Flutter 架構概述

本文旨在提供 Flutter 架構的高階概述，包括構成其設計的核心原則和概念。

Flutter 是一個跨平台 UI 工具包，旨在允許跨 iOS 和 Android 等作業系統重複使用程式碼，同時還允許應用程式直接與底層平台服務互動。目標是使開發人員能夠交付在不同平台上感覺自然的高效能應用程序，包容存在的差異，同時共享盡可能多的程式碼。

在開發過程中，Flutter 應用程式在虛擬機器中運行，該虛擬機器提供更改的有狀態熱重載，而無需完全重新編譯。對於發布，Flutter 應用程式會直接編譯為機器碼（無論是 Intel x64 還是 ARM 指令），或者如果面向 Web，則編譯為 JavaScript。該框架是開源的，擁有寬鬆的 BSD 許可證，並擁有一個蓬勃發展的第三方軟體包生態系統，補充了核心庫的功能。

本概述分為多個部分：

層模型：建構 Flutter 的各個部分。

反應式使用者介面：Flutter 使用者介面開發的核心概念。

小部件簡介：Flutter 使用者介面的基本構建塊。

渲染過程：Flutter 如何將 UI 程式碼轉換為像素。

平台嵌入器概述：讓行動和桌面作業系統執行 Flutter 應用程式的程式碼。

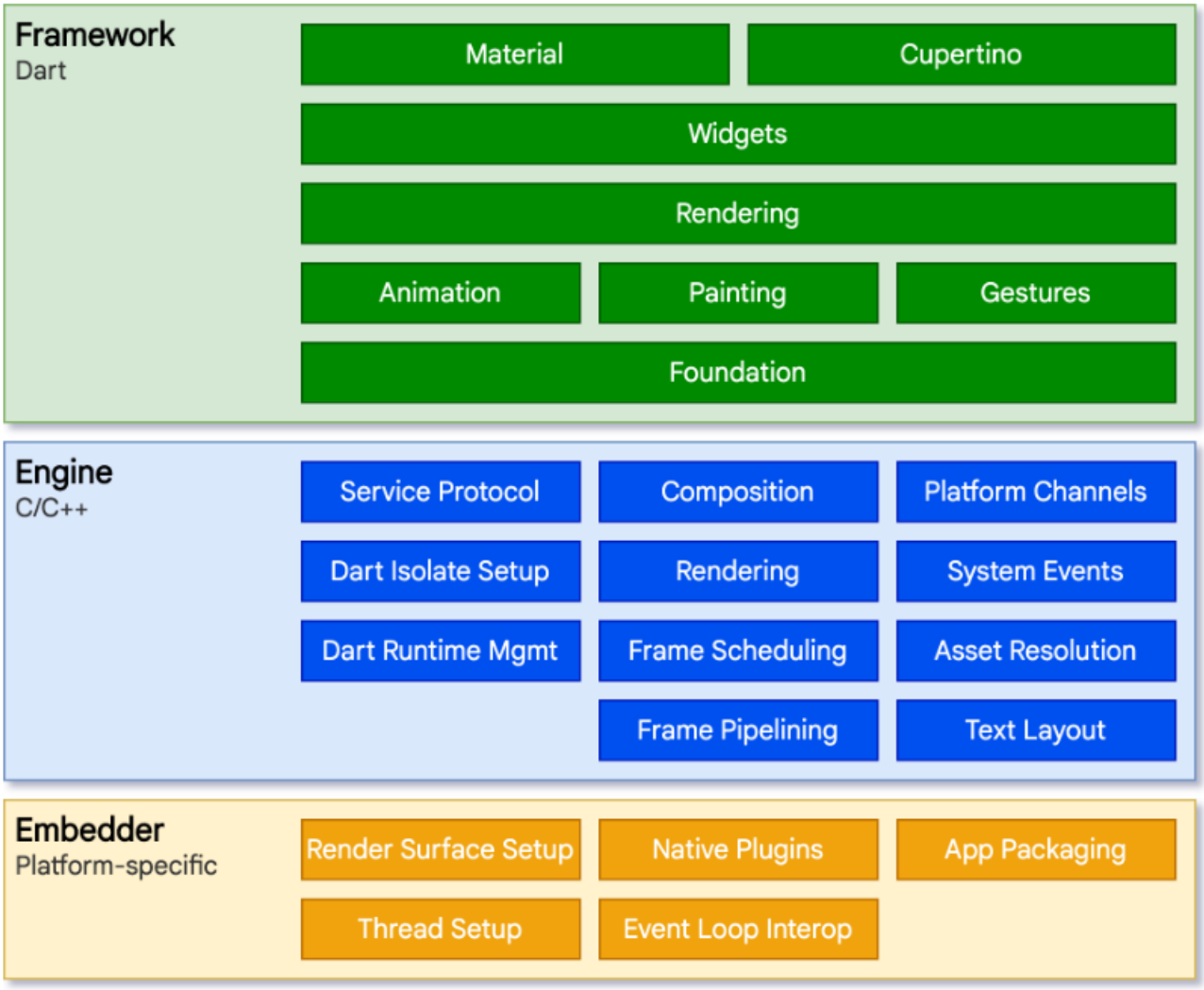
將 Flutter 與其他程式碼整合：有關 Flutter 應用程式可用的不同技術的資訊。

