# 蚂蚁科技

## 实时发布
## 使用指南

文档版本：20230728

蚂蚁集团
ANT GROUP

# 法律声明

## 蚂蚁集团版权所有©2022，并保留一切权利。



## 商标声明



## 免责声明

# 通用约定

| 格式 | 说明 | 样例 |
| --- | --- | --- |
| 危险 | 该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。 | 危险<br>重置操作将丢失用户配置数据。 |
| 警告 | 该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。 | 警告<br>重启操作将导致业务中断，恢复业务时间约十分钟。 |
| 注意 | 用于警示信息、补充说明等，是用户必须了解的内容。 | 注意<br>权重设置为0，该服务器不会再接受新请求。 |
| 说明 | 用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。 | 说明<br>您也可以通过按Ctrl+A选中全部文件。 |
| > | 多级菜单递进。 | 单击**设置**> **网络**> **设置网络类型**。 |
| **粗体** | 表示按键、菜单、页面名称等UI元素。 | 在**结果确认**页面，单击**确定**。 |
| `Courier字体` | 命令或代码。 | 执行 `cd /d C:/window` 命令，进入Windows系统文件夹。 |
| *斜体* | 表示参数、变量。 | `bae log list --instanceid`<br>*Instance_ID* |
| [] 或者 [a\|b] | 表示可选项，至多选择一个。 | `ipconfig [-all\|-t]` |
| {} 或者 {a\|b} | 表示必选项，至多选择一个。 | `switch {active\|stand}` |

# 目录

# 1.产品公告

为提供更高效的实时发布服务，从 2022 年 7 月 14 日 21 点起，实时发布的发布域名由 mcube-prod.cn-hangzhou.oss.aliyuncs.com 变更为 mcube-prod.mpaascloud.com，请使用新域名获取发布包。

域名变更后，需要在全局资源包中将二级目录名称修改为 mcube-prod.mpaascloud.com，否则将无法使用实时发布对接的加速能力。

# 2.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。

  更多关于使用热修复的信息，欢迎搜索群号 41708565 加入钉钉群进行咨询交流。

- **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 99.95%，提供在线分钟级触达能力。

- **系统高性能保障**

  触达率达 99.95%，日 UV 支持 2 亿 +。

# 3.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。

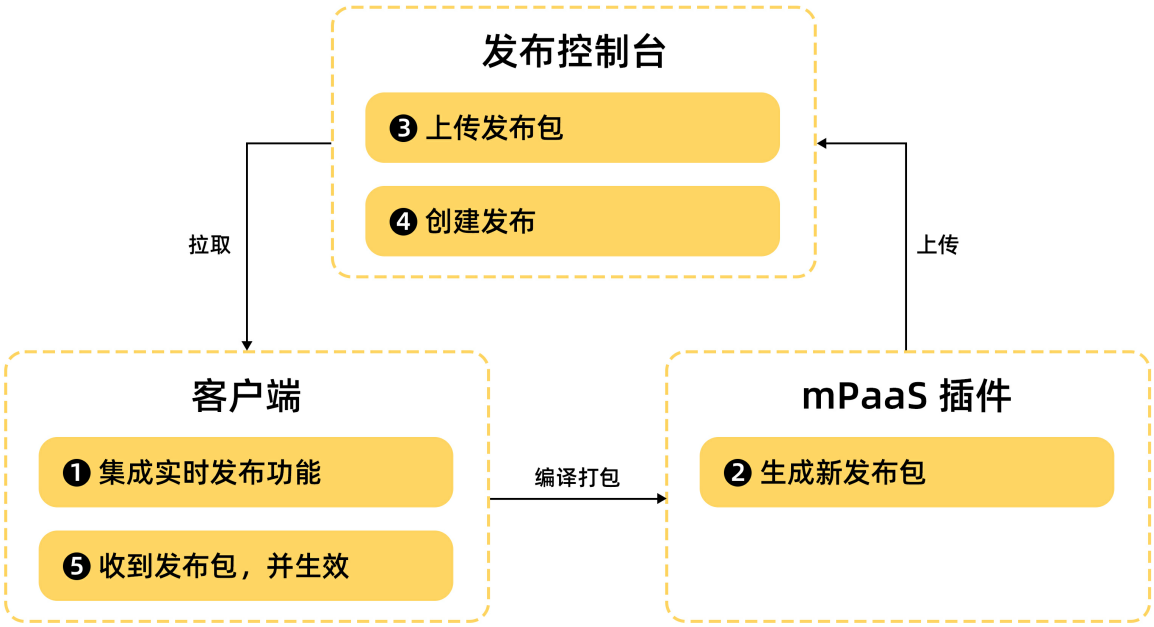
实时发布的流程如下：

1. 在客户端添加相应 SDK，集成实时发布升级、热修复或 H5 离线包的能力。
2. 在 mPaaS 插件中打包生成版本升级包、热修复包、离线包等，上传到发布控制台。
3. 在控制台创建发布任务进行灰度发布、正式发布等。
4. 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缺陷。
- 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。
- 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。

# 4.发布管理

## 4.1. 接入 Android

### 4.1.1. 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** 和 **组件化接入** 两种接入方式。

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 接入需要）
4. 升级检测

#### 前置条件

- 若采用原生 AAR 方式接入，需要先 mPaaS 添加到您的项目中。
- 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

在工程中通过 **组件管理（AAR）** 在工程中安装 **升级（UPGRADE）**组件。更多信息，参考 管理组件依赖。

#### 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 添加组件依赖。

#### 工程配置

#### 配置 AndroidManifest

1. 在 `AndroidManifest.xml` 文件中添加以下权限：

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

2. 在 `AndroidManifest.xml` 文件中添加以下配置。

   将通配符 `${applicationId}` 替换为真实的包名。例如，将 `${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>
```

> **说明**
>
> 更多关于配置 `AndroidManifest.xml` 的信息，请参见 应用清单概览。

3. 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>
```

## 添加资源

> **说明**
>
> 如果您使用的是原生 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
}
```

## 4.1.2. 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();
```

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
- `fileSize`：待下载的文件大小

## 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。

  `MPUpgrade` 内部封装了 `MPaaSCheckVersionService` 的调用；您也可以定制实现。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。

1. 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，然后在本方法内关闭进程。
     }
 }
```

2. 设置回调。

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

> **重要**
>
> - 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，请您设置不可取消的弹框遮盖层，以防止用户绕过强制升级。

### 4.1.3. 默认存储路径

从 10.2.3.3、10.1.68.53 基线起，mPaaS 升级组件默认下载 APK 的路径由原来的 external 存储修改为 internal 存储。

当 targetSdkVersion 在 24 以上，需要在 `file_paths.xml` 里加入如下代码：

```
<files-path
  name="files-path"
  path="." />
```

如果要保留原来的下载路径，请在 manifest 中，加入以下 metadata 内容：

```
<meta-data android:name="use_external" android:value="yes" />
```

## 4.2. 接入 iOS

### 4.2.1. 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

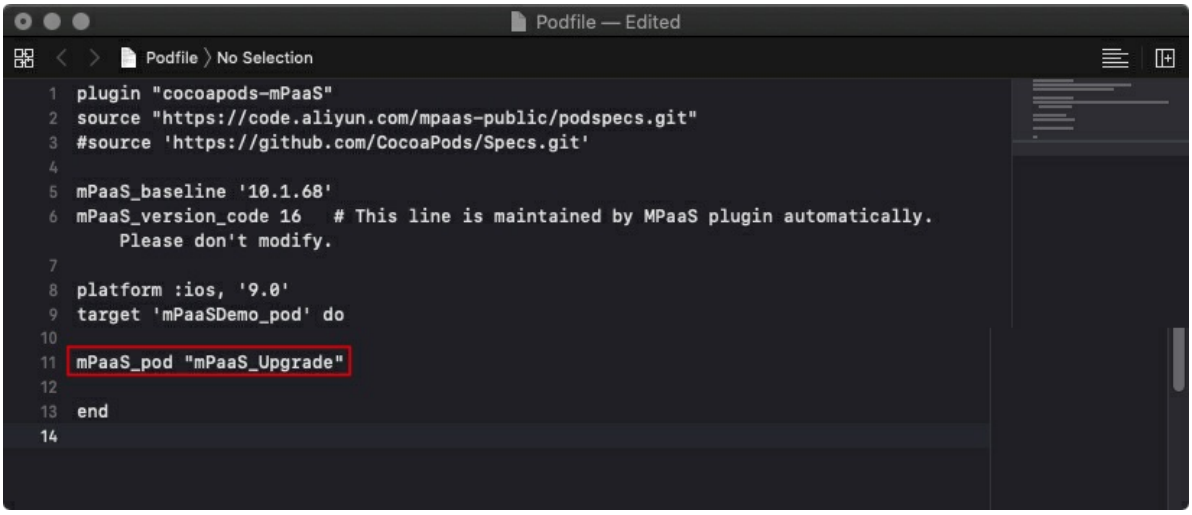
此方式适用于 基于 mPaaS 框架接入 或 基于已有工程且使用 mPaaS 插件接入 的接入方式。

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

## 4.2.2. 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。

> **重要**
>
> 如果移除了 UTDID 依赖会造成时间窗灰度发布无法生效。

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
```

```
- (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" me
ssage: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 *upgradeIn
fos) {
        [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。

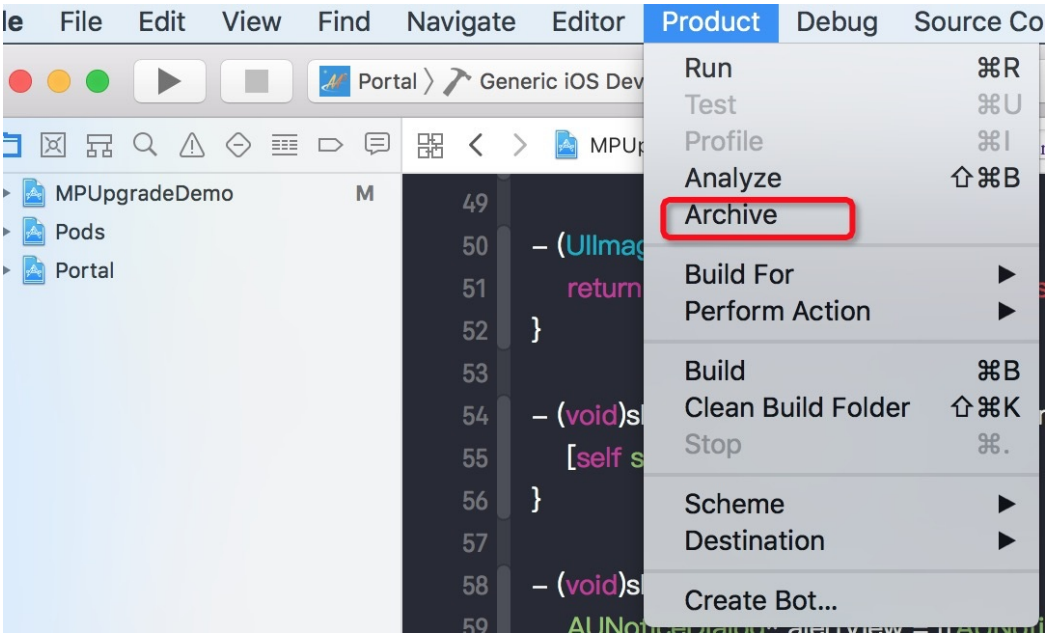
```
@implementation MPaaSInterface (Portal)

- (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，主要分为三种：

- **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

# 4.3. 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。同一版本的升级包可以有多个发布任务。
- 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 **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。
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。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，支持为同一版本的发布包同时创建多个发布任务。单个升级包最多支持同时发布 10 个任务。

发布任务下发规则：

- 当客户端请求匹配到多个发布任务时，优先下发高版本任务。
- 当同一个发布包版本命中多个发布任务时，按照任务类型，发布任务的下发优先级从高到低为：正式 > 白名单（灰度）> 时间窗（灰度）。
- 若发布包版本相同，任务类型也相同，则以最新发布的任务为准。例如，在控制台上发布了一个 5.0 版本的白名单任务 A，针对 4.0 版本进行单次升级；接着又发布了一个白名单任务 B，针对 4.0 版本做强制升级。这两个任务同时存在，当 4.0 版本的客户端请求升级时，首先下发任务 B，在任务 B 终止或者暂停后，下发任务 A。
- 当一个版本同时发布灰度任务和正式任务时，列表发布状态显示为“正式发布”，当暂停或结束正式任务后，发布状态显示为“灰度发布”。如果所有任务都结束了，则显示“已结束发布”。

操作步骤如下：

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**。
3. 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。
5. 如需创建更多发布任务，重复上述步骤即可。

### 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。

# 4.4. iOS 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

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。同一版本的升级包可以有多个发布任务。
- 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

> **说明**
>
> 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。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。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

> **说明**
>
> - 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。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，支持为同一版本的发布包同时创建多个发布任务。单个升级包最多支持同时发布 10 个任务。

发布任务下发规则：

- 当客户端请求匹配到多个发布任务时，优先下发高版本任务。
- 当同一个发布包版本命中多个发布任务时，按照任务类型，发布任务的下发优先级从高到低为：正式 > 白名单（灰度）> 时间窗（灰度）。
- 若发布包版本相同，任务类型也相同，则以最新发布的任务为准。例如，在控制台上发布了一个 5.0 版本的白名单任务 A，针对 4.0 版本进行单次升级；接着又发布了一个白名单任务 B，针对 4.0 版本做强制升级。这两个任务同时存在，当 4.0 版本的客户端请求升级时，首先下发任务 B，在任务 B 终止或者暂停后，下发任务 A。
- 当一个版本同时发布灰度任务和正式任务时，列表发布状态显示为“正式发布”，当暂停或结束正式任务后，发布状态显示为“灰度发布”。如果所有任务都结束了，则显示“已结束发布”。

