【 Flutter 】 InheritedWidget って何？

2022.08.09 2022.07.06

勉強していたら InheritedWidget ってWidget が出てきたけれど、
一体何なんだろう？

状態管理で使えるらしいけれど、どう使うのか気になるわ！

本記事ではそんな疑問にお答えします。

ProviderやRiverpod の内部で使われる、状態管理の基礎となるWidget、
InheritedWidget について解説します。

最近、公式でも紹介されていました。

基本的な使い方を始めとして、
内部でどんなことが行われているのかについても触れていきます。

今はとても優れた状態管理パッケージがたくさんあるので、
わざわざこのInhertitedWidgetを使うことはないかと思います。

ですが、温故知新という言葉があるように、
昔の、基礎となる優れた考え方を知ることは、
新しいことを発見する足がかりとなるかもしれません。

なので本記事は、初心者の方から理解を深めたい中級者の方まで
有用な記事となっているかと思います。
かなり長い記事となりますが、ぜひ読んでみて下さい！

目次 [閉じる]

InheritedWidget の概要と使い方

InheritedWidget の概要と使い方について解説していきます。

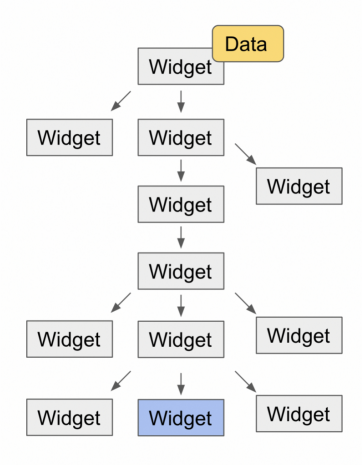
InheritedWidget はProvider や Riverpod の内部でも使われている、
状態管理の基礎となるWidget です。

このため、Inherited Widgetだけでも状態管理は可能となっています。

「そもそも状態管理って何？」と思われる方がいるかも知れません。
以下で状態管理の課題(InheritedWidget で解決したい課題)について、
まずは見ていきましょう。

InheritedWidget で解決したい課題

Flutterのコードを書くことにある程度慣れてくると、
Widget の build メソッドの中にWidgetを何度も追加して、
Widgetの依存関係がどんどん深くなって行くかと思います。（以下の図）



課題になるのは、最下層のWidget(水色)で最上部のWidgetが
持つデータ(黄色)を参照したい時です。

どうやって参照すれば良いでしょうか？

一つの方法は、上のWidgetからコンストラクタを使ってデータを受け渡していく方法です。
（以下のコード）

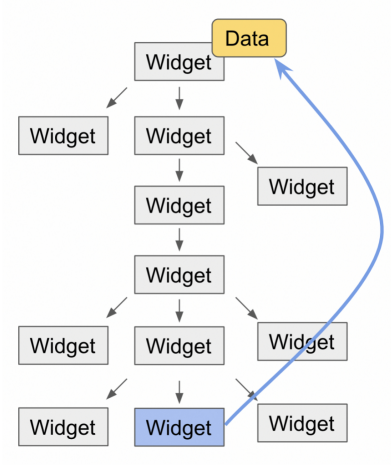
```dart
class GrandpasWidget extends StatefulWidget {
  const GrandpasWidget({super.key});

  @override
  State<GrandpasWidget> createState() => _GrandpasWidgetState();
}

class _GrandpasWidgetState extends State<GrandpasWidget> {
  int data = 100;

  @override
  Widget build(BuildContext context) {
    return FathersWidget(
      data: data,
    );
  }
}

class FathersWidget extends StatelessWidget {
  const FathersWidget({super.key, required this.data});

  final int data;

  @override
  Widget build(BuildContext context) {
    return MyWidget(data: data);
  }
}

class MyWidget extends StatelessWidget {
  const MyWidget({super.key, required this.data});
//・・・
}
```

