CaffreySun
2019年01月13日 阅读 1634

从零搭建 iOS Native Flutter 混合工程

本文来实现一个灵活、无侵入、低耦合的 iOS Flutter 混合工程。 我们希望混合开发至少得保证如下特点：

- 对Native工程无侵入
- 对Native工程零耦合
- 不影响Native工程的开发流程与打包流程
- 易本地调试

一、Flutter 提供的 Native Flutter 混合工程方式

Flutter 官方提供的混合工程搭建方法： Add Flutter to existing apps 文章中介绍了如何在现有 App 里加入Flutter，下面进行逐步介绍一下

1. 创建 Flutter 工程

请自行 百度/Google Flutter 安装教程，安装Flutter。然后到任意目录下执行 `flutter create -t module my_flutter` ， "my_flutter" 是要创建的 Flutter 工程的名称。

2. 通过 Cocoapods 将 Flutter 引入 现有 Native 工程

在Podfile添加以下下代码

```ruby
flutter_application_path = "xxx/xxx/my_flutter"
eval(File.read(File.join(flutter_application_path, '.ios', 'Flutter', 'podhelper.rb')), bin
```

然后执行 `pod install`

3. 修改 Native 工程

```Shell
"$FLUTTER_ROOT/packages/flutter_tools/bin/xcode_backend.sh" build
"$FLUTTER_ROOT/packages/flutter_tools/bin/xcode_backend.sh" embed
```

二、分析 Native Flutter 混合工程

按照上面三个步骤进行逐一分析每一步的问题，并提供优化方案。

1. 创建 Flutter 工程

这一步首先在自己电脑上安装 Flutter，然后使用 `flutter create` 。这里就存在问题，在团队开发中每个人安装的 Flutter 版本可能并不同，这样会出现Dart层Api兼容性或Flutter虚拟机不一致等问题。在团队协作中一定要保证 Flutter 工程依赖相同的 Flutter SDK，所有需要一个工具在执行 `flutter` 指令时可以根据当前 Flutter 工程使用对应版本的 Flutter SDK，目前有一个名叫flutter_wrapper的工具，使用 `flutterw` 代替 `flutter` 指令，工具会自动将 Flutter SDK 放在当前 Flutter 工程目录下，并执行当前工程中的 `flutter` 命令，这样就不再依赖本地电脑安装的 Flutter SDK。

flutter_wrapper使用：

1. `flutter create` 创建 Flutter 工程，这里使用的是本地的 Flutter SDK
2. 进入 Flutter 工程目录安装 'flutter_wrapper'，执行 `sh -c "$(curl -fsSL https://raw.githubusercontent.com/passsy/flutter_wrapper/master/install.sh)"`
3. 此后在当前 Flutter 工程需要使用 `flutter` 命令的地方都使用 `./flutterw` 来代替

2. 通过 Cocoapods 将 Flutter 引入 现有 Native 工程

这一步在 Podfile 里添加里一个 'podhelper.rb' ruby 脚本，脚本会在 `pod install/update` 时执行，脚本主要做4点事情：

1. 解析 'Generated.xcconfig' 文件，获取 Flutter 工程配置信息，文件在'my_flutter/.ios/Flutter/'目录下，文件中包含了 Flutter SDK 路径、Flutter 工程路径、Flutter 工程入口、编译目录 等。
2. 将 Flutter SDK 中的 `Flutter.framework` 通过 pod 添加到 Native 工程。
3. 将 Flutter 工程依赖的插件通过 pod 添加到 Native 工程，因为有些插件有 Native 部分代码。
4. 使用 `post_install` 这个 pod hooks 来关闭 Native 工程的 bitcode，并将 'Generated.xcconfig' 文件加入 Native 工程。

