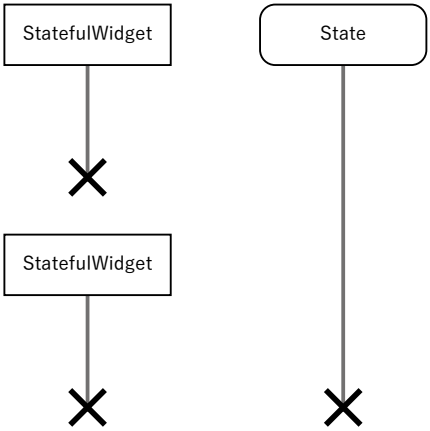
図9.7は「Home Screen Count」をタップする前後の様子です。「Home Screen Count」は2から3に変化しました。これは`HomeScreen`画面の`build`メソッドが呼ばれたことを意味します。しかし、「Counter Button Count」は1の状態を維持しています。この動きから、**StatefulWidgetのStateは、StatefulWidgetよりも長いライフサイクルを持っていることがわかります**(**図9.8**)。

図9.8 **StateのライフサイクルがWidgetのライフサイクルよりも長いイメージ図**



続いてStatefulWidgetが生成するElement(StatefulElement)のソースコードを一部見てみましょう。

```
./flutter/packages/flutter/lib/src/widgets/framework.dart
class StatefulElement extends ComponentElement {
  StatefulElement(StatefulWidget widget)
      : _state = widget.createState(),
        super(widget) {
```

このようにElementのコンストラクタでStatefulWidgetの`createState`メソッドを呼び出しています。また別のコードも見てみましょう。以下はElementが破棄されるときに呼ばれる`unmount`メソッドです。

```
./flutter/packages/flutter/lib/src/widgets/framework.dart
void unmount() {
    super.unmount();
    state.dispose();
```

```
    // 省略
    state._element = null;
    _state = null;
  }
```

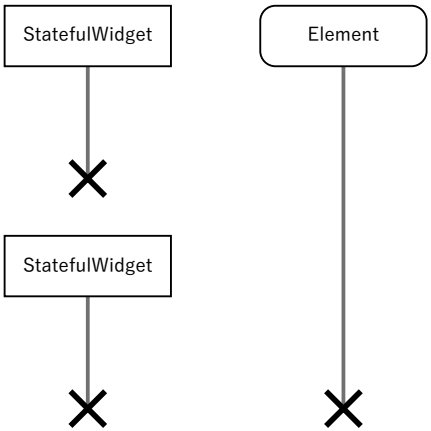
Elementが破棄されるときに、Stateクラスのdisposeメソッドが呼ばれています。これらのコードから**ElementとStateはライフサイクルが一致していることがわかります**。

これまでわかったことを振り返ってみます。

・StatefulWidgetのStateはStatefulWidgetよりもライフサイクルが長い
・StateはElementとライフサイクルが一致している

つまり、StatefulWidgetよりもElementのほうがライフサイクルが長いことになります(**図9.9**)。

図9.9 **ElementのライフサイクルがWidgetのライフサイクルよりも長いイメージ図**



また一方で、先ほどElementがツリーを構成していく過程を解説しました。StatefulWidgetよりもElementのほうがライフサイクルが長いことを念頭に、この過程を再度見てみます。

❶ElementからbuildメソッドがWidgetが呼ばれ、子ウィジェットのインスタンスが作られる
❷子ウィジェットから子Elementが作られ、ツリーに組み込まれる

❸子Elementから子buildメソッドが呼ばれ、孫ウィジェットのインスタンスが作られる

❹孫ウィジェットから孫Elementが作られ、ツリーに組み込まれる

StatefulWidgetで**setState**メソッドを呼び出した場合に当てはめてみましょう。ウィジェットの**build**メソッドが呼び出され、その中で新しい子ウィジェットのインスタンスが作られます(❶)。次の工程(❷)で子ウィジェットが子Elementを作ってしまうと、ウィジェットとElementのライフサイクルが同じになってしまい、辻褄が合いません。

