# **京东技术中台的Flutter实践之路**

在 2019 年，Flutter 推出了多个正式版本，支持的终端越来越多，使用的项目也越来越多。Flutter 正在经历从小范围尝鲜到大面积应用的过程，越来越多的研发团队加入到 Flutter 的学习热潮中，京东作为互联网大厂之一也积极参与了 Flutter 的跨端方案研究。本文将介绍京东在 Flutter 上的应用方案和相关优化成果。

### **为什么考虑Flutter技术方案**

其实京东很早就开始研究并实践跨端的开发解决方案，最早使用的是Hybrid App的技术方案，从2015年低开始逐步转向RN技术栈，目前应该是业内RN技术平台应用最广泛、配套设施比较完善的公司之一。从2018年中开始，我们也关注到了Flutter技术，最吸引我们的特性是高性能和兼容性。这两点也是目前RN技术相对不足的地方。高性能指的是复杂场景和交互下的渲染性能，兼容性指的是不同终端平台上的布局和体验的一致性，这点在碎片化严重的android平台上尤其重要。

### **京东在Flutter的实践**

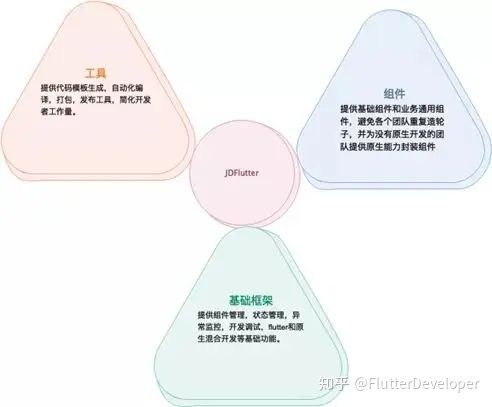
随着2018年底Google正式发布了Flutter预览版本，京东内部也越来越多的研发团队有用Flutter进行开发业务的诉求。我们正式启动研发并内部发布了JDFlutter引擎。在官方Flutter引擎之上，我们做了额外的优化和功能扩展：

* **Flutter工程改造**： 对Flutter开发环境和dart代码管理进行优化，可以无缝集成到现有APP中并支持自动化dart编译打包，便于开发和调试。
* **路由及多页面管理**： 对原生页面和flutter页面实现了集中路由管理，可以双向传参、跳转并且进行了共享内存优化。
* **扩展UI组件库**： 官方支持的Material和Cupertino样式不能满足需求，我们内部实现了自定义样式的组件库。
* **原生能力扩展**： 对官方原生能力进行了扩展，封装了包括网络、登陆、埋点等等基础能力的打通并提供了50+原生扩展API。
* **Android端动态化支持**： 在Android端实现了动态化支持，可以线上热更新业务。iOS端暂不支持动态化。

目前京东商城、京东视频、京东到家、京东物流、7Fresh等APP都有业务采用JDFlutter进行开发。

### **JDFlutter框架设计**

JDFlutter整体的框架结构，主要包含：基础框架、组件、工具三部分，如图所示：



### **基础框架**

JDFlutter基础框架分为三层架构，包含JDFlutter基础层，通用业务层，业务层。

* **基础层**：提供了Flutter的基础组件支持，包括组件管理，状态管理等；基础层完全独立，对业务没有依赖。
* **通用业务层**：提供了通用型业务组件支持，例如登录组件，支付组件等；通用业务层依赖于基础层。
* **业务层**：即具体业务逻辑实现层，根据业务需要进行不同组件的组合，实现业务页面的快速开发。



### **核心组件**

* **组件管理**：组件之间通过标准的协议接口进行通信，降低组件耦合，便于维护及组件升级；
* **状态管理**：实现数据和界面分离，统一状态管理，以数据的变化来驱动界面的改变，更有利于数据的持久化和保存，同时也有利于UI组件的复用；
* **Hybrid Router**：主要解决Flutter和Native之间交叉跳转的问题，减少内存开销，共享同一个Flutter Engine。