频繁修改'flutter_application_path'变量，这样很不友好。

解决这个问题的思路就是将 Flutter 工程放在当前 Native 工程的目录下， 我们可以再加入一个 ruby 脚本，在每次执行 `pod install/update` 时，将 Flutter 工程从 git 上拉取一份放在当前目录下，这样 Flutter 工程的路径就统一了。大致代码如下：

```ruby
flutter_application_path = __dir__ + "/.flutter/app"
`git clone git://xxxx/my_flutter.git #{flutter_application_path}`
# 如果想要调试本地的 Flutter 工程，就把下面这行注释放开
# flutter_application_path = "xxx/xxx/my_flutter"
eval(File.read(File.join(flutter_application_path, '.ios', 'Flutter', 'podhelper.rb')), bin
```

上述代码只是临时代码，为了演示将 Flutter 工程放在当前目录下这个思路，后面会有完整的实现代码。

3. 修改 Native 工程

这里执行了一个' xcode_backend.sh'脚本的两个命令build、embed，两个命令分别的作业是：

- build: 根据当前 Xcode 工程的 'configuration' 和其他编译配置编译 Flutter 工程，'configuration'通常为'debug'或者'release'
- embed: 将 build 出来的 framework、资源包放入 Xcode 编译目录，并签名 framework

这里存在的问题是：Flutter 工程依赖 Native工程来执行编译，并影响Native工程的开发流程与打包流程。

通常 'configuration' 里不止有 'debug' 或者 'release'，可能会有自定义的名称，如果我们使用自定义的 'configuration' 编译，那么 `xcode_backend.sh build` 就会执行失败。因为Flutter 编译模式是通过 'configuration' 获取的，Flutter 支持 Debug、Profil、Release 三种编译模式，而我们自定义的名称不在这三种之中，Flutter 就不知道该怎么编译。

每次 Native 编译时 Flutter 就需要编译，其实是产生了相互依赖：Flutter 编译依赖 Native 编译环境，Native 依赖 Flutter 编译通过。

我们希望做到：Native 依赖 Flutter 编译出来的产物，并且保留依赖 Flutter 源码进行调试的能力。

实现这个目标我们需要两部分：

赖 Flutter 工程源码的 功能。

三、实现 Native Flutter 混合工程

下面我们来实现上文提到的两个部分

第一部分实现“打包脚本”

这一部分我们需要实现脚本自动打包 Flutter 工程，拆分一下这个脚本流程，大致分为一下 几个步骤：

1. 检查 Flutter 环境，拉取 Flutter plugin
2. 编译 Flutter 工程产物
3. 复制 Flutter 插件中的 Native 代码
4. 将产物同步到产物发布的服务器

下面来一步一步的分析并实现每一步：

(1) 检查 Flutter 环境，拉取 Flutter plugin

这一步做的工作是检查是否安装了 'flutter_wrapper'，如果安装则进行安装， 然后执行 `./flutterw packages get` ，Shell代码如下：

```Shell
flutter_get_packages() {
    echo "================================="
    echo "Start get flutter app plugin"

    local flutter_wrapper="./flutterw"
    if [ -e $flutter_wrapper ]; then
        echo 'flutterw installed' >/dev/null
    else
        bash -c "$(curl -fsSL https://raw.githubusercontent.com/passsy/flutter_wrapper/mast
        if [[ $? -ne 0 ]]; then
            # 上一步脚本执行失败
            echo "Failed to installed flutter_wrapper."
            exit -1
        fi
    fi
```

```
        echo "Failed to install flutter plugins."
        exit -1
    fi

    echo "Finish get flutter app plugin"
}
```