操作步骤如下：

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**。
3. 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
       >
       > TestFlight 与 **企业分发** 类型的发布包仅支持 **灰度** 发布。TestFlight 发布页面不展示 **发布类型** 选项，**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。

# 5.热修复管理

## 5.1. 热修复简介

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）。

### 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：**在来不及发布版本的情况下，需要立刻解决线上客户端问题。**

根据最佳实践，热修复只用于修复严重的、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故障
- 客户端某些功能不能使用
- 监管审查导致的紧急修改

### 使用说明

- 目前 iOS 热修复功能仅支持专有云环境，公有云环境中无法使用 iOS 热修复功能。
- 目前 Android 支持两种热修复方式，分别为 DexPatch 和 InstantRun。
- 使用热修复涉及到调用 MDS 的更新发布接口，会产生相应的接口调用费用。有关接口调用的计费说明，参见 产品定价 中的实时发布计费项说明 。

### DexPatch 和 InstantRun 对比

| 对比项 | DexPatch | InstantRun |
| --- | --- | --- |
| 包大小 | 无需插桩，没有变化 | 需要插桩，体积变大 |
| 是否立即生效 | 否，需要重启 | 特定条件下立即生效，<br>重启必定生效 |
| 是否支持 so 文件修复 | 不支持 | 支持 |
| 是否支持资源文件修复 | 不支持 | 支持 |
| 额外依赖 | 无 | 依赖 Gradle 插件插桩 |

> 说明
>
> 两种热修复不能在同一个版本内混合使用，可以跨版本更换方案。

# 5.2. 接入 Android

## 5.2.1. DexPatch 接入方式

### 5.2.1.1. 接入说明

您可使用热修复功能在不发布新版本的情况下热修复线上故障，注意仅在紧急情况下使用此功能。

#### 使用限制

Android 热修复功能暂不支持以下机型或场景：

- Dalvik 的 X86 机型
- OPPO Android 11 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

> 说明
>
> - 在 Android 11 系统下，使用热修复前需要先获取 `READ_PHONE_STATE` 权限。
> - 如果 Android 系统升级，可能导致之前下发的补丁失效。

#### 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

### 5.2.1.2. 快速开始

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 接入和组件化接入两种接入方式。

1. 添加 SDK
2. 初始化热修复（仅原生 AAR 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 管理组件依赖，通过 **组件管理（AAR）**在工程中安装 **热修复（HOTFIX）**组件。

## 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。

## 初始化热修复

## 原生 AAR 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。

1. 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.Acti
vityLifecycleCallbacks {
  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) {
  }
 }
```

2. 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。

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


- **进行修复单击 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。**

- 修复后单击 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。


## 问题排查

- 不要使用非正规方式引入 apache-httpclient、apache-commons，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。

> **说明**
>
> 若出现 RPC 相关异常，可根据错误码进行排查，详细信息参考 RPC 调用。

- **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。

# 5.2.2. InstantRun 接入方式

# 5.2.2.1. 快速开始

## 接入说明

instantRun 方式仅支持原生 AAR 接入方式。

1. 添加 SDK
2. 初始化热修复
3. 生成热修复补丁
4. 发布热修复补丁

**InstantRun 新特性**

- 满足一定条件下，支持不重启修复；
- 支持 so 修复 ；
- 支持资源修复；
- 生成补丁的时候不需要类白名单。

## 技术原理

**Java 修复**

- 通过对 JavaMethod 进行预插桩，实现动态的逻辑替换。
- 构建 Patch 需要修改源码，因此无法修改代码的三方库不能实现修复。

**so 修复**

- 原始 so 未加载可立即生效。

**资源修复**

- 通过固定资源 id 而进行新增和修改。

## 前置条件

- 采用原生 AAR 方式接入，需要先 将mPaaS 添加到您的项目中。
- 不要混用 dexPatch 热修复方式和 instantRun 热修复方式。
- 新旧 APK 包中的 so 文件数量和名字需要保持一致，否则无法打出热修复补丁且不支持修复。
- 必须使用 10.2.3-20 及以上基线版本，如你是 10.1.68 基线版本按照文档进行 mPaaS 10.2.3 升级指南。

## 添加 SDK

**原生 AAR 方式**

参考 管理组件依赖，通过 **组件管理（AAR）**在工程中安装 **热修复（Hotfix）**组件。

## 初始化热修复

## 原生 AAR 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。

1. 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.Acti
vityLifecycleCallbacks {
  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) {
  }
 }
```

2. 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` 。

3. 引入 Apache HTTP 客户端。

   在使用热修复功能的时候，需要调用到 Apache HTTP 客户端相关的功能。只需在 `AndroidManifest.xml` 加入如下代码。更多信息，请参见使用 Apache HTTP 客户端。

```
<uses-library android:name="org.apache.http.legacy" android:required="false"/>
```

## instantRun 插桩配置和依赖

## instantRun Maven

```
maven {
   url "https://mvn.cloud.alipay.com/nexus/content/repositories/open/"
}
```

## instantRun 插桩依赖

工程 app 主 module 下 build.gradle 文件中添加如下依赖：

```
apply plugin: 'com.android.application'
apply plugin: 'com.alipay.instantrun'
```

工程根目录下 build.gradle 文件中添加如下插件依赖：

```
dependencies {
     classpath "com.mpaas.android.patch:patch-gradle-plugin:1.0.7"
 }
```

## instantRun 插桩配置

## 创建 instantrun 文件夹

在工程根目录下创建 instantrun 文件夹，放入生成 patch.jar 需要使用的 `mapping.txt` 文件。

instantrun 文件夹中文件的说明及来源按以下方式操作放入：

> **重要**
>
> 以下文件必须每次发布版本前进行保留，当需要修复时需将上个版本保留好的替换到 instantrun 文件夹中。

在有 Bug 的项目工程中执行命令行 `./gradlew clean assembleRelease` 生成以下文件：

instantrun/InstantRunMapping.txt.gz（构建后生成的 InstantRunMapping_release.txt.gz 或 InstantRunMapping_debug.txt.gz，产物在 ./build/outputs/instantrun/ 如有，则改名后放入）

instantrun/mapping.txt（原工程构建打 Release 包时的 mapping.txt，产物在 ./build/outputs/mapping/[debug|release]/mapping.txt）

instantrun/methodsMap.instantrun（原 Bundle 级别接入插桩生成的 methodsMap.instantrun，产物在./build/outputs/instantrun/methodsMap.instantrun 如有则放入）

## 添加 instantrun.xml 配置文件

在工程根目录下添加 `instantrun.xml` 配置文件，注意配置中的相关内容，请仔细斟酌每个选项：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <switch>
        <!--true代表打开InstantRun，请注意即使这个值为true，InstantRun也默认只在Release模式下开启-->
        <!--false代表关闭InstantRun，无论是Debug还是Release模式都不会运行InstantRun-->
        <turnOnInstantRun>true</turnOnInstantRun>

        <!--是否开启手动模式，手动模式会去寻找配置项patchPackageName包名下的所有类，自动的处理混淆，然后把patchPackageName包名下的所有类制作成补丁-->
        <!--这个开关只是把配置项patchPackageName包名下的所有类制作成补丁，适用于特殊情况，一般不会遇到-->
        <manual>false</manual>

        <!--是否强制插入代码，InstantRun默认在debug模式下是关闭的，开启这个选项为true会在debug下插入代码-->
        <!--但是当配置项turnOnInstantRun是false时，这个配置项不会生效-->
        <forceInsert>false</forceInsert>

        <!--配置Patch函数匹配规则，默认为函数签名，可修改为函数序号，存储在methodsMap.instantrun中-->
        <!--其中signature模式切面占用空间较大，而id模式切面占用空间会更小一些，更适合在乎安装包大小的App使用-->
        <!--<methodMatchMode>signature</methodMatchMode>-->
        <methodMatchMode>id</methodMatchMode>

        <!--是否捕获补丁中所有异常，建议上线的时候这个开关的值为true，测试的时候为false-->
        <catchReflectException>false</catchReflectException>

        <!--是否在补丁加上log，建议上线的时候这个开关的值为false，测试的时候为true-->
        <patchLog>true</patchLog>

        <!--项目是否支持progaurd-->
        <proguard>true</proguard>

    </switch>

    <!--需要HotPatch的包名或者类名，这些包名下的所有类都会被插入代码-->
    <!--这个配置项是各个Bundle需要自行配置，就是你们Bundle里面你们自己代码的包名，
    这些包名下的类会被InstantRun插入代码，没有被InstantRun插入代码的类InstantRun是无法修复的-->
```

```
    这些包名下的类会被InstantRun插入代码，没有被InstantRun插入代码的类InstantRun是无法修复的-->
    <acceptPackageName name="acceptPackage">
        <name>com.</name>
        <name>android.</name>
        <name>org.</name>
    </acceptPackageName>

    <!--不需要InstantRun插入代码的包名，InstantRun库不需要插入代码，如下的配置项请保留，还可以根据各个APP的情况执行添加-->
    <exceptPackageName name="exceptPackage">
        <name>com.alipay.euler</name>
        <name>com.alipay.dexpatch</name>
        <name>com.alipay.instantrun</name>
        <name>ohos.</name>
        <name>com.alipay.mobile.quinox.LauncherApplication</name>
    </exceptPackageName>

    <!--不需要InstantRun插桩的函数访问类型和对应包名列表，按需插桩，可降低接入包大小-->
    <methodExceptConfig name="methodExceptConfig">
        <privateMethodPackage>
            <name>com.instantrun.demo</name>
        </privateMethodPackage>
        <packageMethodPackage>
            <name>com.instantrun.demo</name>
        </packageMethodPackage>
        <protectedMethodPackage>
            <name>com.instantrun.demo</name>
        </protectedMethodPackage>
        <publicMethodPackage>
            <name>com.instantrun.demo</name>
        </publicMethodPackage>
        <syntheticMethodPackage>
            <name>com.instantrun.demo</name>
        </syntheticMethodPackage>
    </methodExceptConfig>

    <!--补丁的包名，请保证唯一性，填写apk包名即可-->
    <patchPackageName name="patchPackage">
        <name>com.instantrun.demo</name>
    </patchPackageName>

</resources>
```

## 修改 Bug 代码

## Java 函数修复方式

- 首先接入上述的构建依赖与配置；
- 需要在改动的方法上面添加以下注解：

```
@com.alipay.instantrun.patch.annotaion.Modify
```

```
//修改方法
@com.alipay.instantrun.patch.annotaion.Modify
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
}
//或者是被修改的方法里面调用 com.alipay.instantrun.patch.InstantRunModify.modify()方法
protected void onCreate(Bundle savedInstanceState) {
    com.alipay.instantrun.patch.InstantRunModify.modify()
        super.onCreate(savedInstanceState);
}
```

- 对于 Lambda 表达式，需要在修改的方法里面调用：`com.alipay.instantrun.patch.InstantRunModify.modify()` 。

```
//修改方法
@com.alipay.instantrun.patch.annotaion.Modify
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
}
//或者是被修改的方法里面调用com.alipay.instantrun.patch.InstantRunModify.modify()方法
protected void onCreate(Bundle savedInstanceState) {
    com.alipay.instantrun.patch.InstantRunModify.modify()
        super.onCreate(savedInstanceState);
}
```

- 新增的类、方法使用 `@com.alipay.instantrun.patch.annotaion.Add` 注解。

```
//增加方法
@com.alipay.instantrun.patch.annotaion.Add
public String getString() {
    return "InstantRun";
}
//增加类
@com.alipay.instantrun.patch.annotaion.Add
public class NewAddCLass {
    public static String get() {
        return "instantRun";
        }
}
```

## so 修复方式

修改相关 C/C++ 代码或 so 即可，然后打包新的 so 替换工程中有问题的 so 库。

## 资源修复方式

支持任何资源文件的修复和添加，但不支持 so 库资源的添加。

## 资源 ID 固定配置

1. 下载 生成资源文件的工具 Jar 包 获取需要修复的 apk 包资源 id，执行以下命令行：

```
java -jar ~/path/stableResourcesId.jar ~/path/old.apk ~/path/values
```

生成的产物在目标目录下。

2. 将上面命令行生成的 `public.xml` 、 `ids.xml` 两个文件产物放入到新工程（修复好的工程）的 `~/res/values` 目录中。
3. 将 `public.txt` 文件放入到新工程（修复好的工程）的根目录中。

   然后在 app moudle 的 `build.gradle` 文件中的 android 节点下进行如下配置：

```
aaptOptions {
    File publicTxtFile = project.rootProject.file( 'public.txt')
    if (publicTxtFile.exists()) {
        additionalParameters "--stable-ids", "${project.rootProject.file
        ('public.txt').absolutePath}"
    }
}
```

## 进行补丁打包配置

在 Android Studio 的 mPaaS 插件中，进入 **基础工具 > 热修复 > 生成补丁（instant run）**页面。

## 说明

如果是修改或者新增 `res/layout` 下的 `xml` 布局文件，则需在 Java 代码层进行 Java patch 加入修改的注解以及获取对应的资源 id，再进行加载。

## 使用 instantRun

## 生成 patch.jar 产物

patch.jar 是生成热修复补丁包的关键，包含了需要修复的内容，在最新修复的工程输出即可，在终端按如下命令行执行生成：

