DroidKaigi 2019

JA Room 6 - 2019/02/08 17:40-18:10

FlutterでのWidgetツリーへの状態伝播とアクセス制限の基本戦略

robo

自己紹介

名前

robo（兼高理恵） @cch_robo

好きなもの

モバイル端末

おしごと

アプリの設計から実装まで

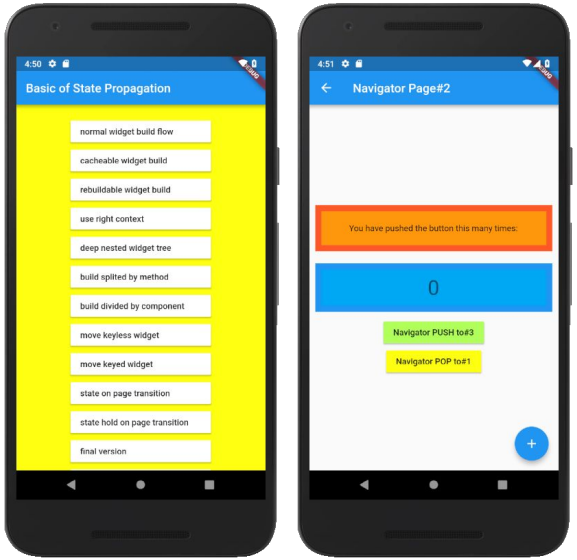
このセッションでは、
Flutterのネストが深くならないように気をつけたり、
基本ウィジェット(*1)のみを使った、アプリ全体での状態や
ロジックの共有とアクセスの制限について説明します。

初学者が対象ですので、BLOCなど応用技術の説明ではありません。

(*1) InheritedWidget

サンプルソース

github にサンプルソースを公開しています。
https://github.com/cch-robo/basic_strategy_of_state_propagation_in_Flutter



サンプルソースは、
個別のアプリ(右)として
ラウンチャー(左)から起動できますので、
セッションで説明した実装や挙動の確認に
御利用ください。

課題）Flutterのツリー定義コードは、ネストが深い

課題）Flutterのツリー定義コードは、ネストが深い

Flutter は、
Widget の入れ子が深くなる宿命を負っています。

これは、単一の機能を持った Widget を組み合わせることで
強力な効果を出す設計思想(*1)を取っているからです。

入れ子を作ることを前提にしているので、
どうしてもネストは深く拡くなります。

Flutter.io docs Technical Overview

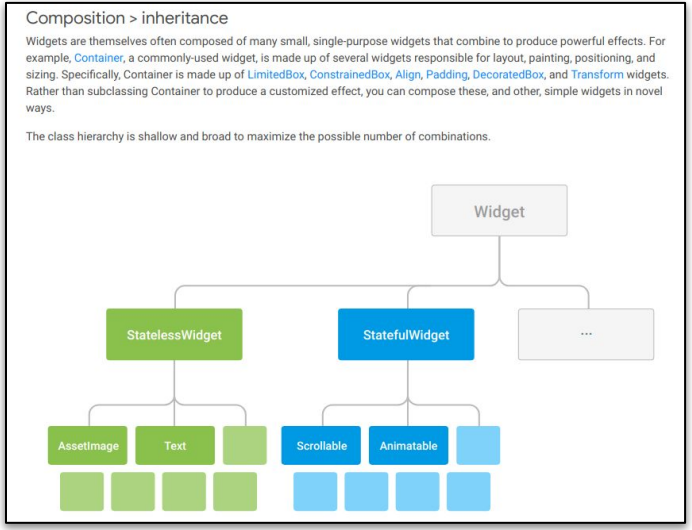
(*1)Composition > inheritance

https://flutter.io/docs/resources/technical-overview#composition--inheritance

Widgets are themselves often composed of many small, single-purpose widgets that combine to produce powerful effects.

