



Flutter Cross-Platform

唐婷

业务中台前端技术部

2021-10-29

Flutter 核心技术

Widget

状态管理

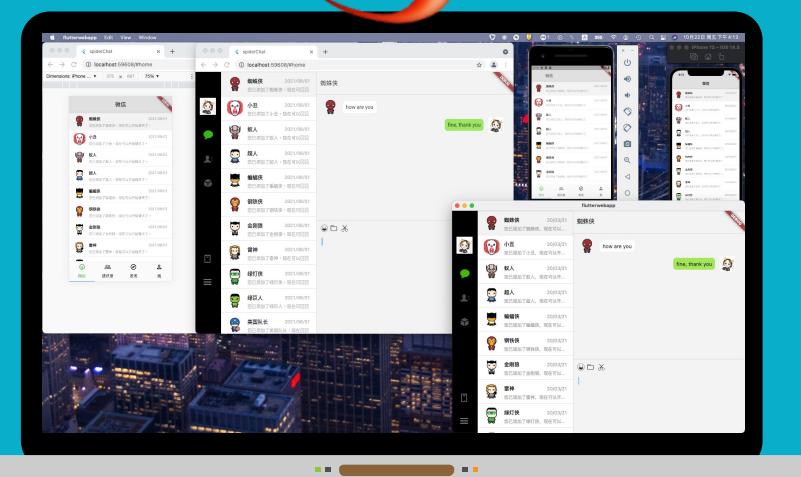
路由管理

依赖管理

调试

错误监控

混合开发



Part

1

Flutter 核心技术

Flutter 架构

Flutter 渲染机制

Dart

Event Loop

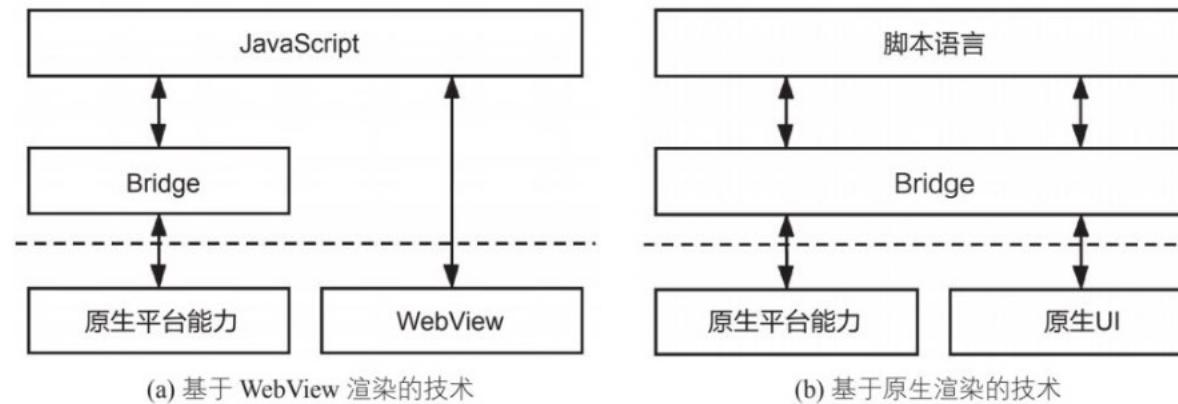
Hot Reload 原理

1、Web 容器时代

以 Hybrid、小程序为代表的基于 WebView 渲染的技术。

2、泛 Web 容器时代

采用类 Web 标准进行开发，在运行时把绘制和渲染交由原生系统接管的技术，代表：React Native、Weex。

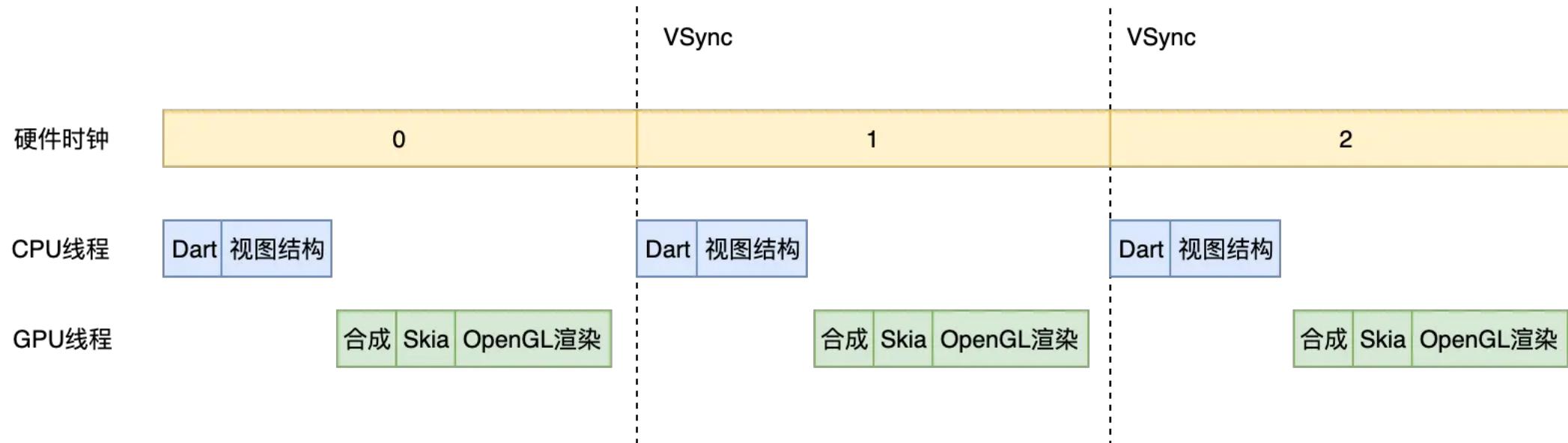


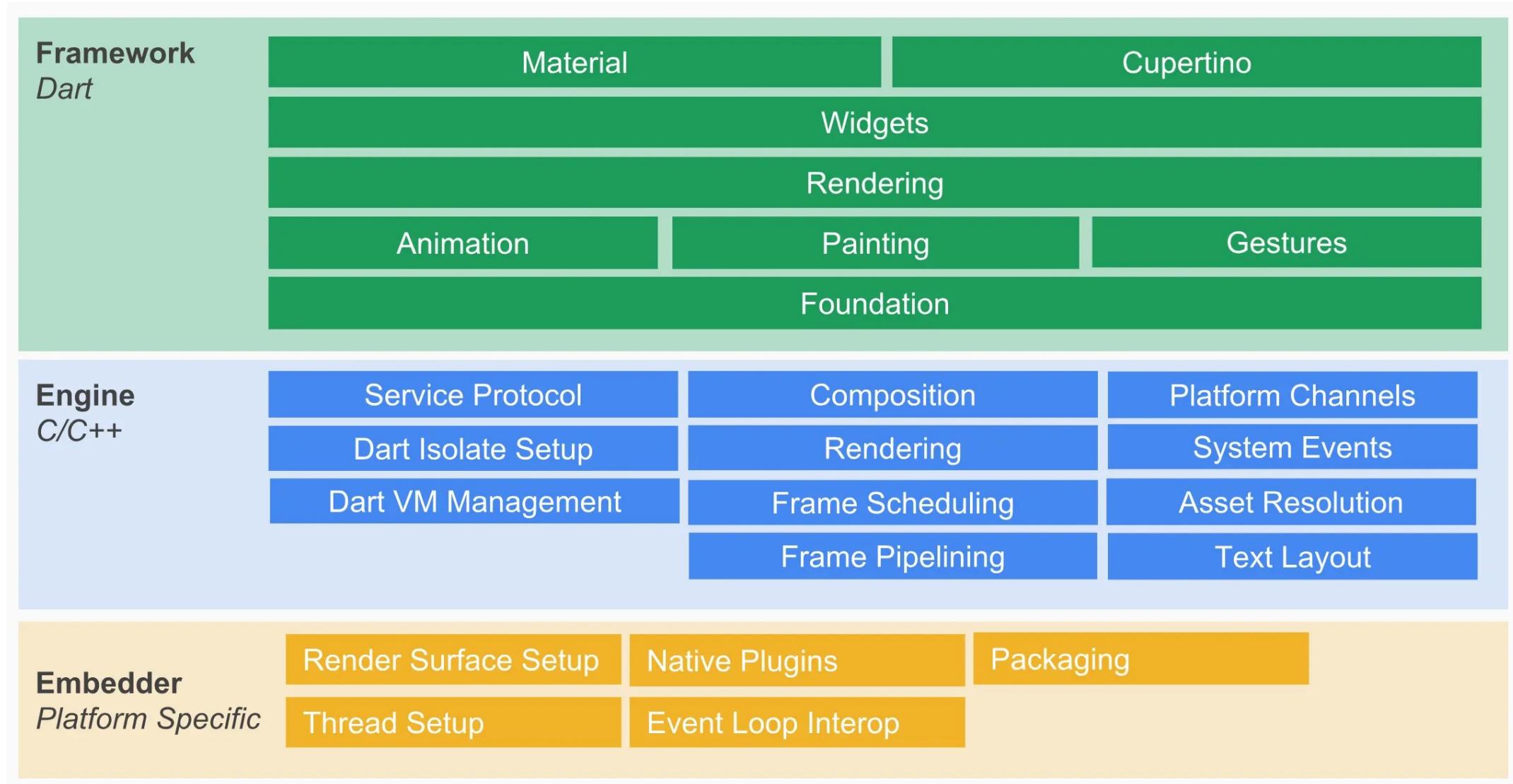
3、自绘引擎时代

自带渲染引擎，业务逻辑到功能呈现的多端高度一致的渲染体验。代表：flutter。

What is Flutter?

Flutter is Google's UI toolkit for building beautiful, natively compiled applications for **mobile**, **web**, **desktop**, and **embedded** devices from a single codebase.





