上記の例だけを見ると大きな変化はないように見えるかもしれませんが、コード生成を利用するとさまざまなメリットがあります。

・Providerに関するコードを記述する際の意思決定が減る
・Providerへ渡すパラメータの制限がなくなる
・Providerの変更がホットリロードできる

本章冒頭のサンプルコードはコード生成を利用しないものになっていますが、以降はコード生成を利用する前提で解説していきます。

静的解析を行うパッケージ

Riverpodのコード特有の問題を静的解析で検出、自動修正するためのパッケージが提供されています。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_lint[注8]

サードパーティパッケージが独自のLintルールを提供するためのツールとして、custom_lint[注9]というパッケージが提供されており、riverpod_lintはこのcustom_lintを利用しています。そのため、riverpod_lintを利用するためにはcustom_lintもインストールする必要があります。

riverpod_generatorでコード生成を行う際の記述ミスを検出するルールがいくつか用意されていますので、riverpod_generatorと併せて利用することをお勧めします。

関連パッケージまとめ

関連パッケージをいくつか紹介しました。結局、どれが必要でどれが不要なのか迷った方は`pubspec.yaml`を以下の内容にして開始するとよいでしょう。hooksパッケージは利用せず、コード生成と静的解析は利用する構成です。バージョンの指定は割愛しています。

注8　https://pub.dev/packages/riverpod_lint
注9　https://pub.dev/packages/custom_lint

./pubspec.yaml
```
dependencies:
  # 省略
  flutter_riverpod:
  riverpod_annotation:

dev_dependencies:
  # 省略
  riverpod_generator:
  build_runner:
  custom_lint:
  riverpod_lint:
```

コマンドラインから導入する場合は以下のコマンドを実行します。

```
$ flutter pub add flutter_riverpod riverpod_annotation
$ flutter pub add --dev riverpod_generator build_runner custom_lint riverpod_lint
```

7.4 Riverpodの使い方

それではRiverpodの使い方をサンプルコードと併せて解説していきます。

Providerの種類

コード生成を利用する前提ですと、状態を外部から「変更不可能」な関数ベースのProviderと、状態を外部から「変更可能」なクラスベースのProviderの2つに分類することができます。

関数ベースのProvider

まずは関数ベースのProviderの使い方を解説します。本章の冒頭で紹介したサンプルの`greetProvider`をコード生成を利用して実装します。

./lib/main.dart
```
part 'main.g.dart';

@riverpod
String greet(GreetRef ref) {
  return 'Hello World!!';
}
```

・@riverpodアノテーションを付与する
・第一引数にRef型のオブジェクトを受け取る

この2つのルールを守れば、関数ベースのProviderを実装できます。

@riverpodアノテーションを付与することで、コード生成の対象となります。生成したコードを参照するためにpart命令文を記述します。ファイル名は編集したファイル名に.g.dartを付与します。たとえば、main.dartにProviderを定義した場合はmain.g.dartとなります。生成されるProviderの名前は、関数名にProviderを付与したものになります。今回の例ではgreetProviderという名前になります。

第一引数にはRef型のオブジェクトを受け取ります。このRef型の名前はラージキャメルケースの関数名にRefを付与したものが生成されます。今回は関数名がgreetなので、GreetRefという名前になります。他に引数が必要な場合は第二引数以降に記述します。

関数の戻り値の型はProviderが提供する型になります。

実装が完了したらのコード生成のために、以下のコマンドを実行します。

```
$ flutter packages pub run build_runner build
```

クラスベースのProvider

続いてクラスベースのProviderの使い方を解説します。本章の冒頭で紹介したサンプルのCounterNotifierをコード生成を利用して実装します。

./lib/main.dart
```
// 省略

part 'main.g.dart';

@riverpod
class CounterNotifier extends _$CounterNotifier {
  @override
  int build() => 0;

  void increment() {
    state = state + 1;
  }
}
```

クラスベースのProviderを実装する際のルールは以下の3つです。

・@riverpodアノテーションを付与する
・_$＋クラス名の型を継承する
・初期値をbuildメソッドで返す

クラスベースの場合も@riverpodアノテーションを付与してコード生成の対象とします。part命令文が必要な点は関数ベースの場合と同じです。Notifierクラスは、_$＋クラス名の型を継承します。このクラスはコード生成によって作られます。今回の例では_$CounterNotifierというクラスになります。初期値はbuildメソッドで返します。Notifierクラスのstateプロパティには、このbuildメソッドの戻り値が設定されます。

実装が完了したらのコード生成のために、以下のコマンドを実行します。

```
$ flutter packages pub run build_runner build
```

Widgetの実装をもう一度見てみましょう。

./lib/main.dart

```
// 省略

class MyHomePage extends ConsumerWidget {
  const MyHomePage({super.key, required this.title});

  final String title;

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final counter = ref.watch(counterNotifierProvider); ❶
    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',
              style: Theme.of(context).textTheme.headlineMedium,
            ),
```

```
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          ref.read(counterNotifierProvider.notifier).increment(); ——❷
        },
        tooltip: 'Increment',
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

CounterNotifierの状態、すなわちカウンタの値を取得する場合はcounterNotifierProviderを監視します(❶)。CounterNotifierの状態を変更する場合は、counterNotifierProviderのnotifierプロパティをrefに渡し、incrementメソッドを呼び出します(❷)(ここでは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)。

非同期処理を行うProvider

Future型やStream型を提供するProviderについて解説します。

```
@riverpod
Future<String> asyncGreet(AsyncGreetRef ref) async {
  await Future.delayed(const Duration(seconds: 1));
  return 'Hello World';
}
```

戻り値をFuture型とし、asyncキーワードを付ける以外は先ほどの関数ベースのProviderと同じです。クラスベースのProviderの場合はbuildメソッドの戻り値をFuture型とし、asyncキーワードを付けるだけです。

