Flutter稳定性与性能优化

AlanGe

一、Flutter异常与Crash

Flutter异常指的是Flutter程序在运行时所抛出的异常分为：

- Dart代码运行时发生的异常
- Flutter框架异常
- 原生代码运行时抛出的异常，如：Android的Java和kotlin，iOS的OC和swift

做Flutter应用Dart代码占绝大多数，所以本文我们重点学习下Flutter中Dart和框架异常的捕获与收集。

Dart代码运行时发生的异常与Java、kotlin、OC等具有多线程模型的编程语言不同，Dart是一门单线程的编程语言，采用事件循环机制来运行任务，所以各个任务的运行状态是互相独立的。也即是说，当程序运行过程中出现异常时，即使没有像Java那样使用try-catch机制来捕获异常，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，而其它功能仍然可以继续使用。

异常捕获

根据异常代码的执行时序，Dart异常可以分为同步和异步异

常两类。首先我们看同步异常的捕获方式：

同步异常的捕获方式：

```
//使用try-catch捕获同步异常
    try {
      throw StateError('There is a dart exception.');
    } catch (e) {
      print(e);
    }
```

异步异常捕获方式：

异步异常的捕获有两种方式：

- 一种是使用 Future 提供的 catchError 语句来进行捕获；
- 另外一种是将异步转同步然后通过 try-catch 进行捕获；

```
    //使用catchError捕获异步异常
    Future.delayed(Duration(seconds: 1))
        .then(
            (e) => throw StateError('This is first Dart
exception in Future.'))
        .catchError((e) => print(e));
    try {
      await Future.delayed(Duration(seconds: 1)).then(
          (e) => throw StateError('This is second Dart
exception in Future.'));
    } catch (e) {
      print(e);
    }
```

集中捕获异常

在 Android 中我们可以通过 `Thread.UncaughtExceptionHandler` 接口来集中收集异常，那么在 Flutter 中如何集中收集异常呢?

Flutter 提供的 `Zone.runZonedGuarded()` 方法。在 Dart 语言中，Zone 表示一个代码执行的环境范围，其概念类似沙盒，不同沙盒之间是互相隔离的。如果想要处理沙盒中代码执行出现的异常，可以使用沙盒提供的 onError 回调函数来拦截那些在代码执行过程中未捕获的异常：

```
    runZonedGuarded(() {
      throw StateError('runZonedGuarded:This is a Dart
exception.');
    }, (e, s) => print(e));
    runZonedGuarded(() {
      Future.delayed(Duration(seconds: 1)).then((e) =>
throw StateError(
          'runZonedGuarded:sThis is a Dart exception in
Future.'));
    }, (e, s) => print(e));
```

从上述代码中不难看出，无论是同步异常还是异步异常，都可以使用 Zone 直接捕获到。同时，如果需要集中捕获 Flutter 应用中未处理的异常，那么可以把 main 函数中的

runApp语句也放置在Zone中，这样就可以在检测到代码运行异常时对捕获的异常信息进行统一处理：

```
  runZonedGuarded<Future<Null>>(() async {
    runApp(BiliApp());
  }, (e, s) => print(e));
```

案例

```
void main() {
  HiDefend().run(BiliApp());
}
...
class HiDefend {
  run(Widget app) {
    //框架异常
    FlutterError.onError = (FlutterErrorDetails details)
async {
      //线上环境，走上报逻辑
      if (kReleaseMode) {
        Zone.current.handleUncaughtError(details.exception,
details.stack);
      } else {
        //开发期间，走Console抛出
        FlutterError.dumpErrorToConsole(details);
      }
    };
    runZonedGuarded<Future<Null>>(() async {
      runApp(app);
    }, (e, s) => _reportError(e, s));
  }
```

```
  ///通过接口上报
  Future<Null> _reportError(Object error, StackTrace stack)
async {
    print('catch error: $error');
  }
}
```

异常上报

捕获到异常后可以在上述`_reportError`方法中上报到服务端，像BAT等一线互联网大厂都有自己的Crash监控平台。如果公司没有自己的Crash平台，可以接入第三方的如：Buggly。

- https://pub.dev/packages?q=bugly

二、Flutter测试

软件测试是发现程序错误衡量软件质量必不少的一个环节，在企业中会有专门的软件测试工程师来负责软件测试和质量的障。作为一名Flutter开发人员了解Flutter测试的方法和手段有助于减少程序的Bug开发出更高质量的应用。那么Flutter是如何进行测试的呢?

