蚂蚁科技

接入 Android
使用指南

文档版本：20231225

蚂蚁集团
ANT GROUP

法律声明

商标声明

蚂蚁集团 ANT GROUP及其他蚂蚁集团相关的商标均为蚂蚁集团所有。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。蚂蚁集团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，并在蚂蚁集团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集团授权渠道下载、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。

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 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 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 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 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` *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.将 mPaaS 添加到您的项目中

1.1. 前提条件

本文介绍了将 mPaaS 接入到项目前，需要完成的准备工作。

在开始将 mPaaS 添加到您的项目之前，需要完成以下准备工作以满足接入条件。

- 安装 Android Studio
- 安装 Android Studio mPaaS 插件
- 注册阿里云账号
- 开通 mPaaS 产品
- 适配不同的 CPU 架构并设定 targetSdkVersion

安装 Android Studio

- 关于 Android Studio 下载，请参见 Android Developers。
- 关于安装 Android Studio，请参见 安装指南。
- 如果您之前使用的是低版本 Android Studio 并已经安装了 mPaaS 插件，那么在您从低版本 Android Studio 升级至最新版本的 Android Studio 之后，您只需要升级 mPaaS 插件至最新版本即可。详情请参见 更新 mPaaS 插件。

安装 Android Studio mPaaS 插件

使用 mPaaS 需要 Android Studio mPaaS 插件进行辅助管理，因此您需要先安装 Android Studio mPaaS 插件。请参见 安装 mPaaS 插件 以获得更多关于插件安装的信息。

注册阿里云账号

使用 mPaaS 需要阿里云账号，进行 mPaaS 控制台管理，因此您需要准备一个阿里云账号。请参见 注册阿里云账号 以获得更多注册流程指导。

开通 mPaaS 产品

登录阿里云控制台，在 mPaaS 产品页 ，单击 **管理控制台**，进入 **开通产品** 页面。单击 **立即开通**，即可开通 mPaaS 产品。

适配不同的 CPU 架构并设定 targetSdkVersion

mPaaS 支持 armeabi、armeabi-v7a、arm64-v8a 三种 CPU 架构，并支持设置 targetSdkVersion 为 26（默认）、28 和 29。在接入 mPaaS 时，需要在工程主 Module 下的 `build.gradle` 文件中添加以下配置，适配 CPU 架构并设定 targetSdkVersion。

```
android {
    ...
    defaultConfig {
        ...
        targetSdkVersion 26
        ndk{
            abiFilters 'armeabi'
        }
        ...
    }
    ...
}
```

如需要设置 targetSdkVersion 为 28 或 29，请参考以下文档完成配置。

- mPaaS 适配 targetSdkVersion 28
- mPaaS 适配 targetSdkVersion 29

> **说明**
>
> 如果有更多接入相关问题，欢迎搜索群号 41708565 加入钉钉群进行咨询交流。该钉钉群已添加 mPaaS 公有云答疑小助手，能够快速回答常见接入问题。更多关于使用公有云答疑小助手的信息，请参见公有云答疑小助手 。

1.2. 步骤 1 选择合适的接入方式

mPaaS Android 提供了原生 AAR 方式和组件化接入（Portal&Bundle）两种接入方式。如果您是初次接触 mPaaS ，建议您使用 **原生 AAR 方式** ，此种方式比组件化接入方式更贴近 Android 技术栈，方便您快速上手。更多关于接入方式的信息，请参见 接入方式简介 。

1.3. 步骤 2 在控制台创建 mPaaS 应用

本文将介绍如何在控制台创建 mPaaS 应用的操作流程。

操作步骤

1. 登录 mPaaS 控制台。
2. 单击 **创建应用** 。最多可免费创建 20 个应用，即创建应用不会产生费用。
3. 完善应用信息。
   i. （必选）输入应用名称。应用名称示例：mPaaS Demo。
   ii. （可选）单击 **+** 上传应用图标。若不上传，则使用默认图标。应用图标大小应不超过 1 MB，尺寸在 50 PX - 200 PX，图片格式为 JPG 或 PNG。
4. 单击 **创建**，完成应用创建，并跳转到应用总览页面。

1.4. 步骤 3 将配置文件添加到项目中

本文档基于 **原生 AAR 方式** 介绍了将配置文件手动导入到项目中的过程。该过程分为三步：一、填写配置信息，并上传签名 APK，二、下载配置到本地，三、将配置文件添加到项目中。

填写配置信息，并上传签名 APK

1. 在应用列表页，单击应用卡片（如上一步中创建的应用 mPaaS Demo），进入应用详情页。
   - 左上方展示应用名称，您可以在此切换应用。
   - 在 **应用详情** 页，单击 **Android 代码配置**，打开应用配置面板。
2. 在应用配置面板中，单击 **下载配置文件** 链接，打开 **代码配置** 页。
3. 在 **代码配置** 页，输入 **Package Name（应用包名）**（此处以 **com.mpaas.demo** 为例），上传编译并添加签名后的 APK 安装包。关于快速生成签名后的 APK 相关信息，请参见 生成控制台用签名 APK。

下载配置到本地

在 **代码配置** 页，填写完成后，单击 **下载配置**，即可获取 mPaaS 的配置文件。

配置文件是一个压缩包文件。该压缩包包含一个 `.config` 文件以及一个 `yw_1222.jpg` 加密图片。

- 如果您是公有云用户，请确认 `.config` 文件中 `base64Code` 的 value 不为空。由于公有云环境已弃用 `yw_1222.jpg` 文件，请忽略。

- 如果您是专有云用户，无需关注 `base64Code` 的 value，只需参考 生成加密图片（专有云配置文件）手动生成专有云加密图片，替换从控制台下载到的 `yw_1222.jpg` 文件。

