```
class _MyAppState extends State<MyApp> {
  bool _isDarkMode = false;

  void _toggleDarkMode() {
    setState(() {
      _isDarkMode = !_isDarkMode;
    });
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        colorSchemeSeed: Colors.green,
        extensions: const [MyTheme(themeColor: Color(0xFF0000FF))],
      ),
      darkTheme: ThemeData(
        colorSchemeSeed: Colors.green,
        brightness: Brightness.dark,
        extensions: const [MyTheme(themeColor: Color(0xFFFF0000))],
      ),
      themeMode: _isDarkMode ? ThemeMode.dark : ThemeMode.light,
      home: Scaffold(
        body: const Center(
          child: ThemedWidget(),
        ),

        floatingActionButton: FloatingActionButton(
          onPressed: () {
            _toggleDarkMode();
          },
          child: const Icon(Icons.settings_brightness),
        ),
      ),
    );
  }
}

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

  @override
  Widget build(BuildContext context) {
    final myTheme = Theme.of(context).extension<MyTheme>()!;
    final color = myTheme.themeColor;
```

```
    return Container(width: 100, height: 100, color: color);
  }
}
```

Tips Cupertino（クパチーノ）デザイン

ここまでMaterialAppウィジェットを使い、マテリアルデザインをベースにした機能を紹介してきました。Flutterはこれ以外にも、iOSのルック＆フィールを再現したウィジェットがあります。テーマの扱いに関してはMaterialAppウィジェットの代わりにCupertinoAppウィジェットを使い、CupertinoThemeDataクラスでアプリのテーマを管理します。

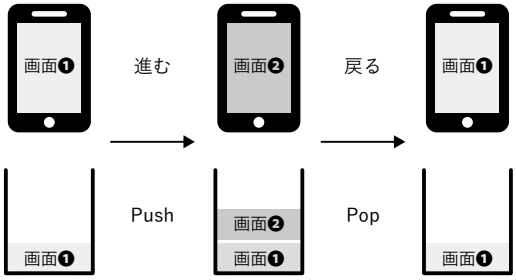
iOSのルック＆フィールを再現したウィジェットの中には、CupertinoAppウィジェットが先祖になければ利用できないものもありますので注意してください。同様に、マテリアルデザインのウィジェットも先祖にMaterialAppウィジェットがあり、ThemeDataクラスが提供されていないと利用できないものがあります。

Flutter公式では、iOS以外のプラットフォームにもアプリを提供する場合はマテリアルデザインを採用することを推奨しています。なお、本書は引き続きマテリアルデザインをベースに解説を進めていきます。

5.2 ナビゲーションとルーティング
—— 画面遷移を実現する3つの手法

Flutterの画面遷移は、画面の履歴をスタック構造のコンテナで管理することで実現しています。スタックへプッシュ（追加）することで新たな画面に遷移し、ポップ（取り除く）ことで元の画面に戻ります（**図5.4**）。

図5.4 **画面スタックのイメージ**



Flutterの提供する画面遷移の全体像を理解するうえで、重要な歴史的背景があります。2020年、FlutterはWebアプリをサポートする際にブラウザと連携する新たなAPIを提供しました。新しいAPI群はNavigator 2.0と呼ばれています(便宜上、Navigator 2.0より前から存在するAPI群をNavigator 1.0と呼びます)。これらのAPIは置き換えではなく追加の形で提供されました。そのため理解を難しくしている以下のようなポイントがあると筆者は感じています。

- **Navigator 1.0から提供されている一部のクラスへ、Navigator 2.0の機能が追加されている**
- **Navigator 1.0とNavigator 2.0とでネーミングに統一感がない**
- **Navigator 1.0とNavigator 2.0は併用が可能**

続いて、**表5.1**で用語を整理しましょう。

表5.1 **用語の一覧**

用語	意味
Navigatorウィジェット	スタックを管理するウィジェット。Navigator 1.0から存在する
Routeクラス	スタックで管理される画面の単位。Navigator 1.0から存在する
Routerウィジェット	プラットフォームと連携した画面遷移を実現する中心的なウィジェット。Navigator 2.0で追加された
Pageクラス	Routeクラスを生成する軽量なオブジェクト。Navigator 2.0で追加された

Flutterの画面スタックを管理しているのは`Navigator`ウィジェットです。Flutterではスタックで管理される画面の単位は`Route`クラスです。画面スタックへ`Route`クラスのインスタンスをプッシュ／ポップすることで画面遷移を実現します。この方法はNavigator 1.0から提供されています。