在本篇教程中将下大家分享Flutter的主流测试方式以和案例，在Flutter中主要有以下三种类型的测试：

- 单元测试
- Widget测试
- 集成测试

单元测试

测试单一功能、方法或类，单元测试通常不会读取/写入磁盘、渲染到屏幕，也不会从运行测试的进程外部接收用户操作。单元测试的目标是在各种条件下验证逻辑单元的正确性。

如果所测试对象有外部依赖，那么外部依赖要能够被模拟出来，否则是无法进行单元测试。

所需依赖

```
dev_dependencies:
  flutter_test:
    sdk: flutter
```

案例

```
///单元测试
void main() {
  ///测试HiCache的存储和读取
  test('测试HiCache', () async {
    //fix ServicesBinding.defaultBinaryMessenger was
accessed before the binding was initialized.
    TestWidgetsFlutterBinding.ensureInitialized();
```

```
    //fix MissingPluginException(No implementation found
for method getAll on channel plugins.flutter.io/
shared_preferences)
    SharedPreferences.setMockInitialValues({});
    await HiCache.preInit();
    var key = "testHiCache", value = "Hello.";
    HiCache.getInstance().setString(key, value);
    expect(HiCache.getInstance().get(key), value);
  });
}
```

在这个案例中我们对项目中的缓存模块`HiCache`进行了单元测试，主要用来测试它的存储和读取功能是否正常。

Widget测试

Widget 测试可以用于测试单独的 class, function, 和 Widget。

Widget测试具有一定的局限性，所测试的Widget必须要能够独立运行，或者所以依赖条件能够被模拟出来。

所需依赖

```
dev_dependencies:
  flutter_test:
    sdk: flutter
```

按照上述要求整个APP有哪些Widget能进行Widget测试?

案例

```
...
class UnKnownPage extends StatefulWidget {
  @override
  _UnKnownPageState createState() => _UnKnownPageState();
}

class _UnKnownPageState extends State<UnKnownPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(),
      body: Container(
        child: Text('404'),
      ),
    );
  }
}
...
///Widget测试
void main() {
  testWidgets('测试UnKnownPage', (WidgetTester tester) async
{
    //UnKnownPage虽然没有Flutter框架之外的依赖，但因为用到了
Scaffold所以需要用MaterialApp包裹
    await tester.pumpWidget(MaterialApp(home:
UnKnownPage()));
    expect(find.text('404'), findsOneWidget);
  });
}
```

在这个案例中我们对项目中的`UnKnownPage`Widget进行了测

试，主要用来测试该页面中是否存在一个内容为 404 的 Text。

集成测试

集成测试主要是测试各部分一起运行或者测试一个应用在真实设备上运行的表现的时候就要用到集成测试。

所需依赖

```
dev_dependencies:
  flutter_test:
    sdk: flutter
  integration_test:
    sdk: flutter
```

integration 结合，整合

主要步骤

1. 添加测试驱动
2. 编写测试用例
3. 运行测试用例
4. 查看结果

添加测试驱动

添加测试驱动的目的是为了方便通过 flutter drive 命令运行集

成测试：

在项目根目录创建`test_driver`目录并添加文件`integration_test.dart`：

```
import 'package:integration_test/
integration_test_driver.dart';
Future<void> main() => integrationDriver();
```

编写测试用例

在项目根目录创建`integration_test`目录并添加文件`app_test.dart`。接下来我们就来测试下登录模块的跳转功能：

```
import 'package:flutter/material.dart';
import 'package:flutter_bili_app/main.dart' as app;
import 'package:flutter_bili_app/navigator/
hi_navigator.dart';
import 'package:flutter_test/flutter_test.dart';
import 'package:integration_test/integration_test.dart';

void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized();

  testWidgets('Test login jump', (WidgetTester tester)
async {
    //构建应用
    app.main();
```

```
    //捕获一帧
    await tester.pumpAndSettle();
    //通过key来查找注册按钮
    var registrationBtn = find.byKey(Key('registration'));
    //触发按钮的点击事件
    await tester.tap(registrationBtn);
    //捕获一帧
    await tester.pumpAndSettle();
    await Future.delayed(Duration(seconds: 3));

    //判断是否跳转到了注册页

expect(HiNavigator.getInstance().getCurrent().routeStatus,
        RouteStatus.registration);

    //获取返回按钮，并触发返回上一页
    var backBtn = find.byType(BackButton);
    await tester.tap(backBtn);
    await tester.pumpAndSettle();
    await Future.delayed(Duration(seconds: 3));
    //判断是返回到登录页
    expect(
        HiNavigator.getInstance().getCurrent().routeStatus,
RouteStatus.login);
  });
}
```

