# 第7章 状態管理とRiverpod

本章ではRiverpod[注1]というパッケージについて解説します。国内ではこのパッケージを中心にアプリを設計される例が多く、後述の状態管理パッケージとしてはスタンダードな存在です。本章ではFlutterアプリを開発するうえで重要な状態管理の考え方を解説したうえで、Riverpodの使い方を紹介します。

## 7.1 Flutterアプリにおける状態管理

Flutterアプリの設計やアーキテクチャの文脈で**状態管理**という言葉がよく使われます。広い意味ではStatefulWidgetも状態管理の一つと言えます。Stateクラスが状態を持ち、setStateメソッドで状態を更新すると、ウィジェットを再構築します。このように、**状態を更新したり、その更新を契機にウィジェットを再構築する戦略を状態管理**と呼びます。

では、StatefulWidgetだけですべての状態を管理することを考えてみましょう。次のような制約が発生します。

- **状態を更新するロジックはStateクラスを起点に実装しなければならない**
- **末端のウィジェットに状態を伝えるためには、コンストラクタの引数で状態をバケツリレーしなければならない**

その結果、コードの修正が困難になったり、パフォーマンスの問題が発生したりする可能性が高まります(パフォーマンスに問題が発生する理由については第10章で解説します)。

そこで多くの場合は状態管理のしくみというと、ウィジェットとロジックを分離させる設計が可能であったり、階層をまたいでウィジェットを再構築する機能を持っているものを指すことが多いです。

あらためて整理すると、以下のような特徴を持つパッケージを状態管理パッケージと呼びます。

- **状態を更新するロジックとウィジェットを分離した設計が可能となる**
- **状態を更新すると、階層の位置に関係なくウィジェットを再構築することができる**

注1 https://riverpod.dev/

# 7.2 Riverpodとはどのようなパッケージか

Riverpodの公式サイトは冒頭で「RiverpodはFlutter/Dartのリアクティブキャッシュフレームワークである」と紹介しています。これはRiverpodの状態管理パッケージとしての側面を表しています。状態変化に反応して(リアクティブ)ウィジェットを更新するしくみが提供されており、その状態を保持する(キャッシュ)という意味です。状態の保持(キャッシュ)に関してはウィジェットのライフサイクルに左右されず、破棄のタイミングをコントロールすることができます(またはウィジェットのライフサイクルと合わせることも可能です)。

また、状態管理パッケージとしての側面以外にも、クラスの依存関係を解決するためのしくみを提供しています。依存性の注入(*Dependency Injection*)、

**Tips 宣言的UIとしてのFlutter**

状態管理の重要性とともに、Flutterの大切な考え方を紹介します。

Flutterでは、UIの設計図はウィジェットの親子関係であり、`build`メソッドの実装そのものです。そして、それ以外の場所でUIを変更する手段はありません。これらの特徴から、Flutterは宣言的UIフレームワークの一つに数えられます。

宣言的UIフレームワークは従来までのUIフレームワークと違い、簡潔で保守性の高いコードを実現できます(しっかりと設計しなければこの限りではありませんが)。iOSやAndroidのネイティブアプリ開発においても、宣言的UIフレームワークが台頭しています。

この宣言的UIはしばしば以下のような式で表現されます。

```
UI = f(State)
```

これをFlutterに置き換えると、左辺の`UI`は画面の表示結果、右辺の`f`はウィジェットの`build`メソッドです。`build`メソッドの中で参照する情報を`State`(状態)ととらえてください。

ウィジェットを設計実装するときに、この式を意識してください。この式が成立しないウィジェットは設計に問題があるサインです。`f`は関数なので、副作用を持ちません。`f`に同じ`State`(状態)を与えると、必ず同じ`UI`になるということです。

たとえば、ウィジェットクラスが`final`以外のフィールドを持っている場合は、この式が成立しない可能性があります。

とりわけサービスロケーターパターンに近いものと筆者は認識しています。アプリの設計、アーキテクチャについて検討した経験のある読者の方は、これだけで強力なパッケージであることがおわかりいただけるかと思います。

## Riverpodの主要なクラス

Riverpodの主要なクラスを紹介します。

- **Provider**
- **RefとWidgetRef**
- **ConsumerWidget**

この3つです。それぞれの役割と関係性をざっくりと説明します。

Providerが状態をキャッシュし、`Ref`や`WidgetRef`を介して状態を提供します。`ConsumerWidget`はウィジェットのサブクラスです。`ConsumerWidget`の`build`メソッドの引数には`WidgetRef`が渡され、それを介して状態を取得しウィジェットを構築します。Providerの持つ状態が変化すると、`build`メソッドが再度呼び出されます。

また、Providerが提供するオブジェクトに大きな制限はない(Riverpod v2の時点ではジェネリックな型に制限があります)ので、扱い方によっては`Ref`を介してクラスの依存関係を解決できるというわけです。

## 実装サンプル

Riverpodの実装例をお見せします。前項の3つのクラスの役割や関係性がイメージできると思います。

以下の例がProviderを活用した最小の実装例です(厳密にはこれだけでは動作しません)。

```
final greetProvider = Provider((ref) {
  return 'Hello, Flutter!!';
});

class HomePage extends ConsumerWidget {
  const HomePage({super.key});

  @override
  Widget build(BuildContext context, WidgetRef ref) {
```

