第9章

フレームワークによるパフォーマンスの最適化

BuildContext、Key

ウィジェットのbuildメソッドの引数に渡されるBuildContextや、ウィジェットのコンストラクタに渡されるKeyについて、ここまで詳しい解説をしてきませんでした。本章ではいよいよBuildContextとKeyの役割を明らかにし、それがアプリのパフォーマンス最適化につながっていることを解説します。

9.1 BuildContextは何者なのか —— Element

ウィジェットのbuildメソッドの引数には必ずBuildContextが渡されます。このBuildContextは何者なのでしょうか。先に結論を言うと、**Element（エレメント）**というクラスです。

祖先の情報にアクセスできるBuildContext

通常のアプリ開発で使う場面は少ないですが、BuildContextには興味深いAPIが用意されています。

```
T? findAncestorWidgetOfExactType<T extends Widget>();
```

ウィジェットの親をたどり、ツリーの中で最も近い位置にあるT型のウィジェットを探して返却するメソッドです。計算量はO(n)[注1]です。実際に動作を確認してみましょう。

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

void main() {
  runApp(const MaterialApp(
    home: HomeScreen(),
  ));
}

class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key});

  @override
  Widget build(BuildContext context) {
```

注1　ウィジェットの階層が深くなると、計算時間が線形に増えていくことを意味します。

```
    final materialApp = context.findAncestorWidgetOfExactType<MaterialApp>(); ❶
    print(materialApp);
    // => MaterialApp
    return Scaffold(
      appBar: AppBar(
        title: const Text('Home Screen'),
      ),
      body: const Center(
        child: Text('Home Screen'),
      ),
    );
  }
}
```

❶で`findAncestorWidgetOfExactType`メソッドを呼び出し、親の`MaterialApp`ウィジェットが取得できることが確認できます。

ウィジェットは親や子にアクセスするAPIを持ちませんし、内部でもその情報は持っていません。しかし、`BuildContext`（Element）は親子関係をツリー構造で管理しているので、このようなAPIが実現できるのです。

ちなみに、似たAPIとして、直近の祖先のStateを取得するAPIがあります。

```
T? findAncestorStateOfType<T extends State<StatefulWidget>>()
```

`BuildContext`を引数に`NavigatorState`を取得する`Navigator.of`メソッドは、このAPIを使って実現されています。

Elementがツリーを構成していく工程

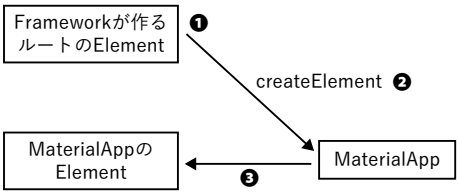
Flutterフレームワークの内部でElementがツリーを構成していく様子を図で表します。次のように`MaterialApp`ウィジェット、その子に`HomeScreen`というウィジェットがあるような状況を想定します。