在这个案例中我们通过获取当前页APP页面上的帧，然后基于捕获的这一帧查找到对应控件，并模拟了点击。为了能够判断登录模块的跳转逻辑是否正常在上述代码中我们通过`HiNavigator`获取到当前的页面路由状态来进行判断。

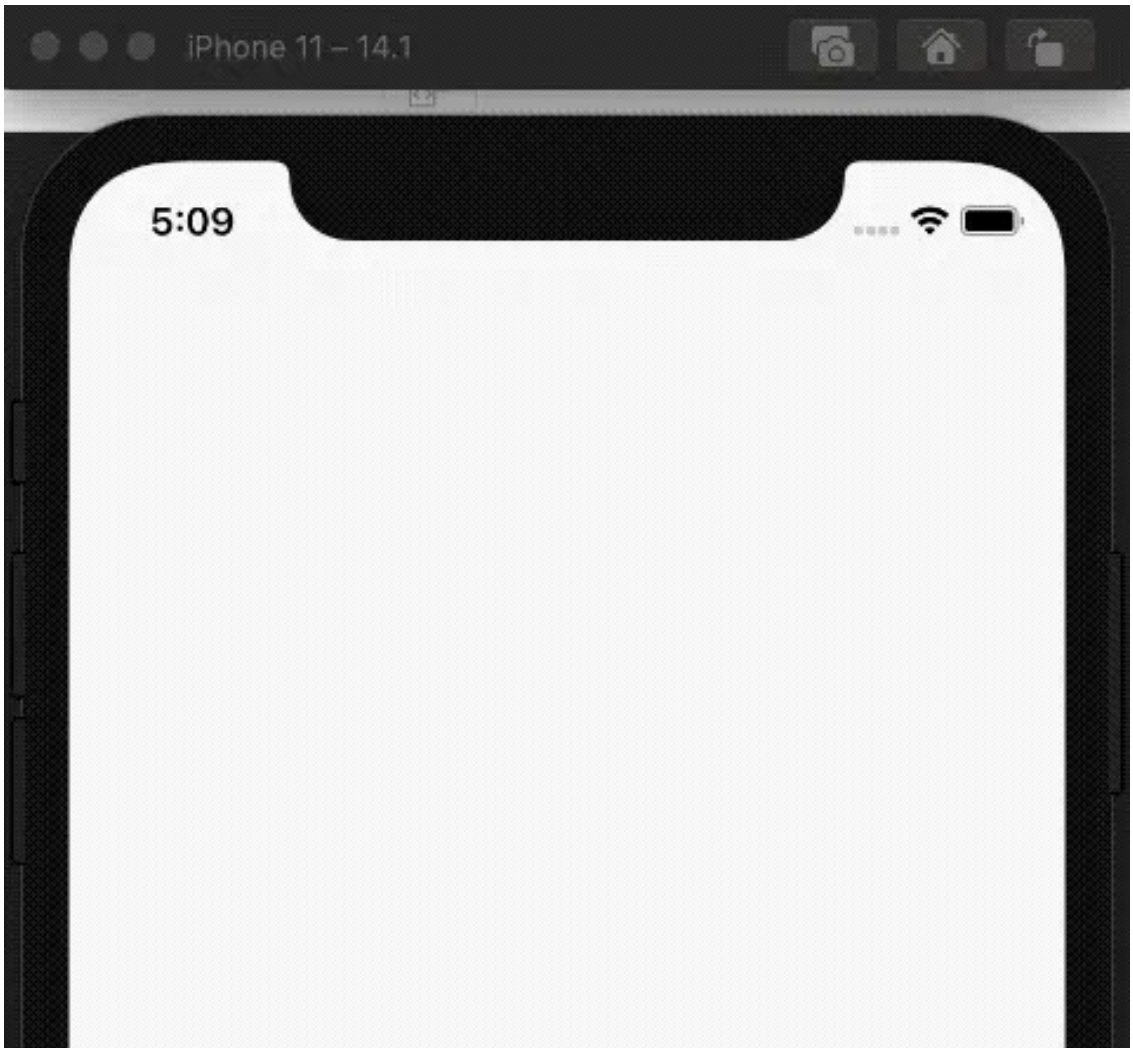
运行测试用例

运行集成测试的测试用例可以通过以下命令来完成：

```
flutter drive   --driver=test_driver/integration_test.dart
--target=integration_test/app_test.dart
```