この方法だと確実にデータは受け渡せますが、
同じようなコードを何度も書くことになり、ちょっと冗長ですよね。

別荘
勉強会
テスト
企業紹介
未カテゴリー

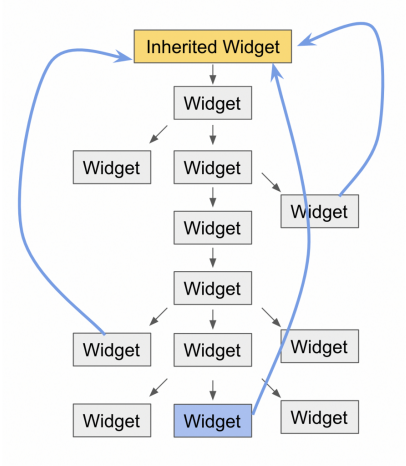
目次

可能であれば以下のように直接参照したいです。



これを可能にするのが`Inherited Widget`です。

データを`InheritedWidget`に持たせることで、
依存関係が`InheritedWidget`の下にある`Widget`ならどこからでも
データを参照できるようになります。



`InheritedWidget` はFlutterのSDK の中の`Widget`なので、
特別にパッケージをインストール必要はありません。

`InheritedWidget` で課題が解決できるなこと、
わかっていただけましたでしょうか。

では、実際のコードにて基本的な使い方を解説していきます。

基本的な使い方

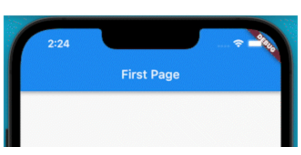
基本的な使い方の概要は以下の通りです。

1. データを共有したい`Widget`群の上部で全てと依存関係がある部分に、
   `InheritedWidget` 継承クラスを配置する
2. `InheritedWidget` 継承クラスに用意したメソッドにてデータを参照する

ベースとなるコードはこちらになります。

2つの画面にカウンターが設定されています。
このアプリでは、2つの画面で同じ値を状態として共有したいです。
そのため、依存関係が上のWidgetに状態（データ）をもたせる必要がある、
そんなアプリとなっています。

＋ サンプルコード全体はこちら



今回のコードはFlutter 3.0.4 にて記載します。

InheritedWidget 継承クラスの配置

まず、InheritedWidgert 継承クラスを用意します。

定義のコードは以下の通りです。

```
class InheritedCounter extends InheritedWidget {
  const InheritedCounter({
    super.key,
    required this.counter,
    //1
    required super.child,
  });

  final int counter;

  //2
  @override
  bool updateShouldNotify(covariant InheritedWidget oldWidget) => tru
}
```

共通で持たせたいデータのcounter の部分以外は、
InheritedWidgetの継承により必要なコードとなっています。

//1
InheritedWidgetの継承クラスはchildを引数に設定する必要があります。

//2
InheritedWidgetの継承クラスはupdateShouldNotifyメソッドを
オーバーライドする必要があります。
このメソッドは、このInheritedWidgetの継承クラスがリビルドされた際に
InheritedWidgetの継承クラスからデータを受け取ったWidgetをリビルドするか否かを
判定するメソッドです。
簡易化のため、今回は常にtrueを返すとしています。

次に、MyHomePageとMySecondPage が共通で依存関係を持っているMyAppにて、
このInheritedCounterクラスを配置します。

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

  @override
  Widget build(BuildContext context) {
    return const InheritedCounter(
      counter: 100,
      child: MaterialApp(
        home: MyHomePage(),
      ),
    );
  }
}
```

データの参照

InheritedCounterにて設定したcounter = 100 という値、
これをMyHomePage、MySecondPageにて参照するためには、
以下のdependOnInheritedWidgetOfExactTypeメソッドを用います。

```
context.dependOnInheritedWidgetOfExactType<InheritedCounter>()!.count
```

ただ、このメソッド、ちょっと長いですよね。

これをもっと短くするためにInheritedCounter に以下のメソッドを追加しましょう。

```
class InheritedCounter extends InheritedWidget {
  const InheritedCounter({
    super.key,
    required this.counter,
    required super.child,
  });

  final int counter;
  //このメソッドを追加
  static InheritedCounter of(BuildContext context) =>
      context.dependOnInheritedWidgetOfExactType<InheritedCounter>()!

