

よう。「ウィジェットの**build**メソッドが生成するウィジェット階層は、少ないほど効率が良い」とされ、Flutterの公式リファレンスに記載されています。

- **StatelessWidget class - Performance considerations**<sup>注1</sup>
- **StatefulWidget class - Performance considerations**<sup>注2</sup>

これを素直に受け取り、すべてのウィジェットの**build**メソッドで1つのウィジェットを生成するように実装すると、細かなウィジェットクラスが増えてしまい、保守性が低下します。

## 10.2

### 高速で保守性の高い実装

パフォーマンスにも寄与しながら、保守性の向上にもつながる実装を紹介します。なお、本章では解説に重きを置くため、関連したコードの断片を掲載している場合があります。省略されている部分がある点に留意してください。

#### buildメソッドで高コストな計算をしない

**build**メソッドは表示更新が必要なたびに繰り返し呼び出されます。この中でコストのかかる処理は避けるべきです。たとえば、巨大なリストから要素を検索するような処理です。画面遷移のために**NavigatorState**クラスを取得する**Navigator.of**メソッドはウィジェットの階層によっては計算量の大きな処理になり得ます。

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

  @override
  Widget build(BuildContext context) {
    // final navigator = Navigator.of(context); — ❶
    return Scaffold(
      appBar: AppBar(
        title: const Text('Home Screen'),
      ),
    );
```

注1 <https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html#performance-considerations>

注2 <https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html#performance-considerations>

```

body: Center(
  child: ElevatedButton(
    onPressed: () {
      final navigator = Navigator.of(context); —❷
      navigator.push(
        MaterialPageRoute(builder: (context) => const DetailScreen()));
    },
    child: const Text('Go to detail'),
  ),
),
);
}
}

```

**build**メソッドの中で❶のように**NavigatorState**を取得すると、**build**メソッドが実行されるたびに**NavigatorState**を検索することになり無駄が多いです。この場合は❷のようにボタンがタップされたときのみ**NavigatorState**を取得するのが好ましい実装と言えます。

また、次の例では**ListView**ウィジェットでデータをリスト表示していますが、表示するリストの要素を**build**メソッド内でフィルタリングしています。

```

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

  @override
  Widget build(BuildContext context) {
    // buildメソッド内で要素をフィルタリング
    final filteredItems = items.where((item) => /* itemのフィルタ条件 */).toList();
    return Scaffold(
      body: ListView.builder(
        itemCount: filteredItems.length,
        itemBuilder: (context, index) {
          return ListTile(
            title: Text(filteredItems[index]),
          );
        },
      ),
    );
  }
}

```

これでは、**build**メソッドが実行されるたびにリストのフィルタリングが行われてしまいます。あらかじめフィルタリングしたリストをウィジェットに渡すことで、**build**メソッドの実行コストを下げられるでしょう。**build**メソッドにロジックがなくなり可読性や再利用性も向上します。

```
class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key, required this.filteredItems});

  final filteredItems; // フィルタリング済みのリストを受け取る

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: ListView.builder(
        itemCount: filteredItems.length,
        itemBuilder: (context, index) {
          return ListTile(
            title: Text(filteredItems[index]),
          );
        },
      ),
    );
  }
}
```

## buildメソッドで大きなウィジェットツリーを構築しない

buildメソッドで構築するウィジェットツリーを小さくすることを意識しましょう。StatefulWidgetのsetStateメソッドの呼び出し、Providerの状態更新など、buildメソッドが繰り返し呼び出されるケースがあります。このときに再構築するウィジェットを少なくすることで、Elementの再利用判定などのコストを減らすことができます。

ウィジェットツリーを小さくといっても、常に1つのウィジェットだけを構築するところまで小さくすると逆に保守性に難が出てしまいます。たとえば、ウィジェットの選択を見なおすことで、ウィジェットツリーを小さくすることができます。

### ウィジェットツリーの階層が浅くなるようウィジェットの選択を見なおす

ウィジェットの選択を見なおすことで階層を浅くすることができます。ウィジェットを右下に配置したい場合、RowウィジェットとColumnウィジェットを組み合わせて構築することができます。

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

  @override
```

```

Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Home Screen'),
    ),
    /* ◆ Row
    子ウィジェットを水平方向に並べる */
    body: Row( // Row
      mainAxisAlignment: MainAxisAlignment.end, // 右寄せ
      children: [
        Column( // Column
          mainAxisAlignment: MainAxisAlignment.end, // 下寄せ
          children: [
            ElevatedButton(
              onPressed: () {
                Navigator.of(context).push(MaterialPageRoute(
                  builder: (context) => const DetailScreen()));
              },
              child: const Text('Go to detail'),
            ),
          ],
        ),
      ],
    ),
  );
}