将配置文件添加到项目中

- 如果您使用 **组件化方式（Portal&Bundle）**，请参考 组件化接入流程简介 。

前提条件

采用 **原生 AAR 方式** 接入时，您应已有一个原生开发工程。

操作步骤

1. 在 Android Studio 中打开已有工程，单击 **mPaaS** > **原生 AAR 接入**。
2. 在弹出的接入面板中，点击导入 App 配置下的 **开始导入**。
3. 选择 **我已经从控制台上下载配置文件（Ant-mpaas-xxxx.config），准备导入到工程** ，单击 **Next**。
4. 在 **导入 mPaaS 配置文件** 窗口中，选择待导入的 App Module，并选择配置文件，单击 **Finish**。
5. 导入成功后，将会收到导入配置文件成功的提示信息。至此，您已完成手动导入配置文件。

另外，mPaaS 插件还支持 **自动拉取** 的方式添加配置文件。如上文所述，手动导入需要在控制台下载配置文件后，再通过 mPaaS 插件手动添加到工程里。自动拉取方式则是使用账号 Access Key 信息登录 mPaaS 插件，mPaaS 插件能够从控制台自动拉取到配置文件并添加到工程中。更多详情，请参见 自动拉取配置文件。

1.5. 步骤 4 mPaaS 10.2.3 支持无线保镖&蓝盾切换（可选）

背景

无线保镖客户端 SDK 与无线保镖图片搭配作为 mPaaS 的基础依赖能力之一，在 mPaaS 产品中广泛使用，为了进一步提升 mPaaS 产品在各类场景下的兼容性以及满足更高的合规等方面的要求，mPaaS 提供蓝盾能力作为无线保镖能力的替代方案，以支持无线保镖无法满足的场景。

现状

目前 mPaaS 已在 Android 10.2.3.23 及以上的基线版本中完成支持无线保镖切换蓝盾的适配和测试工作，使用 10.1.68 基线或更早的基线请升级到 10.2.3 最新版本。

升级基线

将基线版本升级到 **10.2.3.23** 及以上。

当前基线为 10.1.68 主基线

请先参考 mPaaS 10.2.3 升级指南 升级到 10.2.3 最新基线并进行相关适配。

当前基线为定制基线

如果您使用的是定制基线，请搜索群号 41708565 加入钉钉群或提交工单咨询对应的售后及技术支持人员，是否可切换到 10.2.3 基线。

升级工具链&切换蓝盾

安装 Android Studio Flamingo | 2022.2.1 及以上版本和 mPaaS 插件 3.0.230609 及以上。

移除无线保镖组件

在 `app module` 的 `build.gradle` 中通过 `gradle exclude` 移除 `securityguard-build` 依赖库。

```
configurations.all {
    exclude group: 'com.alipay.android.phone.thirdparty', module:
            'securityguard-build'
}
```

添加蓝盾组件

添加蓝盾组件 SDK 依赖。

```
implementation 'com.mpaas.android:blueshield'//蓝盾SDK依赖
```

添加日志组件 SDK 依赖。

```
implementation 'com.mpaas.android:logging'//日志组件SDK依赖
```

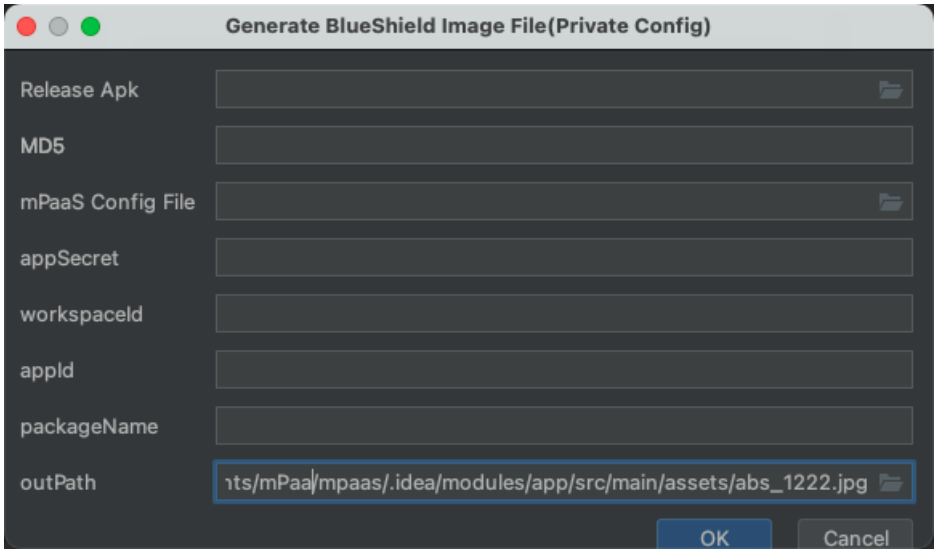
升级 easyconfig 插件依赖。

```
classpath 'com.android.boost.easyconfig:easyconfig:2.8.0'
```