❶
❷

```
    final greet = ref.watch(greetProvider); ──❸
    return Center(
        child: Text(greet)
    );
  }
}
```

greetProviderはHello, Flutter!!という文字列を提供するProviderです(❶)。クロージャ式の引数refはProviderRefというRefの実装クラスです。他のProviderとやりとりする場合には、ここで受け取ったrefを使います。

greetProviderが提供する文字列をウィジェットで利用するにはConsumerWidgetクラスを継承します(❷)。ConsumerWidgetのbuildメソッドにはWidgetRef型の引数が与えられ、このWidgetRefを通してProviderとやりとりします(❸)。greetProviderから文字列を受け取り、Textウィジェットに渡しています。

次により実践的なサンプルを見てみましょう。Flutterのテンプレートプロジェクトを Riverpodを使ってアレンジします。

テンプレートプロジェクトはボタンをタップすると数字がカウントアップするアプリでした。これをRiverpodで実現するため、int型の数値を状態として提供し、かつその値を変更可能なProviderを実装します。

./lib/main.dart

```
// 省略
class CounterNotifier extends Notifier<int> { ──❶
  @override
  int build() => 0; ──❷

  void increment() {
    state = state + 1;
  }                          ──❸
}

final counterNotifierProvider = NotifierProvider<CounterNotifier, int>(() {
  return CounterNotifier();
});                                                                    ──❹
// 省略
```

状態を変更可能なProviderはNotifierというクラスを使います。カウンタの値を保持し、インクリメントするCounterNotifierというクラスを実装しました(❶)。初期値はbuildメソッドで返します(❷)。

Notifierクラスは自分の状態をstateプロパティに保持しています。❶で

Notifier<int>と渡している型パラメータが状態の型となります。incrementメソッドでは、そのstateプロパティにアクセスしてカウンタの値をインクリメントします(❸)。こうして実装したCounterNotifierクラスを、NotifierProviderで提供します(❹)。

続いてウィジェットの実装です。

./lib/main.dart

```
// 省略
class MyHomePage extends ConsumerWidget { ——❶
  const MyHomePage({super.key, required this.title});

  final String title;

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final counter = ref.watch(counterNotifierProvider); ——❷
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: Text(title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$counter', ——❸
              style: Theme.of(context).textTheme.headlineMedium,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          ref.read(counterNotifierProvider.notifier).increment(); ——❹
        },
        tooltip: 'Increment',
        child: const Icon(Icons.add),
      ),
    );
  }
}
// 省略
```

StatefulWidgetを継承したMyHomePageクラスを、ConsumerWidgetを継承するように書き換えました(❶)。カウンタの値をcounterNotifierProviderから取得し(❷)、Textウィジェットに渡しています(❸)。FloatingActionButtonウィジェットのタップイベントでは、CounterNotifierを取得しincrementメソッドを呼び出しています(❹)。

❷のようにWidgetRefのwatchメソッドを使って状態を監視した場合、状態が変化するとbuildメソッドが再び呼び出されます。ウィジェットからカウンタをインクリメントするロジックがなくなり、シンプルになりました(WidgetRefのwatchメソッドとreadメソッドを使い分けていますが、その詳細は「Providerから値を取得する」で解説します)。

# 7.3 Riverpodの関連パッケージ

Riverpodはいくつかのパッケージで構成されています。Riverpodの機能の全体像を把握する意味でも、概要を知っておきましょう。

- **基本機能を提供するパッケージ**
- **Providerのコードを生成するパッケージ**
- **静的解析を行うパッケージ**

以上の3つに分けられます。

## 基本機能を提供するパッケージ

まずはProviderクラスなどの基本機能を提供するパッケージです。Riverpodを利用するために必ず必要です。以下の3つの中から選択します。

- **riverpod**[注2]
- **flutter_riverpod**[注3]
- **hooks_riverpod**[注4]

注2 https://pub.dev/packages/riverpod
注3 https://pub.dev/packages/flutter_riverpod
注4 https://pub.dev/packages/hooks_riverpod

最も基本的なパッケージはriverpodであり、Flutterに依存しません。

flutter_riverpodはriverpodをウィジェットなどと連携し、Flutterアプリで利用するためのパッケージです。

hooks_riverpodはflutter_riverpodに加えて、flutter_hooks[注5]というパッケージと連携するためのパッケージです。flutter_hooksはウィジェットのライフサイクルに関連した実装を簡単に記述できるパッケージです。ReactのHooksをモチーフに実装されています。

通常、Flutterアプリ開発においてはflutter_riverpodを、flutter_hooksが必要な場合はhooks_riverpodを選択するとよいでしょう。

## Providerのコードを生成するパッケージ

Riverpodを中心にアプリを開発していると、Providerに関連するコードをボイラープレートのように繰り返し記述することになります。そのためProviderのコードを生成するためのパッケージを提供しています。これらのパッケージは必須ではありませんが、利用することが推奨されています。

- **riverpod_generator**[注6]
- **riverpod_annotation**[注7]

riverpod_generatorがコード生成を行うためのパッケージ、riverpod_annotationはコード生成のためのアノテーションを提供します。実際にはriverpod_generatorを利用するためにbuild_runnerパッケージも必要になります。

以下はコード生成を利用した場合と、利用しない場合の例です。

```
// コード生成を利用しない場合
final greetProvider = Provider((ref) {
  return 'Hello, Flutter!!';
});

// コード生成を利用する場合
@riverpod
String greet(GreetRef ref) {
  return 'Hello, Flutter!!';
};
```

注5 https://pub.dev/packages/flutter_hooks
注6 https://pub.dev/packages/riverpod_generator
注7 https://pub.dev/packages/riverpod_annotation