- –driver：用于指定测试驱动的路径；
- –targe：用于指定测试用例的路径；

下面实现运行效果图：



IntegrationTest

如果APP之前已经登录过那么需要将其先卸载来清除登录缓存，以便APP能够在测试的时候正常进入到登录页。

三、Flutter性能优化

一般情况下我们通过 Flutter 技术构建的应用程序在默认情况下都是高性能的。但是呢，在编程中难免会掉入一些性能陷阱，在这一节呢我将向大家分享 Flutter 一些性能优化的思路并根据优化实践。

性能优化的最好时机是在编码阶段，如果你能掌握一些性能优化技巧和最佳实践，那么在编码时就可以考虑到怎么设计会性能最优，怎么实现会拖慢性能；而不是开发完成之后在去着手做性能优化，这样先污染后治理的思路。

在我们前面的 Flutter 开发过程中已经应用了不少 Flutter 性能优化的技巧以及一些最佳实践，所以说呢，在这一节主要还是对性能优化做个总结，以帮助大家更新的学习和掌握。

- 内存优化
- build() 方法优化
    - 在 build() 方法中执行了耗时的操作
    - build() 方法中堆砌了庞大的 Widget
- 列表优化方法
- 案例：帧率优化

内存优化

要进行内存优化首先我们需要了解下内存的检测手段，这样

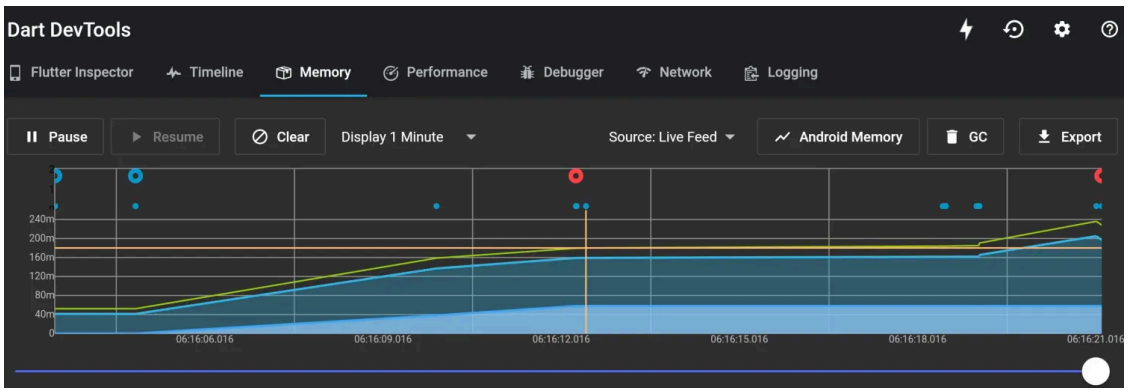
我们才好进行内存优化前后的效果对比。

Flutter性能检测工具Flutter Performance

Performance 表演，表现

在 IDE 的 Flutter plugin 中提供了 `Flutter Performance` 工具，它是一个可用来检测Flutter滑动帧率和内存的工具。

我们可以从IDE的侧边栏中打开这个工具，也可以借助Dart DevTools来查看内存的使用情况：



DartDevTools-memory

此时可以打开一个页面或进行一些操作来观察内存的变化，如果内存突然增大很多就要特别关注是否是合理的增加，必要时要排查导致内存增加的原因和考虑对于的优化方案。

关于如何判断优化后内存有没有变化，可以通过Dart DevTools的Memory选项卡来完成，当你销毁一个FlutterEngine后可以通过`GC`按钮来触发一次`GC`来查看内存的变化。

