

X

Flutter in action

——闲鱼最佳实践

Product Manager Zoey Fan from 

"This book is full of best practices from Xianyu's experiences building large-scale Flutter apps in production."

Recommended!



闲鱼

闲鱼技术 |  阿里云 开发者社区

X



阿里技术

扫一扫二维码图案，关注我吧



「阿里技术」微信公众号



闲鱼技术微信公众号



阿里云开发者社区

目录

第一章 Flutter 开源工具	1
闲鱼 Flutter 应用框架: Fish Redux	1
AOP for Flutter 开发利器: AspectD	15
“码” 上用 FlutterBoost 开始混合开发吧	26
flutter-boot, 一分钟搞定混合工程搭建!	35
第二章 闲鱼: Flutter 企业级应用实践	42
Flutter & FaaS 云端一体化架构	42
基于 Flutter 的架构演进与创新	49
第三章 混合开发实践指南	59
Flutter Plugin 调用 Native APIs	59
Flutter 混合工程改造实践	75
闲鱼 Flutter 混合工程持续集成的最佳实践	80
Flutter 新锐专家之路: 工程研发体系篇	90
Android Flutter 实践内存初探	103
第四章 Flutter 深入进阶教程	115
一章节教会你如何低成本实现 Flutter 富文本	115
揭秘! 一个高准确率的 Flutter 埋点框架如何设计	121
万万没想到 Flutter 这样外接纹理	128
可定制化的 Flutter 相册组件竟如此简单	137
揭晓闲鱼通过数据提升 Flutter 体验的真相	141
打通前后端逻辑, 客户端 Flutter 代码一天上线	148
流言终结者 – Flutter 和 RN 谁才是更好的跨端开发方案?	159

第一章 Flutter 开源工具

| 闲鱼 Flutter 应用框架: Fish Redux

作者: 闲鱼技术 - 吉丰

开源地址: <https://github.com/alibaba/fish-redux>

3月5日，闲鱼宣布在GitHub上开源Fish Redux，Fish Redux是一个基于Redux数据管理的组装式flutter应用框架，特别适用于构建中大型的复杂应用，它最显著的特征是函数式的编程模型、可预测的状态管理、可插拔的组件体系、最佳的性能表现。下文中，我们将详细介绍Fish Redux的特点和使用过程，以下内容来自InfoQ独家对闲鱼Flutter团队的采访和Fish Redux的开源文档。

开源背景

在闲鱼接入Flutter之初，由于我们的落地的方案希望是从最复杂的几个主链路进行尝试来验证flutter完备性的，而我们的详情整体来讲业务比较复杂，主要体现在两个方面：

- 页面需要集中状态管理，也就是说页面的不同组件共享一个数据来源，数据来源变化需要通知页面所有组件。
- 页面的UI展现形式比较多（如普通详情、闲鱼币详情、社区详情、拍卖详情等），工作量大，所以UI组件需要尽可能复用，也就是说需要比较好的进行组件化切分。

在我们尝试使用市面上已有的框架(google提供的redux以及bloc)的时候发现，没有任何一个框架可以既解决集中状态管理，又能解决UI的组件化的，因为本

身这两个问题有一定的矛盾性（集中 vs 分治）。因此我们希望有一套框架能解决我们的问题，fish redux 应运而生。

fish redux 本身是经过比较多次的迭代的，目前大家看到的版本经过了 3 次比较大的迭代，实际上也是经过了团队比较多的讨论和思考。

第一个版本是基于社区内的 flutter_redux 进行的改造，核心是提供了 UI 代码的组件化，当然问题也非常明显，针对复杂的详情和发布业务，往往业务逻辑很多，无法做到逻辑代码的组件化。

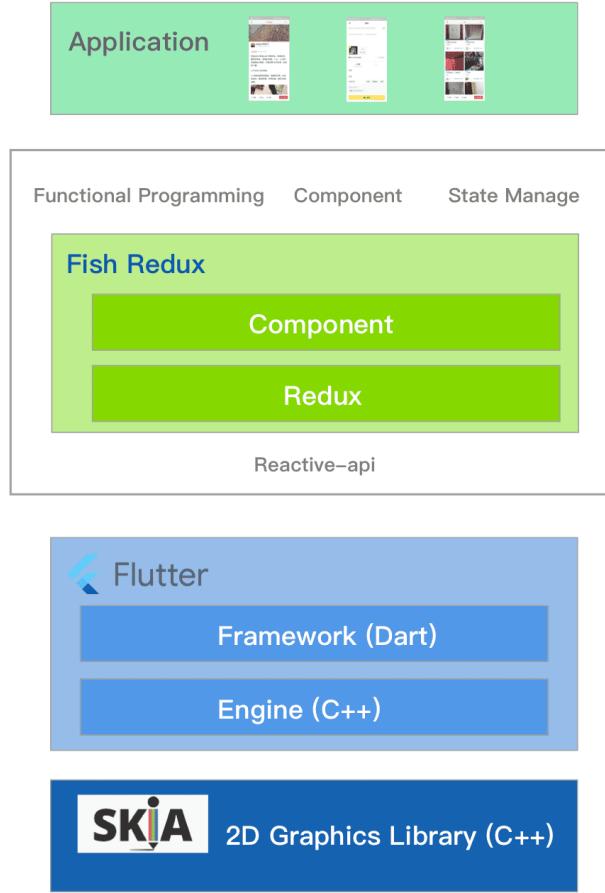
第二个版本针对第一个版本的问题，做出了比较重大的修改，解决了 UI 代码和逻辑代码的分治问题，但同时，按照 redux 的标准，打破了 redux 的原则，对于精益求精的闲鱼团队来讲，不能接受；

因此，在第三个版本进行重构时，我们确立了整体的架构原则与分层要求，一方面按照 reduxjs 的代码进行了 flutter 侧的 redux 实现，将 redux 的原则完整保留下来。另一方面针对组件化的问题，提供了 redux 之上的 component 的封装，并创新的通过这一层的架构设计提供了业务代码分治的能力。

至此，我们完成了 fish redux 的基本设计，但在后续的应用中，发现了业务组装以后的代码性能问题，针对该问题，我们再次提供了对应的 adapter 能力，保障了在长列表场景下的 big cell 问题。目前，fish redux 已经在线上稳定运行超过 3 个月以上，未来，期待 fish redux 给社区带来更多的输入。

Fish Redux 技术解析

分层架构图



架构图：主体自底而上，分两层，每一层用来解决不通层面的问题和矛盾，下面依次来展开。

Redux

Redux 是来自前端社区的一个数据管理框架，对 Native 开发同学来说可能会有一点陌生，我们做一个简单的介绍。

Redux 是做什么的？

Redux 是一个用来做 [可预测][集中式][易调试][灵活性] 的数据管理的框架。所有对数据的增删改查等操作都由 Redux 来集中负责。

Redux 是怎么设计和实现的？

Redux 是一个函数式的数据管理的框架。传统 OOP 做数据管理，往往是定义一些 Bean，每一个 Bean 对外暴露一些 Public-API 用来操作内部数据（充血模型）。

函数式的做法是更上一个抽象的纬度，对数据的定义是一些 Struct（贫血模型），而操作数据的方法都统一到具有相同函数签名 ($T, Action \Rightarrow T$) 的 Reducer 中。

FP:Struct（贫血模型）+ Reducer = OOP:Bean（充血模型）

同时 Redux 加上了 FP 中常用的 Middleware (AOP) 模式和 Subscribe 机制，给框架带了极高的灵活性和扩展性。

贫血模型、充血模型请参考：

https://en.wikipedia.org/wiki/Plain_old_Java_object

Redux 的缺点

Redux 核心仅仅关心数据管理，不关心具体什么场景来使用它，这是它的优点同时也是它的缺点。

在我们实际使用 Redux 中面临两个具体问题：

- Redux 的集中和 Component 的分治之间的矛盾；
- Redux 的 Reducer 需要一层层手动组装，带来的繁琐性和易错性。

Fish Redux 的改良

Fish Redux 通过 Redux 做集中化的可观察的数据管理。然不仅于此，对于传统 Redux 在使用层面上的缺点，在面向端侧 flutter 页面纬度开发的场景中，我们通过更好更高的抽象，做了改良。

一个组件需要定义一个数据 (Struct) 和一个 Reducer。同时组件之间存在着父依赖子的关系。通过这层依赖关系，我们解决了【集中】和【分治】之间的矛盾，同时对 Reducer 的手动层层 Combine 变成由框架自动完成，大大简化了使用 Redux 的困难。我们得到了理想的集中的效果和分治的代码。

对社区标准的 follow

State、Action、Reducer、Store、Middleware 以上概念和社区的 ReduxJS 是完全一致的。我们将原汁原味地保留所有的 Redux 的优势。

如果想对 Redux 有更进一步的理解，请参考：

<https://github.com/reduxjs/redux>

Component

组件是对局部的展示和功能的封装。基于 Redux 的原则，我们对功能细分为修改数据的功能 (Reducer) 和非修改数据的功能 (副作用 Effect)。

于是我们得到了，View、Effect、Reducer 三部分，称之为组件的三要素，分别负责了组件的展示、非修改数据的行为、修改数据的行为。

这是一种面向当下，也面向未来的拆分。在面向当下的 Redux 看来，是数据管理和其他。在面向未来的 UI-Automation 看来是 UI 表达和其他。

UI 的表达对程序员而言即将进入黑盒时代，研发工程师们会把更多的精力放在非修改数据的行为、修改数据的行为上。

组件是对视图的分治，也是对数据的分治。通过逐层分治，我们将复杂的页面和数据切分为相互独立的小模块。这将利于团队内的协作开发。

关于 View

View 仅仅是一个函数签名 : (T,Dispatch,ViewService) => Widget 它主要包含三方面的信息：

- 视图是完全由数据驱动。

- 视图产生的事件 / 回调，通过 Dispatch 发出“意图”，不做具体的实现。
- 需要用到的组件依赖等，通过 ViewService 标准化调用。比如一个典型的符合 View 签名的函数。

```
Widget buildView(
    DetailScreenState state, Dispatch dispatch, ViewService service) {
  return DetailsScreen(
    todo: state.todo,
    onDelete: () =>
        dispatch(ActionCreator.createDetailDeleteTodoAction(state.todo.id)),
    toggleCompleted: (bool isComplete) {
      dispatch(ActionCreator.createDetailUpdateTodoAction(
        state.todo.id,
        state.todo.copyWith(complete: isComplete),
      ));
    },
    editTodoBuilder: (BuildContext buildContext, Todo todo) {
      return AddEditScreen(
        key: ArchSampleKeys.editTodoScreen,
        onSave: (task, note) {
          dispatch(ActionCreator.createDetailUpdateTodoAction(
            todo.id,
            todo.copyWith(
              task: task,
              note: note,
            ),
          ));
        },
        isEditing: true,
        todo: todo,
      ); // AddEditScreen
    },
  ); // DetailsScreen
}
```

关于 Effect

Effect 是对非修改数据行为的标准定义，它是一个函数签名：(Context, Action)
=> Object 它主要包含四方面的信息：

- 接收来自 View 的“意图”，也包括对应的生命周期的回调，然后做出具体的执行。

- 它的处理可能是一个异步函数，数据可能在过程中被修改，所以我们不崇尚持有数据，而通过上下文来获取最新数据。
- 它不修改数据，如果需要，应该发一个 Action 到 Reducer 里去处理。
- 它的返回值仅限于 bool or Future，对应支持同步函数和协程的处理流程。

比如良好的协程的支持：

```
void onShowDeleteDialog(Context<CommentState> ctx, Action action) async {
    final String select = await UX.dialog(
        ctx.context,
        title: '确定删除该留言吗？',
        actions: const <FXAction>[FXAction('确定'), FXAction('取消')],
    );

    if (select == '确定') {
        final PCommentInfo commentInfo = action.payload;
        final PItemInfo itemInfo = ctx.state.itemInfo;

        final PCommentDeleteReq deleteReq = PCommentDeleteReq()
            ..itemId = itemInfo.itemId
            ..commentId = commentInfo.commentId;

        final MtopResponse<Object> resp = await UX.mtop[deleteReq];
        if (resp.isSuccessed) {
            ctx.dispatch(CommentActionCreator.delete(commentInfo));

            UX.toast('留言删除成功');
        } else {
            UX.toast(resperrMsg ?? '删除评论失败，请重试');
        }
    }
}
```

关于 Reducer

Reducer 是一个完全符合 Redux 规范的函数签名 : $(T, Action) \Rightarrow T$ 一些符合签名的 Reducer：

```
DetailScreenState detailScreenReducer(DetailScreenState state, Action action) {
  if (action.type == TODOAction.UpdateTodoAction) {
    final UpdateTodoAction updateTodoAction = action.payload;
    if (state.todo.id == updateTodoAction.id) {
      return state.clone()..todo = updateTodoAction.updatedTodo;
    }
  }
  return state;
}
```

同时我们以显式配置的方式来完成大组件所依赖的小组件、适配器的注册，这份依赖配置称之为 Dependencies。

所以有这样的公式 Component = View + Effect(可选) + Reducer(可选) + Dependencies(可选)。

一个典型的组装：

```
import 'effect.dart';
import 'reducer.dart';
import 'state.dart';
import 'view.dart';

export 'state.dart';

class CountDownBoxComponent extends Component<CountDownBoxState> {
  CountDownBoxComponent()
    : super(
        view: buildCountDownBoxView,
        effect: CountDownBoxEffectBuilder.build(),
        reducer: asReducer(CountDownBoxReducerBuilder.buildMap()),
    );
}
```

通过 Component 的抽象，我们得到了完整的分治，多纬度的复用，更好的解耦。

Adapter

Adapter 也是对局部的展示和功能的封装。它为 ListView 高性能场景而生，它是 Component 实现上的一种变化。

它的目标是解决 Component 模型在 flutter-ListView 的场景下的 3 个问题：

- 1) 将一个”Big-Cell” 放在 Component 里，无法享受 ListView 代码的性能优化；
- 2) Component 无法区分 appear|disappear 和 init|dispose ；
- 3) Effect 的生命周期和 View 的耦合，在 ListView 的场景下不符合直观的预期。

概括的讲，我们想要一个逻辑上的 ScrollView，性能上的 ListView，这样的一种局部展示和功能封装的抽象。做出这样独立一层的抽象是我们看实际的效果，我们对页面不使用框架 Component，使用框架 Component+Adapter 的性能基线对比。

- Reducer is long-lived, Effect is medium-lived, View is short-lived.

我们通过不断的测试做对比，以某 Android 机为例：

- 使用框架前 我们的详情页面的 FPS，基线在 52FPS；
- 使用框架，仅使用 Component 抽象下，FPS 下降到 40，遭遇 “Big-Cell”的陷阱；
- 使用框架，同时使用 Adapter 抽象后，FPS 提升到 53，回到基线以上，有小幅度的提升。

Directory

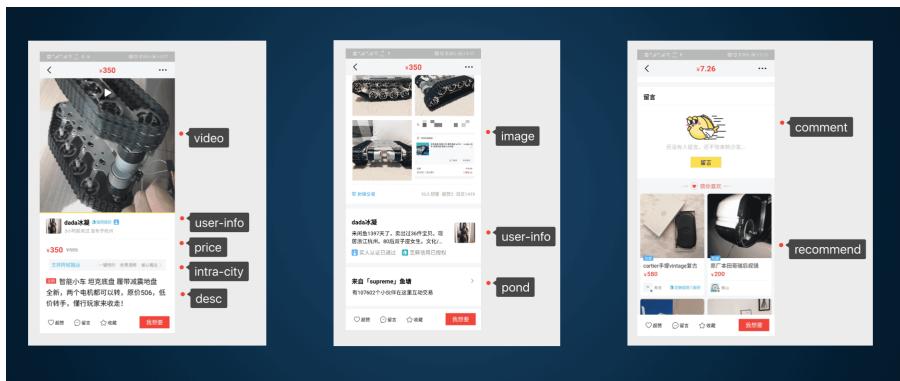
推荐的目录结构会是这样：

```
sample_page
-- action.dart
-- page.dart
-- view.dart
-- effect.dart
-- reducer.dart
-- state.dart
components
sample_component
-- action.dart
-- component.dart
```

```
-- view.dart
-- effect.dart
-- reducer.dart
-- state.dart
```

上层负责组装，下层负责实现，同时会有一个插件提供，便于我们快速填写。

以闲鱼的详情场景为例的组装：

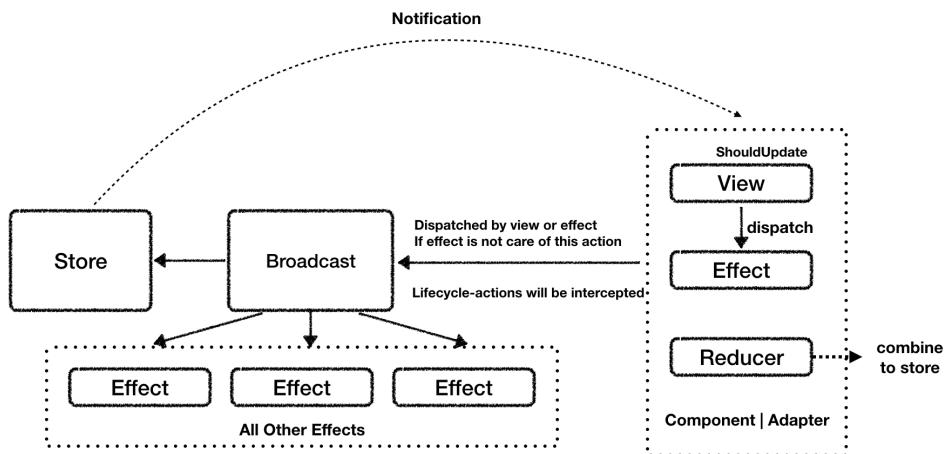


```
29
30 class ItemBodyComponent extends Component<ItemBodyState> {
31   ItemBodyComponent()
32     : super(
33       view: buildItemBody,
34       higherEffect: higherEffect(() => ItemBodyEffectBuilder()),
35       reducer: asReducer(ItemBodyStateReducerBuilder.buildMap()),
36       dependencies: Dependencies<ItemBodyState>(
37         adapter: StaticFlowAdapter<ItemBodyState>(
38           slots: <Dependent<ItemBodyState>>[
39             VideoAdapter().asDependent(videoConnector()),
40             UserInfoComponent().asDependent(userInfoConnector()),
41             ItemPriceComponent().asDependent(itemPriceConnector()),
42             IntraCityComponent().asDependent(intraCityConnector()),
43             SecuredComponent().asDependent(securedConnector()),
44             DescComponent().asDependent(descConnector()),
45             ItemImageComponent().asDependent(itemImageConnector()),
46             OriginDescComponent().asDependent(originDescConnector()),
47             ProductParamComponent().asDependent(productParamConnector()),
48             VisitComponent().asDependent(visitConnector()),
49             XianYuHaoComponent().asDependent(xianYuHaoConnector()),
50             SameMoreComponent().asDependent(sameMoreConnector()),
51             PondComponent().asDependent(pondConnector()),
52             CommentAdapter().asDependent(commentConnector()),
53             RecommendAdapter().asDependent(recommendConnector()),
54             PaddingComponent().asDependent(paddingConnector())
55           ],
56         ),
57       );
58   }
59 }
```

组件和组件之间，组件和容器之间都完全的独立。

Communication Mechanism

- 组件 | 适配器内通信
- 组件 | 适配器间内通信



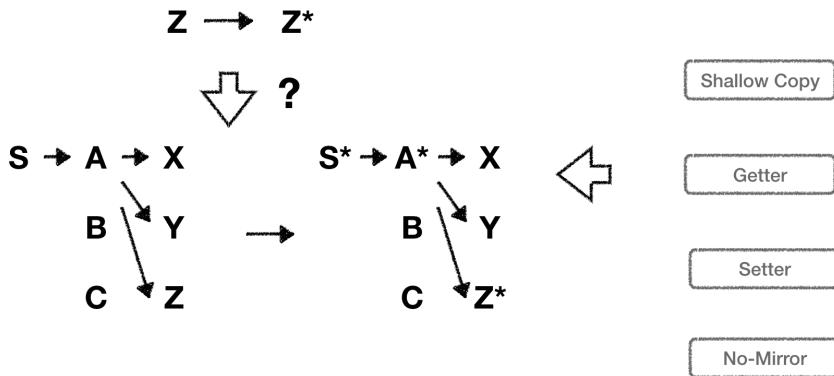
简单的描述：采用的是带有一段优先处理的广播，self-first-broadcast。

发出的 Action，自己优先处理，否则广播给其他组件和 Redux 处理。最终我们通过一个简单而直观的 dispatch 完成了组件内，组件间（父到子，子到父，兄弟间等）的所有的通信诉求。

Refresh Mechanism

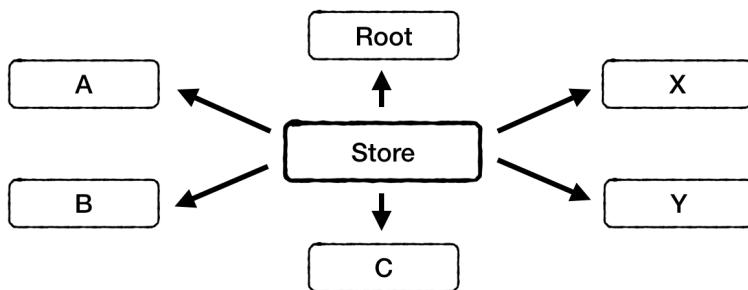
数据刷新

- 局部数据修改，自动层层触发上层数据的浅拷贝，对上层业务代码是透明的。
- 层层的数据的拷贝：
 - 一方面是对 Redux 数据修改的严格的 follow。
 - 另一方面也是对数据驱动展示的严格的 follow。



视图刷新

扁平化通知到所有组件，组件通过 `shouldUpdate` 确定自己是否需要刷新。



Fish Redux 的优点

数据的集中管理

通过 Redux 做集中化的可观察的数据管理。我们将原汁原味地保留所有的 Redux 的优势，同时在 Reducer 的合并上，变成由框架代理自动完成，大大简化了使用 Redux 的繁琐度。

组件的分治管理

组件既是对视图的分治，也是对数据的分治。通过逐层分治，我们将复杂的页面和数据切分为相互独立的小模块。这将利于团队内的协作开发。

View、Reducer、Effect 隔离

将组件拆分成三个无状态的互不依赖的函数。因为是无状态的函数，它更易于编写、调试、测试、维护。同时它带来了更多的组合、复用和创新的可能。

声明式配置组装

组件、适配器通过自由的声明式配置组装来完成。包括它的 View、Reducer、Effect 以及它所依赖的子项。

良好的扩展性

核心框架保持自己的核心的三层关注点，不做核心关注点以外的事情，同时对上层保持了灵活的扩展性。

- 框架甚至没有任何的一行的打印的代码，但我们可通过标准的 Middleware 来观察到数据的流动，组件的变化。
- 在框架的核心三层外，也可以通过 dart 的语言特性 为 Component 或者 Adapter 添加 mixin，来灵活的组合式地增强他们的上层使用上的定制和能力。
- 框架和其他中间件的打通，诸如自动曝光、高可用等，各中间件和框架之间都是透明的，由上层自由组装。

精小、简单、完备

- 它非常小，仅仅包含 1000 多行代码；
- 它使用简单，完成几个小的函数，完成组装，即可运行；
- 它是完备的。

关于未来

开源之后，闲鱼打算通过以下方式来维护 Fish Redux：

- 通过后续的一系列的对外宣传，吸引更多的开发者加入或者使用。目前 Flutter 生态里，应用框架还是空白，有机会成为事实标准；

- 配合后续的一系列的闲鱼 Flutter 移动中间件矩阵做开源；
- 进一步提供，一系列的配套的开发辅助调试工具，提升上层 Flutter 开发效率和体验。

Fish Redux 目前已在阿里巴巴闲鱼技术团队内多场景，深入应用。最后 Talk is cheap, Show me the code，我们今天正式在 GitHub 上开源，更多内容，请到 GitHub 了解。

GitHub 地址：<https://github.com/alibaba/fish-redux>

AOP for Flutter 开发利器：AspectD

作者：闲鱼技术 – 正物

开源地址：<https://github.com/alibaba-flutter/aspectd>

问题背景

随着 Flutter 这一框架的快速发展，有越来越多的业务开始使用 Flutter 来重构或新建其产品。但在我们的实践过程中发现，一方面 Flutter 开发效率高，性能优异，跨平台表现好，另一方面 Flutter 也面临着插件，基础能力，底层框架缺失或者不完善等问题。

举个栗子，我们在实现一个自动化录制回放的过程中发现，需要去修改 Flutter 框架 (Dart 层面) 的代码才能够满足要求，这就会有了对框架的侵入性。要解决这种侵入性的问题，更好地减少迭代过程中的维护成本，我们考虑的首要方案即面向切面编程。

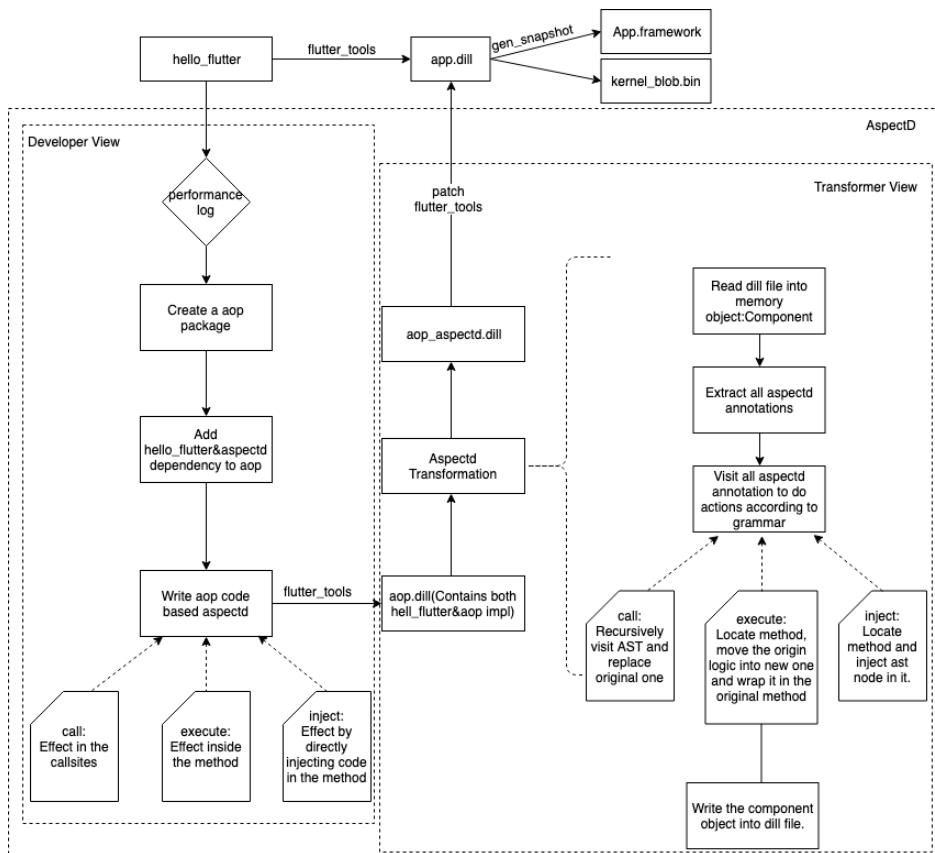
那么如何解决 AOP for Flutter 这个问题呢？本文将重点介绍一个闲鱼技术团队开发的针对 Dart 的 AOP 编程框架 AspectD。

AspectD：面向 Dart 的 AOP 框架

AOP 能力究竟是运行时还是编译时支持依赖于语言本身的特点。举例来说在 iOS 中，Objective C 本身提供了强大的运行时和动态性使得运行期 AOP 简单易用。在 Android 下，Java 语言的特点不仅可以实现类似 AspectJ 这样的基于字节码修改的编译期静态代理，也可以实现 Spring AOP 这样的基于运行时增强的运行期动态代理。那么 Dart 呢？一来 Dart 的反射支持很弱，只支持了检查 (Introspection)，不支持修改 (Modification)；其次 Flutter 为了包大小，健壮性等的原因禁止了反射。

因此，我们设计实现了基于编译期修改的 AOP 方案 AspectD。

设计详图



AOP 设计详图

典型的 AOP 场景

下列 AspectD 代码说明了一个典型的 AOP 使用场景：

```

aop.dart

import 'package:example/main.dart' as app;
import 'aop_impl.dart';

void main()=> app.main();

aop_impl.dart

import 'package:aspectd/aspectd.dart';

```

```

@Aspect()
@pragma("vm:entry-point")
class ExecuteDemo {
    @pragma("vm:entry-point")
    ExecuteDemo();

    @Execute("package:example/main.dart", "_MyHomePageState", "-_incrementCounter")
    @pragma("vm:entry-point")
    void _incrementCounter(PointCut pointcut) {
        pointcut.proceed();
        print('KWLM called!');
    }
}

```

面向开发者的 API 设计

PointCut 的设计

```
@Call("package:app/calculator.dart", "Calculator", "-getCurTime")
```

PointCut 需要完备表征以怎么样的方式 (Call/Execute 等), 向哪个 Library, 哪个类 (Library Method 的时候此项为空), 哪个方法来添加 AOP 逻辑。 PointCut 的数据结构：

```

@pragma('vm:entry-point')
class PointCut {
    final Map<dynamic, dynamic> sourceInfos;
    final Object target;
    final String function;
    final String stubId;
    final List<dynamic> positionalParams;
    final Map<dynamic, dynamic> namedParams;

    @pragma('vm:entry-point')
    PointCut(this.sourceInfos, this.target, this.function, this.stubId, this.
    positionalParams, this.namedParams);

    @pragma('vm:entry-point')
    Object proceed(){
        return null;
    }
}

```

其中包含了源代码信息(如库名,文件名,行号等),方法调用对象,函数名,参数信息等。请注意这里的`@pragma('vm:entry-point')`注解,其核心逻辑在于Tree-Shaking。在AOT(ahead of time)编译下,如果不能被应用主入口(`main`)最终可能调到,那么将被视为无用代码而丢弃。AOP代码因为其注入逻辑的无侵入性,显然是不会被`main`调到的,因此需要此注解告诉编译器不要丢弃这段逻辑。此处的`proceed`方法,类似AspectJ中的`ProceedingJoinPoint.proceed()`方法,调用`pointcut.proceed()`方法即可实现对原始逻辑的调用。原始定义中的`proceed`方法体只是个空壳,其内容将会被在运行时动态生成。

Advice 的设计

```
@pragma("vm:entry-point")
Future<String> getCurTime(PointCut pointcut) async{
    ...
    return result;
}
```

此处的`@pragma("vm:entry-point")`效果同a中所述,pointCut对象作为参数传入AOP方法,使开发者可以获得源代码调用信息的相关信息,实现自身逻辑或者是通过`pointcut.proceed()`调用原始逻辑。

Aspect 的设计

```
@Aspect()
@pragma("vm:entry-point")
class ExecuteDemo {
    @pragma("vm:entry-point")
    ExecuteDemo();
    ...
}
```

Aspect的注解可以使得`ExecuteDemo`这样的AOP实现类被方便地识别和提取,也可以起到开关的作用,即如果希望禁掉此段AOP逻辑,移除`@Aspect`注解即可。

AOP 代码的编译

包含原始工程中的 main 入口

从上文可以看到，aop.dart 引入 `import 'package:example/main.dart' as app;`，这使得编译 aop.dart 时可包含整个 example 工程的所有代码。### Debug 模式下的编译

在 aop.dart 中引入 `import 'aop_impl.dart';` 这使得 aop_impl.dart 中内容即便不被 aop.dart 显式依赖，也可以在 Debug 模式下被编译进去。

Release 模式下的编译

在 AOT 编译 (Release 模式下)，Tree-Shaking 逻辑使得当 aop_impl.dart 中的内容没有被 aop 中 main 调用时，其内容将不会编译到 dll 中。通过添加 `@pragma("vm:entry-point")` 可以避免其影响。

当我们用 AspectD 写出 AOP 代码，透过编译 aop.dart 生成中间产物，使得 dll 中既包含了原始项目代码，也包含了 AOP 代码后，则需要考虑如何对其修改。在 AspectJ 中，修改是通过对 Class 文件进行操作实现的，在 AspectD 中，我们则对 dll 文件进行操作。

Dll 操作

dll 文件，又称为 Dart Intermediate Language，是 Dart 语言编译中的一个概念，无论是 Script Snapshot 还是 AOT 编译，都需要 dll 作为中间产物。

Dll 的结构

我们可以通过 dart sdk 中的 vm package 提供的 dump_kernel.dart 打印出 dll 的内部结构。

```
dart bin/dump_kernel.dart /Users/kylewong/Codes/AOP/aspectd/example/aop/build/app.dll /Users/kylewong/Codes/AOP/aspectd/example/aop/build/app.dll.txt
```

```

13271     library from "package:aop/aop.dart" as aop {
13272
13273     import "package:example/main.dart" as app;
13274     import "package:aop/aop_impl.dart";
13275
13276     static method main() void
13277     {
13278         return main:=main();
13279     }
13280
13281     library from "package:aop/aop_impl.dart" as aop2 {
13282
13283         import "package:aspectd/aspectd.dart";
13284
13285         @asp::Aspect:()
13286         @core::pragma: [\"mentry-point\"]
13287         class ExecuteDemo extends core::Object {
13288             @core::pragma: [\"mentry-point\"]
13289             constructor () = aop2::ExecuteDemo
13290             : super core::Object::();
13291
13292             @asp::Execute: [\"package:example/main.dart\", \"_MyHomePageState\", \"_incrementCounter\"]
13293             @core::pragma: [\"mentry-point\"]
13294             method _incrementCounter(poi::Pointcut pointcut) void {
13295                 pointcut{poi::PointCut::proceed()};
13296                 core::print(\"KWM called!\");
13297             }
13298         }
13299
13300     library from "package:example/main.dart" as main {
13301
13302         import "package:flutter/material.dart";
13303
13304         class MyApp extends fra::StatelessWidget {
13305             synthetic constructor <()> main::MyApp
13306             : super fra::StatelessWidget::();
13307
13308             @core::override
13309             method build(fra::BuildContext context) fra::Widget {
13310                 return new app::MaterialApp<(title: \"Flutter Demo\", theme: the::ThemeData::(primarySwatch: col3::Colors::blue), home: new main::MyHomePage<(title: \"Flutter Demo Home Page\"))>;
13311             }
13312
13313         class MyHomePage extends fra::StatefulWidget {
13314             final field core::String title;
13315             constructor <(key::Key key = null, core::String title = null)> main::MyHomePage
13316             : main::MyHomePage::title = title, super fra::StatefulWidget::(key: key);
13317
13318             @core::override
13319             method createState() main::_MyHomePageState
13320             return new main::_MyHomePageState::();
13321
13322     }

```

Dill 变换

dart 提供了一种 Kernel to Kernel Transform 的方式，通过对 dill 文件的递归式 AST 遍历，实现对 dill 的变换。

基于开发者编写的 AspectD 注解，AspectD 的变换部分可以提取出是哪些库 / 类 / 方法需要添加怎样的 AOP 代码，再在 AST 递归的过程中通过对目标类的操作，实现 Call/Execute 这样的功能。

一个典型的 Transform 部分逻辑如下所示：

```

@Override
MethodInvocation visitMethodInvocation(MethodInvocation MethodInvocation) {
    MethodInvocation.transformChildren(this);
    Node node = MethodInvocation.interfaceTargetReference?.node;
    String uniqueKeyForMethod = null;
    if (node is Procedure) {
        Procedure procedure = node;
        Class cls = procedure.parent as Class;
        String procedureImportUri = cls.reference.canonicalName.parent.name;
        uniqueKeyForMethod = AspectdItemInfo.uniqueKeyForMethod(
            procedureImportUri, cls.name, MethodInvocation.name.name, false, null);
    }
}

```

```

else if(node == null) {
    String importUri = methodInvocation?.interfaceTargetReference?.
canonicalName?.reference?.canonicalName?.nonRootTop?.name;
    String clsName = methodInvocation?.interfaceTargetReference?.
canonicalName?.parent?.parent?.name;
    String methodName = methodInvocation?.interfaceTargetReference?.
canonicalName?.name;
    uniqueKeyForMethod = AspectdItemInfo.uniqueKeyForMethod(
        importUri, clsName, methodName, false, null);
}
if(uniqueKeyForMethod != null) {
    AspectdItemInfo aspectdItemInfo = _aspectdInfoMap[uniqueKeyForMethod];
    if (aspectdItemInfo?.mode == AspectdMode.Call &&
        !_transformedInvocationSet.contains(methodInvocation) &&
        AspectdUtils.checkIfSkipAOP(aspectdItemInfo, _curLibrary) == false) {
        return transformInstanceMethodInvocation(
            methodInvocation, aspectdItemInfo);
    }
}
return methodInvocation;
}

```

通过对于 `dill` 中 AST 对象的遍历 (此处的 `visitMethodInvocation` 函数), 结合开发者书写的 AspectD 注解 (此处的 `_aspectdInfoMap` 和 `AspectdItemInfo`), 可以对原始的 AST 对象 (此处 `methodInvocation`) 进行变换, 从而改变原始的代码逻辑, 即 Transform 过程。

AspectD 支持的语法

不同于 AspectJ 中提供的 Before, 在 AspectD 中, 只有一种统一的抽象即 Around。从是否修改原始方法内部而言, 有 Call 和 Execute 两种, 前者的 PointCut 是调用点, 后者的 PointCut 则是执行点。### Call

```

import 'package:aspectd/aspectd.dart';

@Aspect()
@pragma("vm:entry-point")
class CallDemo{
    @Call("package:app/calculator.dart", "Calculator", "-getCurTime")
    @pragma("vm:entry-point")
    Future<String> getCurTime(PointCut pointcut) async{
        print('Aspectd:KWLMO2');
    }
}

```

```

        print('${pointcut.sourceInfos.toString()}');
        Future<String> result = pointcut.proceed();
        String test = await result;
        print('Aspectd:KWLM03');
        print('${test}');
        return result;
    }
}

```

Execute

```

import 'package:aspectd/aspectd.dart';

@Aspect()
@pragma("vm:entry-point")
class ExecuteDemo{
    @Execute("package:app/calculator.dart","Calculator","-getCurTime")
    @pragma("vm:entry-point")
    Future<String> getCurTime(PointCut pointcut) async{
        print('Aspectd:KWLM12');
        print('${pointcut.sourceInfos.toString()}');
        Future<String> result = pointcut.proceed();
        String test = await result;
        print('Aspectd:KWLM13');
        print('${test}');
        return result;
    }
}

```

Inject

仅支持 Call 和 Execute，对于 Flutter(Dart) 而言显然很是单薄。一方面 Flutter 禁止了反射，退一步讲，即便 Flutter 开启了反射支持，依然很弱，并不能满足需求。举个典型的场景，如果需要注入的 dart 代码里，x.dart 文件的类 y 定义了一个私有方法 m 或者成员变量 p，那么在 aop_impl.dart 中是没有办法对其访问的，更不用说多个连续的私有变量属性获得。另一方面，仅仅对方法整体进行操作可能是不够的，我们可能需要在方法的中间插入处理逻辑。为了解决这一问题，AspectD 设计了一种语法 Inject，参见下面的例子：flutter 库中包含了一下这段手势相关代码：

```

@Override
Widget build(BuildContext context) {

```

```

    final Map<Type, GestureRecognizerFactory> gestures = <Type,
GestureRecognizerFactory>{};

    if (onTapDown != null || onTapUp != null || onTap != null || onTapCancel
!= null) {
        gestures[TapGestureRecognizer] = GestureRecognizerFactoryWithHandlers
<TapGestureRecognizer>(
            () => TapGestureRecognizer(debugOwner: this),
            (TapGestureRecognizer instance) {
                instance
                    ..onTapDown = onTapDown
                    ..onTapUp = onTapUp
                    ..onTap = onTap
                    ..onTapCancel = onTapCancel;
            },
        );
    }
}

```

如果我们想要在 onTapCancel 之后添加一段对于 instance 和 context 的处理逻辑，Call 和 Execute 是不可行的，而使用 Inject 后，只需要简单的几句即可解决：

```

import 'package:aspectd/aspectd.dart';

@Aspect()
@pragma("vm:entry-point")
class InjectDemo{
    @Inject("package:flutter/src/widgets/gesture_detector.dart", "GestureDetector",
"-build", lineNumber:452)
    @pragma("vm:entry-point")
    static void onTapBuild() {
        Object instance; //Aspectd Ignore
        Object context; //Aspectd Ignore
        print(instance);
        print(context);
        print('Aspectd:KWL25');
    }
}

```

通过上述的处理逻辑，经过编译构建后的 dll 中的 GestureDetector.build 方法如下所示：

```

kernel_blob.bin.txt x
8     @core2::override
9     method build(fra::Widget context) {
10       final core2::Map<core2::Type, ges::GestureRecognizerFactory<rec::GestureRecognizer>> gestures = <core2::Type, ges::GestureRecognizerFactory<
11         rec::GestureRecognizer>>{};
12       if(!this.{ges::GestureDetector::onTapDown}.(core2::Object::==)(null) || !this.{ges::GestureDetector::onTapUp}.(core2::Object::==)(null) ||
13         this.{ges::GestureDetector::onTap}.(core2::Object::==)(null) || !this.{ges::GestureDetector::onTapCancel}.(core2::Object::==)(null)) {
14         ges::GestureRecognizerFactory<rec::GestureRecognizer> ges::GestureRecognizerFactory<rec::GestureRecognizer>((ges::GestureDetector this),
15         TapGestureRecognizer recognizer = new TapGestureRecognizer(debugger: this), (tap::TapGestureRecognizer instance) core2::Null {
16           let final tap::TapGestureRecognizer #t3924 = instance in let final dynamic #t3926 = #t3925 = #t3924, {tap::TapGestureRecognizer::onTapDown} =
17             this.{ges::GestureDetector::onTapDown} in let final dynamic #t3927 = #t3924, {tap::TapGestureRecognizer::onTapUp} =
18               this.{ges::GestureDetector::onTapUp} in let final dynamic #t3928 = #t3924, {tap::TapGestureRecognizer::onTap} =
19                 this.{ges::GestureDetector::onTap} in let final dynamic #t3929 = #t3924, {tap::TapGestureRecognizer::onTapCancel} =
20                   this.{ges::GestureDetector::onTapCancel} in #t3924;
21         core2::print(instance);
22         core2::print(context);
23         core2::print("Aspectd:KWL25");
24       });
25     }
26   }

```

此外，Inject 的输入参数相对于 Call/Execute 而言，多了一个 lineNumber 的命名参数，可用于指定插入逻辑的具体行号。

构建流程支持

虽然我们可以通过编译 aop.dart 达到同时编译原始工程代码和 AspectD 代码到 dll 文件，再通过 Transform 实现 dll 层次的变换实现 AOP，但标准的 flutter 构建（即 flutter_tools）并不支持这个过程，所以还是需要对构建过程做细微修改。在 AspectJ 中，这一过程是由非标准 Java 编译器的 Ajc 来实现的。在 AspectD 中，通过对 flutter_tools 打上应用 Patch，可以实现对于 AspectD 的支持。

```

dart kylewong@KyleWongdeMacBook-Pro fluttermaster % git apply
- -3way /Users/kylewong/Codes/AOP/aspectd/0001-aspectd.patch
kylewong@KyleWongdeMacBook-Pro fluttermaster % rm bin/cache/
flutter_tools.stamp kylewong@KyleWongdeMacBook-Pro fluttermas-
ter % flutter doctor -v Building flutter tool...