Flutter 工程实际上就是一个同时内嵌了 Android 和 iOS 原生子工程的父工程。

在 lib 目录下进行 Flutter 代码的开发，而某些特殊场景下的原生功能，则在对应的 Android 和 iOS 工程中提供相应的代码实现，供对应的 Flutter 代码引用。

Flutter 会将相关的依赖和构建产物注入这两个子工程，最终集成到各自的项目中。开发的 Flutter 代码，最终则会以原生工程的形式运行。

```
flutter_app
├── android
├── build
├── ios
└── lib
    └── main.dart
└── test
└── flutter_app_demo.iml
└── pubspec.lock
└── pubspec.yaml
```

- 包含Android特定文件的Android子工程
- Android和iOS的构建产物
- 包含iOS特定文件的iOS子工程
- Flutter应用源文件目录
- 程序运行入口文件
- 测试文件
- 工程配置文件
- 记录当前项目实际依赖信息的文件
- 管理第三方库及资源的配置文件

- 创建应用程序：

```
flutter create myapp
```

- 启动项目：

移动端

```
flutter devices  
flutter run
```

桌面端： macos / linux / windows

```
flutter config --enable-<platform>-desktop  
flutter run -d <platform>
```

web端

```
flutter config --enable-web  
flutter run -d chrome --web-renderer 有三个选项： auto, html, canvaskit.
```

- 打包：不同平台的设置启动图标，混淆代码，app签名，等一系列配置。

```
flutter build xx
```

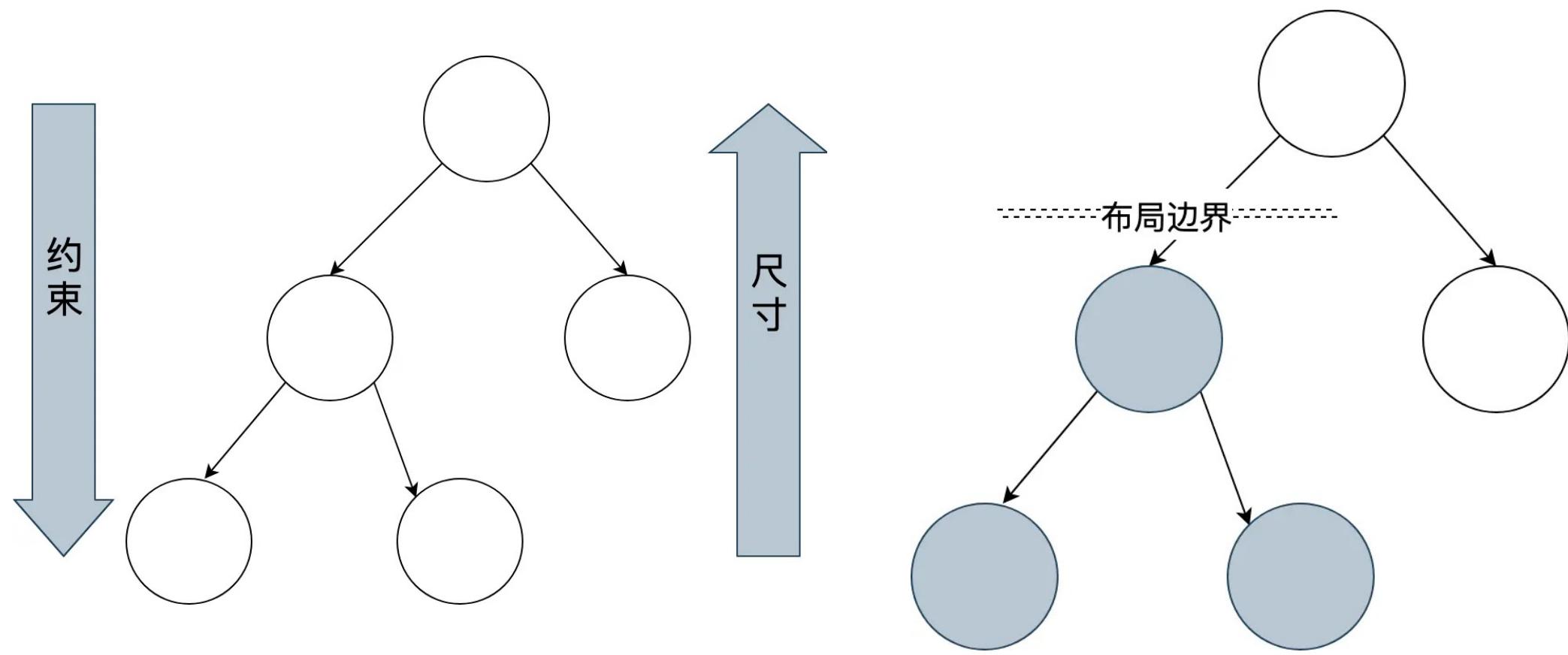
官方文档：<https://flutter.dev/docs/deployment/obfuscate>

①	User input	Responses to input gestures (keyboard, touchscreen, etc.)
②	Animation	User interface changes triggered by the tick of a timer
③	Build	App code that creates widgets on the screen
④	Layout	Positioning and sizing elements on the screen
⑤	Paint	Converting elements into a visual representation
⑥	Composition	Overlaying visual elements in draw order
⑦	Rasterize	Translating output into GPU render instructions

RENDERING

深度优先机制遍历渲染树。

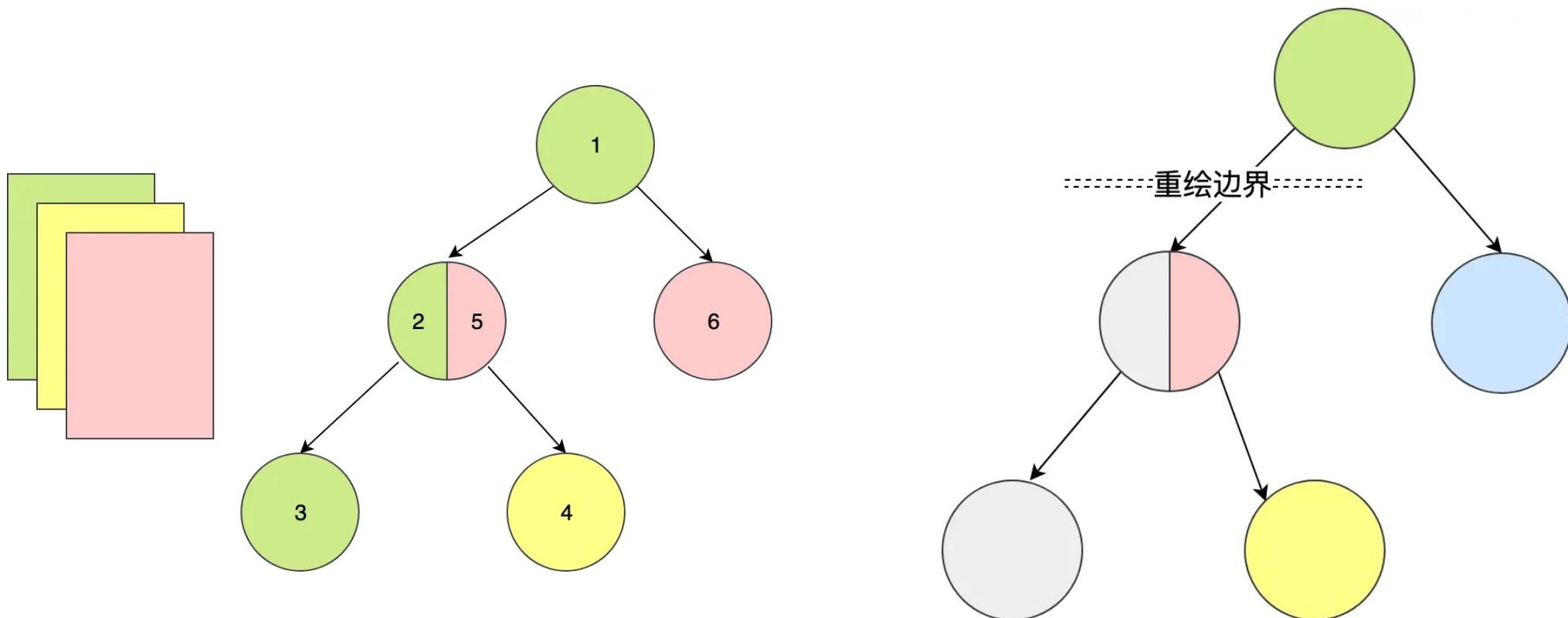
- 在布局过程中，树中的每个渲染对象都会接收父对象的布局约束参数，决定自己的大小，然后父对象按照控件逻辑决定各个子对象的位置，完成布局过程。
- 为了防止因子节点发生变化而导致整个控件树重新布局，Flutter 加入了一个机制——布局边界（Relayout Boundary），当边界内的任何对象发生重新布局时，不会影响边界外的对象。



绘制过程也是深度优先遍历，总是先绘制自身，再绘制子节点。

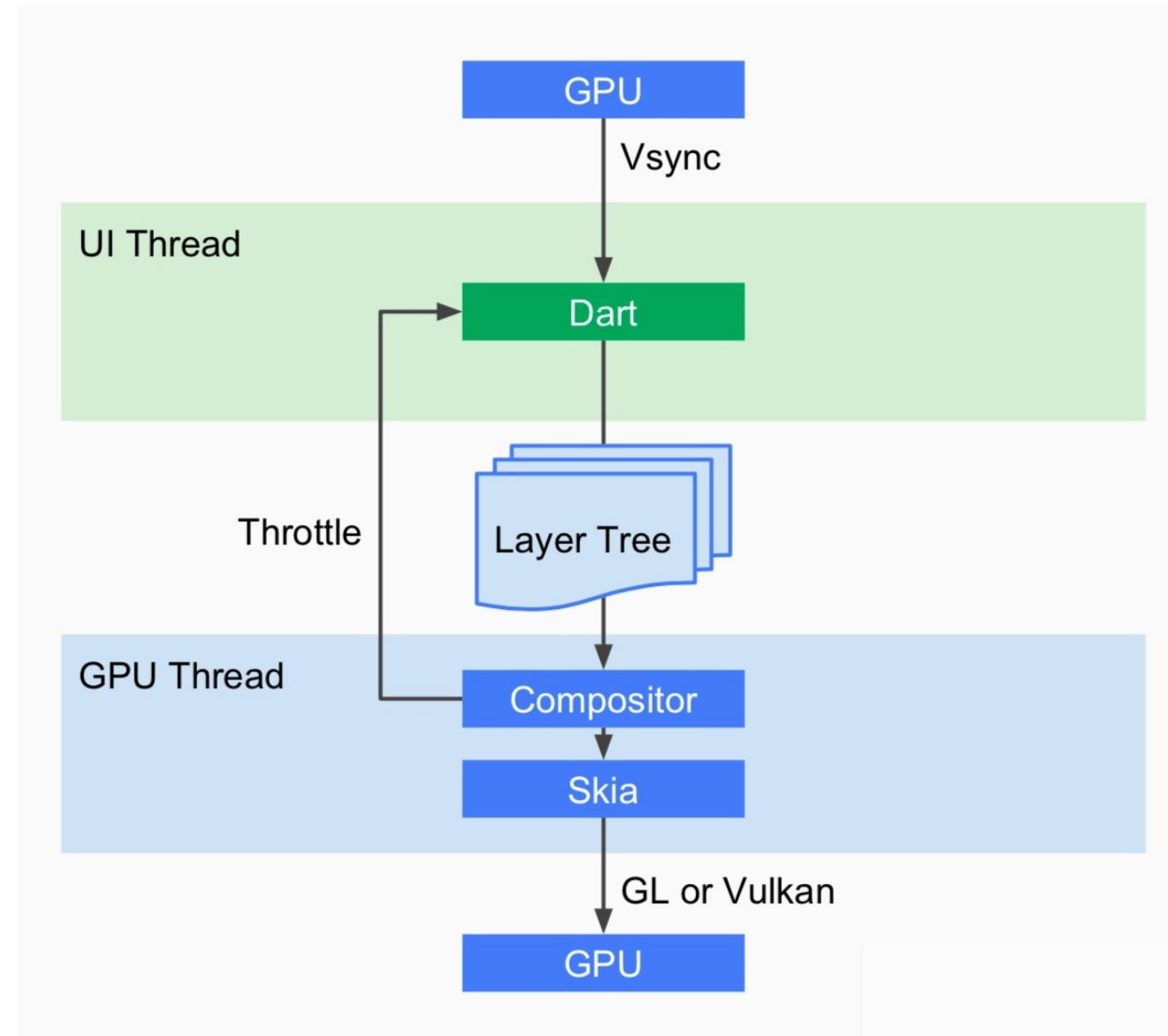
由于一些其他原因（比如，视图手动合并）导致 2 的子节点 5 与它的兄弟节点 6 处于了同一层，这样会导致当节点 2 需要重绘的时候，与其无关的节点 6 也会被重绘，带来性能损耗。

