蚂蚁科技产品手册

实时发布

产品版本： V20210430

文档版本： V20210430

蚂蚁科技技术文档

商标声明

及其他蚂蚁科技服务相关的商标均为蚂蚁科技所有。
本文档涉及的第三方的注册商标，依法由权利人所有。

免责声明

由于产品版本升级、调整或其他原因，本文档内容有可能变更。蚂蚁科技保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在蚂蚁科技授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过蚂蚁科技授权渠道下载、获取最新版的用户文档。如因文档使用不当造成的直接或间接损失，本公司不承担任何责任。

目录

1 实时发布简介

实时发布服务（Mobile Delivery Service，简称 MDS）是 mPaaS 平台的核心基础服务组件之一，提供版本升级包、热修复包、H5 离线包、小程序包的管理和发布服务，同时支持 开关配置 、 白名单 、 发布规则 管理功能。

在客户端集成实时发布服务功能后，您可以在 mPaaS 插件中生成新的包，然后在实时发布控制台发布新包，客户端收到新包并进行升级。实时发布服务还支持通过白名单进行灰度发布，您也可以使用高级过滤规则，比如指定机型，来进行更精准的灰度发布。

功能特性

灰度发布

在正式发布之前，可以通过白名单来做小规模发布（比如内部员工）以验证新包的功能是否达到预期。还可以进行时间窗灰度发布，在规定的时间段内发布给规定用户人数。如果达到预期就可以进行全网推送。

高级过滤

在进行灰度发布的时候还可以利用高级规则来定义更为精准的白名单人群，比如可以只发给小米手机的用户，多个过滤规则可以叠加，只有在所有的过滤规则都符合的情况下才会推送。

实时回滚

仅支持热修复。即使进行了灰度发布，正式上线的时候还是难免会发生问题，这个时候就可以进行实时回滚，自动回滚到发布前的版本。

自定义验签

为了保障安全性，热修复有自定义的验签流程，保证脚本来源的正确性。 mPaaS 插件中提供生成热修复资源包并对包进行加签的功能。

产品优势

支持多产品、多任务、多维度发布管理

多 APP 支持，同时支持正式升级、热修复及 H5 离线包以及实时在线推送。

智能灰度能力，多种升级策略

内部灰度、外部灰度、人群地域、机型网络等多种规则可供选择。

增量差分离线包更新能力

减少数据冗余及设备带宽占用，在移动端网络条件不稳定场景下可体现优势。

高灵敏度、高可用性

升级客户端 RPC 接口能力，可用率可达 99.999%，提供在线分钟级触达能力。

系统高性能保障

触达率达 99.999%，日 UV 支持 2 亿 +。

2 实时发布流程

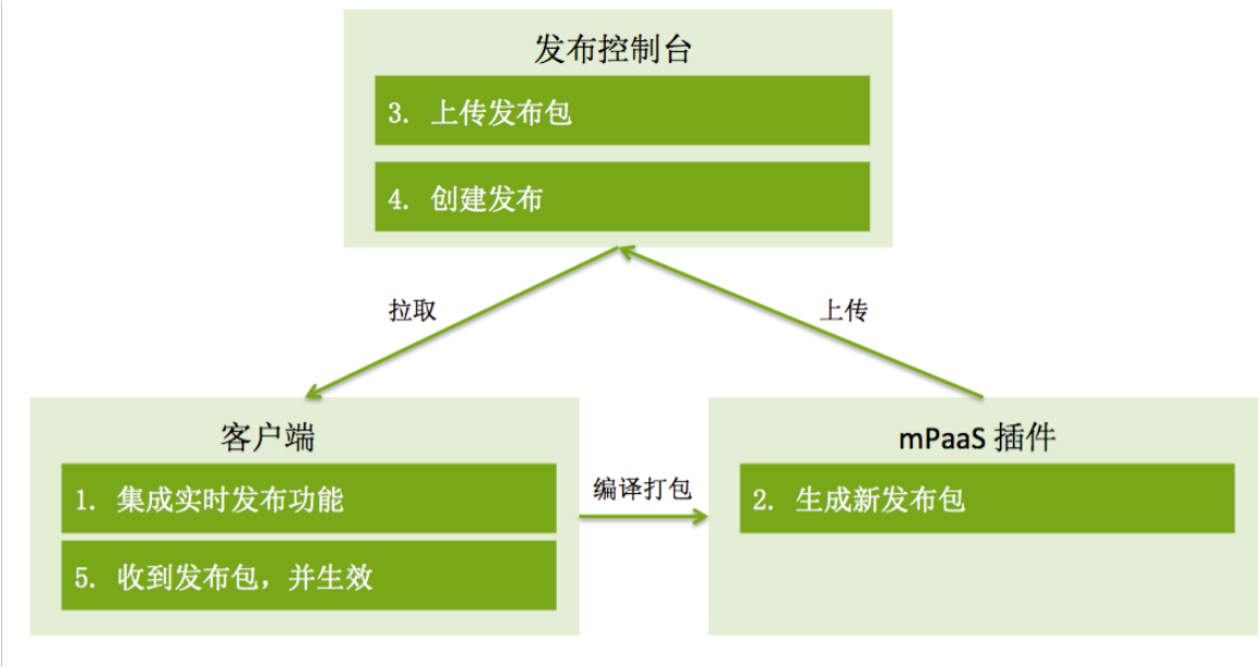
实时发布平台包含了客户端 SDK，您可以便捷地把实时发布的能力集成到客户端。

1. 在 mPaaS 插件中打包生成版本升级包、热修复包、离线包、小程序包等，上传到发布控制台。
2. 根据不同的发布策略进行灰度发布、正式发布等。
3. 客户端再去拉取新的发布包进行升级、热修复、离线发布。

另外，您还可以使用开关配置服务修改客户端代码处理逻辑。通过在控制台增加需要的开关配置项，实现针对性地下发。

使用流程

以下图表展示了实时发布版本升级包、热修复包、离线包的流程：



控制台管理

您可以在实时发布控制台完成以下操作：

- 版本升级包 > 发布管理 ：管理发布客户端新版本的配置。
- 热修复包 > 热修复管理 ：在不发版本的情况下直接修复线上 bug。
- 离线包 > 离线包管理 ：将不同的业务封装打包成离线包，通过发布平台下发，对客户端资源进行更新。
- 小程序包 > 小程序包管理 ：实现小程序包的发布及相关配置。
- 开关配置 > 配置管理 ：实现各种开关的配置、修改、推送。可以按平台、白名单、百分比等进行有针对性地下发。

- 白名单管理 ：为实时发布提供一个白名单的管理平台，用户可以轻松创建十万级的白名单数据供实时发布使用。
- 发布规则管理 ：预先定义实时发布所需要的各种配置数据，无需每次手工输入，提升效率，降低出错可能性。

3 发布管理

3.1 接入 Android

3.1.1 快速开始

本文介绍如何添加与发布管理功能相关的 **升级** SDK。添加 SDK 并完成相关配置后，在 mPaaS 控制台发布 App 的新版本，客户端可以通过升级接口检测到该新版本，进而提醒用户下载更新。

目前，升级 SDK 支持 **原生 AAR 接入**、**mPaaS Inside 接入** 和 **组件化接入** 三种接入方式。

整个过程分为以下四步：

1. 添加 SDK
2. 工程配置
3. 初始化 mPaaS （仅原生 AAR 接入或 mPaaS Inside 接入需要）
4. 升级检测

前置条件

- 若采用原生 AAR 方式接入，需要先 将 mPaaS 添加到您的项目中 。
- 若采用 mPaaS Inside 方式接入，需要先完成 mPaaS Inside 接入流程 。
- 若采用组件化方式接入，需要先完成 组件化接入流程 。

添加 SDK

原生 AAR 方式

参考 AAR 组件管理 ，通过 **组件管理（AAR）** 在工程中安装 **升级（UPGRADE）**组件。

mPaaS Inside 方式

在工程中通过 **组件管理** 安装 **升级（UPGRADE）**组件。

更多信息，参考 管理组件依赖 > 增删组件依赖 。

组件化方式

在 Portal 和 Bundle 工程中通过 **组件管理** 安装 **升级（UPGRADE）** 组件。

更多信息，参考 管理组件依赖 > 增删组件依赖 。

工程配置

配置 AndroidManifest

在 AndroidManifest.xml 中添加以下权限：

```
<uses-permission android:name="android.permission.REQUEST_INSTALL_PACKAGES"/>
```

在 Portal 工程主 module 的 src/main/res/xml 目录下创建文件 file_paths.xml，文件内容为：

```
<?xml version="1.0"encoding="utf-8"?>
<resources>
<paths>
<external-files-path
name="download"
path="com.alipay.android.phone.aliupgrade/downloads"/>
<external-path
name="download_sdcard"
path="ExtDataTunnel/files/com.alipay.android.phone.aliupgrade/downloads"/>
</paths>
</resources>
```

在 AndroidManifest.xml 文件中添加以下配置：

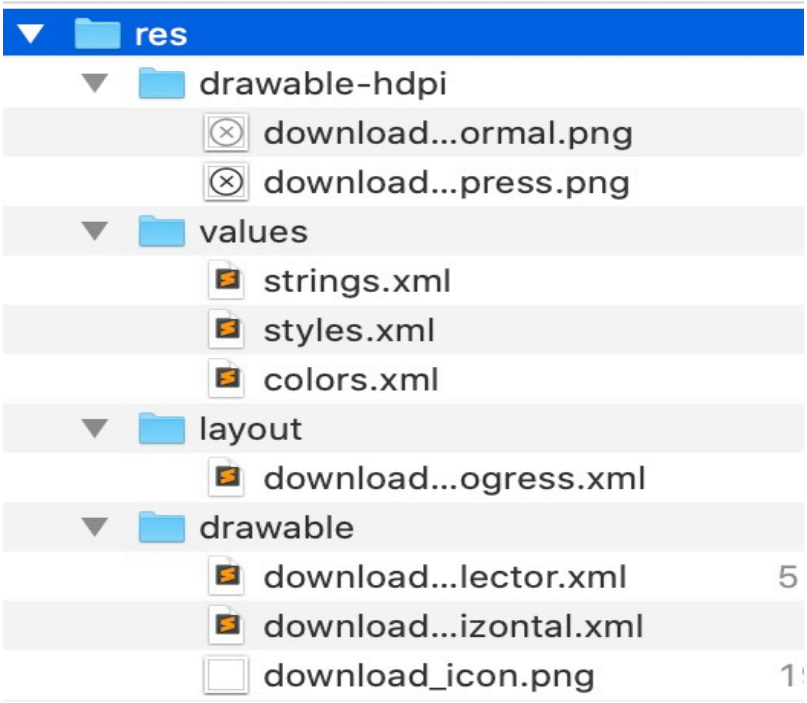
说明：需要将通配符 ${applicationId} 替换为真实的包名。 示例：将 ${applicationId}.fileprovider 替换为 com.mpaas.mobiledeliveryservice.fileprovider。

```
<provider
android:name="android.support.v4.content.FileProvider"
android:authorities="${applicationId}.fileprovider"
android:exported="false"
android:grantUriPermissions="true">
<meta-data
android:name="android.support.FILE_PROVIDER_PATHS"
android:resource="@xml/file_paths"/>
</provider>
```

添加资源

说明：如果您使用的是 mPaaS Inside 或原生 AAR 接入方式，则需要将以下资源加入到您的应用当中，否则将无法正常使用升级组件。点击此处 获取资源文件。

其中，将 values 目录下 strings.xml、styles.xml、colors.xml 的内容合并即可。



初始化 mPaaS

如果您使用原生 AAR 接入或 mPaaS Inside 接入方式，则需要初始化 mPaaS。

在 Application 对象中添加以下代码：

```
public class MyApplication extends Application {

@Override
protected void attachBaseContext(Context base) {
super.attachBaseContext(base);
// mPaaS 初始化回调设置
QuinoxlessFramework.setup(this, new IInitCallback() {
@Override
public void onPostInit() {
// 此回调表示 mPaaS 已经初始化完成，mPaaS 相关调用可在这个回调里进行
}
});
}

@Override
public void onCreate() {
super.onCreate();
// mPaaS 初始化
QuinoxlessFramework.init();
}
}
```

快速升级检测

快速检测新版本，仅返回检测结果：

```
MPUpgrade mMPUpgrade = new MPUpgrade();
// 同步方法，子线程中调用
int result = mMPUpgrade.fastCheckHasNewVersion();
if (result == UpgradeConstants.HAS_NEW_VERSION) {
// 有新版本
} else if (result == UpgradeConstants.HAS_NO_NEW_VERSION) {
// 没有新版本
} else if (result == UpgradeConstants.HAS_SOME_ERROR) {
// 错误
}
```

3.1.2 进阶指南

接入 SDK 之后，您可以根据业务需求，设置升级白名单，使用 SDK 进行升级检测、并提示用户。

设置白名单

设置白名单用户 ID：

```
MPLogger.setUserId("您的白名单ID");
```

检测新版本

快速检测新版本，并弹框提示：

说明：仅快速显示升级弹框，不包含强制升级逻辑，若您需要强制升级，请使用自定义升级来进行实现。

```
MPUpgrade mMPUpgrade = new MPUpgrade();
mMPUpgrade.fastCheckNewVersion(activity, drawable);
```

快速检测新版本，仅返回检测结果：

```
MPUpgrade mMPUpgrade = new MPUpgrade();
// 同步方法，子线程中调用
int result = mMPUpgrade.fastCheckHasNewVersion();
if (result == UpgradeConstants.HAS_NEW_VERSION) {
// 有新版本
} else if (result == UpgradeConstants.HAS_NO_NEW_VERSION) {
// 没有新版本
} else if (result == UpgradeConstants.HAS_SOME_ERROR) {
// 错误
}
```

获取升级详细信息

调用 fastGetClientUpgradeRes 方法获取更多详细信息：

```
MPUpgrade mMPUpgrade = new MPUpgrade();
// 同步方法，子线程中调用
ClientUpgradeRes clientUpgradeRes = mMPUpgrade.fastGetClientUpgradeRes();
```

以下返回的示例显示新版本号、下载地址等信息：

其中部分参数含义如下：

- downloadURL：下载地址
- guideMemo：升级信息
- newestVersion：最新版本
- resultStatus：升级模式
  - 202 为单次提醒
  - 204 为多次提醒
  - 203/206 为强制更新

其它自定义检测

更多定制，参考以下操作示例：

实现 MPaaSCheckCallBack 接口，用于响应升级 SDK 发出的请求，如弹出提示框：

```
MPUpgrade mMPUpgrade = new MPUpgrade();
mMPUpgrade.setUpgradeCallback(new MPaaSCheckVersionService.MPaaSCheckCallBack() {
.....
});
```

调用 MPUpgrade.checkNewVersion 方法检测升级信息。

说明：MPUpgrade 内部封装了 MPaaSCheckVersionService 的调用；您也可以自行定制实现。有关 MPaaSCheckVersionService 和 MPaaSCheckCallBack 的介绍，请参见 API 说明 。

自定义安装包下载目录（10.1.60 及以上版本支持）

配置如下：

```
File dir = getApplicationContext().getExternalFilesDir("自定义目录");
```

```
MPUpgrade mpUpgrade = new MPUpgrade();
mpUpgrade.setDownloadPath(dir.getAbsolutePath());
```

同时在 file_path.xml 中添加以下配置：

```
// external-files-path 对应 getExternalFilesDir 的目录
// 请使用与您自定义的目录对应的元素，如果您不清楚该如何选择，请搜索“适配 File Provider”
<external-files-path
name="download"
path="自定义目录"/>
```

处理强制升级解析包失败的问题

部分 rom 在强制升级后，会出现解析包失败问题。发生该问题的原因是，在部分 rom 中，安装包时会访问相应的 App 进程。而强制升级会强制结束 App 进程，所以导致解析包失败。虽然这种 rom 定制行为本身是不符合原生 Android 的行为，但您仍可以通过以下方式进行解决：设置UpgradeForceExitCallback，在 needForceExit 返回 false 即可。

实现回调。

```
public class UpgradeForceExitCallbackImpl implements UpgradeForceExitCallback {
@Override
public boolean needForceExit(boolean forceExitApp, MicroApplicationContext context) {
// 返回 false，就不强制杀掉进程，不会有安装包解析失败问题；返回 true，需要自行对进程的退出进行处理，会走到下面 doForceExit 方法中。
return false;
}
@Override
public void doForceExit(boolean forceExitApp, MicroApplicationContext context) {
// 如果需要关闭进程，则需要上面 needForceExit 返回 true，然后在本方法内自行关闭进程。
}
}
```

设置回调。

```
MPUpgrade mpUpgrade = new MPUpgrade();
mpUpgrade.setForceExitCallback(new UpgradeForceExitCallbackImpl());
```

注意：

- 使用同一 MPUpgrade 实例设置回调或请求升级。
- 设置回调后可以避免解析包失败问题，但升级组件将不再自动帮您杀进程。因此当用户没有点击 **安装** 而是返回到应用的时候，请您自行设置不可取消的弹框遮盖层，以防止用户绕过强制升级。

3.2 接入 iOS

3.2.1 添加 SDK

本文介绍如何添加与发布管理功能相关的 **升级发布** SDK。添加 SDK 并完成相关配置（详情请参考 使用 SDK ）后，您可以在 mPaaS 控制台发布 App 的新版本：

- 在 mPaaS 控制台做发布时，您可以设置 **更新提示**、**发布类型** 等选项。
- 在 mPaaS 控制台发布新版本后，客户端可以通过升级接口检测到该新版本，并提醒用户下载更新。

重要：由于 App Store 条例中禁止上线应用内置升级检测功能，因此在应用审核期间请不要在 mPaaS 控制台发布新版本。

前置条件

您已接入工程到 mPaaS。更多信息，请参见以下内容：

- 基于 mPaaS 框架接入
- 基于已有工程且使用 mPaaS 插件接入
- 基于已有工程且使用 CocoaPods 接入

添加 SDK

根据您采用的接入方式，请选择相应的添加方式。

使用 mPaaS Xcode Extension 插件

此方式适用于 **基于 mPaaS 框架接入** 或 **基于已有工程且使用 mPaaS 插件接入** 的接入方式。

1. 点击 Xcode 菜单项 **Editor** > **mPaaS** > **编辑工程**，打开编辑工程页面。
2. 选择 **升级发布**，保存后点击 **开始编辑**，即可完成添加。

使用 cocoapods-mPaaS 插件

此方式适用于 **基于已有工程且使用 CocoaPods 接入** 的接入方式。

1. 在 Podfile 文件中，使用 mPaaS_pod"mPaaS_Upgrade" 添加升级发布组件依赖。

2. 执行 pod install 即可完成接入。

后续步骤

使用 SDK

3.2.2 使用 SDK

添加 SDK 后，要将实时发布接入 iOS 客户端，还需完成以下步骤：

1. 检测新版本 ：在代码中调用 SDK 接口方法检查是否有新版本可升级。
2. 配置灰度白名单 ：设置更新提示、灰度等选项。
3. 线上发布 ：在 mPaaS 控制台生成 ipa 文件，并发布新版本。

操作步骤

检测新版本

升级检测 SDK 提供检查应用是否更新的接口文件，方法头文件在 AliUpgradeCheckService.framework > Headers > MPCheckUpgradeInterface.h 文件中。

```
typedef NS_ENUM(NSUInteger, AliUpdateType) {
AliUpgradeNewVersion = 201, /*当前使用的已是最新版本*/
AliUpgradeOneTime, /*客户端已有新版本，单次提醒*/
AliUpgradeForceUpdate, /*客户端已有新版本，强制升级（已废弃）*/
AliUpgradeEveryTime, /*客户端已有新版本，多次提醒*/
AliUpgradeRejectLogin, /*限制登录(已废弃)*/
AliUpgradeForceUpdateWithLogin /*客户端已有新版本，强制升级*/
};

/**
自定义 UI 时调用检测升级的成功回调
```

```
@param upgradeInfos 升级信息
@{upgradeType:202,
downloadURL:@"itunes://downLoader.xxxcom/xxx",
message:@"新版本更新，请升级",
upgradeShortVersion:@"9.9.0",
upgradeFullVersion:@"9.9.0.0000001"
needClientNetType:@"4G,WIFI",
userId:@"admin"
}
*/
typedef void(^AliCheckUpgradeComplete)(NSDictionary *upgradeInfos);
typedef void(^AliCheckUpgradeFailure)(NSException *exception);


@interface MPCheckUpgradeInterface : NSObject

/**
单次提醒时的时间间隔，单位为天，默认为 3
*/
@property(nonatomic, assign) NSTimeInterval defaultUpdateInterval;

/**
修改默认弹框提示 UI 的代理
*/
@property (nonatomic, weak) id<AliUpgradeViewDelegate> viewDelegate;

/**
* 初始化实例
*/
+ (instancetype)sharedService;

/**
主动检查是否有更新，若有更新，使用 mPaaS 默认提示 UI 自动弹框显示
*
*/
- (void)checkNewVersion;

/**
主动检查是否有更新。不会自动弹框提示，一般用于自定义 UI、检查是否有更新、提醒红点等情况

@param complete 成功回调，返回升级信息字典
@param failure 失败回调
*/
- (void)checkUpgradeWith:(AliCheckUpgradeComplete)complete
failure:(AliCheckUpgradeFailure)failure;
@end
```

开发者可在应用启动完成后调用相应接口检查应用是否更新，建议在首页出现后调用，以免影响 App 启动速度。根据展示升级提示信息的 UI 需求不同，提供以下三种调用方式：

使用 mPaaS 默认弹框展示升级提示信息：

```
- (void)checkUpgradeDefault {
```

```
[[MPCheckUpgradeInterface sharedService] checkNewVersion];
}
```

在 mPaaS 默认弹框 UI 的基础上，自定义弹框图片、网络提示 toast 或网络请求组进度条等内容：

```
- (void)checkUpgradeWithHeaderImage {
MPCheckUpgradeInterface *upgradeInterface = [MPCheckUpgradeInterface sharedService];
upgradeInterface.viewDelegate = self;
[upgradeInterface checkNewVersion];
}

- (UIImage *)upgradeImageViewHeader{
return APCommonUILoadImage(@"ilustration_ap_expection_alert");
}

- (void)showToastViewWith:(NSString *)message duration:(NSTimeInterval)timeInterval {
[self showAlert:message];
}

- (void)showAlert:(NSString*)message {
AUNoticeDialog* alertView = [[AUNoticeDialog alloc] initWithTitle:@"Information"message:message
delegate:nil cancelButtonTitle:@"OK"otherButtonTitles:nil, nil];
[alertView show];
}
```

若 mPaaS 提供的弹框样式不满足您的需求，可调用以下接口获取升级信息，自定义 UI 进行展示：

```
- (void)checkUpgradeWIthCustomUI {
[[MPCheckUpgradeInterface sharedService] checkUpgradeWith:^(NSDictionary *upgradeInfos) {
[self showAlert:[upgradeInfos JSONString]];
} failure:^(NSException *exception) {
}];
}
```

配置灰度白名单

要使用发布管理中的白名单灰度功能，确保服务端已获取客户端的唯一标识。客户端需要在 **MPaaSInterface** 的 category 中配置用户唯一标识，根据应用实际情况，在 userId 方法中返回 App 的唯一标识，例如用户名、手机号、邮箱等。

```
@implementation MPaaSInterface (Portal)

- (NSString *)userId
{
return @"mPaaS";
}

@end
```

mPaaS 控制台配置白名单的具体步骤，请查看 实时发布 > 白名单管理 。

线上发布

生成 ipa 文件

- 您可直接使用 Xcode 生成一个 ipa 安装包：

- 您也可以使用 mPaaS 插件提供的打包功能，生成 ipa 安装包，此包会放在当前工程的 product 目录下。

- **Bundle Identifier**：必须与云端配置文件中的 bundle Id 保持一致。
- **Bundle Version**：必须与工程 info.plist 中的 Production Version 保持一致。
- **Provisioning Profile**：签名配置文件，必须与 bundle Id 匹配，否则会打包失败。
- **Debug**：是否生成 debug 安装包。
- **App Store**：是否生成 App Store 发布包。

发布新版本

使用发布平台的发布管理功能，发布新版本。具体流程请参考 发布管理 。

升级模式：

在 mPaaS 控制台创建发布任务时，可选择升级模式，主要分为 3 种：

- **单次提醒**：当 mPaaS 控制台发布新版后，客户端调用一次版本升级接口，在静默周期内只弹框一次，以避免打扰用户。
  - 此升级模式适用于新版本刚上线引导用户升级的场景。
  - 默认的静默期为 3 天，即 3 天内只能提醒用户一次；若需修改此静默值，可在调用升级检测接口前设置以下属性:

```
- (void)checkUpgradeDefault {
[MPCheckUpgradeInterface sharedService].defaultUpdateInterval = 7;
[[MPCheckUpgradeInterface sharedService] checkNewVersion];
}
```