  @override
  bool updateShouldNotify(covariant InheritedWidget oldWidget) => tru
}
```

これにより以下のメソッドで参照可能となります。

```
InheritedCounter.of(context).counter
```

このメソッドを追加してデータを参照したアプリのコードは以下の通りです。

＋ サンプルコード全体はこちら

コードを実行すると、ちゃんとWidgetの依存関係の上部で設定したcounter = 100
という値を取得できていることがわかります。

dependOnInheritedWidgetOfExactTypeについては後半で解説します。

データの更新

カウンターアプリとしては、参照だけでなく値の更新もしたいところです。
InheritedCounter.of(context).counter++
とすればよさそうですが、これはできません。
StatefulWidget のStateと違い、 InheritedWidget (というかWidget)は
一度インスタンスが生成された後、自身を変えることができない、
immutable(不変)の性質を持つからです。

ではどうすればよいでしょうか？

答えは、データを変えることのできるStatefulWidget と
InheritedWidget を組み合わせる、です。

ここから若干テクニカルなことをします。
一旦組み合わせたコードを見てみましょう。

```
//3
class _InheritedCounter extends InheritedWidget {
  const _InheritedCounter({
    required this.data,
    required super.child,
  });
  //4
  final MyCounterState data;

  @override
  bool updateShouldNotify(covariant InheritedWidget oldWidget) => tru
}

class MyCounter extends StatefulWidget {
  const MyCounter({
    super.key,
```

```
    super.key,
    required this.child,
  });

  //5
  final Widget child;

  //6
  static MyCounterState of(BuildContext context) {
    return context
        .dependOnInheritedWidgetOfExactType<_InheritedCounter>()!
        .data;
  }

  @override
  State<MyCounter> createState() => MyCounterState();
}

//7
class MyCounterState extends State<MyCounter> {
  int count = 0;

  void increment() => setState(() {
        count++;
      });
  //8
  @override
  Widget build(BuildContext context) {
    return _InheritedCounter(
      data: this,
      child: widget.child,
    );
  }
}
```

//3
InheritedWidget をMyCounter , MyCounterState (組み合わせるStatefulWidgetとState)
からしか参照しなくなるため、プライベート(アンダーバー付き)にします。

//4
InheritedWidget で保持するデータを、
MyCounterState (組み合わせるStatefulWidgetのState)とします。

//5
MyCounter Widget では受け取ったWidget を_InheritedCounterでラップし返す、
という処理を行うためWidgetを受け取るよう設定します。

//6
InheritedCounterをプライベートにしたため、
InheritedCounterにあったofメソッドをMyCounterに移動しています。

//7
ofメソッドで MyCounterStateを返すため、
MyCounterStateをパブリック(アンダーバーなし)にしています。

//8
childで受け取ったWidgetをbuildメソッドで返す際に
_inheritedCounterで囲んで返します。
これによりMyCounterより依存関係が下のWidgetはInheritedWidget で
囲まれることとなります。
_inheritedCounterに設定したthisはbuildメソッドを実行している
MyCounterState自身を表しています。

先程InheritedWidget 継承クラスで囲んでいた代わりに、
作成したMyCounterで囲います。

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

  @override
  Widget build(BuildContext context) {
    return const MyCounter(
      child: MaterialApp(
        home: MyHomePage(),
      ),
    );
  }
```

```
12 }
```