(2) 编译 Flutter 工程产物

这一步是脚本的核心， 主要逻辑和上文中' xcode_backend.sh build'类似，大致 代码如下：

Shell

```
# 默认debug编译模式
BUILD_MODE="debug"
# 编译的cpu平台
ARCHS_ARM="arm64,armv7"
# Flutter SDK 路径
FLUTTER_ROOT=".flutter"
# 编译目录
BUILD_PATH=".build_ios/${BUILD_MODE}"
# 存放产物的目录
PRODUCT_PATH="${BUILD_PATH}/product"
# 编译出的flutter framework 存放的目录
PRODUCT_APP_PATH="${PRODUCT_PATH}/Flutter"

build_flutter_app() {
    echo "================================="
    echo "Start Build flutter app"
    # 创建目录
    mkdir -p -- "${PRODUCT_APP_PATH}"
    # flutter 工程入口 dart文件
    local target_path="lib/main.dart"
    # flutter sdk 目录解析
    local artifact_variant="unknown"
    case "$BUILD_MODE" in
    release*)
        artifact_variant="ios-release"
        ;;
    profile*)
        artifact_variant="ios-profile"
        ;;
    debug*)
        artifact_variant="ios"
        ;;
```

```
    ;;
esac

if [[ "${BUILD_MODE}" != "debug" ]]; then
    # 非debug编译模式
    # build fLutter app, output App.framework
    ${FLUTTER_ROOT}/bin/flutter --suppress-analytics \
        --verbose \
        build aot \
        --output-dir="${BUILD_PATH}" \
        --target-platform=ios \
        --target="${target_path}" \
        --${BUILD_MODE} \
        --ios-arch="${ARCHS_ARM}"

    if [[ $? -ne 0 ]]; then
        echo "Failed to build flutter app"
        exit -1
    fi
else
    # debug编译模式直接使用编译好的App.framework,
    # 因为在 debug 模式下 flutter 代码并没有编译成二进制机器码，而是在后续build bundle时被打包进资
    # 在'xcode_backend.sh'脚本里，这一步这里只是编译成一个App.framework空壳。
    # 提前编译好的原因是'xcode_backend.sh'脚本执行和Xcode一起执行，所以执行时能获取到Xcode设置的编
    # 而本脚本不依赖Xcode执行，即便把'xcode_backend.sh'对应的代码拷贝出来也不能正确的编译出'App.fr
    #
    # 而我又不想那么麻烦，选择另辟蹊径:
    # 随便创建了一个 Flutter 工程,
    # 在debug模式下，先在模拟器编译运行一下，得到x86_64的App.framework,
    # 再到真机运行一下，得到arm64/armv7的App.framework,
    # 最后使用lipo命令将两个App.framework合并，得到x86_64/arm64/armv7的App.framework,
    # 这样最后得到的App.framework在模拟器和真机都可以用
    # 因为debug模式下App.framework就是占位的空壳，所以其他flutter工程一样用
    local app_framework_debug="iOSApp/Debug/App.framework"
    cp -r -- "${app_framework_debug}" "${BUILD_PATH}"
fi

# copy info.plist to App.framework
app_plist_path=".ios/Flutter/AppFrameworkInfo.plist"
cp -- "${app_plist_path}" "${BUILD_PATH}/App.framework/Info.plist"

local framework_path="${FLUTTER_ROOT}/bin/cache/artifacts/engine/${artifact_variant}"
local flutter_framework="${framework_path}/Flutter.framework"
local flutter_podspec="${framework_path}/Flutter.podspec"

# copy framework to PRODUCT_APP_PATH
```

```
    local precompilation_flag=""
    if [[ "$BUILD_MODE" != "debug" ]]; then
        precompilation_flag="--precompiled"
    fi

    # build bundle
    ${FLUTTER_ROOT}/bin/flutter --suppress-analytics \
        --verbose \
        build bundle \
        --target-platform=ios \
        --target="${target_path}" \
        --${BUILD_MODE} \
        --depfile="${BUILD_PATH}/snapshot_blob.bin.d" \
        --asset-dir="${BUILD_PATH}/flutter_assets" \
        ${precompilation_flag}

    if [[ $? -ne 0 ]]; then
        echo "Failed to build flutter assets"
        exit -1
    fi

    # copy Assets
    local product_app_assets_path="${PRODUCT_APP_PATH}/Assets"
    mkdir -p -- "${product_app_assets_path}"
    cp -rf -- "${BUILD_PATH}/flutter_assets" "${PRODUCT_APP_PATH}/Assets"

    # setting podspec
    # replace:
    # 'Flutter.framework'
    # to:
    # 'Flutter.framework', 'App.framework'
    #   s.resource='Assets/*'
    sed -i '' -e $'s/\'Flutter.framework\'/\'Flutter.framework\', \'App.framework\'\\\n  s.

    echo "Finish build flutter app"
}
```