- **多次提醒**：当 mPaaS 控制台发布新版后，客户端调用一次版本升级接口，就弹框一次。此升级模式适用于新版本上线一段时间后，尽快引导用户升级到新版的场景。
- **强制提醒**：当 mPaaS 控制台发布新版后，客户端调用一次版本升级接口，就弹框一次，且无取消按钮，即不升级则不可使用 App。此升级模式适用于下线客户端旧版本、强制用户升级到新版本的场景。

相关链接

代码示例

3.3 Android 发布管理

发布管理是客户端升级新版本的配置后台，支持用户创建多任务、多维度的升级配置。

关于此任务

Android 发布管理的功能包括以下方面：

- 增加升级资源并提示二维码的下载地址。
- 创建、修改新版本资源包的任务。
- 对已添加的发布包创建多种类型的发布任务，例如白名单灰度、时间窗灰度、正式发布。
- 支持多种条件的升级过滤，例如城市、机型、设备系统版本、网络、发布包版本。

添加发布包

进入 mPaaS 控制台，完成以下步骤：

1. 在左侧导航栏，点击 **实时发布** > **发布管理**，页面显示发布管理列表。

   点击 **添加发布包**，在弹出的窗口中完成以下设置：

- **平台**：选择 **Android**。
- **发布包**：从本地选择发布包进行上传，只支持 .apk 格式。
- **版本号**：发布包的版本号，由数字和符号组成。
- **发布描述**：发布包的描述信息。
- **下载验证**：如开启该开关，则用户在扫描二维码后，需要通过验证码验证才能下载发布包。

点击 **确定**，完成添加，新添加的发布包会出现在页面的最上方。

说明：添加发布包后，在 **二维码** 列中会生成一个下载 .apk 发布包的二维码，扫描该二维码后，即可将发布包安装至手机。

4. 在发布管理列表，点击发布包前的加号图标（ + ）查看升级包的发布任务：
   - 如果升级包未发布过，当前包的状态为 **待发布**，并且没有任何发布任务。
   - 如果升级包发布过，当前包的状态为最新任务的发布状态，并且有相关的发布任务。

创建发布任务

对已添加的发布包创建发布任务，完成以下步骤：

1. 找到要创建发布任务的发布包。
2. 在右侧的 **操作** 列中，点击 **创建发布任务**。

在 **创建发布任务** 页面中，选择或输入以下信息：

- **发布类型**：分为 **灰度** 与 **正式**。
- **升级模式**：分为 **单次**、**多次** 与 **强制升级**。
  - **单次**：在 App 启动后根据静默策略提示升级。

    说明：静默策略指弹出升级提示，用户点击取消后一段时间内处于“静默”状态，不再提醒升级。默认静默时间为 3 天，可自定义。如需自定义静默时间，可参考 setIntervalTime 。
  - **多次**：在 App 每次启动后均提示升级，用户可手动关闭提示框。
  - **强制升级**：在 App 每次启动后提示升级并且无法关闭提示窗。
- **发布模型**（仅限 **灰度** 发布）：分为 **白名单灰度** 和 **时间窗灰度**。
  - 当选择 **白名单灰度** 时，您可在下方配置白名单。

    说明：您可在白名单管理中配置白名单。具体操作步骤，参见 白名单管理 。
  - 当选择 **时间窗灰度** 时，您可在下方选择时间窗的 **结束时间** 以及 **灰度人数**。

- **升级提示信息**（选填）：升级时所显示的信息。
- **发布描述**（选填）：本次发布的描述信息。

高级规则（仅限 **灰度** 发布）：点击 **添加**，您可在弹出的窗口中选择 **包含** 或 **不包含** 特定的 **城市**、**机型**、**网络** 等信息，并选择与 **类型** 对应的 **资源值**。

4. 设置完毕后，点击 **确定**，即可开始发布。您可点击发布包左侧的加号图标（ ）来查看刚刚创建的发布任务。

其他操作

发布任务创建成功后，您可以更改升级包的发布任务。

1. 在发布管理列表，点击发布包前的加号图标（ ）查看升级包的发布任务。
2. 根据需要，进行以下操作：
   - 点击 **暂停**，暂停发布任务。暂停后，如要继续进行该任务，点击 **继续**。
   - 点击 **结束**，终止发布任务。结束后，您不能再对任务做任何操作。

3.4 iOS 发布管理

> **重要**：自 2020 年 6 月 28 日起，mPaaS 停止维护 10.1.32 基线。请使用 10.1.68 或 10.1.60 系列基线。可以参考 mPaaS 10.1.68 升级指南 或 mPaaS 10.1.60 升级指南 进行基线版本升级。

发布管理是客户端升级新版本的配置后台，支持用户创建多任务、多维度的升级配置。

关于此任务

iOS 发布管理的功能包括以下方面：

- 增加升级资源并提示 App 的下载二维码（仅限 **企业分发**）。

- 创建、修改新版本资源包的任务。
- 对已添加的发布包创建多种类型的发布任务，例如白名单灰度、时间窗灰度、正式发布。
- 支持多种条件的升级过滤，例如城市、机型、设备系统版本、网络、发布包版本。

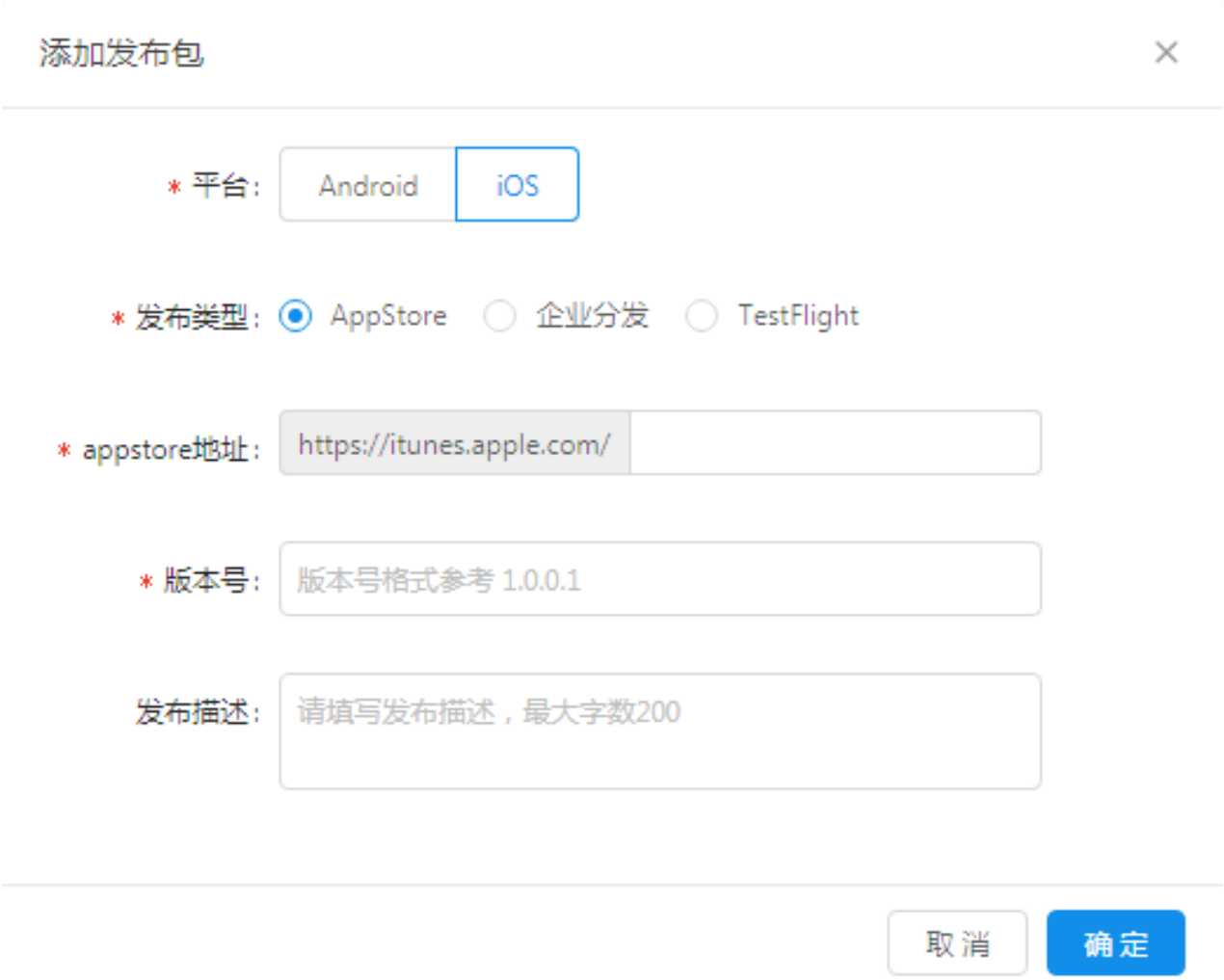
添加发布包

进入 mPaaS 控制台，完成以下步骤：

1. 在左侧导航栏，点击 **实时发布** > **发布管理**，页面显示发布管理列表。
2. 点击 **+ 添加发布包**，在弹出的窗口中完成以下设置：
   - **平台**：选择 **iOS**。
   - **发布类型**：分为 AppStore 、 企业分发 与 TestFlight ，详见下方的说明。
     - **AppStore**：针对从 AppStore 下载的 App 提示升级。
     - **企业分发**：针对在企业内部分发的 App 提示升级。
     - **TestFlight**：针对即将发布到 AppStore 的新版本做上线前的灰度验证。
3. 点击 **确定**，完成添加，新添加的发布包会出现在页面的最上方。
4. 在发布管理列表，点击发布包前的加号图标（ ）查看升级包的发布任务：
   - 如果升级包未发布过，当前包的状态为 **待发布**，并且没有任何发布任务。
   - 如果升级包发布过，当前包的状态为最新任务的发布状态，并且有相关的发布任务。

AppStore

重要：要使用 AppStore 发布，您需要先在苹果官方 App Store 中上架您的 App。
当您选择 **AppStore** 为发布类型时，您需要输入以下信息：



- **appstore 地址**：您的 App 在 App Store 上的地址。
- **版本号**：发布包的版本号。

注意：此版本号需与 iOS 工程 info.plist 文件中的 Product Version 字段保持一致。

- **发布描述**（选填）：发布包的描述信息。

企业分发

当您选择 **企业分发** 为发布类型时，您需要选择或输入以下信息：

- **上传图标**（可选）：可上传 .jpg 或 .png 格式的图片作为图标。
- **发布包**：从本地选择发布包进行上传，只支持 .ipa 格式。
- **bundleId**（选填）：您的 App 的 bundleId，若不填则使用在代码配置页面下载配置文件时填写的 bundleId。
- **版本号**：发布包的版本号。

注意：此版本号需与 iOS 工程 info.plist 文件中的 Product Version 字段保持一致。

- **发布描述**（选填）：发布包的描述信息。

- **下载验证**：如开启该开关，则用户在扫描二维码后，需要通过验证码验证才能下载发布包。

说明：添加 **企业分发** 类型的发布包后，在发布包列表页的 **二维码** 列中会生成一个下载 .ipa 发布包的二维码，扫描该二维码后，即可将发布包安装至手机。

TestFlight

重要：

- 要使用 TestFlight 测试功能，您必须已在 App Store Connect 中创建并启用了公开链接。
- 只有在版本 **≥ 10.1.32** 的客户端中才可使用 TestFlight。
- 您输入的 **包失效时间** 与 **测试人员上限** 必须与您在 App Store Connect 中设置的一致。

当您选择 **TestFlight** 为发布类型时，您需要输入以下信息：

- **公开链接地址**：您在 App Store Connect 中创建的公开链接地址，需保证此链接是启用状态。
- **包失效时间**：TestFlight 包的失效时间，需与您在 App Store Connect 中设置的一致。
- **测试人员上限**：参与测试的人员上限，需与您在 App Store Connect 中设置的一致。
- **版本号**：发布包的版本号。

注意：此版本号需与 iOS 工程 info.plist 文件中的 Product Version 字段保持一致。

- **发布描述**（选填）：发布包的描述信息。

创建发布任务

对已添加的发布包创建发布任务，完成以下步骤：

1. 找到要创建发布任务的发布包。
2. 在右侧的 **操作** 列中，点击 **创建发布任务**。

在 **创建发布任务** 页面中，选择或输入以下信息：

- **发布类型**：分为 **灰度** 与 **正式**。
  - **灰度**：在正式发布前，进行小规模发布以验证新包的功能是否达到预期，发布对象是部分用户。
  - **正式**：正式发布版本，发布对象是全部用户。

说明： 对于 TestFlight 与 **企业分发** 类型的发布包，固定为 **灰度** 类型，且不可选择。

- **升级模式**：分为 **单次**、**多次** 与 **强制升级**。
  - **单次**：在 App 启动后根据静默策略提示升级。

    说明：静默策略指弹出升级提示后，用户点击取消后一段时间内处于“静默”状态，不再提醒升级。默认静默时间为 3 天，可自定义。如需自定义静默时间，可参考发布新版本 。
  - **多次**：在 App 每次启动后均提示升级。
  - **强制升级**：在 App 每次启动后提示升级并且无法关闭提示窗。

说明：TestFlight 类型的发布包无 **强制升级**，只有 **单次** 与 **多次**。

- **发布模型**（仅限 **灰度** 发布）：分为 **白名单灰度** 和 **时间窗灰度**。
- 当选择 **白名单灰度** 时，您可在下方配置白名单。

  说明：您可在白名单管理中配置白名单。具体操作步骤，参见 白名单管理 。
- 当选择 **时间窗灰度** 时，您可在下方选择时间窗的 **结束时间** 以及 **灰度人数**。

说明：**企业分发** 类型的发布包无 **时间窗灰度**，只有 **白名单灰度**。

- **升级提示信息**（选填）：升级时所显示的信息。
- **发布描述**（选填）：本次发布的描述信息。

高级规则（仅限 **灰度** 发布）：点击 **添加**，您可在弹出的窗口中选择 **包含** 或 **不包含** 特定的 **城市**、**机型**、**网络** 等信息，并选择与 **类型** 对应的 **资源值**。

4. 设置完毕后，点击 **确定**，即可开始发布。您可点击发布包左侧的加号图标（ ）来查看刚刚创建的发布任务。

相关操作

- 上传符号表。在发布管理列表，您可对已添加的发布包上传符号表。
  - 一个 .ipa 发布包对应一个符号表文件。
  - 只支持 dSYM 格式的符号表，且需要将文件压缩成 .tgz 格式上传。
- 变更升级包的发布任务。在发布管理列表，点击发布包前的加号图标（ ）查看升级包的发布任务。
  - 点击 **暂停**，暂停发布任务。暂停后，如要继续进行该任务，点击 **继续**。
  - 点击 **结束**，终止发布任务。结束后，您不能再对任务做任何操作。

4 热修复管理

4.1 接入 Android

4.1.1 Android 接入说明

热修复（Hotpatch）用于在不发布新版本的情况下热修复线上故障（Bug）。

说明：公有云环境中仅支持 Android 热修复，iOS 热修复仅限专有云使用。

使用场景

每一次热修复，都是一次 **紧急发布**。因此，mPaaS 限定了热修复的使用范围是：**在来不及发版的情况下**，需要立刻解决线上客户端问题。

根据最佳实践，紧急发布只用于修复严重的、影响面大的、具有高可复现性的问题。包括但不仅限于以下情况：

- 高概率的闪退
- 严重的 UI 问题
- 可能造成资损与用户投诉的 Bug
- 客户端某些功能不能使用
- 监管审查导致的紧急修改

使用限制

暂不支持以下机型或场景：

- Dalvik 的 X86 机型
- 三星 5.0.X 机型
- API Level 21～23 且打开了 Jit 的机型
- Lemur 虚拟机，以及 Dalvik 的 Art 模式

使用流程

热修复完整的使用流程包括：

1. 客户端集成热修复功能
2. 生成热修复包
3. 发布热修复包

4.1.2 接入 Android 客户端

本文介绍如何在当前 Android App 的基础上集成 mPaaS 提供的 Hotpatch 热修复功能。目前，热修复支持 **原生 AAR 接入**、**mPaaS Inside 接入** 和 **组件化接入** 三种接入方式。

整个过程分为以下五步：

1. 添加 SDK

2. 初始化热修复 （仅原生 AAR 接入或 mPaaS Inside 接入需要）
3. 生成热修复补丁
4. 发布热修复补丁
5. 触发热修复补丁

前置条件

- 若采用原生 AAR 方式接入，需要先 将 mPaaS 添加到您的项目中 。
- 若采用 mPaaS Inside 方式接入，需要先完成 mPaaS Inside 接入流程 。
- 若采用组件化方式接入，需要先完成 组件化接入流程 。

添加 SDK

原生 AAR 方式

参考 AAR 组件管理 ，通过 **组件管理（AAR）** 在工程中安装 **热修复（HOTFIX）**组件。

mPaaS Inside 方式

在工程中通过 **组件管理** 安装 **热修复（HOTFIX）**组件。更多信息，参考 管理组件依赖 > 增删组件依赖 。

组件化方式

在 Portal 和 Bundle 工程中通过 **组件管理** 安装 **热修复（HOTFIX）** 组件。
更多信息，参考 管理组件依赖 > 增删组件依赖 。

初始化热修复

原生 AAR 接入/mPaaS Inside 接入

如果需要使用热修复功能，您还需要完成以下两步操作。

需要将 Application 对象重新继承为 QuinoxlessApplicationLike，并注意将该类防混淆。此处以“MyApplication” 为例。

```
@Keep
public class MyApplication extends QuinoxlessApplicationLike implements
Application.ActivityLifecycleCallbacks {
private static final String TAG ="MyApplication";
@Override
protected void attachBaseContext(Context base) {
super.attachBaseContext(base);
Log.i(TAG,"attacheBaseContext");
}
@Override
public void onCreate() {
super.onCreate();
Log.i(TAG,"onCreate");
registerActivityLifecycleCallbacks(this);
```

```
}
@Override
public void onMPaaSFrameworkInitFinished() {
MPHotPatch.init();
LoggerFactory.getTraceLogger().info(TAG, getProcessName());
}
@Override
public void onActivityCreated(Activity activity, Bundle savedInstanceState) {
Log.i(TAG,"onActivityCreated");
}
@Override
public void onActivityStarted(Activity activity) {
}
@Override
public void onActivityResumed(Activity activity) {
}
@Override
public void onActivityPaused(Activity activity) {
}
@Override
public void onActivityStopped(Activity activity) {
}
@Override
public void onActivitySaveInstanceState(Activity activity, Bundle outState) {
}
@Override
public void onActivityDestroyed(Activity activity) {
}
}
```

在 AndroidManifest.xml 文件中将 Application 对象指向 mPaaS 提供的 Application 对象。将刚刚生成的 “MyApplication” 类添加到 key 为 mpaas.quinoxless.extern.application 的 meta-data 中。示例如下：

```
<application
android:name="com.alipay.mobile.framework.quinoxless.QuinoxlessApplication">
<meta-data
android:name="mpaas.quinoxless.extern.application"
android:value="com.mpaas.demo.MyApplication"
/>
</application>
```

其中 com.mpaas.demo.MyApplication 是您自定义的 Application 代理类，继承 QuinoxlessApplicationLike。

组件化接入

由于已经集成了相关内容，因此该接入方式不需要做任何变更。

生成热修复补丁

参见 生成热修复包 。

发布热修复补丁

参见 发布热修复包 。

触发热修复补丁

本节结合 代码示例 中的 **热修复** 示例，对热修复过程进行详细的说明介绍。

该代码示例中的修复内容是弹出的 toast 中的内容。

修复前
点击 **模拟需要被热修复的点击事件** 按钮，弹出如下图所示的 toast。

进行修复
点击 **触发热修复部署检测** 按钮，触发热修复的下载。在下载完成后，彻底关闭 Demo 应用并重新启

动。

- 修复后
  点击 **模拟需要被热修复的点击事件** 按钮，会弹出 “当前点击事件已被热修复” 的 toast。


11:28
热修复会修复该按钮的点击事件里面的代码逻辑
模拟需要被热修复的点击事件
通常情况下，建议开发者在MockLauncherApplicationAgent的postInit中触发热修复部署检测
触发热修复部署检测
注意：
使用热修复时，开发者需要确认加密图片已经正确生成，否则hotpatch包将无法正常下发。
当前点击事件已被热修复

问题排查

注意：

- 不要使用非正规方式引入 apache-httpclient，具体参考 Android 应用开发者平台官方文档。
- 不要使用非正规方式引入 NFC 系统相关的 SDK。
- 确定没有继承任何 Application 相关的类，确定使用 ApplicationLike 代替。

热修复日志请使用 tag：DynamicRelease 过滤。

- **下载阶段**：可以同时过滤 RPCException，如果有相关的异常，那么下载不会成功。
- **合并补丁阶段**：可以过滤 immediately=true，如果发现相关日志，则表示合并补丁成功。合并补丁成功之后，理论上只要重启 App，补丁就会生效。

4.2 接入 iOS

4.2.1 iOS 接入说明

重要：

- iOS 热修复仅限专有云使用，每次基线升级时需要通过 提交工单 申请。
- 使用时需注意，**默认公有云拉取到的 SDK 为空实现。**

在 iOS 开发领域，由于 AppStore 审核标准严格、审核周期长、效率低，应用的发版速度极慢，因此能快速修复线上严重 Bug 而无需发布新版本的热修复方法对于 iOS 应用来说就显得尤其重要。

mPaaS 提供的 Hotpatch 热修复技术，在 Runtime 运行时特性的基础上，通过 JS 替换原有的 Objective-C 方法，从而达到修复线上 bug 的目的。目前包含的能力如下：

- 添加类，修改类（包括添加实例方法、类方法、属性，修改方法实现等）。
- 调用任意 Objective-C 类方法，访问成员变量。
- 使用 block、struct、GCD 等高级语法。
- 回滚操作热生效，被替换的方法能即时恢复。
- 脚本执行时机控制（启动前主线程运行、启动完成之后子线程运行）。
- 完善的安全加密、签名验证系统。

4.2.2 添加 SDK

本文介绍如何将热修复组件接入到 iOS 客户端。

热修复支持基于 mPaaS 框架接入、基于已有工程且使用 mPaaS 插件接入以及基于已有工程且使用 CocoaPods 接入三种接入方式。您可以参考 接入方式介绍 ，根据实际业务情况选择合适的接入方式。

前置条件

您已接入工程到 mPaaS。更多信息，请参见以下内容：

- 基于 mPaaS 框架接入
- 基于已有工程且使用 mPaaS 插件接入
- 基于已有工程且使用 CocoaPods 接入

添加 SDK

根据您采用的接入方式，请选择相应的添加方式。

使用 mPaaS Xcode Extension 插件

此方式适用于 **基于 mPaaS 框架接入** 或 **基于已有工程且使用 mPaaS 插件接入** 的接入方式。

1. 点击 Xcode 菜单项 **Editor** > **mPaaS** > **编辑工程**，打开编辑工程页面。
2. 选择 **热修复**，保存后点击 **开始编辑**，即可完成添加。