値の更新はMyCounterState の メソッド(increment)で行います。

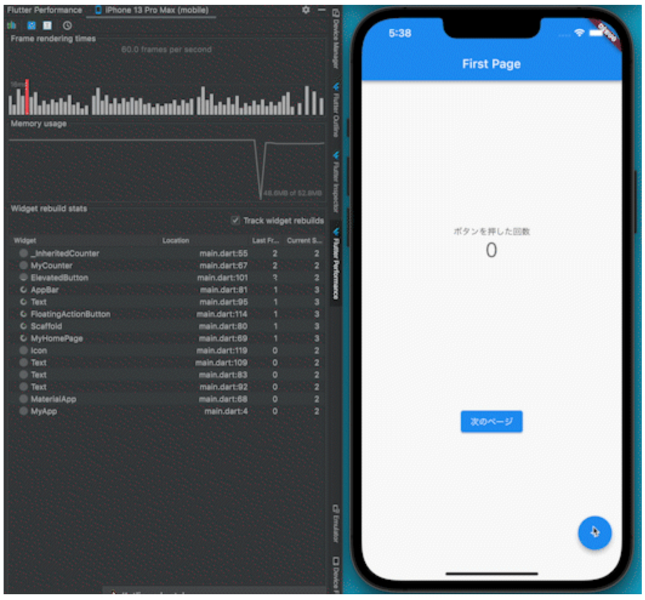
更新の仕組みとしては以下のようになります。

1. MyCounter.of(context).increment()でMyCounterStateのcounterの値を更新、
   setStateが実行される
2. setStateによりMyCounterStateがリビルドされ、
   値の更新されたMyCounterStateが_inheritedCounterに渡される
3. _inheritedCounterがデータの変化を感知し、
   _inheritedCounterのデータを観測しているWidgetにリビルドをリクエストする
4. _inheritedCounterのデータを観測しているWidgetがリビルドされ
   更新されたデータを取得、画面に更新されたデータが表示される

サンプルコード全体はこちら

改良点

Flutter Performance で+ボタンを押した時のリビルドの状況を見てみましょう。



画面的に変化している部分はカウンター部分のTextだけですが、
変化していないScaffoldなどもリビルドされていることがわかります。

このリビルドを最小限に押さえるためにはどうすればよいでしょうか？

手順は2つです。

1. Builderを使ってリビルドされるWidgetを制限する

2. Floating Action Button にて『InheritedWidgetを監視しているものリスト』に登録されないようにする

Builderを使ってリビルドされるWidgetを制限する

MyCounter.of(context).〜(正確にはdependOnInheritedWidgetOfExactType)を使うと、
引数に使用したcontextが、
『Inherited Widgetを監視しているものリスト』に登録されます。

Inherited Widgetが更新されると、このリストに紐付いたWidgetがリビルドされる、
という仕組みとなっています。

Builderで囲まない場合だと、MyCounter.of(context).〜で引数に使用したcontextは、
MyHomePage Widgetのbuildメソッドのcontextのため、
Inherited Widgetが更新されるとMyHomePage Widgetが更新されてしまう訳です。

以下の画像のようにBuilder Widgetを使ってText Widgetを切り出すと、
MyCounter.of(context).〜で引数に使用したcontextはBuilder Widget のcontextとなるため、
リビルドされるWidgetをBuilder以下に制限することができます。

MyHomePageのcontextを使用

→ MyHomePage がリビルド対象になる

Builderのcontextを使用

→ Builder がリビルド対象になる

Floating Action Button にて『InheritedWidgetを監視しているものリスト』に登録されないようにする

タップすることデータを増加させる Floating Action Button でも
MyCounter.of(context).〜 (この内部でのdependOnInheritedWidgetOfExactType)
が使われているため、
タップすると使用しているcontextが『InheritedWidgetを監視しているものリスト』
に登録されてしまいリビルドの範囲が拡大されてしまいます。

そこで、タップされたときにはdependOnInheritedWidgetOfExactTypeではなく、
getElementForInheritedWidgetOfExactTypeを使うようにします。