對 Web 的支援：關於瀏覽器環境下 Flutter 特性的摘要。

架構層

#

Flutter 被設計為一個可擴展的分層系統。它作為一系列獨立的庫存在，每個庫都依賴底層。任何層都沒有特權存取下面的層，並且框架層級的每個部分都被設計為可選和可替換的。



對於底層作業系統來說，Flutter 應用程式的打包方式與任何其他原生應用程式相同。特定於平台的嵌入器提供了一個入口點；與底層作業系統協調以存取渲染表面、可存取性和輸入等服務；並管理訊息事件循環。嵌入器是用適合該平台的語言編寫的：目前 Android 為 Java 和 C++，iOS 和 macOS 為 Objective-C/Objective-C++，Windows 和 Linux 為 C++。使用嵌入器，Flutter 程式碼可以作為模組整合到現有應用程式中，或者程式碼可能是應用程式的全部內容。Flutter 包含許多適用於常見目標平台的嵌入器，但也存在其他嵌入器。

Flutter 的核心是 Flutter 引擎，它主要用 C++ 編寫，並支援支援所有 Flutter 應用程式所需的原語。每當需要繪製新幀時，引擎負責對合成場景進行光柵化。它提供了 Flutter 核心 API 的低階實現，包括圖形（透過 iOS 上的 Impeller 以及 Android 和 macOS 上的 Impeller，以及其他平台上的 Skia）文字佈局、檔案和網路 I/O、輔助功能支援、插件架構以及 Dart 運行時和編譯工具鏈。

該引擎透過 dart:ui 暴露給 Flutter 框架，它將底層 C++ 程式碼包裝在 Dart 類別中。該程式庫公開了最低層級的基元，例如用於驅動輸入、圖形和文字渲染子系統的類別。

通常，開發人員透過 Flutter 框架與 Flutter 交互，該框架提供了用 Dart 語言編寫的現代反應式框架。它包括一組豐富的平台、佈局和基礎庫，由一系列層組成。從下到上，我們有：

基本的基礎類別以及建構塊服務，例如動畫、繪畫和手勢，它們在底層基礎上提供常用的抽象。

渲染層提供了處理佈局的抽象化。透過該圖層，您可以建立可渲染物件樹。您可以動態地操作這些對象，樹會自動更新佈局以反映您的變更。

小部件層是一個組合抽象。渲染層中的每個渲染物件在 widgets 層中都有一個對應的類別。此外，小部件層可讓您定義可重複使用的類別的組合。這是引入反應式程式設計模型的層。

