```
export type Foo = {
    bar?: string;
    baz: string[];
};
```

CodableToTypeScript

例2: 文字列enum

```
public enum Language: String, Codable {
    case ms
    case en
    case ja
}
```

→

```
export type Language = "ms" |
"en" |
"ja";
```

CodableToTypeScript

例3: 値付きenum

```
public enum FilterItem: Codable, Equatable {
    case name(String)
    case email(String)
}
```

→

- `~~~Decode` 関数も自動で生成される
- `kind` を追加することでswitchにおける網羅チェックとsmart castを有効にしている

```
export type FilterItemJSON = {
    name: {
        _0: string;
    };
} | {
    email: {
        _0: string;
    };
};

export type FilterItem = {
    kind: "name";
    name: {
        _0: string;
    };
} | {
    kind: "email";
    email: {
        _0: string;
    };
};

export function FilterItemDecode(json: FilterItemJSON): FilterItem {
    if ("name" in json) {
        return { "kind": "name", name: json.name };
    } else if ("email" in json) {
        return { "kind": "email", email: json.email };
    } else {
        throw new Error("unknown kind");
    }
}
```

CodableToTypeScript

使用例:

```
switch (filter.kind) {
case "name":
    const name = filter.name._0; // .nameをエラー無しに参照できる
    ...
case "email":
    const email = filter.email._0; // .emailをエラー無しに参照できる
    ...
}
```

CodableToTypeScript

- SwiftサーバとTypeScriptクライアントな環境において、Swift側の型定義を変更するだけでTS側もコンパイルエラーになってくれる💪
  - enumのcaseを型に表したり値付きenumが使えて便利
  - `.proto` や `.graphql` などの専用の定義ファイルは不要で、Swiftで書ける
- `[T]` を `T[]` に変換したり、 `T?` を `T|undefined` として変換できる
- （ある程度は）Genericsにも対応

使い方

- CodableToTypeScript単体はライブラリなので、自前でコード生成用ターゲットを作ってそこから使う

```
// Package.swift
.package(url: "https://github.com/omochi/CodableToTypeScript", branch: "main"),

...

.executableTarget(
    name: "CodeGenStage2",
    dependencies: [
        "CodableToTypeScript",
    ]
),
```

使い方

```
// main.swift
import SwiftTypeReader
import CodableToTypeScript

let module = try SwiftTypeReader.Reader().read(file: ...).module
let generate = CodableToTypeScript.CodeGenerator(typeMap: .default)
for swiftType in module.types {
    let tsCode = try generate(type: swiftType)
    _ = tsCode.description // TypeScriptコードそのままの文字列になっている
}
```

SwiftTypeReaderで読み取った型をCodableToTypeScriptに渡す

CallableKit

- https://github.com/sidepelican/CallableKit

CallableKit

- サーバ上のSwift関数をクライアントから実行するためのスタブを生成
- 定義ファイルから複数のソースを生成
  - サーバ用のルーティング用コード
  - クライアント用のリクエスト用コード
- 雰囲気はgRPCと同じ
  - gRPCよりはかなり薄くて、通信の詳細などは規定せずあくまでインターフェースを定義するだけ
- Swift Distributed Actorsのように、サーバ上のasync関数を呼び出せるようにする

CallableKit

例: 定義ファイル

```
public protocol EchoServiceProtocol {
    func hello(request: EchoHelloRequest) async throws -> EchoHelloResponse
}

public struct EchoHelloRequest: Codable, Sendable {
    public var name: String
}

public struct EchoHelloResponse: Codable, Sendable {
    public var message: String
}
```

CallableKit

例: サーバ用ルーティング実装（生成コード）

(VaporかつJSONでやりとりする場合)

```
import APIDefinition // 定義ファイルはそのままモジュールとしても利用する
import Vapor

struct EchoServiceProvider<RequestHandler: RawRequestHandler, Service: EchoServiceProtocol>: RouteCollection {
    var requestHandler: RequestHandler
    var serviceBuilder: (Request) -> Service
    init(handler: RequestHandler, builder: @escaping (Request) -> Service) {
        self.requestHandler = handler
        self.serviceBuilder = builder
    }

    func boot(routes: RoutesBuilder) throws {
        routes.group("Echo") { group in
            group.post("hello", use: requestHandler.makeHandler(serviceBuilder) { s in
                try await s.hello()
            })
        }
    }
}
```