ウィジェット自体は、多くの場合、強力な効果を生み出すために組み合わされた多数の小さな単一目的のウィジェットで構成されます。

カスタマイズされた効果を生み出すためにサブクラス化するのではなく、単純なウィジェットを斬新な方法で構成することができるように、
クラス階層も、可能な組み合わせを最大にするため広く浅いそうです。



Google Developers flutter Playlists より

How to Create Stateless Widgets - Flutter Widgets 101 Ep. 1
https://youtu.be/wE7khGHVkYY?list=PLOU2XLYxmsIJyiwUPCou_OVTpRIn_8UMd&t=102

I'm composing my interface by combining a bunch of simple widgets, each of which handle one particular job.

私はインターフェースを、それぞれが1つの特定の仕事を処理する単純なウィジェットの束を組み合わせることによって作り上げています。

Google Developers flutter Playlists
https://www.youtube.com/user/GoogleDevelopers/playlists?shelf_id=66&sort=dd&view=50

解決案）buildメソッドの入れ子の記述を外出しする。

ビルドツリーのソースから、一部の入れ子構造の記述を外出しすることでコードを読みやくすくします。

- 特定の機能やUI表現を担わせた
  Widget クラス(コンポーネント)を作る。
- 一部の入れ子構造の記述をメソッド化する。
  (クラス内にコードが取り残され、クラス外との共有に難があります。)

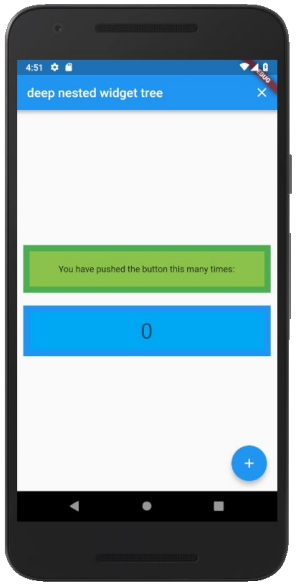
```dart
body: Container(
  alignment: Alignment.center,
  child: MyColumn(
    mainAxisAlignment: MainAxisAlignment.center,
    children: <Widget>[

      Text(
        'You have pushed the button this many times:',
      ), // Text

      Text(
        '$_counter',
        style: Theme.of(context).textTheme.display1,
      ), // Text

    ], // <Widget>[]
  ), // MyColumn
), // Container
```

```dart
body: Container(
  alignment: Alignment.center,
  child: MyColumn(
    mainAxisAlignment: MainAxisAlignment.center,
    children: <Widget>[

      Container(
        color: Colors.green,
        alignment: Alignment.center,
        margin: EdgeInsets.all(10.0),
        padding: EdgeInsets.all(10.0),

        child: Container(
          color: Colors.lightGreen,
          alignment: Alignment.center,
          padding: EdgeInsets.all(20.0),

          child: Text(
            'You have pushed the button this many times:',
          ), // Text

        ), // Container

      ), // Container

      Container(
        color: Colors.blue,
        alignment: Alignment.center,
        margin: EdgeInsets.all(10.0),
        padding: EdgeInsets.all(10.0),

        child: Container(
          color: Colors.lightBlue,
          alignment: Alignment.center,
          padding: EdgeInsets.all(10.0),

          child: Text(
            '$_counter',
            style: Theme.of(context).textTheme.display1,
          ), // Text

        ), // Container

      ), // Container

    ], // <Widget>[]
  ), // MyColumn
), // Container
```

課題例

ノーマルのカウントアプリの
Text 表示をデコってネストを
深くしました。

左）サンプルソース名
　deep_nested_widget_tree.dart



```
body: MyContainer(
  alignment: Alignment.center,
  child: MyColumn(
    mainAxisAlignment: MainAxisAlignment.center,
    children: <Widget>[

      // ラベル表示部のコンポーネント
      MyLabelStatelessComponent(),

      // カウンター表示部のコンポーネント
      MyCounterStatelessComponent(parameter: _counter),

    ], // <Widget>[]
  ), // MyColumn
), // MyContainer
```

