ブロードキャスト

1つのStreamに対して複数回購読すると例外が発生します。複数の購読者へイベントを通知するには、asBroadcastStreamメソッドを使います。

```
import 'dart:async';

Stream<String> languages() async* {
  await Future.delayed(const Duration(milliseconds: 500));
  yield 'Dart';
  await Future.delayed(const Duration(milliseconds: 500));
  yield 'Kotlin';
  await Future.delayed(const Duration(milliseconds: 500));
  yield 'Swift';
  await Future.delayed(const Duration(milliseconds: 500));
  yield* Stream.fromIterable(['JavaScript', 'C++', 'Go']);
}

Future<void> main() async {
  final broadcastStream = languages().asBroadcastStream();

  await Future.delayed(const Duration(milliseconds: 1000)); ——❶

  broadcastStream.listen((i) { ——❷
    print('listener 1: $i');
  });

  await Future.delayed(const Duration(milliseconds: 1100));

  broadcastStream.listen((i) { ——❸
    print('listener 2: $i');
  });
}
// => listener 1: Dart
// => listener 1: Kotlin
// => listener 1: Swift
// => listener 2: Swift
// => listener 1: JavaScript
// => listener 2: JavaScript
// => listener 1: C++
// => listener 2: C++
// => listener 1: Go
// => listener 2: Go
```

ブロードキャストタイプのStreamは最初に購読されたタイミングで元のStreamの購読を開始します。そのため❶で待機している間はlanguages関数の本体は実行されません。❷で購読が開始されるとlanguages関数の本体が

実行され、yieldでイベントが通知されます。❸で2つ目の購読がスタートすると、そのタイミングからイベントが通知され、それまでの値は通知されません。

Streamを変更する

標準でStreamを変化させるメソッドが多く提供されています。ここですべてを紹介しませんが、代表的なものは、

- **Streamの値を変換するmap**
- **Streamの値をフィルタするwhere**
- **Streamの値の最大数を制限するtake**

などがあります。

```
import 'dart:async';

Stream<int> numberStream() {
  return Stream.fromIterable(List.generate(10, (index) => index));
}

void main() {
  numberStream()
      .where((num) => num % 2 == 0) // 0, 2, 4, 6, 8
      .map((num) => num * 2) // 0, 4, 8, 12, 16
      .take(3) // 0, 4, 8
      .listen((num) {
    print(num);
  });
}
// => 0
// => 4
// => 8
```

Zone —— 非同期処理のコンテキスト管理

DartにはZoneという非同期処理のコンテキストを管理するしくみがあります。その機能の一つに非同期処理で捕捉されなかった例外のハンドリングがあります。

以下はZoneを使わない場合の例です。

```
import 'dart:async';

// 戻り値がFuture型、例外をスローする関数
Future<String> fetchUserName() {
  var str =
      Future.delayed(const Duration(seconds: 1), () => throw 'User not found.');
  return str;
}

void main() {
  fetchUserName().then((data) {
    print(data);
  });
}
```

例外をスローする非同期処理です。Futureへ例外発生時のコールバックを登録していません。実行するとプログラムが強制終了します。

続いて、Zoneを使って例外をハンドリングする例です。

```
import 'dart:async';

// 戻り値がFuture型、例外をスローする関数
Future<String> fetchUserName() {
  var str =
      Future.delayed(const Duration(seconds: 1), () => throw 'User not found.');
  return str;
}

void main() {
  runZonedGuarded(() {
    fetchUserName().then((data) {
      print(data);
    });
  }, (error, stackTrace) {
    print('Caught: $error');
  });
}
// => Caught: User not found.
```

runZonedGuardedは第一引数に受け取った処理を自身のZoneで実行します。第二引数には自身のZoneで発生した例外をハンドリングするコールバックを渡します。実行するとプログラムが強制終了することなく、例外がハンドリングされます。ただし、Flutterのエラーハンドリングは Zoneではなく前述のPlatformDispatcherを使うことが一般的です。