- `RouteCollection` なので、Vaporの `RoutesBuilder` にそのままregisterできる

CallableKit

例: クライアント用スタブ実装（生成コード）

```
import APIDefinition

public struct EchoServiceStub: EchoServiceProtocol, Sendable {
    private let client: StubClientProtocol
    public init(client: StubClientProtocol) {
        self.client = client
    }

    public func hello(request: EchoHelloRequest) async throws -> EchoHelloResponse {
        return try await client.send(path: "Echo/hello")
    }
}
```

- 生成コードの役割は型をつけるだけなので、送信部分の実装詳細には関与していない

パッケージ構造

```
.
├── APIDefinition
│   └── Sources
│       └── APIDefinition // 定義だけで実装はなし
│           └── Echo.swift
├── APIServer
│   └── Sources
│       ├── Service // Serviceの具体的な実装。依存にサーバ用モジュールはなし
│       │   └── EchoService.swift
│       └── Server // Vaporに依存し、サーバを起動する
│           ├── EchoProvider.gen.swift
│           └── main.swift
├── ClientApp
│   └── Sources
│       └── APIClient
│           └── EchoStub.gen.swift
```

- クライアントは普通のasync関数を呼び出すかのようにAPIを叩ける

```
try await echoService.hello(request: .init(name: "Foo"))
```

- 現在はHTTPの通信にVaporを利用しているが、直接依存しているわけではないので将来的にVapor以外のフレームワークにも切り替えられる
- クライアントではただのprotocolとして見えているため、モック実装などへの差し替えが容易

Typescript版クライアント

- CodableToTypeScriptと組み合わせて、TypeScriptクライアントもコード生成

Typescript版クライアント

例: TS版クライアント用スタブ実装（生成コード）

```
import { IRawClient } from "./common.gen";

export interface IEchoClient {
  hello(request: EchoHelloRequest): Promise<EchoHelloResponse>
}

class EchoClient implements IEchoClient {
  rawClient: IRawClient;

  constructor(rawClient: IRawClient) {
    this.rawClient = rawClient;
  }

  async hello(request: EchoHelloRequest): Promise<EchoHelloResponse> {
    return await this.rawClient.fetch({}, "Echo/hello") as EchoHelloResponse
  }
}

export const buildEchoClient = (raw: IRawClient): IEchoClient => new EchoClient(raw);

export type EchoHelloRequest = {
    name: string;
};

export type EchoHelloResponse = {
    message: string;
};
```

- CodableToTypeScript
  - Swiftの型をTypeScriptの型に変換できる
- CallableKit
  - Swift protocolを任意の言語のinterfaceに変換できる

→ **WebAssembly × TypeScriptにも応用可能**

WasmCallableKit

- https://github.com/sidepelican/WasmCallableKit

WasmCallableKit

- WasmビルドされたSwift関数をTSから呼び出せる

例:

```swift
// WasmExports.swift
protocol WasmExports {
    static func hello(name: String) -> String
}
```

```swift
// main.swift
struct Foo: WasmExports {
    static func hello(name: String) -> String {
        "Hello, \(name) from Swift"
    }
}
WasmCallableKit.setFunctionList(Foo.functionList)
```

→

```typescript
export type FooExports = {
    hello: (name: string) => string,
};
```

```typescript
console.log(swift.hello("world"))
// > Hello, world from Swift
```

- もちろん、CodableToTypeScriptで変換できるSwiftの型なら何でもやりとりできる

```swift
protocol WasmExports {
    static func newGame() -> GameID
    static func putFence(game: GameID, position: FencePoint) throws
    static func movePawn(game: GameID, position: PawnPoint) throws
    static func aiNext(game: GameID) throws
    static func currentBoard(game: GameID) throws -> Board
    static func deleteGame(game: GameID)
}
```