解決案

サンプルソース名
build_devided_by_component.dart

StatelessWidget / StatefulWidget を継承したコンポーネントWidget クラスを新設し、コードを外出ししました。
カウント値は、コンストラクタ引数で受け取ります。
　　ラベル表示コンポーネントは、割愛しています。

```
/// カウンター表示部のコンポーネント (StatelessWidget)
class MyCounterStatelessComponent<T> extends MyStatelessWidget {
  final T parameter;
  final Color outerColor;
  final Color innerColor;

  MyCounterStatelessComponent({
    this.parameter,
    this.outerColor = Colors.blue,
    this.innerColor = Colors.lightBlue,
    String name = "MyCounterComponent",
    Key key,
  }) :  super(name: name, key: key);

  Widget build(BuildContext context) {
    debugPrint("$name#build(context:${context.hashCode})  instance=

    return
      MyContainer(
        name: "counterOuterMyContainer",
        color: outerColor,
        alignment: Alignment.center,
        margin: EdgeInsets.all(10.0),
        padding: EdgeInsets.all(10.0),

        child: MyContainer(
          name: "counterMyContainer",
          color: innerColor,
          alignment: Alignment.center,
          padding: EdgeInsets.all(10.0),

          child: MyText(
            '$parameter',
            name: "counterMyText",
            style: Theme.of(context).textTheme.display1,
          ), // MyText

        ), // MyContainer

      ); // MyContainer
  }
}
```

結論）コンポーネントWidgetを作る/使う

- Flutterは、設計思想によりネストが深くなる宿命を負う。
- 機能やUI単位でコンポーネントWidgetを作る。
  - 入れ子記述が外出しされコードが読みやすくなる。
  - コンポーネント化されたWidgetは、使いまわせる。
  - I/Oの初期化や破棄などがない限り、
    StatelessWidget 継承で構わない。

サンプルソース、state on page transition では、
3つのページで、同じコンポーネントWidgetクラスを使いまわしています。

課題）ウィジェットからビジネスロジックを分離したい
課題）InheritedWidget クラスを使ってみたい

ウィジェットからビジネスロジックを追い出すため、
InheritedWidget でアプリ全体やページ間で
状態や処理関数を共有したい。

状態と処理関数を提供する ロジッククラス と、
ページごとにユニークな ページ InheritedWidget クラスと、
アプリ全体用の アプリ InheritedWidget クラスを作り、

ページ InheritedWidget にページ単位のスコープを
アプリ InheritedWidget にアプリ全体のスコープをもたせ、

build(context)メソッド中のウィジェットから、
ページ専用の ロジックオブジェクト と、
アプリ全体で共有する ロジックオブジェクト と、
グループで共有する ロジックオブジェクト を選択して、
状態値の取得や処理関数の呼び出しを行えるようにします。

ロジッククラス は、
一般的なビジネスロジッククラスと同じ作りです。

解決案

注意しなければならないことは、
ページ単位のスコープの作り方と、
アプリ全体のスコープの作り方と、

ページ間では、
直接のアクセスはできないため、
グループ共有の ロジックオブジェクト は
アプリ InheritedWidget で管理すること、

アプリ InheritedWidget では、
どのページやグループからアクセスされたのかをチェックして、許可のないページやグループからのアクセスを防ぐ実装を追加することです。

基本的な

ページ InheritedWidget クラス と

アプリ InheritedWidget クラス と

ロジッククラス を作る

ページ や **アプリ** InheritedWidget は、特別なクラスではないので基本構造は、通常 InheritedWIdget と変わりません。

- InheritedWidget を継承
- updateShouldNotify() をオーバライド
  一般的にtrue を返して更新があったら再構築させる。
- context から自分型のインスタンスを返す static 関数の of を提供する。
- 右記例では、状態や処理をロジック用の PageLogic クラスに分離