```
./gradlew clean mpGeneratePatch
```

生成的产物在工程的 `./build/outputs/instantrun/patch.jar` 目录下。

## 生成热修复 instantRun 补丁

在 Android Studio 的 mPaaS 插件工程中进行补丁生成，如下所示：

1. 单击 **基础工具 > 热修复 > 生成补丁（instant run）**。
2. 进入生成补丁页面，填写相关修复信息项。

   **输入项含义说明**

   - **New bundle**：选择最新修复好的 apk 包路径。
   - **Old bundle**：选择有 bug 的 apk 包路径（线上版本）。
   - **PatchJarPath dir**：工程中接入 instantRun 执行命令行 `./gradlew clean mpGeneratePatch` 打出的 `patch.jar` 路径。
   - **Patch out dir**：存放补丁包产物的路径。

   > **重要**
   >
   > New bundle 和 Old bundle 中的 apk 包不能存放到 C 盘的 User 目录下和中文目录下。

3. 单击 Next 进入填写签名信息页面。

   确保填写的签名信息准确，否则会导致生成失败。

4. 填写完成后单击 Create 即可生成补丁。

### 使用热修复

将补丁包发布到控制台，具体参考文档 使用热修复。

### 查看日志

- 热更新请求日志

  过滤 Tag 中包含 DynamicRelease 的日志；

- InstantRun 执行日志

  过滤 Tag 中包含 IR. 的日志；

- Patch 执行日志

  在打 Patch 前，在构建配置 instantrun.xml 中打开 patchLog 选项。 过滤 Tag 中包含 IR.PatchCode 的日志。包含 invoke method is 的日志行，表示执行了对应函数的 Patch。

## 5.2.2.2. 注意事项

使用 InstantRun 热修复需注意以下事项：

- 内部类的构造方法是 private（private 会生成一个匿名的构造函数）时，需要在制作补丁过程中手动修改构造方法的访问域为 public；
- 不支持 Patch 中包含 Lambda 等语法，需要在 Patch 时更换写法；
- 对于方法的返回值是 this 的情况暂不能完美支持，比如 Builder 模式，但在制作补丁代码时，可以通过增加一个类来包装，如下的 B 类：

```
method a(){
    return this;
}
method a(){
 return new B().setThis(this).getThis();
}
```

- 不支持修改、增加字段，可以通过增加新类，把字段放到新类中的方式来实现增加字段的能力；
- 新增的类支持静态内部类和非内部类；
- 对于只有字段访问的函数无法直接修复，可通过调用处理间接修复；
- 不支持构造方法的修复；
- 不支持所有被 AspectJ 切面的函数；
- 被 Patch 的类中，如果静态块里有调用等待其他功能初始化的代码，比如直接或间接调用 `LauncherApplicationAgent.getInstance()` 等，可能会导致启动卡死；
- 如果被 Patch 的代码中，包含类似 `System.arraycopy` 的调用，由于 InstantRun 本身对 Patch 函数内容的反射调用处理，会导致 `System.arraycopy` 对 `modelPaths` 的赋值失效，所以需要将 `java.lang.System` 加入不反射的列表，保证流程正确；如果 `System.arraycopy` 的访问权限不是 public，可能就直接不支持被 Patch 了；

```
@com.alipay.instantrun.patch.annotaion.Modify
private void handleInit() {
 ......
    String[] modelPaths = new String[1 + mExtraModels.length];
    modelPaths[0] = mModelPath;
    System.arraycopy(mExtraModels, 0, modelPaths, 1, mExtraModels.length);
    int[] modelTypes = new int[1 + mExtraModelTypes.length];
    System.arraycopy(mExtraModelTypes, 0, modelTypes, 1, mExtraModelTypes.length);
    ......
}
```

- 如果是 Bundle 级别主动接入，注意 gradle 3.6 及以上版本默认启用 R8，会将插入的 Field 变量优化掉，需要在混淆文件 proguard-rules.pro 中加入以下代码：

```
-keepclassmembers class **{
public static com.alipay.instantrun.ChangeQuickRedirect *;
}
```

### 5.2.2.3. 接入 demo 参考

若需参考此接入方式的 Demo，请单击：

- 热修复前 Demo：instantRunHotPatch_aar_before_demo。
- 热修复后 Demo：instantRunHotPatch_aar_demo。

> 说明
>
> Demo 需要配置自己控制台下载的 config 文件、应用包名及签名文件。

# 5.3. 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 接入和组件化接入（Portal&Bundle）两种接入方式。关于接入方式的更多说明，请参见 接入方式简介。

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。

> 说明
>
> 本使用教程以组件化接入（Portal&Bundle）方式为例介绍 Android 热修复的使用。

## 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 mPaaS 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 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 HOTFIX 和 RPC。

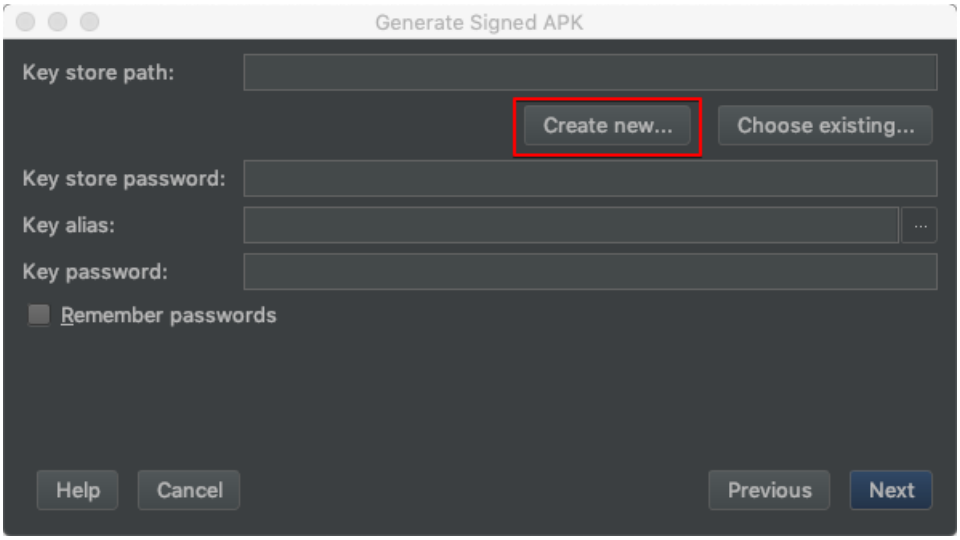
## 签名

签名是为了下一步配置加密信息做准备。

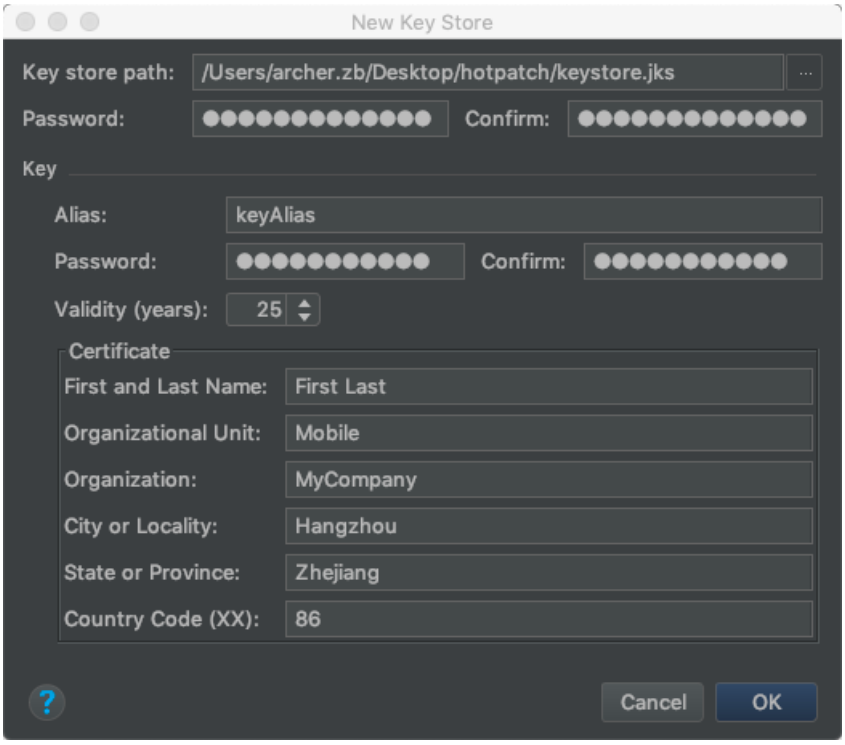
在 Android Studio 中打开 Portal 工程，然后参考 Android 官网文档 给应用签名，并生成带签名的 APK。

签名步骤如下：

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

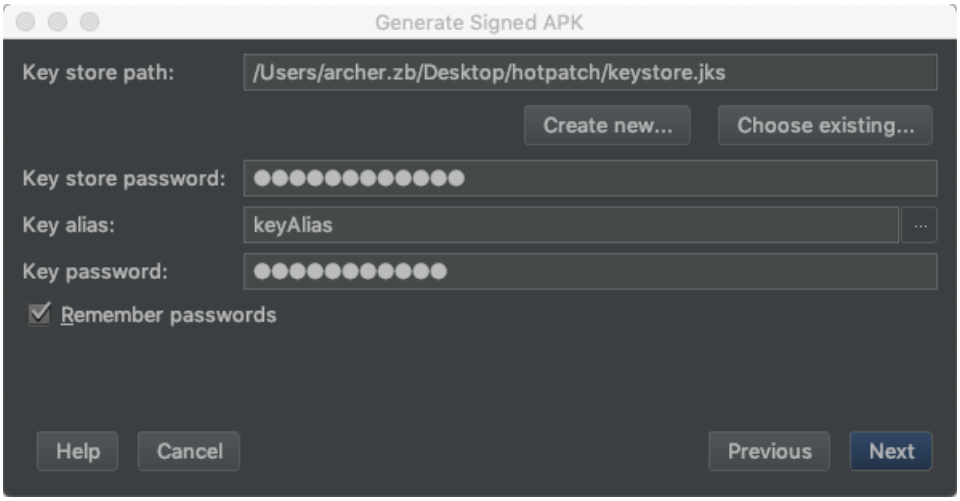


ii. 完善相关信息，然后点击 **OK**。您需要记住此处填写的信息，以便在后续操作中使用。



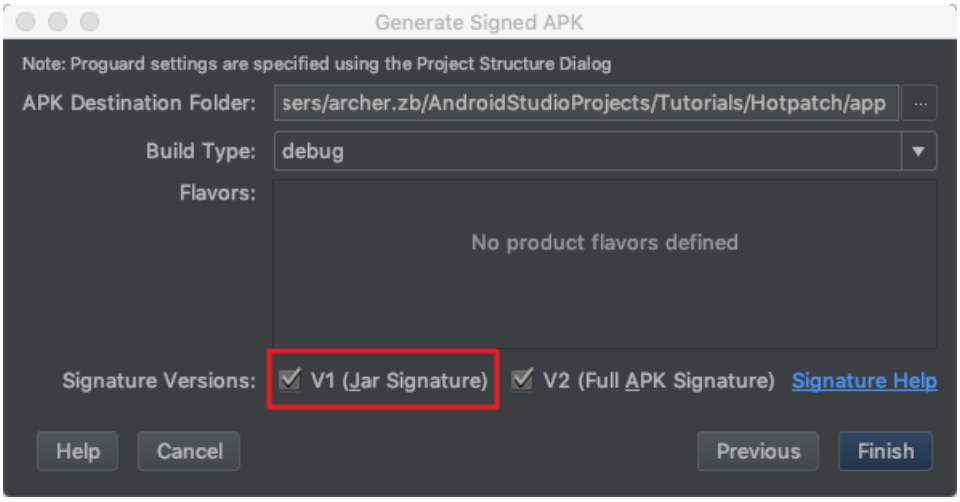

2. 生成带签名的 APK。

i. 选择密钥库，填写相关信息，然后点击 **Next**：



ii. 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：

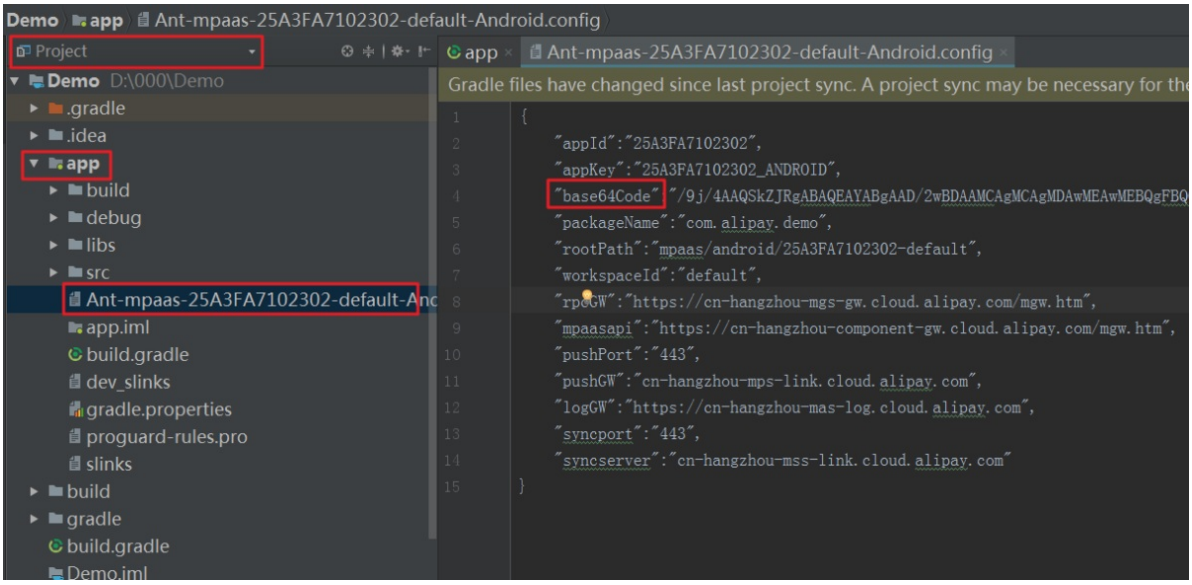
```
signingConfigs {
    release {
        keyAlias 'keyAlias'
        keyPassword 'keyPassword'
        storeFile file('/Users/archer.zb/Desktop/hotpatch/keystore.jks')
        storePassword 'storePassword'
    }
    debug {
        keyAlias 'keyAlias'
        keyPassword 'keyPassword'
        storeFile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