生成蓝盾图片

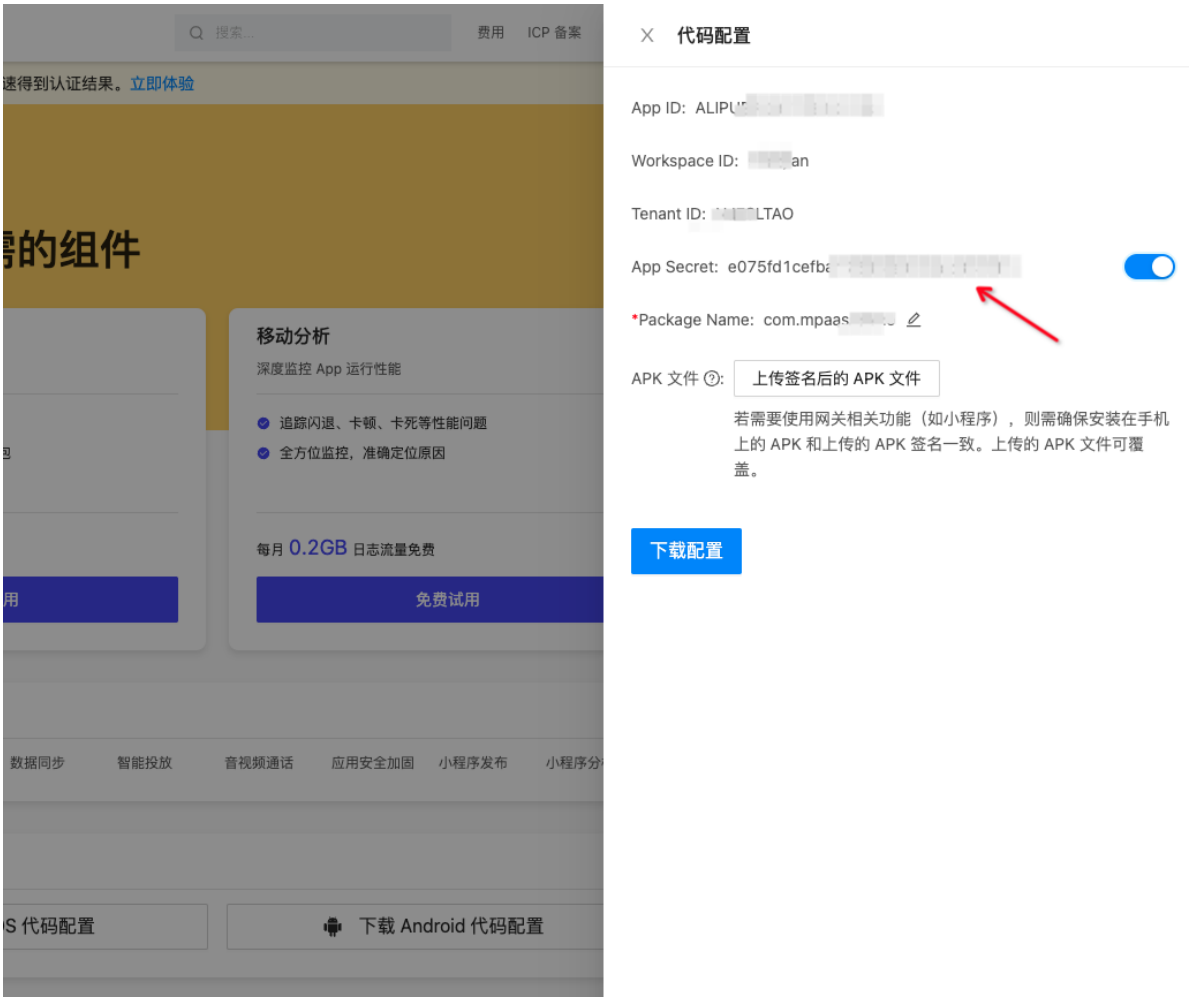
> **说明**
>
> 如果您从 mPaaS 控制台下载的 `.config` 文件中的 `absBase64Code` 值为空，则需进行下面 **生成蓝盾图片** 的操作（适用于私有云场景）。

按下图步骤填写相关信息进行操作即可生成蓝盾图片：



上图中的重点输入项说明：

- **Release Apk：**接入 mPaaS 的工程打包出的 release apk 包，需要进行签名。
- **MD5:** release apk 包上传之后会自动获取填入，即 apk 包的 `public md5 key` 。
- **mPaaS config File:** mPaaS 控制台点击下载配置即可下 `.config` 文件并传入。
- **appSecret:** mPaaS 控制台查看，如下图位置。