Material 和 Cupertino 庫提供了全面的控制集，這些控制項使用小部件層的組合基元來實作 Material 或 iOS 設計語言。

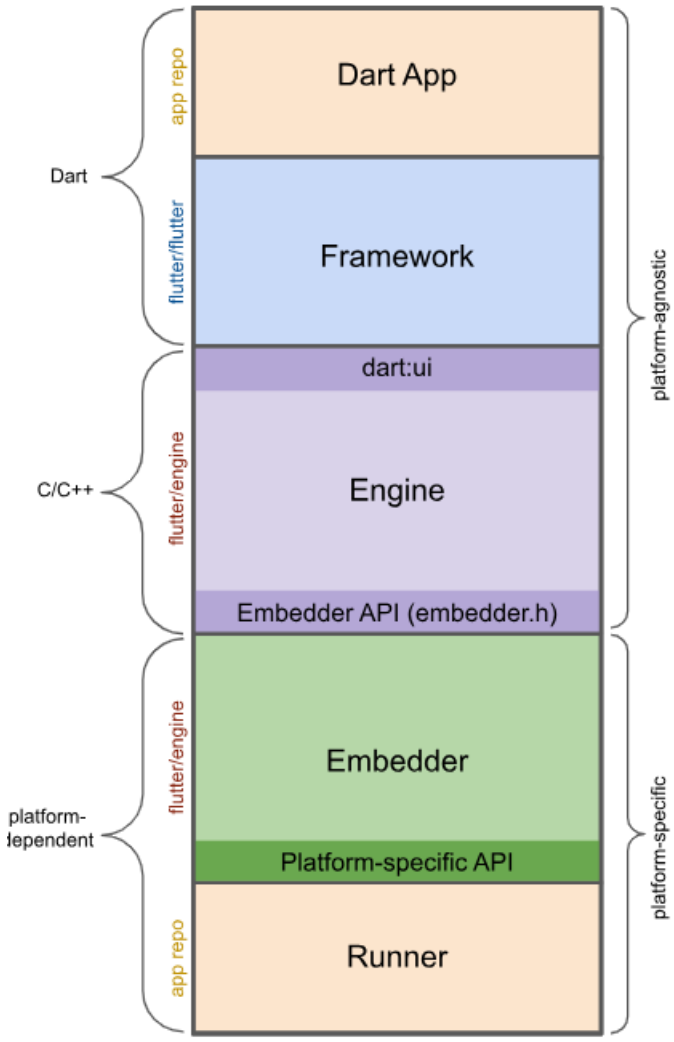
Flutter 框架比較小；開發人員可能使用的許多高級功能都以套件的形式實現，包括相機和 webview 等平台插件，以及基於 Dart 和 Flutter 核心庫構建的與平台無關的功能，如字

元、http 和動畫。其中一些軟體包來自更廣泛的生態系統，涵蓋應用程式內支付、Apple 身份驗證和動畫等服務。

本概述的其餘部分從 UI 開發的反應式範式開始，並廣泛地向下導航。然後，我們描述小部件如何組合在一起並轉換為可以作為應用程式的一部分呈現的物件。我們先描述 Flutter 如何在平台層級與其他程式碼進行互通，然後簡要總結 Flutter 的 Web 支援與其他目標的不同之處。

應用程式剖析

#



下圖概述了由 flutter create 產生的常規 Flutter 應用程式的組成部分。它顯示了 Flutter 引擎在此堆疊中的位置，突出顯示了 API 邊界，並標識了各個部分所在的儲存庫。下面的圖例闡明了一些常用來描述 Flutter 應用程式各個部分的術語。