```

实战与思考

基于 AspectD，我们在实践中成功地移除了所有对于 Flutter 框架的侵入性代码，实现了同有侵入性代码同样的功能，支撑上百个脚本的录制回放与自动化回归稳定可靠运行。

从 AspectD 的角度看，Call/Execute 可以帮助我们便捷实现诸如性能埋点（关键方法的调用时长），日志增强（获取某个方法具体是在什么地方被调用到的详细信息），Doom 录制回放（如随机数序列的生成记录与回放）等功能。Inject 语法则更

为强大，可以通过类似源代码诸如的方式，实现逻辑的自由注入，可以支持诸如 App 录制与自动化回归（如用户触摸事件的录制与回放）等复杂场景。

进一步来说，AspectD 的原理基于 Dill 变换，有了 Dill 操作这一利器，开发者可以自由地对 Dart 编译产物进行操作，而且这种变换面向的是近乎源代码级别的 AST 对象，不仅强大而且可靠。无论是做一些逻辑替换，还是是 Json<--> 模型转换等，都提供了一种新的视角与可能。

写在最后

AspectD 作为闲鱼技术团队新开发的面向 Flutter 的 AOP 框架，已经可以支持主流的 AOP 场景并在 Github 开源，欢迎使用。[Aspectd for Flutter](#) 如果你在使用过程中，有任何问题或者建议，欢迎提 [issue](#) 或者 [PR](#).

“码”上用 FlutterBoost 开始混合开发吧

作者：闲鱼技术 - 福居

开源地址：https://github.com/alibaba/flutter_boost

为什么需要混合方案

具有一定规模的 App 通常有一套成熟通用的基础库，尤其是阿里系 App，一般需要依赖很多体系内的基础库。那么使用 Flutter 重新从头开发 App 的成本和风险都较高。所以在 Native App 进行渐进式迁移是 Flutter 技术在现有 Native App 进行应用的稳健型方式。闲鱼在实践中沉淀出一套自己的混合技术方案。在此过程中，我们跟 Google Flutter 团队进行着密切的沟通，听取了官方的一些建议，同时也针对我们业务具体情况进行方案的选型以及具体的实现。

官方提出的混合方案

基本原理

Flutter 技术链主要由 C++ 实现的 Flutter Engine 和 Dart 实现的 Framework 组成（其配套的编译和构建工具我们这里不参与讨论）。Flutter Engine 负责线程管理，Dart VM 状态管理和 Dart 代码加载等工作。而 Dart 代码所实现的 Framework 则是业务接触到的主要 API，诸如 Widget 等概念就是在 Dart 层面 Framework 内容。

一个进程里面最多只会初始化一个 Dart VM。然而一个进程可以有多个 Flutter Engine，多个 Engine 实例共享同一个 Dart VM。

我们来看具体实现，在 iOS 上面每初始化一个 FlutterViewController 就会有一个引擎随之初始化，也就意味着会有新的线程（理论上线程可以复用）去跑 Dart 代码。Android 类似的 Activity 也会有类似的效果。如果你启动多个引擎实例，注意此时 Dart VM 依然是共享的，只是不同 Engine 实例加载的代码跑在各自独立的 Isolate。

官方建议

引擎深度共享

在混合方案方面，我们跟 Google 讨论了可能的一些方案。Flutter 官方给出的建议是从长期来看，我们应该支持在同一个引擎支持多窗口绘制的能力，至少在逻辑上做到 FlutterViewController 是共享同一个引擎的资源的。换句话说，我们希望所有绘制窗口共享同一个主 Isolate。

但官方给出的长期建议目前来说没有很好的支持。

多引擎模式

我们在混合方案中解决的主要问题是如何去处理交替出现的 Flutter 和 Native 页面。Google 工程师给出了一个 Keep It Simple 的方案：对于连续的 Flutter 页面（Widget）只需要在当前 FlutterViewController 打开即可，对于间隔的 Flutter 页面我们初始化新的引擎。

例如，我们进行下面一组导航操作：Flutter Page1 -> Flutter Page2 -> Native Page1 -> Flutter Page3

我们只需要在 Flutter Page1 和 Flutter Page3 创建不同的 Flutter 实例即可。

这个方案的好处就是简单易懂，逻辑清晰，但是也有潜在的问题。如果一个 Native 页面一个 Flutter 页面一直交替进行的话，Flutter Engine 的数量会线性增加，而 Flutter Engine 本身是一个比较重的对象。

多引擎模式的问题

- 冗余的资源问题。多引擎模式下每个引擎之间的 Isolate 是相互独立的。在逻辑上这并没有什么坏处，但是引擎底层其实是维护了图片缓存等比较消耗内存的对象。想象一下，每个引擎都维护自己一份图片缓存，内存压力将会非常大。
- 插件注册的问题。插件依赖 Messenger 去传递消息，而目前 Messenger 是由 FlutterViewController (Activity) 去实现的。如果你有多个 FlutterViewController，插件的注册和通信将会变得混乱难以维护，消息的传递的源头和目标也变得不可控。

- Flutter Widget 和 Native 的页面差异化问题。Flutter 的页面是 Widget，Native 的页面是 VC。逻辑上来说我们希望消除 Flutter 页面与 Native 页面的差异，否则在进行页面埋点和其它一些统一操作的时候都会遇到额外的复杂度。
- 增加页面之间通信的复杂度。如果所有 Dart 代码都运行在同一个引擎实例，它们共享一个 Isolate，可以用统一的编程框架进行 Widget 之间的通信，多引擎实例也让这件事情更加复杂。

因此，综合多方面考虑，我们没有采用多引擎混合方案。

现状与思考

前面我们提到多引擎存在一些实际问题，所以闲鱼目前采用的混合方案是共享同一个引擎的方案。这个方案基于这样一个事实：任何时候我们最多只能看到一个页面，当然有些特定的场景你可以看到多个 ViewController，但是这些特殊场景我们这里不讨论。

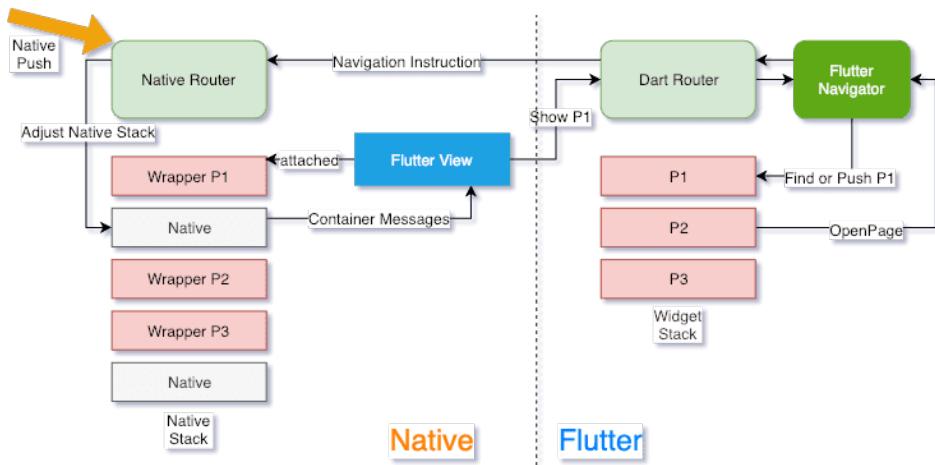
我们可以这样简单去理解这个方案：我们把共享的 Flutter View 当成一个画布，然后用一个 Native 的容器作为逻辑的页面。每次在打开一个容器的时候我们通过通信机制通知 Flutter View 绘制出当前的逻辑页面，然后将 Flutter View 放到当前容器里面。

老方案在 Dart 侧维护了一个 Navigator 栈的结构。栈数据结构特点就是每次只能从栈顶去操作页面，每一次在查找逻辑页面的时候如果发现页面不在栈顶那么需要往回 Pop。这样中途 Pop 掉的页面状态就丢失了。这个方案无法支持同时存在多个平级逻辑页面的情况，因为你在页面切换的时候必须从栈顶去操作，无法再保持状态的同时进行平级切换。

举个例子：有两个页面 A, B，当前 B 在栈顶。切换到 A 需要把 B 从栈顶 Pop 出去，此时 B 的状态丢失，如果想切回 B，我们只能重新打开 B 之前页面的状态无法维持住。这也是老方案最大的一个局限。

如在 pop 的过程当中，可能会把 Flutter 官方的 Dialog 进行误杀。这也一个问题。

而且基于栈的操作我们依赖对 Flutter 框架的一个属性修改，这让这个方案具有了侵入性的特点。这也是我们需要解决的一个问题。



具体细节，大家可以参考老方案开源项目地址：

https://github.com/alibaba-flutter/hybrid_stack_manager

第二代混合技术方案 FlutterBoost

重构计划

在闲鱼推进 Flutter 化过程当中，更加复杂的页面场景逐渐暴露了老方案的局限性和一些问题。所以我们启动了代号 FlutterBoost（向 C++ Boost 致敬）的新混合技术方案。这次新的混合方案我们的主要目标有：

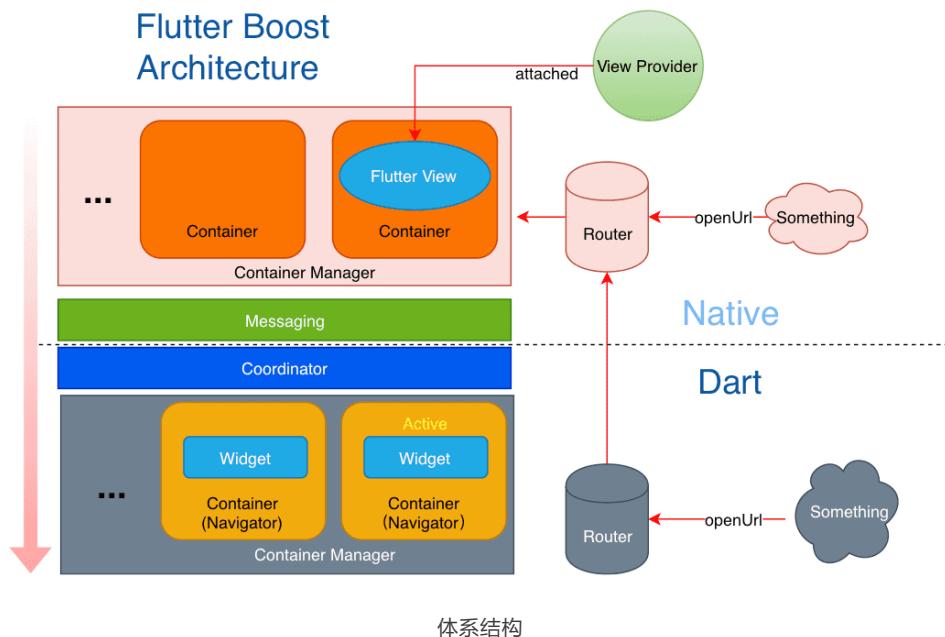
- 可复用通用型混合方案
 - 支持更加复杂的混合模式。比如支持主页 Tab 这种情况
 - 无侵入性方案：不再依赖修改 Flutter 的方案
 - 支持通用页面生命周期
 - 统一明确的设计概念

跟老方案类似，新的方案还是采用共享引擎的模式实现。主要思路是由 Native

容器 Container 通过消息驱动 Flutter 页面容器 Container，从而达到 Native Container 与 Flutter Container 的同步目的。我们希望做到 Flutter 渲染的内容是由 Native 容器去驱动的。

简单的理解，我们想做到把 Flutter 容器做成浏览器的感觉。填写一个页面地址，然后由容器去管理页面的绘制。在 Native 侧我们只需要关心如果初始化容器，然后设置容器对应的页面标志即可。

主要概念



Native 层概念

- **Container:** Native 容器，平台 Controller，Activity，ViewController
- **Container Manager:** 容器的管理者
- **Adaptor:** Flutter 是适配层
- **Messaging:** 基于 Channel 的消息通信

Dart 层概念

- Container: Flutter 用来容纳 Widget 的容器，具体实现为 Navigator 的派生类
- Container Manager: Flutter 容器的管理，提供 show, remove 等 API
- Coordinator: 协调器，接受 Messaging 消息，负责调用 Container Manager 的状态管理。
- Messaging: 基于 Channel 的消息通信

关于页面的理解

在 Native 和 Flutter 表示页面的对象和概念是不一致的。在 Native，我们对于页面的概念一般是 ViewController, Activity。而对于 Flutter 我们对于页面的概念是 Widget。我们希望可统一页面的概念，或者说弱化抽象掉 Flutter 本身的 Widget 对应的页面概念。换句话说，当一个 Native 的页面容器存在的时候，FlutterBoost 保证一定会有一个 Widget 作为容器的内容。所以我们在理解和进行路由操作的时候都应该以 Native 的容器为准，Flutter Widget 依赖于 Native 页面容器的状态。

那么在 FlutterBoost 的概念里说到页面的时候，我们指的是 Native 容器和它所附属的 Widget。所有页面路由操作，打开或者关闭页面，实际上都是对 Native 页面容器的直接操作。无论路由请求来自何方，最终都会转发给 Native 去实现路由操作。这也是接入 FlutterBoost 的时候需要实现 Platform 协议的原因。

另一方面，我们无法控制业务代码通过 Flutter 本身的 Navigator 去 push 新的 Widget。对于业务不通过 FlutterBoost 而直接使用 Navigator 操作 Widget 的情况，包括 Dialog 这种非全屏 Widget，我们建议是业务自己负责管理其状态。这种类型的 Widget 不属于 FlutterBoost 所定义的页面概念。

理解这里的页面概念，对于理解和使用 FlutterBoost 至关重要。

与老方案主要差别

前面我们提到老方案在 Dart 层维护单个 Navigator 栈结构用于 Widget 的切换。而新的方案则是在 Dart 侧引入了 Container 的概念，不再用栈的结构去维护现有的页面，而是通过扁平化 key-value 映射的形式去维护当前所有的页面，每个页

面拥有一个唯一的 id。这种结构很自然的支持了页面的查找和切换，不再受制于栈顶操作的问题，之前的一些由于 pop 导致的问题迎刃而解。同时也不再需要依赖修改 Flutter 源码的形式去进行实现，除去了实现的侵入性。

那这是如何做到的呢？

多 Navigator 的实现

Flutter 在底层提供了让你自定义 Navigator 的接口，我们自己实现了一个管理多个 Navigator 的对象。当前最多只会有一个可见的 Flutter Navigator，这个 Navigator 所包含的页面也就是我们当前可见容器所对应的页面。

Native 容器与 Flutter 容器 (Navigator) 是一一对应的，生命周期也是同步的。当一个 Native 容器被创建的时候，Flutter 的一个容器也被创建，它们通过相同的 id 关联起来。当 Native 的容器被销毁的时候，Flutter 的容器也被销毁。Flutter 容器的状态是跟随 Native 容器，这也就是我们说的 Native 驱动。由 Manager 统一管理切换当前在屏幕上展示的容器。

我们用一个简单的例子描述一个新页面创建的过程：

1. 创建 Native 容器 (iOS ViewController, Android Activity or Fragment)。
2. Native 容器通过消息机制通知 Flutter Coordinator 新的容器被创建。
3. Flutter Container Manager 进而得到通知，负责创建出对应的 Flutter 容器，并且在其中装载对应的 Widget 页面。
4. 当 Native 容器展示到屏幕上时，容器发消息给 Flutter Coordinator 通知要展示页面的 id.
5. Flutter Container Manager 找到对应 id 的 Flutter Container 并将其设置为前台可见容器。

这就是一个新页面创建的主要逻辑，销毁和进入后台等操作也类似有 Native 容器事件去进行驱动。

总结

目前 FlutterBoost 已经在生产环境支撑着在闲鱼客户端中所有的基于 Flutter 开发业务，为更加负复杂的混合场景提供了支持。同时也解决了一些历史遗留问题。

我们在项目启动之初就希望 FlutterBoost 能够解决 Native App 混合模式接入 Flutter 这个通用问题。所以我们把它做成了一个可复用的 Flutter 插件，希望吸引更多感兴趣的朋友参与到 Flutter 社区的建设。我们的方案可能不是最好的，这个方案距离完美还有很大的距离，我们希望通过多分享交流以推动 Flutter 技术社区的发展与建设。我们更希望看到社区能够涌现出更加优秀的组件和方案。

在有限篇幅中，我们分享了闲鱼在 Flutter 混合技术方案中积累的经验和代码。欢迎有兴趣的同学能够积极与我们一起交流学习。

扩展补充

性能相关

在两个 Flutter 页面进行切换的时候，因为我们只有一个 Flutter View 所以需要对上一个页面进行截图保存，如果 Flutter 页面多截图会占用大量内存。这里我们采用文件内存二级缓存策略，在内存中最多只保存 2–3 个截图，其余的写入文件按需加载。这样我们可以在保证用户体验的同时在内存方面也保持一个较为稳定的水平。

页面渲染性能方面，Flutter 的 AOT 优势展露无遗。在页面快速切换的时候，Flutter 能够很灵敏的相应页面的切换，在逻辑上创造出一种 Flutter 多个页面的感觉。

Release 1.0 支持

项目开始的时候我们基于闲鱼目前使用的 Flutter 版本进行开发，而后进行了 Release 1.0 兼容升级测试目前没有发现问题。

接入

只要是集成了 Flutter 的项目都可以用官方依赖的方式非常方便的以插件形式引

入 FlutterBoost，只需要对工程进行少量代码接入即可完成接入。详细接入文档，请参阅 GitHub 主页官方项目文档。

现已开源

目前，第二代混合栈已全面闲鱼全面应用。我们非常乐意将沉淀的技术回馈给社区。欢迎大家一起贡献，一起交流，携手共建 Flutter 社区。同时第三代，即将会跟大家见面，第三代在第二代基础上进行了重构。目前正在测试中。

项目开源地址: https://github.com/alibaba/flutter_boost

flutter-boot，一分钟搞定混合工程搭建！

作者：兴往 向志明 马引

开源地址：<https://github.com/alibaba-flutter/flutter-boot>

背景

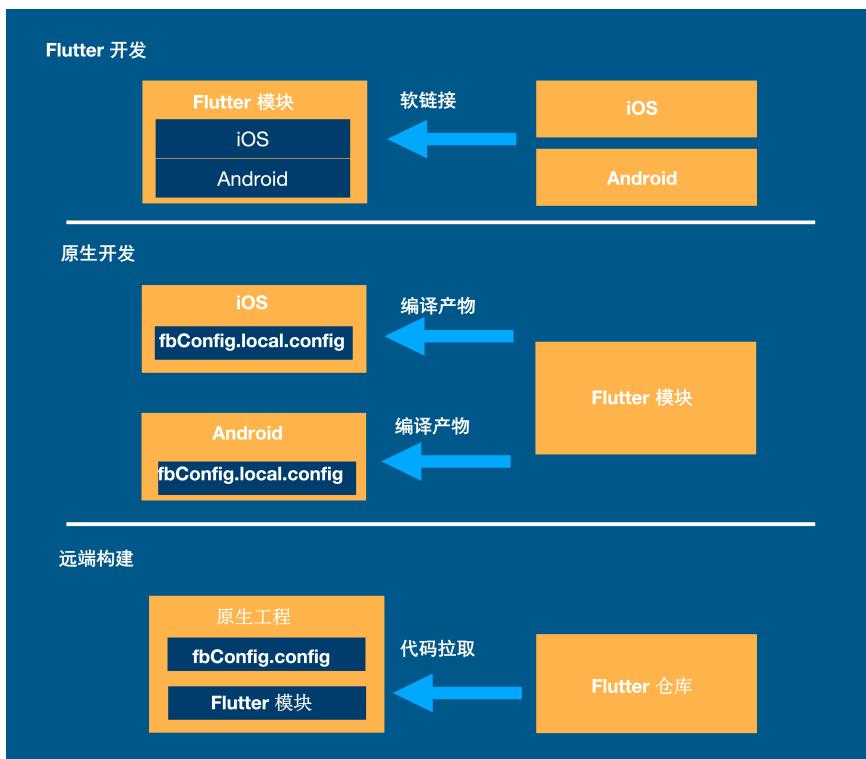
Flutter，从诞生起到现在，已经成为了跨端开发的领跑者，国内外越来越多的公司走上了 flutter 探索之路。Flutter 的主要开发模式分成两种，一种是独立 app 的模式，以 flutter 为主，原生工程会被包含在 flutter 工程下；另一种是让 flutter 以模块 (flutter module) 的形式存在，分别集成在已有的 iOS 和 android 原生应用下，此时原生工程可以在任何的目录结构下，和 flutter 工程地址不产生关联，但需要在原生工程结构中声明 flutter 工程的本地地址。

闲鱼应用在 flutter 能够以模块形式存在前，进行了很长时间的混合 app 架构的探索，对我们的原生工程进行了比较多的改动，在官方推出 flutter 模块模式后，我们进行了大量调研，最终推出了一套开箱即用的混合工程脚手架 flutter-boot，帮助大家快速搭建混合工程。

简介

flutter-boot 核心解决了混合开发模式下的两个问题：flutter 混合开发的工程化设计和混合栈。那 flutter-boot 是如何解决的呢？

首先在工程化设计的问题上，flutter-boot 建立了一套标准的工程创建流程和友好的交互命令，当流程执行完成后，即拥有了混合开发的标准工程结构，这一套工程结构能够帮助我们同时拥有 flutter 和 native (原生) 两种开发视角，本地 flutter 开发和云端 flutter 构建两种 flutter 集成模式，其效果如图：



另外在混合栈的问题上，flutter-boot 能自动注入混合栈依赖，同时将核心的混合栈接入代码封装后注入到原生工程内，在用户按提示插入简单几行模版代码后，即可看到混合栈的效果。

使用 flutter-boot 搭建的混合工程，开箱即可使用，接下来让我们了解下 flutter-boot 解决这些问题的详细过程。

工程化设计

了解官方的 Add Flutter to existing apps 项目

在了解 flutter-boot 的工程化设计细节前，我们需要对 Google 官方提供的 Add Flutter to existing apps 方案有一个初步的了解。

Add Flutter to existing apps 项目会引导我们以 module 的形式创建 flutter, module 形态的 flutter 的工程结构如下：

```
```
some/path/
 my_flutter/
 lib/main.dart
 .ios/
 .android/
```

```

在官方的工程结构下，.ios 和.android 是运行 flutter 时的模版工程，在 flutter 工程目录下运行时即通过这两个工程来启动应用。那我们如何让原生工程和产生关联呢？这里的关联会分成三个部分，分别是 flutter 的 framework，flutter 的业务代码，和 flutter 的插件库。其中 flutter 插件库分成 flutter plugin native（即插件原生代码）和 flutter plugin dart（即插件的 dart 代码）两个部分。这四部分的差异在于：

模块	模块数量	内容变更频率	支持调试
----	----	----	----
flutter framework	唯一	低	否
flutter plugin native	高频变更	低	是
flutter plugin dart	高频变更	低	是
flutter 业务代码	唯一	高	是

因此 flutter framework 只需要在依赖管理中声明即可，flutter plugin native 可以直接以源码的方式集成，flutter plugin dart 只有在被业务代码引用时才有效，因此和业务代码一样，需要支持 dart 代码的调试模式和发布模式，因此 dart 代码的关联会侵入到 app 的构建环节，根据 app 构建的模式来决定 dart 代码的构建模式。具体的实现，拿 iOS 来举例，我们会在 podfile 文件中增加一个自定义的 ruby 脚本 podfilehelper 的调用，podfilehelper 会声明 flutter framework 的依赖，声明 flutter plugin native 的源码引用，同时声明业务代码的路径。接下来会介入构建流程，在 xcode 的 build phase 内加入 shell 脚本 xcode_backend 的调用，xcode_backend 会根据当前构建模式，来产出 dart 构建产物。

flutter-boot 的补充

对于官方的混合工程项目，我们在体验后发现有如下的问题：

1. 文件或配置的添加为手动添加，流程较长
2. 不支持在 flutter 仓库下运行原生工程
3. 不支持 flutter 以独立代码仓库部署时的远端机器构建

因此在 flutter-boot 脚手架中，为了解决这些问题，我们把混合工程的部署分为 create, link, remotelink, update 四个过程。

create

create 过程目的在于帮助我们搭建一个 flutter module，包括 flutter module 的创建和 git 仓库的部署，flutter module 创建命令调用前，我们会做基础的检查来让工程位置和命名的规范满足官方的条件。在 git 仓库部署时，我们会在 .gitignore 中忽略部分文件，同时我们会对仓库的状态进行检查，在仓库为空时，直接添加文件，在仓库非空时，会优先清理仓库。

link

link 过程目的在于关联本地的原生工程和 flutter 工程。关联的过程中，我们会先请求获取 flutter 工程的地址和原生工程的地址，然后我们将上面提到的需要手动集成的部分通过脚本的方式自动集成；为了获得 flutter 开发视角（即 flutter 工程下运行原生工程），我们将原生工程进行了软链接，链接到 flutter 工程的 ios 目录和 android 目录，flutter 在运行前会找到工程下的 ios 或 android 目录然后运行，在 flutter 工程下运行 iOS 工程会存在一个限制，即 iOS 工程的 target 需要指定为 runner，为了解决这个问题，我们将原生工程的主 target 进行了复制，复制了一份名为 runner 的 target。

同时，为了支持远程构建的模式，我们 flutter 仓库本地路径的声明根据构建模式进行了区分，封装在自定义的依赖脚本中，例如在 iOS 工程内，我们会添加 fbpodhelper.rb 脚本文件。然后将 flutter 仓库本地路径添加到了配置文件 fbConfig.local.json 中。

remotelink && update

remotelink 过程目的在于远端构建模式下，能够获取 flutter 仓库的代码，并在远端机器上进行构建。在远端构建模式下，我们会侵入依赖管理的过程，在依赖获取时，拉取 flutter 仓库的代码，将代码放置在原生工程的 .fbflutter 目录下，并将该目录声明为 flutter 仓库本地路径，拉取 flutter 代码并进行本地部署的过程，我们称之为 update 过程。这样在远端构建时就能和本地构建如出一辙。

那远端模式和本地模式如何区分呢？为了区分远端模式与本地模式，我们将远端的 flutter 仓库信息记录在 fbConfig.json，同时在 gitignore 中忽略 fbConfig.local.json 文件，这样只需要初始化混合工程的工程师运行一次 remotelink，其他的开发协同者将不用关注远端构建的配置流程。

init

为了方便快速搭建，我们提供了一个命令集合，命名为 init，我们将必备的环节以命令行交互的模式集成在了 init 命令中。

混合栈

混合栈是闲鱼开源的一套用于 flutter 混合工程下协调原生页面与 flutter 页面交互的框架，目前是混合开发模式下的主流框架。在混合栈开源后，我们关注到大量开发者在集成混合栈时会产生各种环境配置或代码添加导致的集成问题。因此我们决定提供一套快速集成的方案。要做到快速集成我们面临两个问题：

1. flutter 和混合栈的版本兼容
2. 混合栈 demo 代码封装及插入

版本兼容问题

目前混合栈发布版本为 0.1.52，支持 flutter 1.5.4。当 flutter 升级时混合栈势必要进行适配，即我们集成的混合栈版本也需要变更。因此我们将混合栈的版本配置通过文件进行维护，记录当前 flutter 所需要的混合栈版本。在初版的 flutter-boot 中，我们限定了混合栈的版本号，在新版本混合栈发布时，我们将开放版本选择的功能。

代码封装及插入问题

在调研了混合栈的使用过程后，我们将混合栈需要的 demo 代码分成了四个部分：

1. flutter 引擎的托管
2. 页面路由的配置
3. demo 形式的 dart 页面
4. 原生的测试跳转入口

flutter 引擎的托管

引擎的托管我们依赖于应用的初始化，由于初始化过程随着应用的复杂程度提升而提升，因此目前我们提供了一行代码作为接口，使用者在应用初始化时加入这一行代码即可完成托管。

页面路由的配置 && demo 形式的 dart 页面

路由配置即路由到某个标识符时，flutter 或原生页面需要识别并跳转相应页面。路由的配置需要在原生和 flutter 两侧进行部署。在原生侧，我们将混合栈的 demo 路由代码进行了精简，然后添加在了原生工程的固定目录下。由于 iOS 仅添加代码文件是不会被纳入构建范围的，因此我们封装了一套 iOS 侧的代码添加工具来实现文件的插入。在 flutter 侧我们对 main.dart 文件进行了覆盖，将带有路由逻辑的 main.dart 集成进来，同时提供了 demo dart 页面的创建逻辑。

原生的测试跳转入口

为了方便使用者快速看到混合工程的跳转模式，我们在 iOS 和 android 双端封装了一个入口按钮和按钮的添加过程，使用者在测试的页面手动加入一行代码，即可看到跳转 flutter 的入口。

效果

在使用 flutter-boot 前，开发者可能要花费数天来进行混合工程搭建，现在，使用者只需要调用一个命令，加入两行代码即可完成混合工程的搭建，大大降低了开发

者的开发成本。

flutter-boot 的使命还未达成，我们期望使用者能更加流畅的进行 flutter 开发，未来我们会优化多人协同的开发流程，完善持续集成环境的搭建，让使用者拥有更佳的开发体验。如果在使用过程中有任何欢迎在 [github](#) 上进行交流。

第二章 闲鱼：Flutter 企业级应用实践

Flutter & FaaS 云端一体化架构

作者：闲鱼技术 - 国有

讲师介绍

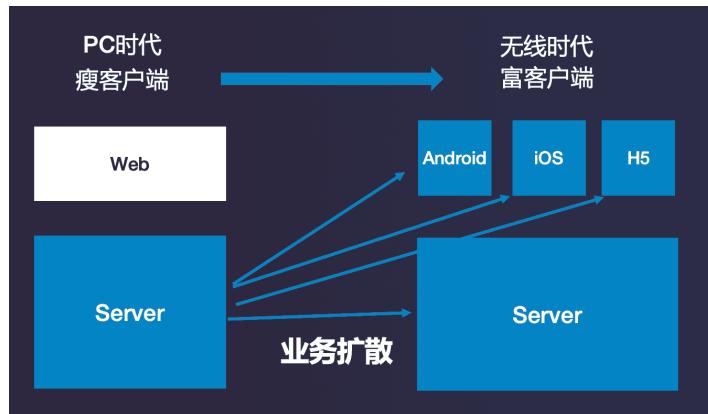


国有，闲鱼架构团队负责人。在 7 月 13 号落幕的 2019 年 Archsummit 峰会上就近一年来闲鱼在 Flutter&FaaS 一体化项目上的探索和实践进行了分享。

传统 Native+Web+ 服务端混合开发的挑战

随着无线，IoT 的发展，5G 的到来，移动研发越发向多端化发展。传统的基于 Native + Web + 服务端的开发方式，研发效率低下，显然已经无法适应发展需要。

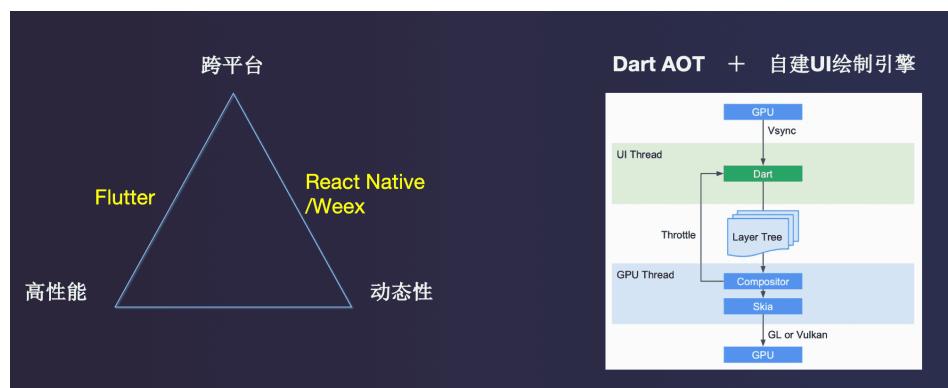
我们希望探索闲鱼这样规模的独立 APP 的高效研发架构。主要思路是围绕 Flutter 解决多端问题，并使 Flutter 与 FaaS 等无服务容能力打通，形成云端一体化的研发能力，支持一云多端的发展需要。在某些场景已经取得效果，希望分享过程中的思考，与大家交流。



跨端方案 Flutter 与 RN 的对比和选择

闲鱼选择 Flutter 主要是出于高性能的考虑。Flutter 高性能主要来源于 2 个原因：

1. Dart 的 AOT 编译能力。
2. 自建渲染引擎，不需要转换到 Native 控件，避免了线程跳跃等问题。



更多比较：

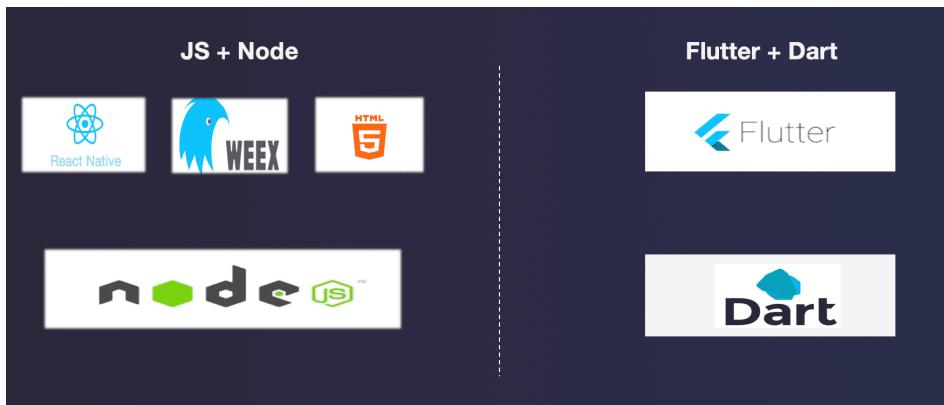
	Flutter	RN / Weex
性能	和Native一致	接近原生，但在长列表、富动画场景弱
一致性	强	弱
动态性	无	强
标准	自建	W3C子集
技术栈	Dart	Javascript & Native
跨平台	Mobile&Web&Desktop&Embedded	Mobile&Weex-Web
社区	高速成长	成熟