inspector 检查员

build() 方法优化

在我们用 Flutter 开发 UI 的时候，打交道最多的方法便是 `build()` 了，使用 `build()` 方法时有两个常见的陷阱，我们一起来看一下：

在 build() 方法中执行了耗时的操作

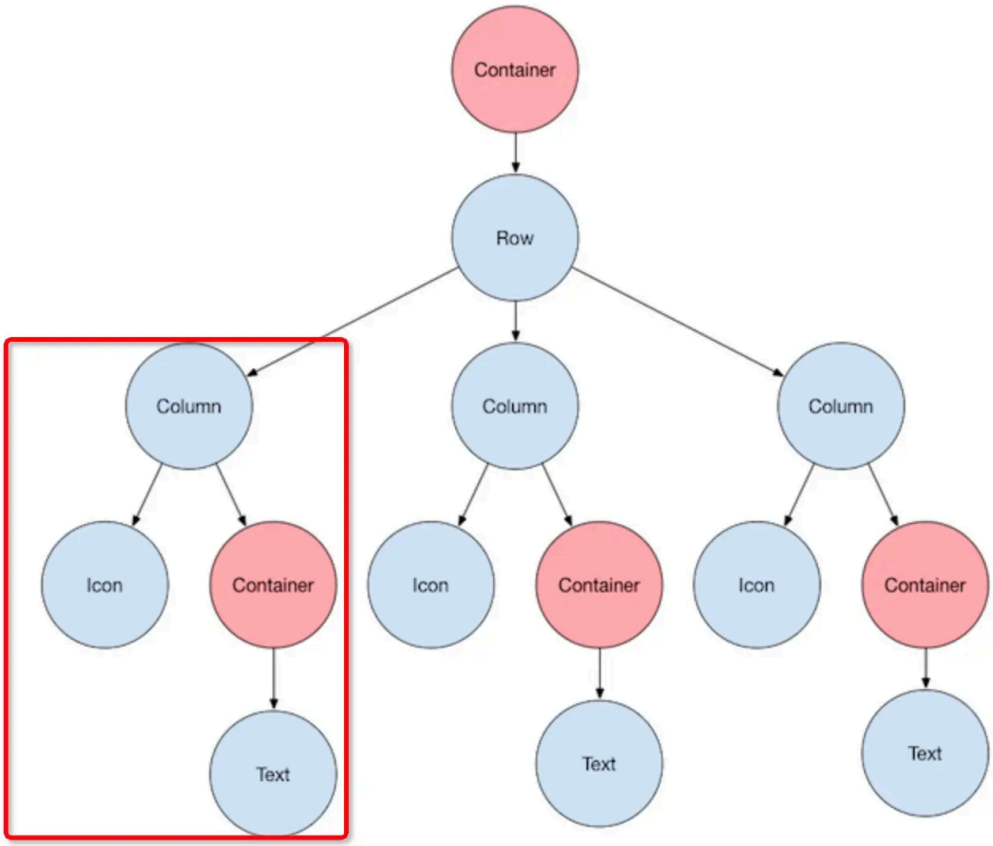
我们应该尽量避免在 build() 中执行耗时的操作，这是因为 build() 方法会频繁的调用，尤其是当父 Widget 重建的时候；所以，耗时的操作建议挪到 `initState()` 这种不会被频繁调用的方法中；

另外，我们尽量不要在代码中进行阻塞式操作，可以将文件读取，数据库操作，网络请求这些操作通过 Future 来转成异步完成；另外对于 CPU 计算频繁的操作比如：图片压缩等可以使用 Isolate 来充分利用多核心 CPU；

build() 方法中堆砌了庞大的 Wdiget

在画 UI 的时候有的小伙伴喜欢一把梭，其实这是一个特别不好的习惯；如果 build() 中返回的 Widget 过于庞大会导致三个问题：

- 代码可读性差：因为 Flutter 的布局方式的特殊性，画界面我们离不了的需要一个 Wdiget 嵌套一个 Wdiget，但如果 Wdiget 嵌套太深则会导致代码的可读性变差，也不利于后期的维护和扩展；
- 复用难：由于所有的代码都在一个 build() 方法中，会到导致无法将公共的 UI 代码服用到其它的页面或模块；
- 影响性能：我们在 State 上调用 setState() 时，所有 build() 中的 Widget 都将被重建；因此 build() 中返回的 Widget 树越大那么需要重新建的 Widget 越多，对性能越不利；见下图：



widget-tree

假如上图是我们在一个 build() 方法中所返回的 widget 树，那么当左侧红框中的 widget 需要更新的时候，最小的更新成本是只更新需要跟新的部分，但是由于它们都在一个 State 的 build 方法所以，调用 setState() 时会导致右边很多不需要更新的 widget 也需要重新；正确的做法是，将 setState() 的调用转移到其 UI 实际需要更改的 Widget 子树部分。

可以回想下在我们项目中有哪些地方又进行过类似的优化：`hi_flexible_header.dart`。

列表优化方法

在构建大型网格或列表时，我们要尽量避免直接使用 `ListView(children: [],)` 或 `GridView(children: [],)`，因为在这种场景下会导致列表中所有的数据都会被一次性绘制出来不管列表内容是否可见，这种用法类似 Android 的 ScrollView；所以说当你的列表中的数据量比较大时建议你用：