(3) 复制 Flutter 插件中的 Native 代码

Flutter 使用的各种插件可能会包含 Native 代码，并且这些代码已经提供了podspec，可以使用 pod 直接引入。我们要做的就是把插件的 Native 代码拷贝到产物目录。 Flutter 创建了一个给 Native 注册插件的 pod 库 'FlutterPluginRegistrant'，这个也需要拷贝出来， 在 Flutter 工程根目录下有一

```Shell
flutter_copy_packages() {
    echo "================================="
    echo "Start copy flutter app plugin"

    local flutter_plugin_registrant="FlutterPluginRegistrant"
    local flutter_plugin_registrant_path=".ios/Flutter/${flutter_plugin_registrant}"
    echo "copy 'flutter_plugin_registrant' from '${flutter_plugin_registrant_path}' to '${P
    cp -rf -- "${flutter_plugin_registrant_path}" "${PRODUCT_PATH}/${flutter_plugin_registr

    local flutter_plugin=".flutter-plugins"
    if [ -e $flutter_plugin ]; then
        OLD_IFS="$IFS"
        IFS="="
        cat ${flutter_plugin} | while read plugin; do
            local plugin_info=($plugin)
            local plugin_name=${plugin_info[0]}
            local plugin_path=${plugin_info[1]}

            if [ -e ${plugin_path} ]; then
                local plugin_path_ios="${plugin_path}ios"
                if [ -e ${plugin_path_ios} ]; then
                    if [ -s ${plugin_path_ios} ]; then
                        echo "copy plugin 'plugin_name' from '${plugin_path_ios}' to '${PRO
                        cp -rf ${plugin_path_ios} "${PRODUCT_PATH}/${plugin_name}"
                    fi
                fi
            fi
        done
        IFS="$OLD_IFS"
    fi

    echo "Finish copy flutter app plugin"
}
```

(4) 将产物同步到保留产物的服务器

经过上面的几个步骤后会生成一个产物目录，这个目录下会有几个二级目录，每个二级目录里都包含一个 podspec 文件。

也就是说这个产物目录里存放的就是 cocoapods 库，将目录拷贝到 Native 工程，然后用 `pod 'pod_name', :path=>'xx/xxx'` 的形式引用就可以了。

是为了和 Android Flutter 产物放在一个地方，并且 Maven 已成做好的产物版本管理。

Maven上传代码比较简单，这里不再赘述，有兴趣可以到文末的github仓库查看代码。

Flutter 工程版本设置是在工程目录下的 'pubspec.yaml' 文件，打包脚本读取这个文件来确定产物的版本。

最后这个脚本使用方式为 `./build_ios.h -m debug` `./build_ios.h -m release`，上文中没有提到的一点是只有 release 模式编译的包才会上传的服务器，debug 只是编译到产物目录。

第二步 Native 依赖 Flutter 产物

这部分我们需要实现获取指定版本 Flutter 工程 release 产物，并集成到 Native 项目，并保留可以调试 Flutter 工程的能力。

也是来拆分一下脚本流程：

- 获取 Flutter 工程产物
  - 获取 release 产物
  - 获取 debug 产物
- 通过 pod 引入 Flutter 工程产物

(1) 获取 Flutter 工程产物

上文说到只有 release 产物放在了产物服务器上，debug 只是编译到产物目录。不上传 debug 的原因是，debug 阶段就是开发阶段，举个不太恰当的例子：哪有开发阶段就把包上传 app store 的？ 也就代表这 release 的产物和 debug 的产物获取逻辑不一样，并且我们的脚本支持两种方式 的切换，所以在 Podfile 添加如下代码：