`Router`ウィジェットはNavigator 2.0で提供され、プラットフォームと連携

した画面遷移を実現する中心的なウィジェットです。プラットフォームとの連携とはブラウザの「進む」「戻る」ボタンを契機にした画面遷移、アプリ内の画面遷移とアドレスバーのURLとの連動などを指します。

また、Navigator 1.0のAPIでは画面を1つずつプッシュ／ポップして履歴を積み上げていくのに対して、Navigator 2.0のAPIでは画面履歴を一度に書き換えてしまうことが可能です(**図5.5**)。たとえば、設定画面からユーザー情報画面に遷移するアプリで、ユーザー情報画面をブックマークしていたとしましょう。ブラウザでブックマークからユーザー情報画面を開いたとき、ブラウザの「戻る」ボタンを押すと設定画面が現れるといったことを実現できるのがNavigator 2.0のAPIです。

このとき、`Route`クラスのインスタンスを複数生成するのではなく、軽量な`Page`クラスが用いられます。`Page`クラスには自身を一意に判定する`key`プロパティがあり、画面履歴書き換え時の最適化に利用されます。この最適化はウィジェットとエレメント(Element)の関係に似ています(ウィジェットとエレメント(Element)の関係は第9章で解説します)。

iOS/Androidをターゲットとしたモバイルアプリの場合、Navigator 1.0のAPIで十分なケースも多いです。また、Navigator 2.0を利用した実装は複雑で、ラップされたライブラリを利用するのがよいでしょう。

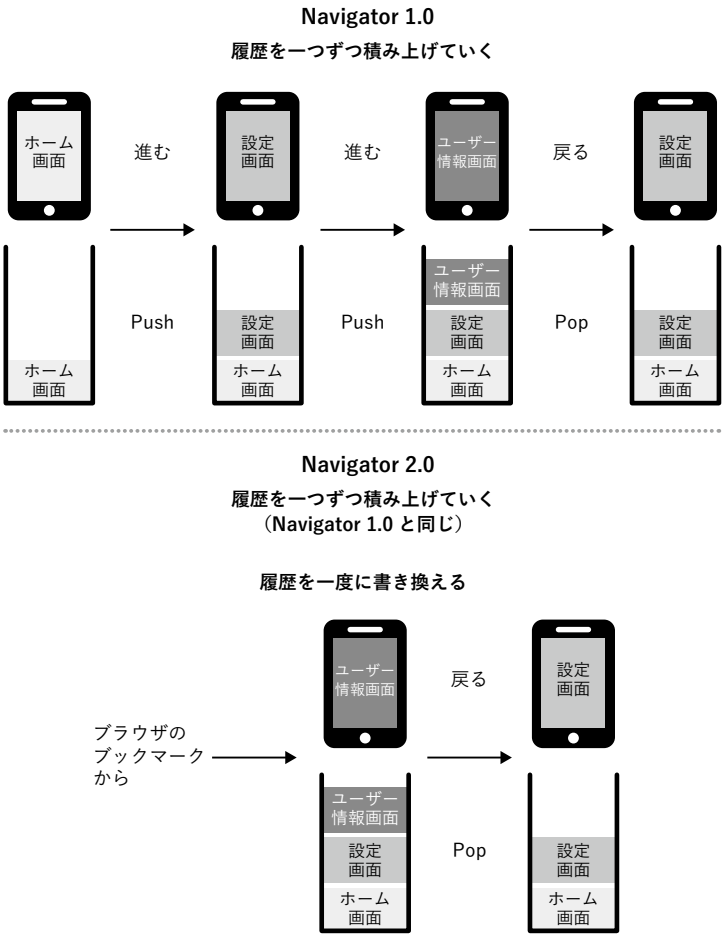
NavigatorウィジェットとRouteクラスによる画面遷移
—— Navigator 1.0

それでは`Navigator`ウィジェットと`Route`クラスを利用したNavigator 1.0の画面遷移を見てみましょう。繰り返しになりますが、Flutterではスタックで管理される画面の1単位を`Route`クラスで表現します。`Navigator`ウィジェットに対して、`Route`クラスのインスタンスをプッシュすることで新しい画面に遷移し、ポップすることで元の画面に戻ります。通常、`Navigator`ウィジェットは`MaterialApp`ウィジェットが内部でインスタンス化したものを利用します。

