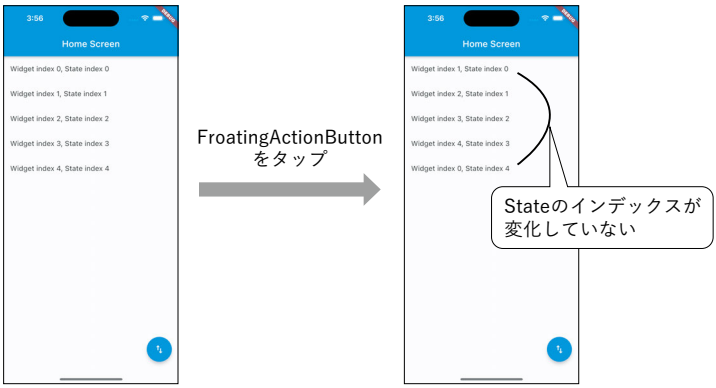
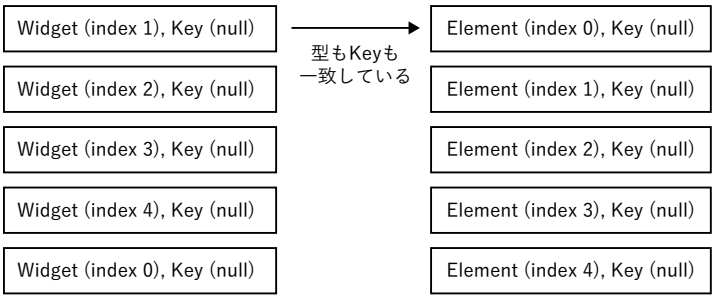
図9.11 **Stateのインデックスが変化しない様子**



ウィジェットのインデックスを表示するための情報はウィジェット（`ListItem`）が保持しています。ウィジェットは`build`メソッドが実行されるたびに作りなおされます。一方で、Stateのインデックスを表示するための情報はStateが保持しています。そして、StateはElementが参照を保持しているのでした。

Elementが再利用される条件に、**ウィジェットの型が同じかつKeyが同じ**というものがありました。今回は`ListItem`ウィジェットのインスタンスは同じですし、`Key`も特に指定していない（nullどうしを比較して一致した）ためElementは上から順番に再利用されたのです（**図9.12**）。

図9.12 **Elementが上から順に再利用されたイメージ図**



Keyを利用したElementの再利用

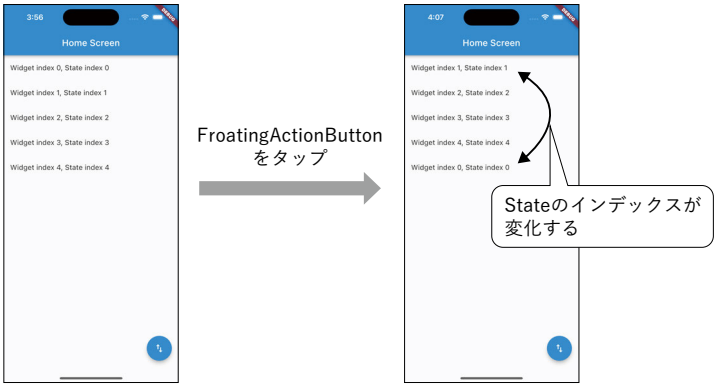
先ほどのサンプル、Stateのインデックスを並べ替えるには`Key`を利用しま

す。ListItemウィジェットのコンストラクタにKeyを渡します。

```
./lib/main.dart
// 省略
      body: Column(
        children: list.map((element) {
          return ListItem(
            key: ValueKey(element), ❶
            widgetIndex: element,
          );
        }).toList(),
      ),
// 省略
```