SamplePageInheritedWidgetは一般例です。

```dart
/// Page のコンポーネント
class SamplePageInheritedWidget extends InheritedWidget {

  SamplePageInheritedWidget({
    Key key,
    Widget child,
  }) :  super(key: key, child: child);

  @override
  bool updateShouldNotify(InheritedWidget oldWidget) {
    return true;
  }

  /// Pageのコンポーネントを取得
  static SamplePageInheritedWidget of(BuildContext context) {
    return context.inheritFromWidgetOfExactType(SamplePageInheritedWidget);
  }

  PageLogic get logic =>_logic;

  final PageLogic _logic = new PageLogic();
}

/// Pageのロジッククラス
class PageLogic {
  int _counter;

  MyPageLogic() {
    clear();
  }

  int get counter => _counter;

  void increment() => _counter++;

  void clear() {
    _counter = 0;
  }
}
```

カウンタアプリの
状態とメソッドの
ロジッククラス化例

BuildContext#inheritFromWidgetOfExactType(Type targetType)

- このメソッドは、
  ツリー内をさかのぼり最も近かった、targetType で指定された InheritedWidget 型のオブジェクトへの参照を返すメソッドです。

- 探索目的と異なるオブジェクトであっても、
  直近にあった指定型のオブジェクトが返ることに注意ください。

ページ InheritedWidget クラスは、ページごとに作成

- 後段の アプリ InheretedWidget でのアクセス元ページの判断を簡素化するため、**ページ InheritedWidget** クラスは、ページごとに作成してください。

ページ InheritedWidget に
ページ単位のスコープをもたせる。

ページ単位のスコープをもたせる

- **ページ InheritedWidget** が、MyHomePage のようなページ表示を作る、**画面作成ウィジェット**の build(context) メソッド内からアクセスできるよう、child プロパティに**画面作成ウィジェット**を指定して入れ子にする必要があります。
- InheritedWidgetは、画面生成を行わず入れ子になったウィジェットを返すだけなので、入れ子にしてもページ表示には影響を与えません。
- MaterialPageRoute にアタッチするため、MaterialApp() や Navigator.push() で入れ子にした **ページ InheritedWidget** を渡します。
- ページツリー内では、**ページ InheritedWidget** を複数使わないでください。

MaterialApp と Navigator.push での入れ子指定

```dart
@override
Widget build(BuildContext context) {
  return MaterialApp(
    title: 'Basic Strategy of State Propagation',
    theme: ThemeData(
      primarySwatch: Colors.blue,
    ), // ThemeData
    home: MyHomePageInheritedWidget(
        child: MyHomePage(title: 'Sample')),
  ); // MaterialApp
}
```

```dart
Navigator.push(
  context,
  MaterialPageRoute<void>(
      builder: (BuildContext context) {
        return Page2InheritedWidget(
            child: NavigatorPushPopPage());
      }
  ), // MaterialPageRoute
);
```

アプリ InheritedWidget に
アプリ全体のスコープをもたせる。

アプリ全体のスコープをもたせる

- ページは、MaterialPageRoute の子要素です。
- 上記より ページ同士は、入れ子になっていないので、お互いの **ページ InheritedWidget** にアクセスできません。
- **アプリ InheritedWidget** にどのページからでもアクセスできる、アプリ全体でのスコープをもたせるには、child プロパティに MaterialApp を生成するアプリ構築ウィジェットを指定して入れ子にし **runApp()** 関数に渡します。

runApp() での入れ子指定

```
void main() => runApp(
                  AppInheritedWidget(
                    child: MyApp()
                  ) // AppInheritedWidget
                );
```

AppInheritedWidget は、
アプリ全体共有の InheritedWidget

許可されたページからのアクセスなのかチェックする。

許可されたページからのアクセスなのかチェックする

基本的に以下のようなロジックを取ります。

- ページごとに **ページ InheritedWidget** クラスをユニークにします。
- アクセスチェックごとに、引数に BuildContext をとる static API を作成します。
- アクセスチェックする API 中で BuildContext#ancestorWidgetOfExactType() で許可されたページの **ページ InheritedWidget** 型なのかチェックします。
- アクセスが許可された場合、**ロジックオブジェクト**を返します。