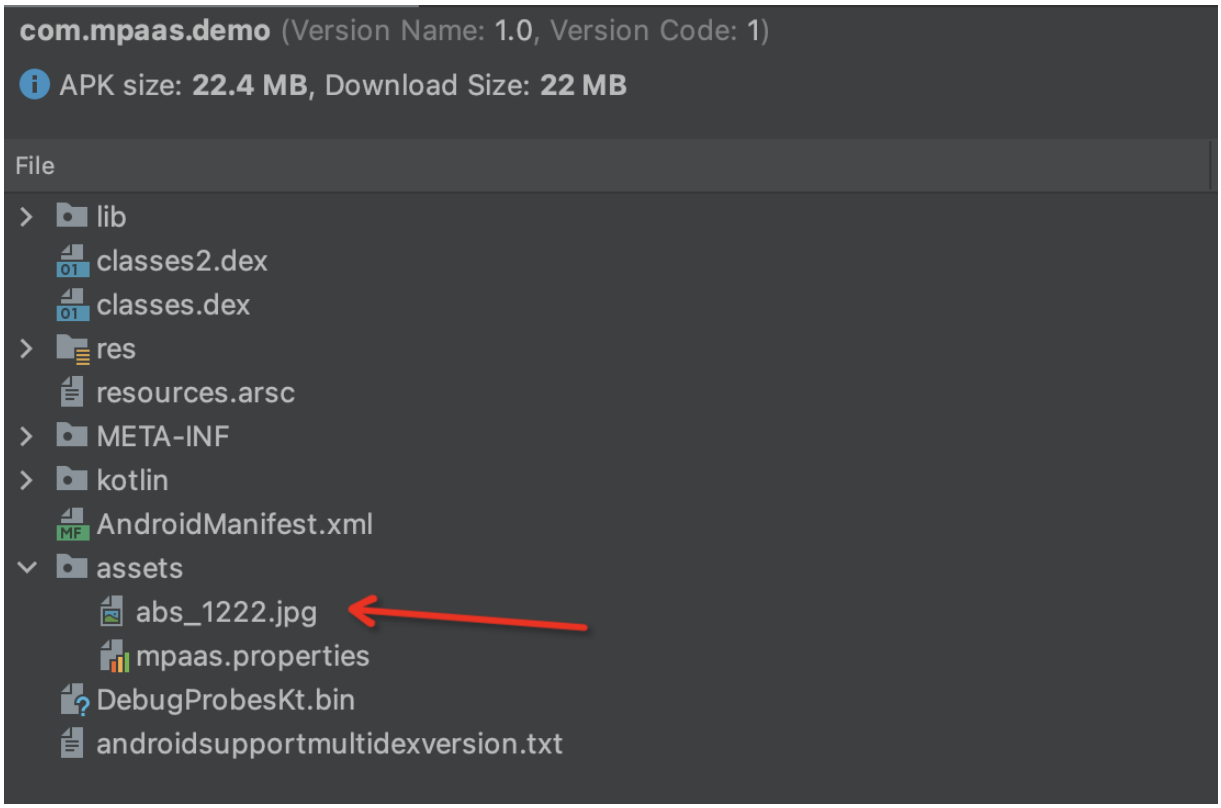


- 其他项 **appId**、**packageName**、**outPath** 传入以上信息后会自动识别填入。

最后将生成的图片添加到工程的 `assets` 目录下。

检查蓝盾图片是否配置成功

将 apk 包拖到 Android Studio 中看 apk 的 `assets` 目录里面是否有 **`abs_1222.jpg`** ，如有，则蓝盾图片配置成功。



配置切换蓝盾

在 `AndroidManifest.xml` 文件中添加 `meta-data` 。

```
<!--value值说明：antGroup是蓝盾-->
        <meta-data
            android:name="mpaas_security_mode"
            android:value="antGroup"/>
```

> **说明**
>
> `mpaas_security_mode` 是 RPC 加签使用的工具的选项。

支持蓝盾更新的库清单

- 移动网关
- 移动调度中心
- 数据同步
- 多媒体
- 小程序

- 定位服务
- 统一存储
- 部分内部依赖组件
- 蚂蚁动态卡片

测试验证范围

在完成切换蓝盾后，根据上述变更清单对 App 进行回归测试。

1.6. 步骤 5 选择合适的基线

基线是指一系列稳定功能的版本集合，是进一步开发的基础。而 mPaaS 产品是基于支付宝的某个特定版本开发的，因此对于 mPaaS 而言，基线则是所基于版本的 SDK 的集合。随着 mPaaS 产品的不断升级，已经提供了多个版本的基线。截止到目前，mPaaS 提供了 10.2.3、10.1.68、10.1.60、10.1.32（已停止维护） 四个基线版本。为保证您获得更丰富的功能和更低的迁移成本，建议您优先选用 10.2.3 版本。

关于基线详细介绍，请参见 基线简介。

操作步骤

1. 在 Android Studio 中打开已有工程，单击 **mPaaS** > **原生 AAR 接入**，打开接入面板。
2. 单击 **开始配置**。
3. 在基线选择窗口中，通过下拉菜单选择合适的基线，单击 **OK**。

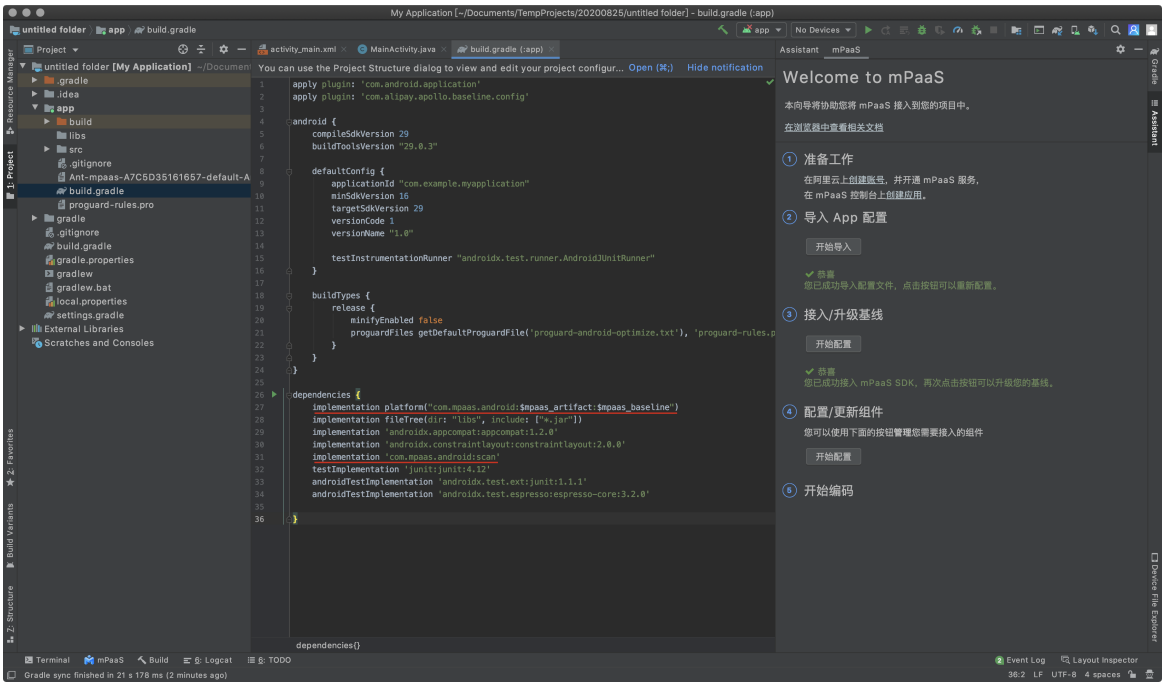
1.7. 步骤 6 添加需要的组件至项目中

本文档基于 **原生 AAR 方式** 使用 Android Studio mPaaS 插件，以扫一扫为例介绍添加 mPaaS 组件到项目中的流程。如果您使用 **组件化方式（Portal&Bundle）**，请参考 组件化接入流程。