```
@riverpod
class CounterNotifier extends _$CounterNotifier {
  @override
  Future<int> build() async {
    await Future.delayed(const Duration(seconds: 1));
    return 0;
  }
// 省略
}
```

このようにProviderを生成すると、Providerが提供する型がAsyncValueと

いう型になります。AsyncValue型はRiverpodが提供するクラスで非同期の値を安全に扱える便利クラスです。loading、error、dataの3つの状態を表現できます。非同期処理が実行中であったり、エラーが発生したりした場合など、状態に応じて場合分けできて便利です。

asyncGreetProviderを監視するウィジェットの実装例を見てみましょう。

```
class HomePage extends ConsumerWidget {
  const HomePage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final AsyncValue<String> greet = ref.watch(asyncGreetProvider); ❶
    return Center(
      child: greet.when(                                          ┐
          loading: () => const Text('Loading'),                   ├❷
          data: (greet) => Text(greet),                           │
          error: (e, st) => Text(e.toString())),                  ┘
    );
  }
}
```

❶でasyncGreetProviderの値を監視します。asyncGreetProviderの状態が変化するたびにHomePageウィジェットのbuildメソッドが呼び出されます。AsyncValueクラスのwhenメソッド[注10]を活用して、状態別のUIを構築しています(❷)。

クラスベースのProviderで、stateを更新する際にもAyncValue型を用います。先ほどのCounterNotifierクラスを例に、非同期処理中でなければ値をインクリメントするように実装してみましょう。

```
@riverpod
class CounterNotifier extends _$CounterNotifier {
  @override
  Future<int> build() async {
    await Future<void>.delayed(const Duration(seconds: 1));
    return 0;
  }

  void increment() async {
    final currentValue = state.valueOrNull;                       ┐
    if (currentValue == null) {                                   ├❶
      return;                                                     │
    }                                                             ┘
```

注10 今後はwhenメソッドではなく、Dart 3にて導入されたswitch式へ移行する方針が示されています。

```
    state = const AsyncLoading(); ——❷
    await Future<void>.delayed(const Duration(seconds: 1));
    state = AsyncValue.data(currentValue + 1); ——❸
  }
}
```

stateから現在の値を取得します(❶)。valueOrNullプロパティはAsyncValueクラスの値を取得するメソッドです。AsyncValueがdataの場合は値を取得できますが、loadingやerrorの場合は基本的にnullが返ります(意図的に前回値をキャッシュさせておく方法もあり、nullが返らないケースもあります)。ここでは、AsyncValueがdata以外の場合は何もせずに処理を終了します。続いてstateをAsyncLoadingに変更し(❷)、最後にstateをAsyncValue.dataに変更、値をインクリメントします(❸)。

非同期なProviderとRaw型

AsyncValueでラップされた非同期処理が扱いづらい場合もあります。たとえば、他のProviderの非同期処理の結果をもとに、データを処理するProviderを実装する場合です。

```
@riverpod
Future<int> fakeFirstApi(FakeFirstApiRef ref) async { ——❶
  await Future.delayed(const Duration(seconds: 1));
  return 1;
}

@riverpod
Future<int> fakeSecondApi(FakeSecondApiRef ref) async { ——❷
  await Future.delayed(const Duration(seconds: 1));
  return 2;
}

@riverpod
Future<int> fakeSumApi(FakeSumApiRef ref) async { ——❸
  final AsyncValue<int> firstApiResult = ref.watch(fakeFirstApiProvider);
  final AsyncValue<int> secondApiResult = ref.watch(fakeSecondApiProvider);
// 省略
}
```

❸のfakeSumApiは、❶と❷の結果を合算して返すProviderです。fakeFirstApiとfakeSecondApiはともに非同期処理に結果を返すProviderで、AsyncValue型で値が提供されます。このような場合、AsyncValue型をそのま

ま扱うと、コードが複雑になります。fakeFirstApiとfakeSecondApiの結果がFuture型であれば、awaitキーワードを使ってシンプルに実装できそうです。このような場合はProviderの提供する型をRaw型でラップします。

```
@riverpod
Raw<Future<int>> fakeFirstApi(FakeFirstApiRef ref) async { ❶
  await Future.delayed(const Duration(seconds: 1));
  return 1;
}

@riverpod
Raw<Future<int>> fakeSecondApi(FakeSecondApiRef ref) async { ❷
  await Future.delayed(const Duration(seconds: 1));
  return 2;
}

@riverpod
Future<int> fakeSumApi(FakeSumApiRef ref) async {
  final int firstApiResult = await ref.watch(fakeFirstApiProvider);
  final int secondApiResult = await ref.watch(fakeSecondApiProvider); ❸
  return firstApiResult + secondApiResult;
}
```

fakeFirstApiとfakeSecondApiの戻り値をRawでラップしました(❶、❷)。すると、fakeSumApiではawaitキーワードを使って結果をint型で受け取り、シンプルに実装できます(❸)。

Providerから値を取得する

Providerから値を取得するには、WidgetRefのwatchメソッドとreadメソッドを使います。watchメソッドは文字どおりProviderの値を監視します。ウィジェットのbuildメソッドで監視した場合には、Providerの値が変化するとウィジェットのbuildメソッドが再度呼び出されます。readメソッドはその時点でのProviderの値を取得するのみです。

Providerから値を取得する際は、可能な限りwatchメソッドを利用することが推奨されています。watchメソッドを利用することで、アプリ全体が状態変化に自動で反応し、メンテナンス性の高いアプリを実現できるとされています。

一方、値を監視する必要のないボタンのタップイベントや、Stateのライフサイクルイベントなどではreadメソッドを利用することが推奨されています。