簡単なサンプルを見てみましょう(理解を優先して、コードは簡略化しています)。

```
class FirstScreen extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    /* ◆ ElevatedButton
    マテリアルデザインのボタン */
```

図5.5 **Navigator 1.0 と Navigator 2.0 の違い**




```
return ElevatedButton(
  child: const Text('次へ'),
  onPressed: () {
    final navigatorState = Navigator.of(context); ❶
    /* ◆ MaterialPageRoute
    実行するプラットフォームに適した画面遷移アニメーションを提供するRoute */
    final route = MaterialPageRoute(
      builder: (context) => const SecondScreen(), ❸
    );                                              ❷
    navigatorState.push(route); ❹
  },
);
```

```
  }

}
```

FirstScreen画面のボタンをタップすると、SecondScreen画面に遷移するサンプルです。ボタンをタップすると、Navigatorウィジェットの静的メソッドofからNavigatorStateクラスのインスタンスを取り出します(❶)。

次にRouteクラスのインスタンスを生成します(❷)。MaterialPageRouteクラスはプラットフォームに合わせて画面遷移のアニメーションを提供してくれ(iOSであれば右から左へスライドイン、Androidであればズームイン)、ほとんどの場合はこれを使用します。**クラス名に「Page」とありますが、Navigator 2.0のPageクラスではないので注意**しましょう。

❸の引数builderへは、遷移先のウィジェットを生成する関数型を渡します。こうして生成したRouteクラスをNavigatorStateクラスへプッシュすることで画面遷移します(❹)。

このサンプルでは理解しやすいように、NavigatorStateを変数に格納していますが、実際には、

```
Navigator.of(context).push(
  MaterialPageRoute(
    builder: (_) => const SecondScreen(),
  ),
);
```

と1つの文で記述されることが多いです。

または、Navigatorウィジェットの静的メソッドpushを直接呼び出しても結果は同じです。

```
Navigator.push(
  context,
  MaterialPageRoute(
    builder: (_) => const SecondScreen(),
  ),
);
```

続いて、遷移した先から元の画面に戻る方法を見てみましょう(こちらもコードは簡略化しています)。

```
class SecondScreen extends StatelessWidget {

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

```
    return ElevatedButton(
      child: const Text('戻る'),
      onPressed: () {
        Navigator.of(context).pop(); ——❶
      },
    );
  }

}
```

SecondScreen画面で「戻る」タップすると、元の画面に戻るサンプルです。NavigatorStateクラスのpopメソッドを呼び出し、元のFirstScreen画面へ戻ります(❶)。

上記のサンプルは理解を優先してコードを簡略化しているため、あまり一般的な実装ではなく外観も不恰好です。以下が完全な動作サンプルとなります。

./lib/main.dart

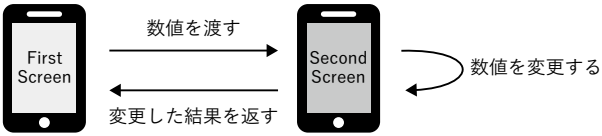
```
import 'package:flutter/material.dart';

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

// アプリ起動時に表示されるFirstScreenウィジェット
class FirstScreen extends StatelessWidget {
  const FirstScreen({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      /* ◆ AppBar
      画面上部のヘッダ部分となるWidget */
      appBar: AppBar(
        title: const Text('FirstScreen'),
      ),
      body: Center(
        child: ElevatedButton(
          child: const Text('次へ'),
          onPressed: () {
            Navigator.of(context).push(
              MaterialPageRoute(
                builder: (_) => const SecondScreen(),
              ),
            );
```

```
          },
        ),
      ),
    );
  }
}

// 画面遷移先として用意したSecondScreenウィジェット
class SecondScreen extends StatelessWidget {
  const SecondScreen({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('SecondScreen'),
      ),
      body: Center(
        child: ElevatedButton(
          child: const Text('戻る'),
          onPressed: () {
            Navigator.of(context).pop();
          },
        ),
      ),
    );
  }
}
```

画面間でのデータの受け渡し

続いて画面間でデータを受け渡す方法を見ていきましょう。数値を表示するFirstScreen画面と、数値を受け取り変更するSecondScreen画面を用意します。SecondScreen画面で変更された数値はFirstScreen画面に反映させるので、FirstScreen画面はStatefulWidgetを継承します(**図5.6**)。

図5.6 **画面間でのデータの受け渡し**



まずはFirstScreen画面を見てみましょう。