操作步骤

1. 单击 **mPaaS** > **开始配置**。
2. 在弹出的窗口中，选择目标 Module，勾选 **扫码**。
3. 单击 **OK**。mPaaS 插件会自动进行部署，稍等片刻单击后，对应 Module 会添加对应的组件。至此已经将 mPaaS 组件添加到 Module 中，可在此 Module 中调用该组件的能力。 mPaaS 会在您的指定的 Module 的 `build.gradle` 中添加以下内容：
   - 基线依赖信息

- **组件依赖信息**


My Application [~/Documents/TempProjects/20200825/untitled folder] - build.gradle (:app)
You can use the Project Structure dialog to view and edit your project configur... Open (⌘;) Hide notification
apply plugin: 'com.android.application'
apply plugin: 'com.alipay.apollo.baseline.config'
android {
compileSdkVersion 29
buildToolsVersion "29.0.3"
defaultConfig {
applicationId "com.example.myapplication"
minSdkVersion 16
targetSdkVersion 29
versionCode 1
versionName "1.0"
testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
}
buildTypes {
release {
minifyEnabled false
proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.p
}
}
}
dependencies {
implementation platform("com.mpaas.android:$mpaas_artifact:$mpaas_baseline")
implementation fileTree(dir: "libs", include: ["*.jar"])
implementation 'androidx.appcompat:appcompat:1.2.0'
implementation 'androidx.constraintlayout:constraintlayout:2.0.0'
implementation 'com.mpaas.android:scan'
testImplementation 'junit:junit:4.12'
androidTestImplementation 'androidx.test.ext:junit:1.1.1'
androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
Welcome to mPaaS
本向导将协助您将 mPaaS 接入到您的项目中。
在浏览器中查看相关文档
1 准备工作
在阿里云上创建账号，并开通 mPaaS 服务，
在 mPaaS 控制台上创建应用。
2 导入 App 配置
开始导入
恭喜
您已成功导入配置文件，点击按钮可以重新配置。
3 接入/升级基线
开始配置
恭喜
您已成功接入 mPaaS SDK，再次点击按钮可以升级您的基线。
4 配置/更新组件
您可以使用下面的按钮管理您需要接入的组件
开始配置
5 开始编码
Gradle sync finished in 21 s 178 ms (2 minutes ago)

2.选择接入方式

2.1. 接入方式简介

接入移动开发平台 mPaaS 有 原生 AAR 方式 和 组件化方式（Portal&Bundle）两种接入方式。本文将向您介绍这两种接入方式，并给出关于选择接入方式的建议。

原生 AAR 方式

原生 AAR 接入方式 是指采用原生 Android AAR 打包方案，更贴近 Android 开发者的技术栈。开发者无需了解 mPaaS 相关的打包知识，通过 mPaaS Android Studio 插件，或者直接通过 Maven 的 pom 和 bom，即可将 mPaaS 集成到开发者的项目中来。该方式降低了开发者的接入成本，能够让开发者更轻松地使用 mPaaS，适合对 **组件化（Portal&Bundle）接入方式** 没有特别需求，想快速使用 mPaaS 能力的客户。

> **说明**
>
> 原生 AAR 接入方式从 10.1.68 起开始支持。

组件化方式（Portal&Bundle）

组件化方式 是指 mPaaS 基于 OSGi（Open Service Gateway Initiative，开放服务网关倡议）技术将一个 App 划分成业务独立的一个或多个 Bundle 工程以及一个 Portal 工程的框架。mPaaS 会对每个 Bundle 工程的生命周期和依赖加以管理，使用 Portal 工程把所有的 Bundle 工程包合并成一个可运行的 `.apk` 包。该方式适用于大型多人并行开发项目。使用 **组件化方式**，需要引入 mPaaS gradle 打包工具，对 gradle 版本以及 `com.android.tools.build:gradle` 版本有一定的要求。

如何选择接入方式

如果使用 mPaaS 需要像使用其他 SDK 一样简单接入并使用，推荐使用 **原生 AAR 方式**。如果使用 mPaaS 来重构您的项目需要引入大规模并行研发的理念，推荐使用 **组件化方式** 。

接入方式对比

	原生 AAR 接入	组件化接入
来源	Google 官方接入方式。	源自支付宝。
打包速度	两者之中打包最慢，和原生接入一模一样。	打包速度快，打包时间分散。
项目组成方式	App module 和 library module。	Portal（一个 App 的壳）和 Bundle（各种业务组件）。
依赖 Gradle 版本	可以升级到官方最新版本，目前是 Gradle 7.x。	4.4/6.3。开发者不可擅自升级。
依赖 AGP[1] 工具链	可以升级到官方最新版本，目前是 AGP 7.0.3。	AGP 3.0.1/3.5.x（开发者不可擅自升级）。

Android Support Library	可使用。	必须使用 mPaaS 提供的版本（23），开发者不可擅自升级。
Android X	完整支持[2]	不支持
databinding	完整支持	v1
kotlin	完整支持	尽量不要使用

> **说明**
> 1. Android Gradle Plugin，Android 打包用的 gradle 插件。
> 2. 借助 `android.enableJetifier=true` 和 `android.useAndroidX=true` 。

2.2. 原生 AAR 方式

2.2.1. 管理组件依赖（原生 AAR）

本文将向您介绍原生 AAR 接入方式下进行组件管理的操作流程。

前置条件

您已完成基线升级。

操作步骤

1. 在 IDE 中点击 **mPaaS** > **原生 AAR 接入**，在弹出的接入面板中，点击配置/更新组件下的 **开始配置**。
2. 在弹出的管理窗口中，点击 **mPaaS 组件管理**，选择要进行管理的 module，勾选要添加的组件，点击 **OK**。如果您的工程中有多个 module，您可以在选择不同的 module 后，分别为其添加组件。
3. 组件添加完成后，点击 **OK**。