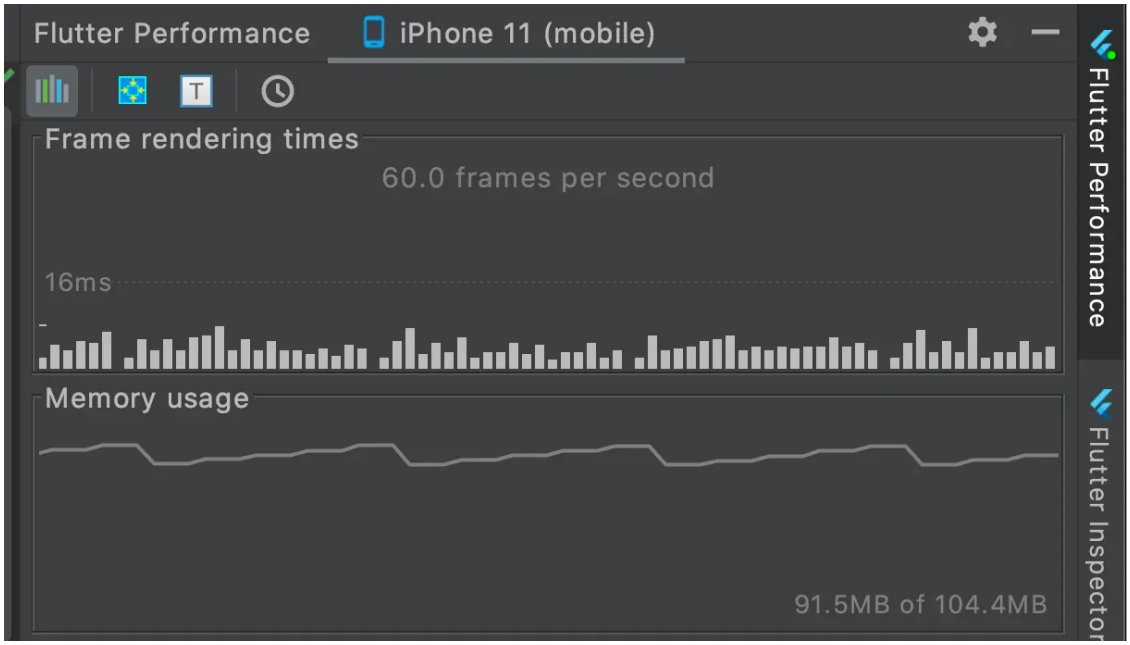
- ListView.builder(itemBuilder: null)
- GridView.builder(gridDelegate: null, itemBuilder: null)

这两个方法，因为这两个方法只有在屏幕的可见部分是在列表的内容才开始被创建，这种又发类似于 Android 的

RecyclerView。

帧率优化

决定列表性能的好坏一个很关键的因素就是帧率，通常情况下手机的刷新频率为 60fps，当然目前陆续出现一些高刷屏的手机能够达到 90 甚至 120 的 fps。在 Flutter 中获取应用的帧率我们可以通过 Flutter Performance 选项卡来查看页面帧率：



Flutter-Performance

另外可以点击上图左上角 Performance overlay 按钮来打开性能图层功能：

The-performance-overlay

通过这个图表我们可以帮助我们分析 UI 是否产生了卡顿，垂直的绿色条条代表的是当前帧，每一帧都应该在 1/60 秒（大约 16 ms）内创建并显示。如果有一帧超时而无法显示，就导致了卡顿，上述图表就会展示出来一个红色竖条。如果是在 UI 图表出现了红色竖条，则表明 Dart 代码消耗了大量资源。红色竖条表明当前帧的渲染和绘制都很耗时。