```
./lib/main.dart
void main() {
  runApp(
    MaterialApp(
      home: HomeScreen()
    ),
  );
}
```

`main`関数では`runApp`関数が呼び出され、引数には`MaterialApp`ウィジェットが渡されます。このとき、`runApp`関数の内部では、ルートになるElement

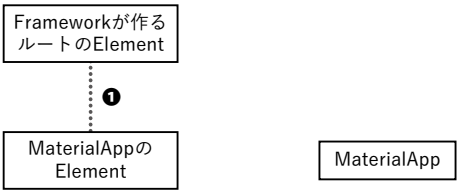
とウィジェットが生成されます(**図9.1**の❶)。ルートのElementは**MaterialApp**のElement生成を命令します(図9.1の❷、❸)。

図9.1 **MaterialAppのElementが生成される様子**



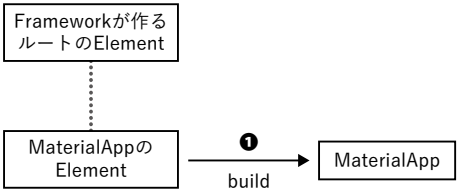
MaterialAppのElementがツリーの一部として構成されます(**図9.2**の❶)。

図9.2 **MaterialAppのElementがツリーの一部として構成される様子**



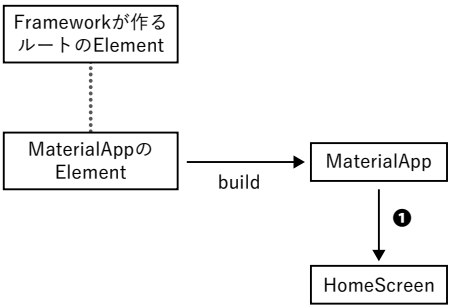
MaterialAppのElementが**MaterialApp**の**build**メソッドを呼び出します(**図9.3**の❶)。

図9.3 **MaterialAppのbuildメソッドが呼ばれる様子**



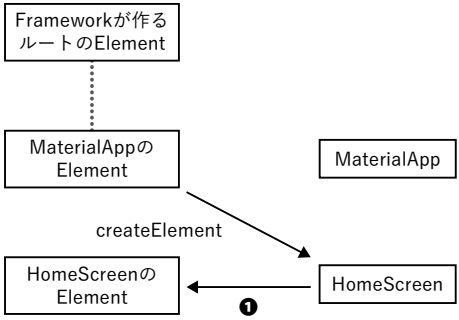
MaterialAppの**build**メソッドで、**HomeScreen**ウィジェットが返却されます(**図9.4**の❶)。

図9.4 **HomeScreenが生成される様子**



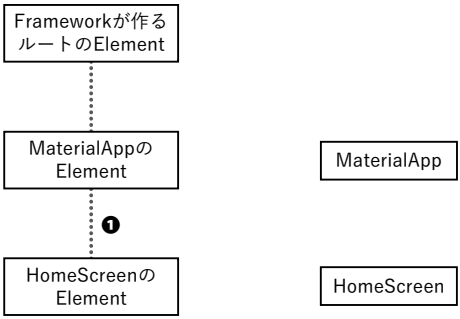
`MaterialApp`のElementが`HomeScreen`のElement生成を命令します（**図9.5**の❶）。

図9.5 **HomeScreenのElementが生成される様子**



`HomeScreen`のElementがツリーの一部として構成されます（**図9.6**の❶）。

図9.6 HomeScreenのElementがツリーの一部として構成される様子



以上を末端のウィジェットまで繰り返し、Elementのツリーを構成していきます。

StatefulWidgetの状態を保持する役割

次は別の視点からBuildContextを見てみましょう。StatefulWidgetのStateは、誰が管理しているのでしょうか？ライフサイクルはStatefulWidgetと同じでしょうか？

StatefulWidgetを入れ子構造にしたサンプルを用意しました。

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

void main() {
  runApp(
    MaterialApp(
      home: HomeScreen(),
    ),
  );
}

// HomeScreenはStatefulWidget
class HomeScreen extends StatefulWidget {
  HomeScreen({super.key}) {
    debugPrint('HomeScreen constructor');
  }

  @override
  State createState() => _HomeScreenState();
}
```

```
class _HomeScreenState extends State<HomeScreen> {
  int _counter = 0; ——❶

  @override
  Widget build(BuildContext context) {
    debugPrint('CounterButton build');
    return Scaffold(
      appBar: AppBar(
        title: const Text('Home Screen'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            // ボタンをタップするとカウントアップする
            ElevatedButton(
              child: Text('Home Screen Count: ($_counter)'), ——❷
              onPressed: () {
                setState(() {
                  _counter++; ——❸
                });
              },
            ),
            CounterButton(), ——❹
          ],
        ),
      ),
    );
  }
}

// CounterButtonはStatefulWidget
class CounterButton extends StatefulWidget {
  CounterButton({super.key}) {
    debugPrint('CounterButton constructor');
  }

  @override
  State createState() => _CounterButtonState();
}

class _CounterButtonState extends State<CounterButton> {
  int _counter = 0; ——❺

  @override
  Widget build(BuildContext context) {
```

```
    debugPrint('CounterButton build');
    // ボタンをタップするとカウントアップする
    return ElevatedButton(
      onPressed: () {
        setState(() {
          _counter++; ——❻
        });
      },
      child: Text('Counter Button Count: ($_counter)'), ——❼
    );
  }
}
```

HomeScreen画面はStatefulWidgetです。内部でカウンタを持っており(❶)、ボタンをタップするとカウントアップします(❷、❸)。さらに、CounterButtonというStatefulWidgetを並べました(❹)。CounterButtonウィジェットも同様にカウンタを持っており(❺)、ボタンをタップするとカウントアップします(❻、❼)。

アプリを実行し、それぞれのボタンをタップするとカウンタがインクリメントされ、期待どおりに動作します。しかし、不思議なところはないでしょうか？ HomeScreen画面のカウンタをインクリメントする、すなわちsetStateメソッドを呼び出すと、buildメソッドが呼ばれCounterButtonウィジェットが新しく作られるハズです。なのに、CounterButtonウィジェットのカウンタはリセットされずに状態を保持しています。

図9.7 「Home Screen Count」ボタンの押下前後