パーミッションチェック・コンセプト

```dart
/// ページ3からのアクセスのときのみ ページ3のロジックを返します。
static PageLogic ofPage3(BuildContext context) {
  AppInheritedWidget appLogic = AppInheritedWidget.of(context);
  return appLogic.checkPermission(context, Page3InheritedWidget)
      ? appLogic._page3Logic : null;
}

// ページ1～3のロジック
final _page1Logic = new PageLogic(name: "MyPageLogic[#1]");
final _page2Logic = new PageLogic(name: "MyPageLogic[#2]");
final _page3Logic = new PageLogic(name: "MyPageLogic[#3]");

/// 指定クラスのインスタンス(継承物は除外)が、context に含まれるか否かを返します。
bool checkPermission(BuildContext context, Type targetType) {
  return context.ancestorWidgetOfExactType(targetType) != null;
}
```

BuildContext#ancestorWidgetOfExactType(Type targetType):Widget

- ツリー内をさかのぼり、指定型のウィジェットがあれば直近の参照を返します。
- 指定型の継承クラスは、探索対象に含まれません。

許可されたグループからのアクセスなのかチェックする。

許可されたグループからのアクセスなのかチェックする

- ページ側は、グループとしたいページの **ページ InheritedWidget** を共通の**ラップウィジェット**で更に入れ子にくるみ、MaterialApp() や Navigator.push() で、MaterialPageRoute にアタッチしてもらいます。

- チェック側は、グループアクセスチェックの API で、
共通の**ラップウィジェット**型が含まれているかをチェックします。

ラップウィジェットは、単純なもので構いません

```dart
/// 簡易許可チェック用ラッパー
class PermissionWidget extends StatelessWidget {
  final Widget child;

  PermissionWidget({Key key, this.child}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return child;
  }
}
```

状態やロジックを共有する

- InheritedWidget を使えば、任意の**ロジッククラス**のオブジェクトが渡されるので、データの授受もJSONなどに変換したりせず直接渡しができます。

- **ロジッククラス**は、プログラマ側にウィジェットを継承する必要のない純粋なプログラム開発を提供しますし、ウィジェットから自由にアクセスできるので、ウィジェット内に書いていた状態の保管や長大な処理コードの移管にも利用できるでしょう。

- また **アプリ InheritedWidget** は、ページよりも長命なので、次の画面表示で復元してほしい、スクロール位置の一時保管も手軽に実装できるでしょう。

サンプルソース紹介

```dart
/// 第1のページ
class NavigatorPushPage extends StatefulWidget {
  NavigatorPushPage({Key key, this.title = "State Propagation Page#1"})
      : super(key: key);

  final String title;

  @override
  _NavigatorPushPageState createState() {
    return _NavigatorPushPageState();
  }
}

class _NavigatorPushPageState extends State<NavigatorPushPage> {

  _NavigatorPushPageState() : super();

  @override
  Widget build(BuildContext context) {

    PageLogic logic = Page1InheritedWidget.getLogic(context);

    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
        actions: <Widget>[
```

サンプルでは、
画面間を遷移(ウィジェット再生成)してもカウンタ値が保持されています。

またカウンタアプリにあった _count 状態と _increment() 関数を全て外出しにしています。

strategy_of_state_propagation.dart

https://github.com/cch-robo/basic_strategy_of_state_propagation_in_Flutter/blob/master/lib/src/strategy_of_state_propagation.dart