帧率优化案例

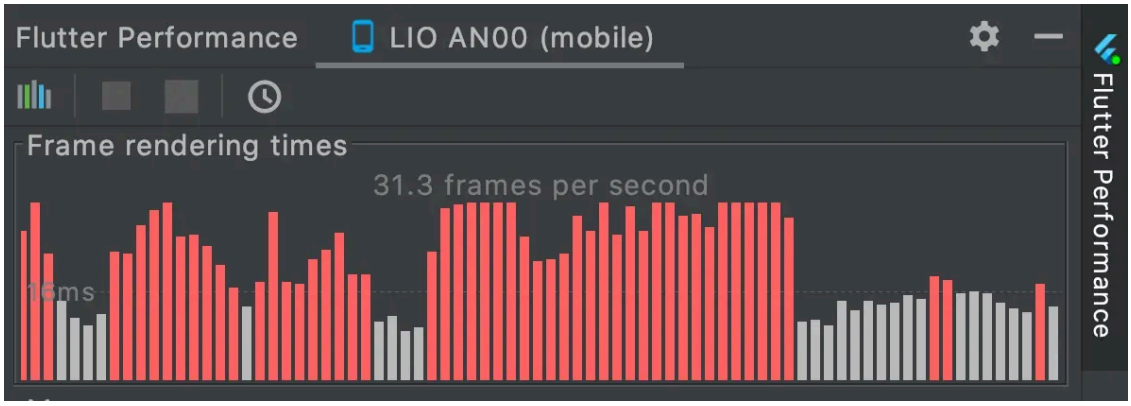
借助Flutter Performance我们对课程项目的首页进行帧率检测：

因为debug模式下Flutter的性能会受到比较大的限制，为了还原检测的真实性我们需要在分析模式运行APP。

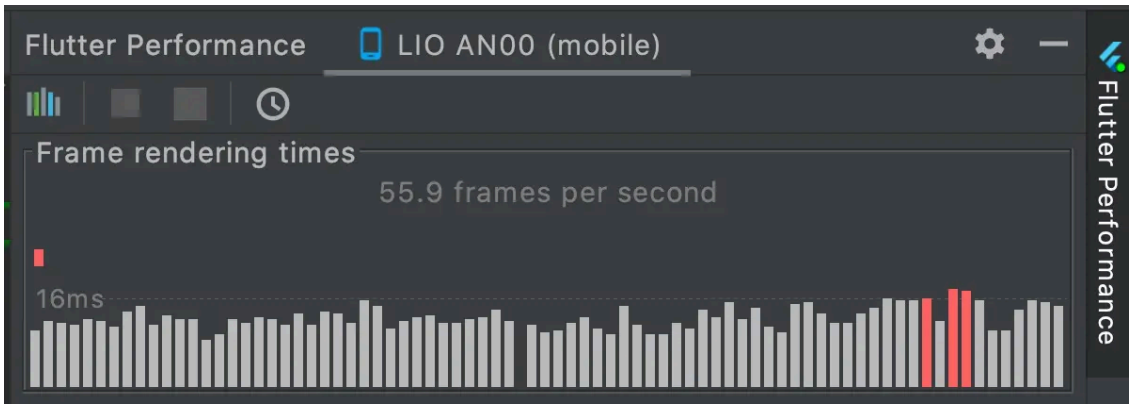
- 在 Android Studio 和 IntelliJ 使用 Run > Flutter Run main.dart in Profile Mode 选项
- 或者通过命令行使用 --profile 参数运行
    - `flutter run --profile` 命令模式下无法使用DEV Tools进行性能分析

注意：模拟器不支持分析模式，可以用真机连接电脑来进行分析

优化前



优化后



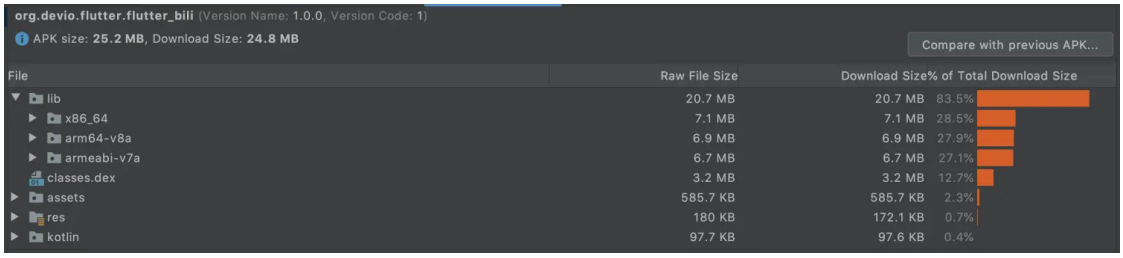
四、Flutter包大小优化

- 包大小分析
- 优化思路
- APK优化实战

Flutter包大小优化需要用到集成打包的知识，对于还不了解如何打包Flutter应用的小伙伴可以先看我们课程后面有关集成打包部分的内容，然后再来学习本节课。

包大小分析

将APK拖拽到AS中可以进行包大小分析：



apk-size

优化思路

- 图片优化
- 移除冗余的二三库
- 启用代码缩减和资源缩减
- 构建单 ABI 架构的包

图片优化

Flutter 资源中占比较多的一般是图片，对于图片的优化常用的有两个方案：

- 图片压缩：对于过大的图片可以使用 https://tinypng.com/ 进行压缩，使用压缩后的图片
- 使用网络图片：也可根据业务需要将本地图片改为网络图片；