使用 cocoapods-mPaaS 插件

此方式适用于 **基于已有工程且使用 CocoaPods 接入** 的接入方式。

1. 在 Podfile 文件中，使用 mPaaS_pod"mPaaS_Hotpatch" 添加热修复组件依赖。

2. 执行 pod install 即可完成接入。

4.2.3 使用 SDK

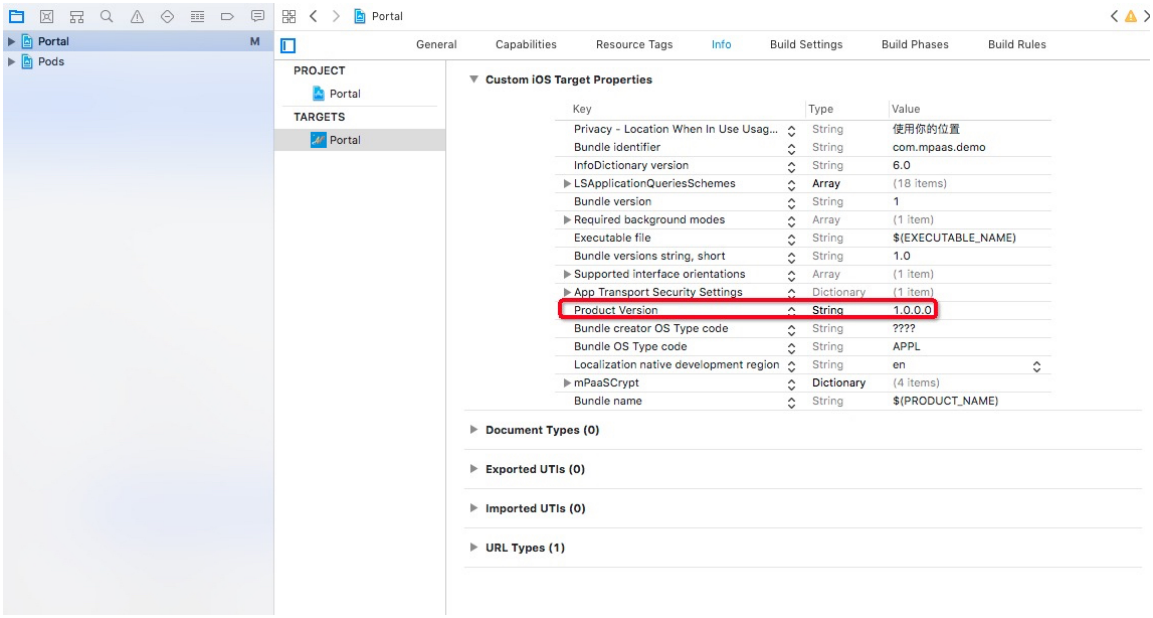
添加 SDK 后，要将热修复接入 iOS 客户端，还需完成以下步骤：

1. 配置工程
2. 管理加密信息
3. 同步服务端脚本
4. 热修复线上问题

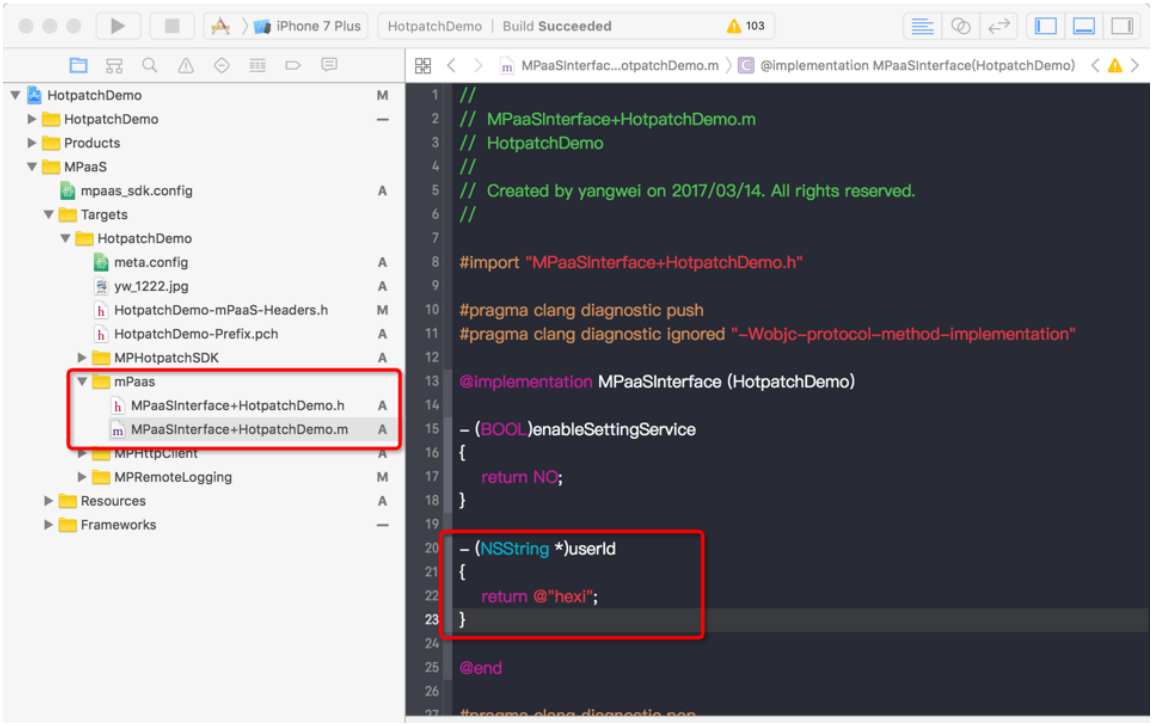
配置工程

添加 SDK 之后，您需要按照以下描述配置工程：

在 info.plist 中配置应用版本 **Product Version**。在发布平台中，需要根据这里配置的版本号进行发布。



发布平台支持白名单灰度发布。在这种情况下，客户端去拉取热修复脚本时，服务端会根据白名单判断是否下发热修复包给该用户。为此，客户端在 MPaaSInterface 的 category 中，需指定用户的 userID 信息：



旧版本升级注意事项

10.1.32 版本之后不再需要 info.plist 中 Product ID，mPaaS 配置中间层会实现包装从 meta.config 中读取，旧版本中的 DynamicReleaseInterface 类的 Category 也不再需要，升级版本后请检查工程中是否存在旧版本配置，如果有请移除。

管理加密信息

Hotpatch 有自定义的验签流程，以此保证脚本来源的正确性。您需要使用 mPaaS Xcode Extension > 热修复生成资源包 对原始的 JS 脚本进行加密加签，再将其上传到发布平台，其流程如下：

1. 读取脚本内容，对内容字符串做 MD5 加密。
2. 以二进制格式读取脚本内容，对二进制内容做 AES128 加密。
3. 创建 zip 文件，将上述两项内容添加到 zip 文件中。
4. 以二进制格式读取上述 zip 文件，再进行一次 AES128 加密，加密后数据保存为 zip 文件。
5. 以二进制格式读取上述 zip 文件，使用 SHA1 算法，做一次签名。
6. 以二进制格式读取上述 zip 文件，加上数据分隔符，加上签名数据，组成脚本包。

客户端从发布平台获取到脚本包，也会使用相反的顺序解签，得到可以真正运行的脚本。

在 Hotpatch 脚本的验签和加解密这个过程中，涉及到一对 RSA 非对称密钥和一个 AES 对称加密密钥，每个接入应用需要使用自己的密钥，保证脚本下发的安全性。

管理 RSA 非对称加密

您需要按照以下描述管理 RSA 非对称加密：

执行以下命令生成公钥文件（public_key.pem）和私钥文件（private_key.pem）：

```
openssl genrsa -out private_key.pem 2048
openssl rsa -in private_key.pem -pubout -out public_key.pem
```

将公钥文件添加到工程中：

在 main.m 中验证公钥自身的签名，以确保公钥文件未被替换。具体步骤如下：

执行以下命令生成公钥文件的签名二进制串：

```
mpaas inst hotpatch sign -i /path/to/rsa_public_key.pem -p /path/to/rsa_private_key.pem -o /path/to/output.sig
```

- -i：openssl 生成的公钥文件 public_key.pem。
- -p：openssl 生成的私钥文件 private_key.pem。
- -o：签名二进制串输出文件。

将终端打印出来的签名数组拷贝到 main.m 方法中，对公钥自身进行签名。示例代码如下：

```
#import <MPDynamicAdapter/MPDynamicInterface.h>
static BOOL initDynamicSec()
```

```
{
char sig[] =
{0xa,0xbb,0xe7,0x59,0x3a,0xf3,0x25,0x71,0x2d,0x24,0x35,0xac,0x69,0x5a,0x6b,0x4e,0x92,0x8f,0xf0,0x8c,0xcd,0xd,0x3
8,0x4,0xe2,0x97,0xb8,0x2a,0xe1,0xf7,0x6a,0x57,0xd9,0x9d,0x1b,0x6d,0x8b,0x3,0xc6,0x8d,0xc5,0xa,0x57,0x39,0x7b,0x
98,0xe1,0xca,0x74,0x93,0xf8,0xf1,0x15,0xbd,0xfe,0x4,0x7,0x24,0xa5,0xda,0xe8,0x37,0x4e,0x8d,0x9b,0x56,0x86,0xe9,0
xc2,0x2c,0x60,0x8f,0x9f,0x99,0x40,0xf1,0x97,0x97,0x15,0xd1,0x26,0x87,0x79,0x24,0x79,0x20,0xd6,0x96,0x62,0x70,0xb
e,0x7c,0xda,0x1,0x63,0xe9,0x19,0xe1,0x1f,0x5a,0xc2,0x1b,0x97,0x1a,0xdb,0x11,0xbd,0xee,0xdc,0x40,0x99,0xc1,0x54,0
x6e,0x9a,0x30,0xb2,0x44,0x45,0x64,0xa9,0xc,0xea,0x86,0x4c,0x7d,0x91,0x30,0xf5,0x41,0xe,0x30,0x8f,0x9c,0x85,0xdb,
0xd,0xc1,0xc1,0xec,0x7d,0x31,0xb,0x77,0xce,0xf2,0xd5,0x5,0xd1,0xe9,0x32,0xf7,0x15,0xa5,0x53,0x79,0xee,0x55,0x86,
0x2c,0x9a,0x30,0x2b,0xd,0xe9,0x36,0x9,0x31,0x26,0xa5,0x6d,0xaf,0xd,0xd5,0x72,0x16,0xd2,0xd7,0x2c,0x88,0x13,0x6a
,0x87,0xf0,0x4c,0x7e,0xb0,0x34,0xc,0x2a,0x75,0xc3,0x71,0x18,0x6f,0xbc,0xc9,0x8a,0xb9,0x50,0xdd,0x7,0x19,0x76,0x7
d,0x2d,0xcf,0x2d,0xc0,0xa6,0xe7,0x1d,0x48,0x26,0x6,0xee,0x2,0xb,0x5a,0x85,0x36,0x54,0x7e,0x9a,0x4d,0xf6,0x27,0x9
c,0x30,0xc8,0x63,0x74,0x8b,0x82,0x9b,0x64,0x3b,0xd6,0x13,0x53,0xdc,0x36,0xb5,0xbc,0xb2,0x6a,0xd0,0x8f,0x18,0xb
d,0x2a,0x5c,0x4b,0x4,0xc1,0x45};
NSString *path;
path = [[NSBundle mainBundle] pathForResource:@"public_key"ofType:@"pem"];
return [MPDynamicInterface initDynamicSecWithPublicKey:path signature:sig sigLength:sizeof(sig)];
}

int main(int argc, char * argv[]) {
@autoreleasepool {
BOOL ret = initDynamicSec();
if (NO == ret) {
NSLog(@"The public key is modified.");
}
return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
}
}
```

说明：若 initDynamicSec() 返回 NO，则说明公私钥不匹配，需要修改，否则平台下发的包在客户端会校验不过，从而导致热修复失败。

管理 AES 对称加密

AES 对称加密的密钥保存在无线保镖图片 yw_1222.jpg 中。

- **公有云**：参照热修复 SDK 接入文档完成基础接入步骤后，工程中会自动生成无线保镖图片，您无需额外操作。
- **专有云**：请参考 mPaaS Xcode Extension > 无线保镖 生成专有云无线保镖图片。

同步服务端脚本

您需要确保客户端调用 SDK 提供的同步接口，以执行热修复逻辑，包括获取服务端下发的修复脚本、执行脚本、回滚等。

基于原生框架

在程序启动后（推荐在 didFinishLauncher 方法中）调用以下方法：

```
[MPDynamicInterface initDynamicSyncLocalFile];
```

基于 mPaaS 框架

mPaaS 框架 DFClientDelegate 接管了应用生命周期，在程序启动时会自动调用热修复同步接口，您无需额外操

作。

热修复线上问题

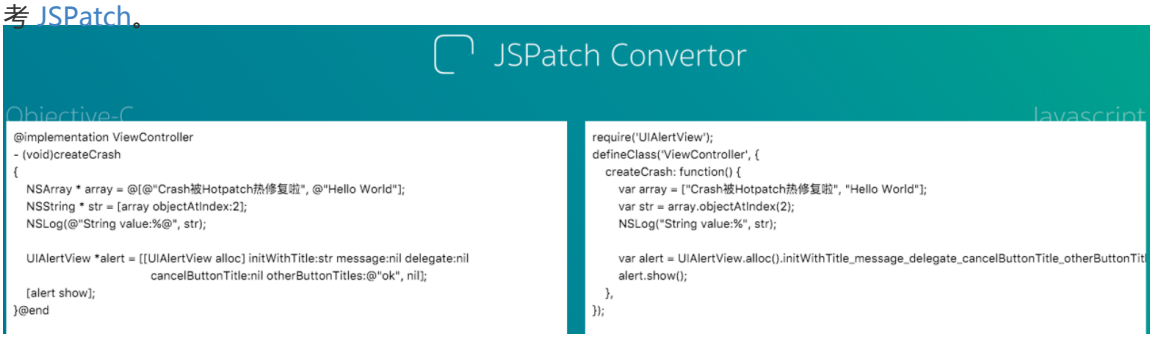
发现线上 Bug 后，可以使用 JSPatch 替换线上的错误代码，然后通过发布平台推送给用户进行修复。步骤如下：

1. 生成脚本
2. 加密脚本
3. 发布脚本

生成脚本

生成脚本的步骤如下：

1. 定位问题原因，使用原生 Objective-C 代码进行本地修复验证。
2. 验证通过后，用在线工具 JSPatchConvertor 将 Objective-C 代码转为 JS 脚本，详细的转换规则请参考 JSPatch。



3. 为验证转化后 JS 脚本（假设为 Test.js）的正确性，可以将写好的 JS 文件拖入到工程中，并在程序启动时（推荐在 didFinishLaunchingWithOptions 中）手动调用 JS 脚本，验证问题是否被正确修复。示例代码如下：

```
NSString *file = [[[NSBundle mainBundle] resourcePath] stringByAppendingPathComponent:[NSString stringWithFormat:@"Test.js"]];
[MPDynamicInterface runDynamicLocalFile:file];
```

4. 测试通过后，将测试代码和 Test.js 脚本删除，以免影响原有代码逻辑。

加密脚本

为了安全，本地测试通过后的 JS 脚本文件，需要进行打包加密后才能提交到发布平台。

具体步骤如下：

1. 使用 mPaaS Xcode Extension > 热修复生成资源包 ：

图中各参数含义如下：

- **脚本文件**：本地测试验证通过后的 JS 脚本。
- **App Secret**：即 mPaaS 控制台上的 appsecret。若 mPaaS Xcode Extension 未自动填入，请手动填入。
- **私钥文件**：与添加到工程中的公钥文件配对的私钥文件。详情请参考上文 **管理加密信息** > **管理RSA 非对称加密**。

2. Hotpatch 资源包默认保存在 JS 脚本同级目录下。包含如下文件：
   - Test.js：上传到发布平台的 JS 脚本文件。
   - Test.sig：加密后的签名文件。
   - Test.zip：加密后的脚本文件。通常在上传到发布平台之前，用来验证加密算法是否正确。

3. 为验证加密后的 JS 脚本的正确性，可以将生成的 Test.zip 文件拖入到工程中，并在程序启动时（推荐在 didFinishLaunchingWithOptions 中）手动执行 Test.zip 脚本，验证问题是否被正确修复。

```
NSString *file = [[[NSBundle mainBundle] resourcePath] stringByAppendingPathComponent:[NSString stringWithFormat:@"Test.zip"]];
[MPDynamicInterface runDynamicLocalSecFile:file];
```

4. 测试通过后，将测试代码及 Test.zip 包删除，以免影响代码原有逻辑。

说明：若此处测试通过，则说明对称加解密正确。此外，您还需要确保非对称加解密正确，详情请参考上文 **管理加密信息** > **管理 RSA 非对称加密** 中 initDynamicSec() 方法。

发布脚本

上一步加密脚本验证通过后，将加密后的 Test.js 文件上传到发布平台进行线上修复验证，详细的热修复发布流程请参考文档 热修复管理 。

4.3 使用热修复

热修复是指通过代码变动在不发版本的情况下直接修复线上的问题。**热修复管理** 是客户端修复紧急问题的配置后台，您可以在该配置后台创建发布任务以及进行多维度的热修复配置。

关于此任务

在 **热修复管理** 页面，通常您需要完成以下操作：

1. 添加热修复 ：将热修复包添加至 mPaaS 控制台。
2. 发布热修复包 ：根据最佳实践，发布热修复包会经历的阶段依次为白名单灰度发布、时间窗灰度发布、正式发布。

   说明：发布热修复包过程中，如果出现代码问题，可以进行回滚操作。
3. 管理发布任务 ：管理发布任务包括修改、暂停、结束发布任务。

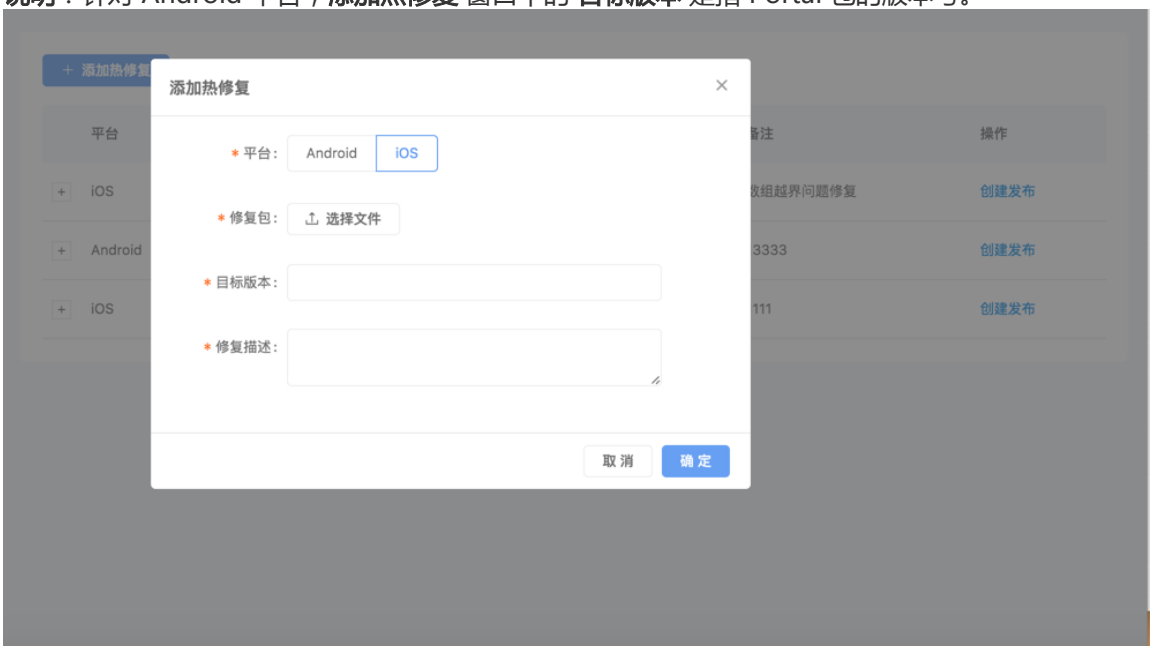
热修复包分为 Android 和 iOS 热修复包。针对 Android 客户端的热修复，一个版本最多只能有一个热修复包。如果一个 Android 客户端的一个版本有两个问题，请先在本地将两个问题的热修复包合成一个，再上传该热修复包。

添加热修复

登录 mPaaS 控制台，点击左侧导航栏中的 **实时发布** > **热修复管理**， 进入热修复资源列表页。

点击 **添加热修复** 新增热修复资源，在弹出的 **添加热修复** 窗口中完成热修复配置。

说明：针对 Android 平台，**添加热修复** 窗口中的 **目标版本** 是指 Portal 包的版本号。



在热修复资源列表中，点击指定热修复左侧的展开按钮（ ）查看热修复包的发布任务。

说明：

- 如果资源包未发布过，当前包的状态为待发布，并且没有发布任务。一个热修复资源在同一时间只能有一个有效的发布任务。
- 如果资源包已发布，则当前包的状态为最新任务的发布状态，并且有相关的发布任务。

发布热修复包

1. 在热修复资源列表中，点击指定热修复右侧的 **创建发布** 新增一个发布任务。
2. 在 **创建发布任务** 窗口中，完成以下设置。

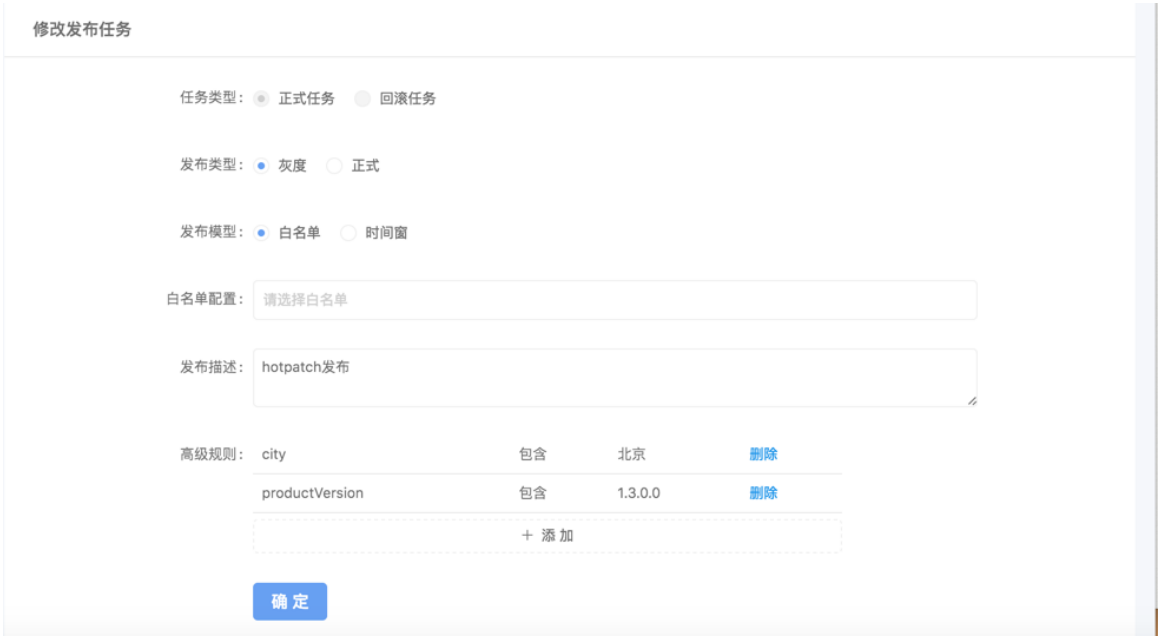
- **任务类型**：分为 **正式任务** 和 **回滚任务**。
  - **正式任务** ：适用于正式发布的热修复包。
  - **回滚任务** ：若热修复过程中出现问题，可进行回滚操作。回滚任务的任务类型固定为 **正式**，不可更改。
- **发布类型**：分为 **灰度** 与 **正式**。
- **发布模型**（仅限 **灰度** 发布）：分为 **白名单** 和 **时间窗**。
  - 当选择 **白名单** 时，您可在下方选择白名单。白名单需在白名单管理中配置。具体操作步骤，参见 白名单管理 。
  - 当选择 **时间窗** 时，您可在下方选择时间窗的 **结束时间** 以及 **灰度人数**。
- **高级规则**：设置高级规则以进行多种条件的过滤，如城市、机型、网络等，高级规则的选择在 资源配置管理 中配置。

3. 点击 **确定**， 完成任务创建。

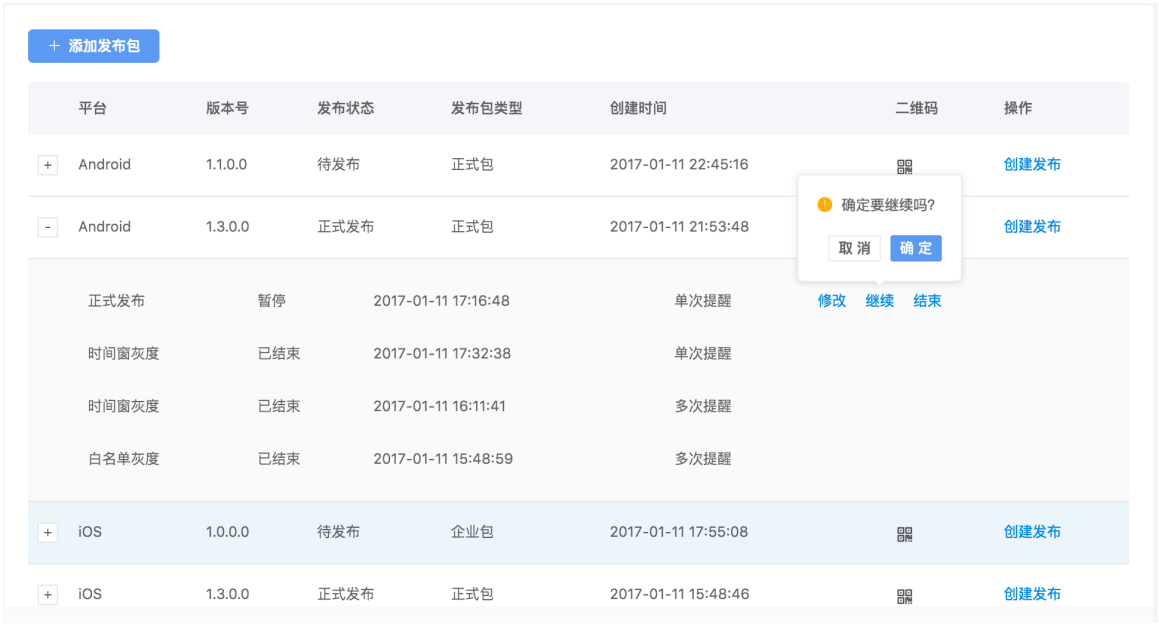
管理发布任务

在热修复资源列表中，选择指定的热修复资源的某个发布任务，点击该任务右侧的 **修改**、**暂停** 或 **结束** 按钮以修改、暂停或结束该任务。

修改任务



暂停任务：暂停的任务可以继续进行操作。



结束任务：结束的任务不能再做任何操作。

4.4 使用教程

4.4.1 Android 热修复使用教程

本文将引导您从零开始创建 Android 工程并使用热修复功能。热修复功能支持 **原生 AAR 接入**、**mPaaS Inside 接入** 和 **组件化接入** 三种接入方式。关于接入方式的更多说明，请参见 接入方式介绍 。