実際はすべてのDartコードはZoneで実行されます（main関数は暗黙的にデ

フォルトZoneで実行されています)。Zoneにはエラーハンドリングのほかにもprint関数の動作を変更する機能や非同期コールバックの登録を捕捉する機能などがあります。

アイソレート

アイソレートはスレッドやプロセスのようなしくみで、

・専用のヒープメモリを持つ
・専用の単一のスレッドを持ち、イベントループを実行する
・アイソレート間でメモリの共有はできない

といった特徴があります。

すべてのDartプログラムはアイソレートの中で実行されます。通常、自動的にメインアイソレートが起動し、その中でプログラムが実行されるので意識することはありません。

Flutterアプリとアイソレート

Flutterアプリを作るうえでアイソレートを意識することはほとんどありません。前述のとおりメインアイソレートが自動的に起動し、その中でDartのプログラムが実行されます。

アプリでよく実装される時間のかかる処理として、HTTP通信やファイルのI/Oが挙げられます。これらはOSなどDartコード外で実行され(その間Dartのアイソレートは他のイベントを処理可能)、完了するとDartが再開されるためアプリがフリーズするようなことは起こりません。

アイソレートを活用するアプリのサンプルを以下に用意しました。Dartのプログラムで数値が素数かどうか判定する関数です。67280421310721という大きな素数を判定させるため計算に時間がかかります。ボタンをタップすると計算が開始します。

lib/main.dart
```
import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';

void main() {
  runApp(
    const MaterialApp(
```

```
      home: HomeScreen(),
    ),
  );
}

// 引数が素数かどうか判定する関数
bool isPrime(int value) {
  if (value == 1) {
    return false;
  }
  for (var i = 2; i < value; ++i) {
    if (value % i == 0) {
      return false;
    }
  }
  return true;
}

class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          crossAxisAlignment: CrossAxisAlignment.center,
          children: [
            const CircularProgressIndicator(),
            ElevatedButton(
                onPressed: () async {
                  const number = 67280421310721;
                  final result = await compute((number) { ❶
                    return isPrime(number);
                  }, number);
                  print('$number is prime: $result');
                },
                child: const Text('button')),
          ],
        ),
      ),
    );
  }
}
```

compute関数は新たにアイソレートを起動し、引数に渡した関数オブジェクトを実行します(❶)。筆者の環境では、compute関数に渡したisPrime(number);

をメインアイソレートで実行するとインジケータのアニメーションが止まり、Dartプログラムがブロックしたことが目視できました[注4]。

2.14 まとめ

Dartの言語仕様について解説しました。変数宣言、組み込み型、オペレータなど他の多くの言語と似たような文法が多く、基本的には習得しやすい言語だと思います。ただ、Dart 3で登場したパターンや、クラス修飾子など、全体像を把握するのが難しい機能も増えてきた印象です。

Dartは現在も活発に開発が進んでおり、これからも新しい機能が増えていくことでしょう。最新情報は公式の情報を参照してください。その前の基礎固めに本章がお役に立てばと思います。

注4 手もとで再現させたいと思った読者の方は注意しながら実行してください。実行環境のCPUを長時間占有する可能性があります。

第3章

フレームワークの中心となるWidgetの実装体験

StatelessWidget、StatefulWidget

本章ではFlutterアプリ開発の中心となる**ウィジェット**（**Widget**）について学びます。Flutterアプリをブラウザ上で実装、動作させることができるDartPadというツールがあります。まずはこのツールを利用して、Widgetを使った小規模なプログラムを書いてみましょう。

3.1 DartPadでアプリ開発を体験しよう

ブラウザでDartPadにアクセスします。

・https://dartpad.dev/

左側に以下のサンプルコードを入力し、「Run」ボタンをクリックしましょう。

```
import 'package:flutter/material.dart';

void main() {
  runApp(
    Container(
      color: Colors.blue,
      child: const Center(
        child: Text(
          'Hello, world!',
          textDirection: TextDirection.ltr,
        ),
      ),
    ),
  );
}
```

すると、ブラウザの右半分に、青い背景で「Hello, world!」と文字が表示されているかと思います（**図3.1**）。