```

同様のレイアウトをAlignウィジェット1つで実現できます。

```

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Home Screen'),
      ),
      /* ◆ Align
      alignmentパラメータに応じて子ウィジェットを配置するWidget */
      body: Align( // Align
        alignment: Alignment.bottomRight, // 右下寄せ
        child: ElevatedButton(
          onPressed: () {
            Navigator.of(context).push(MaterialPageRoute(
              builder: (context) => const DetailScreen()));
          },
        ),
      ),
    );
  }
}

```

```
    }),  
    child: const Text('Go to detail'),  
  ),  
),  
);  
}  
}
```

ウィジェットの階層が減り、可読性向上にもつながります。

## const修飾子を付与する

**const**修飾子を付与することで、ウィジェットがコンパイル時定数として扱われ、常に同じインスタンスが使われるようになります。そのため、**build**メソッドが実行されても**const**修飾子が付与されているウィジェットは再構築されません。

以下の例を見てみましょう。

```
class HomeScreen extends StatelessWidget {  
  const HomeScreen({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return Column(  
      children: [  
        A(  
          child: Text('A'),  
        ),  
        B(  
          child: Text('B'),  
        ),  
        const C(  
          child: Text('C'),  
        ),  
      ],  
    );  
  }  
}
```

**Column**ウィジェットの子ウィジェットに注目してください。**const**修飾子が付与されていないウィジェットとして**A**と**B**、**const**修飾子が付与されているウィジェットとして**C**があります。

この**build**メソッドが実行されるたびに**A**と**B**、さらにその子の**Text**ウィジェットも再構築されます。一方、**C**とその子である**Text**ウィジェットは再構

築されません。Dartの最適化により、`const`修飾子が付与されているCウィジェットは常に同じインスタンスが使われるためです。

では、このCウィジェット配下は表示更新をすることができないのでしょうか。そんなことはありません。ウィジェットが`StatefulWidget`であれば状態を更新することができますし、`InheritedWidget`の更新を購読することで表示更新を行うこともできます(第9章参照)。


`const`修飾子はそのウィジェット以下を更新不可にするのではなく、先祖の再構築の影響を受けない効果があると覚えておきましょう。

### const修飾子が使えるようウィジェットの選択を見なおす

Flutterが提供するウィジェットの中には`constant`コンストラクタを持たないものがあります。必要に応じて、`constant`コンストラクタを持つウィジェットに置き換えることを検討しましょう。

たとえば、背景色を指定可能なウィジェットとして`Container`ウィジェットと`ColoredBox`ウィジェットがあります。

```
ListView(  
  children: [  
    Container(  
      color: Colors.green,  
      child: const Text('Container'),  
    ),  
    const ColoredBox(  
      color: Colors.green,  
      child: Text('ColoredBox'),  
    ),  
  ],  
)
```



`Container`ウィジェットを使い背景色を指定した例(❶)と、`ColoredBox`ウィジェットを使い背景色を指定した例(❷)です。どちらも結果は同じですが、`Container`ウィジェットは`constant`コンストラクタを持たないため、この場合は`ColoredBox`ウィジェットを使用するのがよいでしょう。

### 独自のウィジェットクラスにconstantコンストラクタを実装する

独自のウィジェットクラスを実装する場合は、`constant`コンストラクタを実装しましょう。これには2つの効果があります。1つ目は、`const`修飾子を付与することで祖先の再構築の影響を受けなくなることです。2つ目は、`constant`コンストラクタを実装するためにウィジェットクラスをイミュータブル

ルにする必要があり、ウィジェットが状態を持たないようにすることです。  
ウィジェットクラスの堅牢性が高まります。

Flutter のテンプレートプロジェクトをベースにしたサンプルを用意しました。

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

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

class HomeScreen extends StatefulWidget {
  const HomeScreen({super.key});
  @override
  State<HomeScreen> createState() => _HomeScreenState();
}

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

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Home Screen'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            ColoredBox(
              color: Colors.blueGrey,
              child: Text(
                'You have pushed the button\nnthis many times:',
                style: Theme.of(context).textTheme.headlineSmall,
              ),
            ),
          ],
        ),
      ),
    );
  }
}
```

```

        Text(
          '$_counter',
          style: Theme.of(context).textTheme.headlineMedium,
        ),
      ],
    ),
  ),
  floatingActionButton: FloatingActionButton(
    onPressed: _incrementCounter,
    tooltip: 'Increment',
    child: const Icon(Icons.add),
  ),
);
}
}

```

Flutterのテンプレートプロジェクトをベースにしました。変更点は、❶で囲まれたColoredBoxウィジェットを追加したことです。文字列You have pushed the button\nthis many times:の表示部分に背景色をつけ、少しリッチな見た目にしてみました。

このサンプルは、FloatingActionButtonウィジェットをタップするとHomeScreen画面全体が再構築されます。続いて、ColoredBoxウィジェット以下を別のウィジェットクラスに分割してみましょう。

```

import 'package:flutter/material.dart';

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

class HomeScreen extends StatefulWidget {
  const HomeScreen({super.key});
  @override
  State<HomeScreen> createState() => _HomeScreenState();
}

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

```