### **工具介绍**

* **编译发布**：优化Flutter原有的编译逻辑，管理依赖Flutter原生依赖关联，打包Flutter和原生代码，实现自动化构建发布。
* **资源管理**：管理图片资源，将资源转换成Flutter类，便于资源的读取操作，类似Andorid的R类；
* **模版代码生成**：减少Flutter的代码编写，自动生成Flutter 组件的框架模板代码，提升代码编写效率；
* **JSON转换**：将JSON数据转换成Flutter code，并提供json转Flutter对象的API，减少动手编写Flutter code及解析。

### **JDFlutter业务开发实践**

JDFlutter为业务研发团队提供了全流程的开发解决方案：



### **配置混合工程**

Flutter和原生混合开发有两种情况，其一，开发Flutter业务的同学，需要和原生做交互，因此需要有Flutter和原生的混合编译环境；其二，使用原生SDK开发业务的同学，需要和Flutter业务一起集成打包，此时需对Flutter透明，以减少对Flutter编译环境的依赖，并且，只依赖原生编译环境即可，此时我们将Flutter编译成aar依赖，放入原生项目中即可。接下来，我们将重点介绍Android和iOS的混合编译环境配置。

### **Android平台配置**

创建一个flutter module

flutter create -t module --org com.example my\_flutter

在原生根项目的settings.gradle加入如下配置信息

// MyApp/settings.gradle include ':app' // assumed existing content setBinding(new Binding([gradle: this])) // new evaluate(new File( // new settingsDir.parentFile, // new 'my\_flutter/.android/include\_flutter.groovy' // new ))

在原生App模块中加入flutter依赖

dependencies { implementation project(':flutter') }

这样就可以原生项目一起编译了。具体可以参照官方文档：[http://github.com/flutter/flu](https://link.zhihu.com/?target=https://links.jianshu.com/go?to=http%3A%2F%2Fgithub.com%2Fflutter%2Fflu" \t "https://zhuanlan.zhihu.com/p/_blank)…这样的方式虽可以满足混编需求，但还不是特别方便，开发完项目后，还需要去Android Studio项目中进行编译，比较麻烦，所以我们也可以把Flutter项目settings.gradle改造，在Flutter开发环境下直接运行包含原生代码的混合项目，改造方式如下

// MyApp/settings.gradle //projectName 原生模块名称 //projectPath 原生项目路径 include ":$projectName" project(":$projectName").projectDir = new File("$projectPath")

这样改造之后即可在Flutter IDE中直接编译Flutter混合工程，并进行调试，也可以运行futter run来启动Flutter混合工程，不过在配置的时候，需要注意Flutter中 gradle编译环境和原生编译环境的一致性，如果不一致可能会导致编译错误。

### **iOS平台配置**

创建flutter module

flutter create -t module my\_flutter

进入iOS工程目录，初始化pod环境（如果项目工程已经使用Cocoapods，跳过此步骤）

pod init

编辑Podfile文件

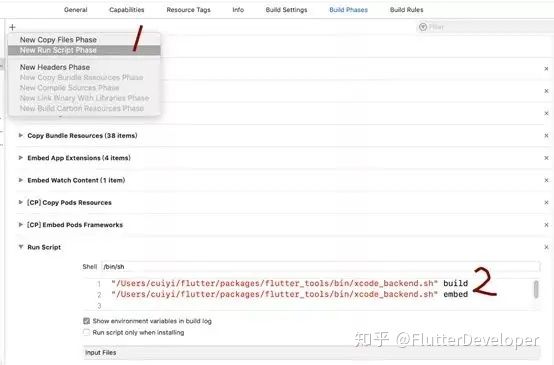
#在Podfile文件添加的新代码 flutter\_application\_path = '/{flutter module目录}/my\_flutter' eval(File.read(File.join(flutter\_application\_path, '.ios', 'Flutter', 'podhelper.rb')), binding)