本文将分为两部分向您完整的介绍并演示热修复的使用流程： 接入热修复 ， 热修复 Bug 演示 。

接入热修复

接入热修复流程如下：

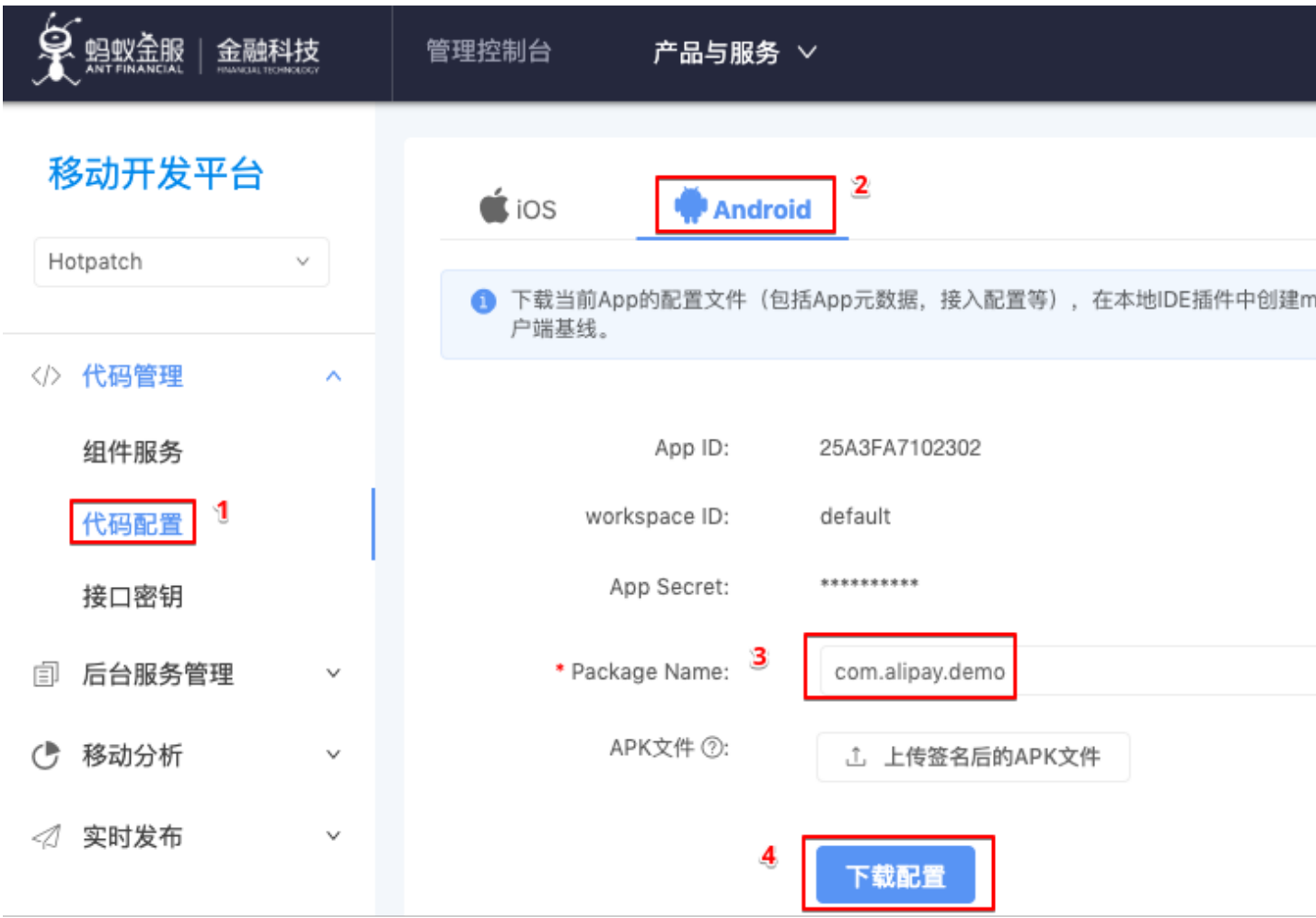
1. 配置开发环境
2. 在控制台创建应用
3. 在客户端创建新工程
4. 签名
5. 配置加密信息
6. 编写代码
7. 发布带有热修复功能的客户端版本

配置开发环境

参考文档 配置开发环境 。

在控制台创建应用

参考文档 在控制台创建应用 。此时，本地还没有带签名的 APK，因此在下载配置文件时，可以暂不上传 APK。

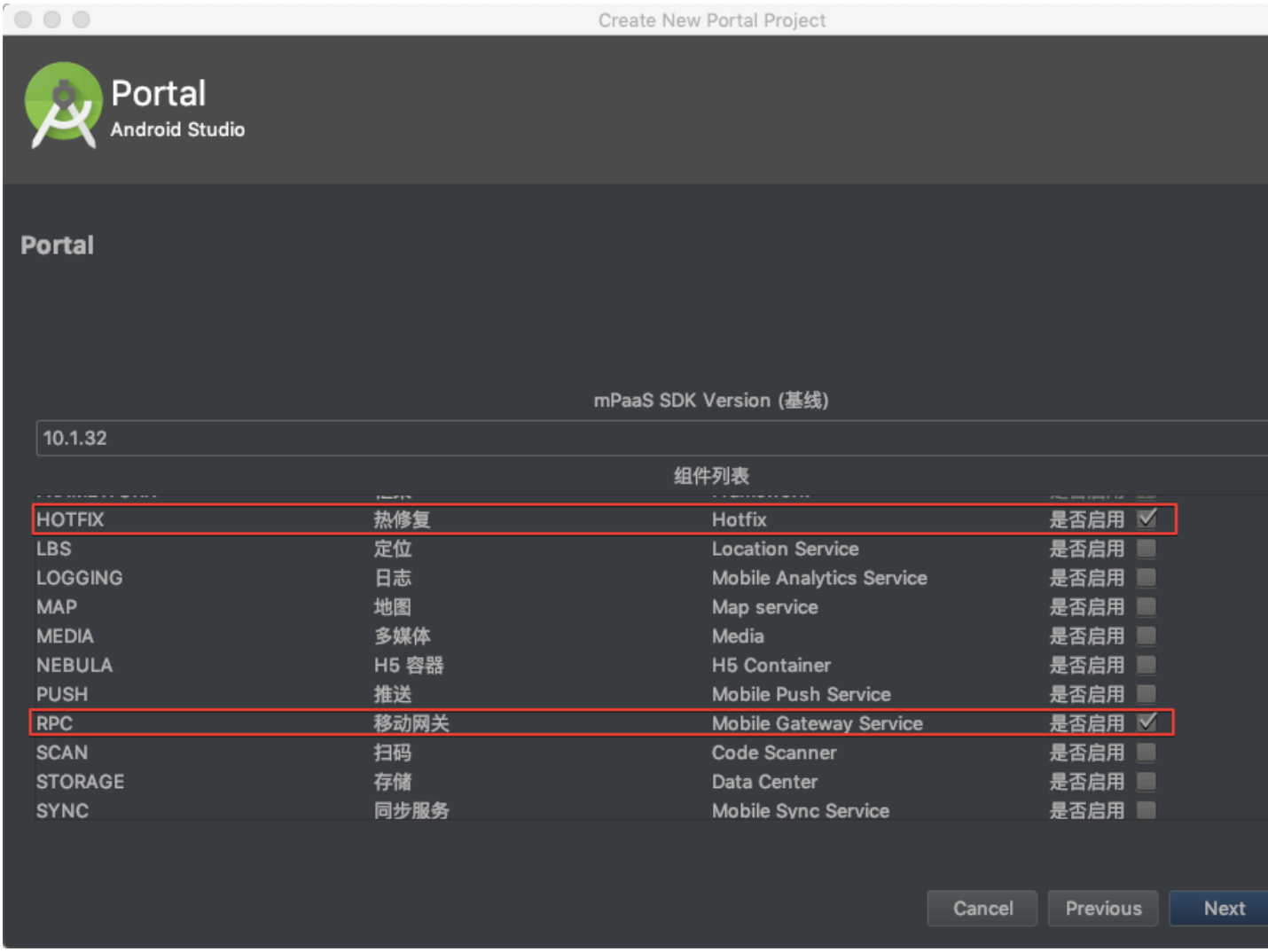


下载的配置文件的文件名示例：Ant-mpaas-4111111111005-default-Android.config。

在客户端创建新工程

参考 接入流程说明 在客户端创建 **基于 mPaaS 框架** 的新工程，并确保使用 **mPaaS** > **Build** 成功 构建工程 。

说明：在添加 SDK 时，需要确保勾选了下图中的 **HOTFIX** 和 **RPC**。



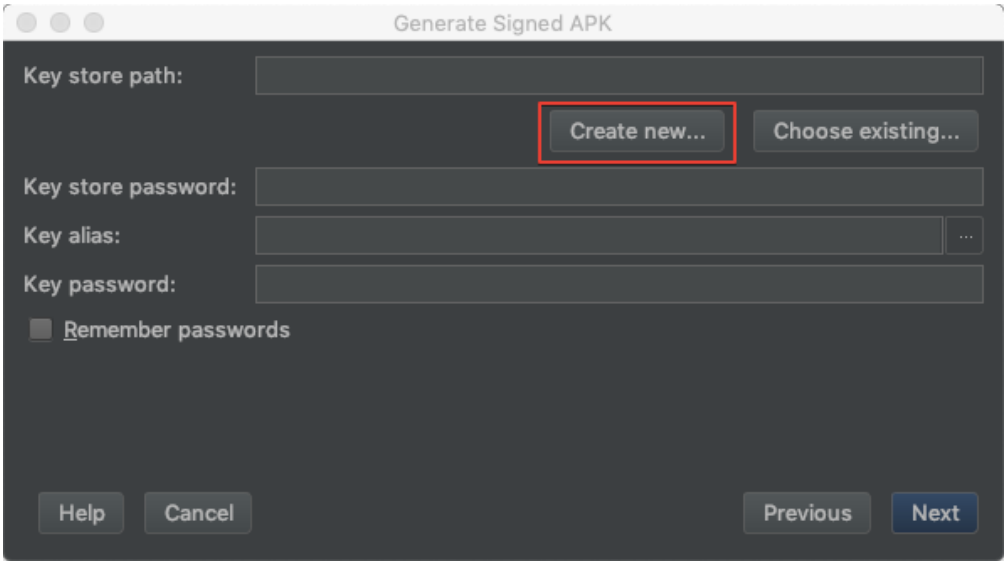
签名

签名是为了下一步配置加密信息做准备。

在 Android Studio 中打开 **Portal 工程**，然后参考 Android 官网文档 给应用签名，并生成带签名的 APK。

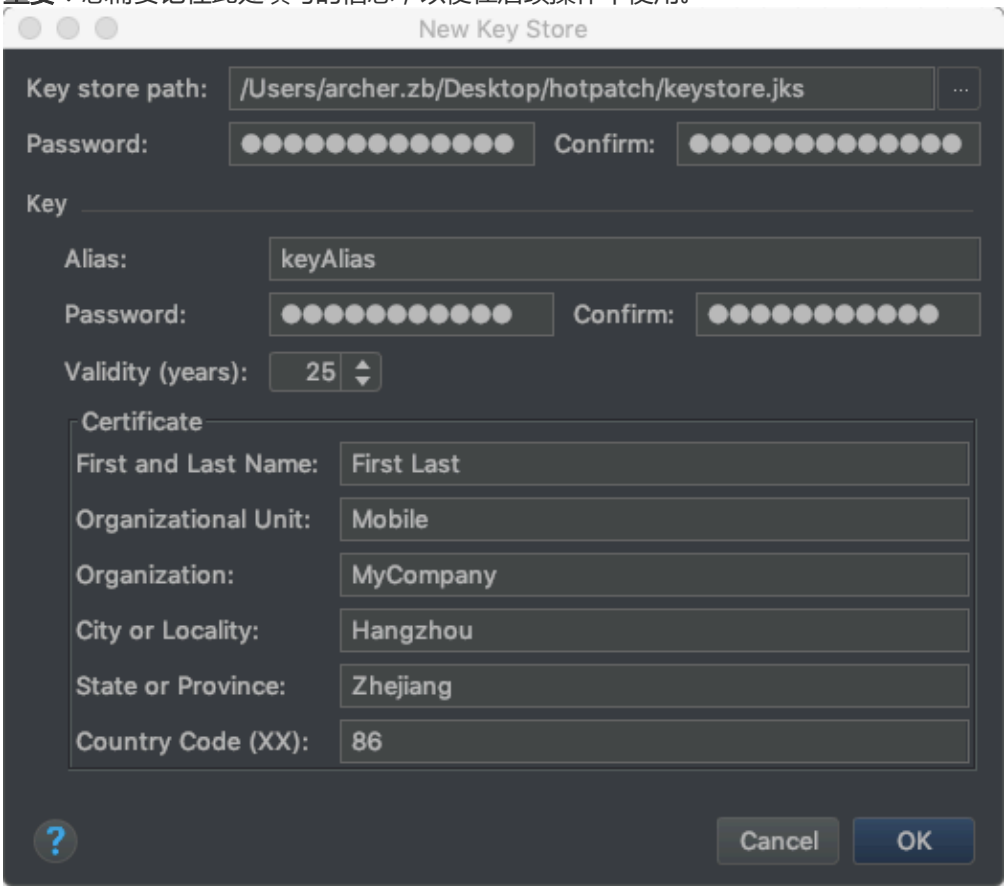
签名步骤如下：

1. 生成密钥和密钥库。若已有秘钥库，则可忽略此步骤。
    - 在 Android Studio 中打开 **Portal 工程**，点击 **Build** > **Generate Signed APK**，然后点击 **Create new**。

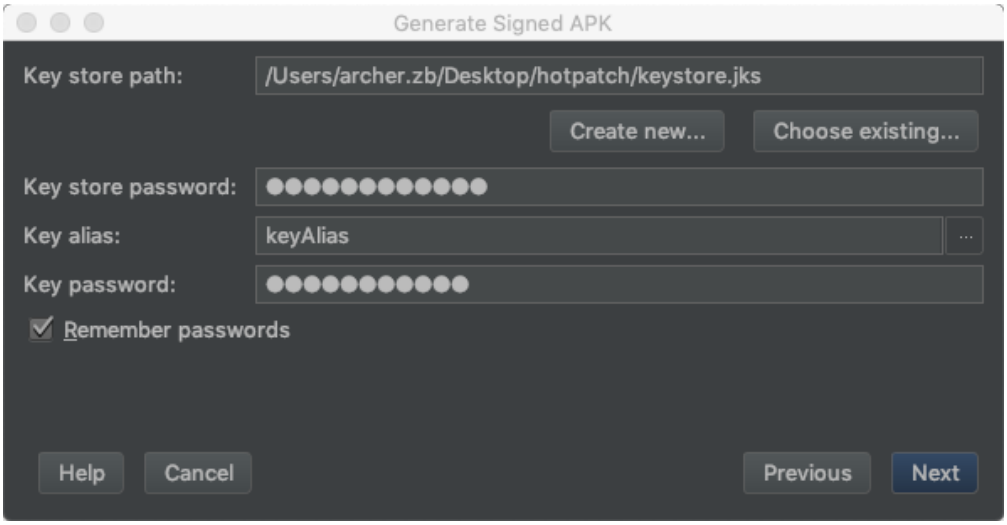


- 完善相关信息，然后点击 **OK**。

重要：您需要记住此处填写的信息，以便在后续操作中使用。



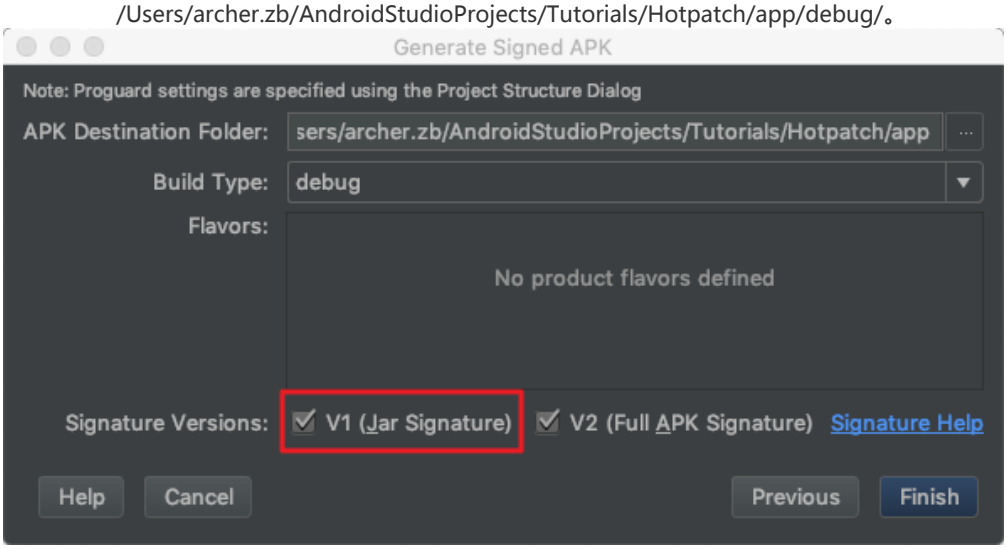
2. 生成带签名的 APK。
   - 选择密钥库，填写相关信息，然后点击 **Next**：



- 如下图完善相关信息，然后点击 **Finish**；等待片刻，即可生成带签名的 APK。

  说明：

  - Signature Versions 中 **V1(Jar Signature)** 必须勾选，**V2(Full APK Signature)** 可按需勾选。
  - 带签名的 APK 将保存在 {APK Destination Folder}/{Build Type} 目录下。在下图的示例中，保存目录为 /Users/archer.zb/AndroidStudioProjects/Tutorials/Hotpatch/app/debug/。



3. 参考 Android 官网文档，在代码中增加签名配置。配置完成后，build.gradle 示例如下：

```
signingConfigs {
release {
keyAlias 'keyAlias'
keyPassword 'keyPassword'
storeFile file('/Users/archer.zb/Desktop/hotpatch/keystore.jks')
storePassword 'storePassword'
}
debug {
keyAlias 'keyAlias'
```

```
keyPassword 'keyPassword'
storeFile file('/Users/archer.zb/Desktop/hotpatch/keystore.jks')
storePassword 'storePassword'
}
}
```

配置加密信息

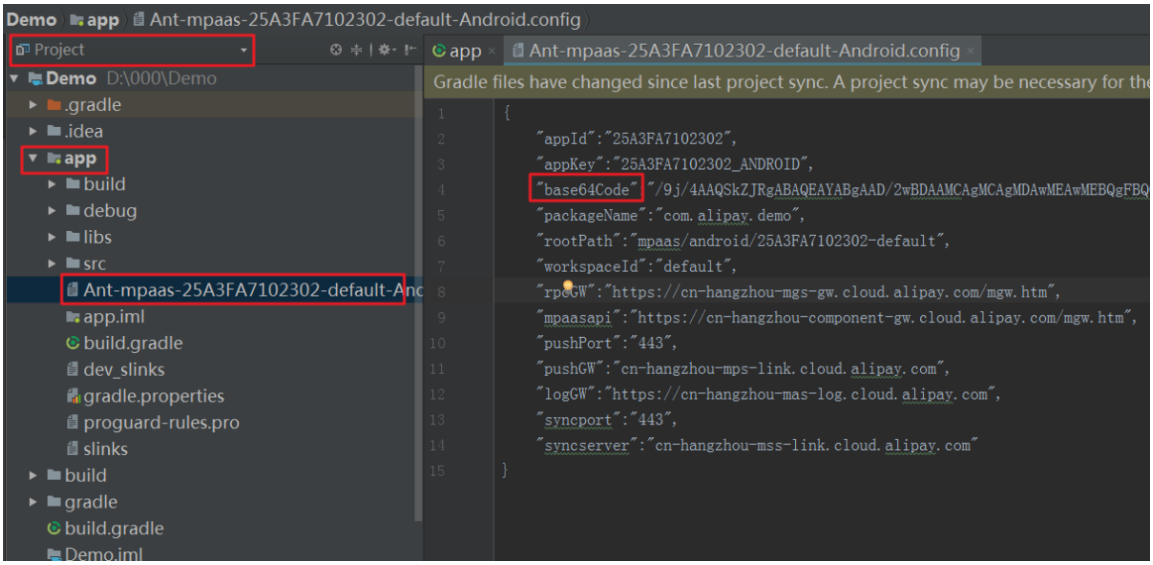
为了保证客户端获取热修复包过程的安全，需要对网络内容进行加密。配置加密信息的步骤如下：

到控制台重新下载配置文件。

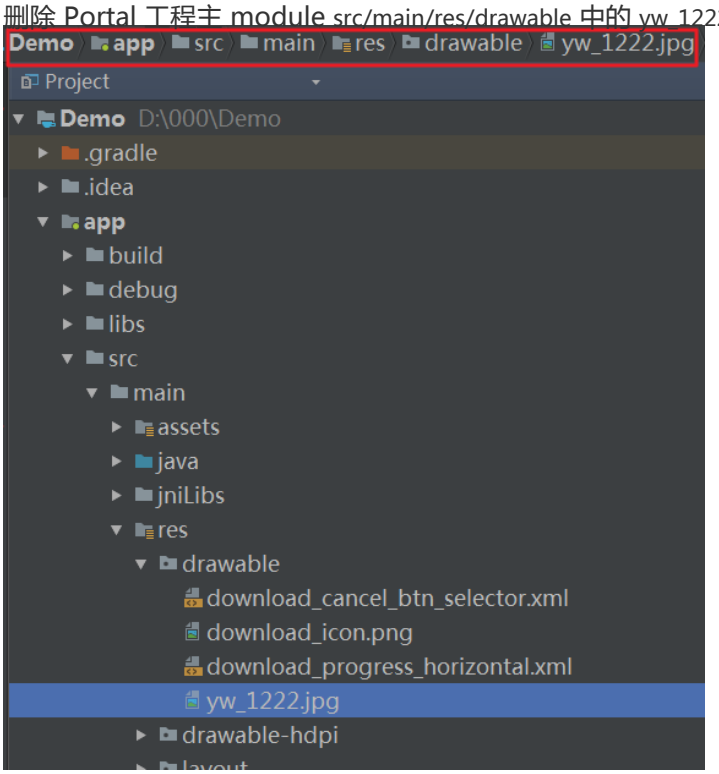
进入 mPaas 控制台 **代码管理** > **代码配置** > **Android** 页面，上传 上一步 生成的带签名 APK，并再次下载配置文件。此时，将下载到 .zip 压缩包。

2. 解压 .zip 包，用其中的 Ant-mpaas-xxx-xxx-Android.config 文件替换 Portal 工程主 module 中的同名文件。

   重要：替换后的 base64Code 的值一定非空。



3. 删除 Portal 工程主 module src/main/res/drawable 中的 yw_1222.jpg 图片：



4. 分别重新构建 Bundle、Portal 工程。

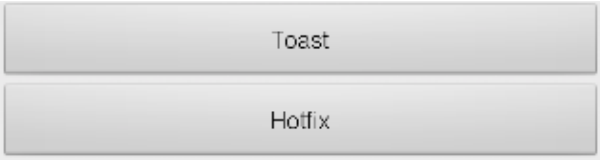
编写代码

至此，热修复就已经接入，您可以根据业务需求编写代码。

示例按钮

说明：下方示例为体验热修复功能的示例代码，以便您在发布前体验热修复功能。

为了体验热修复，您可以在 Bundle 工程的界面中增加两个按钮：



- **Toast**：该按钮对应的代码存在 Bug，点击会造成应用崩溃。
- **Hotfix**：在控制台发布热修复包后，点击该按钮，触发热修复；重启应用后，Toast 按钮对应的代码 Bug 将被修复。

示例代码

对应的布局代码如下：

```
<Button
android:id="@+id/toast"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="Toast"/>

<Button
android:id="@+id/hotfix"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="Hotfix"/>
```