DART 應用程式

將小部件組合到所需的 UI 中。

實現業務邏輯。

由應用程式開發商擁有。

框架（原始碼）

提供更高層級的 API 來建立高品質的應用程式（例如，小部件、命中測試、手勢檢測、輔助功能、文字輸入）。

將應用程式的小部件樹合成到場景中。

引擎（原始碼）

負責光柵化合成場景。

提供 Flutter 核心 API 的低階實作（例如圖形、文字佈局、Dart 執行時期）。

使用 dart:ui API 將其功能公開給框架。

使用引擎的 Embedder API 與特定平台整合。

嵌入器（原始碼）

與底層作業系統協調以存取渲染表面、可存取性和輸入等服務。

管理事件循環。

公開特定於平台的 API 以將 Embedder 整合到應用程式中。

跑者

將 Embedder 的特定於平台的 API 公開的片段組合成可在目標平台上運行的應用程式包。

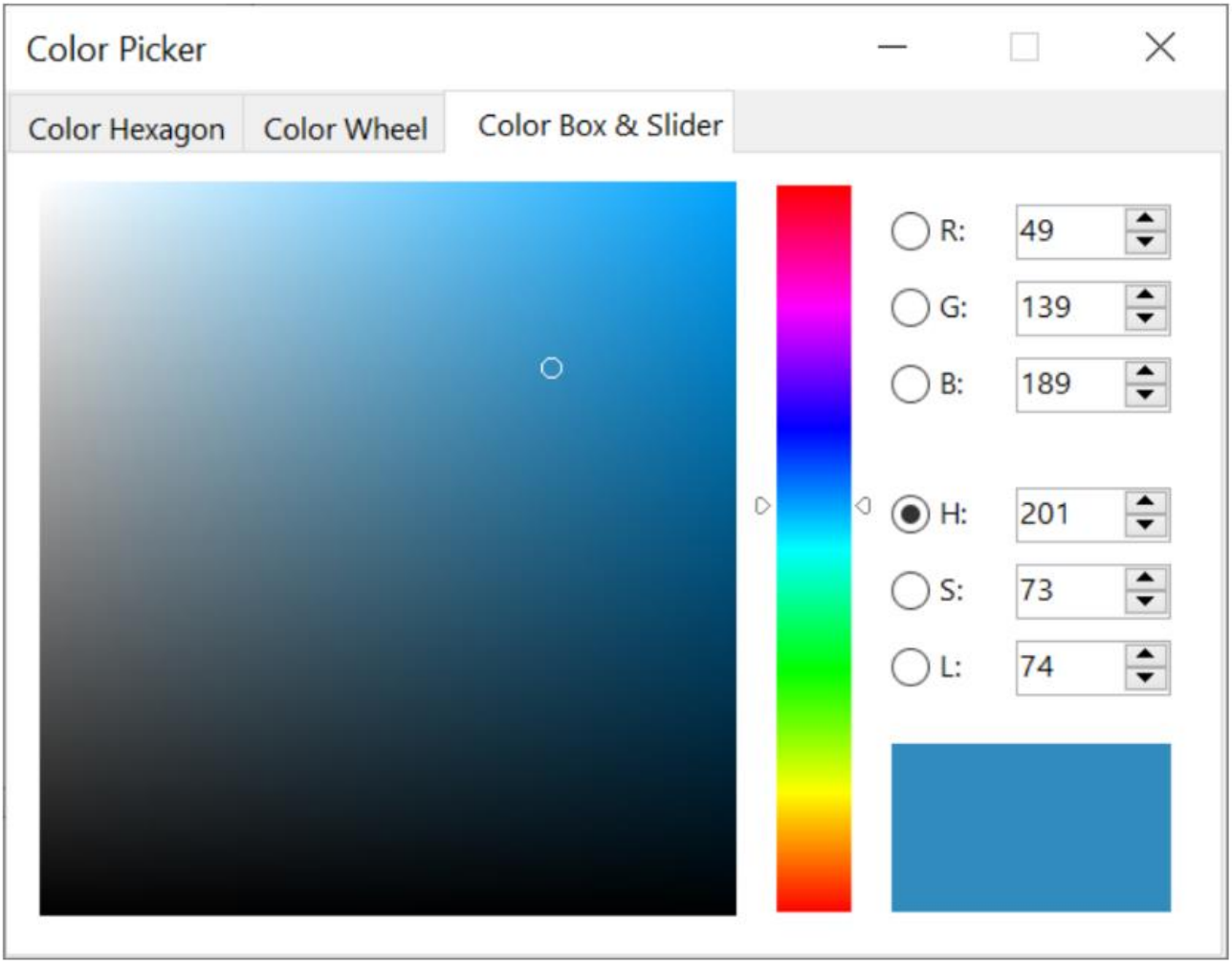
flutter create 產生的應用程式範本的一部分，歸應用程式開發者所有。

反應式使用者介面

#

從表面上看，Flutter 是一個反應式、聲明式的 UI 框架，其中開發者提供了從應用程式狀態到介面狀態的映射，框架承擔了當應用程式狀態發生變化時在運行時更新介面的任務。該模型的靈感來自 Facebook 自己的 React 框架的工作，其中包括對許多傳統設計原則的重新思考。

在大多數傳統的 UI 框架中，使用者介面的初始狀態被描述一次，然後由使用者程式碼在運行時單獨更新，以回應事件。這種方法的一個挑戰是，隨著應用程式複雜性的增加，開發人員需要了解狀態變更如何在整個 UI 中級聯。例如，考慮以下 UI：



有很多地方可以更改狀態：顏色框、色調滑桿、單選按鈕。當使用者與 UI 互動時，變更必須反映在所有其他地方。更糟的是，除非小心謹慎，否則對使用者介面某一部分的微小更改可能會對看似不相關的程式碼片段產生連鎖反應。