没有银弹的解决方案，Flutter 与 RN 各有优点。如何选择因素很多，关键看如何取舍，举个例子：

- 当前团队人员以前端 JS 栈为主还是 Native 为主？如果 JS 为主，写 RN 会更习惯。如果 Android 或 iOS 为主，写 Flutter 会更习惯，因为 Flutter 的研发工具和体验与 Native 更相似。
- 动态性和复杂交互的性能，哪个更重要？动态性重要 RN 合适，性能体验重要 Flutter 不会失望。虽然 Flutter 也有一些动态化解决方案，例如 JS 转接 Flutter 引擎的方案，Dart 代码 CodePush 的方案，组件化服务端组装方案等，但这些动态方案都没有 RN 这样从 JS 层解决的这么好。
- 是否需要 IoT 等多端布局？Flutter 在嵌入式设计上有布局，性能有更好的表现。

Dart 作为 FaaS 层的第一可选语言

云端技术栈的打通，是减少协同的不错的解法。以往前端 + Node.js 的一体化方案大家应该不会陌生，然而如果端侧使用了 Flutter，那云侧 Dart 自然是第一选择。



FaaS 的本质是运行在云端，那 Dart 适合用在云 /Server 上吗？

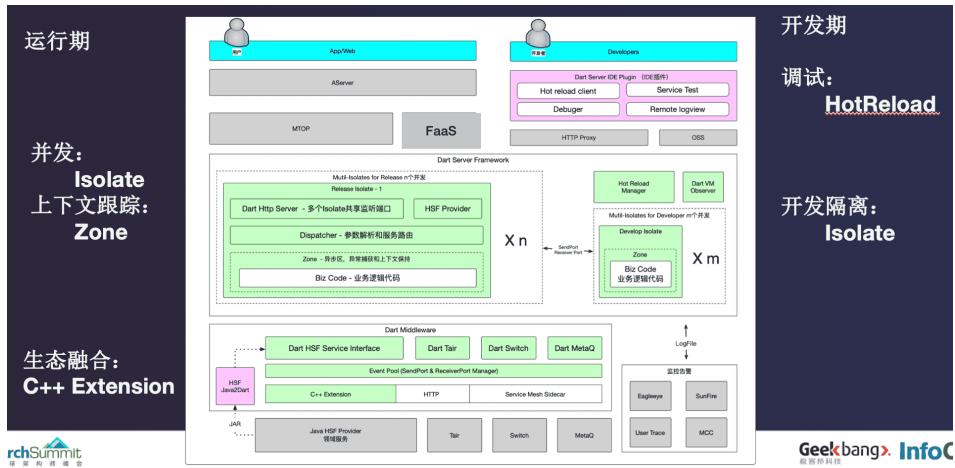
Dart 语言早于 Flutter，在最初的设计上，Dart 就可以用于 Web、Server。

Dart 具备一些服务端语言的特点：

- 强类型，可预测性
- GC
- 异步和并发
- 高性能的 JIT
- Profiler

闲鱼首先尝试将 Dart 作为普通的 Server，替代传统的 Java Server，然后再将 Dart 容器嵌入到 FaaS 容器中。建立 Dart Server 能力是第一步，也是主要的工作量所在。

闲鱼在 Dart Server 方面的建设思路：



开发期:

- 受 Flutter 的 HotReload 启发，将 HotReload 移植到了 Server 侧。
- 利用 Isolate，在开发环境中为每个开发人员分配一个 Isolate，解决以往的环境冲突的问题。

运行期:

- Dart 本身是单线程异步模型，并发能力需要用 Isolate 支持。
- 利用 Dart 的 Zone 的特性，可以方便的实现调用链路的跟踪，方便记录 Trace 日志。
- 利用 Dart 支持的 C++ Extension 能力，可以在 Dart 中访问支持了 C++ 的中间件包。另外，Server Mesh 也是一个重要的思路，用于解耦异构语言之间的服务调用。

一体化的更深层思考

上述内容实现了 Flutter&Dart FaaS 的技术栈的统一，但仅技术栈统一还远远不够，端、云的同学仍然无法真正互补和一体化打通，原因在于还有更多深入问题需要考虑：

- 一体化的业务闭环红利如何最大化？一体化不仅是效率的提升，还使一个同学

可以 Cover 一个云到端的业务，使业务闭环。

- 如何消除云端技术壁垒？仅技术栈打通，端人员还是会写云，原因在于对云的思维模式的不理解，需要真正消除云端的技术壁垒。
- 如何使工作总量减少（ $1+1<2$ ）？如果一体化后把工作量压到一个人身上，那意义不大，需要使一体化下的总工作量降低。
- 如何促进生产关系重塑？生产关系需要适应新的生产力。



面向这些问题，闲鱼的解法思路：

- 业务闭环为业务开发同学带来更好的成长空间，可以完整和专注的思考业务。这是人上的核心动力。
- 业务闭环是业务流程沉淀的方向
- 以往的架构是云、端分开架构的，一体化后有了更多的架构下沉空间，从而带来了总工作量 $1 + 1 < 2$ 的可能
- 领域下沉和工具支撑是一体化的保证

案例效果

案例一，一体化在资源均衡方面的体现。在近期的一个项目中，云端一体化使原本 2 个月的项目时间，减少了 20 天。



ArchSummit

案例二，一体化在业务闭环方面的体现。负责增长的一位开发同学，专注在增长业务上，在合适的情况下为合适的人投放合适的内容，以此带来用户的增长和活跃效果。一体化的方式下，可以统一云、端的切面，业务研发不再受云、端的限制。



说在最后

一体化是建设高效研发框架的方向，并不是所有场景都需要一体化的开发，但一体化的 Flutter、FaaS 等技术组件，可以独立使用，也会带来效率提升，并且与原有的开发模式兼容。从一体化的思路去建设，可以使整体架构体系更加一致，也有机会做一体的架构沉淀。未来闲鱼希望在一体化上做更多尝试和深入探索，包括一体化工具、一体化业务平台、数据化智能化等方向。

基于 Flutter 的架构演进与创新

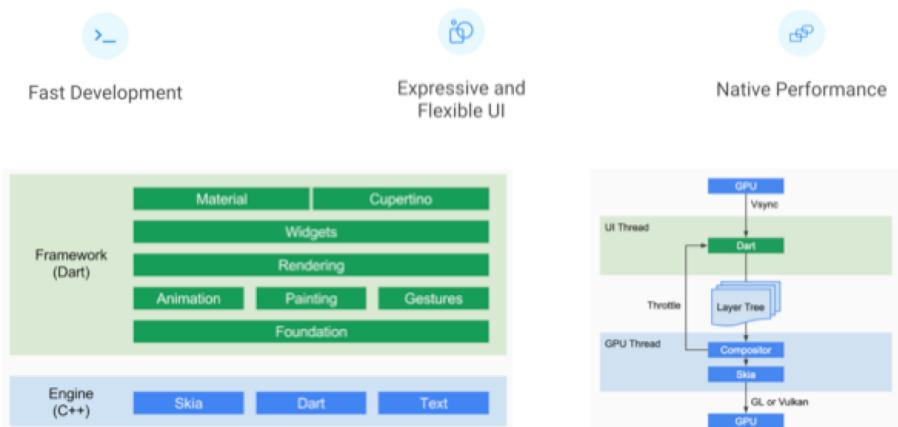
作者：闲鱼技术 - 宗心

讲师介绍



2012 年应届毕业生加入阿里巴巴，主导了闲鱼基于 Flutter 的新混合架构，同时推进了 Flutter 在闲鱼各业务线的落地。未来将持续关注终端技术的演变及趋势。

Flutter 的优势与挑战



Flutter 是 Google 开源的跨端便携 UI 工具包，除了具有非常优秀的跨端渲染一致性，还具备非常高效的研发体验，丰富的开箱即用的 UI 组件，以及跟 Native 媲美的性能体验。由于它的众多优势，也使得 Flutter 成为了近年来热门的新技术。

通过以上的特点可以看出，Flutter 可以极大的加速客户端的研发效率，与此同时得到优秀的性能体验，基于我的思考，Flutter 会为以下团队带来较大的收益：

- 中小型的客户端团队非常适合 Flutter 开发，不仅一端编写双端产出，还有效的解决了小团队需要双端人员（iOS: Android）占比接近 1:1 的限制，在项目快速推进过程中，能让整个团队的产能最大化。
- App 在 Android 市场占比远高于 iOS 的团队，比如出海东南亚的一些 App，Android 市场整体占比在 90% 以上，通过 Flutter 可以将更多的人力 Focus 在 Android 市场上，同时通过在 iOS 端较小的投入，在结果上达到买一送一的效果。
- 以量产 App 为主要策略的团队，不论是量产 ToB 的企业 App，还是有针对性的产出不同领域的 ToC 的 App 的公司，都可以通过一端开发多端产出的 Flutter 得到巨大的产能提升。

三亿人都在用的闲置交易社区



闲鱼在以上的场景中属于第一种场景，服务 3 亿用户的闲鱼 App 的背后，是十几名客户端开发，与竞对相比，我们是一只再小不过的团队，在这种场景下，Flutter 为闲鱼业务的稳定发展以及提供更多的创新产品给予了很大的帮助。

但与此同时，Flutter 在设计上带来的优势同时又会带来新的问题。所有的新技术

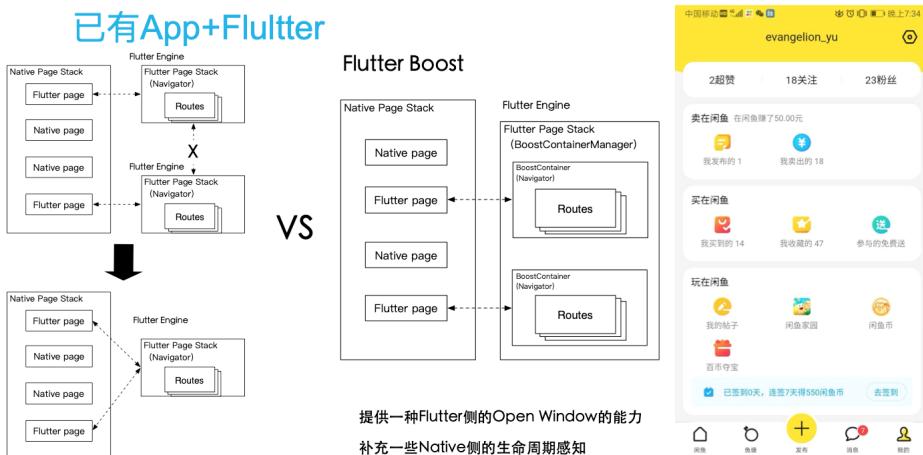
术都是脱胎于老技术的，Flutter 也不例外，其身上带有很多 Chrome 的影子。我们再做一层简化，如果我们认为 Flutter 是一个使用 Dart 语言的浏览器容器，请大家思考一下两个问题如何解决。

- 如果在一个已经存在的 App 中加入 Flutter，如何让 Native 与 Flutter 进行无缝的衔接，同时保证相互开发之间的隔离性
- 如果在 Flutter 的容器中，使用已有的 Native UI 组件，在 Flutter 与 Native 渲染机制不同的情况下，怎么保证两者的无缝衔接以及高性能。

闲鱼的架构演进与创新

带着上面两个问题，我们来到闲鱼场景下的具体 Case 以及解决方案的演进过程。

已有 App+Flutter 容器



在这种情况下，闲鱼需要考虑的是首先要考虑引入 Flutter 容器后的内存压力，保证不要产生更多的内存溢出。与此同时我们希望能让 Flutter 和 Native 之间的页面切换是顺畅的，对不同技术栈之间的同学透明。因此我们有针对性的进行了多次迭代。

在没有任何改造的情况下以 iOS 为例，你可以通过创建新的 FlutterViewController 来创建一个新的 Flutter 容器，这个方案下，当创建多个 FlutterViewController 时会同时在内存中创建多个 Flutter Engine 的 Runtime (虽然底层 Dart VM

依然只有一个)，这对内存消耗是相当大的，同时多个 Flutter Engine 的 Runtime 会造成每个 Runtime 内的数据无法直接共享，造成数据同步困难。

这种情况下，闲鱼选择了全局共享同一个 FlutterViewController 的方式保证了内存占用的最小化，同时通过基础框架 Flutter Boost 提供了 Native 栈与 Flutter 栈的通信与管理，保证了当 Native 打开或关闭一个新的 Flutter 页面时，Dart 侧的 Navigator 也做到自动的打开或关闭一个新的 Widget。目前 Google 官方提供的方案上就是参考闲鱼早先的这个版本进行的实现的。

然而在这种情况下，如果出现如闲鱼图中所示多个 Tab 的场景下，整个堆栈逻辑就会产生混乱，因此闲鱼在这个基础上对 Flutter Boost 的方案进行了升级并开源，通过在 Dart 侧提供一个 BoostContainerManager 的方式，提供了对多个 Navigator 的管理能力，如果打比方来看这件事，就相当于，针对 Flutter 的容器提供了一个类似 WebView 的 OpenWindow 的能力，每做一次 OpenWindow 的调用，就会产生一个新的 Navigator，这样开发者就可以自由的选择是在 Navigator 里进行 Push 和 Pop，还是直接通过 Flutter Boost 新开一个 Navigator 进行独立管理。

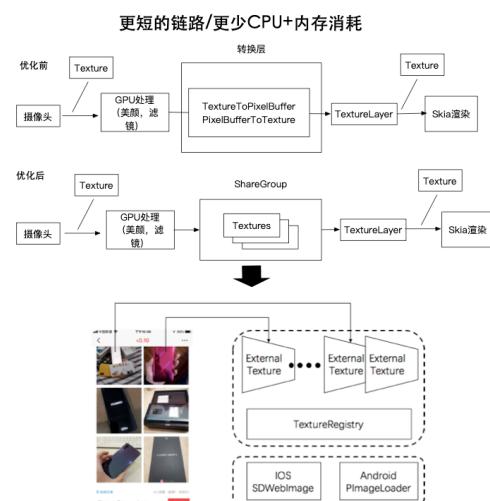
Flutter Boost 目前已在 github 开源，由于闲鱼目前线上版本只支持 Flutter 1.2 的版本，因此需要支持 1.5 的同学等稍等，我们会在近期更新支持 1.5 的 Flutter Boost 版本。

Flutter 页面 +Native UI

Flutter + 已有Native能力



VS



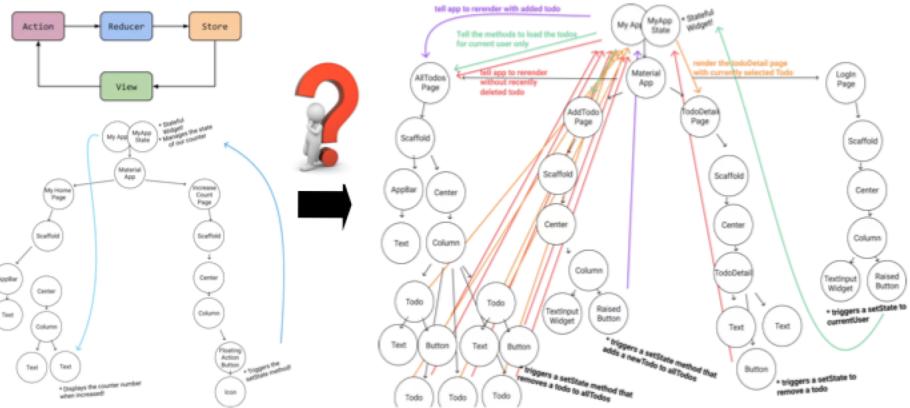
由于闲鱼是一个闲置交易社区，因此图片和视频相对较多，对图片视频的线上性能以及内存占用有较严格的要求。目前 Flutter 已提供的几种方案中 (Platform View 以及 Flutter Plugin)，不论是对内存的占用还是整个的线上流畅度上还存在一定的问题，这就造成了当大部分同学跟闲鱼一样实现一个复杂的图文 Feed 推荐场景的时候，非常容易产生内存溢出。而实际上，闲鱼在以上的场景下有针对性的做出了较大的优化。

在整个的 Native UI 到 Flutter 渲染引擎桥接的过程中，我们选用了 Flutter Plugin 中提供的 FlutterTextureRegistry 的能力，在去年上半年我们优先针对视频的场景进行了优化，优化的思路主要是针对 Flutter Engine 底层的外接纹理接口进行修改，将原有接口中必须传入一个 PixelBuffer 的内存对象这一限制做了扩展，增加一个新的接口保证其可以传入一个 GPU 对象的 TextureID。

如图中所示，优化后的整个链路 Flutter Engine 可以直接通过 Native 端已经生成好的 TextureID 进行 Flutter 侧的渲染，这样就将链路从 Native 侧生成的 TextureID->copy 的内存对象 PixelBuffer-> 生成新的 TextureID-> 渲染，转变为 Native 侧生成的 TextureID-> 渲染。整个链路极大的缩短，保证了整个的渲染效率以及更小的内存消耗。闲鱼在将这套方案上线后，又尝试将该方案应用于图片渲染的场景下，使得图片的缓存，CDN 优化，图片裁切等方案与 Native 归一，在享受已有集团中间件的性能优化的同时，也得到了更小的内存消耗，方案落地后，内存溢出大幅减少。

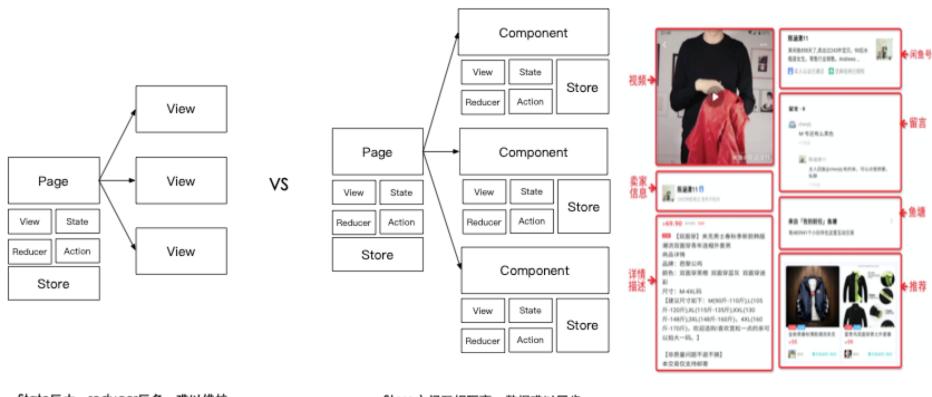
目前该方案由于需要配合 Flutter Engine 的修改，因此暂时无法提供完整的方案至开源社区，我们正在跟 google 积极沟通整个修改方案，相信在这一两个月内会将试验性的 Engine Patch 开源至社区，供有兴趣的同学参考。

复杂业务场景的架构创新实践



将以上两个问题解决以后，闲鱼开始了 Flutter 在业务侧的全面落地，然而很快又遇到新的问题，在多人协作过程中：

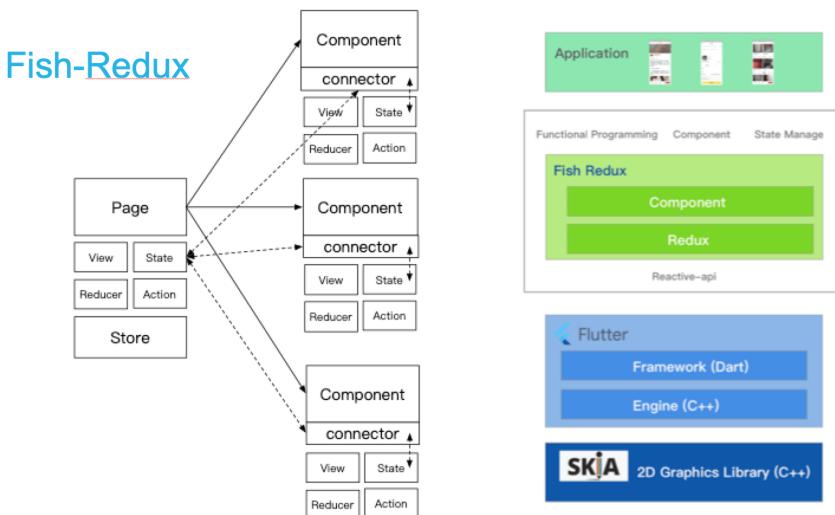
- 如何提供一些标准供大家进行参考保证代码的一致性
- 如何将复杂业务进行有效的拆解变成子问题
- 如何保证更多的同学可以快速上手并写出性能和稳定性都不错的代码



在方案的前期，我们使用了社区的 Flutter Redux 方案，由于最先落地的详情，发布等页面较为复杂，因此我们有针对性的对 View 进行了组件化的拆分，但由于业务的复杂性，很快这套方案就出现了问题，对于单个页面来说，State 的属性以及

Reducer 的数量都非常多，当产生新需求堆叠的时候，修改困难，容易产生线上问题。

针对以上的情况，我们进行了整个方案的第二个迭代，在原有 Page 的基础上提供了 Component 的概念，使得每个 Component 具备完整的 Redux 元素，保证了 UI，逻辑，数据的完整隔离，每个 Component 单元下代码相对较少，易于维护和开发，但随之而来的问题是，当页面需要产生数据同步时，整个的复杂性飙升，在 Page 的维度上失去了统一状态管理的优势。

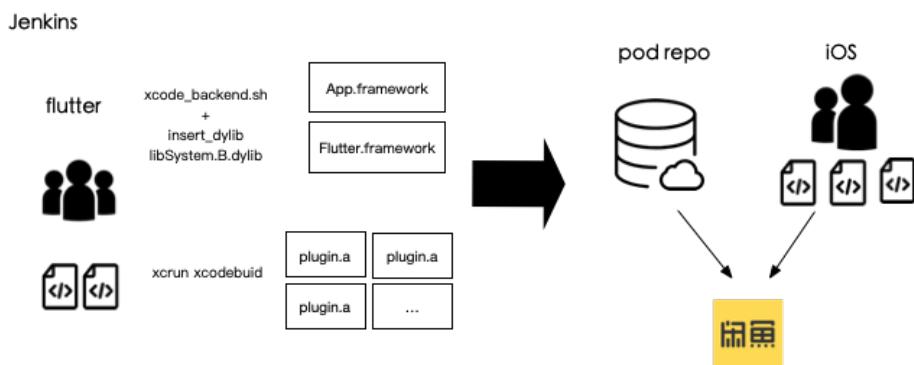


在这种情况下闲鱼换个角度看端侧的架构设计，我们参考 React Redux 框架中的 Connect 的思想，移除掉在 Component 的 Store，随之而来的是新的 Connector 作为 Page 和 Component 的数据联通的桥梁，我们基于此实现了 Page State 到 Component State 的转换，以及 Component State 变化后对 Page State 的自动同步，从而保证了将复杂业务有效的拆解成子问题，同时享受到统一状态管理的优势。与此同时基于新的框架，在统一了大家的开发标准的情况下，新框架也在底层有针对性的提供了对长列表，多列表拼接等 case 下的一些性能优化，保证了每一位同学在按照标准开发后，可以得到相对目前市面上其他的 Flutter 业务框架相比更好的性能。

目前这套方案 Fish Redux 已经在 github 开源，目前支持 1.5 版本，感兴趣的同学可以去 github 进行了解。

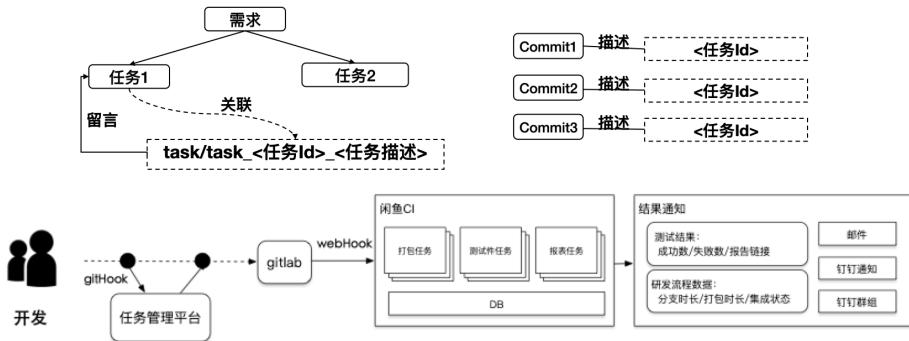
研发智能化在闲鱼的应用

闲鱼在去年经历了业务的快速成长，在这个阶段上，我们同时进行了大量的 Flutter 的技术改造和升级，在尝试新技术的同时，如何能保证线上的稳定，线下的有更多的时间进行新技术的尝试和落地，我们需要一些新的思路和工作方式上的改变。



以我们日常工作为例，Flutter 的研发同学，在每次开发完成后，需要在本地进行 Flutter 产物的编译并上传到远端 Repo，以便对 Native 同学透明，保证日常的研发不受 Flutter 改造的干扰。在这个过程中，Flutter 侧的业务开发同学面临着很多打包上传更新同步等繁琐的工作，一不小心就会出错，后续的排查等让 Flutter 前期的开发变成了开发 5 分钟，打包测试 2 小时。同时 Flutter 到底有没有解决研发效率快的问题，以及同学们在落地过程中有没有 Follow 业务架构的标准，这一切都是未知的。

数据报表+持续集成



在痛定思痛以后，我们认为数据化 + 自动化是解决这些问题的一个较好的思路。因此我们首先从源头对代码进行管控，通过 commit，将代码与后台的需求以及 bug 一一关联，对于不符合要求的 commit 信息，不允许进行代码合并，从而保证了后续数据报表分析的数据源头是健康的。

在完成代码和任务关联后，通过 webhook 就可以比较轻松的完成后续的工作，将每次的 commit 有效的关联到我们的持续集成平台的任务上来，通过闲鱼 CI 工作平台将日常打包自动化测试等流程变为自动化的行为，从而极大的减少了日常的工作。粗略统计下来，在去年自动化体系落地的过程中单就自动打 Flutter 包上传以及触发最终的 App 打包这一流程就让每位同学每天节省一个小时以上的工作量，效果非常明显。另外，基于代码关联需求的这套体系，可以相对容易的构建后续的数据报表对整个过程和结果进行细化的分析，用数据驱动过程改进，保证新技术的落地过程的收益有理有据。

总结与展望

回顾一下上下文

- Flutter 的特性非常适合中小型客户端团队 /Android 市场占比较高的团队 / 量产 App 的团队。同时由于 Flutter 的特性导致其在混合开发的场景下面存在一定劣势。

- 闲鱼团队针对混合开发上的几个典型问题提供了对应的解决方案，使整个方案达到上线要求，该修改会在后续开放给 google 及社区。
- 为全面推动 Flutter 在业务场景下的落地，闲鱼团队通过多次迭代演进出 Fish Redux 框架，保证了每位同学可以快速写出相对优秀的 Flutter 代码。
- 新技术的落地过程中，在过程中通过数据化和自动化的方案极大的提升了过程中的效率，为 Flutter 在闲鱼的落地打下了坚实的基础。

除了本文提及的各种方案外，闲鱼目前还在多个方向上发力，并对针对 Flutter 生态的未来进行持续的关注，分享几个现在在做的事情

- Flutter 整个上层基础设施的标准化演进，混合工程体系是否可以在上层完成类似 Spring-boot 的完整体系构架，帮助更多的 Flutter 团队解决上手难，无行业标准的问题。
- 动态性能力的扩展，在符合各应用商店标准的情况下，助力业务链路的运营效率提升，保证业务效果。目前闲鱼已有的动态化方案会后续作为 Fish-Redux 的扩展能力提供动态化组件能力 + 工具链体系。
- Fish-Redux + UI2Code，打通代码生成链路和业务框架，保证在团队标准统一的情况下，将 UI 工作交由机器生成。
- Flutter + FaaS，让客户端同学可以成为全栈工程师，通过前后端一体的架构设计，极大的减少协同，提升效率。

让工程师去从事更多创造性的工作，是我们一直努力的目标。闲鱼团队也会在新的一年更多的完善 Flutter 体系的建设，将更多已有的沉淀回馈给社区，帮助 Flutter 社区一起健康成长。

第三章 混合开发实践指南

Flutter Plugin 调用 Native APIs

作者：闲鱼技术 – 储睿



关键词：Flutter, Flutter Plugin, Platform Channel, Method Channel, Flutter Package, Flutter 插件

Flutter 是 Google 使用 Dart 语言开发的一套移动应用开发框架。它不同于其他开发框架：

- (1) 因为 Flutter 使用 AOT 预编译代码为机器码，所以它的运行效率更高。
- (2) Flutter 的 UI 控件并没有使用底层的原生控件，而是使用 Skia 渲染引擎绘制而成，因为不依赖底层控件，所以多端一致性非常好。
- (3) Flutter 的扩展性也非常强，开发者可以通过 Plugin 与 Native 进行通信。

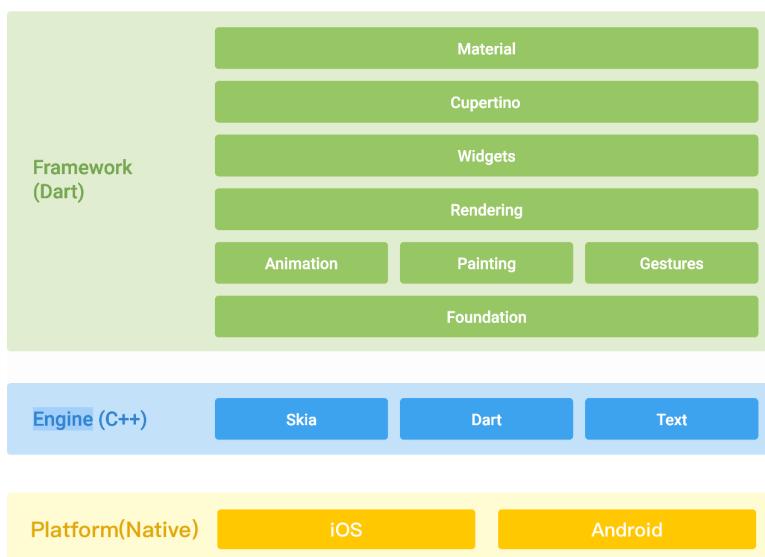
闲鱼开发 Flutter 过程中，经常会需要各种 Native 的能力，如获取设备信息、使用基础网络库等，这时会使用 Plugin 来做桥接。本文将对 Plugin 进行详细的介绍，希望能给 Flutter 开发者一些帮助。

摘要：

本文首先对 Flutter Plugin 以及原理进行了介绍，然后对 Plugin 所依赖的 Platform Channel 进行了讲解，随后对“获取剩余电量 Plugin”进行了分解，最后给大家分享一下之前踩过的坑。

1. Flutter Plugin

在介绍 Plugin 前，我们先简单了解一下 Flutter：



Flutter 框架包括：Framework 和 Engine，他们运行在各自的 Platform 上。

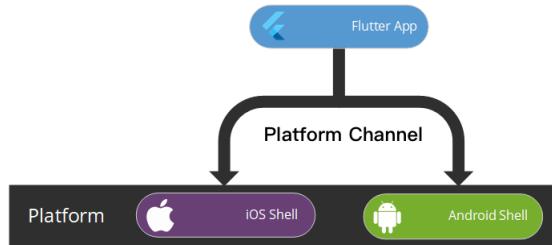
Framework 是 Dart 语言开发的，包括 Material Design 风格的 Widgets 和 Cupertino(iOS-style) 风格的 Widgets，以及文本、图片、按钮等基础 Widgets；还包括渲染、动画、绘制、手势等基础能力。

Engine 是 C++ 实现的，包括 Skia (二维图形库); Dart VM (Dart Runtime); Text (文本渲染) 等。

实际上，Flutter 的上层能力都是 Engine 提供的。Flutter 正是通过 Engine 将各个 Platform 的差异化抹平。而我们今天要讲的 Plugin，正是通过 Engine 提供的 Platform Channel 实现的通信。

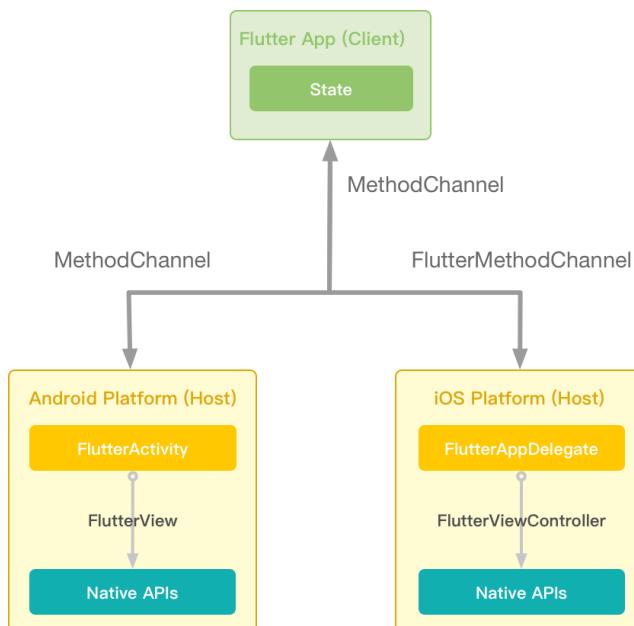
2. Platform Channel

2.1 Flutter App 调用 Native APIs:



通过上图，我们看到 Flutter App 是通过 Plugin 创建的 Platform Channel 调用的 Native APIs。

2.2 Platform Channel 架构图:



Platform Channel:

- Flutter App (Client)，通过 MethodChannel 类向 Platform 发送调用消息；

- Android Platform (Host), 通过 MethodChannel 类接收调用消息;
- iOS Platform (Host), 通过 FlutterMethodChannel 类接收调用消息。

PS: 消息编解码器, 是 JSON 格式的二进制序列化, 所以调用方法的参数类型必须是可 JSON 序列化的。

PS: 方法调用, 也可以反向发送调用消息。

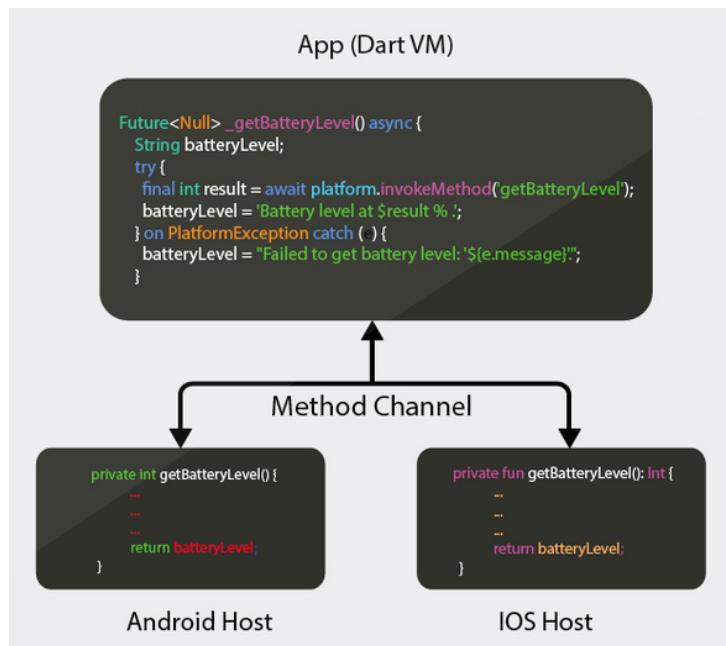
Android Platform

FlutterActivity, 是 Android 的 Plugin 管理器, 它记录了所有的 Plugin, 并将 Plugin 绑定到 FlutterView。

iOS Platform

FlutterAppDelegate, 是 iOS 的 Plugin 管理器, 它记录了所有的 Plugin, 并将 Plugin 绑定到 FlutterViewController (默认是 rootViewController)。

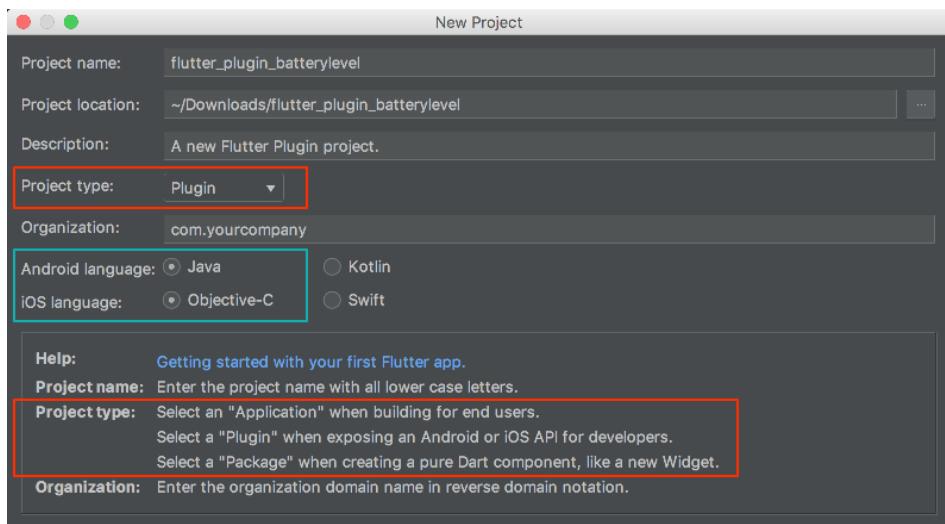
3. 获取剩余电量 Plugin



3.1 创建 Plugin

首先，我们创建一个 Plugin/flutter_plugin_batterylevel 项目。Plugin 也是项目，只是 Project type 不同。

- (1) IntelliJ 欢迎界面点击 Create New Project 或者 点击 File>New>Project…；
- (2) 在左侧菜单选择 Flutter, 然后点击 Next;
- (3) 输入 Project name 和 Project location, **Project type** 选择“Plugin”；
- (4) 最后点击 Finish。



Project type:

- (1) Application, Flutter 应用；
- (2) Plugin, 暴漏 Android 和 iOS 的 API 给 Flutter 应用；
- (3) Package, 封装一个 Dart 组件，如“浏览大图 Widget”。

PS: Plugin 有 Dart、Android、iOS, 3 部分代码组成。

3.2 Plugin Flutter 部分

3.2.1 MethodChannel: Flutter App 调用 Native APIs

```
/**
 * (1) MethodChannel: Flutter App 调用 Native APIs
 */
static const MethodChannel _methodChannel = const MethodChannel('samples.flutter.io/battery');

// Future<String> getBatteryLevel() async {
String batteryLevel;
try {
    final int result = await _methodChannel.invokeMethod('getBatteryLevel',
{'paramName':'paramVale'});
    batteryLevel = 'Battery level: $result%.';
} catch(e) {
    batteryLevel = 'Failed to get battery level.';
}
return batteryLevel;
}
```

首先，我们实例 `_methodChannel` (**Channel 名称必须唯一**)，然后调用 `invokeMethod()` 方法。`invokeMethod()` 有 2 个参数：

- (1) 方法名，不能为空；
- (2) 调用方法的参数，该参数必须可 JSON 序列化，可以为空。

3.2.2 EventChannel: Native 调用 Flutter App

```
/**
 * (2) EventChannel: Native 调用 Flutter App
 */
static const EventChannel _eventChannel = const EventChannel('samples.flutter.io/charging');

void listenNativeEvent() {
    _eventChannel.receiveBroadcastStream().listen(_onEvent, onError:_onError);
}

void _onEvent(Object event) {
    print("Battery status: ${event == 'charging' ? '' : 'dis'}charging.");
}
```

```

    }

    void _onError(Object error) {
        print('Battery status: unknown.');
    }
}