## 配置加密信息

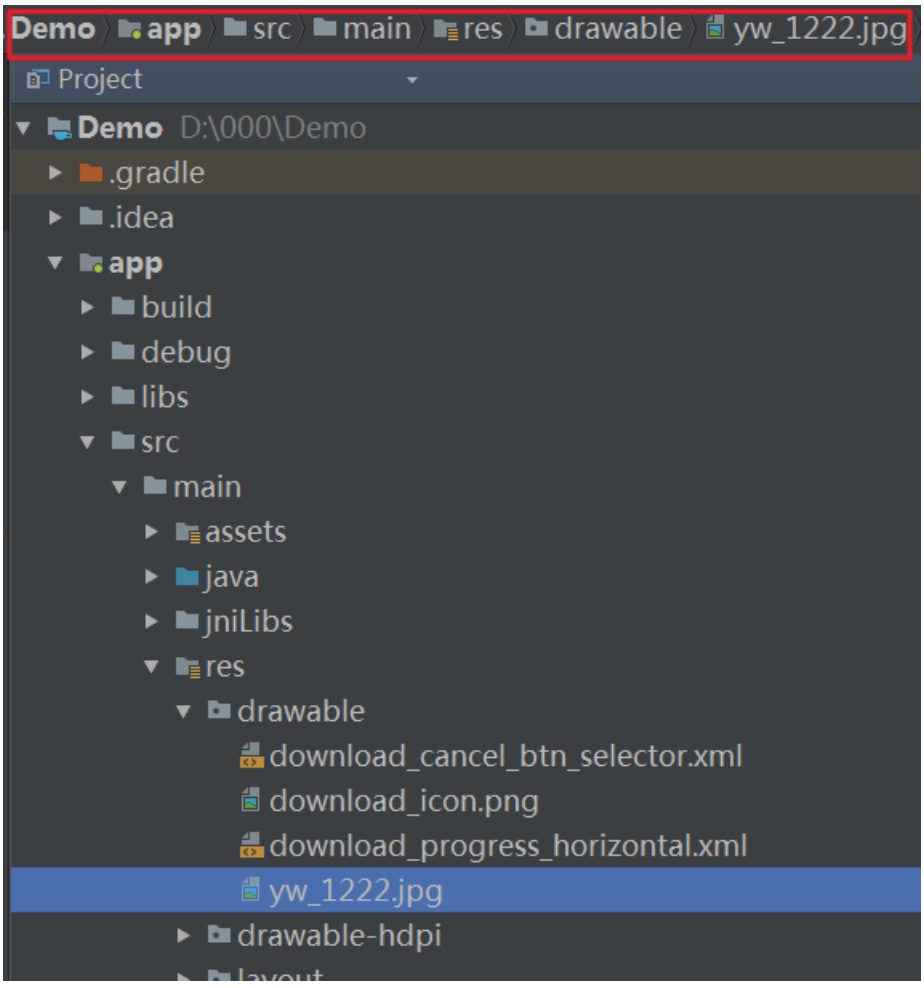
为了保证客户端获取热修复包过程的安全，需要对网络内容进行加密。配置加密信息的步骤如下：

1. 到控制台重新下载配置文件。 登录 mPaaS 控制台 后，进入 总览 > 下载 Android 代码配置 > 代码配置 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。

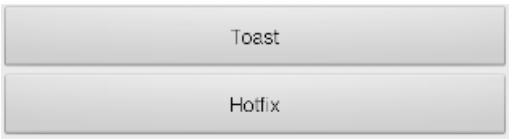
## 编写代码

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();
  }
});
```

```
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

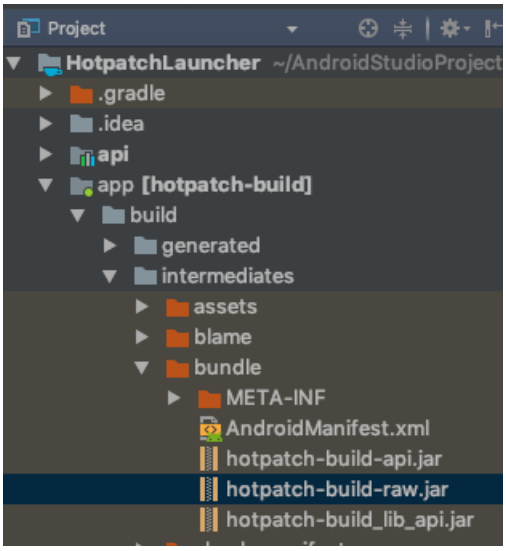
5. 应用重启后，触发热修复，Bug 被修复

## 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：



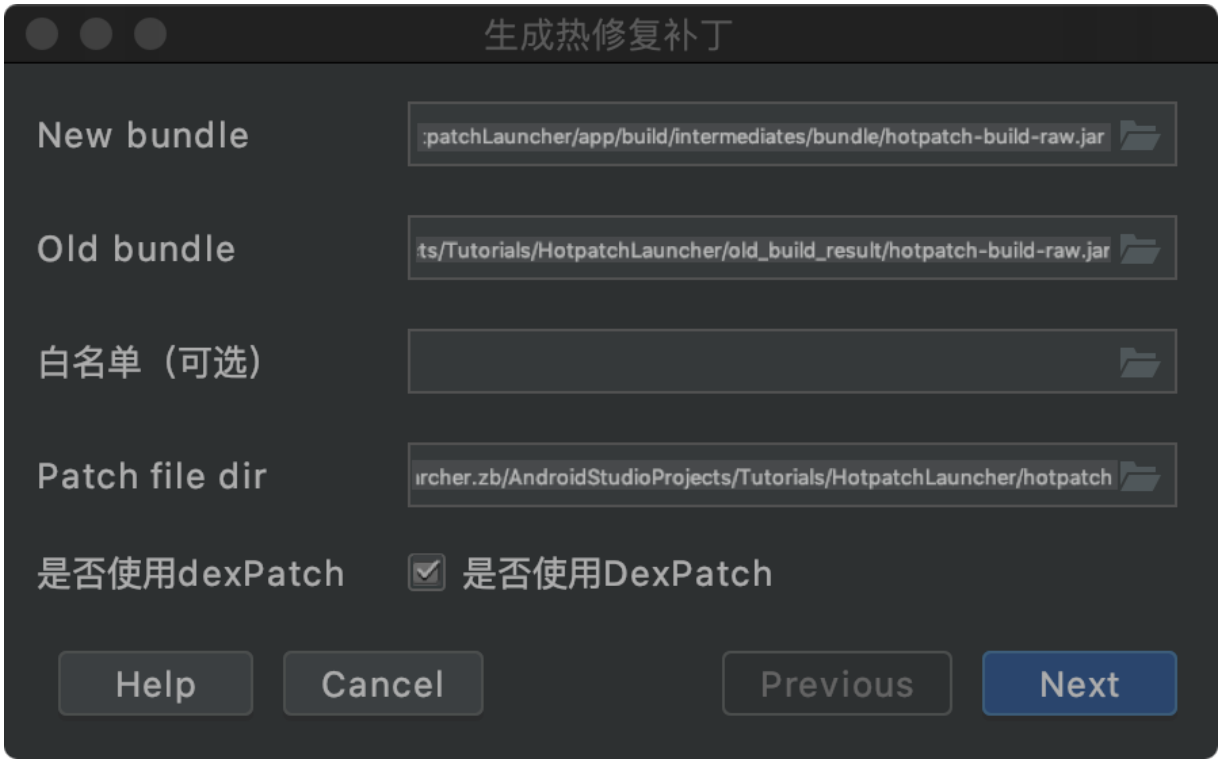

## 修改 Bug 代码

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。使用原生 AAR 工程时强烈推荐使用该功能。
- **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。

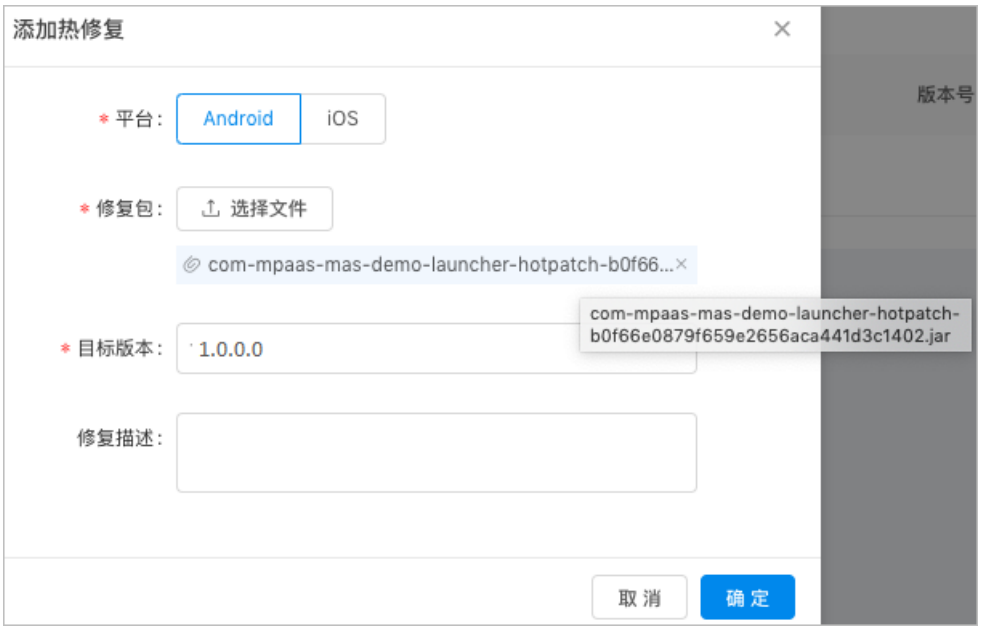
更多信息，参见 生成热修复包。

## 在控制台添加并发布热修复包

### 添加热修复包

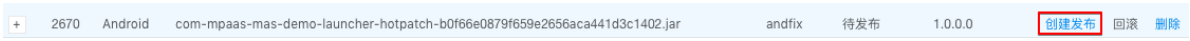
1. 进入mPaas 控制台 > **实时发布** > **热修复管理** 页面。

2. 点击 **添加热修复**，然后完善相关信息，并点击 **确定**。




## 创建发布

1. 如下图，点击 **创建发布**。

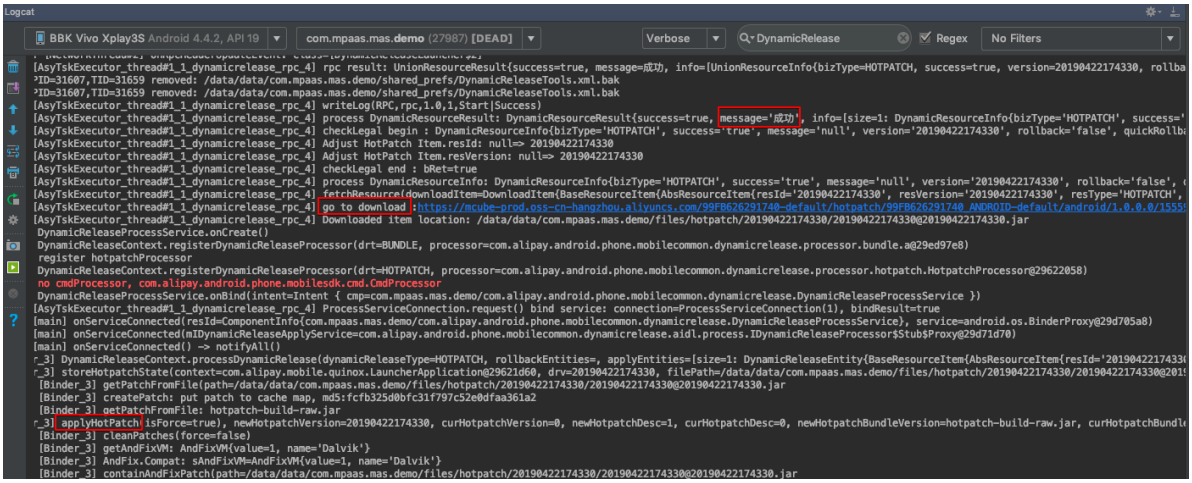



2. 选择发布类型等，然后点击 **确定** 即可完成发布。更多信息，参见 热修复管理。

## App 触发热修复

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


2. 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。
- 接入方式简介：了解原生 AAR 接入和组件化接入（Portal&Bundle）两种接入方式。
- 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。
- 生成热修复包：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。

# 5.4. 常见问题

下面罗列热修复接入和使用过程中的一些常见问题。

## 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

请参见 取消支持 Apache HTTP 客户端 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。

### RPC 调用相关

如果通过 RPC 请求进行资源调用的过程中出现异常，请参考 无线保镖结果码说明 进行排查。

# 6.H5 离线包管理

## 6.1. 配置 H5 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. 单击左侧导航栏的 **实时发布 > 离线包管理**。
2. 在打开的离线包列表页，单击 **配置管理**。
3. 在 **域名管理** 栏，填写虚拟域名，例如 `h5app.com` 。虚拟域名用于客户端加载本地离线包文件时，作为后缀绑定文件名称，以规范本地文件地址名称。