安装pod

pod install

打开工程(\*\*\*.xcworkspace) 配置build phase，为编译Dart 代码添加编译选项打开iOS项目，选中项目的Build Phases选项，点击左上角+号按钮，选择New Run Script Phase，将下面的shell脚本添加到输入框中：

"$FLUTTER\_ROOT/packages/flutter\_tools/bin/xcode\_backend.sh" build "$FLUTTER\_ROOT/packages/flutter\_tools/bin/xcode\_backend.sh" embed



### **搭建PUB私服仓库**

Flutter开发中使用的组件，一般公司内部会采用共享的方式，以避免重复开发，而Flutter组件共享，即需要使用pub仓库。由于公司内部的业务组件不适合上传到pub官方仓库，因此，需要搭建私服仓库，以解决各个业务研发团队，对Flutter组件共享需要。感兴趣的同学可以研究下官方pub仓库的源码 [http://pub.dartlang.org/](https://link.zhihu.com/?target=https://links.jianshu.com/go?to=http%3A%2F%2Fpub.dartlang.org%2F" \t "https://zhuanlan.zhihu.com/p/_blank)，其对Google Cloud 环境有很大的依赖 ， 也可以基于[https://github.com/kahnsen/pub\_server](https://link.zhihu.com/?target=https://links.jianshu.com/go?to=https%3A%2F%2Fgithub.com%2Fkahnsen%2Fpub_server" \t "https://zhuanlan.zhihu.com/p/_blank)来搭建一个简易版本的私服仓库，以满足上传和下载功能，pub协议相对比较简单，我们可以在源码增加协议接口来实现更多功能。运行pub\_server

~ $ git clone [https://github.com/dart-lang/pub\_server.git](https://link.zhihu.com/?target=https://github.com/dart-lang/pub_server.git" \t "https://zhuanlan.zhihu.com/p/_blank) ~ $ cd pub\_server ~/pub\_server $ pub get ... ~/pub\_server $ dart example/example.dart -d /tmp/package-db Listening on http://localhost:8080 To make the pub client use this repository configure your shell via: $ export PUB\_HOSTED\_URL=http://localhost:8080

发布一个Flutter组件需要修改 pubspec.yaml，增加以下内容:

name: hello\_plugin //plugin名称 description: A new Flutter plugin. //介绍 version: 0.0.1//版本号 author: xxx <xxx@xxx.com>//作者和邮箱 homepage: https://localhost:8080 //组件的介绍页面 publish\_to: http://localhost:8080//仓库上传地址

上传时可以使用如下命令检查代码错误，并显示出上传的目录结构。

pub publish --dry-run

如果有不想上传的文件，可以在根目录增加一个.gitignore文件来忽略如下:

/build

Flutter组件的依赖配置，在项目的pubspec.yaml中dependencies:下增加如下信息：

dependencies: hello\_plugin: hosted: name: hello\_plugin url: http://localhost:8080 version: 0.0.2

这样可以在公司内部实现Flutter组件共享，如果不想搭建自己的pub仓库，也可以采用git依赖，配置如下:

dependencies: hello\_plugin: git: url: git://[http://github.com/hello\_plugin.git](https://link.zhihu.com/?target=http://github.com/hello_plugin.git" \t "https://zhuanlan.zhihu.com/p/_blank) //git地址 ref: dev-branch //分支

### **Flutter业务的开发与调试**

在Flutter IDE中编译代码调试会很方便，直接点击debug按钮即可进行代码调试，如果是混合工程在Android studio或者xcode中运行的工程，则没办法这么做，但也可以实现调试：将要调试的App安装到手机中（安装debug版本），连接电脑，执行如下命令，同步Flutter代码到设备的宿主App中