```

3.3 Plugin Android 部分

3.3.1 Plugin 注册

```

import android.os.Bundle;
import io.flutter.app.FlutterActivity;
import io.flutter.plugins.GeneratedPluginRegistrant;

public class MainActivity extends FlutterActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        GeneratedPluginRegistrant.registerWith(this);
    }
}

```

在 `FlutterActivity` 的 `onCreate()` 方法中，注册 `Plugin`。

```

/**
 * Plugin 注册 .
 */
public static void registerWith(Registrar registrar) {
    /**
     * Channel 名称：必须与 Flutter App 的 Channel 名称一致
     */
    private static final String METHOD_CHANNEL = "samples.flutter.io/battery";
    private static final String EVENT_CHANNEL = "samples.flutter.io/charging";

    // 实例 Plugin，并绑定到 Channel 上
    FlutterPluginBatteryLevel plugin = new FlutterPluginBatteryLevel();

    final MethodChannel methodChannel = new MethodChannel(registrar.
messenger(), METHOD_CHANNEL);
    methodChannel.setMethodCallHandler(plugin);

    final EventChannel eventChannel = new EventChannel(registrar.
messenger(), EVENT_CHANNEL);
    eventChannel.setStreamHandler(plugin);
}

```

- (1) Channel 名称：必须与 Flutter App 的 Channel 名称一致；
- (2) MethodChannel 和 EventChannel 初始化的时候都需要传递 Registrar，即 FlutterActivity；
- (3) 设置 MethodChannel 的 Handler，即 MethodCallHandler；
- (4) 设置 EventChannel 的 Handler，即 EventChannel.StreamHandler；

3.3.2 MethodCallHandler & EventChannel.StreamHandler

MethodCallHandler 实现 MethodChannel 的 Flutter App 调用 Native APIs；
 EventChannel.StreamHandler 实现 EventChannel 的 Native 调用 Flutter App。

```
public class FlutterPluginBatteryLevel implements MethodCallHandler,
EventChannel.StreamHandler {

    /**
     * MethodCallHandler
     */
    @Override
    public void onMethodCall(MethodCall call, Result result) {
        if (call.method.equals("getBatteryLevel")) {
            Random random = new Random();
            result.success(random.nextInt(100));
        } else {
            result.notImplemented();
        }
    }

    /**
     * EventChannel.StreamHandler
     */
    @Override
    public void onListen(Object obj, EventChannel.EventSink eventSink) {
        BroadcastReceivier chargingStateChangeReceiver =
        createChargingStateChangeReceiver(events);
    }

    @Override
    public void onCancel(Object obj) {
    }

    private BroadcastReceivier createChargingStateChangeReceiver(final
EventSink events) {
        return new BroadcastReceivier() {
            @Override
            public void onReceive(BroadcastReceivier.Receiver receiver) {
                if (events != null) {
                    events.success("chargingStateChangeReceiver");
                }
            }
        };
    }
}
```

```

        public void onReceive(Context context, Intent intent) {
            int status = intent.getIntExtra(BatteryManager.EXTRA_STATUS, -1);

            if (status == BatteryManager.BATTERY_STATUS_UNKNOWN) {
                events.error("UNAVAILABLE", "Charging status unavailable", null);
            } else {
                boolean isCharging = status == BatteryManager.BATTERY_
STATUS_CHARGING ||

                    status == BatteryManager.BATTERY_STATUS_FULL;
                events.success(isCharging ? "charging" : "discharging");
            }
        };
    }
}

```

MethodCallHandler:

(1) public void onMethodCall(MethodCall call, Result result);

EventChannel.StreamHandler:

(1) public void onListen(Object obj, EventChannel.EventSink eventSink);

(2) public void onCancel(Object obj);

3.4 Plugin iOS 部分

3.4.1 Plugin 注册

```

/**
 * Channel 名称: 必须与 Flutter App 的 Channel 名称一致
 */
#define METHOD_CHANNEL "samples.flutter.io/battery";
#define EVENT_CHANNEL "samples.flutter.io/charging";

@implementation AppDelegate

- (BOOL)application:(UIApplication*)application didFinishLaunchingWithOptions:
(NSDictionary*)launchOptions {
    /**
     * 注册 Plugin
     */
    [GeneratedPluginRegistrant registerWithRegistry:self];

    /**
     * FlutterViewController
     */
}

```

```

/*
FlutterViewController* controller = (FlutterViewController*)self.window.
rootViewController;

/**
 * FlutterMethodChannel & Handler
 */
FlutterMethodChannel* batteryChannel = [FlutterMethodChannel
methodChannelWithName:METHOD_CHANNEL binaryMessenger:controller];
[batteryChannel setMethodCallHandler:^(FlutterMethodCall* call,
FlutterResult result) {
    if ([@"getBatteryLevel" isEqualToString:call.method]) {
        int batteryLevel = [self getBatteryLevel];
        result(@(batteryLevel));
    } else {
        result(FlutterMethodNotImplemented);
    }
}];

/**
 * FlutterEventChannel & Handler
 */
FlutterEventChannel* chargingChannel = [FlutterEventChannel
eventChannelWithName:EVENT_CHANNEL binaryMessenger:controller];
[chargingChannel setStreamHandler:self];

return [super application:application
didFinishLaunchingWithOptions:launchOptions];
}

@end

```

iOS 的 Plugin 注册流程跟 Android 一致。只是需要注册到 AppDelegate(FlutterAppDelegate)。

FlutterMethodChannel 和 FlutterEventChannel 被绑定到 FlutterViewController。

3.4.2 FlutterStreamHandler:

```

@interface AppDelegate () <FlutterStreamHandler>

@property (nonatomic, copy) FlutterEventSink eventSink;

@end

```

```

- (FlutterError*)onListenWithArguments:(id)arguments eventSink:
(FlutterEventSink)eventSink {
    self.eventSink = eventSink;

    // 监听电池状态
    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(onBatteryStateDidChange:)
                                               name:UIDeviceBatteryStateDidChangeNotification
                                              object:nil];
    return nil;
}

- (FlutterError*)onCancelWithArguments:(id)arguments {
    [[NSNotificationCenter defaultCenter] removeObserver:self];
    self.eventSink = nil;
    return nil;
}

- (void)onBatteryStateDidChange:(NSNotification*)notification {
    if (self.eventSink == nil) return;
    UIDeviceBatteryState state = [[UIDevice currentDevice] batteryState];
    switch (state) {
        case UIDeviceBatteryStateFull:
        case UIDeviceBatteryStateCharging:
            self.eventSink(@"charging");
            break;
        case UIDeviceBatteryStateUnplugged:
            self.eventSink(@"discharging");
            break;
        default:
            self.eventSink([FlutterError errorWithCode:@"UNAVAILABLE"
                                              message:@"Charging status unavailable"
                                              details:nil]);
            break;
    }
}
}

```

4. 加载 Plugin

现在我们已经有了 Plugin，但是如何把它加载到 Flutter App 项目中呢？

It's Pub. Pub 是 Dart 语言提供的 Packages 管理工具。

说到 Package，它有 2 种类型：

(1) Dart Packages：只包含 Dart 代码，如“浏览大图 Widget”。

(2) Plugin Packages：包含的 Dart 代码能够调用 Android 和 iOS 实现的 Native APIs，如“获取剩余电量 Plugin”。

4.1 将一个 Package 添加到 Flutter App 中

- (1) 通过编辑 `pubspec.yaml`（在 App 根目录下）来管理依赖；
- (2) 运行 `flutter packages get`，或者在 IntelliJ 里点击 **Packages Get**；
- (3) `import package`，重新运行 App。

管理依赖有 3 种方式：`Hosted packages`、`Git packages`、`Path packages`。

4.2 Hosted packages（来自 `pub.dartlang.org`）

如果你希望自己的 Pulgin 给更多的人使用，你可以把它发布到 `pub.dartlang.org`。

发布 Hosted packages：

```
$ flutter packages pub publish --dry-run  
$ flutter packages pub publish
```

加载 Hosted packages：

编辑 `pubspec.yaml`：

```
dependencies:  
  url_launcher: ^3.0.0
```

4.3 Git packages（远端）

如果你的代码不经常改动，或者不希望别人修改这部分代码，你可以用 Git 来管理你的代码。我们先创建一个 Plugin(`flutter_remote_package`)，并将它传到 Git 上，然后打个 tag。

```
// cd 到 flutter_remote_package  
flutter_remote_package $:git init  
flutter_remote_package $:git remote add origin git@gitlab.alibaba-inc.  
com:churui/flutter_remote_package.git
```

```
flutter_remote_package $:git add .
flutter_remote_package $:git commit
flutter_remote_package $:git commit -m"init"
flutter_remote_package $:git push -u origin master
flutter_remote_package $:git tag 0.0.1
```

加载 Git packages:

编辑 pubspec.yaml:

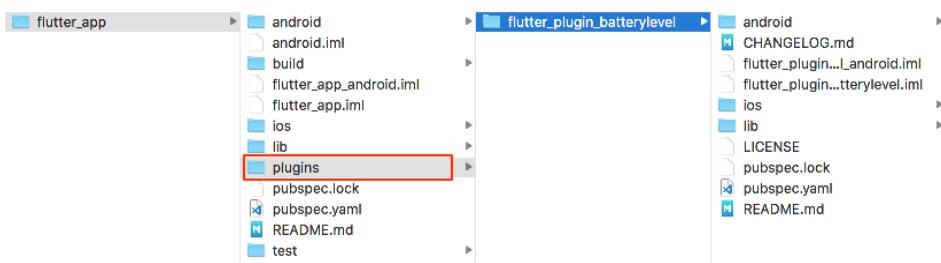
```
dependencies:
  flutter_remote_package:
    git:
      url: git@gitlab.alibaba-inc.com:churui/flutter_remote_package.git
      ref: 0.0.1
```

PS: ref 可以指定某个 commit、branch、或者 tag。

4.4 Path packages (本地)

PS: 如果你的代码没有特殊的场景需要, 可以直接把 Package 放到本地, 这样开发和调试都很方便。

我们在 Flutter App 项目根目录下 (flutter_app), 创建文件夹 (plugins), 然后把插件 (flutter_plugin_batterylevel) 移动到 plugins 下。



加载 Path packages:

编辑 pubspec.yaml:

```
dependencies:
  flutter_plugin_batterylevel:
```

```
path: plugins/flutter_plugin_batterylevel
```

5. 踩过的坑

5.1 用 XCode 编辑 Plugin

我们已经在 pubspec.yaml 里添加了依赖，但是打开 iOS 工程，却看不到 Plugin？

这时需要执行 pod install (或 pod update)。

5.2 iOS 编译没问题，但是运行时找不到 Plugin

```
@implementation AppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions {
    // Plugin注册方法
    [GeneratedPluginRegistrant registerWithRegistry:self];

    // 显示 Window
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    [self.window setRootViewController:[[FlutterViewController alloc]
initWithNibName:nil bundle:nil]];
    [self.window setBackgroundColor:[UIColor whiteColor]];
    [self.window makeKeyAndVisible];

    return [super application:application didFinishLaunchingWithOptions:launchOptions];
}

@end
```

[GeneratedPluginRegistrant registerWithRegistry:self] 默认注册到 self.window.rootViewController 的。

所以需要先初始化 rootViewController, 再注册 Plugin。

5.3 Native 调用 Flutter 失败

Flutter App 启动后，Native 调用 Flutter 失败？

这是因为 Plugin Channel 的初始化大概要 1.5 秒，而且这是一个异步过程。虽然 Flutter 页面显示出来了，但是 Plugin Channel 还没初始化完，所以这时 Native

调用 Flutter 是没反应的。

5.4 iOS Plugin 注册到指定的 FlutterViewController

闲鱼首页是 Native 页面，所以 Window 的 rootViewController 不是 FlutterViewController，直接注册 Plugin 会注册失败。我们需要将 Plugin 注册到指定的 FlutterViewController。

FlutterAppDelegate.h

```

28 @interface FlutterAppDelegate : UIResponder<UIApplicationDelegate, FlutterPluginRegistry>
29
30 @property(strong, nonatomic) UIWindow* window;
31
32 // Can be overriden by subclasses to provide a custom FlutterBinaryMessenger,
33 // typically a FlutterViewController, for plugin interop.
34 //
35 // Defaults to window's rootViewController.
36 - (NSObject<FlutterBinaryMessenger*>*)binaryMessenger;
37
38 // Can be overriden by subclasses to provide a custom FlutterTextureRegistry,
39 // typically a FlutterViewController, for plugin interop.
40 //
41 // Defaults to window's rootViewController.
42 - (NSObject<FlutterTextureRegistry*>*)textures;
43
44 @end

```

```

- (NSObject<FlutterBinaryMessenger*>*)binaryMessenger;
- (NSObject<FlutterTextureRegistry*>*)textures;

```

我们需要在 AppDelegate 重写上面两个方法，方法内返回需要指定的 FlutterViewController。

延展讨论

Flutter 作为应用层的 UI 框架，底层能力还是依赖 Native 的，所以 Flutter App 调用 Native APIs 的应用场景还是挺多的。

在 Plugin 方法调用过程中，可能会遇到传递复杂参数的情况（有时需要传递对象），但是 Plugin 的参数是 JSON 序列化后的二进制数据，所以传参必须是可 JSON 序列化的。我觉得，应该有一层对象映射层，来支持传递对象。

说到 Plugin 传参，Plugin 有个很牛逼的能力，就是传递 textures(纹理)。闲鱼

的 Flutter 视频播放，实际上是用的 Native 播放器，然后将 textures (纹理) 传递给 Flutter App。

因为后面会有 Flutter 视频播放的专题文章《万万没想到 –Flutter 这样外接纹理》，这里就不做延展了。

参考资料

<https://www.dartlang.org/tools/pub/>

<https://pub.dartlang.org>

<https://flutter.io/platform-channels/>

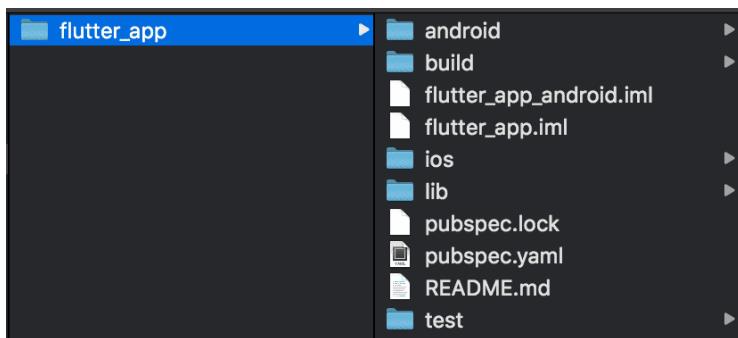
Flutter 混合工程改造实践

作者：闲鱼技术 - 字平

背景

闲鱼技术团队于 2018 年上半年率先引入了 Flutter 技术尝试实现客户端开发的统一，并成功改造和上线了复杂的商品详情业务。这一过程中，由于原有的 iOS 和安卓工程都已相当庞大，如何将 Flutter 无缝桥接到这些大工程并保证开发效率不受影响成为优先要解决的问题。

本文针对项目实践人员给出了一种通用的工程改造方案，希望为准备转型 Flutter 的团队提供参考。## 问题 Flutter 的工程结构比较特殊，由 Flutter 目录再分别包含 Native 工程的目录（即 ios 和 android 两个目录）组成。默认情况下，引入了 Flutter 的 Native 工程无法脱离父目录进行独立构建和运行，因为它会反向依赖于 Flutter 相关的库和资源。



> 典型的 Flutter 目录结构

很显然，在拥有了 Native 工程的情况下，开发者不太可能去创建一个全新的 Flutter 工程重写整个产品，因此 Flutter 工程将包含已有的 Native 工程，这样就带来了一系列问题：

- 1) 构建打包问题：引入 Flutter 后，Native 工程因对其有了依赖和耦合，从而

无法独立编译构建。在 Flutter 环境下，工程的构建是从 Flutter 的构建命令开始，执行过程中包含了 Native 工程的构建，开发者要配置完整的 Flutter 运行环境才能走通整个流程；

- 2) **混合编译带来的开发效率的降低：**在转型 Flutter 的过程中必然有许多业务仍使用 Native 进行开发，工程结构的改动会使开发无法在纯 Native 环境下进行，而适配到 Flutter 工程结构对纯 Native 开发来说又会造成不必要的构建步骤，造成开发效率的降低。

目标

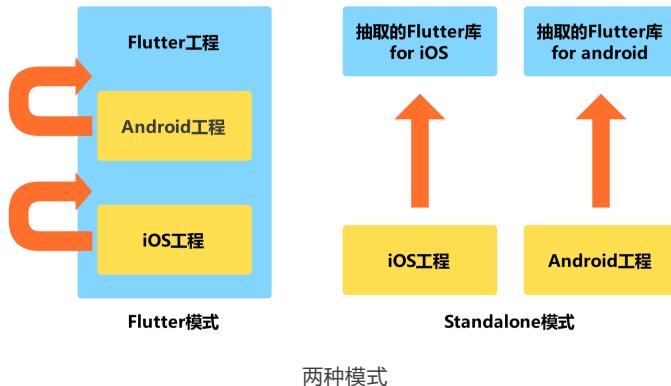
针对以上问题，我们提出了以下的改造目标，力求最小化 Native 工程对 Flutter 相关文件的依赖，使得：

- 1) Native 工程可以独立地编译构建和调试执行，进而最大限度地减少对相关开发同学的干扰并使打包平台不再依赖 Flutter 环境及相关流程；
- 2) Native 工程处在 Flutter 环境中时（即作为 ios 或 android 子目录）能够正确依赖相关库和文件，正常执行各类 Flutter 功能，如 dart 代码的构建，调试，hot reload 等，保证 Flutter 环境下开发的正确性。

方案的制定

两种模式

首先定义 Native 工程处于独立目录环境下称为 Standalone 模式，处于 Flutter 目录下称为 Flutter 模式。目标中纯 Native 开发或平台打包就处于 Standalone 模式，Flutter 对开发人员和打包平台来说是透明的存在，不会影响构建与调试；而 Flutter 的代码则在 Flutter 模式进行开发，其相关库的生成，编译和调试都走 Flutter 定义的流程。



理清依赖

从上面的定义来看，改造的核心就是把 Standalone 模式提取出来，那么就要理清 Standalone 模式对 Flutter 的依赖，并将其提取成第三方的库，资源或源码文件。以 iOS 为例，通过阅读 Flutter 构建的源码，可知 Xcode 工程对 Flutter 有如下依赖：

- 1) App.framework: dart 业务源码相关文件
- 2) Flutter.framework: Flutter 引擎库文件
- 3) pubspec_plugins 目录及用于索引的文件: Flutter 下的插件，包括各种系统的和自定义的 channels
- 4) flutter_assets: Flutter 依赖的静态资源，如字体，图片等

依赖引入的策略

改造过程中闲鱼尝试过两种依赖引入策略，以下分别进行阐述。

- 1) **本地依赖：**通过修改 Flutter 构建流程将其库文件，源码和资源直接放置到 Native 工程的子目录中进行引用，以 iOS 为例，就是将 Flutter.framework 及相关插件等做成本地的 pod 依赖，资源也复制到本地进行维护。由此，Standalone 模式便具备了独立构建和执行的能力，对于纯 Native 开发人员来说 Flutter 只是一些三方库与资源的合集，无需关注。而在 Flutter 模式下，dart 源码的构建流程不变，不影响编译和调试；同时由于是本地依赖，Flutter 模式下的各种改动也实时可以同步到 Native 工程的子目录中，提交

修改后 Standalone 模式也就拥有了最新的 Flutter 相关功能。

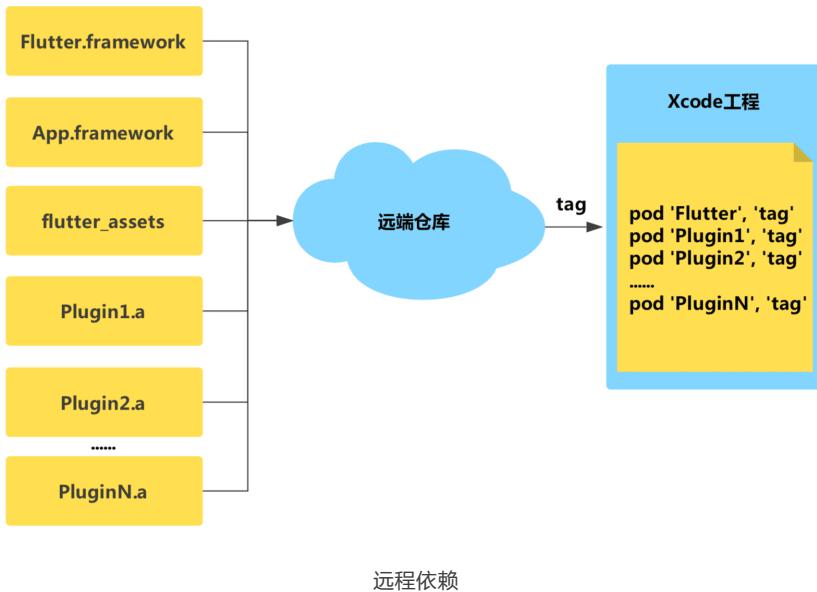
优点: Flutter 相关内容的改动同步到 Standalone 模式也比较方便；

缺点: 需要对 Flutter 原有的构造流程进行稍嫌复杂的改动，并且与后续的 Flutter 代码合并会有冲突，且 Native 工程与 Flutter 的内容还是耦合在本地不够独立。

2) **远程依赖:** 远程依赖的想法是将 Flutter 所有依赖内容都放在独立的远端仓库中，在 Standalone 模式下引用远端仓库中的相关资源，源码和库文件，Flutter 模式下的构建流程和引用方式则不变。

优点: 对 Flutter 自身的构建流程改动较少并且较彻底第解决了本地耦合的问题；

缺点: 同步的流程变得更繁琐，Flutter 内容的变动需要先同步到远端仓库再同步到 Standalone 模式方能生效。PS. 闲鱼最终选择了这个策略。



改造的实现

目录的组织

Flutter 模式下父工程目录下的 ios 和 android 的子目录分别包含对应的 Native

工程，代码管理上子工程可以使用 git 的 submodule 形式，保证目录间的独立。

远程依赖的实现 在 Standalone 模式下，Flutter 的依赖内容都指向远程仓库中的对应文件，而在 Flutter 模式下依赖的方式不变。

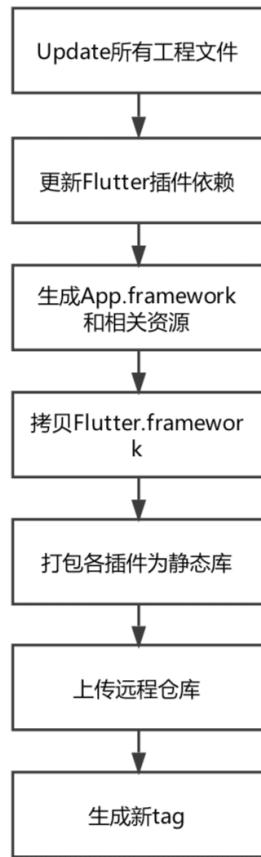
1) 向 Standalone 模式同步 Flutter 的变更

由于远程依赖的问题是同步变动比较麻烦，为此闲鱼开发了一系列脚本工具使该过程尽量自动完成。假设 Flutter 的内容（可能是业务源码，引擎库或某些资源文件）发生变化，那么在 Flutter 模式下构建结束后，脚本会提取生成好的所有依赖文件拷贝到远程仓库，提交并打 tag，然后依据打出的 tag 生成新的远程依赖说明（比如 iOS 下的 podspec），最后在 Standalone 模式下修改 Flutter 的依赖至最新的版本，从而完成整个同步过程。

2) 同步的时机

建议在提测及灰度期间，每次 Flutter 业务的提交都能够触发同步脚本的执行和 app 打包；开发期间保持每日一次的同步即可。# 总结 为解决引入 Flutter 后的工程适配问题，我们抽取了 Flutter 的相关依赖放到远程供纯 Native 工程进行引用，从而保证了 Flutter 与纯 Native 开发的相互独立与并行执行。

该方案已在闲鱼施行了几个版本，并反向输出给了 Flutter 团队，为其后续的 hybrid 工程组织计划提供了方向和参考。同时，相信该方案也可以为转型 Flutter 的团队提供帮助，当然项目间的差异也会导致方案的不同，因此如有更好的方法和意见也期望多多交流！



闲鱼 Flutter 混合工程持续集成的最佳实践

作者：闲鱼技术 – 然道

1. 引言

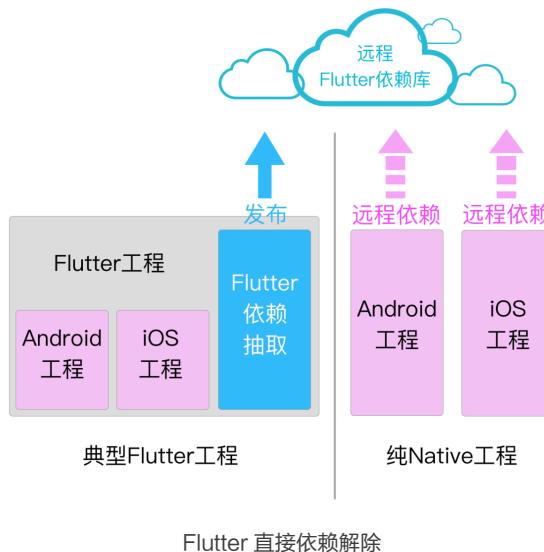
在之前的文章《Flutter 混合工程改造实践》中，有些同学留言想了解抽取 Flutter 依赖到远程的一些实现细节，所以本文重点来讲一讲 Flutter 混合工程中的 Flutter 直接依赖解除的一些具体实现。

2. 思考

因为目前我们闲鱼是 Flutter 和 Native 混合开发的模式，所以存在一部分同学只做 Native 开发，并不熟悉 Flutter 技术。

- (1) 如果直接采用 Flutter 工程结构来作为日常开发，那这部分 Native 开发同学也需要配置 Flutter 环境，了解 Flutter 一些技术，成本比较大。
- (2) 阿里集团的构建系统目前并不支持直接构建 Flutter 项目，这个也要求我们解除 Native 工程对 Flutter 的直接依赖。

鉴于这两点原因，我们希望可以设计一个 **Flutter 依赖抽取模块**，可以将 Flutter 的依赖抽取为一个 Flutter 依赖库发布到远程，供纯 Native 工程引用。如下图所示：



3. 实现

3.1 Native 工程依赖的 Flutter 分析

我们分析 Flutter 工程，会发现 Native 工程对 Flutter 工程的依赖主要有三部分：

1. **Flutter 库和引擎**: Flutter 的 Framework 库和引擎库。
2. **Flutter 工程**: 我们自己实现的 Flutter 模块功能，主要为 Flutter 工程下 lib 目录下的 dart 代码实现的这部分功能。
3. **自己实现的 Flutter Plugin**: 我们自己实现的 Flutter Plugin。

我们解开 Android 和 iOS 的 APP 文件，发现 Flutter 依赖的主要文件如下图所示：



Flutter 依赖的文件 (Flutter 产物)

其中，

Android 的 Flutter 依赖的文件：

1. Flutter 库和引擎：

icudtl.dat、libflutter.so、还有一些 class 文件。这些都封装在 flutter.jar 中，这个 jar 文件位于 Flutter 库目录下的 [flutter/bin/cache/artifacts/engine] 下。

2. Flutter 工程产物：

isolate_snapshot_data、isolate_snapshot_instr、vm_snapshot_data、vm_snapshot_instr、flutter_assets。

3. Flutter Plugin：

各个 plugin 编译出来的 aar 文件。

其中：

- isolate_snapshot_data 应用程序数据段
- isolate_snapshot_instr 应用程序指令段
- vm_snapshot_data VM 虚拟机数据段
- vm_snapshot_instr VM 虚拟机指令段

iOS 的 Flutter 依赖的文件：

1. **Flutter 库和引擎:** Flutter.framework
2. **Flutter 工程的产物:** App.framework
3. **Flutter Plugin:** 编译出来的各种 plugin 的 framework, 图中的其他 framework

那我们只需要将这三部分的编译结果抽取出来，打包成一个 SDK 依赖的形式提供给 Native 工程，就可以解除 Native 工程对 Flutter 工程的直接依赖。

3.2 Android 依赖的 Flutter 库抽取

3.2.1 Android 中 Flutter 编译任务分析

Flutter 工程的 Android 打包，其实只是在 Android 的 Gradle 任务中插入了一个 flutter.gradle 的任务，而这个 flutter.gradle 主要做了三件事：(这个文件可以在 Flutter 库中的 [flutter/packages/flutter_tools/gradle] 目录下能找到。)

1. 增加 flutter.jar 的依赖。
2. 插入 Flutter Plugin 的编译依赖。
3. 插入 Flutter 工程的编译任务，最终将产物（两个 isolate_snapshot 文件、两个 vm_snapshot 文件和 flutter_assets 文件夹）拷贝到 mergeAssets.outputDir，最终 merge 到 APK 的 assets 目录下。

3.2.2 Android 的 Flutter 依赖抽取实现

弄明白 Flutter 工程的 Android 编译产物之后，因此我们对 Android 的 Flutter 依赖抽取步骤如下：

1. 编译 Flutter 工程。

这部分主要工作是编译 Flutter 的 dart 和资源部分，可以用 AOT 和 Bundle 命令编译。

```
echo "Clean old build"
find . -d -name "build" | xargs rm -rf
./flutter/bin/flutter clean

echo "Get packages"
```

```
./flutter/bin/flutter packages get

echo "Build release AOT"
./flutter/bin/flutter build aot --release --preview-dart-2 --output-dir=build/
flutteroutput/aot
echo "Build release Bundle"
./flutter/bin/flutter build bundle --precompiled --preview-dart-2 --asset-
dir=build/flutteroutput/flutter_assets
```

2. 将 flutter.jar 和 Flutter 工程的产物打包成一个 aar。

这边部分的主要工作是将 flutter.jar 和第 1 步编译的产物封装成一个 aar。

(1) 添加 flutter.jar 依赖

```
project.android.buildTypes.each {
    addFlutterJarImplementationDependency(project, releaseFlutterJar)
}

project.android.buildTypes.whenObjectAdded {
    addFlutterJarImplementationDependency(project, releaseFlutterJar)
}

private static void addFlutterJarImplementationDependency(Project project,
releaseFlutterJar) {
    project.dependencies {
        String configuration
        if (project.getConfigurations().findByName("api")) {
            configuration = "api"
        } else {
            configuration = "compile"
        }
        add(configuration, project.files {
            releaseFlutterJar
        })
    }
}
```

(2) Merge Flutter 的产物到 assets

```
// merge flutter assets
def allertAsset ="${project.projectDir.getAbsolutePath()}/flutter/assets/release"
Task mergeFlutterAssets = project.tasks.create(name: "mergeFlutterAssets${
variant.name.capitalize()}", type: Copy) {
    dependsOn mergeFlutterMD5Assets
    from (allertAsset){
        include "flutter_assets/**" // the working dir and its files
```

```

        include "vm_snapshot_data"
        include "vm_snapshot_instr"
        include "isolate_snapshot_data"
        include "isolate_snapshot_instr"
    }
    into variant.mergeAssets.outputDir
}
variant.outputs[0].processResources.dependsOn(mergeFlutterAssets)

```

3. 同时将这个 aar 和 Flutter Plugin 编译出来的 aar 一起发布到 maven 仓库。

(1) 发布 Flutter 工程产物打包的 aar

```

echo 'Clean packflutter input/flutter build'
rm -f -r android/packflutter/flutter/

# 拷贝 flutter.jar
echo 'Copy flutter jar'
mkdir -p android/packflutter/flutter/flutter/android-arm-release && cp
flutter/bin/cache/artifacts/engine/android-arm-release/flutter.jar "$_"

# 拷贝 asset
echo 'Copy flutter asset'
mkdir -p android/packflutter/flutter/assets/release && cp -r build/flutteroutput
/aot/* "$_"
mkdir -p android/packflutter/flutter/assets/release/flutter_assets && cp -r
build/flutteroutput/flutter_assets/* "$_"

# 将 flutter 库和 flutter_app 打成 aar 同时 publish 到 Ali-maven
echo 'Build and publish idlefish flutter to aar'
cd android
if [ -n "$1" ]
then
    ./gradlew :packflutter:clean :packflutter:publish -PAAR_VERSION=$1
else
    ./gradlew :packflutter:clean :packflutter:publish
fi
cd ../

```

(2) 发布 Flutter Plugin 的 aar

```

# 将 plugin 发布到 Ali-maven
echo "Start publish flutter-plugins"
for line in $(cat .flutter-plugins)
do

```

```

plugin_name=${line%%=*}
echo 'Build and publish plugin:' ${plugin_name}

cd android
if [ -n "$1" ]
then
    ./gradlew :${plugin_name}:clean :${plugin_name}:publish -PAAR_VERSION=$1
else
    ./gradlew :${plugin_name}:clean :${plugin_name}:publish
fi
cd ../
done

```

4. 纯粹的 Native 项目只需要 compile 我们发布到 maven 的 aar 即可。

平时开发阶段，我们需要实时能依赖最新的 aar，所以我们采用 SNAPSHOT 版本。

```

configurations.all {
    resolutionStrategy.cacheChangingModulesFor 0, 'seconds'
}

ext {
    flutter_aar_version = 'X.X.X-SNAPSHOT'
}

dependencies {
    //flutter 主工程依赖：包含基于 flutter 开发的功能、flutter 引擎 lib
    compile("XXX.XXX.XXX:IdleFishFlutter:${getFlutterAarVersion(project)}") {
        changing = true
    }
    //... 其他依赖
}

static def getFlutterAarVersion(project) {
    def resultVersion = project.flutter_aar_version
    if (project.hasProperty('FLUTTER_AAR_VERSION')) {
        resultVersion = project.FLUTTER_AAR_VERSION
    }
    return resultVersion
}

```

3.3 iOS 依赖的 Flutter 库的抽取

3.3.1 iOS 中 Flutter 依赖文件如何产生

执行编译命令“flutter build ios”，最终会执行 Flutter 的编译脚本 [xcode_

backend.sh], 而这个脚本主要做了下面几件事:

1. 获取各种参数 (如 project_path, target_path, build_mode 等), 主要来自于 Generated.xcconfig 的各种定义。
2. 删除 Flutter 目录下的 App.framework 和 app.flx。
3. 对比 Flutter/Flutter.framework 与 FLUTTER_ROOT/bin/cache/artifacts/engine/{artifact_variant} 目录下的 Flutter.framework, 若不相等, 则用后者覆盖前者。
4. 获取生成 App.framework 命令所需参数 (build_dir, local_engine_flag, preview_dart_2_flag, aot_flags)。
5. 生成 App.framework, 并将生成的 App.framework 和 AppFramework-Info.plist 拷贝到 XCode 工程的 Flutter 目录下。

3.3.2 iOS 的 Flutter 依赖抽取实现

iOS 的 Flutter 依赖的抽取步骤如下:

1. 编译 Flutter 工程生成 App.framework。

```
echo "==== 清理 flutter 历史编译 ==="
./flutter/bin/flutter clean

echo "==== 重新生成 plugin 索引 ==="
./flutter/bin/flutter packages get

echo "==== 生成 App.framework 和 flutter_assets ==="
./flutter/bin/flutter build ios --release
```

2. 打包各插件为静态库。

这里主要有两步: 一是将 plugin 打成二进制文件, 二是将 plugin 的注册入口打成二进制文件。

```
echo "==== 生成各个 plugin 的二进制库文件 ==="
cd ios/Pods
#/usr/bin/env xcrun xcodebuild clean
#/usr/bin/env xcrun xcodebuild build -configuration Release ARCHS='arm64
armv7' BUILD_AOT_ONLY=YES VERBOSE_SCRIPT_LOGGING=YES -workspace Runner.
```

```

xcworkspace -scheme Runner BUILD_DIR=../build/ios -sdk iphoneos
for plugin_name in ${plugin_arr}
do
    echo "生成 lib${plugin_name}.a..."
    /usr/bin/env xcrun xcodebuild build -configuration Release ARCHS='arm64
armv7' -target ${plugin_name} BUILD_DIR=../../build/ios -sdk iphoneos -quiet
    /usr/bin/env xcrun xcodebuild build -configuration Debug ARCHS='x86_64'
-target ${plugin_name} BUILD_DIR=../../build/ios -sdk iphonesimulator -quiet
    echo "合并 lib${plugin_name}.a..."
    lipo -create "../../build/ios/Debug-iphonesimulator/${plugin_name}/
lib${plugin_name}.a" "../../build/ios/Release-iphoneos/${plugin_name}/
lib${plugin_name}.a" -o "../../build/ios/Release-iphoneos/${plugin_name}/
lib${plugin_name}.a"
done

echo "== 生成注册入口的二进制库文件 =="
for reg_enter_name in "flutter_plugin_entrance" "flutter_service_register"
do
    echo "生成 lib${reg_enter_name}.a..."
    /usr/bin/env xcrun xcodebuild build -configuration Release ARCHS='arm64
armv7' -target ${reg_enter_name} BUILD_DIR=../../build/ios -sdk iphoneos
    /usr/bin/env xcrun xcodebuild build -configuration Debug ARCHS='x86_64'
-target ${reg_enter_name} BUILD_DIR=../../build/ios -sdk iphonesimulator
    echo "合并 lib${reg_enter_name}.a..."
    lipo -create "../../build/ios/Debug-iphonesimulator/${reg_enter_name}/
lib${reg_enter_name}.a" "../../build/ios/Release-iphoneos/${reg_enter_name}/
lib${reg_enter_name}.a" -o "../../build/ios/Release-iphoneos/${reg_enter_
name}/lib${reg_enter_name}.a"
done

```

3. 将这些上传到远程仓库，并生成新的 Tag。

4. 纯 Native 项目只需要更新 pod 依赖即可。

4. Flutter 混合工程的持续集成流程

按上述方式，我们就可以解除 Native 工程对 Flutter 工程的直接依赖了，但是在日常开发中还是存在一些问题：

1. Flutter 工程更新，远程依赖库更新不及时。
2. 版本集成时，容易忘记更新远程依赖库，导致版本没有集成最新 Flutter 功能。
3. 同时多条线并行开发 Flutter 时，版本管理混乱，容易出现远水库被覆盖的问题。

4. 需要最少一名同学持续跟进发布，人工成本较高。

鉴于这些问题，我们引入了我们团队的 CI 自动化框架，从两方面来解决：

(关于 CI 自动化框架，我们后续会撰文分享)