移除冗余的二三库

随着业务的增加，项目中会引入越来越多的二三方库，其中有不少是功能重复的，甚至是已经不再使用的。移除不再使用的和将相同功能的进行合并可以进一步较少包体积。

启用代码缩减和缩减资源

默认情况下压缩和缩减资源是开启的，启用代码缩减和缩减资源后构建出来的release包会减少10%左右的大小，甚至更多。

如果要代码缩减和缩减资源可以参考下如下设置：

```
buildTypes {
    release {
        signingConfig signingConfigs.release
        minifyEnabled false
        shrinkResources false
    }
}
```

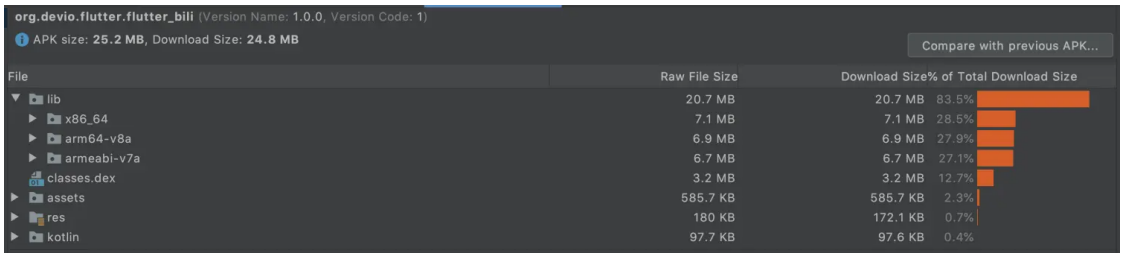
minify 缩小

shrink 收缩，畏缩

- minifyEnabled：是否启用代码缩减
  - 如果将 minifyEnabled 属性设为 true，系统会默认启用 R8 代码缩减功能。代码缩减（也称为“摇树优化”）是指移除 R8 确定在运行时不需要的代码的过程。此过程可以大大减小应用的大小，例如，当您的应用包含许多库依赖项，但只使用它们的一小部分功能时。
- shrinkResources：是否启用缩减资源
  - 资源缩减只有在与代码缩减配合使用时才能发挥作用。在代码缩减器移除所有不使用的代码后，资源缩减器便可确定应用仍要使用的资源。

构建单 ABI 架构的包

默认情况下通过 AS 或者 `./gradlew assembleRelease` 构建出来的 Release 包是包含所有 ABI 架构的，通过分析一个为优化的 Flutter 应用包你会发现 so 是包体积是最大的一项：



apk-size

在这个案例中是使用了多 ABI 架构的安装包，其 so 的大小占整个包大小达到了 83.5%：

CPU	现状
ARMv8	目前主流版本
ARMv7	一些老旧的手机
x86	从 2011 年起, 平板、模拟器用得比较多
x86_64	从 2014 年起, 64 位的平板

目前手机市场上，x86 / x86_64/armeabi/mips / mips6 的占有量应很少，arm64-v8a 作为最新一代架构，是目前的主流，armeabi-v7a 只存在少部分老旧手机。

所以，为了进一步优化包大小我们可以构建出单一架构的安装包，在 Flutter 中我们可以通过以下方式来构建出单一架构的安装包：

Built build/app/outputs/flutter-apk/app-armeabi-v7a-release.apk (6.1MB)
Built build/app/outputs/flutter-apk/app-arm64-v8a-release.apk (6.7MB)
Built build/app/outputs/flutter-apk/app-x86_64-release.apk (6.8MB)

```
cd <flutter应用的android目录>
flutter build apk --split-per-abi
```

默认打包出来的apk包，包含了多种架构的，split 分裂 per 单一 abi 包，可以打包出单一架构的apk包。

- `flutter build`：命令默认会构建出release包
- `--split-per-abi`：表示构建单一架构

运行次命令成功后会看打如下输入出：

```
Removed unused resources: Binary resource data reduced from
518KB to 512KB: Removed 1%
Removed unused resources: Binary resource data reduced from
518KB to 512KB: Removed 1%
Removed unused resources: Binary resource data reduced from
518KB to 512KB: Removed 1%
Running Gradle task 'assembleRelease'...
Running Gradle task 'assembleRelease'... Done
37.1s
✓ Built build/app/outputs/flutter-apk/app-armeabi-v7a-
release.apk (8.5MB).
```

从运行结果不难看出单一架构的安装包仅 8.5MB，比多架构的安装包少了 66% 的体积。