Ruby

```ruby
# 设置要引入的 flutter app 的版本
FLUTTER_APP_VERSION="1.1.1"

# 是否进行调试 flutter app,
# 为true时FLUTTER_APP_VERSION配置失效，下面的三项配置生效
# 为false时FLUTTER_APP_VERSION配置生效，下面的三项配置失效
FLUTTER_DEBUG_APP=false

# Flutter App git地址，从git拉取的内容放在当前工程目录下的.flutter/app目录
# 如果指定了FLUTTER_APP_PATH，则此配置失效
```

```
FLUTTER_APP_BRANCH="master"

# flutter本地工程目录，绝对路径或者相对路径，如果有值则git相关的配置无效
FLUTTER_APP_PATH="../my_flutter"

eval(File.read(File.join(__dir__, 'flutterhelper.rb')), binding)
```

Podfile 其实就是 Ruby 代码，上面几个由大写字母组成的变量是全局变量，最后一句代码的作用为读取'flutterhelper.rb'里的代码并执行，在'flutterhelper.rb'里可以获取到上面定义的全局变量，根据这几个变量做不同的操作，其中选择使用 release 还是 debug 的代码如下：

Ruby

```
if FLUTTER_DEBUG_APP.nil? || FLUTTER_DEBUG_APP == false
    # 使用 flutter release 模式
    puts "开始安装 release mode flutter app"
    install_release_flutter_app()
else
    # 存在debug配置，使用 flutter debug 模式
    puts "开始安装 debug mode flutter app"
    install_debug_flutter_app()
end
```

`install_release_flutter_app` 为操作 release 产物的函数， `install_debug_flutter_app` 为操作 debug 产物的函数。

处理 release 模式主要就是获取 release 产物，代码如下：

Ruby

```
# 安装正式环境环境app
def install_release_flutter_app
    if FLUTTER_APP_VERSION.nil?
        raise "Error: 请在 Podfile 里设置要安装的 Flutter app 版本 ，例如: FLUTTER_APP_VERSION='
    else
        puts "当前安装的 flutter app 版本为 #{FLUTTER_APP_VERSION}"
    end

    # 存放产物的目录
    flutter_release_path = File.expand_path('.flutter_release')
    # 是否已经存在当前版本的产物
    has_version_file = true
    if !File.exist? flutter_release_path
        FileUtils.mkdir_p(flutter_release_path)
        has_version_file = false
    end
```

```
    if !File.exist? flutter_release_version_path
        FileUtils.mkdir_p(flutter_release_version_path)
        has_version_file = false
    end

    # 产物包
    flutter_package = "flutter.zip"
    flutter_release_zip_file =  File.join(flutter_release_version_path, flutter_package)
    if !File.exist? flutter_release_zip_file
        has_version_file = false
    end

    # 产物包下载完成标志
    flutter_package_downloaded = File.join(flutter_release_version_path, "download.ok")
    if !File.exist? flutter_package_downloaded
        has_version_file = false
    end

    if has_version_file == true
        # 解压
        flutter_package_path = unzip_release_flutter_app(flutter_release_version_path, flut
        # 开始安装
        install_release_flutter_app_pod(flutter_package_path)
    else
        # 删除老文件
        FileUtils.rm_rf(flutter_release_zip_file)
        # 删除标志物
        FileUtils.rm_rf(flutter_package_downloaded)

        # 下载
        download_release_flutter_app(FLUTTER_APP_VERSION, flutter_release_zip_file, flutter
        # 解压
        flutter_package_path = unzip_release_flutter_app(flutter_release_version_path, flut
        # 开始安装
        install_release_flutter_app_pod(flutter_package_path)
    end
end
```

`unzip_release_flutter_app` 为解压zip格式产物的函数， `download_release_flutter_app` 为从 Maven 下载产物的函数，这两个比较简单，详细代码请看文末 github 仓库。
`install_release_flutter_app_pod` 为通过 pod 将产物添加到 Native 的函数，后面会详细介绍。