> getElementForInheritedWidgetOfExactTypeについても後半で解説します。

具体的には、MyCounterのofメソッドを以下のように書き換えます。

```
static MyCounterState of(BuildContext context, {bool rebuild = true
  return rebuild
      ? context.dependOnInheritedWidgetOfExactType<_InheritedCounte
      : (context
              .getElementForInheritedWidgetOfExactType<_InheritedCo
              .widget as _InheritedCounter)
          .data;
}
```

> getElementForInheritedWidgetOfExactTypeの返す型の関係上、
> dependOnInheritedWidgetOfExactTypeと若干記載が変わっています。

引数に、リビルド対象とするか否かを判定するフラグを持たせ、
InheritedWidgetの更新にあわせてリビルドしたいときには
dependOnInheritedWidgetOfExactTypeを、
リビルドしたくない際にはgetElementForInheritedWidgetOfExactTypeを

使えるようにしています。

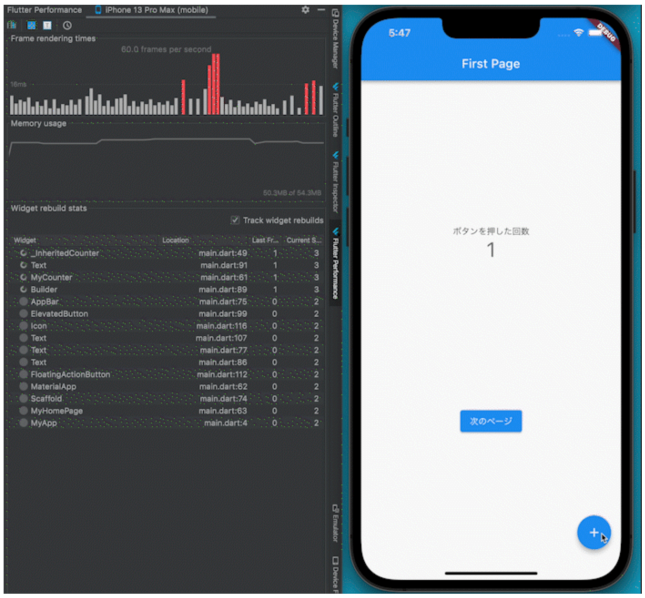
具体的なFloating Action Buttonの実装は以下のようになります。

```
        floatingActionButton: FloatingActionButton(
          onPressed: () {
            MyCounter.of(context, rebuild: false).increment();
          },
          child: const Icon(Icons.add),
        ),
```

以上が改善内容となります。

＋ 改善した全体のコードはこちら

結果、カウントアップした際のリビルドの対象を少なくすることができました。



以上がInherited Widgetの概要でした。
次は具体的にInherited Widgetがどんなことをしているのか、内部の仕組みを追っていきます。

Inherited Widget の仕組み

Inherited Widget の仕組みについて見ていきましょう。

この章を読めば、『何故dependOnInheritedWidgetOfExactType が優れているのか？』
がわかるはずです。

そのためにまず基礎知識としてElement とは何か？について簡単に解説します。

Elementとは何か？

Widgetが描画される時、内部では、3種類の要素が活躍しています。
Widget , Element, RenderObjectです。

それぞれの役割は以下のようになっています。

- Widget：そのWidgetの設定を管理するもの
- Element：そのWidgetのツリー上での位置やライフサイクルを管理するもの
- RenderObject：そのWidgetのサイズやレイアウト、描画を管理するもの

Widgetのツリーとは Widgetの親と子を結ぶことで表現した依存関係（またはその図）のことです。

この3種類については以下の動画がわかりやすいので興味のある方はぜひ見てみて下さい。

特にElementはそのWidgetについてどんな祖先や子がいるのかの依存関係や、
Widgetが更新された時どのように更新、再構築するかなどのライフサイクルを管理する、
重要な役割を担っています。