$ cd flutterProjectPath/ $ flutter attach

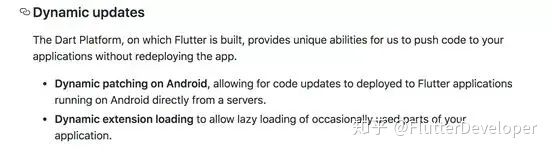
执行完命令后会进行等待设备连接状态，然后打开宿主App，进入Flutter页面，看到如下信息提示则表示同步成功

zbdeMacBook-Pro:example zb$ flutter attach Waiting for a connection from Flutter on MI 5X... Done. Syncing files to device MI 5X... 1.2s  [http://127.0.0.1:54422/](https://link.zhihu.com/?target=http://127.0.0.1:54422/" \t "https://zhuanlan.zhihu.com/p/_blank) For a more detailed help message, press "h". To detach, press "d"; to quit, press "q".

打开[http://127.0.0.1:54422](https://link.zhihu.com/?target=http://127.0.0.1:54422" \t "https://zhuanlan.zhihu.com/p/_blank)可以查看调试信息，如有代码改动可以按r来实时同步界面，如果改动没有实时生效可以按R重新启动Flutter应用。

### **JDFlutter热更新实践**

大部分跨端框架，诸如React Native / Weex / H5等，基本都能做到随时进行热修复，并随时上线，用于及时修复突发的在线问题，架构非常灵活。Flutter因其AOT的设计，预想会很难达到这种灵活度，但技术上仍具有一定的可行性，正如我们在之前的Flutter介绍文章中提到的，按照先有的API设计，是可以支持热修复的，但仅限于Android。官方最新的架构上已经支持了热修复架构，大家可以更新到1.2.1版本查看，但是官方的功能还比较弱，无法做到版本控制和回滚的灵活性，所以JDFlutter并没有采用。我们可以首先一起看一下Google官方热修复方案的设计原理：**Flutter1.2.1 版本引入了 Dynamic Patch**



为了更清楚的了解官方热修复的原理和过程，我们需要首先深入了解Flutter的业务包结构和整体运行过程：

### **Flutter App的包结构**



可以看到主体代码集中在asset目录中，除此之外还有少量Android端的框架java代码及flutter so引擎库外：

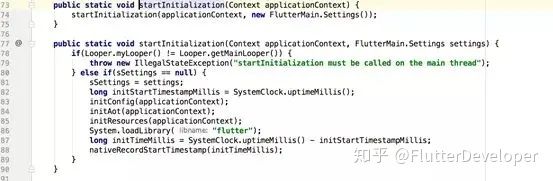
1、icudtl.dat

2、isolate\_snapshot\_data

3、isolate\_snapshot\_instr

### **Flutter包的初始化流程**

Flutter页面启动时是如何加载这些代码的呢？那就要从Flutter的初始化说起了，在页面启动前需要调用FlutterMain.startInitialization来做初始化：



可以看到该初始化是要求在主线程完成的，另外主要完成了以下三点：

* 配置了一些环境数据，比如各个核心包的路径，主要是提供给其他一些模块全局调用



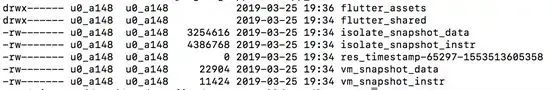
* 检查 asset 下 Flutter 包的完整性，主要是上面介绍的一些核心包，一旦缺少核心的一些库，就会直接抛异常。开发过程中我们经常因为配置导致有些文件没有打包进去，然后会直接 crash，就是在这里触发的，具体代码如下：

IMG_264

* 解压部分 asset 下的资源到 data 分区，以下是一些片段的代码，那为什么要解压呢？放在 asset 下也是可以通过 assetManager 读取的。这里 google 应该是从性能角度要求解压的，因为频繁的使用 assetManager 读取 asset 是很容易造成多线程阻塞的，一旦阻塞了将会导致整个 Flutter 业务全部无法渲染，所以需要解压一些核心的资源库，而不是解压了所有的资源 (例如图片就没有解压)