对应的 Java 代码如下：

```
findViewById(R.id.toast).setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View view) {
// 按钮点击时，先做除法运算，再通过弹出框显示计算结果
int result = 1/0; // 除数为 0，是个 bug，会导致应用崩溃
Toast.makeText(getApplicationContext(),"result =" + result, Toast.LENGTH_SHORT).show();
}
});

findViewById(R.id.hotfix).setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View view) {
// 按钮点击时，触发热修复
// SDK 版本 ≥ 10.1.32 时，调用如下接口：
MPHotpatch.init();
}
});
```

```java
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(com.mpaas.mas.demo.launcher.R.layout.main);

        findViewById(R.id.toast).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                // 按钮点击时，先做除法运算，再通过弹出框显示计算结果
                int result = 1/0; // 除数为 0，是个 bug，会导致应用 Crash
                Toast.makeText(getApplicationContext(), text: "result = " + result, Toast.LENGTH_SHORT).show();
            }
        });

        findViewById(R.id.hotfix).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                // 按钮点击时，触发热修复
                // SDK 版本 ≥ 10.1.32 时，调用如下接口：
                MPHotpatch.init();
                // SDK 版本 < 10.1.32 时，调用如下接口：
                // HotPatchUtils.trigDynamicRelease(getApplicationContext(), true);
            }
        });
    }
}
```

加入示例代码后，重新构建 Bundle 与 Portal 生成的 APK 即可。体验热修复过程详见下方的 热修复 Bug 演示 。

发布带有热修复功能的客户端版本

在编写完客户端代码后，即可将生成的 APK 发布至应用平台，以便 App 用户可下载更新。详情参见 Android 发布管理 或 iOS 发布管理 。

热修复 Bug 演示

热修复 Bug 的示例流程如下：

1. 备份 Bug 版本构建生成的 .jar 包
2. 修改 Bug 代码，生成热修复包
3. 在控制台添加并发布热修复包
4. 客户端调用触发热修复的接口，进而获取热修复包
5. 应用重启后，触发热修复，Bug 被修复

备份 Bug 版本构建生成的 .jar 包

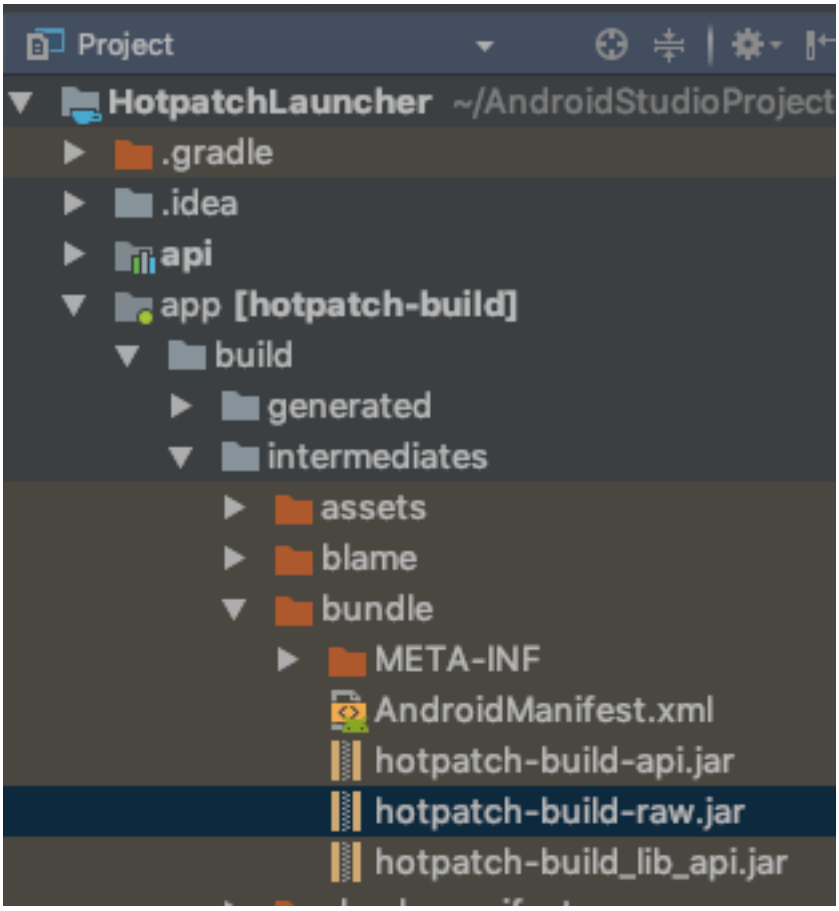
生成热修复包时，需要新老版本（即 Bug 版本和修复后的版本）的构建结果。因此，首先需要备份 Bug 版本构建生成的 .jar 包。

.jar 包路径：

- 若构建 debug 包，则为 Bundle 主 module 下的 build/intermediates/bundle/xxxx-raw.jar。

  若构建 release 包，则 .jar 包名称没有 -raw 后缀。

  示例：



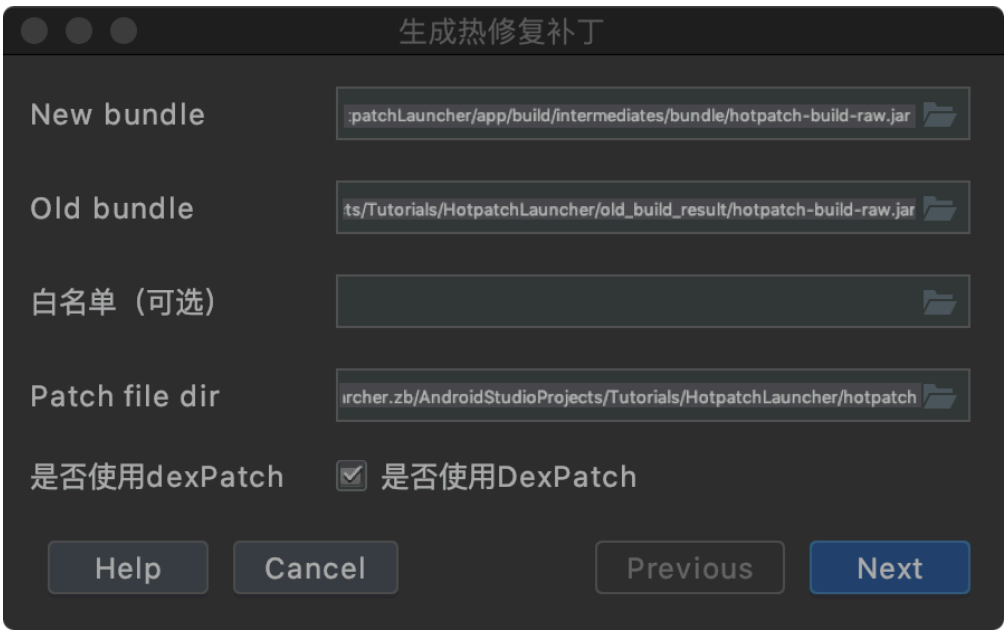
修改 Bug 代码

修改 Bug 代码，并重新构建工程。对应上文示例，可将除数改成 1：

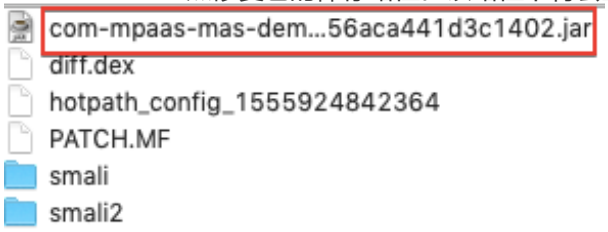
```
findViewById(R.id.toast).setOnClickListener(new View.OnClickListener() {
    @Override
    /**
     * 点击时回调此方法
     */
    public void onClick(View view) {
        // 按钮点击时，先做除法运算，再通过弹出框显示计算结果
        int result = 1/1;
        Toast.makeText(getApplicationContext(), text: "result = " + result, Toast.LENGTH_SHORT).show();
    }
});
```

生成热修复包

在 Android Studio 中，使用 **mPaaS** > **Generate Hotpatch** 功能生成热修复包：



- **New bundle**：修改代码 Bug 后，重新构建生成的 .jar 包。
- **Old bundle**：备份的 Bug 版本构建生成的 .jar 包。详见上文 备份 Bug 版本构建生成的 .jar 包 。
- **白名单（可选）**：用于指定修复的类的配置文件，.txt 格式。该配置文件的编写规则见 白名单配置文件编写规则 。建议您配置白名单，否则可能导致热修复包生成失败。
- **Patch file dir**：热修复包的保存路径。该路径下将会生成很多文件，后续有用的是 .jar 文件：



- **是否使用DexPatch**：建议勾选。

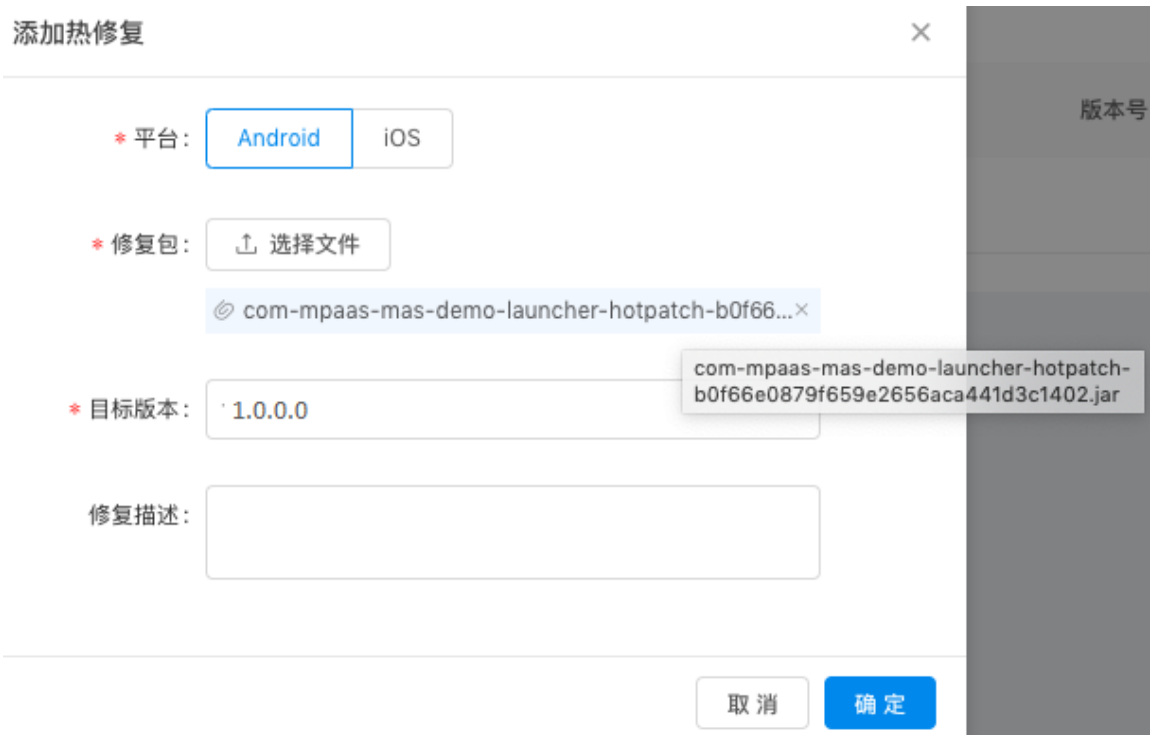
更多信息，参见 热修复包功能 。

在控制台添加并发布热修复包

添加热修复包

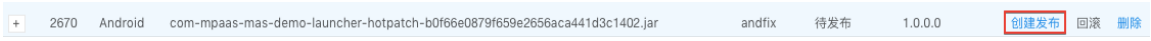
进入 mPaas 控制台 > **实时发布** > **热修复管理** 页面。

点击 **添加热修复包**，然后完善相关信息，并点击 **确定**。



创建发布

如下图，点击 **创建发布**。

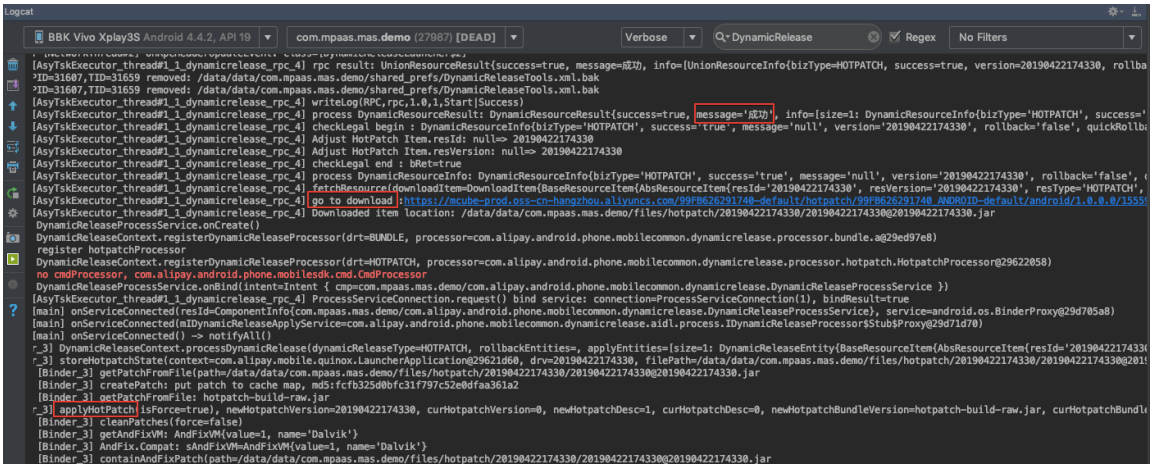


2. 选择发布类型等，然后点击 **确定** 即可完成发布。更多信息，参见 热修复管理 。

App 触发热修复

打开 Android Studio **Logcat**，关键词填写 DynamicRelease，过滤器选择 **No Filters**。

确保手机连接 Android Studio，然后打开手机上安装的 Bug 版 App，点击 **Hotfix** 按钮，可以看到如下日志：



3. 关闭应用进程、重启应用后，点击 **Toast** 按钮，可以正常看到弹出框，说明 Bug 已被修复。

说明：若热修复未生效，且日志中出现 **RPCException [7001]** 异常，则说明签名错误。重复 签名 步骤，并确保：

- Portal 工程主 module Ant-mpaas-xxx-xxx-Android.config 文件中的 base64Code 的值非空。
- Portal 工程主 module build.gradle 文件中 signingConfigs 配置正确。

相关链接

- 热修复使用场景 ：了解热修复的使用场景和限制。
- 接入方式介绍 ：了解原生 AAR 接入、mPaaS Inside 接入和组件化接入（Portal&Bundle）三种接入方式。
- 在客户端创建新工程 ：创建基于 mPaaS 框架的 Android 工程。
- 编译打包 ：使用 mPaaS 插件构建工程。
- mPaaS 插件 > 热修复包功能 ：使用 mPaaS 插件生成热修复包。
- 热修复管理 ：在控制台添加并发布热修复包。

5 离线包管理

5.1 配置离线包

您可以在实时发布平台上传、发布离线包，将离线包快速推送到客户端。关于离线包的详细介绍，请参考 离线包简介 。

在添加离线包之前，您需要添加离线包的相关配置。

操作步骤

进入 mPaaS 控制台，完成以下步骤：

1. 点击左侧导航栏的 **实时发布** > **离线包管理**。
2. 在打开的离线包列表页，点击 **配置管理**。
3. 在 **域名管理** 栏，填写虚拟域名，例如 h5app.com。

说明：

- 为了规范本地文件地址名称，当客户端加载本地离线包文件时，给本地文件绑定虚拟域名作为后缀。
- 虚拟域名不能是以 http 或 https 开头的两级或三级域名。

4. 勾选 **已确认以上信息准确，提交后不再修改**，点击 **保存**。
5. 在 **密钥管理** 栏，上传密钥文件。

说明：

- 如果客户端收到离线包后关闭验签，此处可以不上传密钥文件。
- 此处上传的文件是利用 OpenSSL 生成的 RSA 私钥，用来对离线包进行加签，在客户端利用对应的公钥进行签名验证。您可按照以下方法生成私钥文件和公钥文件：

```
生成私钥:
openssl genrsa -out private_key.pem 2048
生成公钥：
openssl rsa -in private_key.pem -outform PEM -pubout -out public.pem
```

6. 勾选 **已确认以上信息准确，提交后不再修改**，点击 **上传**，即可完成配置离线包。

后续步骤

生成离线包

5.2 生成离线包

根据不同需求，您可以将不同的业务封装成为一个离线包，通过发布平台下发对客户端资源进行更新。生成一个离线包主要分为以下 2 步：

1. 构建前端
2. 在线生成

构建前端 .zip 包

根据离线包使用的场景不同，配置路径分为以下两种：

- 全局资源包
- 普通资源包

说明：

- 在同一个 H5 离线包中，全局资源包与普通资源包不可共存。
- 离线包 ID（即下文中的一级目录）必须为 8 位数字。

全局资源包

可以将被其他多个普通资源包引用的通用资源放置在全局资源包内，并按下列规则指定包内的资源路径。

- 一级目录：全局资源包的 ID，如 77777777。
- 二级目录：指向资源可访问的服务器域名地址。

- 公有云：固定为 mcube-prod.oss-cn-hangzhou.aliyuncs.com（在 **离线包管理** > **新增离线包** 页面的 **资源包类型** 中可看到相关提示信息）。
- 专有云：请查询专有云部署的 mdsweb 服务器域名地址。

- 三级目录：appId _ workspaceId，例如 53E5279071442_test。
- 三级目录往后即为业务自定义的公共资源文件。在公共资源文件的文件夹名、文件名以及文件中，避免使用特殊字符。特殊字符是指会被 urlencode 函数转换的字符。

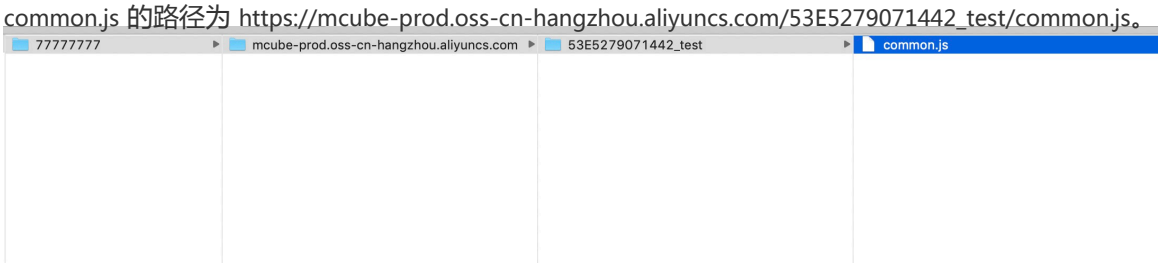
根据以上规则组织资源文件后，即可按照如下格式快速获得资源文件的路径。

- 公有云：http://域名/appID_workspace/资源文件路径。
- 专有云：http://域名/mcube/appID_workspace/资源文件路径。

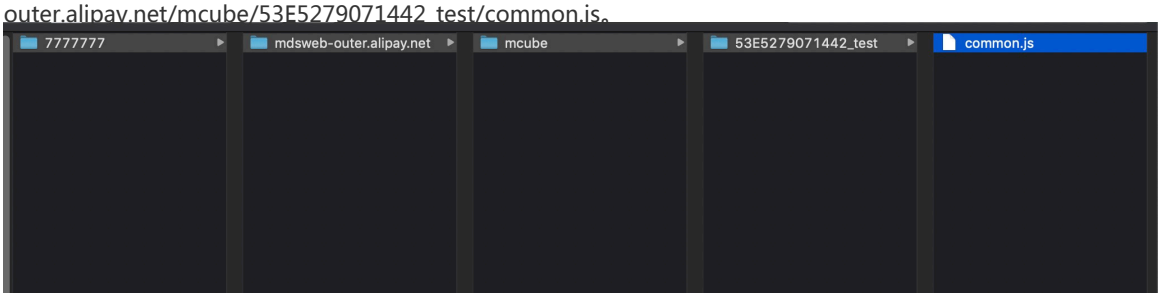
注意：专有云环境下资源文件的路径需要在二级目录（服务器域名）后添加/mcube。

示例：

- 在公有云中，二级目录固定为 mcube-prod.oss-cn-hangzhou.aliyuncs.com，所以下图中资源文件 common.js 的路径为 https://mcube-prod.oss-cn-hangzhou.aliyuncs.com/53E5279071442_test/common.js。

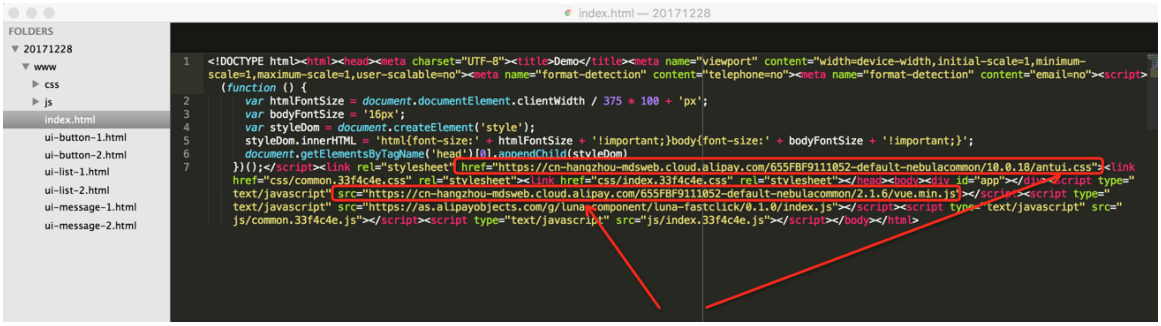


- 在专有云环境中，二级目录为专有云部署的 mdsweb 服务器域名地址，此处以 mdsweb-outer.alipay.net 为例。下图中资源文件common.js 的路径为 https://mdsweb-outer.alipay.net/mcube/53E5279071442_test/common.js。



说明：

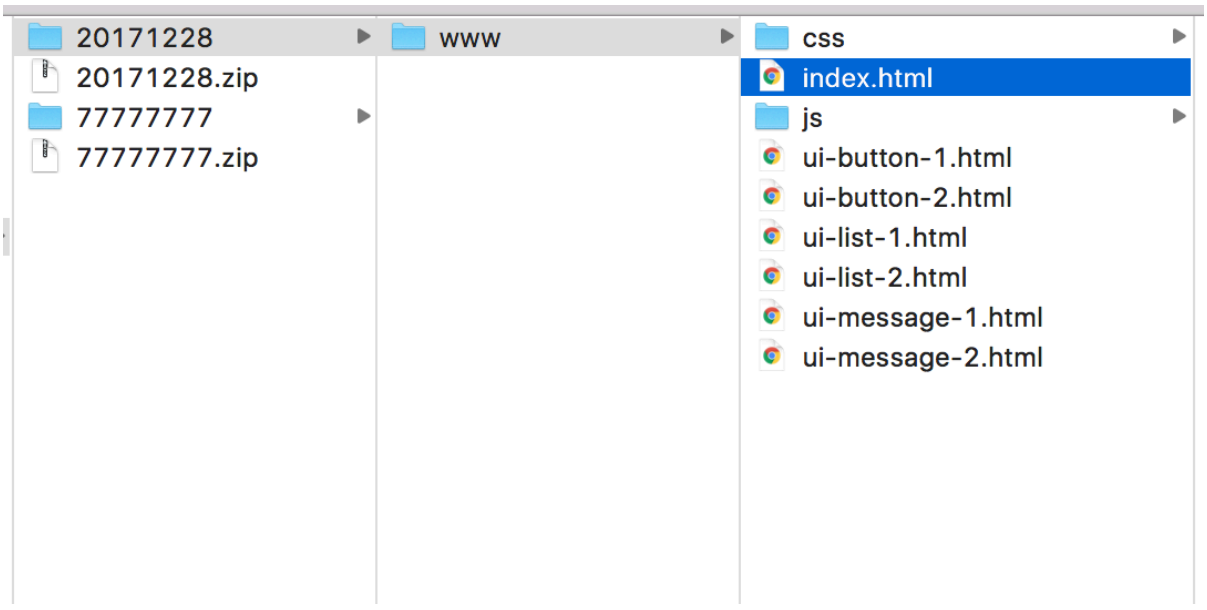
- 公共资源的绝对路径长度不要超过 100 字符，否则会导致客户端加载资源失败以及页面白屏。
- 服务端未控制全局资源包版本，用户可根据实际需求，通过在三级目录以后添加文件目录结构的方式，来自定义控制文件的高低版本。
- 在专有云环境中，如果服务端采用的文件存储格式为 HDFS 或 AFS，则需要在上述第三级目录前增加一个目录，该目录名称为 mdsweb 服务器中的存储空间（bucket）的名称。
- 引用公共资源：在普通离线包内访问全局资源包中的内容，必须通过绝对路径访问，如 https://mcube-prod.oss-cn-hangzhou.aliyuncs.com/53E5279071442_test/common.js。