为了解决这一问题，提出了重绘边界（Repaint Boundary）。在重绘边界内，Flutter 会强制切换新的图层，这样就可以避免边界内外的互相影响，引起不必要的重绘。



合成：所有的图层根据大小、层级、透明度等规则计算出最终的显示效果，将相同的图层归类合并，简化渲染树，提高渲染效率。

渲染：Flutter 会将几何图层数据交由 Skia 引擎加工成二维图像数据，最终交由 GPU 进行渲染，完成界面的展示。



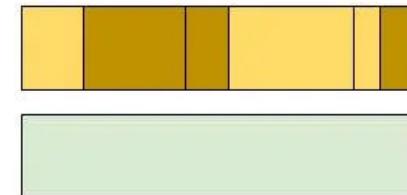
- Dart 同时支持即时编译 JIT 和事前编译 AOT。
- 单线程模型：Dart 中并没有线程，只有 Isolate（隔离区）。Isolates 之间不会共享内存，就像几个运行在不同进程中的 worker，通过事件循环在事件队列上传递消息通信。
- 内存分配：创建对象时只需要在堆上移动指针，内存增长始终是线性的，省去了查找可用内存的过程。
- 垃圾回收：“新生代”，“老生代”。专门设计了调度器，当检测到空闲且没有用户交互时进行GC操作。

新对象被分配到连续、可用的内存空间，包含两个部分：活跃区和非活跃区，新对象在创建时被分配到活跃区、一旦填充完毕，仍然活跃的对象会被移动到非活跃区，不再活跃的对象会被清理掉，然后非活跃区变成活跃区，活跃区变成非活跃区，以此循环。

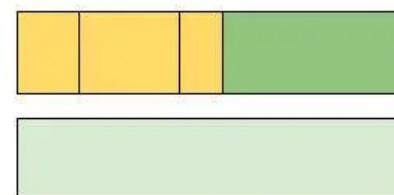
1: Semispaces



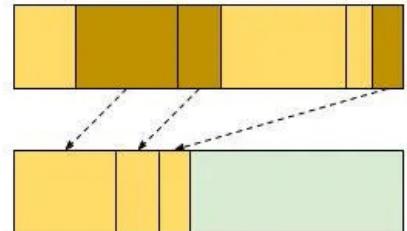
4: Live objects determined



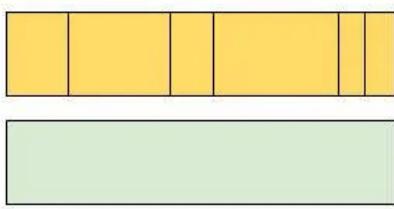
2: Objects allocated in active



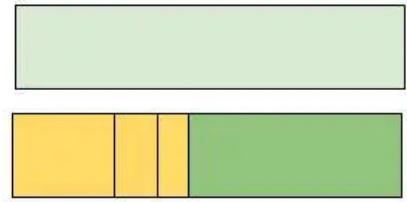
5: Live objects moved



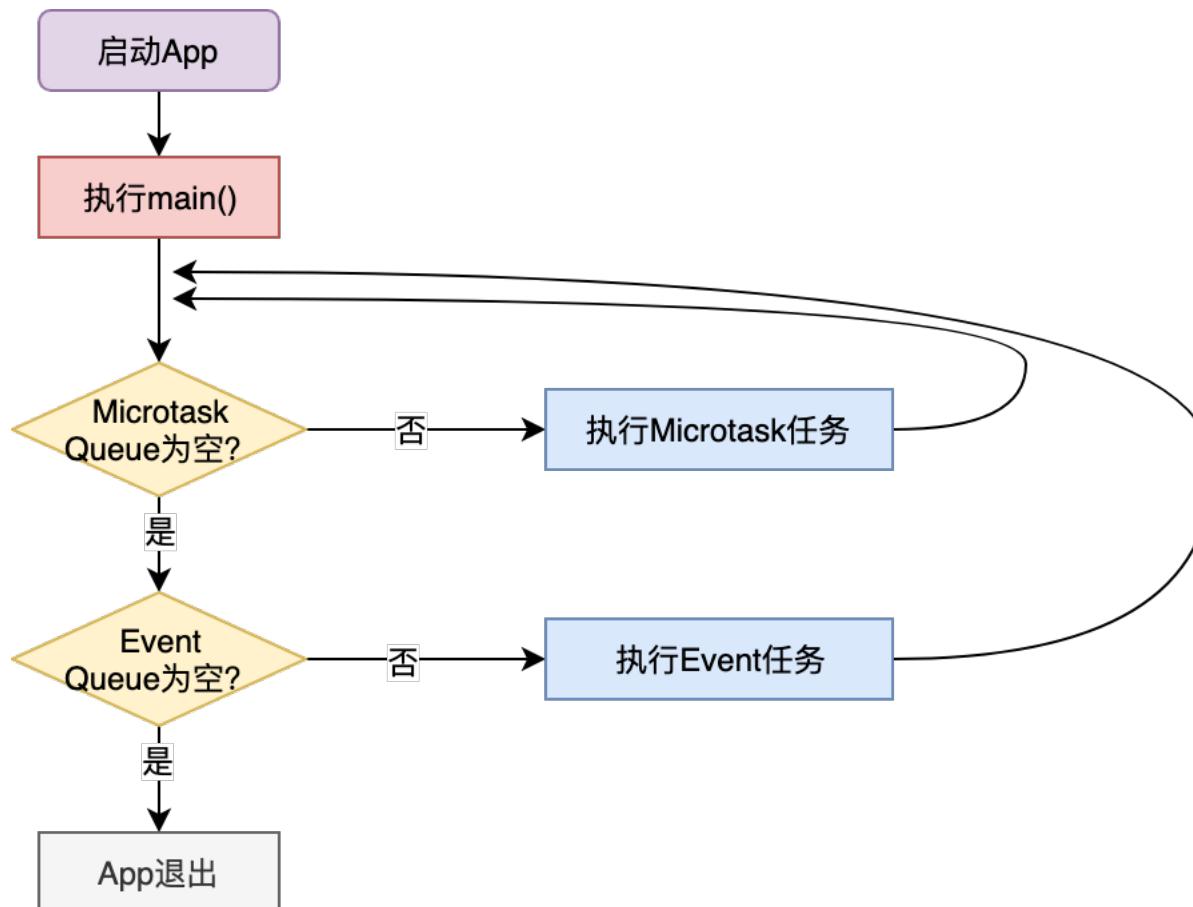
3: Active space fills



6: Spaces swap active state



在 Dart 中，有两个队列，一个事件队列（Event Queue），另一个则是微任务队列（Microtask Queue）。在每一次事件循环中，Dart 总是先去第一个微任务（手势识别、文本输入、滚动视图等）队列中查询是否有可执行的任务，如果没有，才会处理后续的事件队列的流程。