一方面是自动化，通过自动化减少人工成本，也减少人为失误。

另一方面是做好版本控制，自动化的形式来做版本控制。

具体操作：

首先，每次需要构建纯粹 Native 工程前自动完成 Flutter 工程对应的远端库的编译发布工作，整个过程不需要人工干预。

其次，在开发测试阶段，采用五段式的版本号，最后一位自动递增产生，这样就可以保证测试阶段的所有并行开发的 Flutter 库的版本号不会产生冲突。

最后，在发布阶段，采用三段式或四段式的版本号，可以和 APP 版本号保持一致，便于后续问题追溯。

整个流程如下图所示：



5. 写在最后

构建作为项目必须的第一步，很多团队都有自己不同的模式和流程。但是基于混合 Flutter 的项目，Flutter 混合构建是无法越过的一步坎。所以可以借鉴本文思路，我们可以针对不同的标准制定个性化混合构建流程，我们也开始尝试将我们的构建模式对接摩天轮中，变成一种集团标准打包模式。同时也欢迎和我们联系讨论 Flutter 混合构建的新的可能模式。

Flutter 新锐专家之路：工程研发体系篇

作者：闲鱼技术 – 正物

写在前面

当前，闲鱼客户端已经实现了基于 Flutter 的商品详情页的全量重构，线上效果良好。从 alpha 一路走来，我们遇到了很多问题，或基于原理，或透过社区，或与官方合作，都一个个解决了，是时候梳理和总结下，也希望为其他的开发者们，尤其是已有工程中引入 Flutter（混合场景）实现渐进式重构带来启发和帮助。鉴于存在多个问题一个原因或解法的情况，而本系列的重点在于说明各种问题的解决方案与思路，就不一一列出问题。所有调试 / 热重载相关的 Flutter 均为 Debug 模式的 Flutter，不再特殊说明。

本系列文章包含三篇：引入篇，运行篇，上线篇。引入篇重点介绍工程研发体系；运行篇介绍混合情景下的栈管理与能力补齐等；上线篇介绍兼容 / 稳定性保障及方法。

工程研发体系的关键点包括：

- a. 混合工程下的 Flutter 研发结构

混合工程中一个全局视角的研发结构如何。

- b. 工程结构

已有的 Native 工程如何引入 Flutter，工程结构如何组织，如何管理 Flutter 环境，如何去编译构建，集成打包等。

- c. 构建优化

这里主要介绍如何去针对 Flutter 的工具链 (flutter_tools, IntelliJ 插件等) 进行调试与优化。

- d. Native 启动下的 Flutter 调试

不同于 Flutter 启动下的一体化调试，这种 Native 启动 (Xcode/Android Studio 启动，或点击图标打开应用) 下的 Flutter 调试，我称之为分离式调试。分离式调试可

以简化 flutter_tools 带来的复杂度，提高调试的稳定性和灵活性。

e. Native 启动下的 Flutter 热重载

同 d。

f. 联合调试

即同时调试 Flutter 和 Android/iOS。

g. 持续集成

即混合环境下的 Flutter 构建与持续集成。

环境说明

```
[kylewong@KyleWongdeMacBook-Pro fwn_idlefish % flutter/bin/flutter doctor -v
[KWLM]:[doctor, -v]
[!] Flutter (Channel alf_stable_v1.5.4, v1.5.4-hotfix.3-pre.44, on Mac OS X 10.15 19A536g, locale en-CN)
• Flutter version 1.5.4-hotfix.3-pre.44 at /Users/kylewong/Codes/fwn_idlefish/flutter
• Framework revision 1041df667e (4 days ago), 2019-08-22 01:28:05 +0800
• Engine revision 52c7a1e849
• Dart version 2.3.0 (build 2.3.0-dev.0.5 a1668566e5)
✗ Downloaded executables cannot execute on host.
See https://github.com/flutter/flutter/issues/6207 for more information

[✓] Android toolchain - develop for Android devices (Android SDK version 29.0.2)
• Android SDK at /Users/kylewong/Library/Android/sdk
• Android NDK location not configured (optional; useful for native profiling support)
• Platform android-29, build-tools 29.0.2
• Java binary at: /Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java
• Java version OpenJDK Runtime Environment (build 1.8.0_202-release-1483-b49-5587405)
• All Android licenses accepted.

[!] iOS toolchain - develop for iOS devices (Xcode 11.0)
• Xcode at /Applications/Xcode-beta.app/Contents/Developer
• Xcode 11.0, Build version 11M382q
• ios-deploy 1.9.4
! Unknown CocoaPods version installed.
  Flutter is unable to determine the installed CocoaPods's version.
  Ensure that the output of 'pod --version' contains only digits and . to be recognized by Flutter.
To upgrade:
  brew upgrade cocoapods
  pod setup

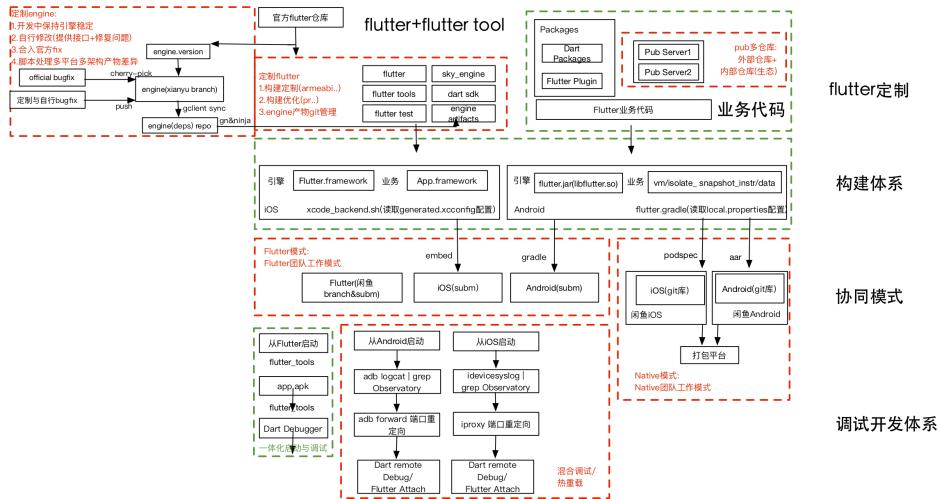
[✓] Android Studio (version 3.5)
• Android Studio at /Applications/Android Studio.app/Contents
• Flutter plugin version 38.2.3
• Dart plugin version 191.8405
• Java version OpenJDK Runtime Environment (build 1.8.0_202-release-1483-b49-5587405)

[✓] VS Code (version 1.37.1)
• VS Code at /Applications/Visual Studio Code.app/Contents
• Flutter extension version 3.3.0

[✓] Connected device (1 available)
• MHA AL00 • GWY7N16A31002764 • android-arm64 • Android 9 (API 28)
```

本系列使用的环境

混合工程下的 Flutter 研发结构



混合工程下的 Flutter 研发结构

工程结构

这部分的核心逻辑是如何在最小改动已有 iOS/Android 工程的前提下运行 Flutter。我们可以将 Flutter 部分理解成为一个单独的模块，通过 pod 库 (iOS),aar 库 (Android) 的方式，由 CocoaPods 和 Gradle 引入到主工程。

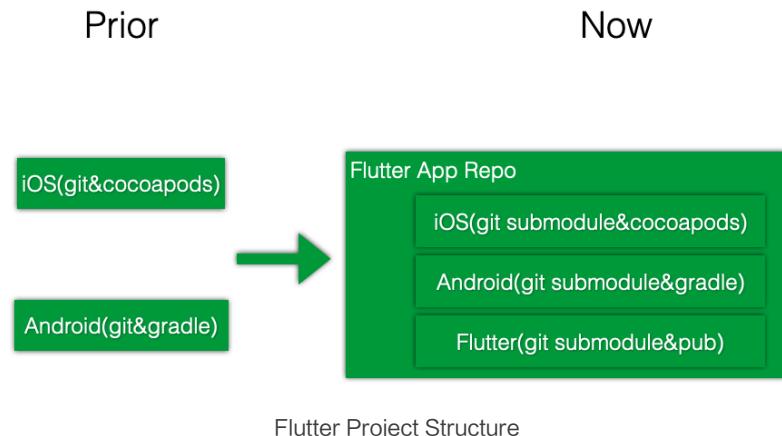
具体的原理与实践请参见：

[深入理解 flutter 的编译原理与优化](#)

[Flutter 混合工程改造实践](#)

[Add Flutter to existing apps](#)

其中，我们将整套 Flutter 环境作为 Git Submodule 统一管理，以保证团队内环境一致，遇到的个性化的问题 / 需求能够统一处理。



构建优化篇

编译速度的优化 (Android)

问题: Android 在由 Flutter 启动时构建缓慢。

原因: 在 flutter 工具链 (flutter_tools) 的逻辑中, 未找到 android/app/build.gradle 时, 会运行 gradle build 从而执行多个编译配置的构建, 而不是 gradle assembleDebug。

解法: 重构 Android 工程, 使工程应用 Module 对应的 build.gradle 位于 android/app 下, 从而符合 flutter_tools 的逻辑。

原理: flutter_tools 的调试

a. 修改 flutter_tools.dart, 使之可打印参数

```
flutter_tools.dart x
1  /...
2
3
4
5  import 'package:flutter_tools/executable.dart' as executable;
6
7  void main(List<String> args) {
8    print('[KWL]: ' + args.toString());
9    executable.main(args);
10 }
11
```

修改 flutter_tools 打印参数

b. 删除 flutter/bin/cache/flutter_tools.stamp 使得 flutter_tools 可以被重建

```

103 local revision=$(cd "$FLUTTER_ROOT"; git rev-parse HEAD)
104
105 # Invalidate cache if:
106 # * SNAPSHOT_PATH is not a file, or
107 # * STAMP_PATH is not a file with nonzero size, or
108 # * Content of STAMP_PATH is not our local git HEAD revision, or
109 # * Content of pubspec.yaml last modified after pubspec.lock
110 if [[ ! -f "$SNAPSHOT_PATH" || ! -s "$STAMP_PATH" || ! $(cat "$STAMP_PATH") != "$revision" || ! "$FLUTTER_TOOLS_DIR/pubspec.yaml" -nt "$FLUTTER_TOOLS_DIR/pubspec.lock" ]]
111   rm -f "$FLUTTER_TOOLS_DIR/.stamp"
112   touch "$FLUTTER_ROOT/bin/cache/dartignore"
113   "$FLUTTER_ROOT/bin/internal/update_dart_sdk.sh"
114   VERBOSE=-v Verbosity=error
115
116 echo Building flutter tool...
117 if [[ $CI == "true" || $ABORT == "true" || ${CONTINUOUS_INTEGRATION} == "true" || ${CHROME_HEADLESS} == "1" ]]; then
118   PUB_ENVIRONMENT=$PUB_ENVIRONMENT:flutter_bot
119   VERBOSE=-v Verbosity=normal
120 fi
121 export PUB_ENVIRONMENT=$PUB_ENVIRONMENT:flutter_install
122
123 if [[ -d "$FLUTTER_ROOT/.pub-cache" ]]; then
124   export PUB_CACHE=${PUB_CACHE:-$FLUTTER_ROOT/.pub-cache}
125 fi
126
127 retry_upgrade
128
129 "dart" $FLUTTER_TOOL_ARGS --snapshot="$SNAPSHOT_PATH" --packages="$FLUTTER_TOOLS_DIR/.packages" "$SCRIPT_PATH"
130 echo "Revision" > "$STAMP_PATH"
131
132 # The exit here is duplicitous since the function is run in a subshell,
133 # but this serves as documentation that running the function in a
134 # subshell is required to make sure any lockfile created by shlock
135 # is cleaned up.
136 exit $?
137
138 }
139
140 PROG_NAME=${path_uri "$(follow_links "$BASH_SOURCE")"}
141 BIN_DIR=$(cd "${PROG_NAME%/*}"; pwd -P)
142 export FLUTTER_ROOT=$(cd "$BIN_DIR"../; pwd -P)
143
144 FLUTTER_TOOLS_DIR="$FLUTTER_ROOT/packages/flutter_tools"
145 SNAPSHOT_PATH="$FLUTTER_ROOT/bin/cache/flutter_tools.snapshot"
146 STAMP_PATH="$FLUTTER_ROOT/bin/cache/flutter_tools.stamp"
147 SCRIPT_PATH="$FLUTTER_TOOLS_DIR/bin/flutter_tools.dart"
148 DART_SDK_PATH="$FLUTTER_ROOT/bin/cache/dart-sdk"
149
150 DART="$DART_SDK_PATH/bin/dart"
151 PUB="$DART_SDK_PATH/bin/pub"
152

```

flutter_tools build principle

c. 从 flutter 运行构建，获取其入口参数

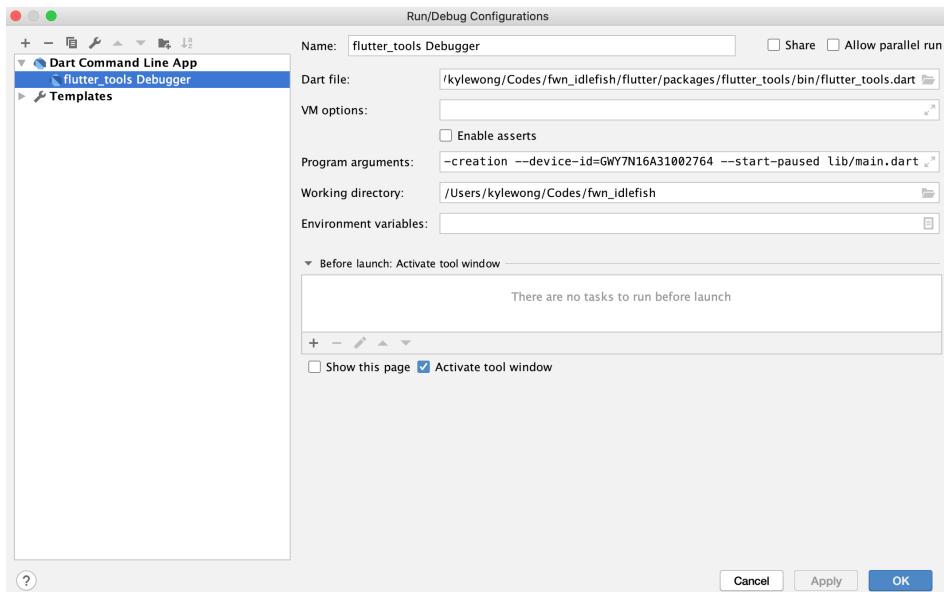
```

Building flutter tool...
[KWLM]:[--no-color, run, --machine, --track-widget-creation, --device-id=Gwy7N16A31002764, --start-paused, lib/main.dart]
Running "flutter packages get" in fwn_idlefish...

```

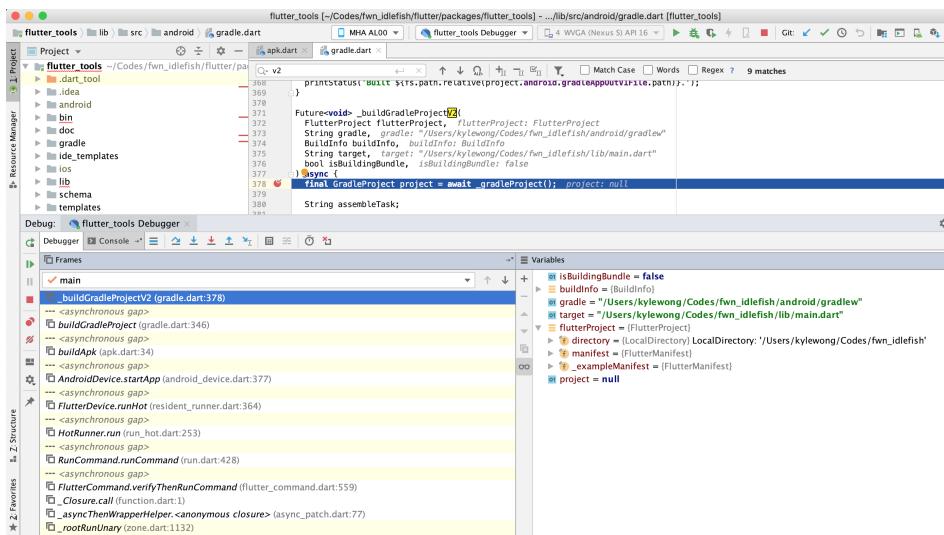
flutter_tools_arguments_print.png

d. 用 IntelliJ(或 Android Studio 下同) 打开 flutter_tools 工程，新建 Dart Command Line App，并基于步骤 c 获得的入参配置“Program arguments”



Dart–Command–Line–App–Flutter_Tools_Debugging

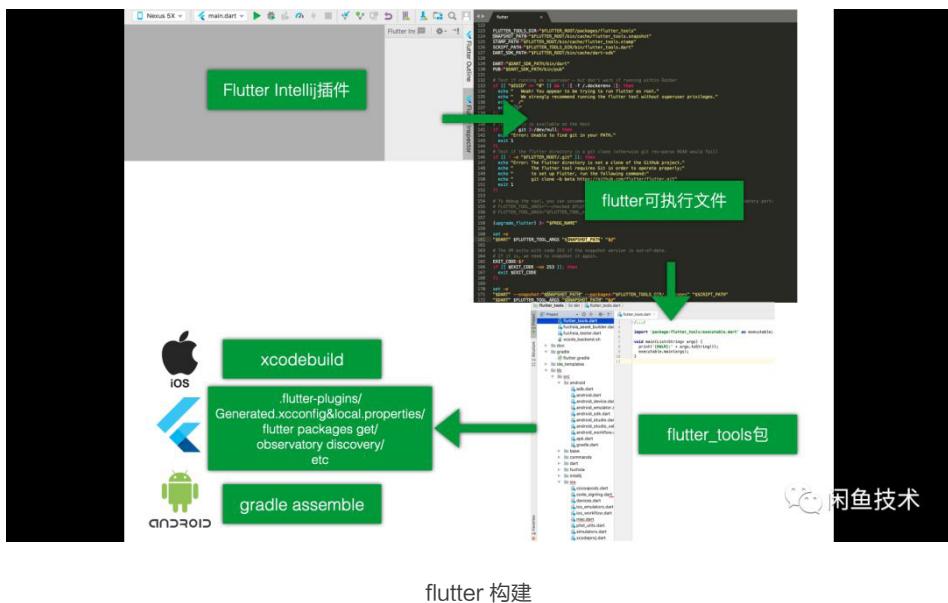
e. 开始你的 flutter_tools 调试之旅吧



flutter_tools_debugger_frame_variables

Native 视角下的 Flutter 调试

在 Flutter 模式下，Flutter 插件调用 xcodebuild(gradle) 命令去构建 iOS(Android) 工程。对于 Native 背景的开发者来说，这不仅有些不适应，也常因为 xcodebuild 等命令的参数问题，导致重复编译，当 Native 工程规模庞大时尤为复杂。如何解决这个问题呢？这就涉及到 Flutter 视角和 Native 视角下的 Flutter 调试与热重载。



Flutter 启动下的 Flutter 的调试与热重载逻辑

实际上，当 Native 工程配置好 Flutter 支持后，Flutter 启动下做的事情主要有：

- a. 检查是否需要重新生成 flutter_tools.snapshot。
 - b. 基于 pubspec.yaml 获取依赖 (pub packages get)，并生成插件描述文件 .flutter-plugins 和 pubspec.lock。
 - c. 基于 Flutter 配置 (如 Framework 路径，Debug/Release 模式，是否开启 Dart2 等)，生成 Generated.xcconfig(iOS) 和 local.properties(Android)。
 - d. 基于 gradle 和 xcodebuild 构建应用 (Flutter 相关构建请参见前文中深入理解)。

- 解 flutter 的编译原理与优化)。
- e. 基于 adb 和 lldb 启动应用。
 - f. 等待应用中 Flutter 启动，寻找 Observatory 端口，通过 Dart Debugger 连接以便调试。
 - g. 寻找到端口后同步 Hot Reload 依赖的文件，同时透过 Daemon 监听命令(如用户点击插件按钮)实现 Full Restart 或 Hot Reload。

换个角度来看，如果我们能够解决 Native 启动下的 Dart 调试和 Hot Reload，由 flutter_tools 造成的编译慢等问题将不是问题，且可解决调试环境不稳定的情况(如我们的场景下，应用启动后，仅当用户点击进入详情页面的时候才会启动 Flutter，此时 flutter_tools 才能去发现 Observatory 端口，调试和热重载，常有不好用的情况)。当从 Xcode 启动(或点击桌面图标启动，不再重复)包含了 Debug 模式 Flutter 内容的 iOS(Android Studio 启动 Android 类似，这里不再重复)应用时，我们需要关注 abcfg。而 abc 除非 flutter_tools 或 pubspec.yaml 或 Flutter 配置变化等，否则都不需要重新执行。fg 则是研发依赖的调试与热重载，必须考虑此模式下如何支持。

Native 启动下的 Flutter 的调试与热重载逻辑

- a. 寻找 iOS 设备上 Observatory 端口
命令行通过 idevicesyslog 获取，此处涉及到 libimobiledevice 库，其包含了 idevicesyslog, iproxy 等命令。

```
kylewong@KyleWongdeMacBook-Pro ios % idevicesyslog | grep listening
Aug 26 14:07:18 KyleWongs-iPhone Runner(Flutter)[686] <Notice>: flutter: Observatory listening on http://127.0.0.1:56486/oB7rzB0DQ3vU=/
```

```
observatory-log-from-command-line
```

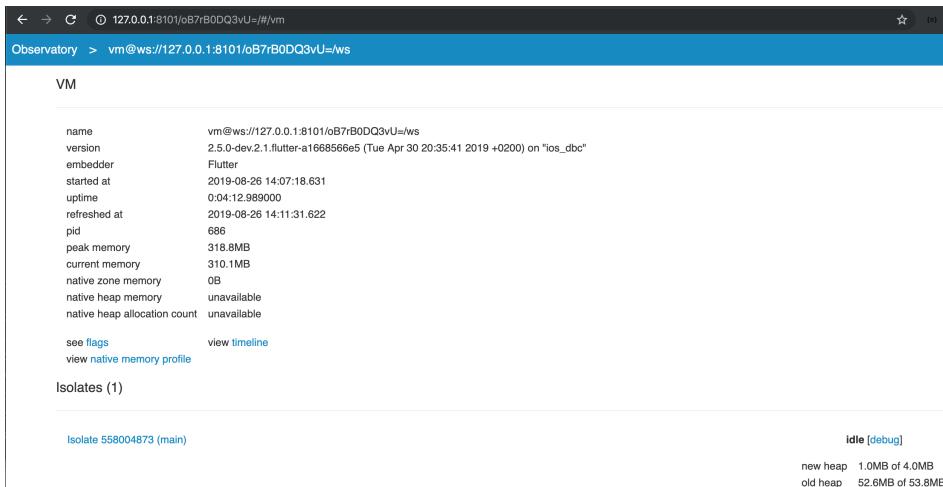
可以看到 iOS 设备上 Observatory 启动了一个 x 的端口(端口号随机)，认证码为 y。

- b. 透过 iproxy 将 iOS 设备上端口 x 映射到本机端口 z

```
kylewong@KyleWongdeMacBook-Pro ios % iproxy 8101 56486 1c8e085cf2ff6fa27643ab4afec4bf4a077688af
waiting for connection
```

```
using-iproxy-to-forward-ios-debug-port
```

- c. 可以看到 waiting for connection, 此时就可以访问 `http://127.0.0.1:z/y/#/vm` 打开 Observatory 如下：

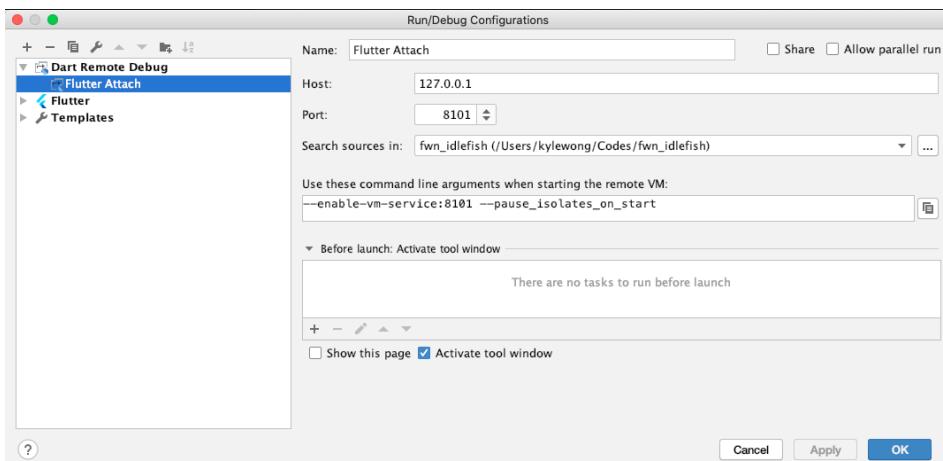


observatory-snapshot

可以使用 Observatory 去检查诸多 dart 相关的内存，调试等，这里不展开。

也可以通过 IDE 链接去调试：

- d. 配置 Dart Remote Debug

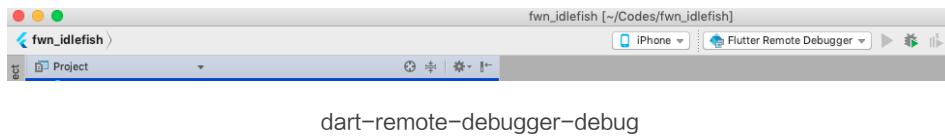


dart-remote-debug

这里需要注意的是端口要使用刚转发到电脑的端口 z，搜索源码路径是 Flutter 工程的根目录。

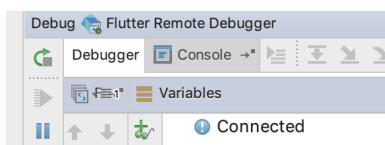
并且为了避免因为认证码造成无法连接的问题，启动时需要传入 '--disable-service-auth-codes' 标志。

e. 配置好之后点击 Debug 按钮，连接到调试端口



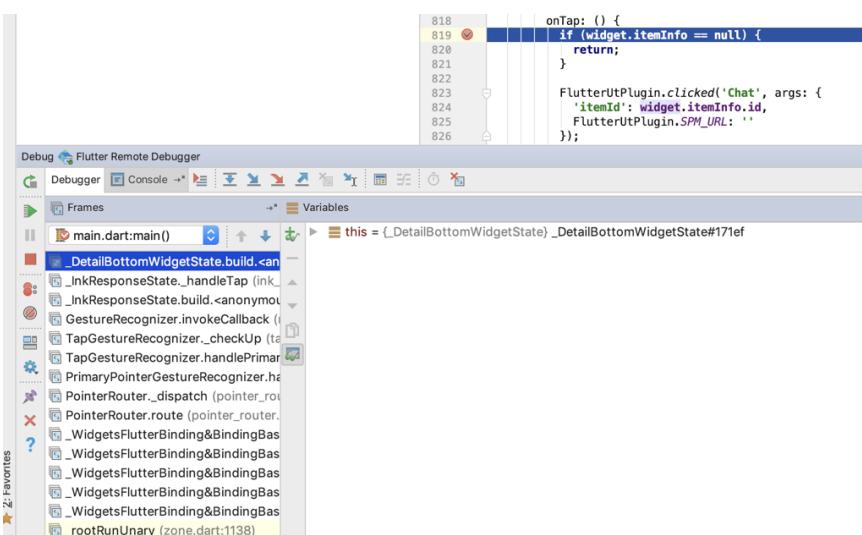
dart-remote-debugger-debug

f. 成功后可以看到 Debugger 显示 Connected (如果没有显示，再点击一次绿色的调试按钮 ())



dart-remote-debugger-connected

g. 之后便可以正常地使用 IDE 设置断点和调试 dart(Flutter) 代码



dart-debugger-remote-debug-connected-frame-info

Native 视角下的 Flutter 热重载

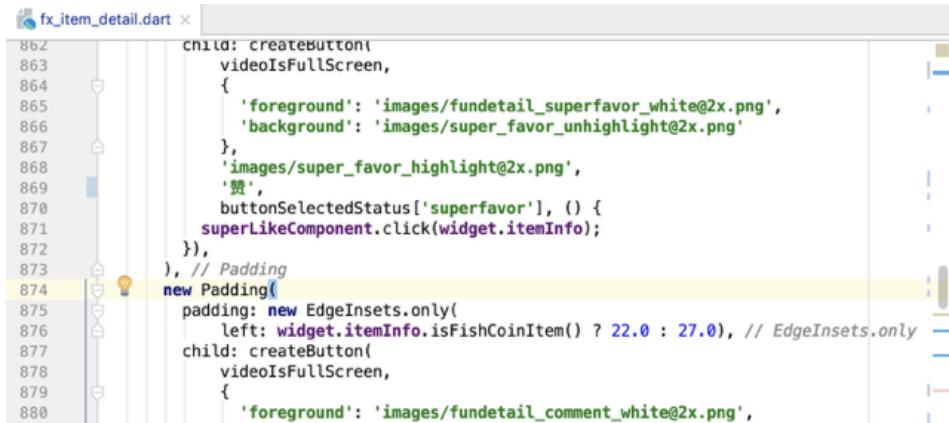
- 启动 App，进入 Flutter 页面，查找 Observatory 端口 x 和认证码 y(同上面 ab)
- 在 Flutter 工程目录下，执行 flutter attach --debug-uri=http://127.0.0.1:x/y/

```
Kylewong@KyleWongdeMacBook-Pro fwn_idlefish % flutter/bin/flutter attach --debug-uri=http://127.0.0.1:63515/2T0iU5TV0As=/
Syncing files to device KyleWong's iPhone...
3,949ms (!)

🔥 To hot reload changes while running, press "r". To hot restart (and rebuild state), press "R".
An Observatory debugger and profiler on KyleWong's iPhone is available at: http://127.0.0.1:1025/2T0iU5TV0As=/
For a more detailed help message, press "h". To detach, press "d"; to quit, press "q".
```

flutter-attach-command

- 修改 dart 源代码，然后在 b 中 Terminal 中输入 r(这一输入位于上图中 'To quit,press"q"' 之后)



flutter-attach-hotreload-code-changes

这里我们将超赞文案换成了赞。

- 可以看到 Terminal 显示“Initializing hot reload...Reloaded...”，结束后，设备上变更生效 (左下角文案变成了赞)



flutter-hot-reload-effected-result

Android 下，Native 启动的的 Flutter 调试 / 热重载类似 iOS，不同的是获取端口时可通过 IDE logcat 或者 adb logcat | grep Observatory，端口转发使用 adb forward。#Native 与 Flutter 联调 上文中已经介绍了如何在任意时刻 (Flutter 启动后) 调试 Flutter。此外我们还可以使用 Android Studio 的 Attach Debugger to Android Process 来调试 Android，这就实现了 Android 与 Flutter 联调。同样，结合 Xcode 的 Attach to Process，可以实现 iOS 与 Flutter 联调。

持续集成

目前团队包括 Native 同学和 Flutter 同学，因此我们区分了 Flutter 模式和 Native 模式。有一台公共设备 (Mac Mini) 安装了 Flutter 环境并负责 Flutter 相关的

构建，构建好的产物以 aar(Android) 或 pod 库 (iOS) 的形式集成到 Native 工程下（可以认为 Flutter 相关的代码就是一个模块），用于构建最终产物 apk(Android) 或 ipa(iOS) 的 CI 平台最终也通过产物方式集成 Flutter 并打包。

更多细节请参见：

[闲鱼 flutter 混合工程持续集成的最佳实践](#)

写在后面

本文着重介绍了混合场景下的工程研发体系。解决这一问题后，接下来就要解决实际业务开发中遇到的问题。比如 Native 与 Flutter 互相跳转场景下的栈如何管理，Flutter 不能实现的功能（平台特性等）如何去补全，Flutter Plugin/Dart Package 包管理的方式有哪些等，这些敬请关注本系列的运行篇。

Android Flutter 实践内存初探

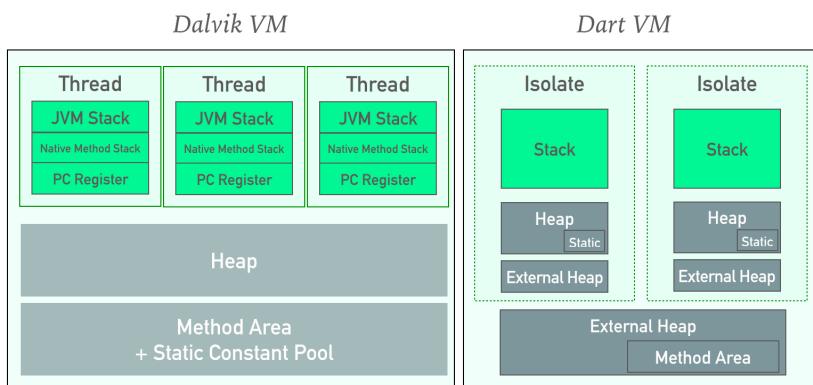
作者：闲鱼技术 - 匠修

我们想使用 Flutter 来统一移动 App 开发并做了一些实践。移动设备上的资源有限，通常内存使用都是一个我们日常开发中十分关注的问题。那么，Flutter 是如何使用内存，又会对 Native App 的内存带来哪些影响呢？本文将简单介绍 Flutter 内存机制，结合测试和我们的开发实践，对日常关心的 Bitmap 内存使用，View 绘制内存使用方面做一些探索。

Dart RunTime 简介

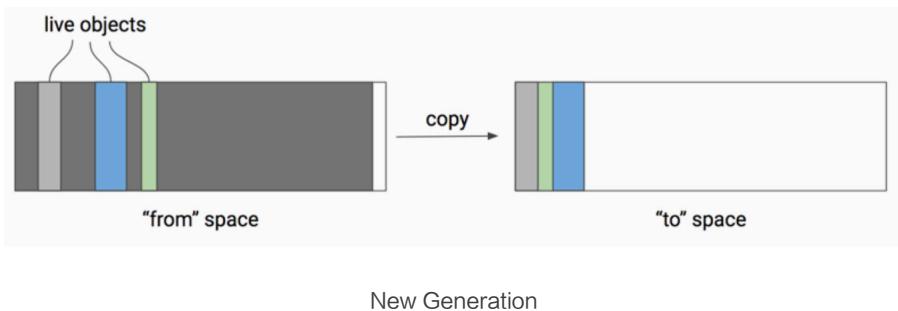
Flutter Framework 使用 Dart 语言开发，所以 App 进程中需要一个 Dart 运行环境（VM），和 Android Art 一样，Flutter 也对 Dart 源码做了 AOT 编译，直接将 Dart 源码编译成了本地字节码，没有了解释执行的过程，提升执行性能。这里重点关注 Dart VM 内存分配（Allocate）和回收（GC）相关的部分。

和 Java 显著不同的是 Dart 的“线程”（Isolate）是不共享内存的，各自的堆（Heap）和栈（Stack）都是隔离的，并且是各自独立 GC 的，彼此之间通过消息通道来通信。Dart 天然不存在数据竞争和变量状态同步的问题，整个 Flutter Framework Widget 的渲染过程都运行在一个 isolate 中。

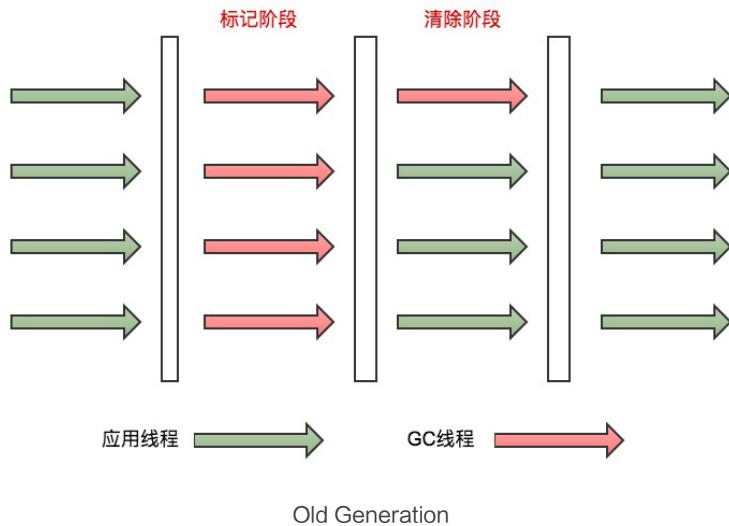


Dart VM 将内存管理分为新生代 (New Generation) 和老年代 (Old Generation)。

- 新生代 (New Generation): 通常初次分配的对象都位于新生代中，该区域主要是存放内存较小并且生命周期较短的对象，比如局部变量。新生代会频繁执行内存回收 (GC)，回收采用“复制 – 清除”算法，将内存分为两块 (图中的 from 和 to)，运行时每次只使用其中的一块 (图中的 from)，另一块备用 (图中的 to)。当发生 GC 时，将当前使用的内存块中存活的对象拷贝到备用内存块中，然后清除当前使用内存块，最后，交换两块内存的角色。



- 老年代 (Old Generation): 在新生代的 GC 中“幸存”下来的对象，它们会被转移到老年代中。老年代存放生命力周期较长，内存较大的对象。老年代通常比新生代要大很多。老年代的 GC 回收采用“标记 – 清除”算法，分成标记和清理两个阶段。在标记阶段会触发停顿 (stop the world)，多线程并发的完成对垃圾对象的标记，降低标记阶段耗时。在清理阶段，由 GC 线程负责清理回收对象，和应用线程同时执行，不影响应用运行。



注：对老年代的描述可能并不准确，Flutter 并没有直接相关文档，这是作者在沟通和学习中自己的理解。Flutter 的技术本身也在不断的迭代升级中。读者可以参考，然后自己求证技术细节。

Image 内存初探

对图片的合理使用和优化是 UI 编程的重要部分，Flutter 提供了 Image Widget，我们可以方便的使用：

```
// 使用本地图片
new Image.asset("images/xxxx.jpg");

// 使用网络图片
new Image.network("https://xxxxxx");
```

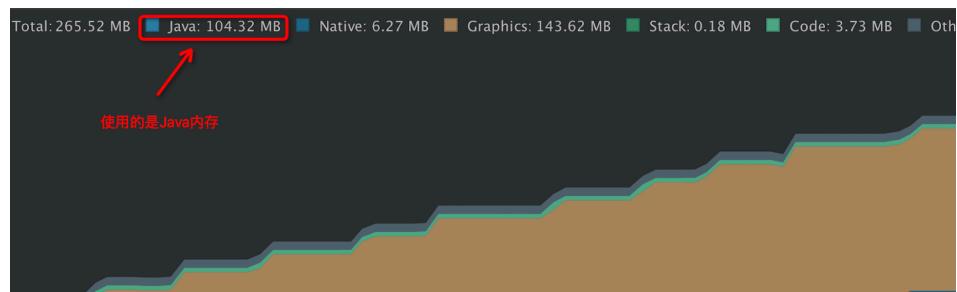
我们知道 Android 将内存分为 Java 虚拟机内存和 Native 内存，各大厂商都对 Java 虚拟机内存有一个上限限制，到达上限就会触发 OOM 异常，而对 Native 内存的使用没有太严格的限制，现在的手机内存都很大，一般有较大的 Native 内存富余。那么 Android 中 ImageView 使用的是 Java 虚拟机内存还是 Native 内存呢？