普通资源包

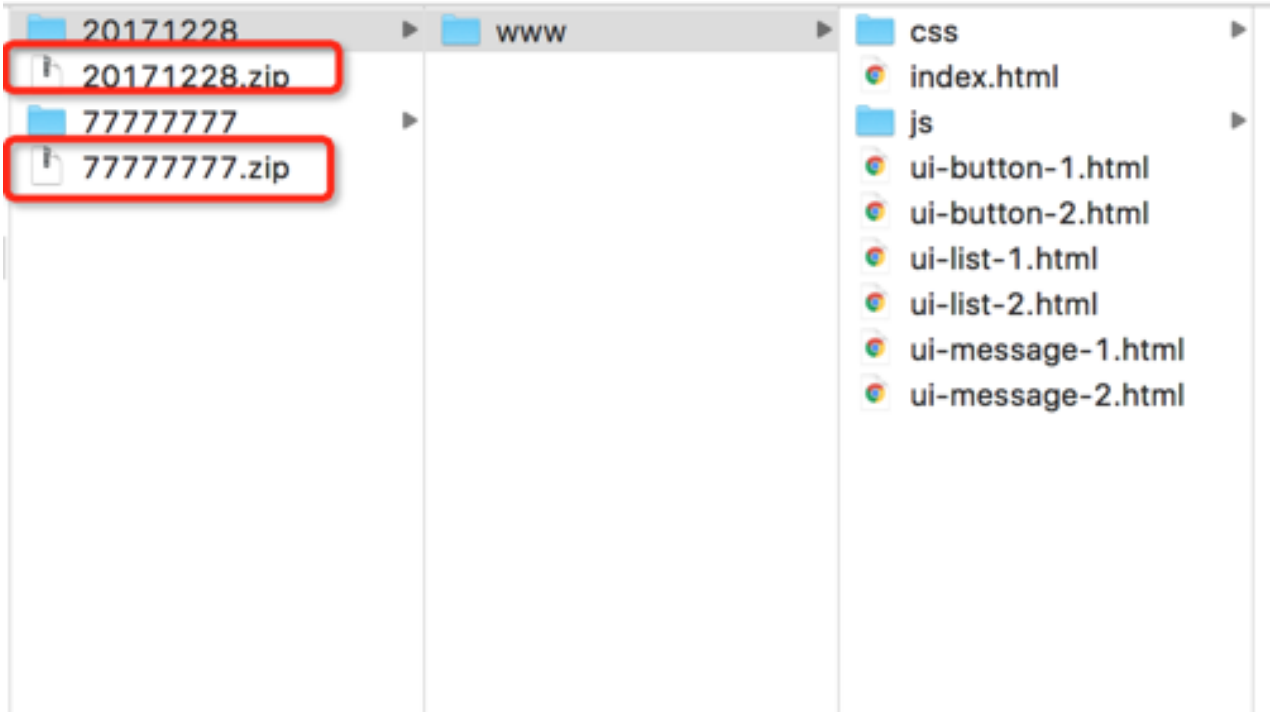
按业务将相关的 HTML、CSS、JavaScript、图片等前端资源放置在同一个离线包内，目录结构如下：

- 一级目录：普通资源包的 ID，如 20171228。
- 二级目录及往后即为业务自定义的资源文件。建议所有的前端文件最好保存在一个统一的目录下，如 /www，并设定当前离线包默认打开的主入口文件，如 /www/index.html。



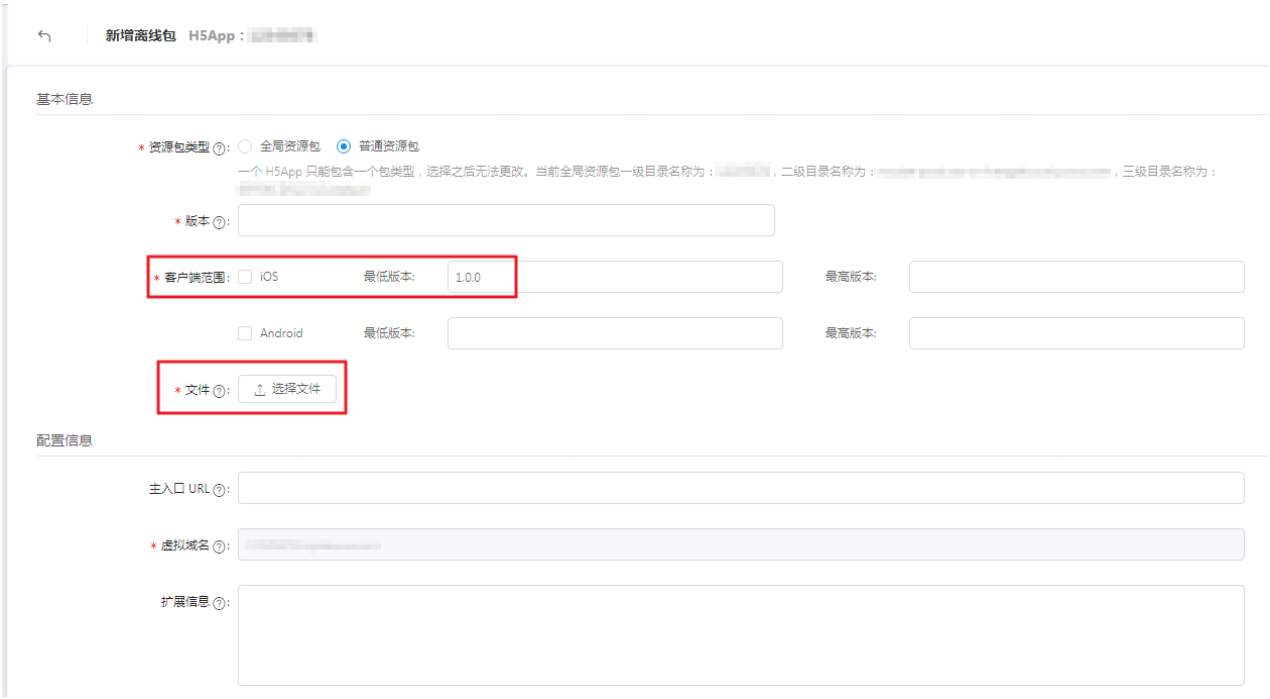
生成 .zip 包

配置完资源包的路径后，即可直接将 appId 所在的目录整体压缩为一个 .zip 包。



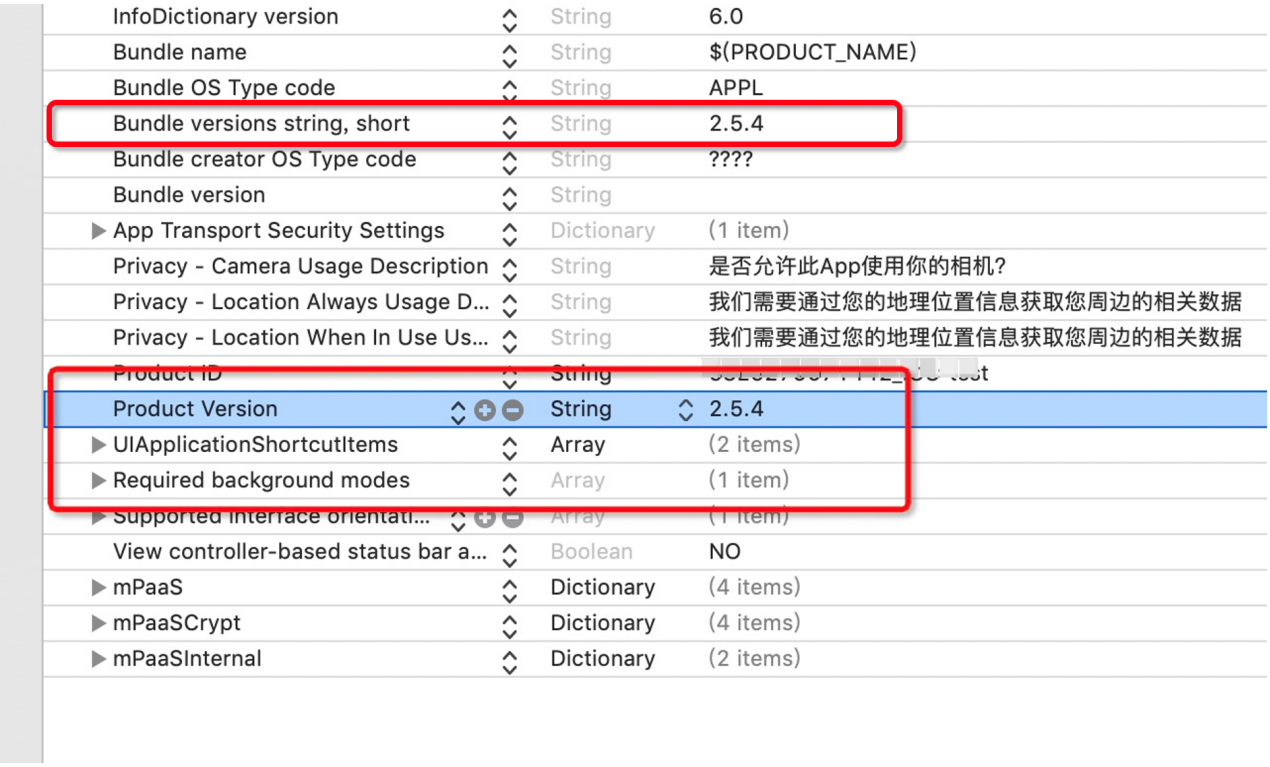
在线生成 .amr 包

进入控制台的 **实时发布** > **离线包管理** 页面，将上一步中生成的 .zip 包上传到 MDS 发布平台，生成 .amr 包。具体操作步骤，参考 实时发布 > 创建离线包 。



重要：

- 在上图的新增离线包配置中，离线包 **客户端范围** 的 iOS **最低版本** 需低于 iOS 客户端 info.plist 文件中的 **Product Version** 字段（见下图），iOS 客户端的 **最低版本** 建议填写 **1.0.0**。
- info.plist 文件中的 **Product Version** 建议与 **Bundle versions string, short** 的值保持一致，否则可能导致离线包不生效。



5.3 创建离线包

在创建离线包资源时，您需要填写基本信息和配置信息。

前置任务

您已经在配置管理界面，完成离线包相关配置。详细信息，参见 配置离线包 。

关于此任务

在首次上传一个 H5App 的离线包时，您必须选择离线包的类型。一旦选择完成不可更改，每个 H5App 有且只有一个离线包类型。

操作步骤

进入 mPaaS 控制台，完成以下步骤：

1. 点击左侧导航栏的 **实时发布** > **离线包管理**。
2. 在打开的离线包列表页，点击 **创建 H5App**。（如果您已创建 H5App，可忽略此步。）
3. 在 **新增 H5App** 窗口，填写 **H5App ID** 和 **H5App 名称**，点击 **提交**。（如果您已创建 H5App，可忽略此步。）

   说明：H5App ID 为 8 位数字。
4. 在离线包列表上方，点击 **添加离线包**，以创建新的离线包。

   在 **基本信息** 栏，完成以下配置：

- **资源包类型**：选择 **全局资源包** 或 **普通资源包**。
- **离线包版本号**：填写离线包的版本号，例如 1.0.0.1。
- **文件**：上传离线包资源文件，文件格式为 .zip。
- **客户端范围**：选择客户端类型，并填入最低版本和最高版本。

说明：至少选择一个客户端类型。只有在此版本范围内的客户端，才能够得到推送的新版本离线包。iOS 客户端版本需低于客户端工程的 info.plist 文件中的 Product Version 字段。

在 **配置信息** 栏，完成以下配置：

- **主入口 URL**：选填，离线包的首页。

  说明：需要填写完整的路径名，如：/www/index.html，其中，/www 为您自定义的二级目录的名称。
- **虚拟域名**：自动显示配置离线包时填写的虚拟域名。
- **扩展信息**：选填，填写页面加载参数，格式为 KV，用逗号（ ，）分隔多个 KV。

  说明：mPaaS 支持配置 H5 离线包的请求时间间隔，可单个配置或全局配置。
  - **单个配置**：即只对当前离线包配置。可在 **扩展信息** 中填入 {"asyncReqRate":"1800"} 来设置请求时间间隔。其中 1800 代表间隔时长，单位为秒，设置范围为 0 ~ 86400 秒（即 0 ~ 24 小时，0 代表无请求间隔限制）。
  - **全局配置**：全局配置需在客户端代码中进行配置，请参见 接入 Android 及 接入 iOS 。
- **下载时机**：选择用户下载该离线包的时机。
  - 若选择 **仅 Wi-Fi**，则只有在 Wi-Fi 网络时会在后台自动下载离线包。
  - 若选择 **所有网络都下载**，则在非 Wi-Fi 网络时会消耗用户流量自动下载，慎用。
- **安装时机**：选择用户安装该离线包的时机。
  - 若选择 **不预加载**，则只有进入离线包或小程序页面时才安装。
  - 若选择 **预加载**，则离线包或小程序下载完成后自动安装。

7. 点击 **提交**，完成离线包创建。

后续步骤

发布离线包

5.4 发布离线包

要发布您已经创建的离线包，您需要创建该离线包的发布任务并完成相关配置。

操作步骤

进入 mPaaS 控制台，完成以下步骤：

1. 点击左侧导航栏的 **实时发布** > **离线包管理**。
2. 在打开的离线包列表页，选择您要发布的离线包与版本，点击 **创建发布**。

3. 在 **创建发布任务** 栏，完成以下配置：
    - 发布类型：选择 **灰度** 或者 **正式** 发布类型。
        - **灰度**：在正式发布前，进行小规模发布以验证新包的功能是否达到预期，发布对象是部分用户。
        - **正式**：正式发布版本，发布对象是全部用户。
    - 发布模型：选择 **白名单** 或者 **时间窗** 发布类型。仅 **灰度** 发布时需设置。
        - 如果选择 **白名单** 发布类型，在 **白名单配置** 中选择白名单。

          说明：关于创建白名单，参见 白名单管理 。
        - 如果选择 **时间窗**，选择 **结束时间** 和 **灰度人数**。
    - 发布描述：填写该离线包发布任务的描述。
    - 高级规则：为该发布任务添加一条或多条高级规则。可选，仅 **灰度** 发布时可设置。
        - 类型：选择 **城市**、**机型**、**网络** 或 **设备系统版本**。
        - 操作类型：选择操作类型。
        - 资源值：在下拉菜单中，选择所选类型对应的资源值。
4. 点击 **确定** 完成发布创建。

结果

在离线包列表页，您可以看到该发布的离线包状态显示 **灰度发布中** 或 **正式发布中**。

说明：由于当前服务器缓存刷新机制原因，在控制台发布离线包后，客户端会在约 1 分钟后才会收到。

后续步骤

管理已发布的离线包

5.5 管理离线包

发布离线包后，您可以管理已发布的离线包。管理操作包括查看、暂停、结束、删除离线包。

查看离线包发布任务

进入 mPaaS 控制台，完成以下步骤：

1. 点击左侧导航栏的 **实时发布** > **离线包管理**。
2. 在打开的离线包列表页，选择您要查看的离线包。
3. 点击下拉按钮（  ）。
4. 点击 **查看**。您可以看到发布任务详情。

暂停离线包发布任务

进入 mPaaS 控制台，完成以下步骤：

1. 点击左侧导航栏的 **实时发布** > **离线包管理**。
2. 在打开的离线包列表页，选择您要暂停发布的离线包。
3. 点击下拉按钮（ ）。
4. 点击 **暂停**，并确认，中止离线包发布。

说明：中止发布任务后，如果您要继续发布该离线包，点击 **继续**。

结束离线包发布任务

进入 mPaaS 控制台，完成以下步骤：

1. 点击左侧导航栏的 **实时发布** > **离线包管理**。
2. 在打开的离线包列表页，选择您要结束发布的离线包。
3. 点击下拉按钮（ ）。
4. 点击 **结束**，并确认，结束离线包发布。

注意：

- 结束发布任务后，如果您要再次发布该离线包，您需要重新创建发布。
- 结束发布后，离线包将不能下载。但是，如果离线包有其他版本在发布中，则该发布版本的离线包依然可以被下载。例如，某离线包结束了 V1.1 版本的发布，而此时 V1.0 版本仍在发布中，则客户端无法下载 V1.1 版本的离线包，但仍然可以下载 1.0 版本的离线包。

删除离线包

进入 mPaaS 控制台，完成以下步骤：

1. 点击左侧导航栏的 **实时发布** > **离线包管理**。
2. 在打开的离线包列表页，选择您要删除的离线包。
3. 点击 **删除**，并确认，删除该离线包。

重要：删除离线包后，该离线包资源无法找回。

6 小程序管理

6.1 配置小程序包

在添加小程序包之前，您需要前往配置管理界面，添加小程序包的相关配置。

关于此任务

为了规划本地文件地址名称，当客户端加载本地小程序包文件的时候，给本地文件绑定虚拟域名作为后缀。

密钥为利用 OpenSSL 生成的 RSA 私钥，用来对小程序包进行加密。在客户端利用对应的公钥进行解密。

按照以下步骤生成私钥文件和公钥文件：

```
生成私钥:
openssl genrsa -out private_key.pem 2048
生成公钥：
openssl rsa -in private_key.pem -outform PEM -pubout -out public.pem
```

操作步骤

进入 mPaaS 控制台，完成以下步骤：

1. 点击左侧导航栏的 **小程序** > **小程序发布**。
2. 在打开的小程序包列表页，点击 **配置管理**。
3. 在 **域名管理** 栏，填写虚拟域名，例如 h5app.com。

   限制：虚拟域名不能是以 http 或是 https开头的两级或三级域名。

4. 在 **密钥管理** 栏，点击 **选择文件** 选择要上传的密钥文件。

   说明：如果客户端收到小程序包后关闭验签，此处可以不上传密钥文件。

5. 勾选 **已确认以上信息准确，提交后不再修改**，点击 **上传**。

后续步骤

创建小程序包

6.2 创建小程序包

在创建小程序包资源时，您需要填写基本信息和配置信息。

前置条件

你已经在配置管理界面，完成小程序包相关配置。欲了解详细信息，参见 配置小程序包 。

操作步骤

进入 mPaaS 控制台，完成以下步骤：

1. 点击左侧导航栏的 **实时发布** > **小程序包管理**。
2. 在打开的小程序包列表页，点击 **新建**。

   说明：
   - **小程序测试包管理** 用于在正式发布前对测试包进行内部自测、验证。
   - **小程序正式包管理** 用于自测无误后的正式包发布。
3. 在 **新建小程序** 窗口，填写小程序的 ID 和小程序名称，点击 **提交**。其中，小程序 ID 为 16 位数字。

   说明：新建小程序后，该小程序在 **小程序正式包管理** 及 **小程序测试包管理** 标签中均可见，您可在对应的标签页中为对应的小程序添加正式包或测试包，两者操作步骤相同。
4. 在小程序 App 列表下，找到新增的小程序，点击 **添加**。

5. 在 **基本信息** 栏，完成以下配置：
    - **版本**：填写小程序包的版本号，例如 1.0.0.1。
    - **客户端范围**：选择客户端类型，并输入最低版本和最高版本。

      说明：至少选择一个客户端类型。
    - **图标**：点击 **选择文件** 上传小程序包的图标。图片格式必须为 png、jpeg、jpg，建议上传像素为 180×180，图标核心图形在白色 160 px 范围内。

      说明：首次创建小程序包时，必须上传该小程序的图标。之后发布该小程序包的其他版本，不上传则默认使用上个版本的图标。
    - **包类型**：定义上传的小程序包属于哪个类型。
        - 在 **小程序测试包管理** 中，可选择 **真机测试包** 或 **真机预览包**。
        - 在 **小程序正式包管理** 中，可选择 **功能包** 或 **插件包**。
    - **文件**：上传小程序包资源文件，文件格式为 .zip。
6. 在 **配置信息** 栏，完成以下配置：
    - **主入口 URL**：必填，小程序包的首页，例如 /index.html#pages/index/index。

      说明：从小程序全局配置文件 app.json 中 page 数组的第一项获取首页地址。更多关于 app.json 的介绍，参见 应用 。
    - **显示底部导航栏**：选择是否在小程序的底部显示导航栏。
    - **显示右上角功能选项**：选择是否在小程序的右上角显示更多的功能选项。
    - **虚拟域名**：自动显示配置小程序包时填写的虚拟域名。
    - **扩展信息**：选填，填写页面加载参数，格式为 KV，用逗号（ ， ）分隔多个 KV。

      说明：mPaaS 支持配置小程序包的请求时间间隔，可单个配置或全局配置。
        - **单个配置**：即只对当前离线包配置。可在 **扩展信息** 中填入 {"asyncReqRate":"1800"} 来设置请求时间间隔。其中 1800 代表间隔时长，单位为秒，设置范围为 0 ~ 86400 秒（即 0 ~ 24 小时，0 代表无请求间隔限制）。
        - **全局配置**：全局配置需在客户端代码中进行配置，请参见 接入 Android 及 接入 iOS 。
    - **下载时机**：选择用户下载该离线包的时机。
        - 若选择 **仅 Wi-Fi**，则只有在 Wi-Fi 网络时会在后台自动下载离线包。
        - 若选择 **所有网络都下载**，则在非 Wi-Fi 网络时会消耗用户流量自动下载，慎用。
    - **安装时机**：选择用户安装该离线包的时机。
        - 若选择 **不预加载**，则只有进入离线包或小程序页面时才安装。
        - 若选择 **预加载**，则离线包或小程序下载完成后自动安装。
7. 勾选 **已确认以上信息准确，提交后不再修改。**
8. 点击 **提交。**

后续步骤

发布小程序包

6.3 发布小程序包

要发布您已经创建的小程序包，您需要创建该小程序包的发布任务并完成相关配置。

操作步骤

进入 mPaaS 控制台，完成以下步骤：

1. 点击左侧导航栏的 **实时发布** > **小程序包管理**。
2. 在打开的 **小程序包列表**页，选择您要发布的小程序包与版本，点击 **创建发布**。
3. 在 **创建发布任务** 栏，完成以下配置：
    - **发布类型**：选择 **灰度** 或者 **正式** 发布类型。
    - **发布模型**：选择 **白名单** 或者 **时间窗** 发布模型。
        - 如果选择 **白名单** 发布模型，在下方的 **白名单配置** 中选择白名单。

          说明：当选中的某个白名单用户数超过 10 万时，仅取前 10 万。
        - 如果选择 **时间窗**，在下方选择 **结束时间** 和 **灰度人数**。
    - **发布描述**：填写该小程序包发布任务的描述。
    - **高级规则**：可选，为该发布任务添加一条或多条高级规则。
        - **类型**：选择 **城市** 或 **机型** 或 **网络** 或 **设备系统版本**。
        - **操作类型**：选择操作类型。
        - **资源值**：在下拉菜单中，选择所选类型对应的资源值。
4. 点击 **确定** 即可发布。

结果

在 **小程序包列表** 页，您可以看到该发布的小程序包状态显示 **灰度发布中** 或 **正式发布中**。同时，在小程序包右侧详情页，将鼠标悬停在 **查看图标**，您可以看到发布的小程序图标。

后续步骤

管理 已发布的小程序包。

6.4 管理小程序包

发布小程序包后，您可以管理已发布的小程序包。管理操作包括查看、暂停、结束、删除小程序包。

查看小程序包发布任务

进入 mPaaS 控制台，完成以下步骤：

1. 点击左侧导航栏的 **实时发布** > **小程序包管理**。

2. 在左侧的小程序包列表中，选择您要查看的小程序包。
3. 在右侧的页面中，点击小程序包版本左侧的下拉按钮（  ）展开更多信息。
4. 点击右侧的 **查看**。您可以看到发布任务详情。

暂停小程序包发布任务

进入 mPaaS 控制台，完成以下步骤：

1. 点击左侧导航栏的 **实时发布** > **小程序包管理**。
2. 在左侧的小程序包列表中，选择您要暂停发布的小程序包。
3. 在右侧的页面中，点击小程序包版本左侧的下拉按钮（  ）展开更多信息。
4. 点击右侧的 **暂停** 。
5. 点击 **确定** 中止小程序包发布。

   说明：中止后，如果您要继续发布该小程序包，点击 **继续**。

结束小程序包发布任务

进入 mPaaS 控制台，完成以下步骤：

1. 点击左侧导航栏的 **实时发布** > **小程序包管理**。
2. 在左侧的小程序包列表中，选择您要结束发布的小程序包。
3. 在右侧的页面中，点击小程序包版本左侧的下拉按钮（  ）展开更多信息。
4. 点击右侧的 **结束** 。
5. 点击 **确定** 结束小程序包发布。

   说明：结束后，如果您要再次发布该小程序包，您需要重新创建发布。

删除小程序包

进入 mPaaS 控制台，完成以下步骤：

1. 点击左侧导航栏的 **实时发布** > **小程序包管理**。
2. 在左侧的小程序包列表中，选择您要删除的小程序包。
3. 点击删除按钮（  ）。
4. 点击 **确定**。

   说明：删除后，该小程序包资源无法找回。

6.5 小程序权限控制

在控制台中，您可对已有的小程序进行权限控制。

关于此任务

为了对小程序进行权限控制，您可在 mPaaS 控制台中为小程序添加 **服务器域名白名单**、**API 调用白名单** 以及 **内嵌 webview 域名白名单**。

- 服务器域名白名单：支持 HTTPS 协议，最多可添加 30 个域名。
- API 调用白名单：管理 API 调用。
- 内嵌 webview 域名白名单：支持 HTTPS 协议。