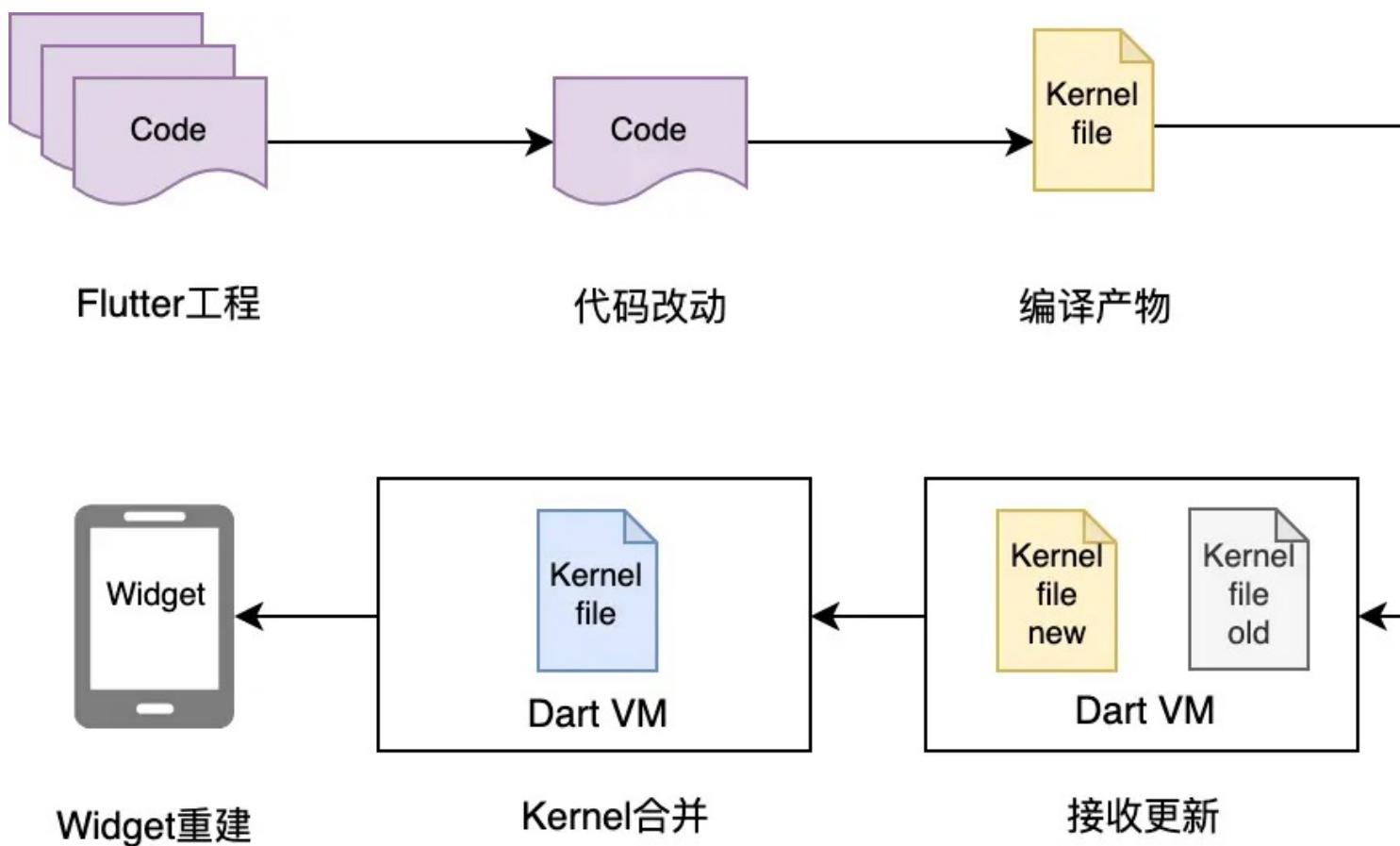


Future：把一个函数体放入 Future，就完成了从同步任务到异步任务的包装。

Isolate：为了进一步利用多核 CPU，将 CPU 密集型运算进行隔离，Dart 也提供了多线程机制，即 Isolate，之间不共享任何资源，只能依靠消息机制通信。

Flutter 的热重载是基于 JIT 编译模式的代码增量同步。JIT 属于动态编译，将 Dart 代码编译成生成中间代码，让 Dart VM 在运行时解释执行，因此可以动态更新中间代码实现增量同步。

在收到代码变更后，并不会让 App 重新启动执行，而只会触发 Widget 树的重新绘制，因此可以保持改动前的状态，这就大大节省了调试复杂交互界面的时间。



1. 代码出现编译错误；
2. Widget 状态无法兼容: 比如将某个类的定义从 StatelessWidget 改为 StatefulWidget 时, 热重载就会直接报错;
3. 全局变量和静态属性的更改: 全局变量和静态属性都被视为状态, 在第一次运行应用程序时, 会将它们的值设为初始化语句的执行结果, 因此在热重载期间不会重新初始化;
4. main 方法里的更改: 由于热重载之后只会根据原来的根节点重新创建控件树, 因此 main 函数的任何改动并不会在热重载后重新执行;
5. initState 方法里的更改: initState 方法是 Widget 状态的初始化方法, 这个方法里的更改会与状态保存发生冲突, 因此热重载后不会产生效果。;
6. 枚举和泛类型更改: 枚举和泛型也被视为状态, 因此对它们的修改也不支持热重载。

Part

2 Widget

经典控件

布局控件



Text: 展示单一样式的文本

Text.rich: 支持多种混合样式的富文本

Hello world

Hello world! Welcome to Flutter. This part is f...

Home: <https://flutterchina.club>

```
Text(  
  "Hello world",  
  style: TextStyle(  
    color: Colors.blue,  
    fontSize: 18.0,  
    fontFamily: "Courier",  
    backgroundColor: Colors.yellow,  
    decoration: TextDecoration.underline,  
    decorationStyle: TextDecorationStyle.dashed),  
,  
Text(  
  "Hello world! Welcome to Flutter. This part is for Text Widget!",  
  maxLines: 1,  
  overflow: TextOverflow.ellipsis,  
,  
Text.rich(TextSpan(children: [  
  TextSpan(text: "Home: "),  
  TextSpan(  
    text: "https://flutterchina.club",  
    style: TextStyle(color: Colors.blue),  
  ),  
]),
```

基础 widget—图片

AssetImage: 加载本地资源图片,

或者Image.asset;

NetworkImage: 加载网络图片,

或者Image.network;

FadeInImage: 控件提供了图片占位的功能，并且支持在图片加载完成时淡入淡出的视觉效果；

CachedNetworkImage: 除了支持图片缓存外，还提供了加载过程占位与加载错误占位；



```
Image.asset('images/zhihuxia.png', width: 100.0),  
Image.network('https://picsum.photos/250?image=1', width: 100.0),  
FadeInImage.memoryNetwork(  
    placeholder: kTransparentImage,  
    image: 'https://picsum.photos/250?image=2',  
    width: 100.0),  
FadeInImage.assetNetwork(  
    placeholder: 'assets/loading.gif',  
    image: 'https://picsum.photos/250?image=9',  
    width: 100.0),  
CachedNetworkImage(  
    imageUrl: "http://xxx/xxx/jpg",  
    placeholder: (context, url) => CircularProgressIndicator(),  
    errorWidget: (context, url, error) => Icon(Icons.error),  
    width: 100.0)
```

MaterialButton：默认按钮，扁平，背景透明。按下后，会有背景色。

RaisedButton：“漂浮”按钮，带有阴影和背景。按下后，阴影会变大。

FlatButton：扁平按钮，默认背景透明。按下后，会有背景色，与MaterialButton一致。

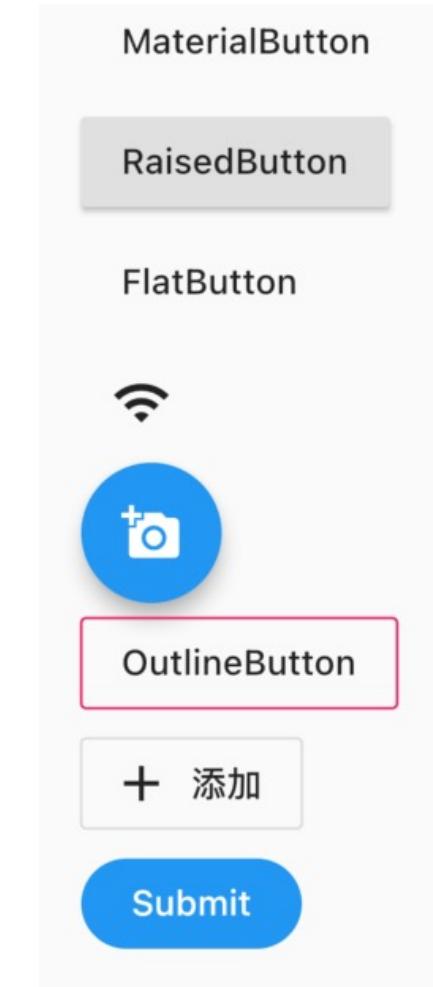
IconButton：图标按钮，只能是纯图标，按钮不可展示文案。

FloatingActionButton：浮动按钮，可显示文字和图标，二者选其一。

OutlineButton：外边框按钮，可设置按钮外边框颜色。

ButtonBar：水平布局的按钮容器，可放置多个Button或Text。

IButton.icon()：带图标文字混合按钮，RaisedButton、FlatButton、OutlineButton都有一个icon 构造函数，它可以轻松创建带图标和文字的按钮。

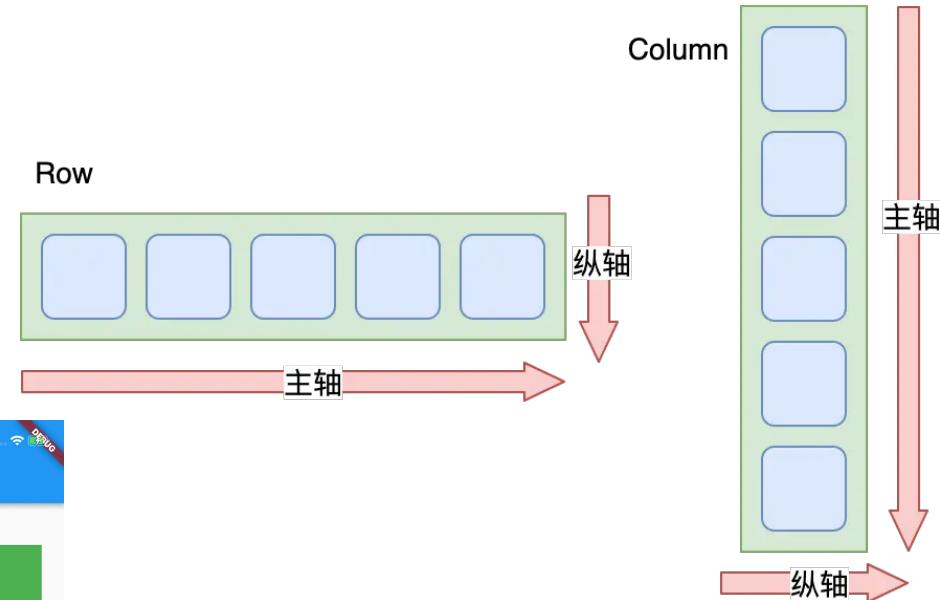
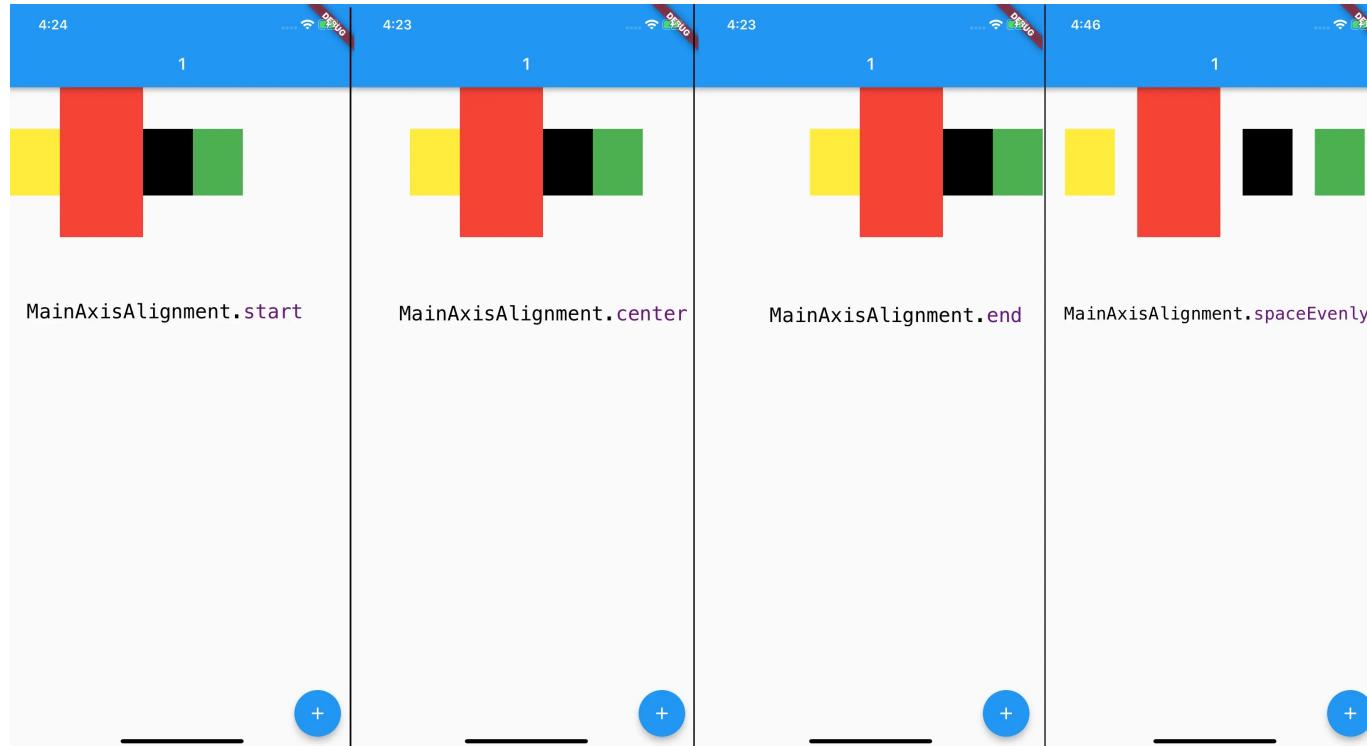