从代码来看，先增加要解压的核心库的目录，然后启动 task 从 asset 中解压库到 data 分区对应 app 数据下的 app\_flutter 目录，以下是解压后的目录结构：



其中 res\_timestamp 文件用于标记一些时间戳，算法比较固定，根据客户端的安装时间及 app 的 version code 生成，也就是说当用户打开 Flutter 页面后这个值就是固定的，如果有任何修改引擎会默认有变化，删除现有 app\_flutter 的包，重新解压



### **运行原理**

上面是对Flutter程序加载的分析，最终Flutter页面显示是需要呈现在原生组件Flutter View中的，这个组件会和底层Flutter Native View 进行绑定，并最终运行上面说到的data分区的Dart代码来渲染UI。如果使用的是Flutter Activity，则默认Flutter View是全屏显示，如需要定制页面，需要自己设计Activity。



### **热修复实验**

了解了这些，其实热修复方案已经呼之欲出，替换原有解压后的app\_flutter包，杀进程，然后重新加载Flutter页面即可。这里我们可以做个简单的实验：采用adb命令push一些修改过的并编译的dart代码到app\_flutter目录：

* 先打开Flutter页面，默认会加载asset下的包，并解压到data分区
* 修改一个Flutter工程，并编译代码，最终在工程目录my\_flutter/.android/Flutter/build/intermediates/flutter/release中看到打包生成的文件



* 这么文件目录中只有 flutter\_assets 目录和 isolate\_snapshot\_data 文件是包含业务代码和图片的，其他部分基本不会变化，所以我们这里要替换的目录也就是这两个，大家可以使用 adb push 命令将资源文件 push 到对应的 data 分区来做个实验。

adb push my\_flutter/.android/Flutter/build/intermediates/flutter/release/isolate\_snapshot\_data /data/data/app包名 /app\_flutter

* 关闭 Flutter 页面，在 Task 中杀掉进程，回来后重新打开 Flutter 页面，就能看到改动的效果，图片资源是存放在 flutter\_asset 目录的，将图片放到这个目录，同样能更新图片

上面这个实验，验证了方案基本是可行的，但这里只是简单替换，实际使用中替换还是有很多问题的。那 Google 官方是如何设计的呢？

### **Google热修复设计**

### **热修复步骤**

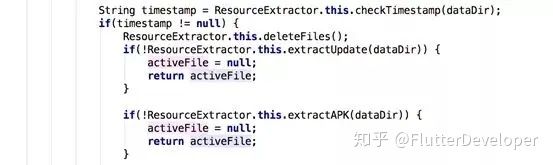
Flutter SDK 1.2.1中，Google提供了ResourceUpdater，用来做包的检查和下载解压。升级步骤如下：

* 在页面初始化时，检查固定的下载更新目录有没有业务升级包，从代码来看，必须在manifest中打开该功能，设置DynamicPatching



从逻辑上来看，只有在页面 onResume 或者 App 重新开启的时候会下载升级包，整体下载是通过 http 请求完成的，整体实现代码大家可以参考 ResourceUpdater 中 DownloadTask 的实现部分，这里就不细说了。

* 每次 init 的时候都会触发检查 data 分区的 app\_flutter 包，如果不存在就会从 aaset 目录解压出来，而升级包的替换就是在这步完成的，按照逻辑会优先检查升级目录有没有包存在，如果存在则优先从升级目录解压，如果不存在还是从 asset 目录解压；



* 当然在检查到有升级包时，会对升级包的一些配置做校验，主要是 manifest.json 文件，里面会包含 buildNumber/baselineChecksum 字段，同时也会对"isolate\_snapshot\_data", "isolate\_snapshot\_instr", "flutter\_assets/isolate\_snapshot\_data"等文件做 CRC32 校验。