我们可以来做一个测试：在一个界面上，每点击一次，就在上面堆加一张图片。为了防止后面的图片完全覆盖前面的图片而出现优化的情况，每次都缩小几个像素，

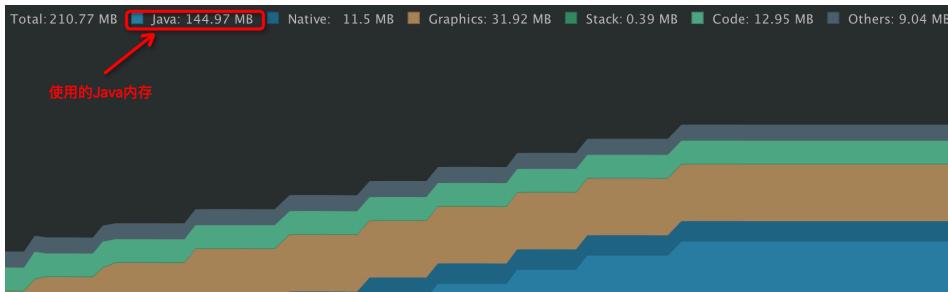
这样就不会出现完全覆盖。



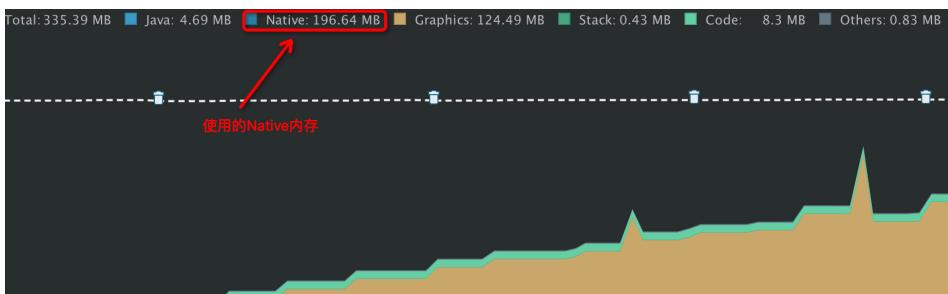
打开 Android Profiler，一张一张添加图片，观察内存数据。分别测试了 Android 的 6.0, 7.0 和 8.0 系统，结果如下：



Android 6.0 (Google Nexus5)



Android 7.0 (Meizu pro5)



Android 8.0 (Google pixel)

在测试中，随着图片一张张增加，Android 6.0 和 7.0 都是 Java 部分的内存在增长，而 Android 8.0 则是 Native 部分的内存在增长。由此有结论，Android 原生的 ImageView 在 6.0 和 7.0 版本中使用的 Java 虚拟机内存，而在 Android 8.0 中则使用的 Native 内存。

而 Flutter Image Widget 使用的是哪部分内存呢？我们用 Flutter 界面来做相同的测试。Flutter Engine 的 Debug 版本和 Release 版本存在很大的性能差异，所以我们测试最好使用 Release 版本，但是，Release 版本的 Apk 又不能使用 Android profiler 来观察内存，所以我们需要在 Debug 版本的 Apk 中打包一个 Release 版本的 Flutter Engine，可以修改 flutter tool 中的 flutter.gradle 来实现：

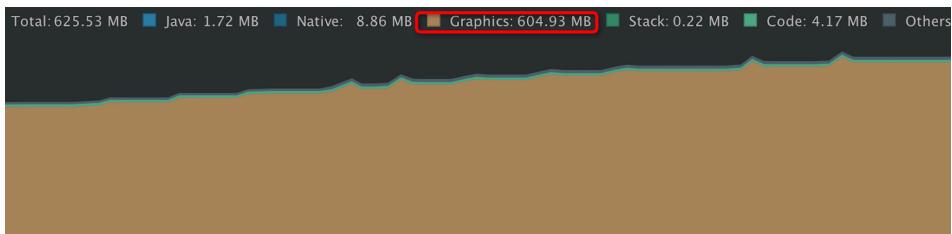
```
// 不做判断，强制改为打包 release 版本的 engine
private static String buildModeFor(BuildType) {
    // if (buildType.name == "profile") {
    //     return "profile"
}
```

```
// } else if (buildType.debuggable) {  
//     return "debug"  
// }  
return "release"  
}
```

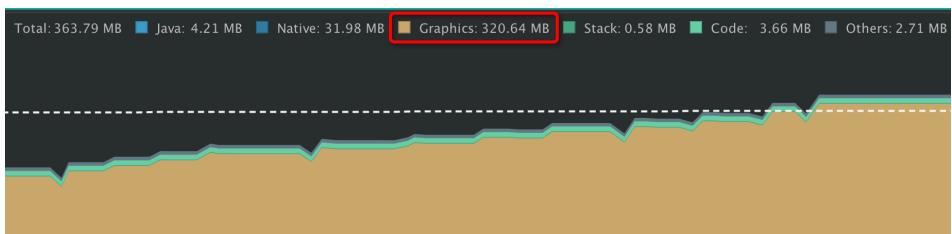
相同地，我们向 Flutter 界面中添加图片并用 Android Profiler 来观察内存，测试使用的 dart 代码：

```
class StackImageState extends State<StackImages> {  
    var images = <String>[];  
    var index = 0;  
  
    @override  
    Widget build(BuildContext context) {  
        var widgets = <Widget>[];  
  
        for (int i = 0; i <= index; i++) {  
            var pos = i - (i ~/ 103) * 103;  
            widgets.add(new Container(  
                child: new Image.asset("images/${pos}.jpg", fit: BoxFit.cover),  
                padding: new EdgeInsets.only(top: i * 2.0)));  
        }  
  
        widgets.add(new Center(  
            child: new GestureDetector(  
                child: new Container(  
                    child: new Text("添加图片 (${index})",  
                        style: new TextStyle(color: Colors.red)),  
                    color: Colors.green,  
                    padding: const EdgeInsets.all(8.0)),  
                onTap: () {  
                    setState(() {  
                        index++;  
                    });  
                }));  
        );  
        return new Stack(  
            children: widgets, alignment: AlignmentDirectional.topCenter);  
    }  
}
```

得到的结果是：



Android 6.0



Android 8.0

可以看到，Flutter Image 使用的内存既不属于 Java 虚拟机内存也不属于 Native 内存，而是 Graphics 内存（在 Meizu pro5 设备上也不属于 Graphics，事实上 Meizu pro5 设备不能归类 Flutter Image 所使用的内存），官方对 Graphics 内存的解释是：

Total: 126.21MB Java: 6.3MB Native: 71.48MB Graphics: 37.95MB Stack: 0.72MB Code: 7.54MB Others: 2.23MB Allocated: 106028

图 2. Memory Profiler 顶部的内存计数图例

内存计数中的类别如下所示：

- **Java**: 从 Java 或 Kotlin 代码分配的对象内存。

- **Native**: 从 C 或 C++ 代码分配的对象内存。

即使您的应用中不使用 C++，您也可能会看到此处使用的一些原生内存，因为 Android 框架使用原生内存代表您处理各种任务，如处理图像资源和其他图形时，即使您编写的代码采用 Java 或 Kotlin 语言。

- **Graphics**: 图形缓冲区队列向屏幕显示像素（包括 GL 表面、GL 纹理等等）所使用的内存。（请注意，这是与 CPU 共享的内存，不是 GPU 专用内存。）

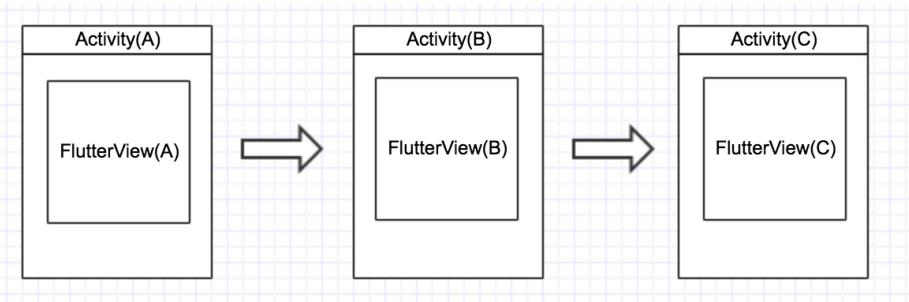
那么至少 Flutter Image 所使用的内存不会是 Java 虚拟机内存，这对不少 Android 设备都是一个好消息，这意味着使用 Flutter Image 没有 OOM 的风险，能够较好的利用 Native 内存。

使用 Image 的时候，建立一个内存缓存池是个好习惯，Flutter Framework 提供了一个 ImageCache 来缓存加载的图片，但它不同于 Android Lru Cache，不能精确的使用内存大小来设定缓存池容量，而是只能粗略的指定最大缓存图片张数。

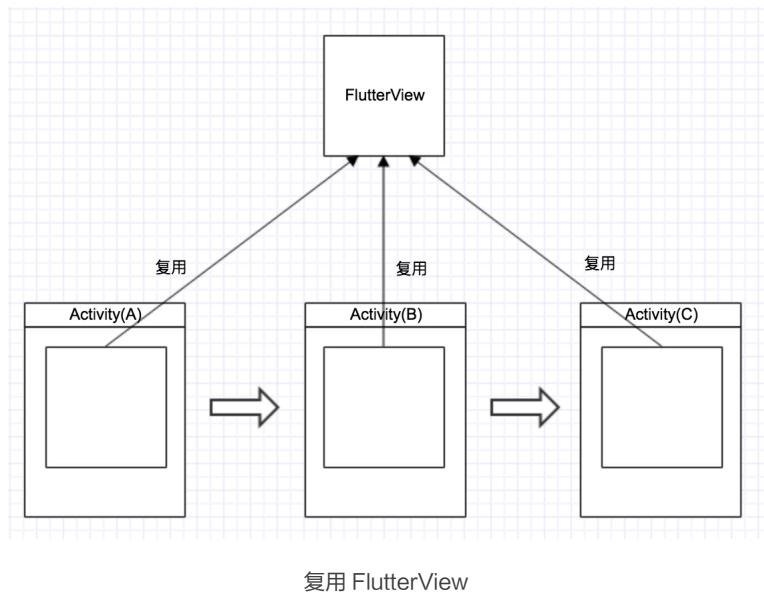
FlutterView 内存初探

下面这部分内容在新版本的 flutter(flutter 1.5+) 已经不太适用，但思路依然可以借鉴。新的 flutter 已经逐渐将 Flutter Engine 从 FlutterView 中独立出来，这非常有意义，可以让 Engine 脱离界面，我们可以直接去复用 Engine 而不受 View 的拘束，这样工程实现上会清晰很多。甚至可以用来执行一些后台的逻辑，比如消息收发等。在阅读下面的内容时可以想象把复用 FluterView 改成复用 FlutterEngine

Flutter 设计之初是想统一 Android 和 IOS 的界面编程，所以理想的基于 Flutter 的 apk 只需要提供一个 MainActivity 做入口即可，后面所有的页面跳转都在 FlutterView 中管理。但是，如果是一个已有规模的 app 接入 Flutter 开发，我们不可能将已有的 Activity 页面都用 Flutter 重新实现一遍，这时候就需要考虑本地页面和 Flutter 页面之间的跳转交互了。iOS 可以方便的管理页面栈，但是 Android 就很复杂 (Android 有任务栈机制，低内存 Activity 回收机制等)，所以通常我们还是使用 Activity 作为页面容器来展示 flutter 页面。这时有两种选择，可以每次启动一个 Activity 就启动一个新的 FlutterView，也可以启动 Activity 的时候复用已有的 FlutterView。



不复用 FlutterView

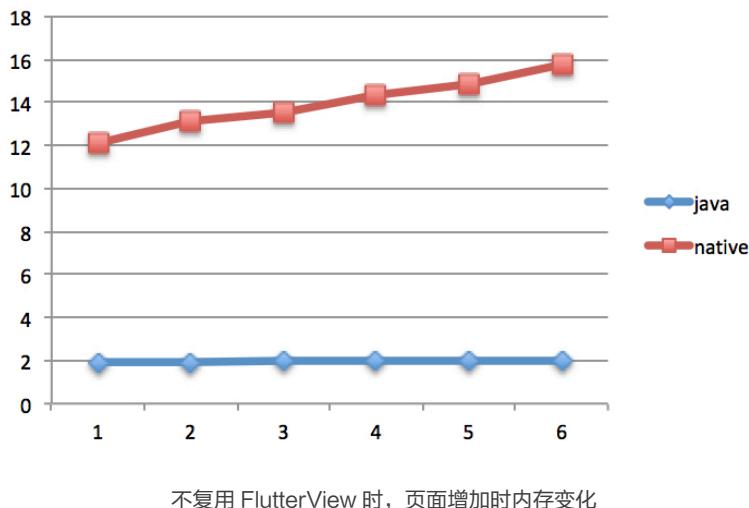


Flutter Framework 中 `FlutterView` 是绑定 `Activity` 使用的，要复用 `FlutterView` 就必须能够把 `FlutterView` 单独拎出来使用。所幸现在 `FlutterView` 和 `Activity` 耦合程度并不很深，最关键的地方是 `FlutterNativeView` 必须 attach 一个 `Activity`:

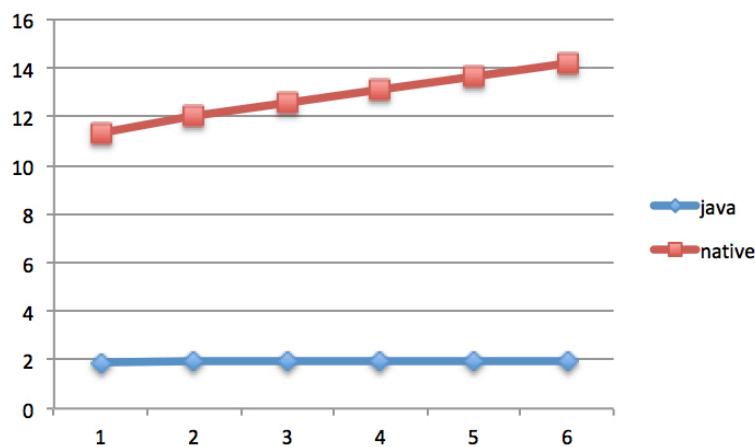
```
//attach 到当前 Activity
mNativeView.attachViewAndActivity(this, activity);
```

初始化 `FlutterView` 时必须传入一个 `Activity`，当其他 `Activity` 复用 `FlutterView` 时再调用该 `Attach` 方法即可。这里有个问题，就是 `FlutterView` 中必须保存一个 `Activity` 引用，这个一个内存泄露隐患，我们可以在 `FlutterView` `detach` 时候将 `MainActivity` 传入，因为通常整个 App 交互过程中 `MainActivity` 都是一直存在的，可以避免其他 `Activity` 泄露。

为了更好的权衡两种方法的利弊，我们先用空页面来测试一下当页面增加时内存的变化：



不复用 FlutterView 时，页面增加时内存变化

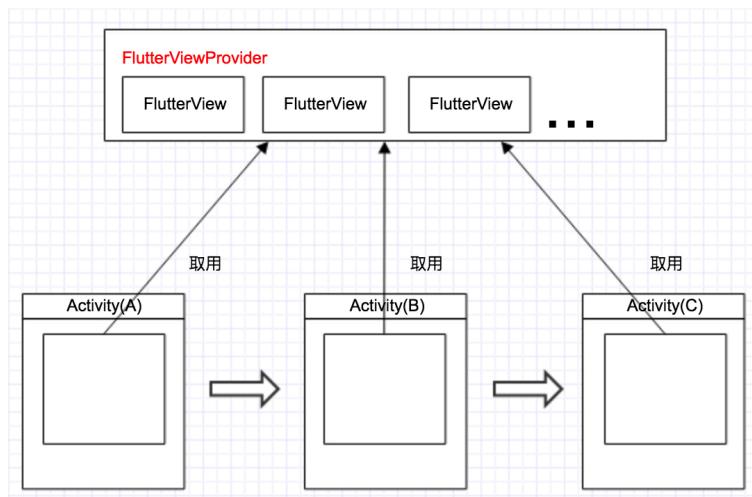


复用 FlutterView 时，页面增加时内存变化

不复用 FlutterView 时平均打开一个页面（空页面），Java 内存增长 0.02M，Native 内存增长 0.73M。复用 FlutterView 时平均打开一个页面（空页面），Java 内存增长 0.019M，Native 内存增长 0.65M。可见复用 FlutterView 在内存使用上是有优势的，但主要复用的还是 Native 部分的内存。复用 FlutterView 必然带来额外的一些复杂逻辑，有时候为了逻辑简单，后期维护上的方便，牺牲一些相对不太珍贵的 Native 内存也是值得的。

复用单个 FlutterView 有时会有些“意外”，比如当 Activity 切换时，就不得不将当前 FlutterView detach 掉给后面新建的 Activity 使用，当前界面就会空白闪动，有个想法是可以将当前界面截屏下来遮挡住后面的界面变化，这种方式有时会带来额外的适配问题。

FlutterView 复用与否不是绝对的，有时候可以使用一些综合性折中方案，比如，我们可以建立一个 FlutterViewProvider，里面维护 N 个可复用的 FlutterView，如图：



这样的好处是，可以在一定程度上的复用，又可以避免只有一个 FlutterView 出现的一些尴尬问题。

在新版本的 flutter/flutter 1.5+ 已经将 FlutterEngine 从 FlutterView 中分离的前提下，FlutterEngine 可以早于 FlutterView 启动，将一部分耗时的逻辑预先执行，这样，当 FlutterView 启动 Attach Engine 时界面可以较快的渲染出来。是一种更合理的优化方法

FlutterView 的首帧渲染耗时较高，在 Debug 版本有明显感受，大概会黑屏 2 秒，release 版本会好很多。但我们观察 Cpu 曲线，发现还是一个较为耗时的过程。有一种体验优化的思路是，我们可以预先让将要使用的 FlutterView 加载好首帧，这样，在真正使用的时候就很快了，可以先建立一个只有 1 个像素的窗口，在这个窗口里面完成 FlutterView 首帧渲染，代码如下：

```
final WindowManager wm = mFakeActivity.getWindowManager();
final FrameLayout root = new FrameLayout(mFakeActivity);

//一个像素足矣
FrameLayout.LayoutParams params = new FrameLayout.LayoutParams(1, 1);
root.addView(flutterView,params);
WindowManager.LayoutParams wlp = new WindowManager.LayoutParams();
wlp.width = 1;
wlp.height = 1;
wlp.flags |= WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE;
wlp.flags |= WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE;
wm.addView(root,wlp);

final FlutterView.FirstFrameListener[] listenerRef = new FlutterView.
FirstFrameListener[1];
listenerRef[0] = new FlutterView.FirstFrameListener() {
    @Override
    public void onFirstFrame() {
        //首帧渲染完后取消窗口
        wm.removeView(root);
        flutterView.removeFirstFrameListener(listenerRef[0]);
    }
};

flutterView.addFirstFrameListener(listenerRef[0]);
String appBundlePath = FlutterMain.findAppBundlePath(mFakeActivity.
getApplicationContext());
flutterView.runFromBundle(appBundlePath, null, "main", true);
```

以上就是闲鱼团队在 Flutter 的应用过程中的一些实践，希望有更多的新技术尝试和技术挑战的同学，请在闲鱼公众号留言，联系我们！

第四章 Flutter 深入进阶教程

I 一章节教会你如何低成本实现 Flutter 富文本

作者：闲鱼技术 – 玄川

背景

闲鱼是国内最早使用 Flutter 的团队，作为一个电商 App 商品详情页是非常重要场景，其中最主要的技术能力是文字混排。



我们面对文本类的需求是复杂而且多变，然而 Flutter 历史的几个版本，Text 只能显示简单样式文本，它只有包含一些控制文本样式显示的属性，而通过 TextSpan 连接实现的 RichText 也只能显示多种文本样式（例如：一个基础文本片段和一个链接片段），这些远远达不到设计需要的能力。被产品和设计总监为啥别人别的平台能做，Flutter 为何做不了，不管，必须支持。

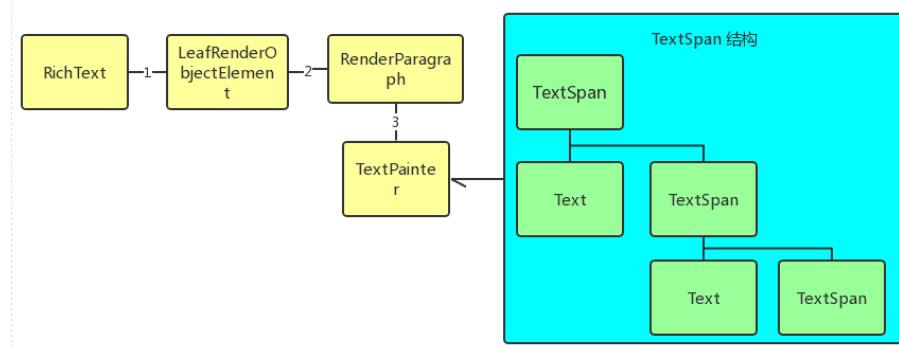


因此，需要开发一个能力更强的文字混排组件就变得迫在眉睫。

富文本的原理

再讲文字混批组件设计实现前，先来讲讲系统 RichText 的富文本的原理。

- 创建过程



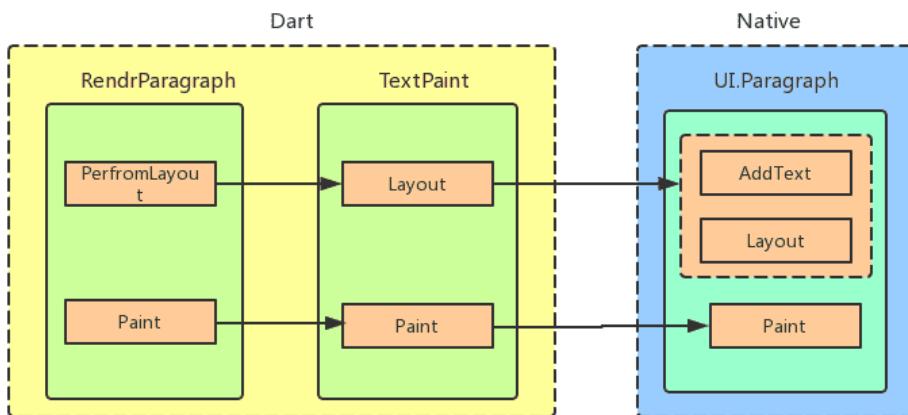
创建 RichText 节点的时候其实会创建以下几个对象：

1. 先创建 LeafRenderObjectElement 实例。
2. ComponentElement 方法当中会调用 RichText 实例的 CreateRenderObject 方法，生成 RenderParagraph 实例。

3. RenderParagraph 会创建 TextPainter 负责其就计算宽高和绘制文本到 Canvas 的代理类，同时 TextPainter 持有 TextSpan 文本结构。

RenderParagraph 实例最后会将自身登记到渲染模块的 Dirty Nodes 当中去，渲染模块会遍历 Dirty Nodes 将进入 RenderParagraph 渲染环节。

- 渲染过程



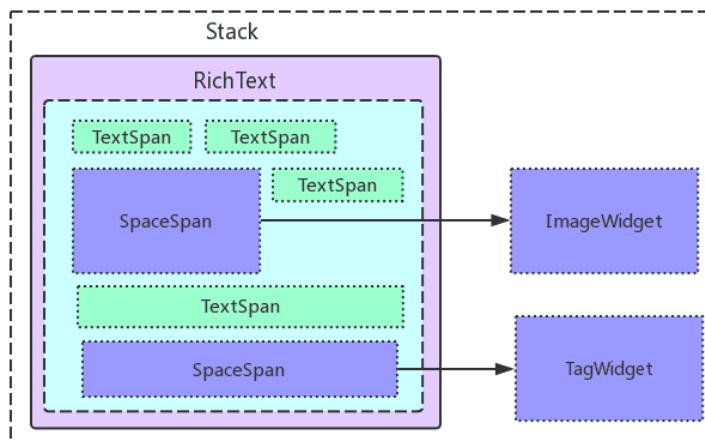
`RenderParagraph` 方法当中封装的是将文本绘制到 canvas 上面的逻辑，主要是用了一个叫做 `TextPainter` 的模块，其调用过程遵循 `RenderObject` 调用。

1. `PerfromLayout` 过程通过调用 `TextPaint` 的 `Layout`，在期过程中通过 `TextSpan` 结构树，依次通过 `AddText` 添加各个阶段的文本，最后通过 `Paragraph` 的 `Layout` 计算文本高度。
2. `Paint` 过程，先绘制 `clipRect`，接着通过 `TextPaint` 的 `Paint` 函数调用，`Paragraph` 的 `Paint` 绘制文本，最后绘制 `drawRect`。

设计思路

通过 `RichText` 的文本绘制原理，我们不难发现 `TextSpan` 记录了各段文本信息，`TextPaint` 通过记录的信息调用 Native 接口计算宽高，以及将文本绘制到 canvas 上面。传统的方案实现复杂的混排，会通过 HTML 去做一个 `WebView` 的富文本，

使用 WebView 在性能上自然不及原生实现，出于性能的考虑，我们设想通过通过原生的方式去实现图文混排。一开始的方案是设计几种特殊的 Span(例如: ImageSpan, EmojiSpan 等)，通过 Span 记录的信息，在 TextPaint 的 Layout 重新根据各种类型重新计算布局，在 Paint 过程再分别绘制特殊的 Widget，然而这种方案对上面几个涉及的类封装破坏的特别大，需要将 RichText、RenderParagraph 源码 Copy 出来重新修改。最后设想是后可以通过特殊的文字先占位置，(例如: 空字符串)，然后在这个文字的位置上面把特殊的 Span 分别独立移动到上面。



然而上面这种方案会带来两个难点：

- 难点一：如何在文本中先占位，并且能制定任意想要的宽高。

通过 Google 发现 200B 字符代表 ZERO WIDTH SPACE (宽带为 0 的空白)，结合对 TextPainter 测试，我们发现 layout 出来的 Width 总是 0，fontSize 只决定了高度，结合 TextStyle 里面的 letterSpacing

```
/// The amount of space (in logical pixels) to add between each letter
/// A negative value can be used to bring the letters closer.
final double letterSpacing;
```

这样我们就能任意的控制这个特殊文字的宽高度。

- 难点二：如何将特殊的 Span 移动到位置上面。

通过上面的测试不难发现，特殊的 Span 其实还是独立 Widget 和 RichText 并不融合。所以我们需要知道当前 widget 相对 RichText 空间的相对位置，并且结合 Stack 将其融合。结合 TextPaint 里面的 getOffsetForCaret 方法

```
// Returns the offset at which to paint the caret.  
///  
/// Valid only after [layout] has been called.  
Offset getOffsetForCaret(TextPosition position, Rect caretPrototype)
```

可以天然的获取到当前占位符相对位置。

实现方案



关键部分代码实现如下：

- 统一的占位 SpaceSpan

```
SpaceSpan({      this.contentWidth,      this.contentHeight,      this.  
widgetChild,      GestureRecognizer recognizer,    }) : super(  
style: TextStyle(          color: Colors.transparent,  
letterSpacing: contentWidth,          height: 1.0,  
fontSize: contentHeight),          text: '\u200B',  
recognizer: recognizer);
```

- SpaceSpan 相对位置获取

```
``` for (TextSpan textSpan in widget.text.children) { if (textSpan is SpaceSpan) { final SpaceSpan targetSpan = textSpan; Offset offsetForCaret = painter.getOffsetForCaret( TextPosition(offset: textIndex), Rect.fromLTRB( 0.0, targetSpan.contentHeight, targetSpan.contentWidth, 0.0), ); ..... } textIndex += textSpan.toPlainText().length; }
```

```

- RichText 和 SpaceSpan 融合

```
``` Stack( children: [ RichText(), Positioned(left: position.dx, top: position.dy, child: child), ], );
```

```

效果

先上图看看效果：



这种方案的优点是任意 Widget 可通过 SpaceSpan 和 RichText 进行组合，无论是图片、自定义标签、甚至是按钮都可以融合进来，同时对 RichText 本身封装性破坏较小。

未来

上面只是富文本显示的部分，依然存在着很多局限，还有较多需要优化的点，目前通过 SpaceSpan 控件，必需要指定宽高，另外对于文本选择、自定义文字背景这些都是无法支持，其次对富文本编辑器的支持，可以使其编辑文字时，让图片、货币格式化等控件输入等。

揭秘！一个高准确率的 Flutter 埋点框架如何设计

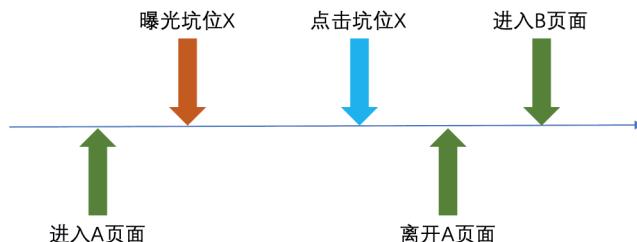
作者：闲鱼技术 - 兰昊

背景

用户行为埋点是用来记录用户在操作时的一系列行为，也是业务做判断的核心数据依据，如果缺失或者不准确将会给业务带来不可恢复的损失。闲鱼将业务代码从 Native 迁移到 Flutter 上过程中，发现原先 Native 体系上的埋点方案无法应用在 Flutter 体系之上。而如果我们只把业务功能迁移过来就上线，对业务是极其不负责任的。因此，经过不断探索，我们沉淀了一套 Flutter 上的高准确率的用户行为埋点方案。

用户行为埋点定义

先来讲讲在我们这里是如何定义用户行为埋点的。在如下用户时间轴上，用户进入 A 页面后，看到了按钮 X，然后点击了这个按钮，随即打开了新的页面 B。



这个时间轴上有如下 5 个埋点事件发生：

- 进入 A 页面。A 页面首帧渲染完毕，并获得了焦点。
- 曝光坑位 X。按钮 X 处于手机屏幕内，且停留一段时间，让用户可见可触摸。
- 点击坑位 X。用户对按钮 X 的内容很感兴趣，于是点击了它。按钮 X 响应点击，然后需要打开一个新页面。
- 离开 A 页面。A 页面失去焦点。
- 进入 B 页面。B 页面首帧渲染完毕，并获得焦点。

在这里，打埋点最重要的是时机，即在什么时机下的事件中触发什么埋点，下面来看看闲鱼在 Flutter 上的实现方案。

实现方案

进入 / 离开页面

在 Native 原生开发中，Android 端是监听 Activity 的 onResume 和 onPause 事件来做为页面的进入和离开事件，同理 iOS 端是监听 UIViewController 的 viewDidAppear 和 viewDidDisappear 事件来做为页面的进入和离开事件。同时整个页面栈是由 Android 和 iOS 操作系统来维护。

在 Flutter 中，Android 和 iOS 端分别是用 FlutterActivity 和 FlutterViewController 来做为容器承载 Flutter 的页面，通过这个容器可以在一个 Native 的页面内 (FlutterActivity/FlutterViewController) 来进行 Flutter 原生页面的切换。即在 Flutter 自己维护了一个 Flutter 页面的页面栈。这样，原来我们最熟悉的那套在 Native 原生上方案在 Flutter 上无法直接运作起来。

针对这个问题，可能很多人会想到去注册监听 Flutter 的 NavigatorObserver，这样就知道 Flutter 页面的进栈 (push) 和出栈 (pop) 事件。但是这会有两个问题：

- 假设 A、B 两个页面先后进栈 (A enter → A leave → B enter)。然后 B 页面返回退出 (B leave)，此时 A 页面重新可见，但是此时是收不到 A 页面 push (A enter) 的事件。
- 假设在 A 页面弹出一个 Dialog 或者 BottomSheet，而这两类也会走 push 操作，但实际上 A 页面并未离开。

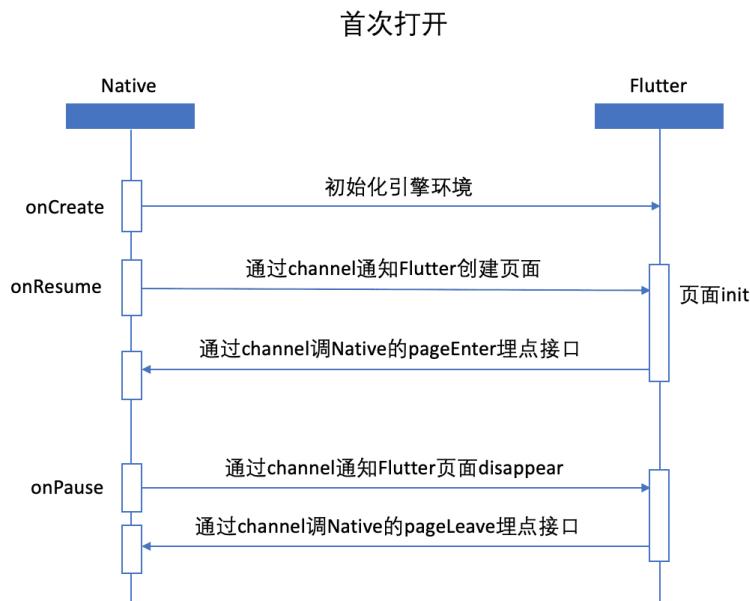
好在 Flutter 的页面栈不像 Android Native 的页面栈那么复杂，所以针对第一个问题，我们可以来维护一个和页面栈匹配的索引列表。当收到 A 页面的 push 事件时，往队列里塞一个 A 的索引。当收到 B 页面的 push 事件时，检测列表内是否有页面，如有，则对列表最后一个页面执行离开页面事件记录，然后再对 B 页面执行进入页面事件记录，接着往队列里塞一个 B 的索引。当收到 B 页面的 pop 事件时，先

对 B 页面执行离开页面事件记录，然后对队列里存在的最后一个索引对应的页面（假设为 A）进行判断是否在栈顶 (`ModalRoute.of(context).isCurrent`)，如果是，则对 A 页面执行进入页面事件记录。

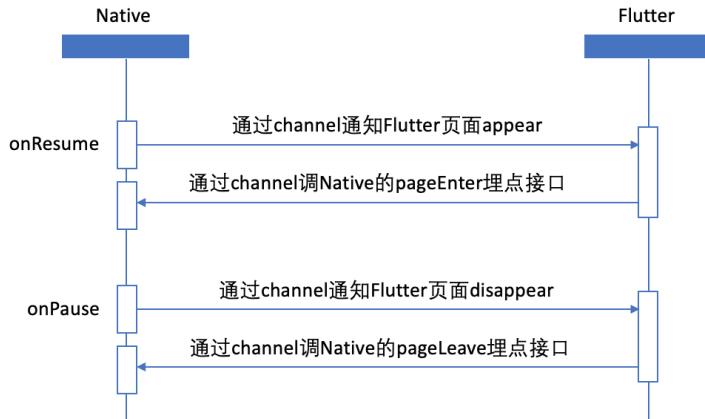
针对第二个问题，`Route` 类内有个成员变量 `overlayEntries`，可以获取当前 `Route` 对应的所有图层 `OverlayEntry`，在 `OverlayEntry` 对象中有个成员变量 `opaque` 可以判断当前这个图层是否全屏覆盖，从而可以排除 `Dialog` 和 `BottomSheet` 这种类型。再结合问题 1，还需要在上述方案中加上对 `push` 进来的新页面来做判断是否为一个有效页面。如果是有效页面，才对索引列表中前一个页面做离开页面事件，且将有效页面加到索引列表中。如果不是有效页面，则不操作索引列表。

以上并不是闲鱼的方案，只是笔者给出的一个建议。因为闲鱼 APP 在一开始落地 Flutter 框架时，就没有使用 Flutter 原生的页面栈管理方案，而是采用了 Native+Flutter 混合开发的方案。具体可参考前面的一篇文章 [《已开源 | 码上用它开始 Flutter 混合开发——FlutterBoost》](#)。因此接下来也是基于此来阐述闲鱼的方案。

闲鱼的方案如下（以 Android 为例，iOS 同理）：



非首次打开



注：首次打开指的是基于混合栈新打开一个页面，非首次打开指的是通过回退页面的方式，在后台的页面再次到前台可见。

看似我们将何时去触发进入 / 离开页面事件的判断交给 Flutter 侧，实际上依然跟 Native 侧的页面栈管理保持了一致，将原先在 Native 侧做打埋点的时机告知 Flutter 侧，然后 Flutter 侧再立刻通过 channel 来调用 Native 侧的打埋点方法。那么可能会有人问，为什么这么绕，不全部交给 Native 侧去直接管理呢？交给 Native 侧去直接管理这样做针对非首次打开这个场景是合适的，但是对首次打开这个场景却是不合适的。因为在首次打开这个场景下，onResume 时 Flutter 页面尚未初始化，此时还不知道页面信息，因此也就不知道进入了什么页面，所以需要在 Flutter 页面初始化 (init) 时再回过来调 Native 侧的进入页面埋点接口。为了避免开发人员去关注是否为首次打开 Flutter 页面，因此我们统一在 Flutter 侧来直接触发进入 / 离开页面事件。

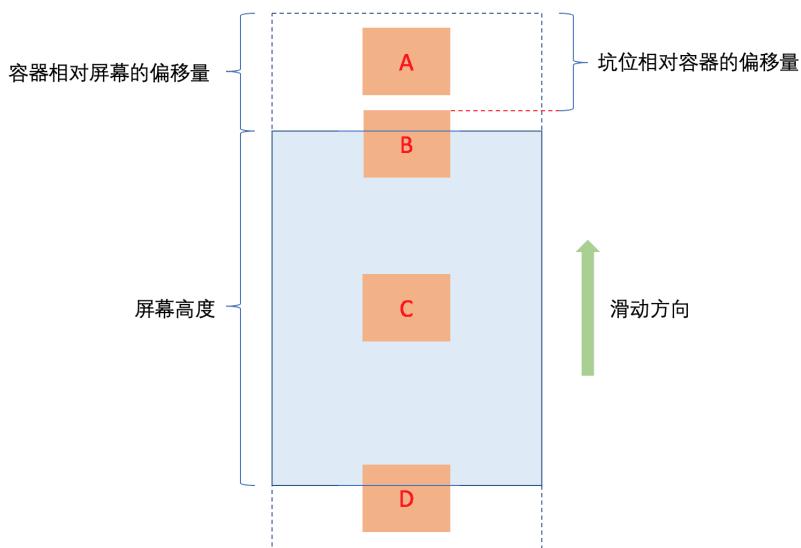
曝光坑位

先讲下曝光坑位在我们这里的定义，我们认为图片和文本是有曝光意义的，其他用户看不见的是没有曝光意义的，在此之上，当一个坑位同时满足以下两点时才会被认为是一次有效曝光：

- 坑位在屏幕可见区域中的面积大于等于坑位整体面积的一半。
- 坑位在屏幕可见区域中停留超过 500ms。

基于此定义，我们可以很快得出如下图所示的场景，在一个可以滚动的页面上有 A、B、C、D 共 4 个坑位。其中：

- 坑位 A 已经滑出了屏幕可见区域，即 invisible；
- 坑位 B 即将向上从屏幕中可见区域滑出，即 visible->invisible；
- 坑位 C 还在屏幕中央可视区域内，即 visible；
- 坑位 D 即将滑入屏幕中可见区域， invisible->visible；