2.2.2. 检查构建脚本配置

本文介绍了在添加组件后、编写代码前需要对构建脚本配置进行的检查项。

操作步骤

1. 检查根目录下的 `build.gradle` 配置。
   i. 是否已经引入了 easyconfig。

   ```
   classpath 'com.android.boost.easyconfig:easyconfig:?'
   ```

   ii. 是否已经指定了基线版本。

   ```
   ext.mpaas_artifact = "mpaas-baseline"
   ext.mpaas_baseline = "10.1.68-41"
   ```

2. 检查 App 目录下的配置，确认已经应用 easyconfig 插件。

```
apply plugin: 'com.alipay.apollo.baseline.config'
```

3. 检查使用的 Android Gradle Plugin 版本。

   在项目里查找 `com.android.tools.build:gradle` ，即可获知 Android Gradle Plugin 版本。

   - 如果您使用的是 4.0 以下版本的 Android Gradle Plugin，无需特别配置。
   - 如果您使用的是 4.0 ~ 7.0 的 Android Gradle Plugin ，请在 `gradle.properties` 文件中添加 `android.enableResourceOptimizations=false` 。并在 App 工程中的 `build.gradle` 的签名配置中，显式增加一行 `v1SigningEnabled true` 。该部分整体样例如下：

     ```
     android {
         ...
         signingConfigs {
             release {
                 ...
                 v1SigningEnabled true
             }
             debug {
                 ...
                 v1SigningEnabled true
             }
         }
     }
     ```

   - 如果您使用的是 Android Gradle Plugin 7.0 及以上的版本，还需要在根目录的 `build.gradle` 中将 easyconfig 升级到 2.8.0 版本。

     ```
     classpath 'com.android.boost.easyconfig:easyconfig:2.8.0'
     ```

2.2.3. 初始化 mPaaS

在使用 mPaaS 框架前需要进行一些初始化操作对 Application 对象进行改造。由于是否使用热修复功能后采取不同的初始化内容，因此本文将根据是否使用热修复功能分别向您介绍相应的初始化操作。

不使用热修复功能

不使用 热修复 功能时，只需在 Application 中添加如下代码：

```
@Override
    protected void attachBaseContext(Context base) {
        super.attachBaseContext(base);
        QuinoxlessFramework.setup(this, new IInitCallback() {
            @Override
            public void onPostInit() {
                // 在这里开始使用 mPaaS 功能
            }
        });
    }

    @Override
    public void onCreate() {
        super.onCreate();
        QuinoxlessFramework.init();
    }
```

使用热修复功能

使用 热修复 功能时，需要完成以下操作。

操作步骤

1. 使 Application 对象重新继承 `QuinoxlessApplicationLike` ，并注意将此类防混淆。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() {
      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>
```

3. 引入 Apache HTTP 客户端。

   在使用 RPC 或者热修复功能的时候，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"/>
```

2.2.4. 添加混淆规则

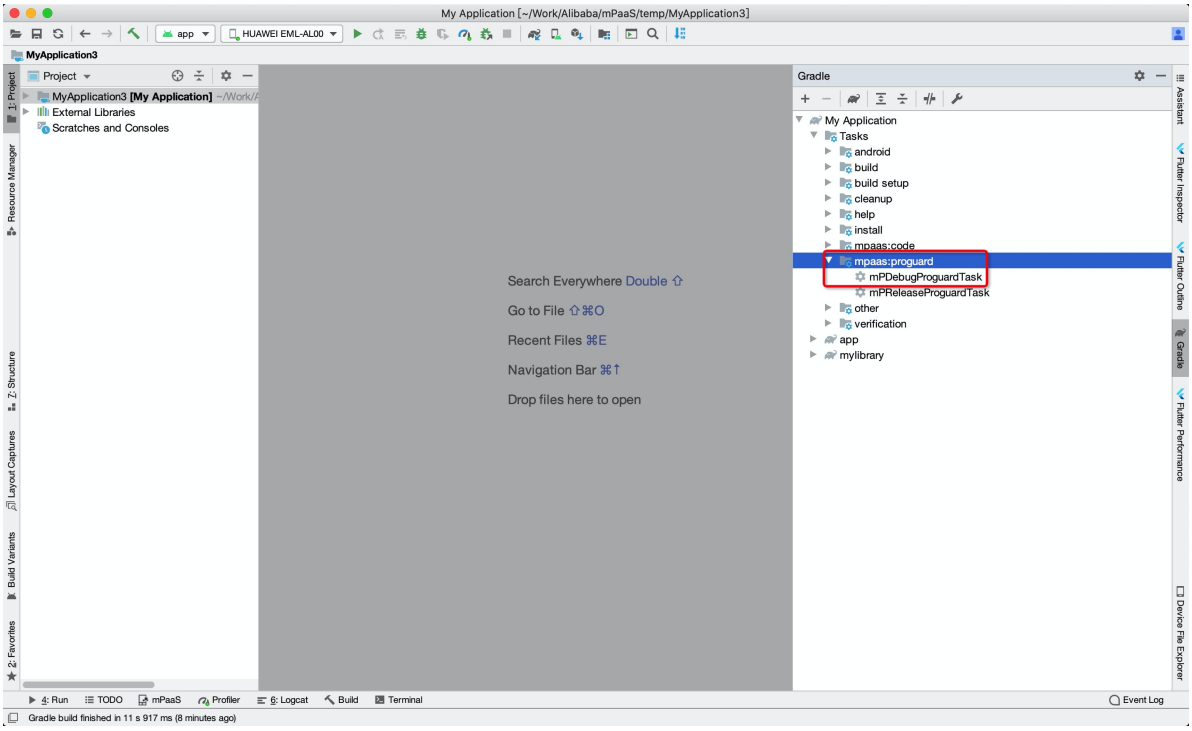
mPaaS Android 客户端开发的应用程序是通过 Java 代码编写而成，而 Java 代码易被反编码，因此为了保护 Java 源代码，需要使用 ProGuard 混淆 Android 文件。本文介绍了在原生 AAR 接入方式下添加混淆规则的流程。