处理 debug 模式的操作为，获取 Flutter 工程源代码，执行 `build_ios.sh -m debug` 进行打包后得到 debug 产物目录，详细代码如下：

```
puts "如果是第一次运行开发环境Flutter项目，此过程可能会较慢"
puts "请耐心等待☕☕☕\n"

# 默认Flutter App 目录
flutter_application_path = __dir__ + "/.flutter/app"
flutter_application_url = ""
flutter_application_branch = 'master'

# 指定了FLUTTER_APP_PATH就用本地代码，复制从git拉取
if FLUTTER_APP_PATH != nil
    File.expand_path(FLUTTER_APP_PATH)
    if File.exist?(FLUTTER_APP_PATH)
        flutter_application_path = FLUTTER_APP_PATH
    else
        flutter_application_path = File.expand_path(FLUTTER_APP_PATH)
        if !File.exist?(flutter_application_path)
            raise "Error: #{FLUTTER_APP_PATH} 地址不存在!"
        end
    end

    puts "\nFlutter App路径: "+flutter_application_path
else
    if FLUTTER_APP_URL != nil
        flutter_application_url = FLUTTER_APP_URL
        if FLUTTER_APP_BRANCH != nil
            flutter_application_branch = FLUTTER_APP_BRANCH
        end
    else
        raise "Error: 请在'Podfile'里增加Flutter App git地址配置，配置格式请查看'flutterhelpe
    end
    puts "\n拉取 Flutter App 代码"
    puts "Flutter App路径: "+flutter_application_path
    update_flutter_app(flutter_application_path, flutter_application_url, flutter_appli
end

puts "\n编译 Flutter App"
# PUB_HOSTED_URL FLUTTER_STORAGE_BASE_URL 为了加快速度，使用国内镜像地址
`export PUB_HOSTED_URL=https://pub.flutter-io.cn && \
export FLUTTER_STORAGE_BASE_URL=https://storage.flutter-io.cn && \
cd #{flutter_application_path} && \
#{flutter_application_path}/build_ios.sh -m debug`

if $?.to_i == 0
    flutter_package_path = "#{flutter_application_path}/.build_ios/debug/product"
    # 开始安装
    install_release_flutter_app_pod(flutter_package_path)
```

```
end
```

`update_flutter_app` 为从 git 拉取代码的函数也不赘述，详情见文末 github 仓库。

(2) 通过 pod 引入 Flutter 工程产物

上文两个函数执行完成后，就得到了产物的存放目录，下面只需要引入到 Native 仓库就可以了，也就是 `install_release_flutter_app_pod` 函数，从代码如下：

Ruby

```ruby
# 将 Flutter app 通过 pod 安装
def install_release_flutter_app_pod(product_path)
    if product_path.nil?
        raise "Error: 无效的 flutter app 目录"
    end

    puts "将 flutter app 通过 pod 导入到 工程"

    Dir.foreach product_path do |sub|
        if sub.eql?('.') || sub.eql?('..')
            next
        end

        sub_abs_path = File.join(product_path, sub)
        pod sub, :path=>sub_abs_path
    end

    post_install do |installer|
        installer.pods_project.targets.each do |target|
            target.build_configurations.each do |config|
                config.build_settings['ENABLE_BITCODE'] = 'NO'
            end
        end
    end
end
```

如果要修改 release 产物版本，则设置 `FLUTTER_APP_VERSION` 。 如果想要 debug flutter，则设置 `FLUTTER_DEBUG_APP=true` ，如果调试本地代码则设置 `FLUTTER_APP_PATH="../my_flutter"` ，负责将 `FLUTTER_APP_PATH` 注释掉，配置 `FLUTTER_APP_URL` `FLUTTER_APP_BRANCH` 。

四、总结

- Flutter 工程完全不依赖 Native 工程，而是通过 'build_ios.sh' 脚本进行编译打包；
- 通过 pod 引入 Flutter 工程对 Native 也没有浸入，不要在 Native 工程里增加 Flutter 打包脚本；
- Native 开发工程师只需要执行 `pod install` 所有的 Flutter 依赖就都加入进工程，不需要工程师配置 Flutter 开发环境；也不影响 Native 打包；
- 也保留了本地调试 Flutter 工程的功能；

Github 仓库：iOS_Flutter_Hybrid_Project