布局 widget—多子 Widget 布局

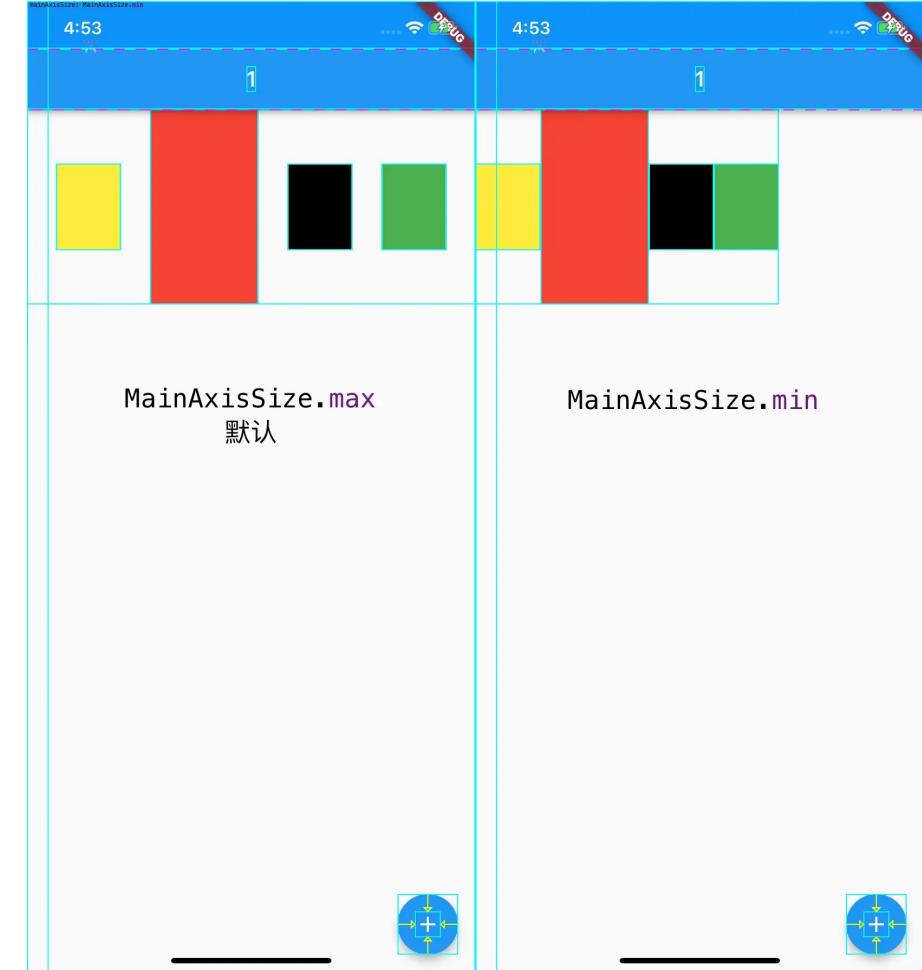
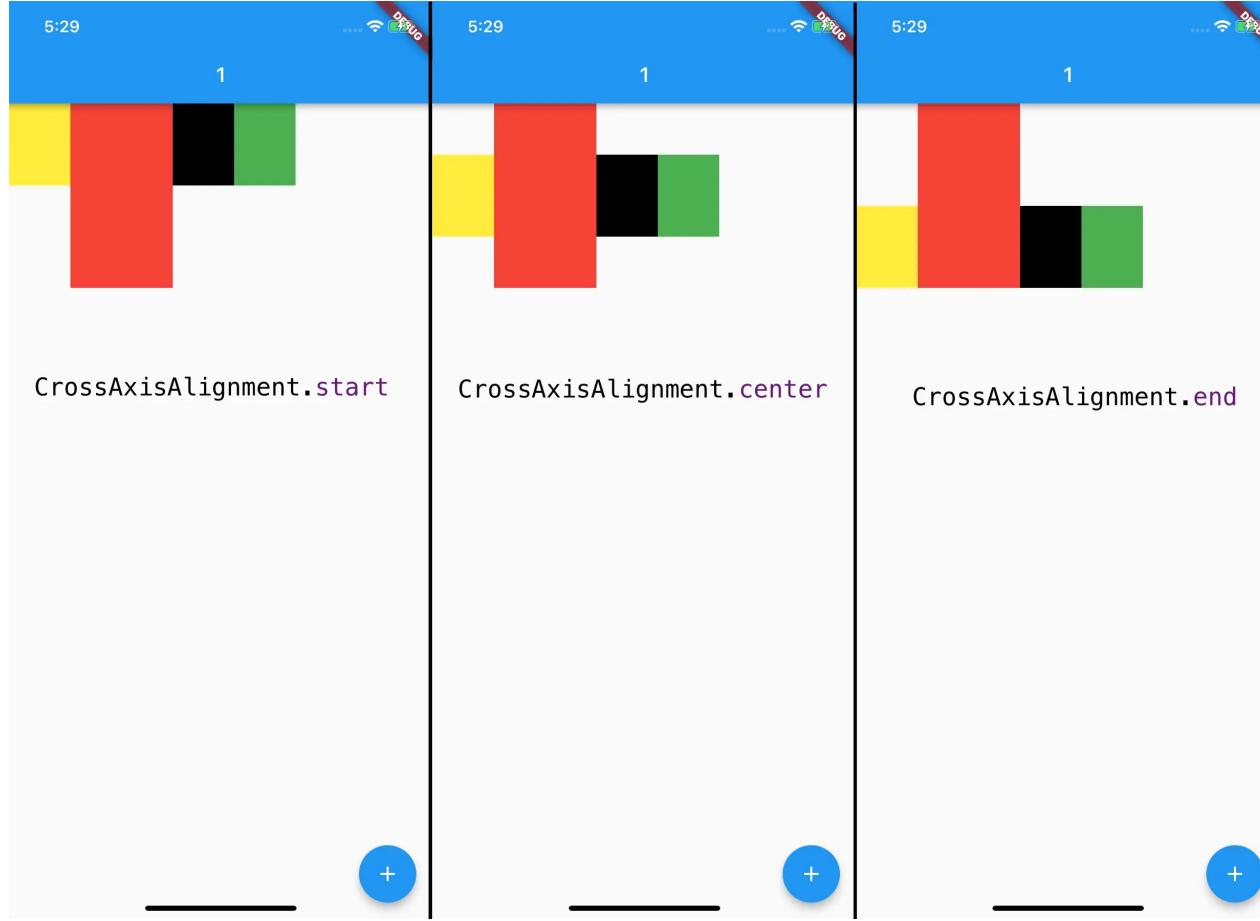
Row: 将子 Widget 按行水平排列

Column: 按列垂直排列的

Expanded: 分配这些子 Widget 在行 / 列中剩余空间



布局 widget—多子 Widget 布局



层叠 Widget 布局：Stack 与 Positioned

Stack 容器与前端中的绝对定位、Android 中的 Frame 布局非常类似，子 Widget 之间允许叠加，还可以根据父容器上、下、左、右四个角的位置来确定自己的位置。Positioned 则提供了设置子 Widget 位置的能力。

```
Stack(  
    children: <Widget>[  
        Container(color: Colors.yellow, width: 300, height: 300),  
        Positioned(  
            left: 18.0,  
            top: 18.0,  
            child: Container(color: Colors.green, width: 50, height: 50),  
        ),  
        Positioned(  
            left: 18.0,  
            top: 70.0,  
            child: Text("Stack提供了层叠布局的容器"),  
        )  
    ],  
)
```