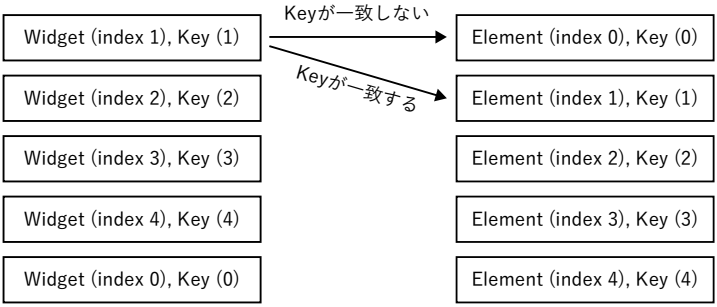
ValueKeyとは、コンストラクタ引数に与えられた値で識別するKeyです。これを実行すると、Stateのインデックスも並べ替えられることが確認できます(**図9.13**)。

図9.13 **Stateのインデックスが並び替えられる様子**



Elementのツリー構造からKeyが一致するElementを探し出します。このときKeyは❶に与えられたValueKeyです。ValueKeyはウィジェットのインデックスで一意に識別されます。このValueKeyが一致するようにElementを再利用するので、Stateのインデックスも一緒に並べ替えられるのです(**図9.14**)。

図9.14 **KeyによってElementが並び替えられるイメージ図**



以上のように、Elementの再利用により意図しないウィジェットと紐付いてしまうケースなどにKeyを利用します。

Keyの種類

Keyにはいくつか種類があります。**表9.1**に代表的なものを4つ紹介します。

表9.1 **代表的なKeyの種類**

Keyの種類	特徴
ValueKey	コンストラクタ引数に与えられた値で識別するKey
ObjectKey	コンストラクタ引数に与えられたオブジェクトで識別するKey
UniqueKey	インスタンスごとに一意に識別するKey
GlobalKey	ウィジェットの階層を越えて一意に識別するKey

前項の例ではint型のインデックスで一意に識別するため、ValueKeyを利用しました。ObjectKeyはコンストラクタ引数に与えられたオブジェクトで一意に識別します。引数を取らないUniqueKeyはインスタンスごとに一意に識別します。これら3つのKeyは同一階層のウィジェットから一致するKeyを探します。前項の例のようにリストの要素などを識別するのに利用します。

GlobalKeyはウィジェットの階層を越えて一意に識別します。異なる画面で特定のウィジェットの状態を一致させる場合や、アニメーションなどに利用します。

9.4 局所的にWidgetを更新するしくみ —— InheritedWidget

フレームワークが持つ最適化のしくみの一つとして、InheritedWidgetについても触れておきます。

InheritedWidgetは階層を越えてデータを渡すことのできるウィジェットです。身近な例ではThemeウィジェットが内部で生成している_InheritedThemeウィジェットがInheritedWidgetです。MaterialAppウィジェットの配下であれば、Theme.ofメソッドから階層のどこからでもThemeDataを取得できるのでした。

./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) {
    final theme = Theme.of(context); // ThemeDataを取得
    return Scaffold(
      backgroundColor: theme.backgroundColor,
      appBar: AppBar(
        title: const Text('Home Screen'),
      ),
      body: const Center(
        child: Text('Home Screen'),
      ),
    );
  }
}
```

先ほど、祖先のウィジェットを検索するBuildContextのAPIを紹介しました。

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

これに対して、祖先のInheritedWidgetを検索するAPIも提供されており、2つの特徴を持っています。