解決此問題的一種方法是像 MVC 這樣的方法，其中透過控制器將資料變更推送到模型，然後模型透過控制器將新狀態推送到視圖。然而，這也是有問題的，因為建立和更新 UI 元素是兩個獨立的步驟，很容易不同步。

Flutter 以及其他反應式框架採用了另一種方法來解決這個問題，即明確地將使用者介面與其底層狀態解耦。使用 React 風格的 API，您只需建立 UI 描述，框架負責使用該配置來根據需要建立和/或更新使用者介面。

在 Flutter 中，小部件（類似於 React 中的元件）由用於配置物件樹的不可變類別表示。這些小部件用於管理用於佈局的單獨物件樹，然後用於管理用於合成的單獨物件樹。 Flutter 的核心是一系列機制，用於有效地遍歷樹的修改部分、將物件樹轉換為較低層級的物件樹，以及在這些樹之間傳播變更。

小部件透過重寫 build() 方法來聲明其使用者介面，該方法是將狀態轉換為 UI 的函數：

UI = f(state)

build() 方法在設計上執行速度很快，並且應該沒有副作用，允許框架在需要時呼叫它（可能每個渲染幀調用一次）。

這種方法依賴語言運行時的某些特徵（特別是快速物件實例化和刪除）。幸運的是，Dart 特別適合這項任務。

小部件 Widgets

#

如前所述，Flutter 強調將 widget 作為一個組合單元。小部件是 Flutter 應用程式使用者介面的構建塊，每個小部件都是用戶介面部分的不可變聲明。

小部件根據組合形成層次結構。每個小部件都嵌套在其父級內部，並且可以從父級接收上下文。這個結構一直延伸到根元件（託管 Flutter 應用程式的容器，通常是 MaterialApp 或 CupertinoApp），如這個簡單的範例所示：

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

void main() => runApp(const MyApp());