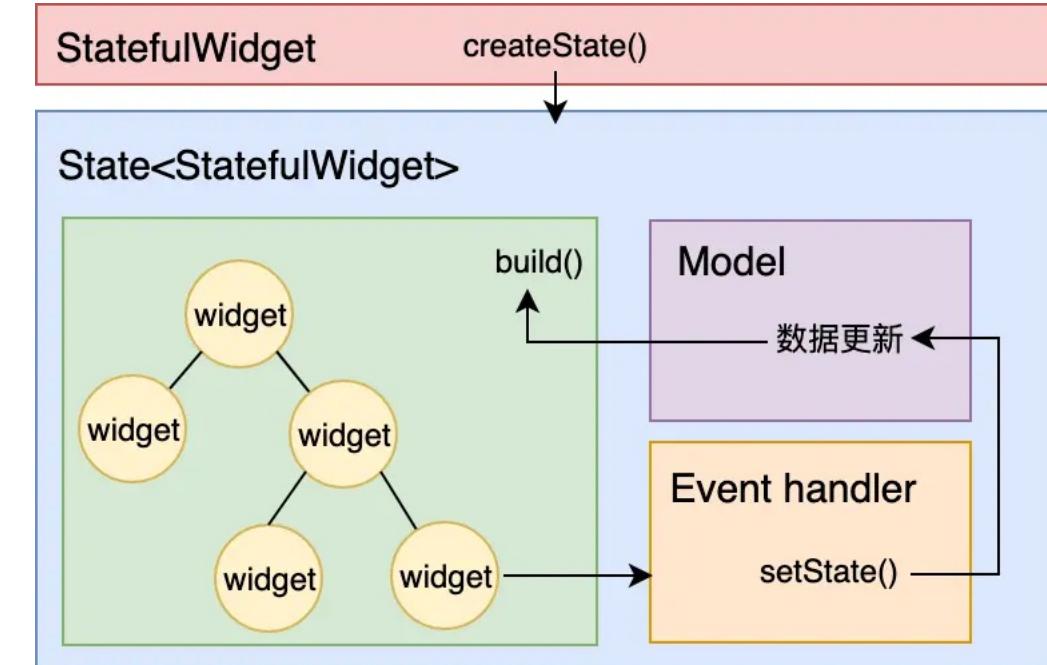
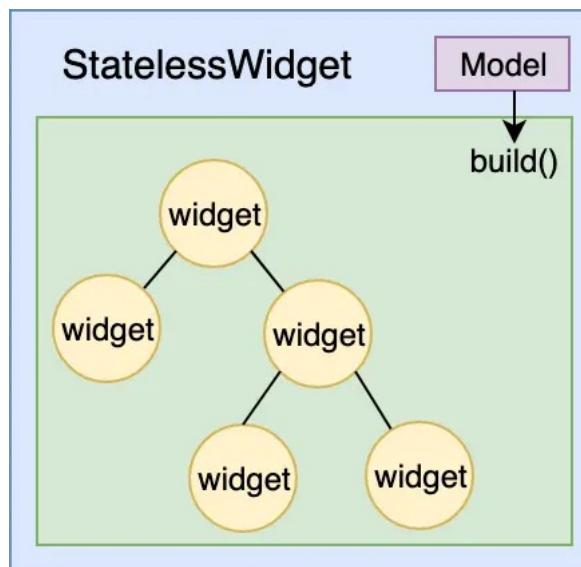
Part

3 状态管理



Widget 有 StatelessWidget 和 StatefulWidget 两种类型。
 StatefulWidget 应对有交互、需要动态变化视觉效果的场景。
 StatelessWidget 则用于处理静态的、无状态的视图展示。

Flutter 的视图开发是声明式的，其核心设计思想就是将视图和数据分离，这与 React 的设计思路完全一致。

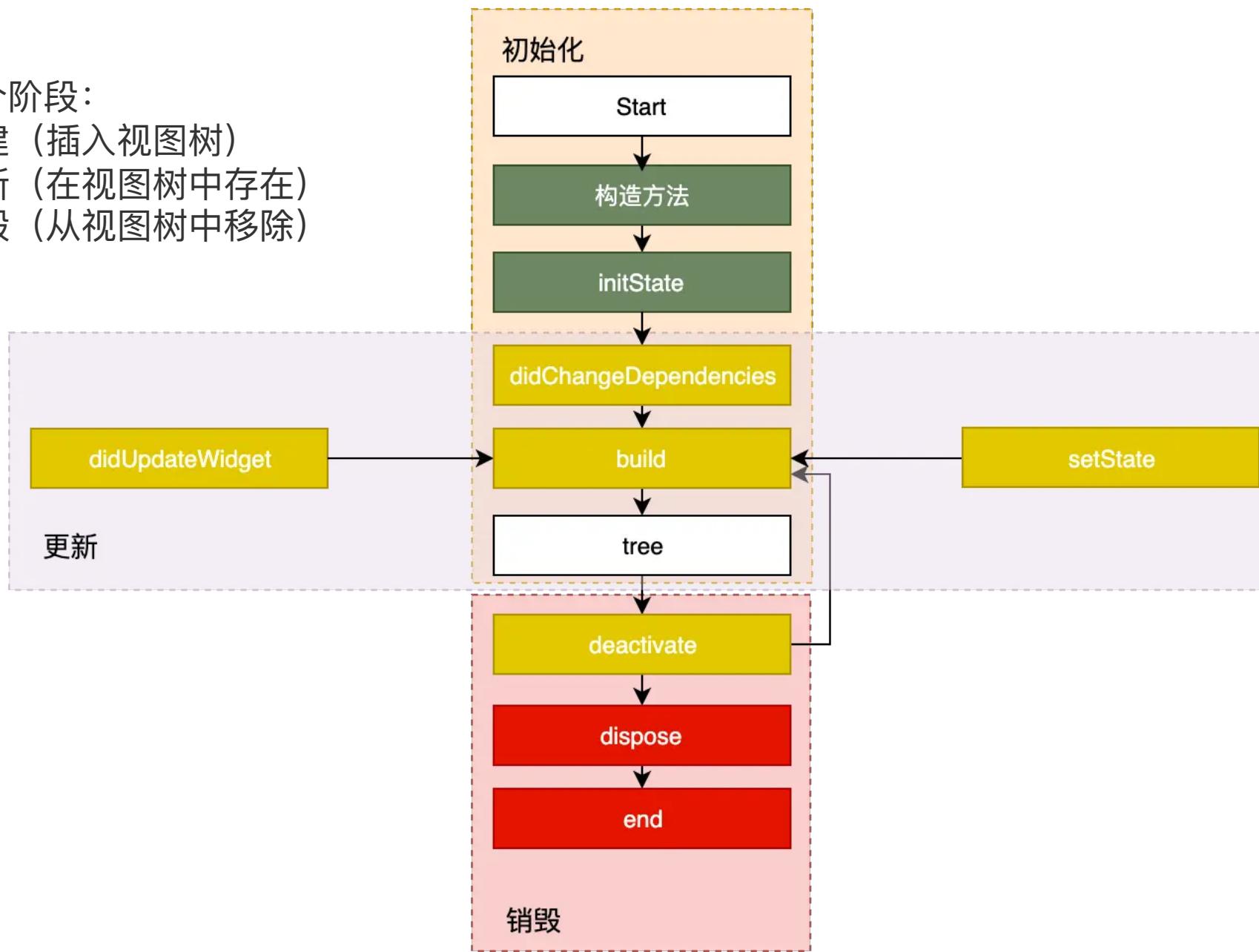


3 个阶段：

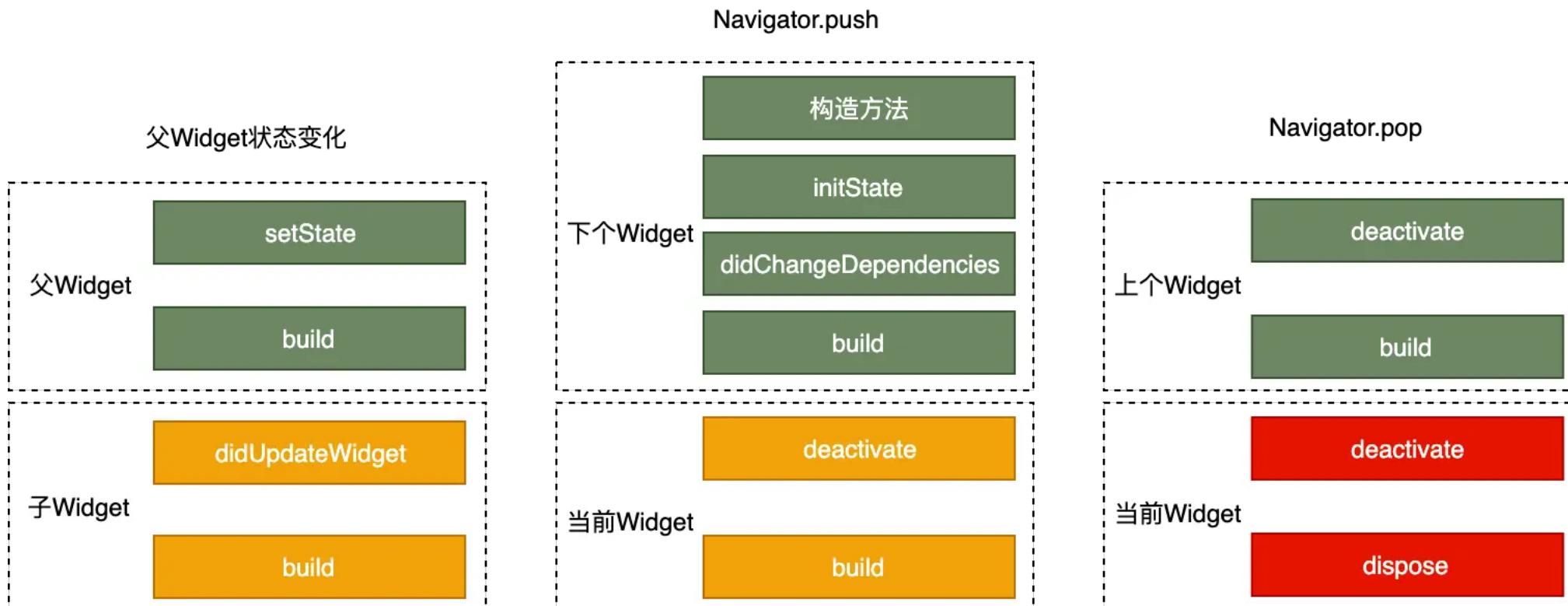
创建（插入视图树）

更新（在视图树中存在）

销毁（从视图树中移除）



左边部分展示了当父 Widget 状态发生变化时，父子双方共同的生命周期；而中间和右边部分则描述了页面切换时，两个关联的 Widget 的生命周期函数是如何响应的。