操作步骤

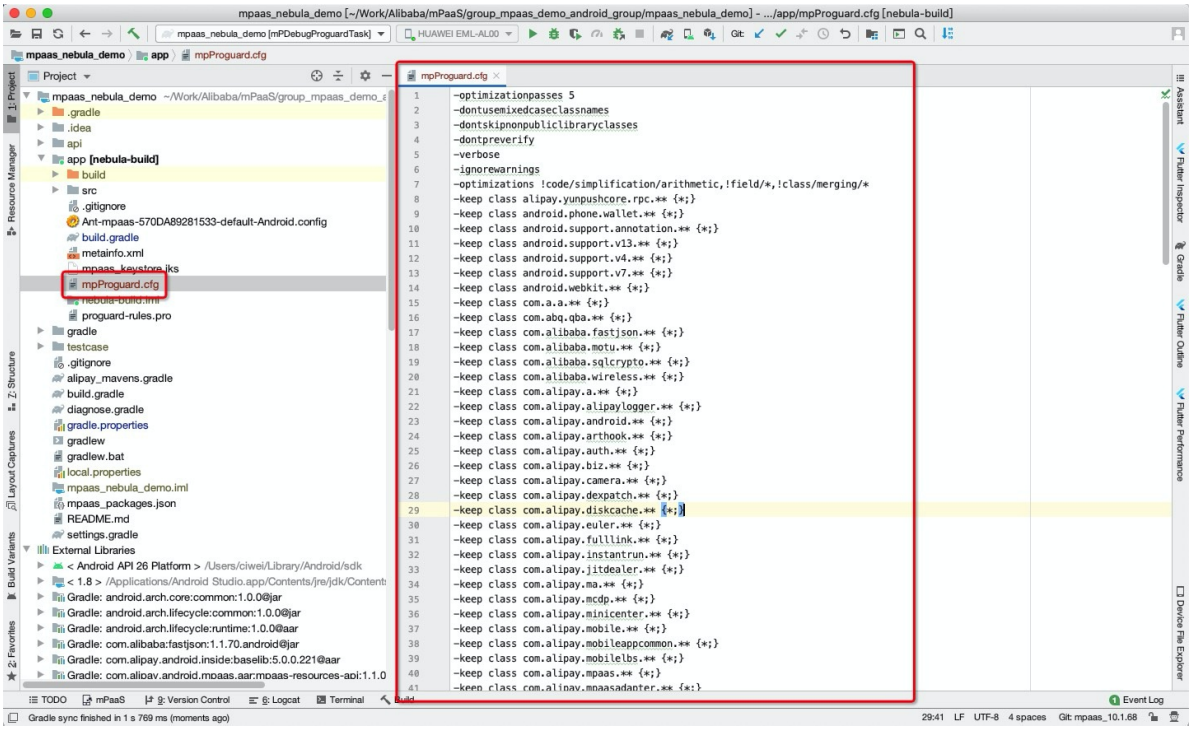
1. 将 `yw_1222.jpg` 自定义为要保留的资源。 在项目中创建一个包含 `<resources>` 标记的 XML 文件，并在 `tools:keep` 属性中指定 `yw_1222.jpg` 为要保留资源。如有需要，还可以在 `tools:discard` 属性中指定每个要舍弃的资源。这两个属性都接受以逗号分隔的资源名称列表。可以将星号（*）字符用作通配符。

```
<?xml version="1.0" encoding="utf-8"?>
<resources xmlns:tools="http://schemas.android.com/tools"
 tools:keep="@drawable/yw_1222"/>
```

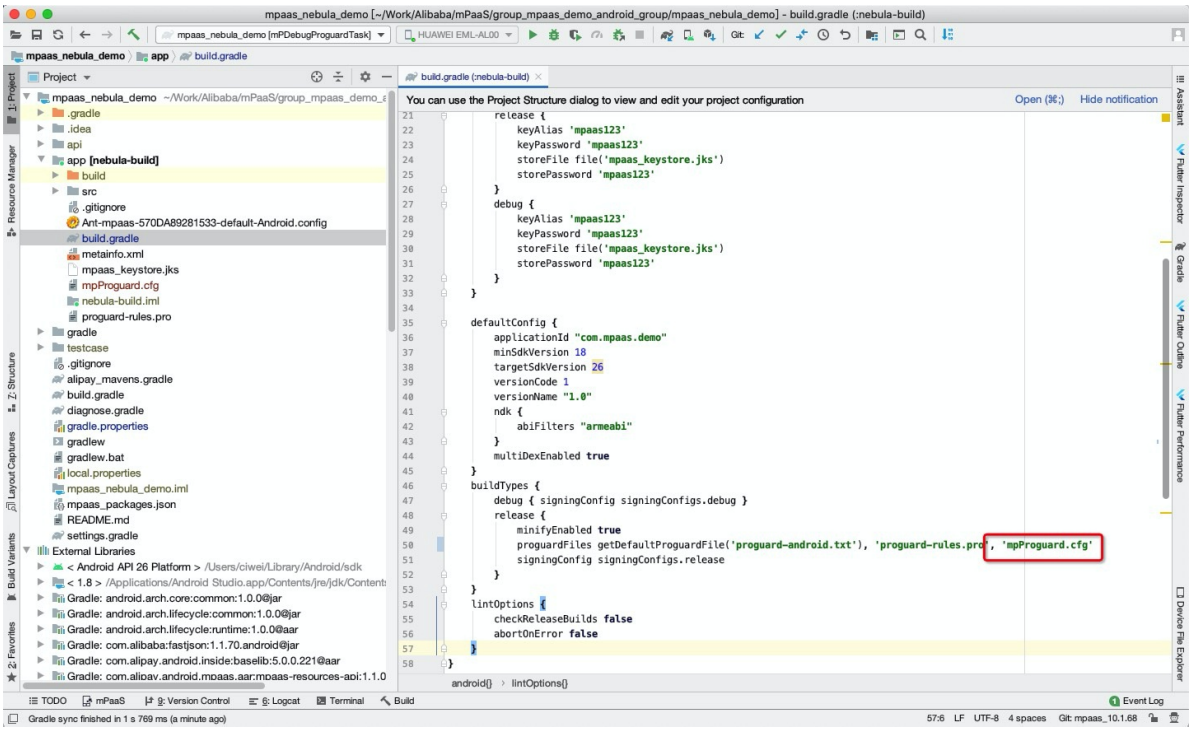
2. 执行任务生成混淆文件。 点击 **mPDebugProguardTask**（或 **mPReleaseProguardTask**）。