   > **重要**
   >
   > 虚拟域名不能是以 http 或 https 开头的两级或三级域名且一定要是自己注册的域名。

4. 勾选 **已确认以上信息准确，提交后不再修改**，单击 **保存**。
5. 在 **密钥管理** 栏，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
   ```

   > **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。

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

### 后续步骤

生成离线包

## 6.2. 生成 H5 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两步：

1. 构建前端 .zip 包
2. 在线生成 .amr 包

### 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：在公有云中，二级目录需固定为 `mcube-prod.mpaascloud.com`，否则将无法使用实时发布对接的加速能力。
  - 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`。

> 重要
>
> 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。
> - 若引用公共资源，则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.mpaascloud.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。
> - 建议 info.plist 文件中的 `Product Version` 与 `Bundle versions string, short` 的值保持一致，否则可能导致离线包不生效。

# 6.3. 创建 H5 离线包

在创建 H5 离线包资源时，您需要填写基本信息和配置信息。

### 前置任务

您已经在配置管理页面，完成 H5 离线包相关配置。详细信息，参见 配置离线包。

### 关于此任务

您可以选择单个创建 H5 离线包，也可以选择以批量导入 H5 离线包文件的方式一次创建多个离线包。

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

### 创建单个离线包

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. 在打开的离线包管理页面，点击 **新建 H5App**。（如果您已创建 H5App，可忽略此步。）
3. 在 **新建 H5App** 窗口，填写 **H5App ID** 和 **H5App 名称**，点击 **确定**。（如果您已创建 H5App，可忽略此步。）

> **重要**
>
> - H5App ID 为 8 位数字。

- 20000196、66666692、68687029、68687209 是 SDK 内置的离线包 ID，H5App ID 建议不要使用，否则会发生冲突。
- H5App ID 建议不要使用以 666666 或者 20000 开头的数字。

4. 在 H5App 列表中，选择 H5App，然后点击离线包列表的右上方的 **添加离线包** 按钮，创建离线包。
5. 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**。

     > **说明**
     >
     > 若使用全局资源包需要在全局资源包中将二级目录名称修改为 mcube-prod.mpaascloud.com，否则将无法使用实时发布对接的加速能力。

   - **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` 。
   - **客户端生效范围**：选择 App 对应的客户端类型，并设置版本范围。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。

     > **说明**
     >
     > - 至少选择一个客户端类型。若同时选择 Android 及 iOS，客户端最高版本策略需保持一致，即两个客户端均采用系统默认，或者均输入自定义值。
     > - 最高版本为系统默认时表示支持后续所有新版本，建议采用系统默认，以免在客户端升级后版本高于填写的最高版本而使得离线包不生效。
     > - 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。

6. 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。

     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 和 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**，点击 **提交**，完成离线包创建。

## 批量导入离线包

如果需要创建多个离线包，为避免多个离线包配置时信息配置出错，提升发布效率，可选择以批量导入的方式进行创建。

- 导入后，若离线包所属的 App 在系统中不存在，将默认创建一个 H5App。
- 导入后，若离线包所属的 App 在系统中已存在，配置完成后，离线包将添加至该 H5App。

进入 mPaaS 控制台，完成以下步骤：

1. 从左侧导航栏进入 **实时发布 > 离线包管理** 页面，点击 **批量导入 H5 离线包**。
2. 在 **批量导入 H5 离线包** 窗口中，根据提示上传 H5 离线包文件（.zip）。

> 说明
> - 批量导入的离线包文件大小不能超过 300 MB，且离线包个数不能超过 100。
> - 每个离线包资源文件需以离线包 ID 命名。离线包 ID 为 8 位数字。

3. 导入结果页以列表的形式显示成功加载的离线包，在导入结果页面，点击 **操作** 列的 **编辑** 按钮，编辑离线包的基本信息。具体配置项解释，请参考 创建单个离线包。

   导入结果页面，离线包版本号默认遵循以下规则，您可以进行编辑。
   - 若离线包所属的 App 在系统中不存在，导入离线包的版本号默认为 0.0.0.1。
   - 若离线包所属的 App 在系统中已存在，导入离线包的版本号默认为最高版本的基础上 +1。
4. 编辑完成所有离线包后，勾选 **以上信息提交后不再支持修改**，点击 **提交**。系统会对提交的离线包信息进行校验。若校验不通过，页面会出现错误提示；若校验通过，则 H5 离线包管理页面会展示相应的 H5 离线包信息，即表示离线包创建成功。

## 后续步骤

发布离线包

# 6.4. 发布 H5 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。您可以选择发布单个 H5 离线包，也可以选择批量发布多个离线包。

## 操作步骤

## 发布单个离线包

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。

## 批量发布离线包

如果您有多个离线包需要发布，可使用批量发布功能。操作步骤如下：

1. 从左侧导航栏进入 **实时发布** > **离线包管理** 页面，点击 **批量发布**。
2. 在弹出的窗口列表中，从左侧 H5App 列表勾选要发布的 App，将其添加至右侧的 **已选中** 列表，点击 **确定**。

   > **说明**
   >
   > 列表中仅显示最高版本是 **待发布** 和 **结束发布** 状态的 App。

3. 在 **创建批量发布** 页面，配置 **发布类型**、**发布模式** 等，具体参数项含义，请参考 发布单个离线包。
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

# 6.5. 管理 H5 离线包

发布 H5 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、结束发布、导出、删除 H5 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. 单击左侧导航栏的 **实时发布 > 离线包管理** 菜单。
2. 在 H5App 列表中，选择目标 H5App，然后在右侧的 H5 离线包列表中选择要查看的离线包版本，单击相应的展开图标（ + ）。
3. 在展开的任务列表中，单击 **查看**，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. 单击左侧导航栏的 **实时发布 > 离线包管理** 菜单。
2. 在 H5App 列表中，选择目标 H5App，然后在右侧的 H5 离线包列表中选择要暂停的离线包版本，单击相应的展开图标（ + ）。
3. 在展开的任务列表中，单击 **暂停**，并确认中止离线包发布。

   中止后，如果您要继续发布该离线包，单击 **继续**。

## 结束离线包发布任务

进入 mPaaS 控制台，完成以下步骤：

1. 单击左侧导航栏的 **实时发布 > 离线包管理** 菜单。
2. 在 H5App 列表中，选择目标 H5App，然后在右侧的 H5 离线包列表中选择要结束的离线包版本，单击相应的展开图标（ + ）。
3. 在展开的任务列表中，单击 **结束**，并确认结束离线包发布。

结束后，如果您要再次发布该离线包，需要重新创建发布。

> **重要**
>
> 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 V1.0 版本的离线包。

## 导出离线包

MDS 支持单个下载离线包资源文件（.amr）或配置文件（.json）。

1. 进入 mPaaS 控制台后，从左侧导航栏进入 **实时发布 > 离线包管理** 页面。
2. 在 H5App 列表中，选择目标 H5App，在右侧的离线包列表中，选择离线包版本，单击 **下载 AMR 文件** 或 **下载配置文件** 即可完成下载。

### 删除 H5App

进入 mPaaS 控制台，完成以下步骤：

1. 单击左侧导航栏的 **实时发布 > 离线包管理** 菜单。
2. 在左侧的 H5App 列表中，将鼠标悬浮至您要删除的 H5App，单击删除图标，并在弹窗中单击 **确认**。删除 H5App 后，该 H5App 包含的所有离线包及资源文件均会被删除。

> **重要**
>
> H5App 删除后无法恢复，请谨慎操作。

# 6.6. 开放接口

## 6.6.1. 概述及准备

实时发布为离线包提供 Java SDK，您可以通过调用开放接口来实现配置、创建、发布及管理离线包的操作。

### 准备工作

在使用 OpenAPI 前，您需要先获取 AccessKey、App ID、Workspace ID 与 Tenant ID，并配置 Maven 依赖及配置文件上传。

### 获取 AccessKey

AccessKey 包括 AccessKey ID 与 AccessKey Secret，点击此处 查看获取方式。

- **AccessKey ID**：用于标识用户。
- **AccessKey Secret**：用于验证用户的密钥，必须保密。

### 获取 App ID、Workspace ID 与 Tenant ID

1. 登录 mPaaS 控制台，进入应用。
2. 在 **总览** 页，依次点击 **代码配置**（可视情况选择 Android 或 iOS）> **下载配置文件** > **立即下载**，在右侧弹出的 **代码配置** 窗口中，您可以看到 App ID、Workspace ID 和 Tenant ID 的值。

### 配置 Maven 依赖

在使用 OpenAPI 之前，您需要完成以下 Maven 依赖配置。

```
<dependency>
  <groupId>com.aliyun</groupId>
  <artifactId>aliyun-java-sdk-mpaas</artifactId>
  <version>3.0.5</version>
</dependency>

<dependency>
  <groupId>com.aliyun</groupId>
  <artifactId>aliyun-java-sdk-core</artifactId>
  <optional>true</optional>
  <version>[4.3.2,5.0.0)</version>
</dependency>
```

## 环境变量配置

**配置环境变量 MPAAS_AK_ENV 和 MPAAS_SK_ENV。**

- **Linux 和 macOS 系统配置方法执行以下命令：**

```
export MPAAS_AK_ENV=<access_key_id>
export MPAAS_SK_ENV=<access_key_secret>
```

> **说明**
>
> `access_key_id` 替换为已准备好的 AccessKey ID，`access_key_secret` 替换为 AccessKey Secret。

- **Windows 系统配置方法**

  i. 新建环境变量，添加环境变量 **MPAAS_AK_ENV 和 MPAAS_SK_ENV**，并写入已准备好的 AccessKey ID 和 AccessKey Secret。

  ii. 重启 Windows 系统。

## 使用示例

```
import com.aliyuncs.DefaultAcsClient;
import com.aliyuncs.IAcsClient;
import com.aliyuncs.mpaas.model.v20201028.QueryMcubeVhostRequest;
import com.aliyuncs.mpaas.model.v20201028.QueryMcubeVhostResponse;
import com.aliyuncs.profile.DefaultProfile;

public class MpaasApiDemo {

    /**
     * mPaaS控制台上对应的APP ID
     */
    private static final String APP_ID = "ALIPUB40DXXXXXXX";

    /**
     * mPaaS控制台上对应的工作空间id
     */
    private static final String WORKSPACE_ID = "default";

    /**
     * mPaaS控制台上对应的租户id
     */
    private static final String TENANT_ID = "XVXXXXXF";

    /**
     * 地域ID，默认为 cn-hangzhou
     */
    private static final String REGION_ID = "cn-hangzhou";

    /**
     * 产品名称
     */
    private static final String PRODUCT = "mpaas";
```

```
    /**
     * 调用的endpoint
     */
    private static final String END_POINT = "mpaas.cn-hangzhou.aliyuncs.com";

    public static void main(String[] args) {
        // 阿里云账号AccessKey拥有所有API的访问权限，建议您使用RAM用户进行API访问或日常运维。
        // 强烈建议不要把AccessKey ID和AccessKey Secret保存到工程代码里，否则可能导致AccessKey泄露，威胁您账号下所有资源的安全。
        // 本示例以将AccessKey ID和AccessKey Secret保存在环境变量为例说明。您也可以根据业务需要，保存到配置文件里。
        String accessKeyId = System.getenv("MPAAS_AK_ENV");
        String accessKeySecret = System.getenv("MPAAS_SK_ENV");

        DefaultProfile.addEndpoint(REGION_ID, PRODUCT, END_POINT);
        DefaultProfile profile = DefaultProfile.getProfile(REGION_ID, accessKeyId, accessKeySecret);
        IAcsClient iAcsClient = new DefaultAcsClient(profile);
        QueryMcubeVhostRequest queryMcubeVhostRequest = new QueryMcubeVhostRequest();
        queryMcubeVhostRequest.setAppId(APP_ID);
        queryMcubeVhostRequest.setWorkspaceId(WORKSPACE_ID);
        queryMcubeVhostRequest.setTenantId(TENANT_ID);
        QueryMcubeVhostResponse acsResponse = null;
        try {
            acsResponse = iAcsClient.getAcsResponse(queryMcubeVhostRequest);
            System.out.println(acsResponse.getResultCode());
            System.out.println(acsResponse.getQueryVhostResult());
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

## 配置文件上传

由于在所有的 API 接口中均不允许出现文件流，所以需要上传的文件都应先调用上传工具类来将文件上传至 OSS，再将返回的 OSS 地址作为参数传递到指定的 API 中。

您可下载相关的文件的上传工具类 OssPostObject.java.zip。

## 使用示例

文件上传示例如下：

```
        GetMcubeFileTokenRequest getMcubeFileTokenRequest = new GetMcubeFileTokenRequest();
        getMcubeFileTokenRequest.setAppId(APP_ID);
        getMcubeFileTokenRequest.setOnexFlag(true);
        getMcubeFileTokenRequest.setTenantId(TENANT_ID);
        getMcubeFileTokenRequest.setWorkspaceId(WORKSPACE_ID);
        GetMcubeFileTokenResponse acsResponse = iAcsClient.getAcsResponse(getMcubeFileToken
Request);
        System.out.println(JSON.toJSONString(acsResponse));

        GetMcubeFileTokenResponse.GetFileTokenResult.FileToken fileToken = acsResponse.getG
etFileTokenResult().getFileToken();
        OssPostObject ossPostObject = new OssPostObject();
        ossPostObject.setKey(fileToken.getDir());
        ossPostObject.setHost(fileToken.getHost());
        ossPostObject.setOssAccessId(fileToken.getAccessid());
        ossPostObject.setPolicy(fileToken.getPolicy());
        ossPostObject.setSignature(fileToken.getSignature());
        ossPostObject.setFilePath("your/local/file/path");
        String s = ossPostObject.postObject();
```