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: const Text('My Home Page'),
        ),
        body: Center(
          child: Builder(
            builder: (context) {
              return Column(
                children: [
                  const Text('Hello World'),
                  const SizedBox(height: 20),
                  ElevatedButton(
                    onPressed: () {
                      print('Click!');
                    },
```

應用程式透過告訴框架用另一個小部件替換層次結構中的一個小部件來更新其用戶介面，以回應事件（例如用戶互動）。然後，框架會比較新舊小部件，並有效地更新使用者介面。

Flutter 對每個 UI 控制項都有自己的實現，而不是遵循系統提供的實作：例如，iOS Toggle 控制項和 Android 等效控制項都有一個純 Dart 實作。

這種方法有幾個好處：

提供無限的可擴展性。想要 Switch 控制項變體的開發人員可以以任意方式建立一個變體，並且不限於作業系統提供的擴充點。

透過允許 Flutter 一次合成整個場景，無需在 Flutter 程式碼和平台程式碼之間來回轉換，避免了嚴重的效能瓶頸。

將應用程式行為與任何作業系統依賴項分開。即使作業系統更改了其控制項的實現，應用程式在所有版本的作業系統上的外觀和感覺都是相同的。

作品 Composition

#

小部件通常由許多其他小型、單一用途的小部件組成，這些小部件組合在一起可產生強大的效果。

在可能的情況下，設計概念的數量保持在最低限度，同時允許總詞彙量較大。例如，在 widgets 層中，Flutter 使用相同的核心概念（Widget）來表示螢幕繪製、佈局（定位和大小調整）、使用者互動性、狀態管理、主題、動畫和導航。在動畫圖層中，動畫和補間這對概念涵蓋了大部分設計

空間。在渲染層中，RenderObjects 用於描述佈局、繪製、命中測試和可訪問性。在每種情況下，相應的詞彙表最終都會很大：有數百個小部件和渲染對象，以及數十種動畫和補間類型。

類別層次結構故意淺而寬，以最大化可能的組合數量，重點關注每個都擅長做一件事的小型可組合小部件。核心功能是抽象的，甚至像填充和對齊這樣的基本功能也是作為單獨的元件實現的，而不是內建到核心中。 （這也與更傳統的 API 形成對比，在傳統 API 中，填充等功能內置於每個佈局組件的公共核心中。）因此，例如，要使小部件居中，而不是調整名義上的對齊屬性，請將其包裝在 Center 小工具中。

有用於填充、對齊、行、列和網格的小部件。這些佈局小部件沒有自己的視覺表示。相反，它們的唯一目的是控制另一個小部件佈局的某些方面。 Flutter 還包括利用這種組合方法的實用小工具。

例如，Container，一種常用的 widget，由多個負責佈局、繪製、定位和調整大小的 widget 組成。具體來說，Container 由 LimitedBox、ConstrainedBox、Align、Padding、DecoratedBox 和 Transform 小工具組成，正如您閱讀其原始程式碼所能看到的那樣。 Flutter 的一個決定性特徵是您可以深入研究任何小部件的原始程式碼並檢查它。因此，您不必透過子類化 Container 來產生自訂效果，而是可以以新穎的方式組合它和其他小部件，或者只是使用 Container 作為靈感來創建一個新的小部件。

建造小部件 Building widgets

#

如前所述，您可以透過重寫 build() 函數以傳回新的元素樹來確定小部件的視覺表示。該樹以更具

體的方式表示使用者介面中小部件的部分。例如，工具列小工具可能具有傳回某些文字和各種按鈕的水平佈局的建構函數。根據需要，框架遞歸地要求每個小部件構建，直到樹完全由特定的可渲染物件描述。然後，框架將可渲染物件拼接在一起形成可渲染物件樹。

小部件的建構函數應該沒有副作用。每當要求建構函數時，小部件都應該傳回一個新的小部件樹[1]，無論小部件先前傳回什麼。該框架執行繁重的工作來確定需要根據渲染物件樹呼叫哪些建置方法（稍後將更詳細地描述）。有關此過程的更多資訊可以在 Inside Flutter 主題中找到。

在每個渲染幀上，Flutter 可以透過呼叫該 widget 的 build() 方法來重新建立狀態已變更的 UI 部分。因此，建置方法應該快速返回非常重要，並且繁重的計算工作應該以某種非同步方式完成，然後儲存為狀態的一部分以供建置方法使用。

雖然方法相對簡單，但這種自動比較非常有效，可以實現高效能的互動式應用程式。而且，建立函數的設計透過專注於聲明小部件的構成來簡化您的程式碼，而不是將使用者介面從一種狀態更新到另一種狀態的複雜性。

小部件狀態 Widget state

#

該框架引入了兩類主要的小部件：有狀態小部件和無狀態小部件。

許多小部件沒有可變狀態：它們沒有任何隨時間變化的屬性（例如圖標或標籤）。這些小部件是 StatelessWidget 的子類別。

但是，如果小部件的獨特特徵需要根據使用者互動或其他因素進行更改，則該小部件是有狀態的。例如，如果某個小部件有一個計數器，每當使用者點擊按鈕時該計數器就會遞增，那麼計數器的值就是該小部件的狀態。當該值發生變化時，需要重新建構小部件以更新其 UI 部分。這些小部件是 StatefulWidget 的子類，並且（因為小部件本身是不可變的）它們將可變狀態儲存在一個單獨的 State 子類中。 StatefulWidgets 沒有建構方法；相反，他們的使用者介面是透過 State 物件建構的。

每當您改變 State 物件（例如，透過遞增計數器）時，您必須呼叫 setState() 來通知框架透過再次呼叫 State 的 build 方法來更新使用者介面。

擁有單獨的狀態和小部件物件可以讓其他小部件以完全相同的方式處理無狀態和有狀態小部件，而不必擔心丟失狀態。父級不需要保留子級來保留其狀態，而是可以隨時建立子級的新實例，而不會遺失子級的持久狀態。此框架會在適當的時候完成尋找並重複使用現有狀態物件的所有工作。

狀態管理

#

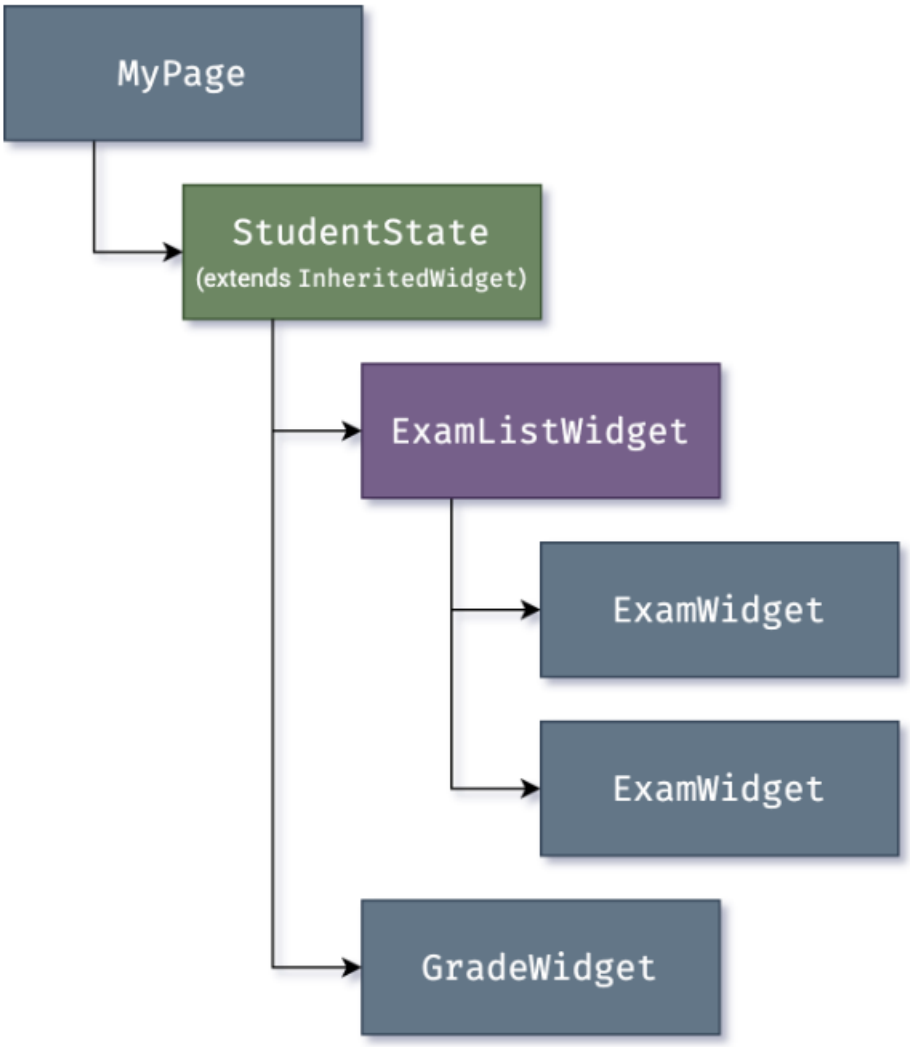
那麼，如果許多小部件可以包含狀態，那麼狀態是如何在系統中管理和傳遞的呢？

與任何其他類別一樣，您可以在小部件中使用構造函數來初始化其數據，因此 build() 方法可以確保使用其所需的數據實例化任何子小部件：