実はフレームワークの内部で、Elementを再利用するしくみがあり、常に新しいElementを作るわけではないのです。これまで解説してきたことをあらためて整理します。

・StatefulWidgetよりもElementのほうがライフサイクルが長い

・Elementは再利用されるしくみがある

Tips 宣言的UIとElementの再利用

第7章で状態管理の解説をする際に、宣言的UIについて触れました。

```
UI = f(State)
```

右辺のfはウィジェットの**build**メソッドでした。そして、先ほどStatefulWidgetのStateを管理しているのはElementだということを解説しました。つまり、Flutterは次のような式ととらえることもできそうです。

```
UI = Widget.build(Element.state)
```

UIの設計図を提供するウィジェットと、それを実体化するための状態を持つElement、責務を分けることで宣言的UIを実現していると言えます。

9.2 Elementの再利用とパフォーマンス —— RenderObject

Elementの中には**RenderObjectElement**というクラスがあり、**RenderObject**というクラスを管理しています。この**RenderObject**はElementと同様に独自のツリー構造を持ちます。

RenderObjectは高コストな計算を行う

RenderObjectはウィジェットのレイアウト計算を行います。RenderObjectの親から子へ、サイズ制約を渡し、子のサイズが決まったら自身とのオフセット量を計算します。この操作をツリーの末端まで繰り返します。この処理はコストの高いものになります。

レイアウトが決定したのち、RenderObjectは描画処理を行います。RenderObjectは描画命令を発行しFlutterフレームワークよりも下層のFlutter Engineに対して描画を依頼します。この描画処理もまた、ツリーの末端まで繰り返すことになり、やはりコストの高いものになります。

RenderObjectは状態を持つ

RenderObjectは描画に必要な状態を保持します。色のついた矩形を表現するColoredBoxウィジェットを例にとってみましょう。ColoredBoxウィジェットは、colorというプロパティを持ちます。

以下は、ColoredBoxウィジェットの実装を簡略化したものです。

./flutter/packages/flutter/lib/src/widgets/basic.dart
```
class ColoredBox extends SingleChildRenderObjectWidget {
  const ColoredBox({ required this.color, super.child, super.key })

  final Color color;
```

ColoredBoxウィジェットは、RenderObjectWidgetを継承しており、_RenderColoredBoxというRenderObjectを生成します。_RenderColoredBoxもまた、colorというプロパティを持ちます。

以下は_RenderColoredBoxの実装を簡略化したものです。

./flutter/packages/flutter/lib/src/widgets/basic.dart
```
class _RenderColoredBox extends RenderProxyBoxWithHitTestBehavior {
  _RenderColoredBox({ required Color color })
    : _color = color;

  Color _color;
```

そして、_colorのカスタムセッタが重要な役割を果たします。

./flutter/packages/flutter/lib/src/widgets/basic.dart
```
  Color get color => _color;
```

```
set color(Color value) {
  if (_color == value)
    return;
  _color = value;
  markNeedsPaint();
}
```

新しいcolorが現在のcolorと一致する場合は何もせずに終了します。一致しない場合は、_colorに新しい値をセットし、markNeedsPaint()を呼び出します。このmarkNeedsPaint()は、次の描画タイミングで自身が再描画を行うようにフレームワークに予約するメソッドです。このようにRenderObjectは描画に必要な状態を保持し、コストの高い処理をスキップするかどうかの判断を行っているのです。

Elementの再利用はパフォーマンスに影響する

ここまでのRenderObjectについての解説をまとめます。

- **RenderObjectはElementによって管理されており、Elementの再利用はRenderObjectの再利用につながる**
- **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う**
- **RenderObjectはレイアウト計算や描画に必要な情報を保持しており、更新が不要な場合はスキップする**

つまりは、Elementの再利用はRenderObjectが行うコストが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。