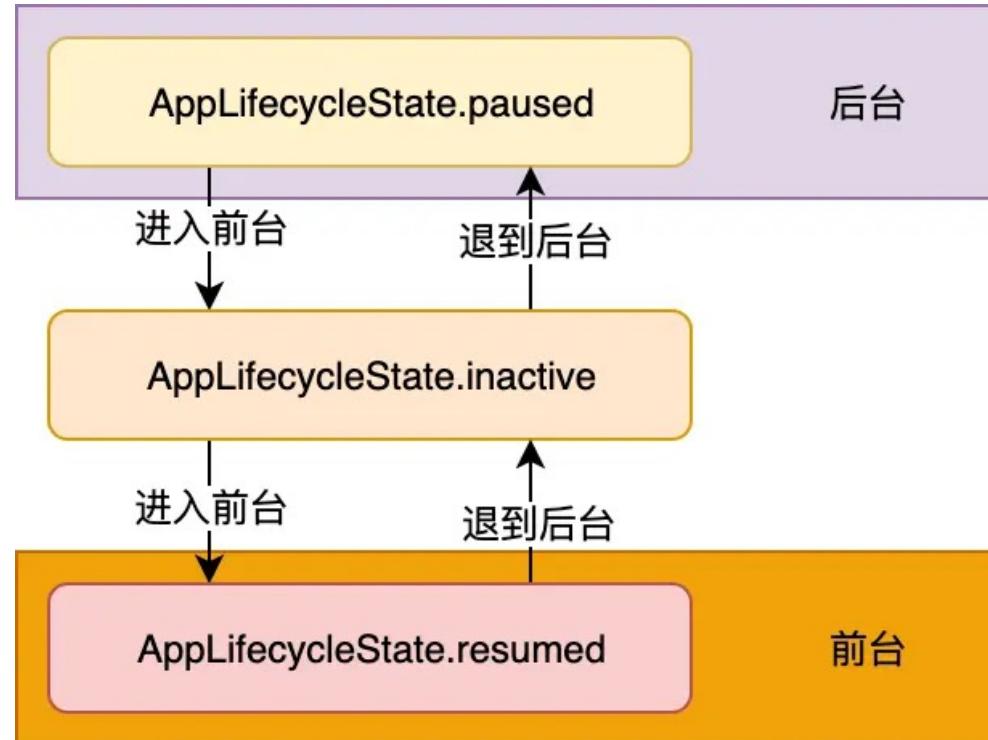


State 生命周期中的方法调用对比

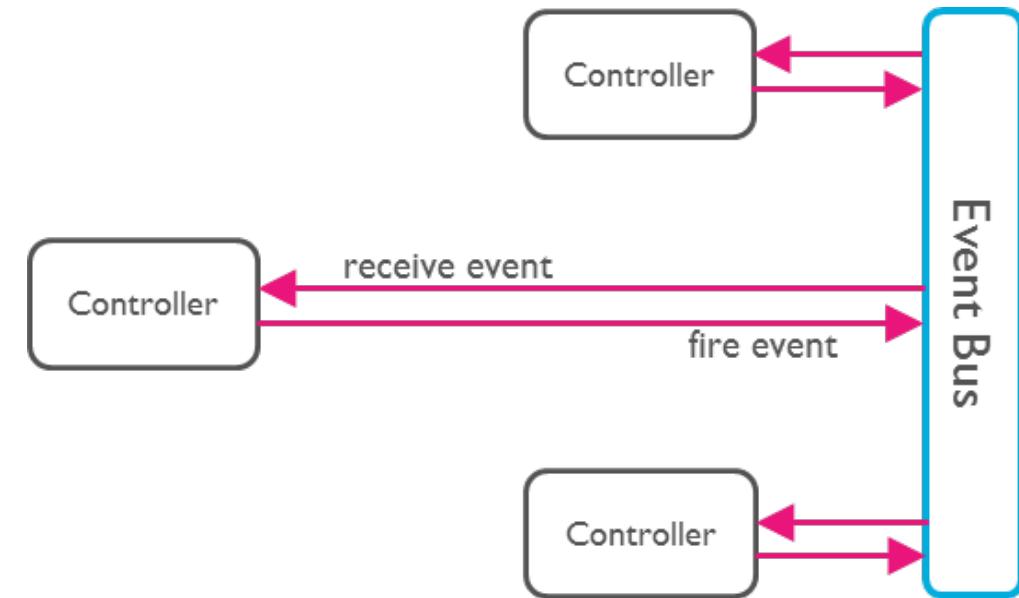
方法名	功能	调用时机	调用次数
构造方法	接收父Widget传递的初始化UI配置数据	创建State时	1
initState	与渲染相关的初始化工作	在State被插入视图树时	1
didChangeDependencies	处理State对象依赖关系变化	initState后及State对象依赖关系变化时	≥ 1
build	构建视图	State准备好数据需要渲染时	≥ 1
setState	触发视图重建	需要刷新UI时	≥ 1
didUpdateWidget	处理Widget的配置变化	父Widget setState触发子Widget重建时	≥ 1
deactivate	组件被移除	组件不可视	≥ 1
dispose	组件被销毁	组件被永久移除	1

定义了 App 从启动到退出的全过程。

生命周期的回调 `didChangeAppLifecycleState`, 有一个参数类型为 `AppLifecycleState` 的枚举类, 是对 App 生命周期状态的封装。常用状态包括 `resumed`、`inactive`、`paused`。



方式	数据流动方式	使用场景
属性传值	父到子	简单数据传递
InheritedWidget	父到子	跨层数据传递
Notification	子到父	状态通知
EventBus	发布订阅	消息批量同步



Part

4 路由管理



页面之间的跳转是通过 Route 和 Navigator 来管理的：

- Route 是页面的抽象，主要负责创建对应的界面，接收参数，响应 Navigator 打开和关闭；
- Navigator 则会维护一个路由栈管理 Route，Route 打开即入栈，Route 关闭即出栈，还可以直接替换栈内的某一个 Route。

根据是否需要提前注册页面标识符，Flutter 中的路由管理可以分为两种方式：

- 基本路由：无需提前注册，在页面切换时需要自己构造页面实例。
- 命名路由：需要提前注册页面标识符，在页面切换时通过标识符直接打开新的路由。



```
class FirstScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return RaisedButton(
      // 打开页面
      onPressed: ()=> Navigator.push(context, MaterialPageRoute(builder: (context) => SecondScreen()));
    );
  }
}

class SecondPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return RaisedButton(
      // 回退页面
      onPressed: ()=> Navigator.pop(context)
    );
}
```



```
MaterialApp(
  ...
  // 注册路由
  routes:{ 
    "second_page":(context)=>SecondPage(),
  },
);
// 使用名字打开页面
Navigator.pushNamed(context,"second_page");
```

Part

5 依赖管理



Dart 提供了包管理工具 Pub，用来管理代码和资源。

pubspec.yaml 文件。

在完成了所有依赖包的下载后，Pub 会在应用的根目录下创建 packages 文件，将依赖的包名与系统缓存中的包文件路径进行映射。

会自动创建 pubspec.lock 文件，作用类似于前端的 package-lock.json 文件。

访问 <https://pub.dev/> 来获取可用的第三方包。

对于不对外公开发布，使用本地路径或 Git 地址的方式进行包声明。

```
name: flutter_app_example #应用名称
description: A new Flutter application. #应用描述
version: 1.0.0
#Dart运行环境区间
environment:
  sdk: ">=2.1.0 <3.0.0"
#Flutter依赖库
dependencies:
  flutter:
    sdk: flutter
  cupertino_icons: ">0.1.1"
  package1:
    path: ../../package1/ #路径依赖
  date_format:
    git:
      url: https://github.com/xxx/package2.git #git依赖
```

Part

6 调试



1

inspector 视图检查

The screenshot shows the Flutter Inspector interface with the following details:

- Widget Inspector - flutterwebapp**: The title bar.
- iPhone 12 iOS 14.5**: The simulator device and OS information.
- chat_container.dart**: The file path.
- Select Widget Mode**: A dropdown menu.
- Layout Explorer**: The active tab in the inspector.
- Details Tree**: An alternative tab.
- Widget Hierarchy**: A tree view of the widget structure:
 - TabBarView
 - Scaffold
 - ChatList
 - ListView
 - Column
 - Divider
 - Container
 - ListTile
 - Image
 - Row
 - Text
- Layout Details** (Cross Axis, Main Axis):
 - Column: Total Flex Factor: 0
 - Divider: flex: null, unconstrained vertical, fit: tight, height is 1.0 (height is unconstrained)
 - Container: w=390.0; (0.0 <= w <= 390.0)
 - Container: w=390.0; (w=390.0)
- BUG CONSOLE**:


```
Launching lib/main.dart on iPhone 12 in debug mode... lib/main.dart:1
[Code build done.]
29.3s
Connecting to VM Service at ws://127.0.0.1:56076/loM9WV70UtE=/ws
28 flutter: 屏幕 Size -> Size(390.0, 844.0)
flutter: login success
3 flutter: 屏幕 Size -> Size(390.0, 844.0)
```
- TERMINAL**:


```
[oh-my-zsh] To fix your permissions you can do so by disabling
[oh-my-zsh] the write permission of "group" and "others" and making sure th
at the
[oh-my-zsh] owner of these directories is either root or your current user.
[oh-my-zsh] The following command may help:
[oh-my-zsh]   compaudit | xargs chmod g-w,o-w
[oh-my-zsh] If the above didn't help or you want to skip the verification o
f
[oh-my-zsh] insecure directories you can set the variable ZSH_DISABLE_COMPF
IX to
[oh-my-zsh] "true" before oh-my-zsh is sourced in your zshrc file.

(base) ➜ flutterwebapp git:(master) ✘
```
- Bottom Navigation**:
 - .metadata
 - OUTLINE
 - TIMELINE
 - DEPENDENCIES
 - master* ↻ 0 △ 1 ⌂ 3 ↗ Login Configure Debug my code
 - 存在0个异常标点 Dart DevTools ⌂ Go Live iPhone 12 (ios simulator) ⌂

Flutter DevTools Flutter Inspector Performance CPU Profiler Memory Debugger Network Logging App Size

Profile granularity: medium Performance Overlay Track Widget Builds Export

Frame Time (UI) Frame Time (Raster) Jank (slow frame)

31 ms 24 ms 17 ms 10 ms 3 ms 0 ms

60 FPS 9 FPS (average)

Timeline Events Search

0.0 ms 1939.704 ms 3879.408 ms 5819.112 ms 7758.816 ms 9698.520 ms 11638.224 ms

UI Dart... Dartisolat... Layout Build B... Warm up shader

CPU Profile: VsyncProcessCallback (1.3 ms)