InheritedWidgetもこのElementと深く関わっています。
次の節で見てみましょう。

InheritedElement の継承

Elementのクラスのコードの中に、_inheritedWidgets、というプロパティがあります。

これはその名の通り、先祖のInherited Widget のElement (InheritedElement)を保管しているMapです。

> _inheritedWidgetsはMap<Type, InheritedElement>?型で、
> Type は Inherited Widget を継承した型となります。

InheritedElementがmountされる(ツリー上に配置される)際に、
自身が_inheritedWidgetsに登録されます。

実装コードを見る

Elementがmountされる度に、親から子へ継承されるため、
すべてのInheritedElementの子のElementは、_inheritedWidgetsにて
祖先のInheritedElementを持っていることとなります。

ざっくばらんに言えば、Elementは祖先のInheritedElementの情報を保持している、ということです。

dependOnInheritedWidgetOfExactTypeについて

dependOnInheritedWidgetOfExactType メソッドはBuildContext のメソッドとして定義され、
BuildContextの実装であるElementにて実装されています。

> build メソッドで使っているcontextとは、そのWidgetのElementです。

実装コードは以下のようになっています。

framework.dart

```
1
2
3   @override
4   T? dependOnInheritedWidgetOfExactType<T extends InheritedWidget>({O
5     assert(_debugCheckStateIsActiveForAncestorLookup());
6     final InheritedElement? ancestor = _inheritedWidgets == null ? nu
7     if (ancestor != null) {
8       return dependOnInheritedElement(ancestor, aspect: aspect) as T;
9     }
10    _hadUnsatisfiedDependencies = true;
11    return null;
12  }
```

4行目にて、_inheritedWidgetsから指定した型のInheritedWidgetのInheritedElementを取得し、
ancestorに格納していることがわかります。

dependOnInheritedElementメソッドは処理後にancestorの持つWidgetを返すので、
結論、_inheritedWidgetsで保管していた祖先のInheritedElement、

亜ひにInheritedWidetを取得するメソッドとなっています。

このメソッドのすごいところは、祖先のInheritedWidgetを取得するのに、
がとても簡単なことです。

祖先のInheritedElementをすべての子のElementで保管していて、
その中から探すだけなので、どれだけ子が多かったとしても(ツリーが深かったとしても)
とても簡単に取得できるのです。
（計算量がO(1)で済みます。）

getElementForInheritedWidgetOfExactTypeについて

本記事で祖先のInheritedWidgetを取得する方法として紹介したものに、
getElementForInheritedWidgetOfExactTypeがありました。
こちらについても実装を見てみましょう。

framework.dart

```
  @override
  InheritedElement? getElementForInheritedWidgetOfExactType<T extends
    assert(_debugCheckStateIsActiveForAncestorLookup());
    final InheritedElement? ancestor = _inheritedWidgets == null ? nu
    return ancestor;
  }
```

こちらも同様に_inheritedWidgetsから欲しいInheritedWidgetのInheritedElementを取得している
ことがわかります。

dependOnInheritedWidgetOfExactTypeとの違いは、
dependOnInheritedElementメソッドを間に噛ませているか否かです。

では、dependOnInheritedElementメソッドの実装を見てみましょう。

```
  @override
  InheritedWidget dependOnInheritedElement(InheritedElement ancestor, { Object?
aspect }) {
    assert(ancestor != null);
    _dependencies ??= HashSet<InheritedElement>();
    _dependencies!.add(ancestor);
    ancestor.updateDependencies(this, aspect);
    return ancestor.widget as InheritedWidget;
  }
```

ポイントは、6行目のancestor.updateDependencies(this, aspect)です。
この行ではancestor(祖先のInheritedElement)のupdateDependenciesメソッドを呼び出し、
子孫である自身を引数に与えています。
このメソッドにより、祖先のInheritedElement内の_dependentsに子孫である自身が登録されます。