有关 `GetMcubeFileTokenRequest` 的说明请参见 获取上传文件 token。

# 6.6.2. 接口说明

> **说明**
>
> 参数说明表格中未包含是否必填项的参数均为必填参数。

## 通用参数说明

所有接口都包含三个参数：`appId`、`workspaceId` 和 `tenantId`，这三个参数的含义如下。本文档后续接口说明中会省略对这三个参数的说明。

| 参数名称 | 类型 | 说明 |
| --- | --- | --- |
| appId | String | 所属的应用 |
| workspaceId | String | 所属的工作空间 |
| tenantId | String | 所属的租户 |

## 通用返回值说明

| 参数名称 | 类型 | 说明 |
| --- | --- | --- |

| | | |
|---|---|---|
| resultCode | String | 请求正常返回 OK，其他情况表明 API 请求异常。 |
| requestId | String | 标识请求的 ID。 |
| resultMessage | String | 请求异常时的描述。 |
| ***Result | Object | 返回的具体对象，具体含义看具体返回值。 |

所有接口返回的具体对象均包含两个字段：`success`、`resultMsg`，这两个字段的含义如下：

| 名称 | 类型 | 说明 |
|---|---|---|
| success | Boolean | 查询是否成功。 |
| resultMsg | String | 查询失败后的返回值。 |

## 创建虚拟域名

## 请求 - CreateMcubeVhostRequest

| 名称 | 类型 | 说明 |
|---|---|---|
| vhost | String | 虚拟域名的值。 |

## 返回值 - CreateMcubeVhostResponse

```
{
    "createVhostResult":{
        "data":"success",
        "resultMsg":"",
        "success":true
    },
    "requestId":"F9C681F2-6377-488D-865B-1144E0CE69D2",
    "resultCode":"OK"
}
```

### 返回值说明

| 返回值名称 | 类型 | 说明 |
|---|---|---|

| | | |
|---|---|---|
| data | String | 如果创建成功，返回 success。如果创建失败，success 字段的值为 false。 |
| createVhostResult | Object | 返回的具体对象，仅包含通用返回值。 |

## 查询虚拟域名

## 请求 - QueryMcubeVhostRequest

## 返回值 - QueryMcubeVhostResponse

```
{
    "queryVhostResult":{
        "data":"test.com",
        "resultMsg":"",
        "success":true
    },
    "requestId":"637D5BE0-0111-4C53-BCEE-473CFFA0DBAD",
    "resultCode":"OK"
}
```

### 返回值说明

| 返回值名称 | 类型 | 说明 |
|---|---|---|
| queryVhostResult | Object | 返回的具体对象，具体含义见下表。 |

在返回的对象中，包含的字段含义如下：

| 名称 | 类型 | 说明 |
|---|---|---|
| data | String | 查询到的虚拟域名信息。 |
| resultMsg | String | 查询失败后的返回值。 |
| success | Boolean | 查询是否成功。 |

## 查询密钥文件是否存在

## 请求 - ExistMcubeRsaKeyRequest

## 返回值 - ExistMcubeRsaKeyResponse

```
{
    "checkRsaKeyResult":{
        "data":"fail",
        "resultMsg":"",
        "success":false
    },
    "requestId":"8F76783A-8070-4182-895D-14E5D66F8BA3",
    "resultCode":"OK"
}
```

**返回值说明**

| 返回值名称 | 类型 | 说明 |
| --- | --- | --- |
| checkRsaKeyResult | Object | 返回的具体对象，具体含义见下表。 |

**在返回的对象中，包含的字段含义如下：**

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| data | String | 查询密钥是否存在返回结果。`fail` 表示密钥不存在，`success` 表示密钥存在。 |
| resultMsg | String | 查询失败后的返回值。 |
| success | Boolean | 查询是否成功。 |

## 获取上传文件 token

## 请求 - GetMcubeFileTokenRequest

| 参数名称 | 类型 | 说明 |
| --- | --- | --- |
| onexFlag | Boolean | 固定传值为 `true` 。 |

## 返回值 - GetMcubeFileTokenResponse

```
{
    "getFileTokenResult":{
        "fileToken":{
            "accessid":"LTAI7z7XPfKU****",
            "dir":"mds/tempFileForOnex/ONEXE9B092D/test/PUQYHL/8b574cb7-3596-403f-a0e9-2086
60fc2081/",
            "expire":"1584327372",
            "host":"https://mcube-test.oss-cn-hangzhou.aliyuncs.com",
            "policy":"QwM2YtYTBlOS0yMDg2NjBmYzIwODEvIl1dfQ==",
            "signature":"kisfP5YhbPtmES8+w="
        },
        "resultMsg":"",
        "success":true
    },
    "requestId":"8BAA3288-662E-422C-9960-2EEBFC08369F",
    "resultCode":"OK"
}
```


**返回值说明**

| 返回值名称 | 类型 | 说明 |
| --- | --- | --- |
| fileToken | Object | 按照文件上传示例中的设置方法，将 fileToken 中对应的字段设置到 OssPostObject 中。 |
| getFileTokenResult | Object | - |

## 上传密钥文件

## 请求 - UploadMcubeRsaKeyRequest

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| onexFlag | Boolean | 固定传值为 `true` 。 |
| fileUrl | String | 密钥文件在 OSS 中的存储地址。 |

## 返回值 - UploadMcubeRsaKeyResponse

```
{
    "requestId":"519E35CF-CC60-4890-8C8E-89A98CEA6BB0",
    "resultCode":"OK",
    "uploadRsaResult":{
        "data":"处理成功",
        "resultMsg":"",
        "success":true
    }
}
```

## 返回值说明

| 返回值名称 | 类型 | 说明 |
| --- | --- | --- |
| data | String | 如果创建成功，返回处理成功。<br>如果创建失败，success 字段值为 fasle。 |
| uploadRsaResult | Object | 返回的具体对象。 |

## 获取离线包 App 列表

## 请求 - ListMcubeNebulaAppsRequest

仅包含通用参数。参见 通用参数说明。

## 返回值 - ListMcubeNebulaAppsResponse

```
{
    "listMcubeNebulaAppsResult":{
        "nebulaAppInfos":[
            {
                "h5Id":"12345678",
                "h5Name":"12345678"
            },
            {
                "h5Id":"12345679",
                "h5Name":"openapiTest"
            }
        ],
        "resultMsg":"",
        "success":true
    },
    "requestId":"BE728F09-6EBD-4688-9329-896813EAD075",
    "resultCode":"OK"
}
```

## 返回值说明

| 返回值名称 | 类型 | 说明 |
| --- | --- | --- |
| h5Id | String | 离线包 ID。 |
| h5Name | String | 离线包名称。 |

## 创建离线包 App

### 请求 - CreateMcubeNebulaAppRequest

| 参数名称 | 类型 | 说明 |
| --- | --- | --- |
| h5Name | String | 离线包名称。 |
| h5Id | String | 离线包 ID，8 位数字。 |

### 返回值 - CreateMcubeNebulaAppResponse

```
{
    "createNebulaAppResult":{
        "resultMsg":"",
        "success":true
    },
    "requestId":"5B588AFE-8D58-4460-B0AA-6A48A9FD0852",
    "resultCode":"OK"
}
```

## 删除离线包 App

### 请求 - DeleteMcubeNebulaAppRequest

| 参数名称 | 类型 | 说明 |
| --- | --- | --- |
| h5Id | String | 离线包 ID，8 位数字 |

### 返回值 - DeleteMcubeNebulaAppResponse

```
{
    "deleteMcubeNebulaAppResult":{
        "resultMsg":"",
        "success":true
    },
    "requestId":"E24C760E-4849-4341-91C6-6DA97F5B6B76",
    "resultCode":"OK"
}
```

## 上传离线资源包

## 请求 - CreateMcubeNebulaResourceRequest

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| h5Id | String | H5App 的 ID。 |
| h5Name | String | H5App 的名称。 |
| h5Version | String | 离线包的版本。需要保证在单个 H5App 中唯一。 |
| mainUrl | String | 离线包主入口，满足正则 `^/[\w\|/]+\.html$` 。 |
| vhost | String | H5App 的虚拟域名。 |
| extendInfo | String | JSON 格式字符串。 |
| autoInstall | Integer | 下载时机。<br>• 0：仅 Wi-Fi（非 Wi-Fi 需用户使用应用时才会下载）；<br>• 1：所有网络都下载（会对用户流量造成负面影响，非特殊场景禁用。 |
| resourceType | Integer | 资源类型，一个 H5App 只能存在一种类型。0：全局资源包，1：普通资源包。 |

| | | |
|---|---|---|
| installType | Integer | 安装时机，0：不预加载（只有进入离线包或小程序页面时才安装），1：预加载（离线包或小程序下载完成后则自动安装）。 |
| platform | String | 使用平台，分为 all（全平台）、Android 和 iOS。 |
| clientVersionMin | String | 客户端最低版本，选择指定的 `platform` 后，最低版本必传，格式为 `aaa;bbb`，其中 `aaa` 对应 iOS 的版本，`bbb` 对应 Android 的版本，如果平台只选择了一个，参数值中的分号也不能省略，比如只选择 Android，那么值为”;bbb”。 |
| clientVersionMax | String | 客户端最高版本，可以不填，若 `platform` 为 `all`，则这个值必须成对存在，iOS 与 Android 的最高版本或都填，或都为空。 |
| fileUrl | String | 文件在 OSS 的 URL。离线包资源文件，必须为 `zip` 格式。 |
| repeatNebula | Integer | 是否复用全局包，在资源类型为全局资源包时需要填写。0：否，1：是。 |
| onexFlag | Boolean | 固定传值为 `true`。 |

## 返回值 - CreateMcubeNebulaResourceResponse

```
{
    "createMcubeNebulaResourceReslult":{
        "nebulaResourceId":"4154",
        "resultMsg":"",
        "success":true
    },
    "requestId":"DFCA28DF-0F97-4C41-B3D4-351D284B51E7",
    "resultCode":"OK"
}
```

`nebulaResourceId` 为上传的资源包对应的 ID。

## 获取资源包列表

### 请求 - ListMcubeNebulaResourcesRequest

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| h5Id | String | H5App 的 ID。 |

### 返回值 - ListMcubeNebulaResourcesResponse

```
{
    "listMcubeNebulaResourceResult":{
        "nebulaResourceInfos":[
            {
                "appCode":"ONEX97C5D29290957-default",
                "autoInstall":1,
                "clientVersionMax":"100;100",
                "clientVersionMin":"0;0",
                "creator":"demo",
                "debugUrl":"",
                "downloadUrl":"https://pre-mpaas.cn-hangzhou.oss.aliyuncs.com/ONEX97C5D2929
0957-default/12345678/1.0.0.1_all/nebula/12345678_1.0.0.1.amr",
                "extendInfo":"",
                "extraData":"{"resourceType":"1"}",
                "fallbackBaseUrl":"https://pre-mpaas.cn-hangzhou.oss.aliyuncs.com/ONEX97C5D
29290957-default/12345678/1.0.0.1_all/nebula/fallback/;https://pre-mpaas.cn-hangzhou.oss.al
iyuncs.com/ONEX97C5D29290957-default/12345678/1.0.0.1_all/nebula/fallback/",
                "fileSize":"0",
                "gmtCreate":"2021-02-01 14:11:21",
                "gmtModified":"2021-02-01 14:11:21",
                "h5Id":"12345678",
                "h5Name":"12345678",
                "h5Version":"1.0.0.1",
                "id":4154,
                "installType":1,
                "lazyLoad":0,
                "mainUrl":"/test.html",
                "md5":"3b9b7caaea6e5b0cb0db4db551454a33",
                "memo":"https://pre-mpaas.cn-hangzhou.oss.aliyuncs.com/ONEX97C5D29290957-de
fault/12345678/1.0.0.1_all/nebula/nebula_json/h5_json.json",
                "metaId":7848,
                "modifier":"success",
                "packageType":1,
                "platform":"all",
                "publishPeriod":0,
                "releaseVersion":"20210201141121",
                "resourceType":"1",
                "status":1,
                "vhost":""
            }
        ],
        "resultMsg":"",
        "success":true
    },
    "requestId":"C88DEB27-FF7E-43F7-97F8-B2AA12FB0A5D",
    "resultCode":"OK"
}
```


### 返回值说明

| 名称 | 类型 | 说明 |
| --- | --- | --- |

| | | |
|---|---|---|
| appCode | String | appId+” - “+workspaceId |
| autoInstall | Integer | 含义和上传离线包中的一致 |
| clientVersionMax | String | 含义和上传离线包中的一致 |
| clientVersionMin | String | 含义和上传离线包中的一致 |
| creator | String | 创建者，目前没有使用 |
| debugUrl | String | 当前返回中无意义 |
| downloadUrl | String | 下载离线包 AMR 文件地址 |
| extendInfo | String | 上传时传递的扩展信息 |
| extraData | String | 扩展参数 |
| fallbackBaseUrl | String | 离线包 fallback 地址，使用分号分隔，分号前是内网地址，分号后是外网地址 |
| fileSize | String | 文件大小 |
| gmtCreate | Date | 创建时间 |
| gmtModified | Date | 更新时间 |
| h5Id | String | H5App 的 ID |
| h5Name | String | H5App 的名称 |
| h5Version | String | 当前离线包的版本号 |
| id | Long | 主键 |

| | | |
|---|---|---|
| installType | Integer | 含义和上传离线包中的一致 |
| lazyLoad | Integer | 启动加载，目前都是 0 |
| mainUrl | String | 含义和上传离线包中的一致 |
| md5 | String | 离线包文件的md5 |
| memo | String | 离线包的 `h5.json` 文件的下载地址 |
| metaId | Long | 无意义 |
| modifier | 修改者 | 目前没有使用 |
| platform | 平台 | 含义和上传离线包中的一致 |
| publishPeriod | Integer | 发布状态。<br>0：初始化；<br>1：内部灰度发布；<br>2：外部灰度发布；<br>3：正式发布；<br>4：回滚发布；<br>5：发布任务结束。 |
| releaseVersion | String | 发布版本号 |
| resourceType | Integer | 含义和上传离线包中的一致 |
| status | Integer | 状态 |