* 升级后的版本时间戳是从配置的 manifest.json 文件中读取 patchNumber 和文件下载时间确定的，完成文件覆盖后会重新生成。

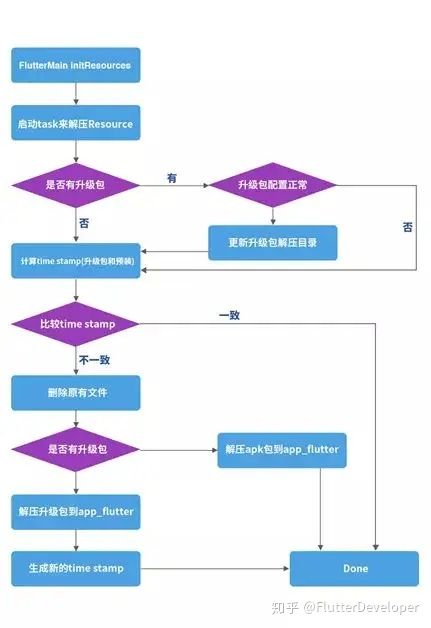
以下是升级包的大概路径如下：



### **如何配置服务器**

文章上部分介绍了怎么打开升级patch的功能，因升级涉及到服务端，那Google是怎么做到关联到服务器的呢？其实原理比较简单，需要配置客户端的manifest文件的meta属性，增加PatchServerURL，也就是我们服务的地址，以及下载模式PatchDownloadMode和加载模式PatchInstallMode，默认是ON\_NEXT\_RESTART（下次初始化时）

### **整体流程**



### **存在的缺陷**

* 过于定制化，全部在引擎完成，很难适配一些特殊的需求定制；
* 不支持现在比较主流的升级流程，诸如灰度和白名单等功能；
* 版本号的维度不好控制，同时不能做版本回滚等操作。

### **JDFlutter如何实现热修复**

### **实现原理**

JDFlutter的整体实现原理，其实和Google是一样的，目前来看不修改引擎的前提下，只有这种方案最简单，但是我们没有使用Google的这套升级架构，默认关闭了patch功能，并框架之外实现了替换包和加载的逻辑，优点是整体兼容性更强、更灵活。1、服务端根据客户端的唯一标识支持了白名单和灰度下发升级包；2、优化下载和替换流程。Flutter的升级包一般有4-5M，而且从网络端获取，失败率较高，替换过程又涉及到文件操作，操作不当容易产生UI阻塞或者包异常。接入JDFlutter的客户端下载包后，并不会直接替换文件，而是修改名称后解压到app\_flutter目录，等待业务页面重新打开或者重新初始化时再修改成Flutter标准名称的文件。这种操作不存在性能问题，另外会把旧版的文件备份，以便回滚代码；3、同时并发运行的Flutter页面较多，需避免因为升级出现一些中间状态，使得业务或者页面无法打开的情况；4、升级失败或者下载后业务包有问题，出现无法加载的情况或者文件丢失的情况可以控制回滚代码；5、线上出现大量异常后，可以指定对应的Flutter业务执行降级策略，让该业务迅速降级到H5页面。

### **热修复规划**

未来，JDFlutter会继续在热修复方面进行探索和验证，以满足京东业务的快速发展需要。而针对目前的方案，我们思考了如下的优化点：Flutter业务包差量升级：现有的升级模式都是全量包覆盖，即使压缩后升级包还是很大，影响升级成功率及用户流量，后续会采用一些diff工具，对比生成差量的patch，通过服务端下发后，在客户端合并成完整包，但升级次数较多后会导致最终版本碎片化，需要做好版本之前的维护关系，难度较大。

升级后及时更新页面：现有方案（包括标准google升级方案）没有办法做到下载业务包或者替换业务包后及时刷新页面，需要restart进程后重新开启才能刷新页面。未来我们会优化引擎，通过释放底层资源并重新加载，来完成随时刷新页面的功能。

### **未来展望**

Google Flutter是非常出色的跨端开发技术，现在已经取得了长足的发展。社区生态和框架成熟度也正在快速追赶RN。相信不久的将来，Flutter+RN一定会成为跨端开发平台的绝代双骄。