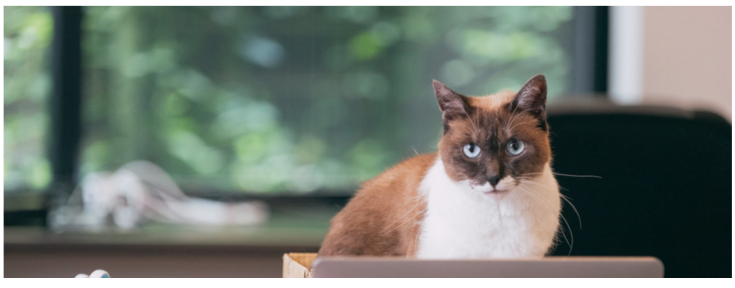
登録されたものはInheritedElement(正確には継承元のProxyElement)で実装されている
updateメソッドが発火した際に、子孫であるElementのWidgetをリビルドするように設定します。

まとめると、dependOnInheritedElementメソッドは呼び出しているElementを、
InheritedWidgetに関係するElementの集合
(前述の『『InheritedWidgetを監視しているものリスト』)に登録するメソッドです。

これが、getElementForInheritedWidgetOfExactTypeでは呼び出されていないため、
こちらでは監視対象リストに入らないということがわかります。

以上がInheritedWidgetの仕組みの話でした。

まとめ



本記事では`InheritedWidget`の基本的な使い方を始めとして、
内部でどんなことが行われているのかについて解説していきました。

かなり長い記事でしたが、いかがだったでしょうか？

現在は優れた状態管理パッケージが様々作成されているため、
今回の記事の内容が直接役に立つことはあまりないかもしれません。

ですが、最初に述べたように、昔のやり方を知ること、
基礎を知ることは、Flutter力の底上げとしてとても良いことだと考えます。

興味がある方は、Riverpod や Providerの内部実装コードのリーディングに挑戦してみて下さい。
今回記載した内容が多数見つかり興味深いはずです。

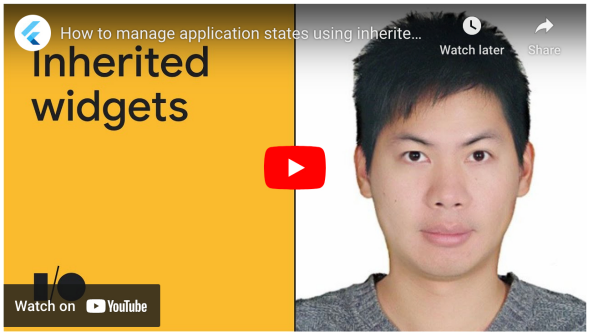
本記事があなたのアプリ開発の一助となれば幸いです。

参考

InheritedWidget class - widgets library - Dart API

API docs for the InheritedWidget class from the widgets library, for the Dart programming language.

api.flutter.dev



DartPad Workshops

dartpad.dev

InheritedWidgetの目的と使い方【Flutter】 - Qiita

まえがき InheritedWidgetの使い方の基本をまとめます！ Flutterを勉強していてよくわからなくなるポイントの一つがこのInheritedWidgetだと思います。筆者自身、これを理解するのにかなり時間がか...

qiita.com

「内側」から理解する Flutter 入門

モバイルアプリ開発の選択肢の1つとして大きな人気を得ている Flutter フレームワーク、みなさんはその「内側」を理解して使いこなしているでしょうか？この本では、Flutter が UI を作り上げるための中心的な役割を担っている…

zenn.dev

InheritedWidget を完全に理解する 🎯

Flutterフレームワーク・providerパッケージを支える重要なWidget

medium.com

Eric Windmill: Using Flutter Inherited Widgets Effectively

Eric Windmill Software Engineer

ericwindmill.com

Flutter - Widget - State - Context - InheritedWidget

Flutter - This article covers the important notions of Widget, State, Context and InheritedWidget in Flutter Applications. Special attention is paid on the Inhe...