9.3 Keyは何に使うのか

BuildContextに加えて、掘り下げてこなかったものにKeyクラスがあります。ウィジェットのコンストラクタ引数には、いつもKeyがありますよね。このKeyは何者で、何に使われるのでしょうか。

Elementが再利用される条件

先ほどElementは適宜再利用されると説明しました。このElementの再利用とKeyは密接な関係があります。ここでElementが再利用される条件を列挙します。

❶ウィジェットのインスタンスが同じ
❷ウィジェットの型が同じかつKeyが同じ
❸GlobalKeyが同じ

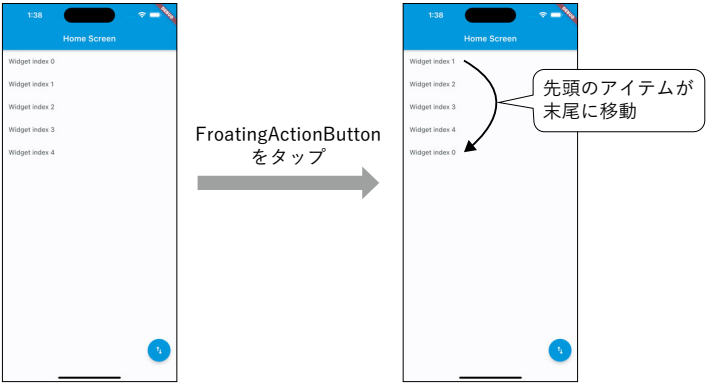
これだけだとイメージしづらいので、Keyを利用してElementを再利用する例を見てみましょう。

Elementが再利用される様子を見てみよう

先ほど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。この動作を確認するために、以下のようなサンプルを用意しました。再利用の様子を確認するために、少々強引なコードになっていますが、ご容赦ください。

5つの要素を並べたリストを並べ替えるサンプルです。FloatingActionButtonウィジェットをタップすると先頭の要素が末尾に移動します(**図9.10**)。

図9.10 **リストを並べ替えるサンプル**



./lib/main.dart

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

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

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

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

class _HomeScreenState extends State<HomeScreen> {
  final list = List.generate(5, (index) => index); ❶

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Home Screen'),
      ),
      body: Column(
        children: list.map((element) {
          return ListItem(
            widgetIndex: element, ❷
          );
        }).toList(),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          setState(() {
            debugPrint('Swap first and last element');
            final value = list.removeAt(0); ❸
            list.add(value);
          });
        },
        child: const Icon(Icons.swap_vert),
      ),
    );
  }
}
```

```
class ListItem extends StatefulWidget {
  const ListItem({super.key, required this.widgetIndex});

  final int widgetIndex; ——❹

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

class _ListItemState extends State<ListItem> {
  static int counter = 0;

  final int _stateIndex = counter++; ——❺

  @override
  Widget build(BuildContext context) {
    return ListTile(
      title: Text(
        'Widget index ${widget.widgetIndex}, ' ——❻
            'State index $_stateIndex', ——❼
      ),
    );
  }
}
```

❶では、0から4までの整数を要素とするリストを作成しています。リストの要素を❷で並べて表示しています。このリストを❸で並べ替えています。FloadingActionButtonウィジェットをタップすると、リストの先頭の要素を末尾に移動させます。この操作はsetState引数の中で行っているので表示は更新されます。

リストの要素は独自に実装したListItemウィジェットです。ListItemウィジェットはwidgetIndexというプロパティ(❹)を持ち、❶のインデックスが渡ります。また、State(_ListItemStateクラス)は_stateIndexというプロパティを持ち(❺)、こちらはStateのインスタンスが作られた順にインデックスを保持します。それぞれ❻と❼でウィジェットのインデックス、Stateのインデックスとして表示しています。

このサンプルを実行しFloatingActionButtonウィジェットをタップすると、ウィジェットのインデックスは変化しますがStateのインデックスは変化しません(**図9.11**)。