Summary Bottom Up Call Tree CPU Flame Chart Filter by tag: none Expand All Collapse All

Total Time	Self Time	Method
4222.02 ms (100.00%)	0.00 ms (0.00%)	▼ all
1204.28 ms (28.52%)	5.64 ms (0.13%)	> _RawReceivePortImpl._handleMessage
		dart:_internal/vm/lib/isolate_patchl

flutter.dev/devtools/performance x64-64 ios

Part

7 错误监控



Dart 采用事件循环的机制来运行任务，所以各个任务的运行状态是互相独立的。

即便某个任务出现了异常我们没有捕获它，Dart 程序也不会退出，只会导致当前任务后续的代码不会被执行，用户仍可以继续使用其他功能。

按来源分：App 异常和Framework 异常。

App 异常：就是应用代码的异常，通常由未处理应用层其他模块所抛出的异常引起。

同步异常可以通过 try-catch 机制捕获，异步异常采用 Future 提供的 catchError 语句捕获。

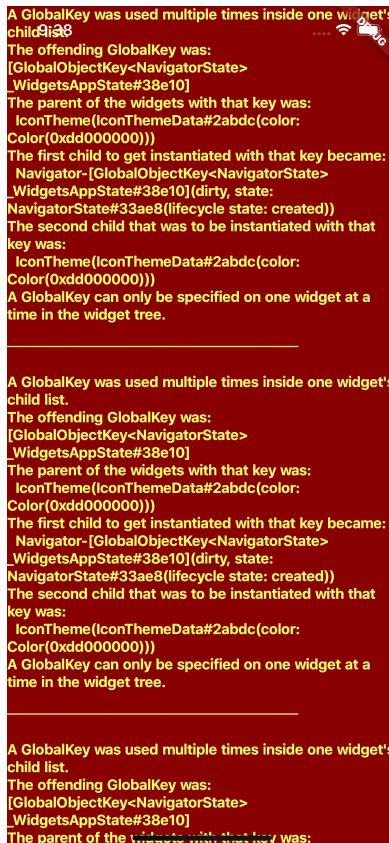
Flutter 提供了runZonedGuarded方法：集中管理代码中的所有异常。

捕获 Flutter 应用中的未处理异常，可以把 main 函数中的 runApp 语句也放置在 Zone 中。这样在检测到代码中运行异常时，我们就能根据获取到的异常上下文信息，进行统一处理了。

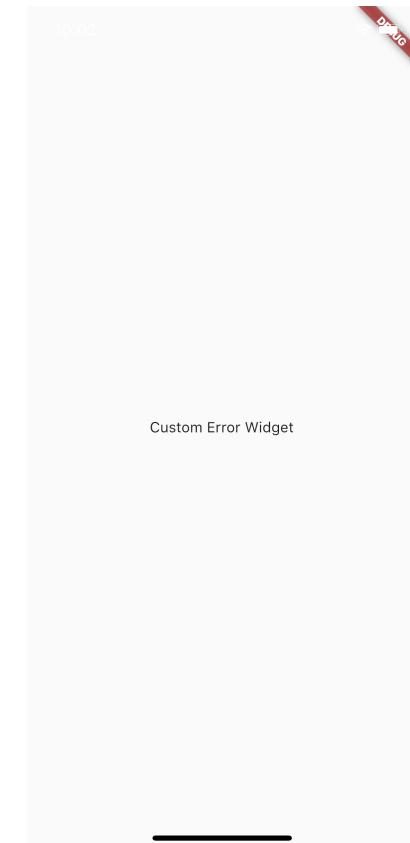
```
runZonedGuarded<Future<Null>>(() async {
  runApp(MyApp());
}, (error, stackTrace) async {
  //这里可以捕捉同步异常，也可以捕捉异步异常
  print("expectInTest zone error ${stackTrace.toString()}");
});
```

Framework 异常：Flutter 框架引发的异常，通常是由应用代码触发了 Flutter 框架底层的异常判断引起的。可以使用 `FlutterError.onError` 回调进行拦截。

Flutter 框架在调用 `build` 方法构建页面时进行了 `try-catch` 的处理，并提供了一个 `ErrorWidget`，用于在出现异常时进行信息提示。



通常会重写 `ErrorWidget.builder` 方法，将这样的错误提示页面替换成一个更加友好的页面。



第三方 SDK 服务厂商，比如友盟、Bugly，以及开源的 Sentry 等。

Bugly 的社区活跃度比较高，以它为例，为 Bugly 的数据上报提供 Dart 层接口。总体上可以分为两个步骤：

- 初始化 Bugly SDK；
- 使用数据上报接口。

这两步对应着在 Dart 层需要封装的 2 个原生接口调用，即 `setup` 和 `postException`，它们都是在方法通道上调用原生代码宿主提供的方法。

```
class FlutterCrashPlugin {  
    //初始化方法通道  
    static const MethodChannel _channel =  
        const MethodChannel('flutter_crash_plugin');  
  
    static void setUp(appID) {  
        //使用app_id进行SDK注册  
        _channel.invokeMethod("setUp", {'app_id':appID});  
    }  
    static void postException(error, stack) {  
        //将异常和堆栈上报至Bugly  
        _channel.invokeMethod("postException",  
            {'crash_message':error.toString(), 'crash_detail':stack.toString()});  
    }  
}
```

Part

8 混合开发



使用 Flutter 从头开始写一个 App，是一件轻松惬意的事情。但，对于成熟产品来说，完全摒弃原有 App 的历史沉淀，而全面转向 Flutter 并不现实。用 Flutter 去统一 iOS/Android 技术栈，把它作为已有原生 App 的扩展能力，通过逐步试验有序推进从而提升终端开发效率，可能才是现阶段 Flutter 最具吸引力的地方。

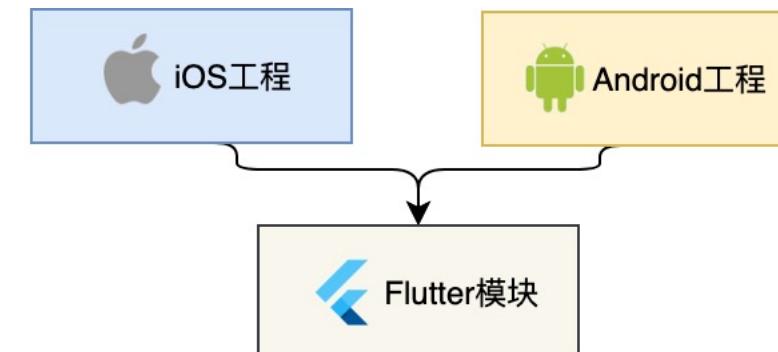
有两个办法：

统一管理模式：将原生工程作为 Flutter 工程的子工程，由 Flutter 统一管理。

三端分离模式：将 Flutter 工程作为原生工程共用的子模块，维持原有的原生工程管理方式不变。



统一管理模式



三端分离模式

三端工程分离模式的关键是抽离 Flutter 工程，将不同平台的构建产物依照标准组件化的形式进行管理，即 Android 使用 aar、iOS 使用 pod。将 Flutter 模块打包成 aar 和 pod，这样原生工程就可以像引用其他第三方原生组件库那样快速接入 Flutter 了。

原生工程对 Flutter 的依赖主要分为两部分：

- Flutter 库和引擎，也就是 Flutter 的 Framework 库和引擎库；
- Flutter 工程，也就是 Flutter 模块功能，主要包括 Flutter 工程 lib 目录下的 Dart 代码实现的这部分功能。

在已经有原生工程的情况下，需要在同级目录创建 Flutter 模块，构建 iOS 和 Android 各自的 Flutter 依赖库。

```
flutter create -t module flutter_library
```

官方文档：<https://flutter.dev/docs/development/add-to-app>

对 Android 的 Flutter 依赖抽取步骤如下：

- ✓ 首先在 Flutter_library 的根目录下，执行 aar 打包构建命令：flutter build apk –debug
- ✓ 其次，打包构建的 flutter-debug.aar 位于.android/Flutter/build/outputs/aar/ 目录下，把它拷贝到原生 Android 工程 AndroidDemo 的 app/libs 目录下，并在 App 的打包配置 build.gradle 中添加对它的依赖：

```
dependencies {  
    ...  
    implementation(name: 'flutter-debug', ext: 'aar')  
    ...  
}
```

- ✓ 最后，改一下 MainActivity.java 的代码，把它的 contentView 改成 Flutter 的 widget：

```
View FlutterView = Flutter.createView(this, getLifecycle(), "defaultRoute");  
setContentView(FlutterView)
```

- ✓ 首先在 Flutter_library 的根目录下，执行 iOS 打包构建命令：flutter build ios –debug
- ✓ 其次，在 iOSDemo 的根目录下创建一个名为 FlutterEngine 的目录，并把这两个 framework 文件拷贝进去。把这两个产物手动封装成 pod。还需要在该目录下创建 FlutterEngine.podspec，即 Flutter 模块的组件定义：

```
Pod::Spec.new do |s|
  s.name = 'FlutterEngine'
  s.version = '0.1.0'
  s.summary = 'XXXXXXX'
  s.description = <<-DESC
```

- ✓ 再修改 Podfile 文件把它集成到 iOSDemo 工程中：

```
target 'iOSDemo' do
  pod 'FlutterEngine', :path => './'
end
```

- ✓ 最后，修改一下 AppDelegate.m 的代码，把 window 的 rootViewController 改成 FlutterViewController：

```
FlutterViewController *vc = [[FlutterViewController alloc] init];
[vc setInitialRoute:@"defaultRoute"];
self.window.rootViewController = vc;
```



资料

- Flutter官网：<https://flutter.dev/>
- Dart官网：<https://dart.dev/>
- Bugly iOS SDK 使用指南：<https://bugly.qq.com/docs/user-guide/instruction-manual-ios/>
- Bugly Android SDK 使用指南：<https://bugly.qq.com/docs/user-guide/instruction-manual-android/>
- event_bus：https://pub.dev/packages/event_bus
- FlutterExampleApps：<https://github.com/iampawan/FlutterExampleApps>
- awesome-flutter：<https://github.com/Solido/awesome-flutter>
- 单元测试：<https://flutter.dev/docs/cookbook/testing/unit/introduction>

应用安全

自动化测试

Next

工程化

性能优化

T H A N K S