www.didierboelens.com

Managing Flutter Application State With InheritedWidgets

Everyone has heard that interactive applications can be decomposed into three parts: model, view, and controller. Anyone who has given…

medium.com

編集後記（2022/7/6）

本記事はInheritedWidgetについての記事でした。

過去最高に学ぶことが多く、労力を注いだ記事となりました。
いかがだったでしょうか？

本記事がすぐに誰かの役に立つことはあまりないかもしれません。
ただこのような記事を書く意義はあるかと思っています。

すぐに役に立たない研究論文を書くのは有用なことか？
という議論を耳にしたことがあります。
その研究論文を読んで誰かが新たな発見をし論文を書き、
また異なる誰かがその論文を読んで新たな発見をする、
といった形で連鎖することがある、だから有用だ、という意見が寄せられていました。

本記事もこのような形で誰かの知識の糧になれば良いと思っています。

高望みかもしれませんが、本記事があなたの、
ないしまだ見ぬ誰かのアプリ開発の一助となれば幸いです。

週刊Flutter大学では、Flutterに関する技術記事、Flutter大学についての紹介記事を投稿していきます。
記事の更新情報はFlutter大学Twitterにて告知します。
ぜひぜひフォローをお願いいたします。

Widget

人気記事

【Dart】 List の使い方 【Flutter】

【2022年最新】 Flutter × Firebase でアプリを作ろう！

【Flutter】 余白 の付け方【padding,margin】

Flutter入門 ~ 環境構築から初心者向け学習方法まで ~ 【動画付き】

【2022年最新】Flutter × Riverpod の基本的な使い方解説！

新着記事

【Flutterニュース】Chompy終了、Dart 3.0.0リリース情報、最高のriverpod解説動画、ChatGPTとGitHub Copilotの実践レビューほか【2023年4月17日】

【2023年4月12日】今週のFlutterニュース

【Flutter】 BorderRadiusの使い方

【Flutter】サイトの画像を表示する

【Flutter】アプリ内の画像を表示する

Aoi

関連記事

【 Divider 】区切り線を実装しよう！【Flutter】

本記事では区切り線を実装できるウィジェット、Divider ウィジェットについて解説します。基本的な使い方からカスタマイズ方法まで詳細に解説します。ぜひ読んでみてください！

【Flutter】 IconButton 使ってみた！【Material 3】

本記事では IconButton ウィジェットの使い方について解説を行います。 Flutter3.3で追加されたMaterial3対応についても解説を行います。 ぜひ読んでみてください！

【 Flutter 】Medium & Large AppBar の実装【 Material 3 】

本記事では、Fluttter 3.3 で更新が告知された、Material 3 に対応したMedium とLarge の AppBar の 実装方法について解説します。 動きは派手ですが、意外と簡単に実装可能です。ぜひ読んでみてください！

【Flutter】 ドロップダウンボタン使ってみた！

Flutterでドロップダウン（プルダウン）の表示に使えるWidget、 DropdownButton Widgetを紹介します。 基本的な使い方から細かい設定まで、詳細に解説しています。 初心者必見の内容です。ぜひ読んでみてください！

【Flutter】 Stepper 使い倒してみた！

「ステップ毎にユーザーに作業してもらうUIを作りたいんだけど、 ゼロから作るの大変そう、、、」 そんな悩みに応えるのが Stepper Widgetです！ 本記事では、Stepper Widgetの紹介を行います。 基本的な使い方から詳細設定まで、徹底的に解説していきます。

【 Icon 】アイコンを実装しよう！ 【 Flutter 】

本記事ではFlutterでボタンや装飾として用いる記号のウィジェット、Icon ウィジェットの紹介を行います。基本的な使い方からカスタマイズ方法まで詳しく解説します。ぜひ読んでみてください！

Flutter ニュース 【2022年7月第1週】

【 NavigationRail 】 サイドメニュー を実装しよう！【 Flutter 】