## 创建离线包发布任务

### 请求 - CreateMcubeNebulaTaskRequest

| 名称 | 类型 | 是否必填 | 说明 |
|---|---|---|---|

| | | | |
|---|---|---|---|
| publishType | Integer | 是 | 发布类型。<br>• 2：灰度发布<br>• 3：正式发布 |
| publishMode | Integer | 否 | 发布模式。若 `publishType` 为 3，则不填。<br>• 0：未知<br>• 1：白名单<br>• 2：时间窗<br>• 3：百分比<br>• 4：全量<br>• 5：第三方灰度 |
| memo | String | 否 | 发布描述 |
| id | Long | 是 | 只能传 0，表示创建，不可修改。 |
| greyEndtimeData | String | 否 | 灰度时间窗发布的结束时间，格式为“YYYY-MM-dd HH:mm:ss”，时间必须大于当前时间并且与当前时间的间隔小于 7 天。当 publishMode 为 2 的时候必填。 |
| greyEndTime | Date | 否 | Date 类型，值和 `greyEndTimeData` 一致。 |
| greyNum | Integer | 否 | 时间窗灰度的人数。当 `publishMode` 为 `2` 时必填。 |
| whitelistIds | String | 否 | 白名单主键 ID。当 `publishMode` 为 `1` 时必填。多个 ID 使用”,”分隔。 |
| packageId | Long | 是 | 发布的资源包主键 ID |

| | | | |
|---|---|---|---|
| greyConfigInfo | String | 否 | 发布的高级规则条件，JSON 字符串，具体含义见下表。示例：[{ “ruleElement” :” city” ,” operation” :1,” value” :” 上海市,北京市,天津市” }, { “ruleElement” :” mobileModel” ,” operation” :2,” value” :” REDMI NOTE 3,VIVO X5M” }, { “ruleElement” :” osVersion” ,” operation” :3,” value2” :” 9.2.1” ,” value1” :” 9.2.1” ,” value” :” 9.2.1-9.2.1” }] |

**高级规则说明**

| 名称 | 类型 | 说明 |
|---|---|---|
| ruleElement | String | 规则类型：<br>• city：城市<br>• mobileModel：机型<br>• netType：网络<br>• osVersion：设备系统版本 |
| value | String | 规则值，多个规则使用”,” 分隔，当 `operation` 为 `3` 或 `4` 时，`value` 值是 `aa-bb` 的格式，其中 `aa` 是较小的值，`bb` 是较大的值。 |

| | | |
|---|---|---|
| operation | Integer | 操作关系：<br>• 1：包含<br>• 2：不包含<br>• 3：范围内<br>• 4：在范围外。<br>当 `ruleElement` 为 `city` 、 `mobileModel` 和 `netType` 时， `operation` 取值仅可以为 `1` 或 `2` ；当 `ruleElement` 为 `osVersion` 时， `operation` 的值可以是 4 种里面的任意一种。 |

## 返回值 - CreateMcubeNebulaTaskResponse

```
{
    "createMcubeNebulaTaskResult":{
        "nebulaTaskId":"6664",
        "resultMsg":"",
        "success":true
    },
    "requestId":"BBDF54E1-2783-4E5A-AE19-F7BC3A1BB3C2",
    "resultCode":"OK"
}
```

返回的nebulaTaskId为创建的发布任务对应的 ID。

## 获取发布任务列表

## 请求 - ListMcubeNebulaTasksRequest

| 名称 | 类型 | 说明 |
|---|---|---|
| id | Long | 任务对应的离线资源包 ID |

## 返回值 - ListMcubeNebulaTasksResponse

```
{
    "listMcubeNebulaTaskResult":{
        "nebulaTaskInfos":[
            {
                "appCode":"ONEX97C5D29290957-default",
                "bizType":"nebula",
                "creator":"",
                "gmtCreate":"2021-02-01 14:16:58",
                "gmtModified":"2021-02-01 14:16:58",
                "gmtModifiedStr":"2021-02-01 14:16:58",
                "greyConfigInfo":"",
                "greyEndtimeData":"",
                "greyNum":0,
                "greyUrl":"",
                "id":6664,
                "memo":"test",
                "modifier":"",
                "packageId":4154,
                "percent":0,
                "platform":"all",
                "productId":"ONEX97C5D29290957-default-12345678",
                "productVersion":"1.0.0.1",
                "publishMode":4,
                "publishType":3,
                "releaseVersion":"20210201141121",
                "status":1,
                "syncResult":"",
                "taskName":"12345678",
                "taskStatus":1,
                "taskType":0,
                "taskVersion":1612160218556,
                "upgradeNoticeNum":0,
                "upgradeProgress":"",
                "whitelistIds":""
            }
        ],
        "resultMsg":"",
        "success":true
    },
    "requestId":"B9A07543-4B8B-43D0-AB33-7F2ACB954909",
    "resultCode":"OK"
}
```


### 返回值说明

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| appCode | String | appId+workspaceId |

| | | |
|---|---|---|
| bizType | String | 离线包为 `nebula` |
| bundles | Array | 目前没有使用 |
| creator | String | 目前没有使用 |
| gmtCreate | Date | 创建时间 |
| gmtModified | Date | 更新时间 |
| gmtModifiedStr | String | 更新时间字符串 |
| greyConfigInfo | String | 高级规则的字符串，和上传时的展示方式不同，具体见下表。 |
| greyEndtime | Date | 时间窗灰度截止时间 |
| greyEndtimeData | String | 时间窗灰度截止时间字符串 |
| greyNum | Integer | 时间窗灰度人数 |
| id | Long | 当前发布任务主键 ID |
| memo | String | 发布描述 |
| modifier | String | 更新者，没有使用 |
| packageId | Long | 当前任务对应离线资源包的 ID |
| percent | Integer | 灰度百分比，目前都是 0。 |
| platform | String | 当前发布任务的平台，可选 all（双平台）、iOS、Android。 |
| productId | String | 产品 ID，格式为“appId + workspaceId + h5id”。 |

| | | |
|---|---|---|
| productVersion | String | 离线资源包的版本号 |
| publishMode | Integer | 发布模型。0：默认值，1：白名单，2：时间窗。 |
| publishType | Integer | 发布类型。2：灰度发布；3：正式发布。 |
| releaseVersion | String | 内部发布版本号 |
| resIds | String | 对应的离线资源包 ID |
| status | Integer | 状态。0：无效，1：有效 |
| syncResult | String | 目前没有使用 |
| taskName | String | 任务名称，和小程序 App 名称相同 |
| taskStatus | Integer | 任务状态。0：待发布；1.发布中；2：已结束；3：暂停 |
| taskType | Integer | 任务类型。0：普通任务 1：回滚任务。 |
| taskVersion | Long | 任务版本号，使用的是任务创建的当前时间 |
| upgradeNoticeNum | Integer | 目前没有使用 |
| upgradeProgress | String | 目前没有使用 |
| whitelistIds | String | 白名单主键 ID，多个 ID 使用”,” 分隔 |

**greyConfigInfo 字段内容解释**

| 名称 | 类型 | 说明 |
|---|---|---|

| | | |
|---|---|---|
| operator | String | 规则关系，`and` 表示“与”规则，对 `subRules` 里的结果进行“与”操作。 |
| defaultResult | boolean | 默认返回的结果 |
| subRules | List | 规则集合 |
| operator | String | 规则名称<br>• contains：包含<br>• excludes：不包含<br>• vLimitIn：在范围内<br>• vLimitOut：在范围外 |
| left | List/Object | 当 `operator` 为 `contains` 或 `excludes` 时，是 List 字符集合，每个元素表示一个规则的值；当 `operator` 为 `vLimitIn` 和 `vLimitOut` 时，是一个对象，里面的 `lower` 表示较低的值，`upper` 表示较高的值。 |
| right | String | 规则类型名称 |
| defaultResult | Boolean | 默认结果 |

**说明**

greyConfigInfo 中两个 operator 字段表示的含义不同。

```
{
    "operator":"and",
    "subRules":[
        {
            "operator":"excludes",
            "left":[
                "青岛市",
                "长沙市",
                "重庆市"
            ],
            "right":"city",
            "defaultResult":false
        },
        {
            "operator":"contains",
            "left":[
                "2G",
                "4G",
                "WIFI"
            ],
            "right":"netType",
            "defaultResult":false
        },
        {
            "operator":"contains",
            "left":[
                "phone4",
                "plusx"
            ],
            "right":"mobileModel",
            "defaultResult":false
        },
        {
            "operator":"vLimitOut",
            "exclusive":true,
            "defaultResult":true,
            "left":{
                "lower":"12.0",
                "upper":"17.0"
            },
            "right":"osVersion"
        }
    ],
    "defaultResult":false
}
```

## 根据 ID 获取任务详情

## 请求 - GetMcubeNebulaTaskDetailRequest

| 名称 | 类型 | 说明 |
| --- | --- | --- |

| taskId | Long | 想要查询的任务 ID 主键 |
| --- | --- | --- |

## 返回值 - GetMcubeNebulaTaskDetailResponse


```
{
    "getMcubeNebulaTaskDetailResult":{
        "nebulaTaskDetail":{
            "appCode":"ONEX97C5D29290957-default",
            "appId":"",
            "atomic":0,
            "baseInfoId":0,
            "bizType":"nebula",
            "creator":"",
            "cronexpress":0,
            "downloadUrl":"https://pre-mpaas.cn-hangzhou.oss.aliyuncs.com/ONEX97C5D29290957
-default/12345678/1.0.0.1_all/nebula/12345678_1.0.0.1.amr;https://pre-mpaas.cn-hangzhou.oss
.aliyuncs.com/ONEX97C5D29290957-default/12345678/1.0.0.1_all/nebula/12345678_1.0.0.1.amr",
            "extraData":"{"resourceType":"1"}",
            "fileSize":"0",
            "fullRepair":0,
            "gmtCreate":"2021-02-01 14:16:58",
            "gmtModified":"2021-02-01 14:16:58",
            "gmtModifiedStr":"2021-02-01 14:16:58",
            "greyConfigInfo":"",
            "greyEndtimeData":"",
            "greyNum":0,
            "greyUrl":"",
            "id":6664,
            "issueDesc":"",
            "memo":"test",
            "modifier":"",
            "ossPath":"",
            "packageId":4154,
            "percent":0,
            "platform":"all",
            "productId":"ONEX97C5D29290957-default-12345678",
            "productVersion":"1.0.0.1",
            "publishMode":4,
            "publishPeriod":3,
            "publishType":3,
            "quickRollback":0,
            "releaseVersion":"20210201141121",
            "ruleJsonList":[

            ],
            "sourceId":"",
            "sourceName":"",
            "sourceType":"",
            "status":1,
            "syncResult":"",
            "syncType":0,
            "taskName":"12345678",
```

```
            "taskStatus":1,
            "taskType":0,
            "taskVersion":1612160218556,
            "upgradeNoticeNum":0,
            "upgradeProgress":"",
            "whitelistIds":"",
            "workspaceId":""
        },
        "resultMsg":"",
        "success":true
    },
    "requestId":"072AE251-B9F8-4A44-A621-9F0325EECC1E",
    "resultCode":"OK"
}
```


**返回值说明**

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| appCode | String | appId+workspaceId |
| appId | String | 没有使用 |
| atomic | Integer | 1 为原子包， 0 为组合包，目前可以忽略 |
| baseInfoId | Long | 关联的基础信息的主键 ID，可以忽略 |
| bizType | String | 离线包为 `nebula` |
| bundles | List | 没有使用 |
| creator | String | 没有使用 |
| cronexpress | Integer | iOS 使用 0，表示执行一次，1 表示执行多次 |
| downloadUrl | String | 下载地址。分号前半部分为内网地址，后半部分为公网地址 |
| extraData | String | JSON 字符串，扩展数据 |

| | | |
|---|---|---|
| fileSize | String | 文件大小 |
| gmtCreate | Date | 创建时间 |
| gmtModified | Date | 更新时间 |
| greyConfigInfo | String | 高级规则字符串 |
| greyEndTime | Date | 时间窗灰度结束时间 |
| greyEndtimeData | String | 时间窗灰度发布结束时间字符串 |
| id | Long | 主键 ID |
| issueDesc | String | 问题描述，目前没有使用 |
| mds | String | 文件的 md5 值 |
| memo | String | 发布描述 |
| modifier | String | 修改者，没有使用 |
| ossPath | Stirng | 离线包没有使用 |
| packageId | Long | 发布任务对应的离线资源包 ID |
| percent | Integer | 发布百分比，离线包没有使用 |
| platform | String | 发布平台，值可以是 all、iOS、Android |
| product_id | Sting | 格式为“appId+workspaceId + H5Appid” |
| productVersion | String | 离线资源包的版本 |

| | | |
|---|---|---|
| resIds | String | 离线资源包的 ID |
| ruleJsonList | List | 发布高级规则的对象形式，按照上面的字符串形式使用即可 |
| sourceId | String | 来源 ID，离线包没有使用 |
| sourceName | String | 离线包没有使用 |
| sourceType | String | 来源类型，离线包没有使用 |
| status | Integer | 状态：<br>• 0：失效<br>• 1：有效 |
| syncResult | String | 离线包目前没有使用 |
| syncType | String | 离线包没有使用 |
| taskName | String | 任务名称 |
| taskStatus | Integer | 任务状态。<br>• 0：待发布<br>• 1：发布中<br>• 2：已结束<br>• 3：暂停 |
| taskType | Integer | 任务类型<br>• 0：普通任务<br>• 1：回滚任务 |
| taskVersion | Long | 发布版本号，是创建发布的当前时间戳 |
| upgradeNoticeNum | Integer | 目前没有使用 |
| upgradeProgress | String | 目前没有使用 |

| | | |
|---|---|---|
| vmType | Stirng | Android 虚拟机类型，逗号分隔。<br>• 1：art<br>• 2：dalvik<br>• 3：lemur<br>• 4：aoc |
| whitelist | List | 离线包发布任务的白名单信息。详情请参考 白名单管理。 |