↓

```typescript
export type WasmLibExports = {
  newGame: () => GameID,
  putFence: (game: GameID, position: FencePoint) => void,
  movePawn: (game: GameID, position: PawnPoint) => void,
  aiNext: (game: GameID) => void,
  currentBoard: (game: GameID) => Board,
  deleteGame: (game: GameID) => void,
};
```

WasmCallableKitの仕組み

- 文字列をやりとりできるように最低限のランタイムライブラリの用意
  - Wasmはそのままだと数値型しか直接やりとりできない
- SwiftTypeReaderとCodableToTypeScriptでTS用の型定義
- JS ⇔ Swift間で引数と返り値をJSON文字列としてやりとりする

tsランタイム: https://github.com/sidepelican/WasmCallableKit/blob/main/Codegen/Sources/Codegen/templates/SwiftRuntime.ts

swiftランタイム: https://github.com/sidepelican/WasmCallableKit/blob/main/Sources/WasmCallableKit/WasmCallableKit.swift

使用例

Swift Quoridor: https://swiftwasmquoridor.iceman5499.work

- Quoridor（コリドール）というボードゲームとそのAIをSwiftで実装
- UIだけReact
- リポジトリ: https://github.com/sidepelican/SwiftWasmQuoridor

JavaScriptKitとの比較？

- JavaScriptKitはSwiftからJS関数を呼び、SwiftがJSを利用する形になっている。これはReactのような、JSフレームワークからSwiftを利用したい場合に使いづらかった
- あとは単純にやってみたかった

課題

- 関数を呼び出すたびにJSON文字列との変換が入るのでめちゃくちゃ遅い
  - Reactの場合、1ビルド中に100回程度関数を呼び出すとそのオーバヘッドだけで遅延を体感できる
- シリアライズをより軽量な方法で行う、数値型はそのまま渡す、などの工夫が必要そう

ここまではブラウザからSwiftのWebAPIやWasmのSwift関数を利用していた。

JS上でSwiftを使いたい需要、他には・・・？

Cloud Functions for Firebase上でSwift関数を実行

Cloud Functions for Firebase上でSwift関数を実行

- サンプル: https://github.com/sidepelican/CFSwiftWasmExample

例:

```
export const hello = functions.https.onRequest(async (request, response) => {
  const name = request.query["name"] as string ?? "world";
  response.send(swift.hello(name));
});
```

Cloud Functions for Firebase上でSwift関数を実行

1. WASIのセットアップ
    - Cloud Functions上のNodeではWASIが利用できない（`--experimental-wasi-unstablre-preview0`を有効にする方法がない？）ので、`@wasmer/wasi`を使ってWASIを構築する

```
const wasi = new WASI();
```

2. 通常のWebAssembly利用時のボイラープレート通りにセットアップ

```
const swift = new SwiftRuntime();
const wasmPath = path.join(__dirname, 'Gen/MySwiftLib.wasm');
const module = new WebAssembly.Module(fs.readFileSync(wasmPath));
const instance = new WebAssembly.Instance(module, {
  ...wasi.getImports(module),
  ...swift.callableKitImpodrts,
});
swift.setInstance(instance);
wasi.start(instance);
return bindMySwiftLib(swift);
```

Cloud Functions for FirebaseでSwiftWasmを使うことは実用的か？

- Webと違い、バイナリサイズを（そこまで）気にしなくて良い
- NIOがないため、既存のサーバ用Swiftコードの多くが利用できない
  - NIOのWasm対応はかなり厳しいらしい
  - https://github.com/apple/swift-nio/pull/1404#issuecomment-587357512
  - AsyncHTTPClientなどの基本的なHTTPクライアントが利用できない
- Firebase Admin SDKのSwift版がないので、大変
- 用途はかなり限定されそう

おわり