前置条件

- 您已在小程序包管理中创建了小程序。

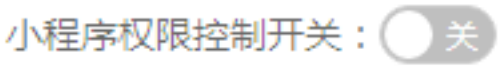
选择小程序

在页面上方，您可以通过下拉列表选择已有的小程序。选择后，下方会展示该小程序的名称及 AppId。

说明：在左侧的小程序包管理标签页中创建的小程序，会实时同步至此下拉列表中。

权限控制开关

通过小程序权限控制开关，您可选择是否启用后续配置的 **服务器域名白名单**、**API 调用白名单** 以及 **内嵌 webview 域名白名单**，以此实现对所选小程序的权限控制。



服务器域名白名单

在下方的白名单配置区域，您可向白名单中添加服务器域名。

添加服务器域名白名单

1. 登录 mPaaS 控制台并选择应用，在左侧导航栏中，选择 **实时发布** > **小程序包管理**。
2. 选择 **开放平台小程序管理** 标签页，并在下方的 **服务器域名白名单** 标签页中点击 **添加**。
3. 在弹出的 **添加服务器域名白名单** 窗口中，输入以下信息：

   - **域名**：必填，此处仅支持 HTTPS 协议的服务器域名，对于非 HTTPS 的域名，在调用时会被拦截。
   - **备注**：选填，输入对此域名的描述信息，最多 200 个字符。
4. 点击 **确定** 完成添加。 **说明**：你最多可在白名单中添加 30 个服务器域名。

编辑、删除服务器域名

白名单列表中的服务器域名均支持编辑，点击右侧的 **操作** 列中的 **编辑**，可更改服务器域名及其描述。

若要从白名单中删除该服务器域名，点击右侧的 **操作** 列中的 **删除**，并在弹出的确认框中点击 **确定**，即可删除此服务器域名。

API 调用白名单

在下方的白名单配置区域，您可向白名单中添加小程序 API。

添加 API 调用白名单

1. 登录 mPaaS 控制台并选择应用，在左侧导航栏中，选择 **实时发布** > **小程序包管理**。
2. 选择 **开放平台小程序管理** 标签页，并在下方的 **API 调用白名单** 标签页中点击 **添加**。
3. 在弹出的 **添加小程序 API白名单** 窗口中，输入以下信息：

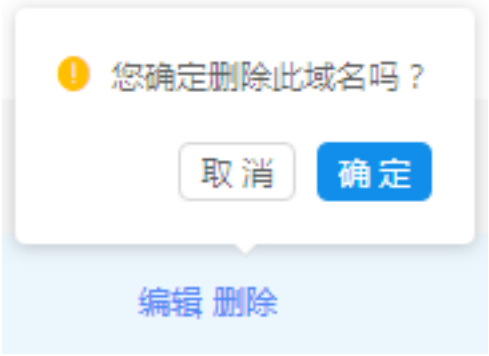
- **API**：要添加至白名单中的 API。
- **备注**：选填，输入对此 API 的描述信息，最多 200 个字符。

4. 点击 **确定** 完成添加。

编辑、删除 API

白名单列表中的 API 均支持编辑，点击右侧的 **操作** 列中的 **编辑**，可更改 API 及其描述。

若要从白名单中删除该 API，点击右侧的 **操作** 列中的 **删除**，并在弹出的确认框中点击 **确定**，即可删除此 API。



内嵌 webview 域名白名单

在下方的白名单配置区域，您可向白名单中添加内嵌 webview 域名。

添加内嵌 webview 域名白名单

1. 登录 mPaaS 控制台并选择应用，在左侧导航栏中，选择 **实时发布** > **小程序包管理**。
2. 选择 **开放平台小程序管理** 标签页，并在下方的 **内嵌 webview 域名白名单** 标签页中点击 **添加**。
3. 在弹出的 **添加 webview 域名白名单** 窗口中，输入以下信息：

   - **域名**：必填，此处仅支持 HTTPS 协议的服务器域名，对于非 HTTPS 的域名，在调用时会被拦截。
   - **备注**：选填，输入对此域名的描述信息，最多 200 个字符。
4. 点击 **确定** 完成添加。

编辑、删除内嵌 webview 域名

白名单列表中的 webview 域名均支持编辑，点击右侧的 **操作** 列中的 **编辑**，可更改 webview 域名及其描述。

若要从白名单中删除该 webview 域名，点击右侧的 **操作** 列中的 **删除**，并在弹出的确认框中点击 **确定**，即可删除此 webview 域名。

7 开关配置管理

7.1 接入 Android

开关配置是一种在客户端不用发布新版本的情况下，动态修改客户端代码处理逻辑的能力。客户端根据拉取后台动态配置的开关值，来控制相关处理。通过开关配置服务，您可以实现各种开关的配置、修改、推送。开关是指 key-value 的键值对。

本文介绍如何集成 mPaaS 提供的开关配置功能。目前，开关配置支持 **原生 AAR 接入**、**mPaaS Inside 接入** 和 **组件化接入** 三种接入方式。

整个过程分为以下三步：

1. 添加 SDK
2. 初始化 mPaaS （仅原生 AAR 接入或 mPaaS Inside 接入需要）
3. 使用 SDK

前置条件

- 若采用原生 AAR 方式接入，需要先 将 mPaaS 添加到您的项目中 。
- 若采用 mPaaS Inside 方式接入，需要先完成 mPaaS Inside 接入流程 。
- 若采用组件化方式接入，需要先完成 组件化接入流程 。

添加 SDK

原生 AAR 方式

参考 AAR 组件管理 ，通过 **组件管理（AAR）** 在工程中安装 **开关配置（CONFIGSERVICE）** 组件。

mPaaS Inside 方式

在工程中通过 **组件管理** 安装 **开关配置（CONFIGSERVICE）** 组件。

更多信息，请参考 管理组件依赖 > 增删组件依赖 。

组件化方式

在 Portal 和 Bundle 工程中通过 **组件管理** 安装 **开关配置（CONFIGSERVICE）** 组件。

更多信息，请参考 管理组件依赖 > 增删组件依赖 。

初始化 mPaaS

如果您使用原生 AAR 接入或 mPaaS Inside 接入方式，则需要初始化 mPaaS。

请在 Application 对象中添加以下代码：

```
public class MyApplication extends Application {

@Override
protected void attachBaseContext(Context base) {
super.attachBaseContext(base);
// mPaaS 初始化回调设置
QuinoxlessFramework.setup(this, new IInitCallback() {
@Override
public void onPostInit() {
// 此回调表示 mPaaS 已经初始化完成，mPaaS 相关调用可在这个回调里进行
}
});
}

@Override
public void onCreate() {
super.onCreate();
// mPaaS 初始化
QuinoxlessFramework.init();
}
}
```

使用 SDK

mPaaS 提供开关配置管理接口 MPConfigService 来实现开关配置。

实现开关配置的操作步骤如下：

1. 在 mPaaS 控制台的 **实时发布 > 配置开关管理** 页面中增加需要的开关配置项，并按照平台、白名单、百分比、版本号、机型、Android 版本等信息进行针对性下发配置。具体操作步骤参考 配置管理 。

   当控制台发布了开关键后，客户端可通过调用接口获取开关键对应的键值。

   开关配置管理接口 MPConfigService 对外暴露了很多接口，根据命名就能了解接口的含义，以下为各个接口及注释：

```
public class MPConfigService {
/**
* 获取开关
*
* @param key
* @return
*/
public static String getConfig(String key);
```

```
/**
* 加载开关，默认达到半小时间隔才会去服务端拉取最新开关。
*/
public static void loadConfig();
/**
* 马上加载开关
*
* @param delay 加载开关的延迟时间，单位毫秒，0 为立刻加载
*/
public static void loadConfigImmediately(long delay);
/**
* 注册开关改变监听器
* @param configChangeListener 监听器
* @return
*/
public static boolean addConfigChangeListener(ConfigService.ConfigChangeListener configChangeListener);
/**
* 移除开关改变监听器
* @param configChangeListener 监听器
*/
public static void removeConfigChangeListener(ConfigService.ConfigChangeListener configChangeListener);
}
```

7.2 接入 iOS

> **重要**：自 2020 年 6 月 28 日起，mPaaS 停止维护 10.1.32 基线。请使用 10.1.68 或 10.1.60 系列基线。可以参考 mPaaS 10.1.68 升级指南 或 mPaaS 10.1.60 升级指南 进行基线版本升级。

开关配置是一种在客户端不用发布新版本的情况下，动态修改客户端代码中的处理逻辑的能力。客户端根据拉取后台动态配置的开关值，来控制相关处理。通过开关配置服务，您可以实现各种开关的配置、修改和推送。开关是指 key/value 的键值对。

mPaaS 提供配置管理服务（ConfigService）来实现开关配置。默认的拉取逻辑为冷启动时拉取一次，或从后台回前台时，若距离上一次拉取时间超过 **半小时**，也会触发一次拉取。同时，配置管理服务也提供了立即拉取的接口以及对配置项改变的监听逻辑，实现配置一旦改变就能立即刷新。

要实现开关配置管理服务，需要添加相关的 iOS SDK，并配置工程、读取配置。

前置条件

您已接入工程到 mPaaS。更多信息，请参见以下内容：

- 基于 mPaaS 框架接入
- 基于已有工程且使用 mPaaS 插件接入
- 基于已有工程且使用 CocoaPods 接入

关于此任务

本文结合 开关配置代码示例，进行详细的说明介绍。

添加 SDK

根据您采用的接入方式，请选择相应的添加方式。

使用 mPaaS Xcode Extension 插件

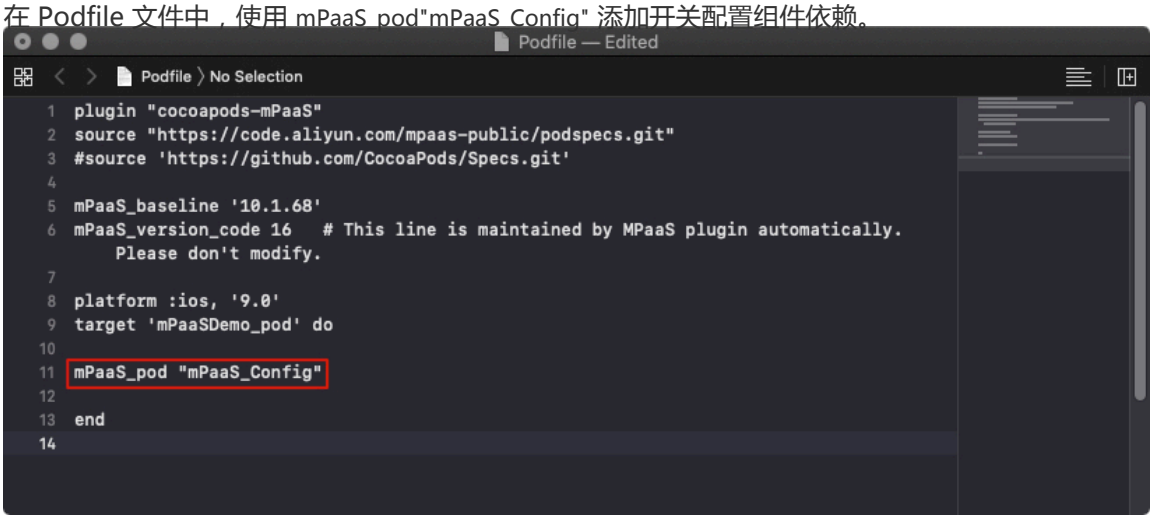
此方式适用于 **基于 mPaaS 框架接入** 或 **基于已有工程且使用 mPaaS 插件接入** 的接入方式。

1. 点击 Xcode 菜单项 **Editor** > **mPaaS** > **编辑工程**，打开编辑工程页面。
2. 选择 **开关配置**，保存后点击 **开始编辑**，即可完成添加。

使用 cocoapods-mPaaS 插件

此方式适用于 **基于已有工程且使用 CocoaPods 接入** 的接入方式。

1. 在 Podfile 文件中，使用 mPaaS_pod"mPaaS_Config" 添加开关配置组件依赖。

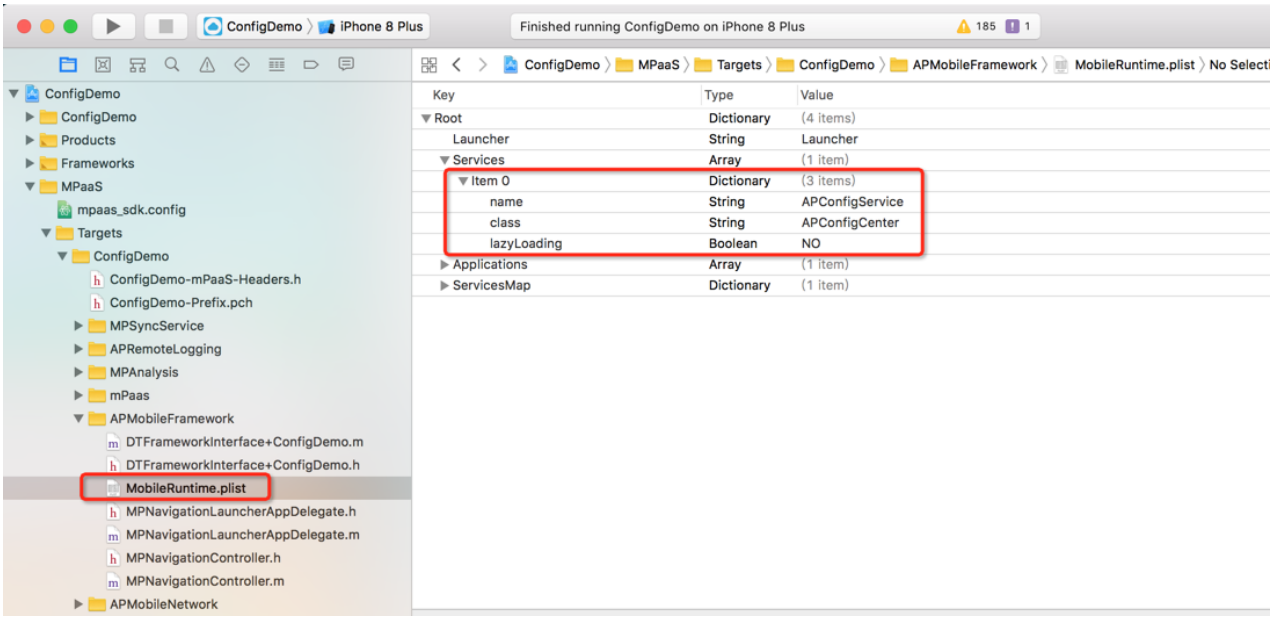


2. 根据需要 执行 pod install 或 pod update 即可完成接入。

配置工程

说明：该步骤适用于 10.1.32 版本。由于 10.1.60 及 10.1.68 版本中内置了工程配置，所以在 10.1.60 及 10.1.68 版本中可忽略此步骤。

mPaaS 将提供的开关配置能力封装为一个 服务 ，使用前需要在服务管理器中注册此服务，如下图所示：



读取配置

开关键对应的值可以通过 mPaaS 控制台动态发布。在左侧导航栏中点击 **实时发布** > **配置管理** > **配置键** 查看具体内容。

后续操作

获取开关值

在 mPaaS 控制台 **实时发布** > **配置管理** 中增加需要的开关配置项，并按照平台、白名单、百分比、版本号、机型、iOS 版本等信息进行针对性下发配置。具体操作步骤，参考 配置管理 。

在控制台发布了开关键后，客户端可通过调用接口获取开关键对应的键值。

```
+ (void)testStringForKey
{
id<APConfigService>configService = [DTContextGet() findServiceByName:@"APConfigService"];
NSString *configValue = [configService stringValueForKey:@"BillEntrance"];
assert (configValue && [configValue isKindOfClass:[NSString class]]);
}
```

说明：开关键值是通过 RPC 拉取返回的，存在一定的失败几率，因此开发者在使用时要考虑到客户端本地的处理逻辑以应对拉取失败的情况。建议在客户端本地逻辑中设置开关默认值，当控制台发布了新开关时采用新的配置逻辑，拉取失败则采用本地默认逻辑。

进阶指南

客户端拉取开关配置的时机：

- 应用冷启动时会拉取。

  回前台时，若距上次请求配置超过 30 分钟，会重新拉取。

  说明：30 分钟为默认的时间间隔，可在控制台的 **实时发布** > **配置开关管理** 页面中添加开关 Load_Config_Interval 修改此时间间隔。具体操作参见 配置管理 。

- 动态监听开关变化
  - 可对指定的 key 添加观察者，动态监听开关值的变化。

  - 当触发客户端拉取开关配置时，可在回调方法里获取指定 key 对应的最新开关值。

- 强制拉取开关值：SDK 提供强制拉取控制台最新配置的方法

7.3 配置管理

作为开发者，您可以在 mPaaS 控制台 **实时发布** > **配置开关管理** 中增加需要的开关配置项，并且按照平台、白名单、百分比、版本号、机型、Android 或 iOS 版本等信息进行针对性地下发配置。

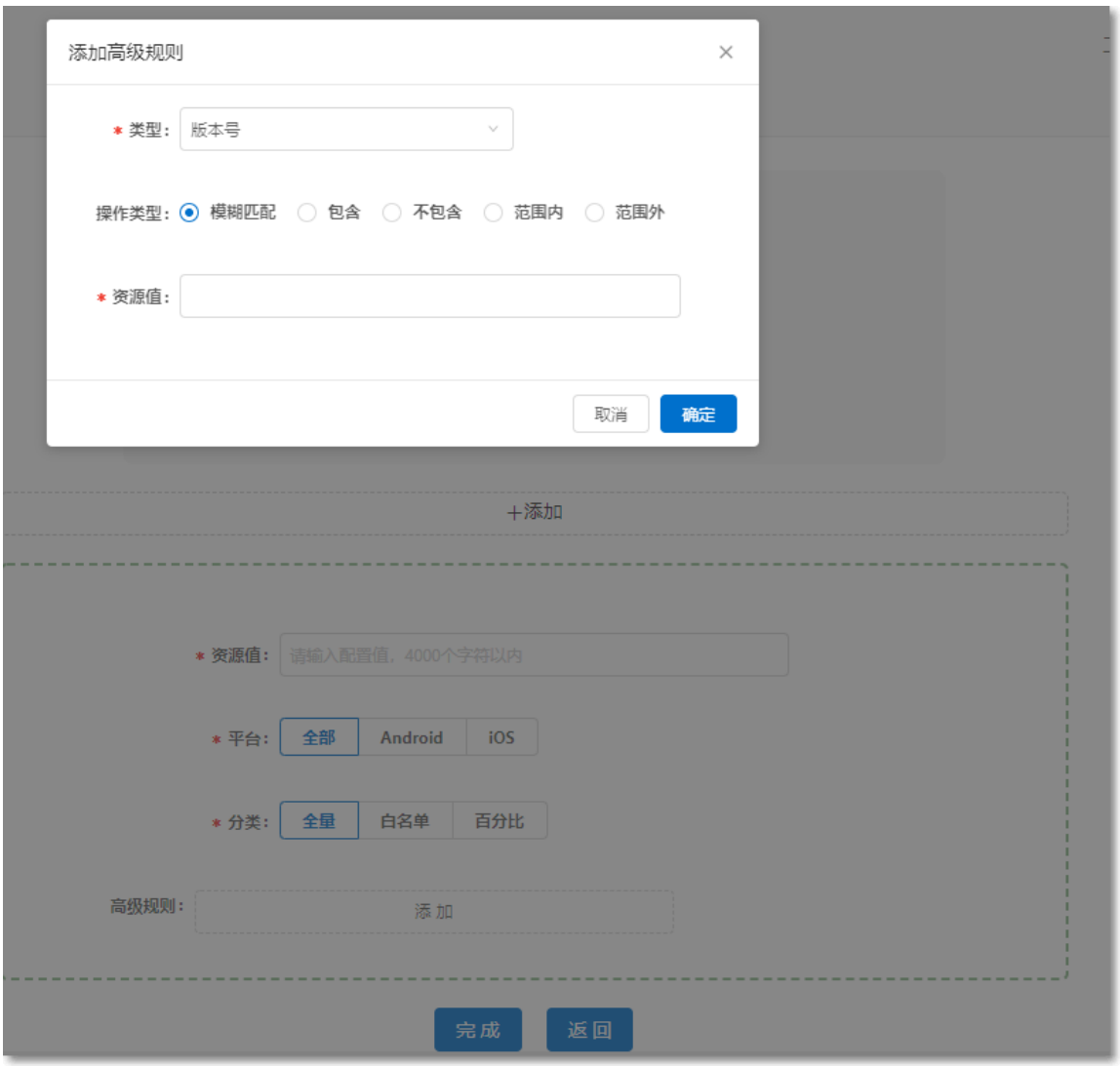
前提条件

在 Android 或 iOS 客户端添加开关配置服务的 SDK。

操作步骤

进入 mPaaS 控制台，完成以下步骤：

1. 在左侧导航栏，点击 **实时发布** > **配置开关管理**。
2. 点击 **添加配置** 增加新的配置。
3. 添加开关配置键、资源值以及相应的配置类型，包括平台和分类。此外，点击添加 **高级规则** 可选择版本号、osVersion、机型等信息进行有针对性地下发。



后续操作

当控制台发布了开关键后，客户端可通过调用接口获取开关键对应的键值：

- Android 客户端
- iOS 客户端

8 白名单管理

删除白名单

要删除白名单，单击白名单列表中指定白名单右侧的 **删除**，删除该白名单。

9 发布规则管理

资源配置管理是实时发布的一项基础功能，用户可以预先定义实时发布所需要的各种配置数据，无需每次手工输入，提升效率，降低出错可能性。

各种配置数据也称为资源，比如城市，机型等。在增加配置时，资源名称是展示给用户看的，资源值才是真正和客户端的请求参数进行匹配的值。

在资源配置管理界面上，您可以进行以下操作：

- 添加资源
- 修改资源配置
- 删除资源

添加资源

进入 mPaaS 控制台，单击左侧导航栏中的 **实时发布** > **发布规则管理**， 进入资源配置列表页面。

在资源配置列表页中单击 **添加资源**，在弹出的窗口中选择资源类型和平台类型，输入资源名称和资源值，然后单击 **确定**，完成资源创建。

- **资源类型**：支持四种资源类型，包括城市、机型、网络和设备系统版本。

- **平台类型**：选择移动端平台，可以是 Android、iOS 或不区分平台。
- **资源名称**：自定义，用来展示，一般与资源值保持一致。

资源值：不支持同时填写多个资源值。各类型资源值说明如下：

- 城市：地、市级别的城市名称，名称中需包含行政单位（市、地区、自治州、盟），例如：上海市、海东地区、黔南布依族苗族自治州、兴安盟。
- 机型：移动设备的机型，例如 VIVO X5M、IPHONE 6S。
- 网络：网络类型，如 2G、3G、4G、5G、WIFI、WWAN。
- 设备系统版本：移动设备的系统版本，例如 10.0.1、5.1.1。

如果不清楚移动设备的机型、网络、设备系统版本信息，可以通过调用接口获取移动设备客户端相关信息。具体参考下文的 调用接口获取资源配置 。

修改资源配置

要修改资源配置信息，单击资源配置列表中指定资源右侧的 **修改**，对该资源配置进行编辑。编辑完毕后，单击 **确定** 以保存修改。

删除资源

要删除资源配置信息，单击资源配置列表中指定资源右侧的 **删除**，删除该资源。也可以在列表中同时选中多个资源，单击 **批量删除**，确定后即可删除资源。

调用接口获取资源配置

在添加资源时，如果不清楚网络、机型、设备系统版本对应的具体资源值时，可以通过调用相应的接口来获取相关信息。

具体操作如下：

在本地工程中，调用以下接口，获取移动客户端的相关信息。

Android 客户端

```
DeviceInfo deviceInfo = DeviceInfo.createInstance(context);
AppInfo appInfo = AppInfo.createInstance(context);

deviceInfo.getOsVersion(); //设备系统版本
deviceInfo.getmMobileModel(); //机型
appInfo.getmProductVersion(); //产品版本

int networkType = NetworkUtils.getNetworkType(context);//网络类型
networkType = 1（2G）
networkType = 2（3G）
networkType = 3（WIFI）
networkType = 4（4G）
```

iOS 客户端

类型	网络	设备系统版本（系统接口）	机型（mPaaS封装接口）
开关配置	无	[[UIDevice currentDevice] systemVersion]	[APMobileIdentifier shareIdentifier].deviceModel
升级检测	无线：WIFI 移动网络：WWAN	[[UIDevice currentDevice] systemVersion]	[APMobileIdentifier shareIdentifier].deviceModel
热修复管理 离线包管理 小程序管理	[DTReachability networkName]	[[UIDevice currentDevice] systemVersion]	[APMobileIdentifier shareIdentifier].deviceModel