```
T? dependOnInheritedWidgetOfExactType<T extends InheritedWidget>();
```

まず1つ目は計算量です。InheritedWidgetを検索するAPIは、O(1)[注2]の計算量です。フレームワークが提供するウィジェットの多くはThemeDataを参照していますが、計算量が小さいので大きな問題にはなりません。

もう一つはInheritedWidgetの更新を購読する効果があることです。InheritedWidgetは、提供するデータが更新されたことを子孫に通知することができます。dependOnInheritedWidgetOfExactTypeを呼び出したウィジェットは、通知を受けるとbuildメソッドが呼び出されます。Themeウィジェットの例で言えば、ThemeDataが更新されるとThemeDataを参照するウィジェットが再構築され、新しいThemeDataに合わせてUIが更新されます。このしくみを用いると、StatelessWidgetであってもbuildメソッドが複数回呼ばれる可能性があります。

InheritedWidgetの機能をまとめると、

・InheritedWidgetはウィジェットの階層を越えてデータを提供することができる
・階層を越えてウィジェットの再構築をトリガすることができる

となります。第7章で解説した状態管理の概念と一致していますよね。InheritedWidgetはフレームワークが提供する状態管理のしくみの一つです。

Themeウィジェットのように上位に配置されるウィジェットがStatefulWidgetとして実装されていたとしたら、ThemeDataが更新されるとその配下すべてを再構築する必要があります。一方、InheritedWidgetは情報を必要とするウィジェットを局所的に再構築することができ、パフォーマンスへの影響を抑えることができるのです。

注2 ウィジェットの階層が深くなっても、計算時間が一定であることを意味します。

9.5 まとめ

Elementを起点に、フレームワークが内部で行っている最適化について解説しました。Elementはウィジェットよりもライフサイクルが長く（なる場合がある）、そのElementはRenderObjectを管理しています。このRenderObjectはレイアウト計算や描画といったコストの高い処理を行います。また、RenderObjectは状態を持ち、更新が不要な場合はスキップするしくみになっています。このRenderObjectを管理するElementの再利用は、このコストの高い処理をスキップする可能性をあげることにつながります。

また、Element再利用の条件には、Keyというクラスが密接に関わっていることを解説しました。Elementの再利用は、時として意図しない表示結果の原因につながることがあり、Keyを利用することで解決することができます。

最後に、フレームワークが提供する最適化のしくみの一つであるInheritedWidgetにも触れました。

本章ではフレームワークの最適化により、アプリのパフォーマンスが高められていることがわかりました。次章では私たちエンジニアがパフォーマンスを意識してどのようにコーディングすべきかを解説します。

第10章

高速で保守性の高いアプリを開発するためのコツ

第9章ではフレームワークが内部で行っているパフォーマンスの最適化を解説しました。本章も同じくパフォーマンスをテーマに、コードを書く際に考慮すべきポイントを紹介します。

10.1 パフォーマンスと保守性、どちらを優先すべきか

一般に、高速な(パフォーマンスを最優先した)実装と、保守性を意識した実装は、相反する場合があります。Flutterのパフォーマンスを最大限に引き出す実装は、時にソースコードの可読性や保守性を低下させます。

では、パフォーマンスと保守性はどのようなバランスでアプリを開発すればよいのでしょうか。基本的には保守性を第一に実装を進めるのが良いと筆者は考えます。前章で解説したとおり、FlutterにはパフォーマンスをElementの再利用など、最適化のしくみがあります。それらのしくみにより、パフォーマンスに深刻な問題が起こることは多くないからです。

上記の前提はありますが、パフォーマンスを意識した実装と、保守性を意識した実装が必ずしも両立しないわけではありません。本章では、パフォーマンスに寄与しつつも保守性が高まるような実装の考え方を紹介します。意識すべきポイントとして、常に頭の片隅に置いておいてください。

高速でないアプリとは

前項では「高速な」という言葉を使いましたが、本章で示す「高速な」アプリとは表示がカクカクしないアプリ、画面がフリーズしないアプリのことを指します。

Flutterは60fps(毎秒60フレーム)または対応デバイスでは120fpsで描画を行うことを目標にしています。毎秒60フレームの場合ですと1フレームあたり約16ミリ秒、この間に次のフレームの準備が整わなければ、表示がカクカクしたり、画面がフリーズしたように見えたりします。

高速だが保守性が低い実装

パフォーマンスを優先した結果、保守性が下がる実装の一例を紹介しまし