ピンポイントで再描画と再描画抑止をするウィジェット

サンプルでは、
子ウィジェットを任意で再生成する
(可変にする) RebuildableWidget と、

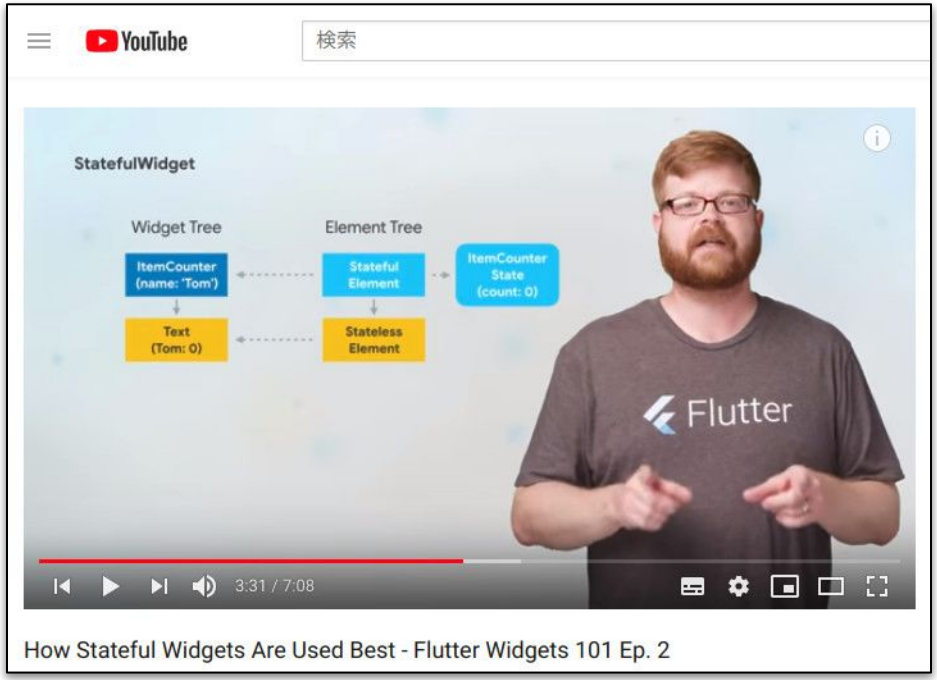
ウィジェットをキャッシュして、
再生成させない(不変にする) ConstantWidget
のカスタムウィジェットを使っています。

```
class _MyHomePageState extends MyState<MyHomePage> {
  int _counter = 0;
  final GlobalKey<MyRebuildableWidgetState>
    _myRebuildableWidgetKey = new GlobalKey<MyRebuild

  _MyHomePageState({String name}): super(name: name);

  void _incrementCounter() {
    _counter++;
    _myRebuildableWidgetKey.currentState.rebuild();
  }
```

```
// RebuildableWidget は、rebuild() 指定で、
// 子ウィジェットを再生成する(可変にする) Widget。
child: MyRebuildableWidget(
    key: _myRebuildableWidgetKey,
    builder: (context, setStateFunc) =>
      MyColumn(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[

          // ConstantWidget は、自分が再生成されても、
          // 子ウィジェットは再生成させない(不変にする) Widget。
          MyConstantWidget(
            builder: (context, setStateFunc) =>
              MyText(
                'You have pushed the button this many times:',
                name: "labelMyText",
              ), // MyText
          ), // MyConstantWidget
          MyText(
            '$_counter',
            name: "counterMyText",
            style: Theme.of(context).textTheme.display1,
          ), // MyText

        ], // <Widget>[]
      ), // MyColumn
```

Flutter Widgets 101 視聴のススメ

基本Widgetの紹介の中で
ディープな基礎知識を紹介



5～10分のミニセッション形式ながら
今まで、説明されてこなかった
Flutterでの ツリー構築の効率化が
紹介されています。

Flutter Widgets 101
https://www.youtube.com/playlist?list=PLOU2XLYxmsIJyiwUPCou_OVTpRIn_8UMd

初歩的な内容でしたが、

いかがだったでしょうか。

ご清聴、ありがとうございました。