通过日志将客户端资源信息上报至服务端，然后通过服务端查看相应的资源配置信息。

10 常见问题

iOS 客户端

热修复错误码

以下表格显示错误码及其含义。

错误码	含义
300	未知错误
301	本地 patch 文件为空。请检查 patch 文件是否成功下载，或本地测试时文件路径是否正确。
302	.js 文件解析错误。请检查由 OC 转化的 .js 文件是否正确。
303	本地 patch 文件为空。请检查 patch 文件是否成功下载，或本地测试时文件路径是否正确。
304	patch 文件解密失败。请检查无线保镖验签图片 yw_1222.jpg 是否正确。
305	patch 文件解压失败。请重试。
306	patch 文件 MD5 校验失败。请确认发布的 patch 文件是否为加密后的 .js 格式。

本地测试由 OC 转化后的 .js 文件，为什么修复没有生效？

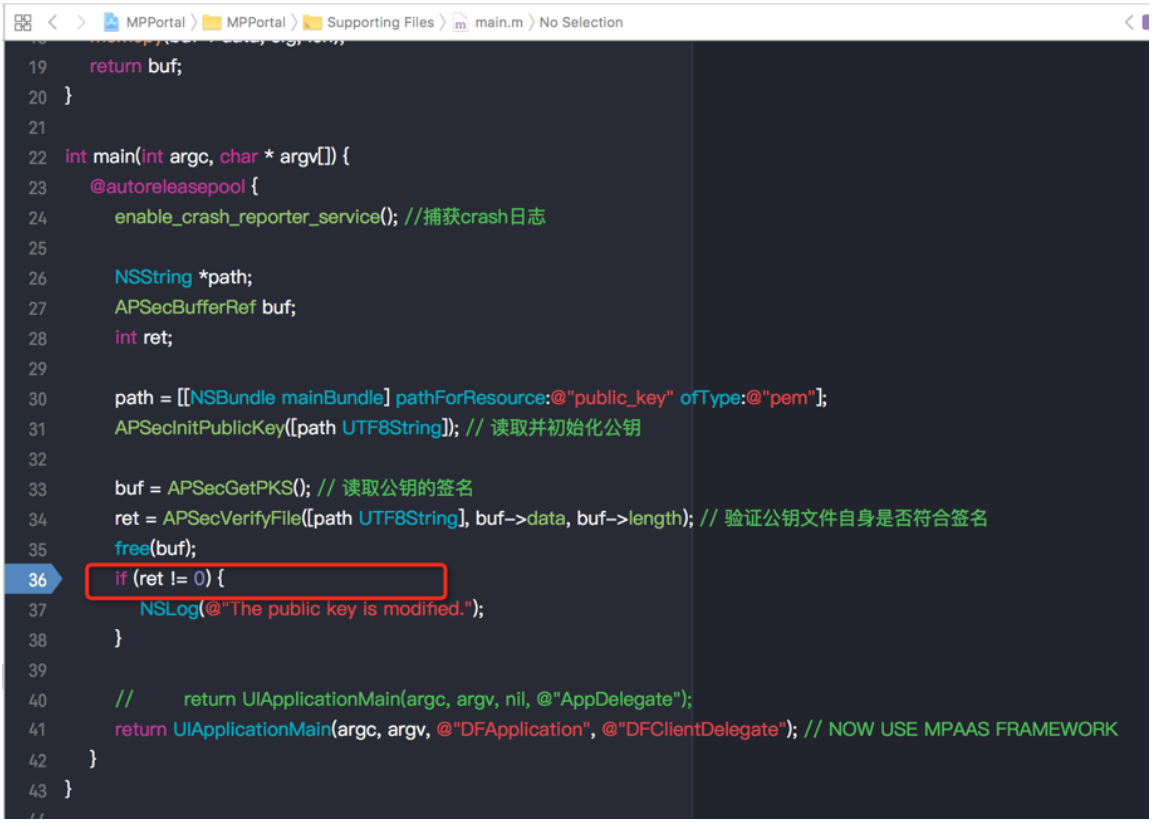
查看下图中调用的方法返回的错误信息，若不为 nil，根据上文的错误码进一步排查。

```
NSString *jsFile = [[NSBundle mainBundle] pathForResource:@"Test"ofType:@"js"];
NSError *errorJS = [MPDynamicInterface runWithResultDynamicLocalFile:jsFile];
```

若上一步无报错，请检查 .js 文件语法是否正确，可参考 OC 转 JS 语法。

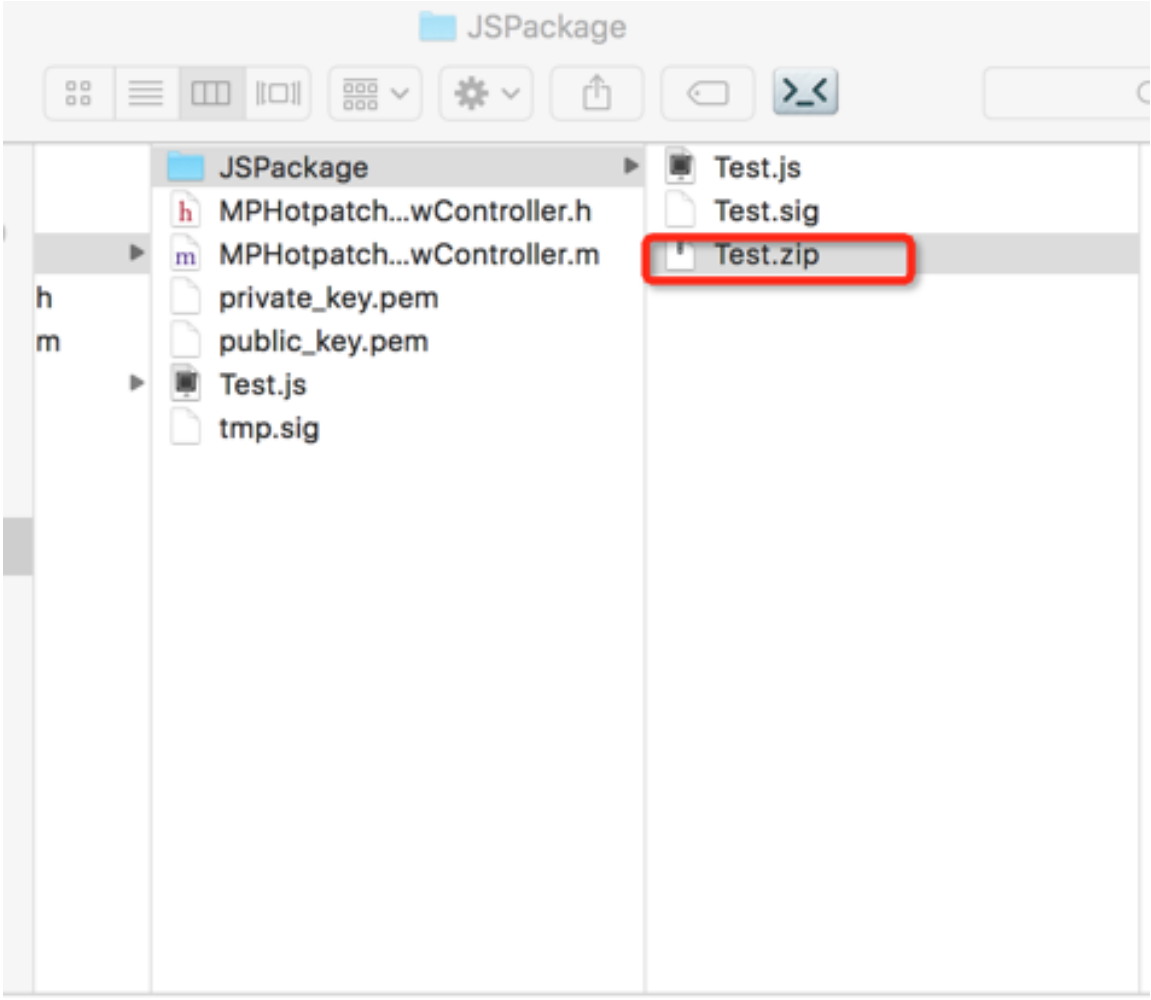
本地测试加密后的 .zip 文件，为什么修复没有生效？

查看工程中 RSA 非对称加密信息是否正确，可通过下图中 ret 的值是否为 0 判断。



确保工程中的无线保镖图片是正确的且与工程中的 meta.config 是匹配的。如无法确定，可尝试重新生成无线保镖图片以及更新热修复 .zip 文件。

确保测试的是加密之后的 .zip 文件。



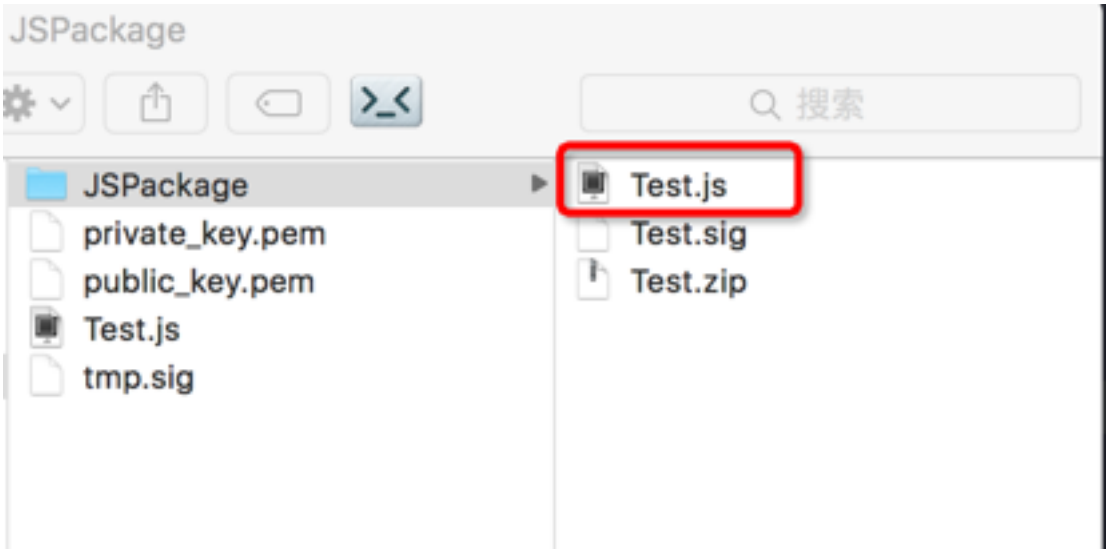
查看下图中调用的方法返回的错误信息，若不为 nil，根据上文的错误码进一步排查。

```
NSString *jsZip = [[NSBundle mainBundle] pathForResource:@"Test"ofType:@"zip"];
NSError *error = [MPDynamicInterface runWithResultDynamicLocalSecFile:jsZip];
```

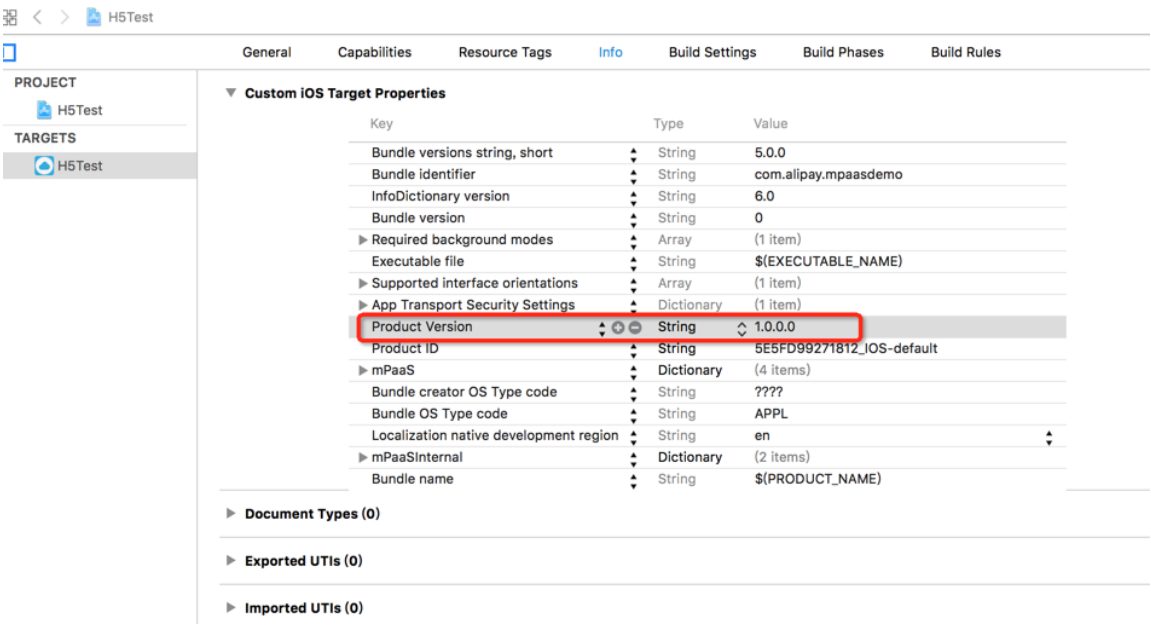
在发布平台下发加密后的 .js 文件后，为什么客户端没有修复生效？

原始 .js 文件加密后，本地验证 .zip 文件是否修复生效。

在控制台上传新的修复任务时，确保上传到发布平台的修复包是加密后的 .js 文件。



目标版本需与工程中 info.plist 文件中 Product Version 字段保持一致。

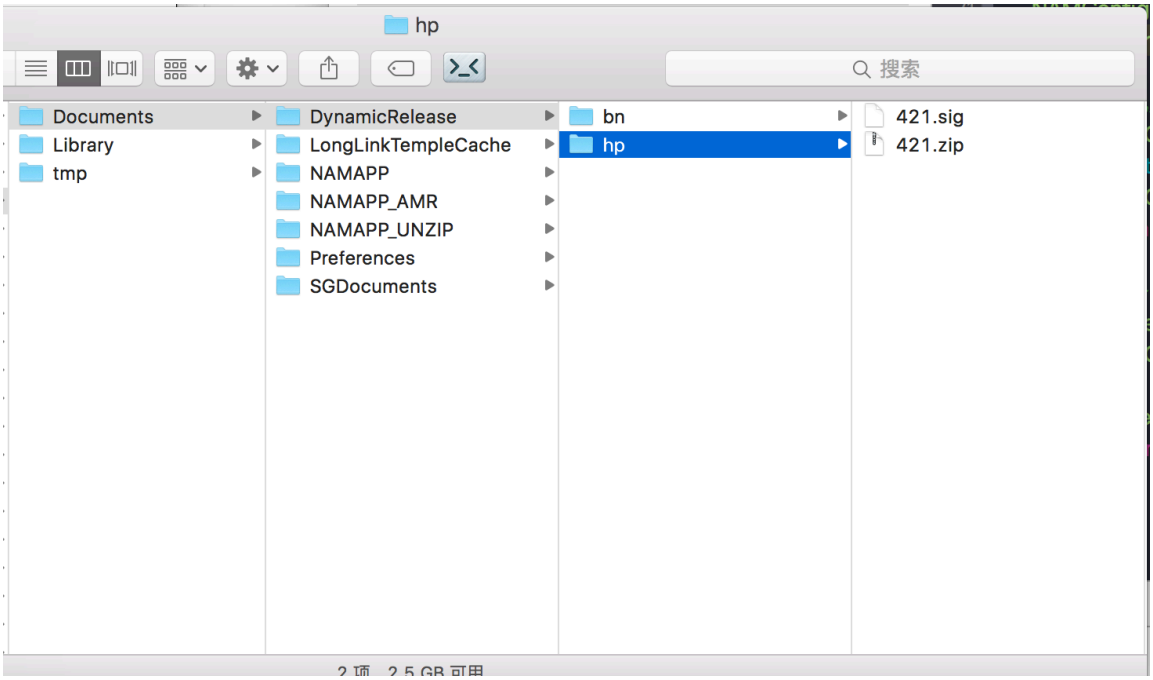


在 Xcode 控制台查看 alipay.client.getUnionResource 的网关返回结果是否为 1000，保证网络请求成功。

```
,len:228
2017-05-10 12:47:36.238 H5Test[4539:1678846] alipay.client.getUnionResource,cdn:0,etag:0 spdy:0 pb0-
Response:
{
    resultStatus = 1000;
}
```

- 若网关返回结果不是 1000，请根据 移动网关 > 客户端编程 > FAQ 进一步排查。

网络请求成功后，在本地沙盒路径中查看 patch 包是否已下发到本地，若目录下已生成了 .zip 和 .sig 文件，则表示客户端已获取到发布平台下发的脚本，杀进程重启应用后 patch 即可生效。



Android 客户端

使用热修复后，和 RPC 有关的调用发生 apache http 相关的 crash。

请使用 Android 官网上的方式引入 apache http client，禁止使用导入 Jar 包或者 gradle implementation/compile 的方式导入 http client。否则会引起 classloader 加载类混乱。

内部类的白名单热修复

内部类的引用需要完全限定名。如果一定要修复内部类，最简单的方式是反编译成 smali，smali 的文件名就是内部类的类名。

11 参考

11.1 API 说明

了解 Android 的升级 SDK 中相关 API 接口的使用方法。

- MPaaSCheckVersionService API
- MPaaSCheckCallBack API

MPaaSCheckVersionService API

checkNewVersion

检查应用是否有更新，该方法启动异步任务执行更新检查，无论是否有更新，都会调用 MPaaSCheckCallBack 的相应回调方法：

```
void checkNewVersion(Activity activity)
```

setIntervalTime

设置单次提醒的间隔时间：

```
void setIntervalTime(long interval202)
```

默认是 3 天，单位：毫秒。

setMPaasCheckCallBack

设置升级 SDK 检测更新时调用的回调实例：

```
void setMPaaSCheckCallBack(MPaaSCheckCallBack mPaaSCheckCallBack)
```

installApk

安装新版本安装包，可在MPaaSCheckCallBack.alreadyDownloaded 方法中调用：

```
void installApk(String filePath)
void installApk(ClientUpgradeRes res)
```

update

执行下载安装包请求，可在 MPaaSCheckCallBack.showUpgradeDialog 方法中调用：

```
void update(ClientUpgradeRes res)
```

MPaaSCheckCallBack API

startCheck

调用检测升级接口后被调用，接入方可以在此方法内提示用户加载中：

```
void startCheck()
```

isUpdating

当重复调用检测升级接口时被调用：

```
void isUpdating()
```

onException

当检测升级过程中发生异常时调用：

```
void onException(Throwable throwable)
```

dealDataInValid

检测升级返回的升级信息有效时被调用：

```
void dealDataInValid(Activity activity, ClientUpgradeRes result)
```

dealHasNoNewVersion

检测升级返回的升级信息无效时被调用：

```
void dealHasNoNewVersion(Activity activity, ClientUpgradeRes result)
```

alreadyDownloaded

检测升级时发现新版本安装包已经下载完成时被调用。接入方可以在此时提示用户安装升级包。如果选择安装，调用 MPaaSCheckVersionService.installApk 方法安装：

```
void alreadyDownloaded(Activity activity, ClientUpgradeRes result)
```

showUpgradeDialog

当检测到新版本信息但未下载完安装包时被调用，接入方可在此时提示用户是否升级，如果选择升级的话，调用 MPaaSCheckVersionService.update 方法触发下载任务：

```
void showUpgradeDialog(Activity activity, ClientUpgradeRes result)
```

onLimit

当检测到新版本信息但距上次检测的时间小于设定间隔时间时被调用，仅在配置为 **单次提示** 时有效：

```
void onLimit(Activity activity, ClientUpgradeRes result, String reason)
```

11.2 代码示例

11.2.1 版本升级代码示例

Android 代码示例

要查看该功能在移动设备中的样式和交互效果，下载 Android 代码示例，在本地 Android Studio 中编译 bundle，并安装 .apk 文件到您的 Android 移动设备中。要了解详细信息，查看 获取代码示例。

iOS 代码示例

检测升级

通过调用升级检测接口，mPaaS 会在后台自动连接 mPaaS 发布功能，检测是否有新版本。如有新版本，则会自动跳出默认升级窗口提醒用户升级。用户点击 **升级** 自动升级，无需其他编码。如需自定义升级提示窗口，请参考下方 自定义升级提示 UI 。

```
- (void)checkUpdate
{
UpgradeCheckService *service = [UpgradeCheckService sharedService];
service.delegate = self;
[service checkUpgradeAndShowAlertWith:YES];
}
```

说明：添加 SDK 时，会自动添加对发布服务网关的依赖 mPaaS > Targets > MPHttpClient > DTRpcInterface+upgradeComp.m，所以您只需调用 checkUpgradeAndShowAlertWith 方法即可，发布组件会自动在后台连接发布服务。

自定义升级提示 UI

通过实现 delegate 可以自定义升级检测 UI。

```
# pragma mark UpgradeViewDelegate
- (UIImage *)upgradeViewHeader
{
return [UIImage imageNamed:@"FinancialCloud"];
}
- (void)showProgressHUD:(BOOL)animation
{
self.toast = [APToastView presentToastWithin:self.view withIcon:APToastIconLoading text:nil];
}
- (void)hideProgressHUD:(BOOL)animation
{
[self.toast dismissToast];
}

- (void)showToastViewWith:(NSString *)message duration:(NSTimeInterval)timeInterval
{
[self showAlert:message];
}
```

11.2.2 热修复代码示例

Android 代码示例

基于 mPaaS 框架

参考 代码示例 获取示例代码。

基于原生框架

Demo 地址
参考 代码示例 获取示例代码。

下载应用体验热修复功能
在 Android 设备中，使用支付宝、钉钉等扫描二维码安装应用：

体验流程如下：

第一次启动应用，点击应用中的按钮，会弹出类似 **存在 Bug** 的提示信息，说明该版本应用存在 Bug。

由于该应用接入了热修复组件，在应用启动检测到 Bug 时，会下载热修复包。

关闭该进程，再次启动应用，点击应用中的按钮，会弹出类似 **Bug 已修复** 的提示信息。

第一次启动时下载的热修复包会在应用重启时生效，进而修复 Bug。

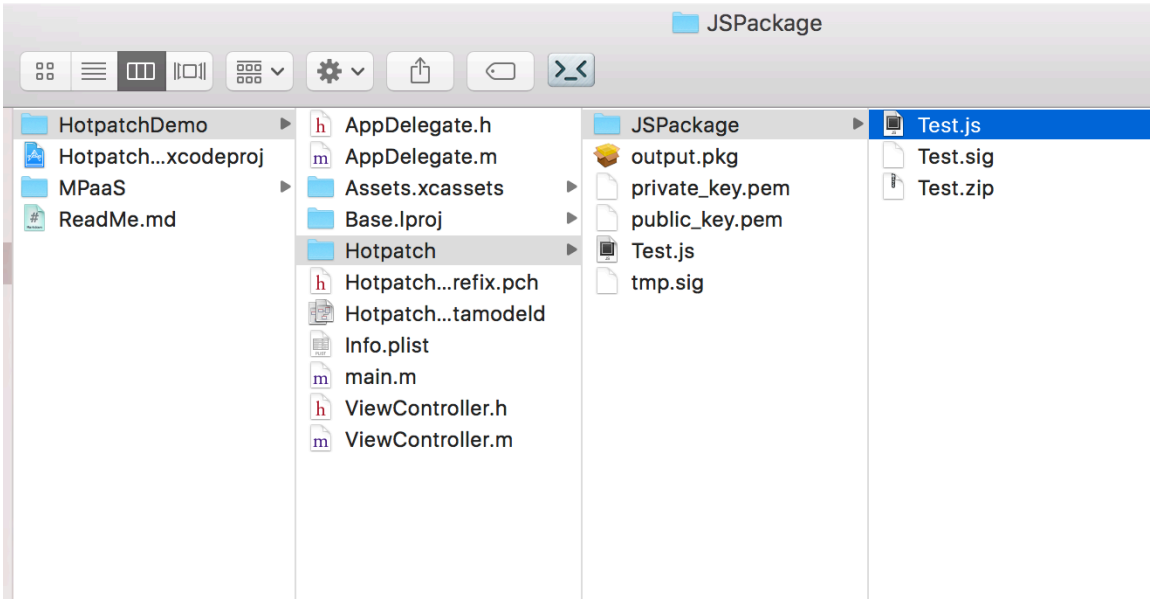
iOS 代码示例

有关 Hopatch 热修复的详细介绍，请参考 实时发布 > 热修复管理 > iOS 接入简介 。

快速开始

1. 直接运行工程，点击 **Create Crash** 按钮，程序闪退。

   将工程目录下的修复脚本 Test.js 上传到发布平台，并创建发布任务。创建发布任务的详细介绍请参考 实时发布 > 热修复 > 服务端 。



   再次点击 **Create Crash** 按钮，程序正常运行，闪退被修复。


Carrier
3:13 PM
Hotpatch Demo
Create Crash
Crash被Hotpatch热修复啦
ok

11.2.3 开关配置代码示例

根据不同客户端，选择下载不同的示例代码。

- iOS：开关配置代码示例（Cocoapods 和 mPaaS 插件接入）
- Android：开关配置代码示例（ mPaaS Inside 和 AAR 接入）

更多内容，参见 mPaaS 代码示例 。


蚂蚁集团
ANT GROUP