那么我们的问题就是如何算出坑位在屏幕内曝光面积的比例。要算出这个值，需要知道以下几个数值：

- 容器相对屏幕的偏移量
- 坑位相对容器的偏移量
- 坑位的位置和宽高
- 容器的位置和宽高

其中坑位和容器的宽和高很容易获取和计算，这里就不再累述。

获取容器相对屏幕的偏移量

```
// 监听容器滚动，得到容器的偏移量
double _scrollContainerOffset = scrollNotification.metrics.pixels;
```

获取坑位相对容器的偏移量

```
// 曝光坑位 Widget 的 context
final RenderObject childRenderObject = context.findRenderObject();
final RenderAbstractViewport viewport = RenderAbstractViewport.of(childRenderObject);
if (viewport == null) {
    return;
}
if (!childRenderObject.attached) {
    return;
}
// 曝光坑位在容器内的偏移量
final RevealedOffset offsetToRevealTop = viewport.getOffsetToReveal
(childRenderObject, 0.0);
```

逻辑判断

```
if (当前坑位是 invisible && 曝光比例 >= 0.5) {
    记录当前坑位是 visible 状态
    记录出现时间
} else if (当前坑位是 visible && 曝光比例 < 0.5) {
    记录当前坑位是 invisible 状态
    if (当前时间 - 出现时间 > 500ms) {
        调用曝光埋点接口
    }
}
```

点击坑位

点击坑位埋点没什么难点，很容易就可以想到下面的方案：



效果

经过多轮迭代和优化，目前线上 Flutter 页面的埋点准确率已经达到 100%，有力地支持了业务的分析和判断。同时这套方案让业务同学在做开发时，对于页面进入 / 离开、曝光坑位可以做到无感知，即不用关心何时去触发，做到了简单易用和无侵入性。

展望

此外，针对页面进入 / 离开这个场景，由于闲鱼是基于 Flutter Boost 混合栈的方案，因此我们的解决方案还不够通用。不过未来随着闲鱼上的 Flutter 页面越来越多，我们后续也会去实现基于 Flutter 原生的方案。

在闲鱼做数据驱动业务是一件非常重要且有意义事，而埋点直接影响着数据采集，埋点的丢失和错误将会让我们在大海上航行时失去灯塔的指引。在这里大家都习惯着用数据来指导工作方向，试验 -> 取数据分析 -> 调整实验 -> 再取数据分析 -> 再调整实验。如此循环着，只为找到最适合用户的那一个设计。

万万没想到 Flutter 这样外接纹理

作者：闲鱼技术 – 炉军

前言

记得在 13 年做群视频通话的时候，多路视频渲染成为了端上一个非常大的性能瓶颈。原因是每一路画面的高速上屏（PresentRenderBuffer or SwapBuffer 就是讲渲染缓冲区的渲染结果呈现到屏幕上）操作，消耗了非常多的 CPU 和 GPU 资源。

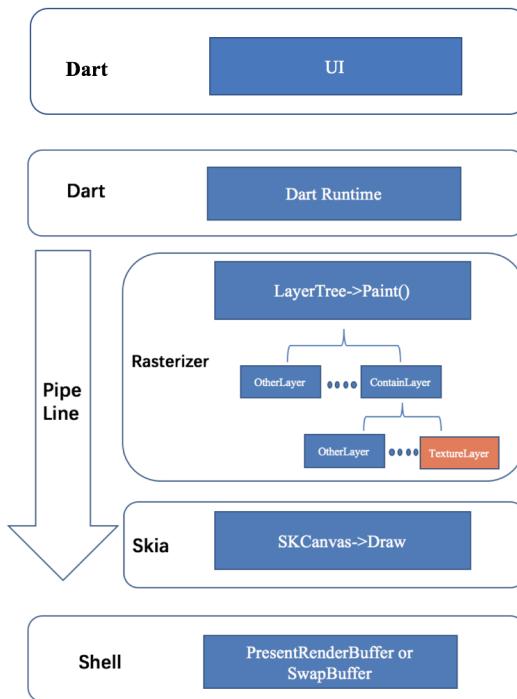
那时候的解法是将绘制和上屏进行分离，将多路画面抽象到一个绘制树中，对其进行遍历绘制，绘制完成以后统一做上屏操作，并且每一路画面不再单独触发上屏，而是统一由 Vsync 信号触发，这样极大的节约了性能开销。

那时候甚至想过将整个 UI 界面都由 OpenGL 进行渲染，这样还可以进一步减少界面内诸如：声音频谱，呼吸效果等动画的性能开销。但由于各种条件限制，最终没有去践行这个想法。

万万没想到的是这种全界面 OpenGL 渲染思路还可以拿来做跨平台。

Flutter 渲染框架

下图为 Flutter 的一个简单的渲染框架：



Layer Tree: 这个是 dart runtime 输出的一个树状数据结构，树上的每一个叶子节点，代表了一个界面元素 (Button, Image 等等)。

Skia: 这个是谷歌的一个跨平台渲染框架，从目前 IOS 和 android 来看，SKIA 底层最终都是调用 OpenGL 绘制。Vulkan 支持还不太好，Metal 还不支持。

Shell: 这里的 Shell 特指平台特性 (Platform) 的那一部分，包含 IOS 和 Android 平台相关的实现，包括 EAGLContext 管理、上屏的操作以及后面将会重点介绍的外接纹理实现等等。

从图中可以看出，当 Runtime 完成 Layertree 以后，在管线中会遍历 Layertree 的每一个叶子节点，每一个叶子节点最终会调用 Skia 引擎完成界面元素的绘制，在遍历完成后，在调用 glPresentRenderBuffer (IOS) 或者 glSwapBuffer(Android) 按完成上屏操作。

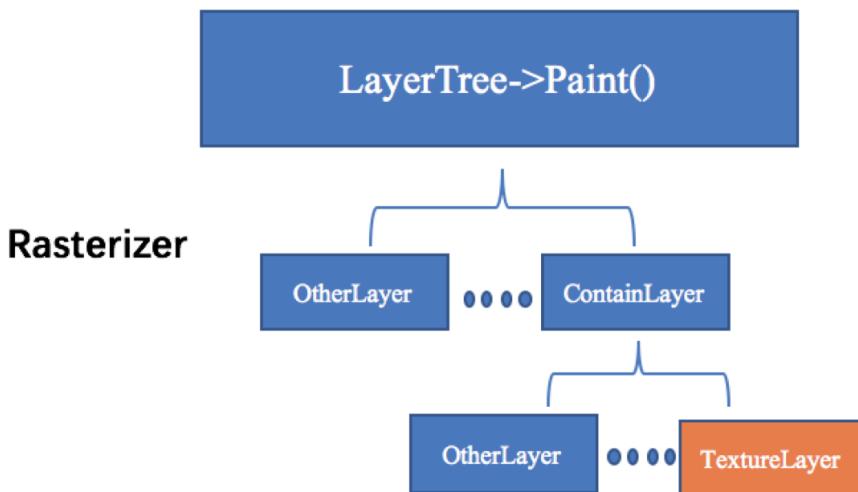
基于这个基本原理，Flutter 在 Native 和 Flutter Engine 上实现了 UI 的隔离，书写 UI 代码时不用再关心平台实现从而实现了跨平台。

问题

正所谓凡事有利必有弊，Flutter 在与 Native 隔离的同时，也在 Flutter Engine 和 Native 之间竖立了一座大山，Flutter 想要获取一些 Native 侧的高内存占用图像（摄像头帧、视频帧、相册图片等等）会变得困难重重。传统的如 RN，Weex 等通过桥接 NativeAPI 可以直接获取这些数据，但是 Flutter 从基本原理上就决定了无法直接获取到这些数据，而 Flutter 定义的 channel 机制，从本质上说是提供了一个消息传送机制，用于图像等数据的传输必然引起内存和 CPU 的巨大消耗。

解法

为此，Flutter 提供了一种特殊的机制：外接纹理 (ps: 纹理 Texture 可以理解为 GPU 内代表图像数据的一个对象)



上图是前文提到的 LayerTree 的一个简单架构图，每一个叶子节点代表了 dart 代码排版的一个控件，可以看到最后有一个 TextureLayer 节点，这个节点对应的是 Flutter 里的 Texture 控件 (ps. 这里的 Texture 和 GPU 的 Texture 不一样，这个是 Flutter 的控件)。当在 Flutter 里创建出一个 Texture 控件时，代表的是在这个控件上显示的数据，需要由 Native 提供。

以下是 IOS 端的 TextureLayer 节点的最终绘制代码 (android 类似，但是纹理获取方式略有不同)，整体过程可以分为三步：

1. 调用 external_texture copyPixelBuffer，获取 CVPixelBuffer
2. CVOpenGLTextureCacheCreateTextureFromImage 创建 OpenGL 的 Texture(这个是真的 Texture)
3. 将 OpenGL Texture 封装成 SKImage，调用 Skia 的 DrawImage 完成绘制。

```
void IOSExternalTextureGL::Paint(SkCanvas& canvas, const SkRect& bounds) {
    if (!cache_ref_) {
        CVOpenGLTextureCacheRef cache;
        CVReturn err = CVOpenGLTextureCacheCreate(kCFAllocatorDefault, NULL,
                                                [EAGLContext currentContext],
                                                NULL, &cache);
        if (err == noErr) {
            cache_ref_.Reset(cache);
        } else {
            FXL_LOG(WARNING) << "Failed to create GLES texture cache: " << err;
            return;
        }
    }
    fml::CFRef<CVPixelBufferRef> bufferRef;
    bufferRef.Reset([external_texture_ copyPixelBuffer]);
    if (bufferRef != nullptr) {
        CVOpenGLTextureRef texture;
        CVReturn err = CVOpenGLTextureCacheCreateTextureFromImage(
            kCFAllocatorDefault, cache_ref_, bufferRef, nullptr, GL_TEXTURE_2D,
            GL_RGBA,
            static_cast<int>(CVPixelBufferGetWidth(bufferRef)),
            static_cast<int>(CVPixelBufferGetHeight(bufferRef)), GL_BGRA, GL_
            UNSIGNED_BYTE, 0, &texture);
        texture_ref_.Reset(texture);
        if (err != noErr) {
            FXL_LOG(WARNING) << "Could not create texture from pixel buffer: " <<
            err;
            return;
        }
    }
    if (!texture_ref_) {
        return;
    }
    GrGLTextureInfo textureInfo = {CVOpenGLTextureGetTarget(texture_
ref_), CVOpenGLTextureGetName(texture_ref_), GL_RGBA8_OES};
}
```

```

    GrBackendTexture backendTexture(bounds.width(), bounds.height(),
GrMipMapped::kNo, textureInfo);
    sk_sp<SkImage> image =
        SkImage::MakeFromTexture(canvas.getGrContext(), backendTexture,
kTopLeft_GrSurfaceOrigin,
                                kRGBA_8888_SkColorType, kPremul_SkAlphaType, nullptr);

    if (image) {
        canvas.drawImage(image, bounds.x(), bounds.y());
    }
}

```

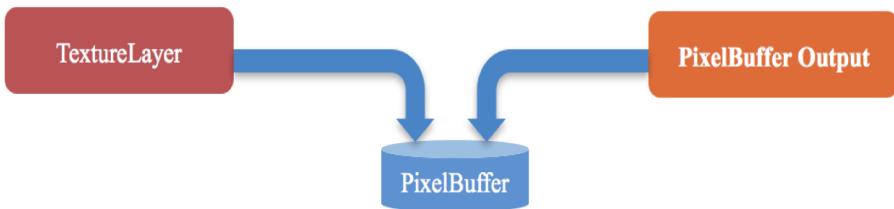
最核心的在于这个 `external_texture_` 对象，它是哪里来的呢？

```

void PlatformViewIOS::RegisterExternalTexture(int64_t texture_id, NSObject
<FlutterTexture*>*texture) {
    RegisterTexture(std::make_shared<IOSEExternalTextureGL>(texture_id, texture));
}

```

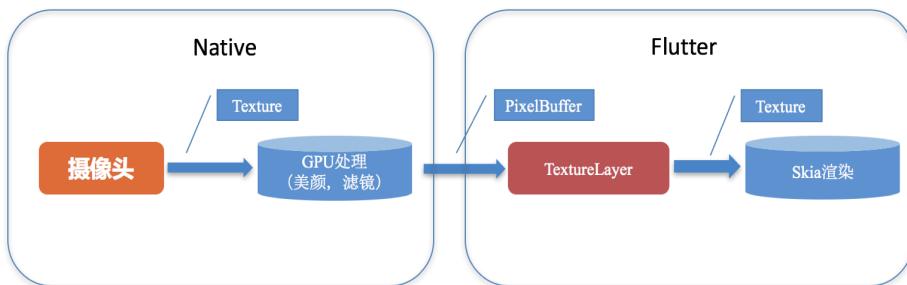
可以看到，当 Native 侧调用 `RegisterExternalTexture` 前，需要创建一个实现了 `FlutterTexture` 这个 protocol 的对象，而这个对象最终就是赋值给这个 `external_texture_`。这个 `external_texture_` 就是 Flutter 和 Native 之间的一座桥梁，在渲染时可以通过他源源不断的获取到当前所要展示的图像数据。



如图，通过外接纹理的方式，实际上 Flutter 和 Native 传输的数据载体就是 `PixelBuffer`，Native 端的数据源（摄像头、播放器等）将数据写入 `PixelBuffer`，Flutter 拿到 `PixelBuffer` 以后转成 `OpenGL ES Texture`，交由 `Skia` 绘制。

至此，Flutter 就可以容易的绘制出一切 Native 端想要绘制的数据，除了摄像头播放器等动态图像数据，诸如图片的展示也提供了 `Image` 控件之外的另一种可能（尤其对于 Native 端已经有大型图片加载库诸如 `SDWebImage` 等，如果要在 Flutter

端用 dart 写一份也是非常耗时耗力的)。优化 上述的整套流程，看似完美解决了 Flutter 展示 Native 端大数据的问题，但是许多现实情况是这样：



如图工程实践中视频图像数据的处理，为了性能考虑，通常都会在 Native 端使用 GPU 处理，而 Flutter 端定义的接口为 copyPixelBuffer，所以整个数据流程就要经过：GPU->CPU->GPU 的流程。而熟悉 GPU 处理的同学应该都知道，CPU 和 GPU 的内存交换是所有操作里面最耗时的操作，一来一回，通常消耗的时间，比整个管道处理的时间都要长。

既然 Skia 渲染的引擎需要的是 GPU Texture，而 Native 数据处理输出的就是 GPU Texture，那能不能直接就用这个 Texture 呢？答案是肯定的，但是有个条件：EAGLContext 的资源共享 (这里的 Context，也就是上下文，用来管理当前 GL 环境，可以保证不同环境下的资源的隔离)。

这里我们首先需要介绍下 Flutter 的线程结构：



如图所示，Flutter 通常情况下会创建 4 个 Runner，这里的 TaskRunner 类似于 IOS 的 GCD，是以队列的方式执行任务的一种机制，通常情况下 (一个 Runner 会对应一个线程，而 Platform Runner 会在跑在主线程)，这里和本文相关的有三个 Runner: GPU Runner、IORunner、Platform Runner。

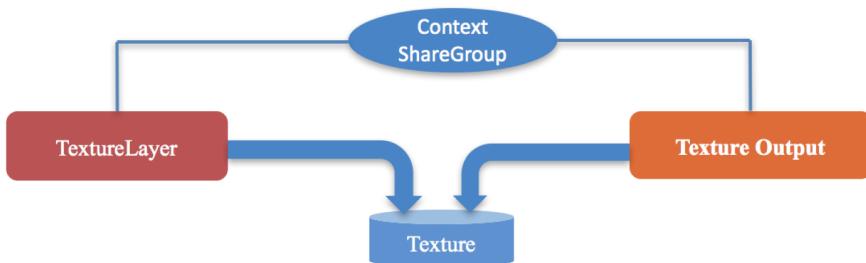
GPU Runner：负责 GPU 的渲染相关操作。

IO Runner: 负责资源的加载操作。

Platform Runner: 运行在 main thread 上, 负责所有 Native 与 Flutter Engine 的交互。

通常情况下一个使用 OpenGL 的 APP 线程设计都会有一个线程负责加载资源(图片到纹理), 一个线程负责渲染的方式。但是经常会发现为了能够让加载线程创建出来的纹理, 能够在渲染线程使用, 两个线程会共用一个 EAGLContext。但是从规范上来说这样使用是不安全的, 多线程访问同一对象加锁的话不可避免会影响性能, 代码处理不好甚至会引起死锁。因此 Flutter 在 EAGLContext 的使用上使用了另一种机制: **两个线程各自使用自己的 EAGLContext, 彼此通过 ShareGroup (android 为 shareContext) 来共享纹理数据。**(这里需要提一下的是: 虽然两个 Context 的使用者分别是 GPU 和 IO Runner, 但是现有 Flutter 的逻辑下两个 Context 都是在 Platform Runner 下创建的, 这里不知道是 Flutter 是出于什么考虑, 但是因为这个设计给我们带来很大的困扰, 后面会说到。)

对于 Native 侧使用 OpenGL 的模块, 也会在自己的线程下面创建出自己线程对应的 Context, 为了能够让这个 Context 下创建出来的 Texture, 能够输送给 Flutter 端, 并交由 Skia 完成绘制, 我们在 Flutter 创建内部的两个 Context 时, 将他们的 ShareGroup 透出, 然后在 Native 侧保存好这个 ShareGroup, **当 Native 创建 Context 时, 都会使用这个 ShareGroup 进行创建。**这样就实现了 Native 和 Flutter 之间的纹理共享。



通过这种方式来做 external_texture 有两个好处:

第一: 节省 CPU 时间, 从我们测试上看, android 机型上一帧 720P 的 RGBA

格式的视频，从 GPU 读取到 CPU 大概需要 5ms 左右，从 CPU 在送到 GPU 又需要 5ms 左右，哪怕引入了 PBO，也还是有 5ms 左右的耗时，这对于高帧率场景显然是不能接受的。

第二：节省 CPU 内存，显而易见数据都在 GPU 中传递，对于图片场景尤其适用（因为可能同一时间会有很多图片需要展示）。

后语

至此，我们介绍完了 Flutter 外接纹理的基本原理，以及优化策略。但是可能大家会有疑惑，既然直接用 Texture 作为外接纹理这么好，为什么谷歌要用 Pixelbuffer？这里又回到了那个命题，凡事有利必有弊，使用 Texture，必然需要将 ShareGroup 透出，也就是相当于将 Flutter 的 GL 环境开放了，如果外部的 OpenGL 操作不当（OpenGL 的对象对于 CPU 而言就是一个数字，一个 Texture 或者 FrameBuffer 我们断点看到的就是一个 GLuint，如果环境隔离，我们随便操作 deleteTexture, deleteFrameBuffer 不会影响别的环境下的对象，但是如果环境打通，这些操作很可能会影响 Flutter 自己的 Context 下的对象），**所以作为一个框架的设计者，保证框架的封闭完整性才是首要。**

我们在开发过程中，碰到一个诡异的问题，定位了很久发现就是因为我们在主线程没有 setCurrentContext 的情况下，调用了 glDeleteFrameBuffer，从而误删了 Flutter 的 FrameBuffer，导致 flutter 渲染时 crash。所以建议如果采用这种方案的同学，Native 端的 GL 相关操作务必至少遵从以下一点：

1. 尽量不要在主线程做 GL 操作。
2. 在有 GL 操作的函数调用前，要加上 `setCurrentContext`。

还有一点就是本文大多数逻辑都是以 IOS 端为范例进行陈述，Android 整体原理是一致的，但是具体实现上稍有不同，Android 端 Flutter 自带的外接纹理是用 SurfaceTexture 实现，其机理其实也是 CPU 内存到 GPU 内存的拷贝，Android OpenGL 没有 ShareGroup 这个概念，用的是 shareContext，也就是直接把

Context 传出去。并且 Shell 层 Android 的 GL 实现是基于 C++ 的，所以 Context 是一个 C++ 对象，要将这个 C++ 对象和 **AndroidNative 端的 java Context 对象进行共享**，需要在 jni 层这样调用：(这里由于 android5.0 之前，EGLContext 的构造函数的参数类型为 int 型。)

```
static jobject GetShareContext(JNIEnv* env, jobject jcallee, jlong shell_holder) {
    void* ctxt = ANDROID_SHELL HOLDER->GetPlatformView()->GetContext();

    jclass versionClass = env->FindClass("android/os/Build$VERSION" );
    jfieldID sdkIntFieldID = env->GetStaticFieldID(versionClass, "SDK_INT", "I" );
    int sdkInt = env->GetStaticIntField(versionClass, sdkIntFieldID );
    __android_log_print(ANDROID_LOG_ERROR, "andymao", "sdkInt %d", sdkInt);
    jclass eglcontextClassLocal = env->FindClass("android/opengl/EGLContext");
    jmethodID eglcontextConstructor;
    jobject eglContext;
    if (sdkInt >= 21) {
        //5.0and above
        eglcontextConstructor=env->GetMethodID(eglcontextClassLocal, "<init>", "(J)V");
        if ((EGLContext)ctxt == EGL_NO_CONTEXT) {
            return env->NewObject(eglcontextClassLocal, eglcontextConstructor,
                                   reinterpret_cast<jlong>(EGL_NO_CONTEXT));
        }
        eglContext = env->NewObject(eglcontextClassLocal, eglcontextConstructor,
                                   reinterpret_cast<jlong>(jlong(ctxt)));
    }else{
        eglcontextConstructor=env->GetMethodID(eglcontextClassLocal, "<init>", "(I)V");
        if ((EGLContext)ctxt == EGL_NO_CONTEXT) {
            return env->NewObject(eglcontextClassLocal, eglcontextConstructor,
                                   reinterpret_cast<jlong>(EGL_NO_CONTEXT));
        }
        eglContext = env->NewObject(eglcontextClassLocal, eglcontextConstructor,
                                   reinterpret_cast<jint>(jint(ctxt)));
    }

    return eglContext;
}
```

最后我们本文相关修改已经提了一个单独的 pull request 给 Flutter，正在推动 flutter 将其作为另一个标准的外接纹理方案。<https://github.com/flutter/engine/pull/11276>

可定制化的 Flutter 相册组件竟如此简单

作者：闲鱼技术 – 邻云

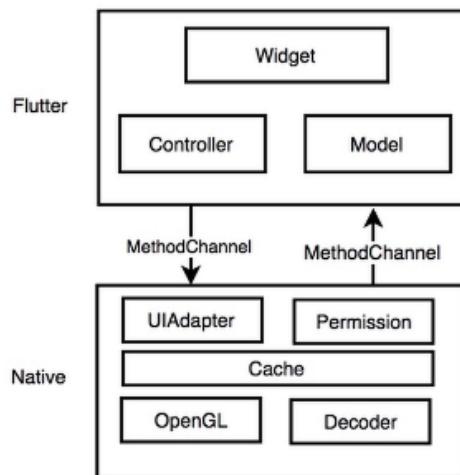
背景

开发图片、视频相关功能时，相册是一个绕不开的话题，因为大家基本都有从相册获取图片或者视频的需求。最直接的方式是调用系统相册接口，虽然基本功能是满足的，却无法满足一些高级功能，例如自定义 UI、多选图片等。

设计思路

闲鱼这套相册组件 API 使用简单，功能丰富灵活，具有较高的订制性。业务方可以选择完全接入组件，也可以选择在组件上面进行 UI 定制。

Flutter 做 UI 展现层，具体的数据由各 Native 平台提供。这种模式，天然从工程上把 UI、数据进行了隔离。我们在开发一个 native 组件的时候常常会使用 MVC 架构。Flutter 组件的开发的思路也基本类似。整体架构如下：



可以看出，在 Flutter 侧是一个典型的 MVC 架构，Model 就是代表图片、视频的 bean，View 就是 flutter 的 widget，Controller 就是调用各平台的一些接口。

在 Model 改变的时候 View 会重新 build 反映出 Model 的变化。View 的事件会触发 Controller 去 Native 获取数据然后更新 Model。Native 和 Flutter 通过 Method Channel 进行通信，两层之间没有强依赖关系，只需要按约定的协议进行通信即可。

Native 侧的组成部分，UIAdapter 主要是负责机型的适配、刘海屏、全面屏之类的识别。Permission 负责媒体读写权限的申请处理。Cache 主要负责缓存 GPU 纹理，在大图预览的时候提高响应速度。Decoder 负责解析 Bitmap，OpenGL 负责 Bitmap 转纹理。

需要说明的是：我们的这一套实现依赖于 flutter 外接纹理。在整个相册组件看到的大多数图片都是一个 GPU 纹理，这样给 java 堆内存的占用相对于以前的相册实现有大幅的降低。在低端机上面如果使用原生的系统相册，由于内存的原因，在应用放到后台的时候有被系统杀掉的风险。现象就是，从系统相册返回，app 重新启动了。使用 Flutter 相册组件，在低端机上面体验会有所改观。

一些难点

1. 分页加载

相册列表需要加载大量图片，Flutter 的 GridView 组件有好几个构造函数，比较容易犯的错误是使用了第一个函数，这需要在一开始就提供大量的 widget。应该选择第二个构造函数，GridView 在滑动的时候会回调 IndexedWidgetBuilder 来获取 widget，相当于一种懒加载。

```
GridView.builder({
  ...
  List<Widget> children = const <Widget>[],
  ...
})
GridView.builder({
  ...
  @required IndexedWidgetBuilder itemBuilder,
  int itemCount,
  ...
})
```

滑动过程中，图片滑过后，也就是不可见的时候要进行资源的回收，我们这里这里对应的就是纹理的删除。不断的滑动 GridView，内存上升后会处于稳定，不会一直增长。如果快速的来回滑动纹理会反复的创建和删除，这样会有内存的抖动，体验不是很好。

于是，我们维护了一个图片的状态机，状态有 None, Loading, Loaded, Wait_Dispose, Disposed。开始加载的时候，状态从 None 进入 Loading，这个时候用户看到的是空白或者是占位图，当数据回调回来会把状态设置为 Loaded 的这时候会重新 build widget 树来显示图片 icon，当用户滑走的时候状态进入 Wait_Dispose，这时候并不会马上 Dispose，如果用户又滑回来则会从 Wait_Dispose 进入 Loaded 状态，不会继续 Dispose。如果用户没有往回滑则会从 Wait_Dispose 进入 Disposed 状态。当进入 Disposed 状态后，再需要显示该图片的时候就需要重新走加载流程了。

2. 相册大图展示

当点击 GridView 的某张图片的时候会进行这张图片的大图展示，方便用户查看的更清楚。我们知道相机拍摄的图片分辨率都是很高的，如果完全加载，内存会有很大的开销，所以我们在 Decode Bitmap 的时候进行了缩放，最高只到 1080p。Android 原生的 Bitmap Decode 经验同样适用，先 Decode 出 Bitmap 的宽高，然后根据要展示的大小计算出缩放倍数，然后 Decode 出需要的 Bitmap。

Android 相册的图片大多是有旋转角度的，如果不处理直接显示，会出现照片旋转 90 度的问题，所以需要对 Bitmap 进行旋转，采用 Matrix 旋转一张 1080p 的图片在我的测试机器上面大概需要 200ms，如果使用 OpenGL 的纹理坐标进行旋转，大约只需要 10ms 左右，所以采用 OpenGl 进行纹理的旋转是一个较好的选择。

在进行大图预览的时候会进入一个水平滑动的 PageView，Flutter 的 PageView 一般来说是不会去主动加载相邻的 page 的。这里有一个取巧的办法，对于 PageController 的 viewportFraction 参数我们可以设置成为 0.9999，如下所示：

```
PageController(viewportFraction=0.9999)
```

还有另外一种办法，就是在 Native 侧做预加载。例如：在加载第 5 张图片的时候，相邻的 4, 6 的图片纹理提前进行加载，当滑动到 4, 6 的时候直接使用缓存的纹理。

3. 内存

相册图片使用 GPU 纹理，会大幅减少 Java 堆内存的占用，对整个 app 的性能有一定的提升。需要注意的是，GPU 的内存是有限的需要在使用完毕后及时删除，不然会有内存的泄漏的风险。另外，在 Android 平台删除纹理的时候需要保证在 GPU 线程进行，不然删除是没有效果的。

在华为 P8, Android5.0 上面进行了对比测试，Flutter 相册和原 native 相册总内存占用基本一致，在 GridView 列表页面，新增最大内存 13M 左右。它们的区别在于原 native 相册使用的是 Java 堆内存，Flutter 相册使用的是 Native 内存或者 Graphic 内存。

总结

这套相册组件 API 简单、易用，高度可定制。Flutter 侧层次分明，有 UI 订制需求的可以重写 Widget 来达到目的。另外这是一个不依赖于系统相册的相册组件，自身是完备的，能够和现有的 app 保持 UI、交互的一致性。同时为后面支持更多和相册相关的玩法打好基础。

后续计划

由于我们使用的是 GPU 纹理，可以考虑支持显示高清 4K 图片，而且客户端内存不会有太大的压力。但是 4k 图片的 Bitmap 转纹理需消耗更多的时间，UI 交互上面需要做些 loading 状态的支持。

组件功能丰富，稳定后，进行开源，回馈给社区。

揭晓闲鱼通过数据提升 Flutter 体验的真相

作者：闲鱼技术 - 三莅

背景

闲鱼客户端的 flutter 页面已经服务上亿级用户，这个时候 Flutter 页面的用户体验尤其重要，完善 Flutter 性能稳定性监控体系，可以及早发现线上性能问题，也可以作为用户体验提升的衡量标准。那么 Flutter 的性能到底如何？是否像官方宣传的那么丝滑？Native 的性能指标是否可以用来检测 Flutter 页面？下面给大家分享我们在实践中总结出来的 Flutter 的性能稳定性监控方案。

目标

过度的丢帧从视觉上会出现卡顿现象，体现在用户滑动操作不流畅；页面加载耗时过长容易中断操作流程；Flutter 部分 exception 会导致发生异常代码后面的逻辑没有走到从而造成逻辑 bug 甚至白屏。这些问题很容易考验用户耐心，引起用户反感。

所以我们制定以下三个指标作为线上 Flutter 性能稳定性标准：

1. 页面滑动流畅度
2. 页面加载耗时（首屏时长 + 可交互时长）
3. Exception 率

最终目标是让这些数据指标驱动 Flutter 用户体验升级。

页面滑动流畅度

我们先大概了解下屏幕渲染流程：CPU 先把 UI 对象转变 GPU 可以识别的信息存储进 displaylist 列表，GPU 执行绘图指令来执行 displaylist，取出相应的图元信息，进行栅格化渲染，显示到屏幕上，这样一个循环的过程实现屏幕刷新。

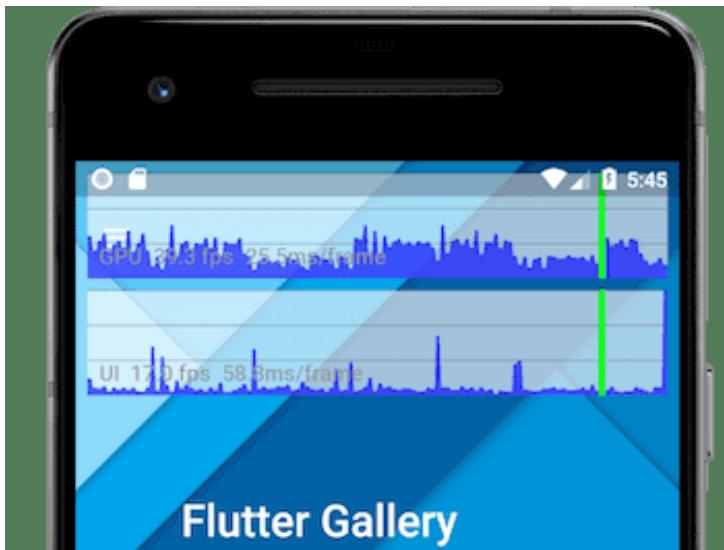
闲鱼客户端采用的 Native、Flutter 混合技术方案，Native 页面 FPS 监控采用

集团高可用方案，Flutter 页面是否可以直接采用这套方案监控？

普遍的 FPS 检测方案 Android 端采用的是 Choreographer.FrameCallBack，IOS 采用的是 CADisplayLink 注册的回调，原理是类似的，在每次发出 Vsync 信号，并且 CPU 开始计算的时候执行到对应的回调，这个时候表示屏幕开始一次刷新，计算固定时间内屏幕渲染次数来得到 fps。（这种方式只能检测到 CPU 卡顿，对于 GPU 的卡顿是无法监控到的）。由于这两种方法都是在主线程做检测处理，而 flutter 的屏幕绘制是在 UI TaskRunner 中进行，真正的渲染操作是在 GPU TaskRunner 中，关于详细的 Flutter 线程问题可以参考闲鱼之前的文章：[深入理解 Flutter 引擎线程模式](#)。

这里我们得出结论：Native 的 FPS 检测方法并不适用于 Flutter。

Flutter 官方给我们提供了 Performance Overlay（具体参考 [Flutter performance profiling](#)）作为检测帧率工具，可否直接拿来用？



上图显示了 Performance Overlay 模式下的帧率统计，可以看到，Flutter 分开计算 GPU 和 UI TaskRunner。UI Task Runner 被 Flutter Engine 用于执行 Dart root isolate 代码，GPU Task Runner 被用于执行设备 GPU 的相关调用。通过对 flutter engine 源码分析，UI frame time 是执行 window.onBeginFrame 所花费的

总时间。GPU frame time 是处理 CPU 命令转换为 GPU 命令并发送给 GPU 所花费的时间。

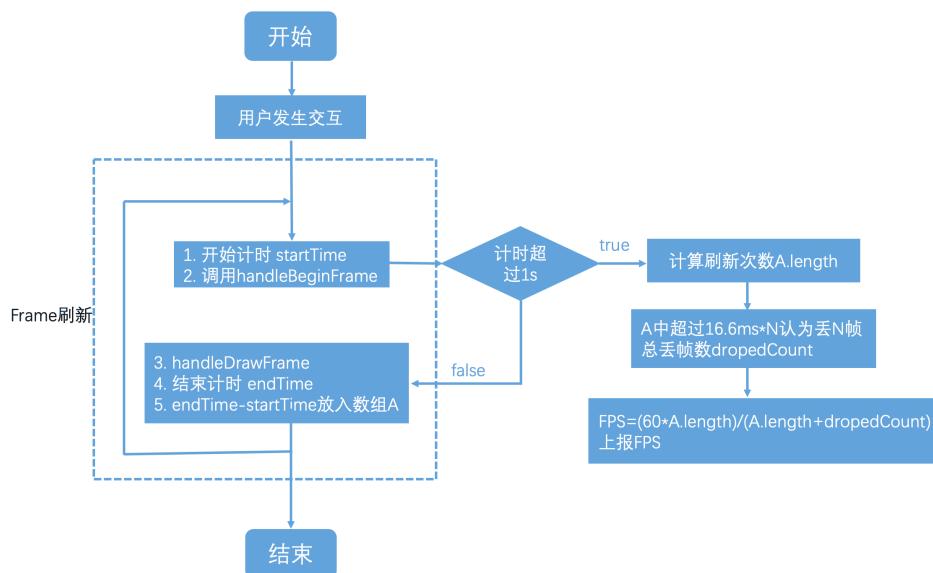
这种方式只能在 debug 和 profile 模式下开启，没有办法作为线上版本的 fps 统计。但是我们可以通过这种方式获得启发，通过监听 Flutter 页面刷新回调方法 handleBeginFrame()、handleDrawFrame() 来计算实际 FPS。

具体实现方式：

注册 WidgetsFlutterBinding 监听页面刷新回调 handleBeginFrame()、handleDrawFrame()

```
handleBeginFrame: Called by the engine to prepare the framework to produce a new frame.  
handleDrawFrame: Called by the engine to produce a new frame.
```

通过计算 handleBeginFrame 和 handleDrawFrame 之间的时间间隔计算帧率，主要流程如下图：



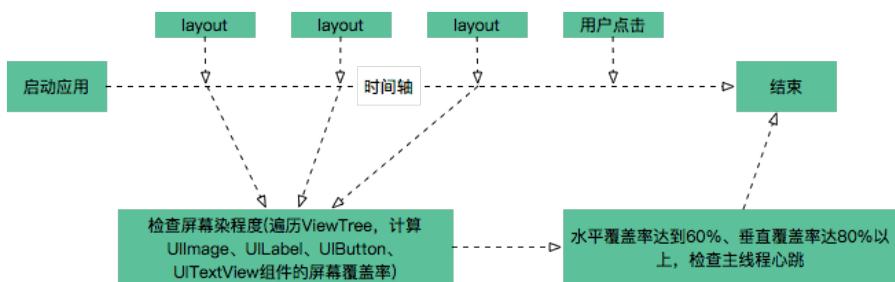
效果

到这里，我们完成 Flutter 中页面帧率的统计，这种方式统计的是 UI TaskRunner 中的 CPU 操作耗时，GPU 操作在 Flutter 引擎内部实现，要修改引擎来监控完整的渲染耗时，我们目前大部分的场景没有复杂到 gpu 卡顿，问题主要还是集中在 CPU，所以说可以反应出大部分问题。从线上数据来看，release 模式下 Flutter 的流畅度还是蛮不错的，ios 的主要页面均值基本维持在 50fps 以上，android 相对 ios 略低。这里需要注意的是帧率的均值 fps 在反复滑动过程中会有一个稀释效果，导致一些卡顿问题没有暴露出来，所以除了 fps 均值，需要综合掉帧范围、卡顿秒数、滑动时长等数据才能反应出页面流畅度情况。

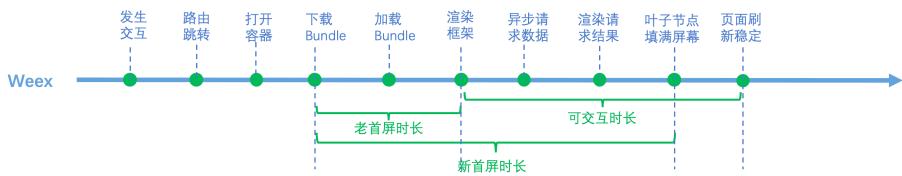
页面加载时长

Native 和 Weex 页面加载算法对比

集团内部高可用方案统计 Native 页面加载时长是通过容器初始化后开启定时器在容器 layout 的时候检查屏幕渲染程度，计算可见组件的屏幕覆盖率，满足条件水平 $>60\%$ ，垂直 $>80\%$ 以上认为满足页面填充程度，再检查主线程心跳判断是否加载完成。