```dart
@override
Widget build(BuildContext context) {
   return ContentWidget(importantState);
}
```

其中 importantState 是包含對 Widget 重要的狀態的類別的佔位符。

然而，隨著小部件樹變得越來越深，在樹層次結構中上下傳遞狀態資訊變得很麻煩。因此，第三種小部件類型 InheritedWidget 提供了一種從共享祖先獲取資料的簡單方法。您可以使用 InheritedWidget 建立狀態小部件，該狀態小部件包裝小部件樹中的共同祖先，如本範例所示：



Whenever one of the ExamWidget or GradeWidget objects needs data from StudentState, it can now access it with a command such as:

```
final studentState = StudentState.of(context);
```

of(context) 呼叫取得建構上下文（目前小部件位置的句柄），並傳回樹中與 StudentState 類型相符的最近的祖先。 InheritedWidgets 還提供 updateShouldNotify() 方法，Flutter 呼叫該方法來確定狀態變更是否應觸發使用它的子視窗小部件的重建。

Flutter 本身廣泛使用 InheritedWidget 作為共享狀態框架的一部分，例如應用程式的視覺主題，其中包括在整個應用程式中普遍存在的顏色和類型樣式等屬性。 MaterialApp build() 方法在建構時在樹中插入一個主題，然後在層次結構的更深處，小部件可以使用 .of() 方法來尋找相關主題資料。

For example,

```
Container(
   color: Theme.of(context).secondaryHeaderColor,
   child: Text(
      'Text Theme.of(context).textTheme.titleLarge,
      ),
);
```