## 修改离线包任务状态

## 请求 - ChangeMcubeNebulaTaskStatusRequest

| 名称 | 类型 | 说明 |
|---|---|---|
| bizType | String | 传 nebula |
| packageId | Long | 任务对应的离线资源包的 ID |
| taskId | Long | 当前发布任务的 ID |
| taskStatus | Integer | 需要改变到的状态。<br>• 0：待发布<br>• 1：发布中<br>• 2：已结束<br>• 3：暂停 |

## 返回值 - ChangeMcubeNebulaTaskStatusResponse

```
{
    "changeMcubeNebulaTaskStatusResult":{
        "resultMsg":"",
        "success":true
    },
    "requestId":"595F4CB4-ACFE-4A5B-AF5B-4ED837CAEF95",
    "resultCode":"OK"
}
```

# 7.开关配置管理

## 7.1. 接入 Android

本文介绍如何集成 mPaaS 提供的开关配置功能。

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。目前，开关配置支持 **原生 AAR 接入** 和 **组件化接入** 两种接入方式。

使用开关配置涉及到调用 MDS 的更新发布接口，会产生相应的接口调用费用。有关接口调用的计费说明，参见 产品定价 中的实时发布计费项说明 。

整个过程分为以下三步：

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

### 前置条件

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

#### 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，参考 管理组件依赖。

### 初始化 mPaaS

如果您使用原生 AAR 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。
2. 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。

> **重要**
>
> 监听器会以软引用形式存在，当内存较低时，系统会进行回收。因此请尽量避免使用全局监听，而是采用随时注册、用完移除的方式使用开关监听。

```
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 con
figChangeListener);
    /**
     * 移除开关改变监听器
     * @param configChangeListener 监听器
     */
    public static void removeConfigChangeListener(ConfigService.ConfigChangeListener con
figChangeListener);
}
```

# 7.2. 接入 iOS

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。

使用开关配置涉及到调用 MDS 的更新发布接口，会产生相应的接口调用费用。有关接口调用的计费说明，参见 产品定价 中的实时发布计费项说明 。

## 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

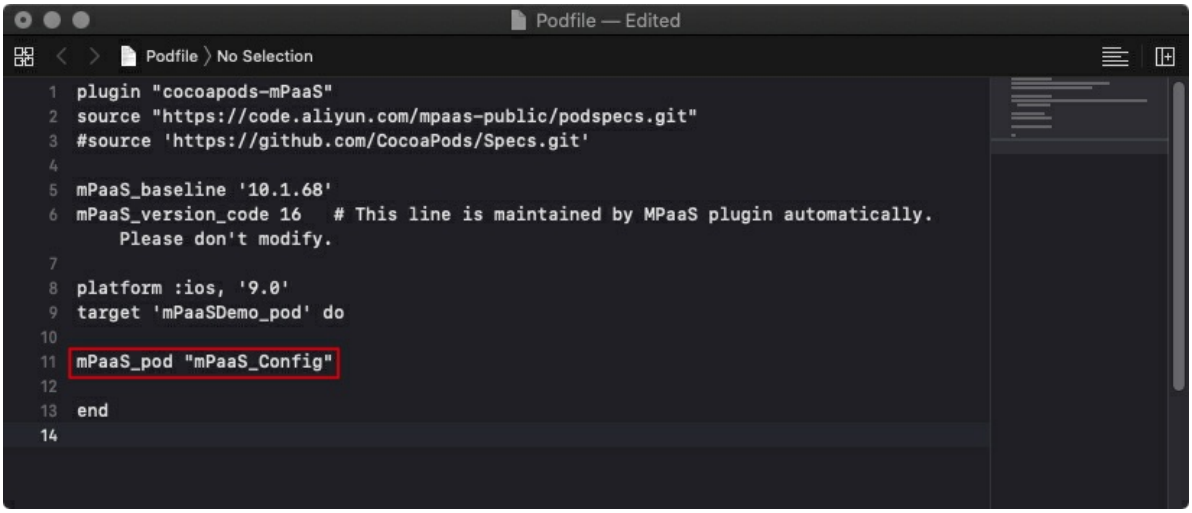
此方式适用于 基于 mPaaS 框架接入 或 基于已有工程且使用 mPaaS 插件接入 的接入方式。

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。

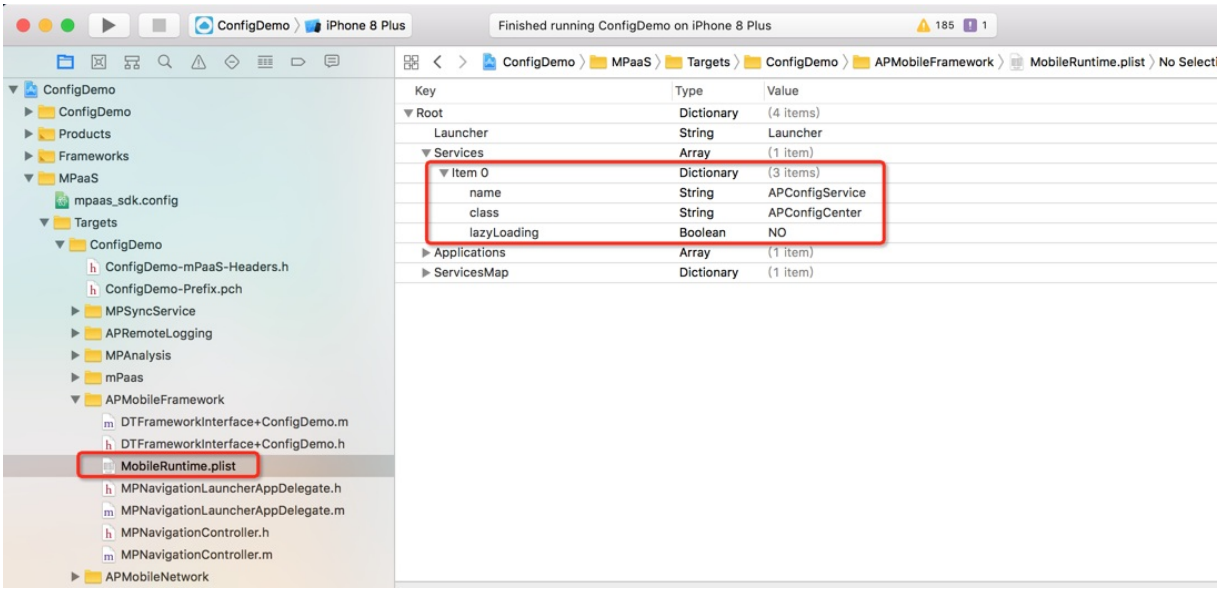



2. 参考 CocoaPods 使用指南，根据需要执行 `pod install` 或 `pod update` 即可完成接入。

## 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。

APConfig 〉 Sources 〉 APConfigService.h 〉 APConfigObserverProtocol

```
48
49  /**
50   * 对指定的配置加观察者。
51   *
52   * @param observer 观察者
53   * @param key      指定配置的key
54   *
55   * @return 成功返回YES，否则返回NO，失败原因：参数错误或者重复观察。
56   */
57  - (BOOL)addConfigChangedObserver:(id<APConfigObserverProtocol>)observer key:(NSString *)key;
```

- 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 版本等信息针对性地下发配置。

## 前提条件

已在 Android 或 iOS 客户端添加开关配置服务的 SDK。

## 添加开关配置

添加开关配置时，既可以逐个添加配置，也可以通过导入 JSON 文件的方式一次添加多个开关配置。

开关配置列表展示了各个开关配置项的配置键、状态、创建时间、更新时间、创建人、修改人信息。新建的开关配置默认处于激活状态。

## 单个添加

进入 mPaaS 控制台，完成以下步骤：

1. 在左侧导航栏，单击 **实时发布 > 配置开关管理**。
2. 单击 **添加配置**，在 **新建配置** 页面中，输入配置键、备注、资源值，选择相应的平台和分类。资源值即配置键对应的配置值，一个配置键可以对应多个资源值，单击 **添加** 按钮可设置多个资源值。
3. 添加高级规则（可选）。在资源值配置框中，单击 **高级规则** 右侧的 **添加** 按钮，可选择资源类型（版本号、osVersion、机型、城市）、操作类型，并设置相应的资源值。 每个资源值下面，都可以添加多条高级规则。

4. 配置完毕后，单击 **完成**，新添加的开关配置将展示在配置列表中。

## 批量导入

在 **配置开关管理** 页面，单击配置列表上方的 **更多操作** > **导入文件** 菜单，导入 JSON 格式的开关配置文件。文件中可以包含多条开关配置。导入成功后，导入的配置将展示在配置列表中。

> **重要**
>
> 如配置文件中包含的开关配置信息已存在，则该条配置将无法导入。

在提供批量导入功能的同时，实时发布平台还提供配置导出功能。在 **配置开关管理** 页面，单击配置列表上方的 **更多操作** > **配置导出** 菜单。勾选要导出的配置项后，单击 **配置导出** 按钮，即可下载 JSON 配置文件到本地。

## 查询开关配置

在 **配置开关管理** 页面，输入配置键关键字查询开关配置。

## 修改开关配置

完成以下步骤，修改开关配置：

1. 在 **配置开关管理** 页面，选择目标配置，单击配置键，打开编辑页面。
2. 修改备注、资源值、平台、分类或高级规则后，单击 **完成** 即可。配置键一旦添加，不可修改。

## 使开关生/失效

新建的开关配置默认处于激活状态，若不需要某个配置，可单击 **失效** 使该配置项处于失效状态。同样，针对失效状态的配置项，单击 **激活** 可使其重新生效。

## 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

白名单管理是实时发布的一项基础功能，它为实时发布提供了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。在白名单管理界面上，您可以创建白名单、添加白名单的用户信息、删除白名单。

## 创建白名单

1. 进入 mPaaS 控制台，单击左侧导航栏中的 **实时发布 > 白名单管理**， 进入白名单列表页。
2. 在白名单列表页中单击 **添加白名单**，在弹出的窗口中输入白名单名称，选择白名单类型，然后单击 **确定**，完成白名单创建。
   - **普通类型**：白名单内容为具体的用户 ID，只有完全一致才表示命中白名单。
   - **正则模式**：白名单内容为多个正则表达式，只要满足其中任一正则表达式，即代表命中白名单。

## 添加白名单的用户信息

1. 单击白名单列表中指定白名单右侧的 **增加**，在弹出的窗口中输入要加入白名单的用户 ID 或正则表达式。
   - 添加普通类型白名单用户 ID 由客户端配置，具体方法参考 添加用户 ID。多个用户 ID 用英文逗号或换行分隔，也可以上传包含用户信息的白名单文件。

- 添加正则模式白名单用户


2. 输入完毕后，单击 **确定**。

## 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，您可以添加资源、修改资源配置、删除资源。

## 添加资源

1. 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。
2. 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：

1. 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] | ■ 若基线版本 < 10.1.68.32，使用 [APMobileIdentifier shareIdentifier].deviceModel。<br>■ 若基线版本 ≥ 10.1.68.32，使用 [MPaaSDVInfo sharedInstance].deviceModel。 |
| 升级检测 | 无线：WIFI移动网络：WWAN | [[UIDevice currentDevice] systemVersion] | ■ 若基线版本 < 10.1.68.32，使用 [APMobileIdentifier shareIdentifier].deviceModel。<br>■ 若基线版本 ≥ 10.1.68.32，使用 [MPaaSDVInfo sharedInstance].deviceModel。 |
| 热修复管理离线包管理小程序管理 | [DTReachability networkName] | [[UIDevice currentDevice] systemVersion] | ■ 若基线版本 < 10.1.68.32，使用 [APMobileIdentifier shareIdentifier].deviceModel。<br>■ 若基线版本 ≥ 10.1.68.32，使用 [MPaaSDVInfo sharedInstance].deviceModel。 |

2. 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。

# 10.参考

# 10.1. API 说明

了解 Android 的升级 SDK 中相关 API 接口的使用方法。

- MPaaSCheckVersionService
- MPaaSCheckCallBack

## MPaaSCheckVersionService

### 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

### 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)
```

# 10.2. 代码示例

# 10.2.1. 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。用户单击 升级 自动升级，无需其他编码。如需自定义升级提示窗口，请参考下方的自定义升级提示 UI 说明。

```
- (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];
}
```

# 10.2.2. 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

**获取 Demo**

参考 代码示例 获取示例代码。

# 10.2.3. 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。