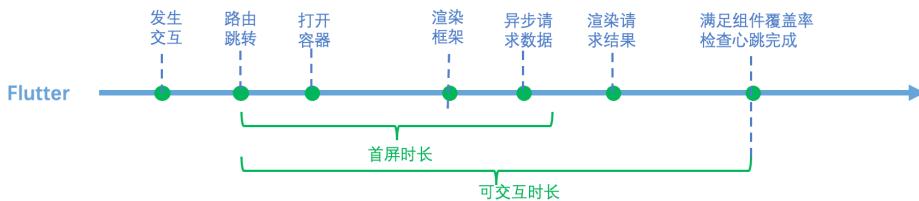
再来看看 weex 页面加载流程和统计数据的定义。



Weex 的页面刷新稳定定义：屏幕内 view 渲染完成且 view 树稳定的时间

具体实现：当屏幕内发生 view 的 add/rm 操作时，认为是可交互点，记录数据。

直到没有再发生为止。

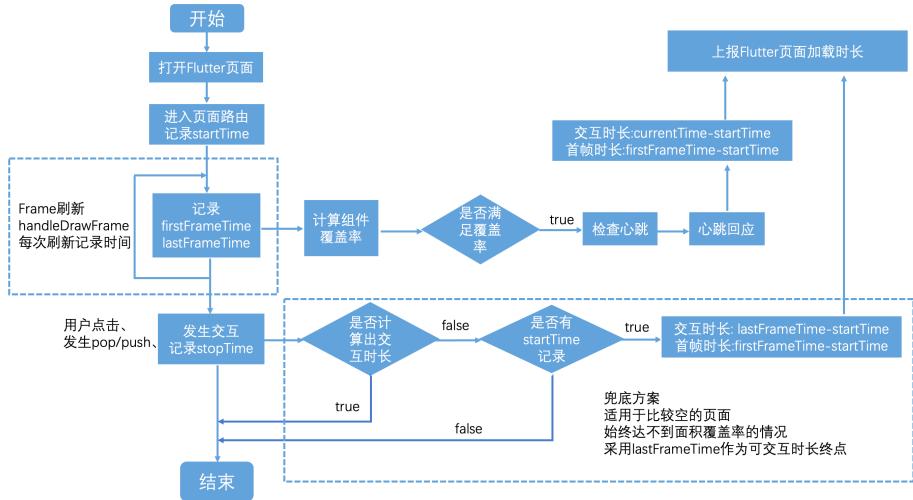


在概念上 Flutter 和 weex 的首屏时长和可交互时长并不完全一致，Flutter 之所以选择从路由跳转开始计算时长主要是因为这种计算方式更贴近用户体验，可以获取更多的问题信息，比如路由跳转的时长问题等。

Flutter 的具体实现

Flutter 的可交互时长 end 点采用的算法与 native 一致，可见组件满足页面填充程度并且完成心跳检查的情况下任务可交互，另外对于一些比较空的页面，组件面积小，无法达到水平 $>60\%$ ，垂直 $>80\%$ 的条件，就用交互前最后一次 Frame 刷新时间点作为 end 点。

具体流程如下图：



效果

由于 debug 模式采用的 JIT 编译，debug 模式下体验加载时长偏长，但是 release 模式下的 AOT 编译时长明显缩短很多，整体页面加载时长还是要优于 weex。

Exception 率

Flutter部分 exception/error 会导致代码后面的逻辑没有走到造成页面或逻辑 bug，所以 flutter 的 exception 需要作为稳定性的标准之一

定义

FlutterException 率 = exception 发生次数 / flutter 页面 PV

分子: exception 发生次数 (已过滤掉白名单)

Flutter 内部 assert、try-catch 和一些异常逻辑的地方会统一调用 Flutter-Error.onError

通过重定向 FlutterError.onError 到自己的方法中监测 exception 发生次数，并上报 exception 信息

分母: flutter 页面 PV

具体实现如下：

```
Future<Null> main() async {
  FlutterError.onError = (FlutterErrorDetails details) async {
    Zone.current.handleUncaughtError(details.exception, details.stack);
  };

  runZoned<Future<Null>>(() async {
    runApp(new HomeApp());
  }, onError: (error, stackTrace) async {
    await _reportError(error, stackTrace);
  });
}
```

其中，FlutterError.onError 只会捕获 Flutter framework 层的 error 和 exception，官方建议将这个方法按照自己的 exception 捕获上报需求定制。在实践过程中，我们遇到很多不会对用户体验产生任何影响的 exception 会被频繁触发，这类没有改善意义的 exception 可以添加白名单过滤上报。

效果

有了线上 exception 的监控，可以及早发现隐患，获取问题堆栈信息，方便定位 bug，提示整体稳定性。

总结

到这里，我们完成 Flutter 页面滑动流畅度、页面加载时长和 Exception 率的统计，对于 Flutter 的性能有一个具体的数字化标准，对以后的用户体验提升和性能问题排查提供基础。目前闲鱼客户端的商品详情页和主发布页已经全量 Flutter 化，感兴趣的同學可以体验下这两个页面和其他页面的性能差异，最后欢迎大家提供反馈和建议。

打通前后端逻辑，客户端 Flutter 代码一天上线

作者：闲鱼技术 - 景松

一、前沿

随着闲鱼的业务快速增长，运营类的需求也越来越多，其中不乏有很多界面修改或运营坑位的需求。闲鱼的版本现在是每 2 周一个版本，如何快速迭代产品，跳过窗口期来满足这些需求？另外，闲鱼客户端的包体也变的很大，Android 的包体大小，相比 2016 年，已经增长了近 1 倍，怎么能将包体大小降下来？首先想到的是动态化的解决此类问题。

对于原生的能力的动态化，Android 平台各公司都有很完善的动态化方案，甚至 Google 还提供了 Android App Bundles 让开发者们更好地支持动态化。由于 Apple 官方担忧动态化的风险，因此并不太支持动态化。因此动态化能力就会考虑跟 Web 结合，从一开始基于 WebView 的 Hybrid 方案，到现在与原生相结合的 React Native 、Weex。

与此同时，随着闲鱼 Flutter 技术的推广，已经有 10 多个页面用 Flutter 实现，Flutter 的动态化诉求也随之增多。上面提到的几种方式都不适合 Flutter 场景，如何解决这个问题？

二、动态方案

2.1 CodePush

CodePush 是谷歌官方的动态化方案，Dart VM 在执行的时候，加载 `isolate_snapshot_data` 和 `isolate_snapshot_instr` 2 个文件，通过动态更改这些文件，就达到动态更新的目的。官方的 Flutter 源码当中，已经有相关的提交来做动态更新的内容，具体可以参考 [ResourceExtractor.java](#)。目前，此功能还在开发中，期待中 ing。

2.2 动态模板

动态模板，就是通过定义一套 DSL，在端侧解析动态的创建 View 来实现动态化，比如 [LuaViewSDK](#)、[Tangram-iOS](#) 和 [Tangram-Android](#)。这些方案都是创建的 Native 的 View，如果想在 Flutter 里面实现，需要创建 Texture 来桥接；Native 端渲染完成之后，再将纹理贴在 Flutter 的容器里面，实现成本很高，性能也有待商榷，不适合闲鱼的场景。

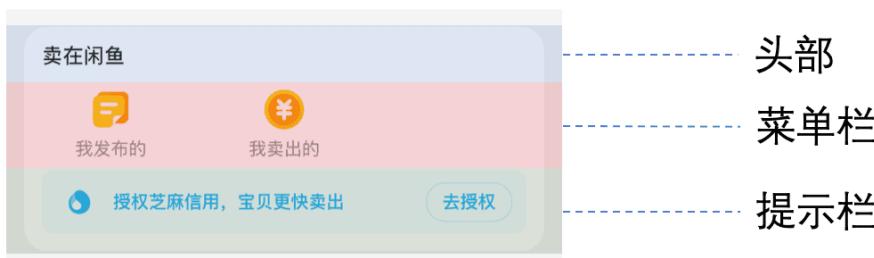
所以我们提出了闲鱼自己的 Flutter 动态化方案，前面已经有同事介绍过方案的原理：[《做了2个多月的设计和编码，我梳理了Flutter动态化的方案对比及最佳实现》](#)，下面看下具体的实现细节。

三、模板编译

自定义一套 DSL，维护成本较高，怎么能不自定义 DSL 来实现动态加载？闲鱼的方案就是直接将 Dart 文件作为模板，中间将其转化成 JSON 格式的协议数据，端侧拿到协议数据再进行解析；这样做好处就是 Dart 模板文件可以快速沉淀到端侧，可以很方便的进行二次开发。

3.1 模板规范

先来看下一个完整的模板文件，以新版我的页面为例，这是一个列表结构，每个区块都是一个独立的 Widget，现在我们期望将“卖在闲鱼”这个区块动态渲染，对这个区块拆分之后，需要 3 个子控件：头部、菜单栏、提示栏；因为这 3 部分界面有些逻辑处理，所以先把他们的逻辑内置。



内置的子控件分别是 `MenuTitleWidget`、`MenuItemWidget` 和 `HintItemWidget`，编写的模板如下：

```

@Override
Widget build(BuildContext context) {
    return new Container(
        child: new Column(
            children: <Widget>[
                new MenuTitleWidget(data), // 头部
                new Column( // 菜单栏
                    children: <Widget>[
                        new Row(
                            children: <Widget>[
                                new MenuItemWidget(data.menus[0]),
                                new MenuItemWidget(data.menus[1]),
                                new MenuItemWidget(data.menus[2]),
                            ],
                        )
                    ],
                ),
                new Container( // 提示栏
                    child: new HintWidgetItem(data.hints[0]),
                ),
            ],
        );
    }
}

```

中间省略了样式描述，可以看到写模板文件就跟普通的 widget 写法一样，但是有几点要注意：

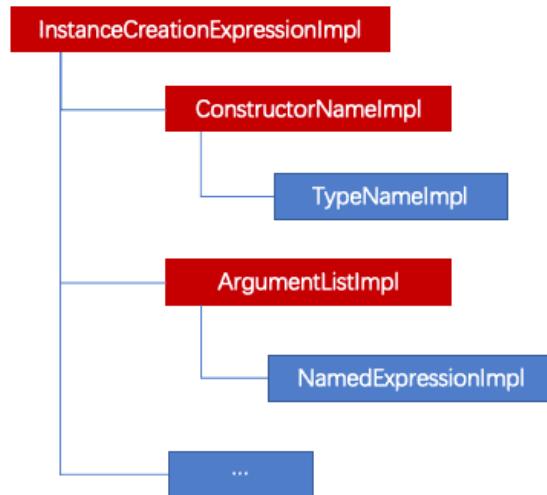
1. 每个 Widget 都需要用 `new` 或 `const` 来修饰
2. 数据访问以 `data` 开头，数组形式以 `[]` 访问，字典形式以 `.` 访问

模板写好之后，就要考虑怎么在端上渲染，早期版本是直接在端侧解析文件，但是考虑到性能和稳定性，还是放在前期先编译好，然后下发到端侧。

3.2 编译流程

编译模板就要用到 Dart 的 `Analyzer` 库，通过 `parseCompilationUnit` 函数直接将 Dart 源码解析成为以 `CompilationUnit` 为 Root 节点的 AST 树中，它

包含了 Dart 源文件的语法和语义信息。接下来的目标就是将 `CompilationUnit` 转换成为一个 JSON 格式。



上面的模板解析出来 `build` 函数孩子节点是 `ReturnStatementImpl`，它又包含了一个子节点 `InstanceCreationExpressionImpl`，对应模板里面的 `new Container(...)`，它的孩子节点中，我们最关心的就是 `ConstructorNameImpl` 和 `ArgumentListImpl` 节点。`ConstructorNameImpl` 标识创建节点的名称，`ArgumentListImpl` 标识创建参数，参数包含了参数列表和变量参数。

定义如下结构体，来存储这些信息：

```

class ConstructorNode {
    // 创建节点的名称
    String constructorName;
    // 参数列表
    List<dynamic> argumentsList = <dynamic>[];
    // 变量参数
    Map<String, dynamic> arguments = <String, dynamic>{};
}
  
```

递归遍历整棵树，就可以得到一个 `ConstructorNode` 树，以下代码是解析单个 Node 的参数：

```

ArgumentList argumentList = astNode;

for (Expression exp in argumentList.arguments) {
    if (exp is NamedExpression) {
        NamedExpression namedExp = exp;
        final String name = ASTUtils.getNodeString(namedExp.name);
        if (name == 'children') {
            continue;
        }

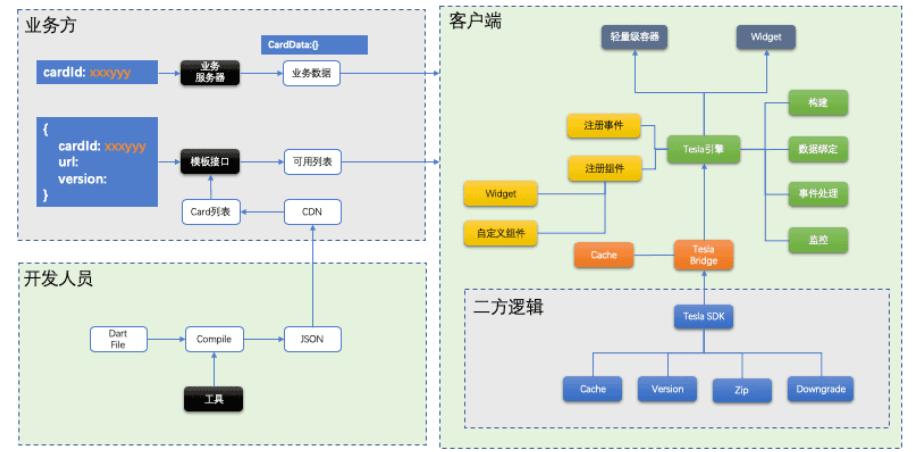
        /// 是函数
        if (namedExp.expression is FunctionExpression) {
            currentNode.arguments[name] =
                FunctionExpressionParser.parse(namedExp.expression);
        } else {
            /// 不是函数
            currentNode.arguments[name] =
                ASTUtils.getNodeString(namedExp.expression);
        }
    } else if (exp is PropertyAccess) {
        PropertyAccess propertyAccess = exp;
        final String name = ASTUtils.getNodeString(propertyAccess);
        currentNode.argumentsList.add(name);
    } else if (exp is StringInterpolation) {
        StringInterpolation stringInterpolation = exp;
        final String name = ASTUtils.getNodeString(stringInterpolation);
        currentNode.argumentsList.add(name);
    } else if (exp is IntegerLiteral) {
        final IntegerLiteral integerLiteral = exp;
        currentNode.argumentsList.add(integerLiteral.value);
    } else {
        final String name = ASTUtils.getNodeString(exp);
        currentNode.argumentsList.add(name);
    }
}
}

```

端侧拿到这个 `ConstructorNode` 节点树之后，就可以根据 Widget 的名称和参数生成一棵 Widget 树。

四、渲染引擎

端侧获得 JSON 格式的模板信息，渲染引擎的工作，就是解析模板信息并创建 Widget。整个工程的框架和工作流如下所示：



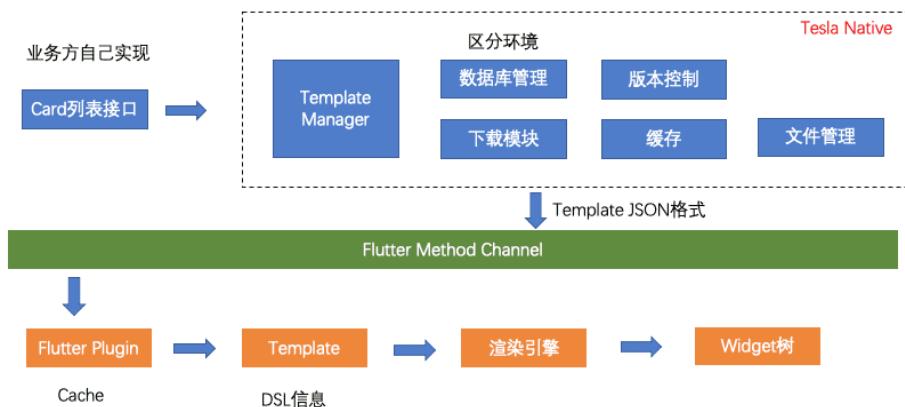
流程简介：

1. 开发人员编写 dart 文件，编译上传到 CDN
2. 端侧拿到模板列表，并在端侧存库
3. 业务方直接下发对应的模板 id 和模板数据
4. Flutter 侧再通过桥接获取到模板，并创建 Widget 树

对于 Native 测，主要负责模板的管理，通过桥接输出到 Flutter 侧。

4.1 模板获取

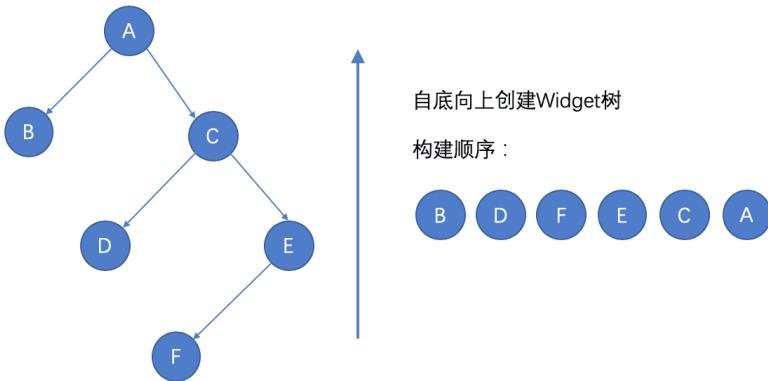
模板获取分为 2 部分：Native 和 Flutter，Native 主要负责模板的管理，包括下载、降级、缓存等。



程序启动后，会先获取模板列表，业务方需要自己实现，Native 层获取到模板列表会先存储在本地数据库中。Flutter 侧业务代码用到模板的时候，再通过桥接获取模板信息，就是我们前面提到的 JSON 格式的信息，Flutter 也会有缓存，以减少 Flutter 和 Native 的交互。

4.2 Widget 创建

Flutter 侧当拿到 JSON 格式的，先解析出 `ConstructorNode` 树，然后递归创建 Widget。



创建每个 Widget 的过程，就是解析节点中的 `argumentsList` 和 `arguments` 并做数据绑定。例如，创建 `HintItemWidget` 需要传入提示的数据内容，`new HintItemWidget(data.hints[0])`，在解析 `argumentsList` 时，会通过 `key-path` 的方式从原始数据中解析出特定的值。



解析出来的值都会存储在 `WidgetCreateParam` 里面，当递归遍历每个创建节点，每个 widget 都可以从 `WidgetCreateParam` 里面解析出需要的参数。

```

/// 构建 widget 用的参数
class WidgetCreateParam {
  String constructorName; // 构建的名称
  dynamic context; // 构建的上下文
  Map<String, dynamic> arguments = <String, dynamic>{}; // 字典参数
  List<dynamic> argumentsList = <dynamic>[]; // 列表参数
  dynamic data; // 原始数据
}
  
```

通过以上的逻辑，就可以将 `constructorNode` 树转换为一棵 `Widget` 树，再交给 Flutter Framework 去渲染。

至此，我们已经能将模板解析出来，并渲染到界面上，交互事件应该怎么处理？

4.3 事件处理

界面交互，一般都会通过 `GestureDetector`、`InkWell` 等来处理点击事件，处理逻辑是函数，这块怎么做动态化？

以 `InkWell` 组件为例，定义它的 `onTap` 函数为 `openURL(data.hints[0].href, data.hints[0].params)`，在解析逻辑中，会解析成为一个以 `OpenURL` 作为 ID 的事件。在 Flutter 侧，会有一个事件处理的映射表。当用户点击 `InkWell`

时，会查找对应的处理函数，并解析出对应的参数列表并传递过去，代码如下：

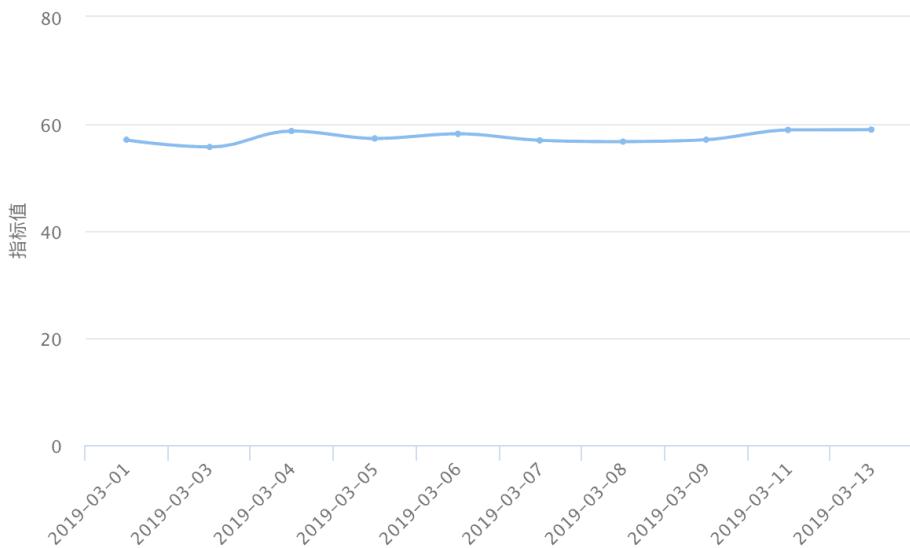
```
...
final List<dynamic> tList = <dynamic>[];
// 解析出参数列表
exp.argumentsList.forEach((dynamic arg) {
    if (arg is String) {
        final dynamic value = valueFromPath(arg, param.data);
        if (value != null) {
            tList.add(value);
        } else {
            tList.add(arg);
        }
    } else {
        tList.add(arg);
    }
});
// 找到对应的处理函数
final dynamic handler =
    TeslaEventManager.sharedInstance().eventHandler(exp.actionName);
if (handler != null) {
    handler(tList);
}
...
```

五、效果

新版我的页面添加了动态化渲染能力之后，如果有需求新添加一种组件类型，就可以直接编译发布模板，服务端下发新的数据内容，就可以渲染出来了；动态化能力有了，大家会关心渲染性能怎么样。

5.1 帧率

在加了动态加载逻辑之后，已经开放了 2 个动态卡片，下图是新版本我的页面近半个月的的帧率数据：



从上图可以看到，帧率并没有降低，基本保持在 55–60 帧左右，后续可以多添加动态的卡片，观察下效果。

注：因为我的页面会有本地的一些业务判断，从其他页面回到我的 tab，都会刷新界面，所以帧率会有损耗。

从实现上分析，因为每个卡片，都需要遍历 `ConstructorNode` 树来创建，而且每个构建都需要解析出里面的参数，这块可以做一些优化，比如缓存相同的 Widget，只需要映射出数据内容并做数据绑定。

5.2 失效率

现在监控了渲染的逻辑，如果本地没有对应的 Widget 创建函数，会主动抛 Error。监控数据显示，渲染的流程中，还没有异常的情况，后续还需要对桥接层和 native 层加错误埋点。

六、展望

基于 Flutter 动态模板，之前需要走发版的 Flutter 需求，都可以来动态化更改。而且以上逻辑都是基于 Flutter 原生的体系，学习和维护成本都很低，动态的代码也

可以快速的沉淀到端侧。

另外，闲鱼正在研究 UI2Code 的黑科技，不了解的老铁，可以参考闲鱼大神的这篇文章《[重磅系列文章！UI2CODE 智能生成 Flutter 代码——整体设计篇](#)》。可以设想下，如果有个需求，需要动态的显示一个组件，UED 出了视觉稿，通过 UI2Code 转换成 Dart 文件，再通过这个系统转换成动态模板，下发到端侧就可以直接渲染出来，程序员都不需要写代码了，做到自动化运营，看来以后程序员失业也不是没有可能了。

基于 Flutter 的 Widget，还可以拓展更多个性化的组件，比如内置动画组件，就可以动态化下发动画了，更多好玩的东西等待大家来一起探索。

参考文献

1. <https://github.com/flutter/flutter/issues/14330>
2. <https://www.dartlang.org/>
3. https://mp.weixin.qq.com/s/4s6MaiuW4VoHr_7f0S_vuQ
4. <https://github.com/flutter/engine>

流言终结者 – Flutter 和 RN 谁才是更好的跨端开发方案？

作者：闲鱼技术 – 灯阳

背景

论坛上很多小伙伴关心为什么闲鱼选择了 Flutter 而不选择其他跨端方案？站在质量的角度，高性能是一个很重的因素，我们使用 Flutter 重写了宝贝详情页之后，对比了 Flutter 和 Native 详情页的性能表现，结论是中高端机型上 Flutter 和 Native 不相上下，在低端机型上，Flutter 会比 Native 更加的流畅，其实闲鱼团队在使用 Flutter 做详情页过程中，没有更多地关注性能优化，为了更快地上线，也是优先功能的实现，不过测试结果出来之后，却出乎意料地优于原先的 Native 的实现（具体的测试结果，属于敏感数据，要走披露流程，伤不起…）

但是这样很显然不能敷衍过去，仔细想了想，确实 Flutter 的定位并不是要替代 Native，他只想做一个极致的跨端解决方案，所以还是要回到跨端解决方案的赛道，给您从性能角度比一比，谁才是更好的跨端开发方案？

参赛选手

[Flutter]

Flutter is Google's mobile app SDK for crafting high-quality native interfaces on iOS and Android in record time. Flutter works with existing code, is used by developers and organizations around the world, and is free and open source.

[REACT NATIVE]

We're working on a large-scale rearchitecture of React Native to make

it more flexible and integrate better with native infrastructure in hybrid JavaScript/native apps.

鸣锣开赛

怎么比

怎么比较确实伤脑筋，自己也写了一个 Flutter 和一个 RN 的 App，但是实在太丑陋，担心大家关注点都到我的烂代码上了，所以在 Github 上找到了一个跨端开发高手 Car Guo，用 Flutter 和 RN 分别实现的一个实际可用的 App，Car Guo 谦虚表示其实也写的比较粗糙，但是在我看来这个是具备真实使用场景的 App (Github 客户端 App，提供丰富的功能，旨在更好的日常管理和维护个人 Github)，还是有代表性的 [Flutter] <https://github.com/CarGuo/GSYGithubAppFlutter> [REACT NATIVE] <https://github.com/CarGuo/GSYGithubApp>

场景

1. 默认登录成功。
2. “动态”页，点击搜索按钮，搜索关键字“Java”，正常速度浏览 3 页，等等第 4 页加载完成后回退。
3. 点击“趋势”页 Tab，浏览 Feeds 到页面底部，点击最底部的 Item，进入 Item 后，浏览详情 + 浏览 3 页的动态后回退，到“我的”Tab 页。
4. 查看“我的”Feeds 到底部，点击右上角搜索按钮，搜索关键字“C”，浏览 3 页后，等等第 4 页加载完成后场景结束。

测试工具

- iOS
- 掌中测 (iOS 端)：CPU，内存
- Instruments：FPS
- Android

- 基于 Adb 的 Shell 脚本: CPU, 内存, FPS

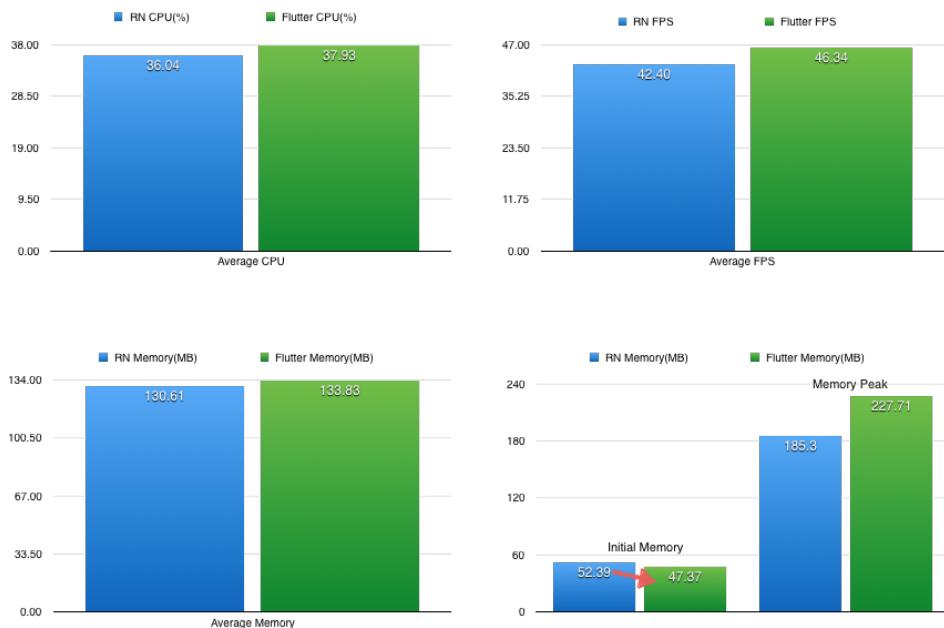
测试机型

- iOS: iPhone 5c 9.0.1 / iPhone 6s 10.3.2
- Android: Xiaomi 2s 5.0.2 / Samsung S8 7.0

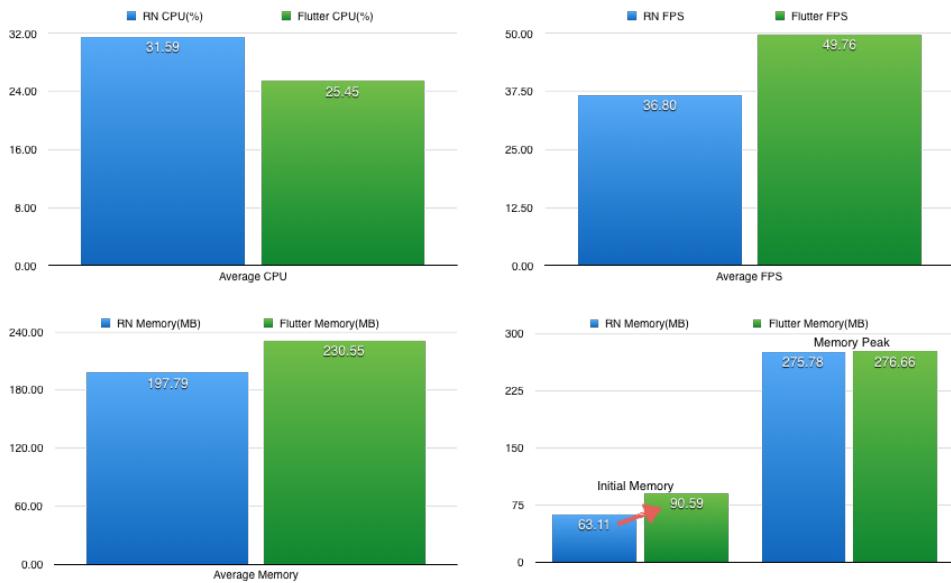
数据分析

iOS

iPhone 5c 9.0.1



iPhone 6s 10.3.2

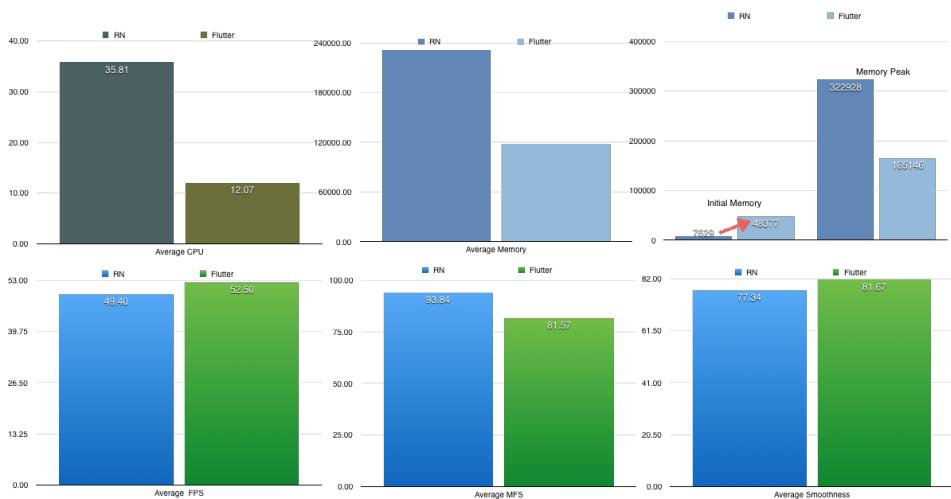


测试结论

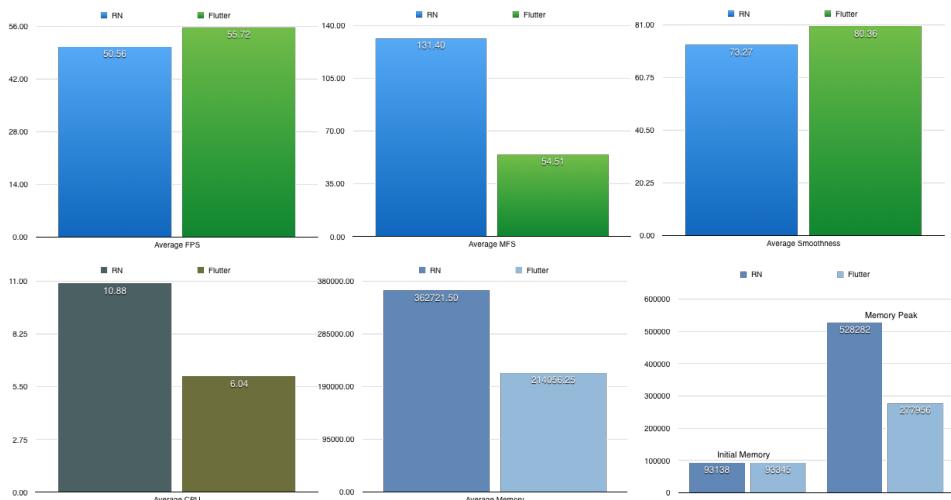
1. Flutter 在低端和中端的 iOS 机型上，FPS 的表现都优于 RN。
2. CPU 的使用上 Flutter 在低端机上表现略差于 RN，中端机型略优于 RN。
3. 值得注意的是内存上的表现（上图红色箭头区域），Flutter 在低端机型上的起始内存和 RN 几乎一致，在中端机型上会多 30M 左右的内存（分析为 Dart VM 的内存），可以想到这应该是 Flutter 针对低端和中端机型上内存策略是不一样的，可用内存少的机型，Dart VM 的初始内存少，运行时进行分配（这样也可以理解为什么在低端机上带来了更多的 CPU 损耗），中端机器上预分配了更多的 VM 内存，这样在处理时会更加的游刃有余，减少 CPU 的介入，带来更流畅的体验。可以看出，Flutter 团队在针对不同机型上处理更加的细腻，目的就是为了带来稳定流畅的体验。

Android

Xiaomi 2s 5.0.2



Sumsung S8 7.0



注: MFS – Max Frame Space: 指的是去掉 buffer 之后的两帧的时间差

测试结论

1. Flutter 在高低端机的 CPU 上的表现都优于 RN，尤其在低端的小米 2s 上有着更优的表现。
2. Android 端在原来 FPS 基础上增加了流畅度的指标，FPS 和流畅度的表现 Flutter 优于 RN(计算规则见附参考文章)。
3. Android 端的内存也是值得关注的一点，在小米 2s 上起始内存 Flutter 明显比 RN 多 40M，RN 在测试过程中内存飞涨，Flutter 相比之下会更稳定，内存上 RN 侧的代码是需要调优的，同一套代码 Flutter 在 Android 和 iOS 上并没有很大的差异，但是 RN 的却要在单端调优，Flutter 在这项比拼上又更胜一筹。比较奇怪的是三星 S8 上 Flutter 和 RN 的初始内存是一致的，猜测是 RN 也 Android 高端机型上也会预分配一些内存，具体细节还需要更进一步的研究。

升旗仪式

看了之前的数据，做为裁判的我会把金牌颁给 Flutter，在测试过程中的体验和数据上来看 Flutter 都优于 RN，并且开发这个 App 的是一位 Android 的开发同学，Flutter 和 RN 对于他来说都是全新的技术栈，Car Guo 同学更倾向性地让大家得到一致性的使用体验，性能方面并没有投入太多的时间进行调优，由此看出 Flutter 在跨端开发上在同样投入的情况下，可以获得更佳的性能，更好的用户体验。

一些思考

拿到了这些数据，也感受到 Flutter 带来福利，那 Flutter 为什么可以做到这么流畅呢？Flutter 是如何优化了渲染，Dart VM 的 Runtime 是怎么玩的？请大家继续关注后续解密文章。

参考

- Android FPS& 流畅度：<https://testerhome.com/topics/4775>
- Android 内存获取方式：`dumpsys meminfo packageName`

- Android CPU 通过 busybox 执行 top 命令获取
- iOS CPU 获取方式：累计每个线程中的 CPU 利用率

```

for (j = 0; j < thread_count; j++)
{
    ATCPUDO *cpuDO = [[ATCPUDO alloc] init];
    char name[256];
    pthread_t pt = pthread_from_mach_thread_np(thread_list[j]);
    if (pt) {
        name[0] = '\0';
        __unused int rc = pthread_getname_np(pt, name, sizeof name);
        cpuDO.threadid = thread_list[j];
        cpuDO.identify = [NSString stringWithFormat:@"%s", name];
    }
    thread_info_count = THREAD_INFO_MAX;
    kr = thread_info(thread_list[j], THREAD_BASIC_INFO, (thread_info_t)thinfo,
    &thread_info_count);
    if (kr != KERN_SUCCESS) {
        return nil;
    }
    basic_info_th = (thread_basic_info_t)thinfo;
    if (!(basic_info_th->flags & TH_FLAGS_IDLE)) {
        tot_sec = tot_sec + basic_info_th->user_time.seconds + basic_info_th->system_
        time.seconds;
        tot_usec = tot_usec + basic_info_th->system_time.microseconds + basic_info_
        th->system_time.microseconds;
        tot_cpu = tot_cpu + basic_info_th->cpu_usage / (float)TH_USAGE_SCALE * 100.0;
        cpuDO.usage = basic_info_th->cpu_usage / (float)TH_USAGE_SCALE * 100.0;
        if (container) {
            [container addObject:cpuDO];
        }
    }
}
} // for each thread

```

- iOS 内存获取方式：测试过程中使用的是 phys_footprint，是最准确的物理内存，很多开源软件用的是 resident_size（这个值代表的是常驻内存，并不能很好地表现出真实内存变化，这可以另开文章细谈）

```

if ([[UIDevice currentDevice].systemVersion intValue] < 10) {
    kern_return_t kr;
    mach_msg_type_number_t info_count;
    task_vm_info_data_t vm_info;
    info_count = TASK_VM_INFO_COUNT;
    kr = task_info(mach_task_self(), TASK_VM_INFO_PURGEABLE, (task_info_t)&vm_

```

```
info,&info_count);
if (kr == KERN_SUCCESS) {
    return (vm_size_t)(vm_info.internal + vm_info.compressed - vm_info.purgeable_
volatile_pmap);
}
return 0;
}

task_vm_info_data_t vmInfo;
mach_msg_type_number_t count = TASK_VM_INFO_COUNT;
kern_return_t result = task_info(mach_task_self(), TASK_VM_INFO, (task_info_
t) &vmInfo, &count);
if (result != KERN_SUCCESS)
    return 0;
return (vm_size_t)vmInfo.phys_footprint;
```



阿里技术

扫一扫二维码图案，关注我吧



「阿里技术」微信公众号



闲鱼技术微信公众号



阿里云开发者社区