3. 执行完成后，项目中会增加混淆文件，如下图所示。

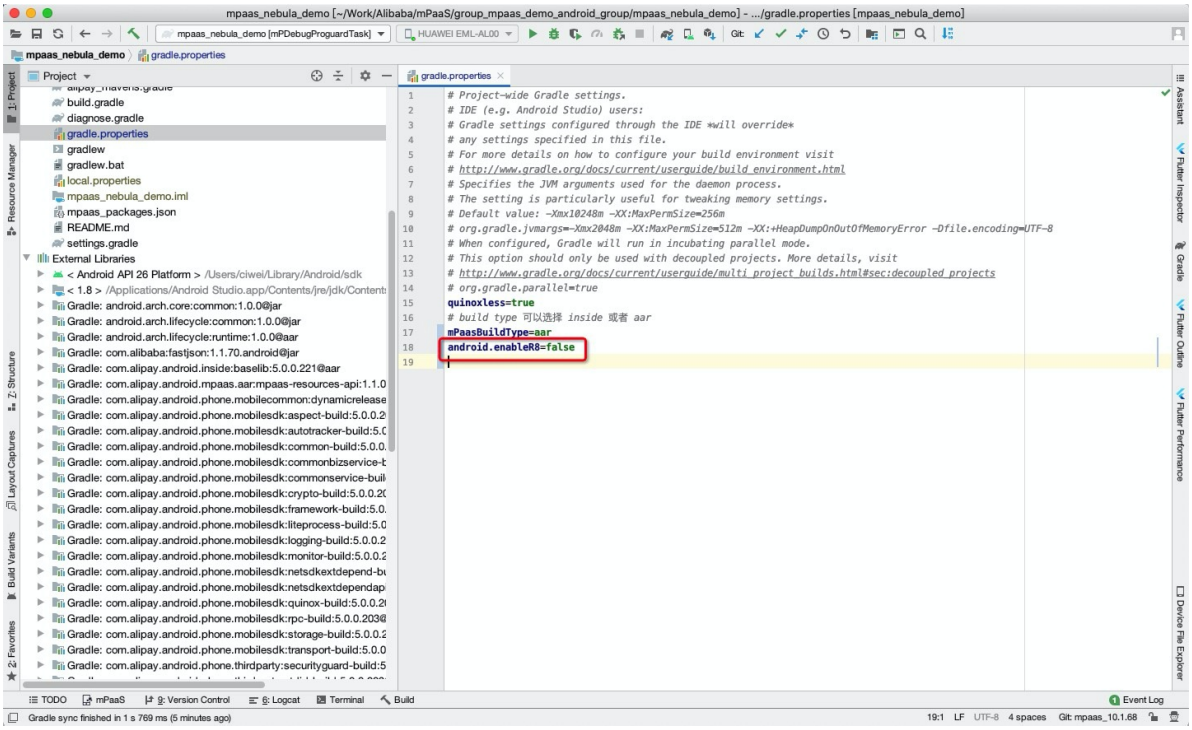


4. 将生成的混淆文件追加到混淆策略中。



如果您混淆过程中遇到 `transformClassesAndResourcesWithR8ForRelease` 卡住，建议您关闭 R8 后再进行混淆。关闭 R8 的方法如下：

在 `gradle.properties` 中添加 `android.enableR8=false` 。



2.2.5. 升级组件化/mPaaS Inside 接入方式到 AAR 接入方式

AAR 接入方式 是指基本采用原生接入的方案。在采用 AAR 接入方式时，由于对 mPaaS 基线管理的需要，请使用最新稳定版的 Android Gradle Plugin 和 Gradle Wrapper。建议使用 Android Gradle Plugin 3.5.3 和 Gradle Wrapper 5.6 以上版本，目前稳定版是 Android Gradle Plugin 3.6.x 和 Gradle Wrapper 6.3。

准备工作

1. 升级 easyconfig 插件到 2.8.0 版本。
2. 升级 Gradle 到 5.0 以上。

升级组件化接入方式到 AAR 接入方式

插件变化

- 更改 Gradle Wrapper 和 Android Gradle Plugin 到您需要的版本。Gradle 应为 5.0 或以上版本。
- 在所有项目的根目录的 build.gradle 文件中，移除 `classpath 'com.alipay.android:android-gradle-plugin'` 。
- 在 bundle 中移除所有的 `com.android.application` 插件，所有的 bundle 使用原生项目里的 `com.android.library` 。
- 移除 bundle 中所有的 `com.alipay.bundle` 插件。
- 移除 bundle 中的 `bundle {}` DSL 定义，移除 `public.xml` 定义（如果有特别需要，可以保留）。
- 移除 portal 中所有的 `com.alipay.portal` 插件。
- 移除 portal 中的 `portal {}` DSL 定义，移除 `public.xml` 定义（如果有特别需要，可以保留）。
- 更改 `apply plugin: 'com.alipay.apollo.baseline.update'` 为 `apply plugin: 'com.alipay.apollo.baseline.config'` 。

依赖变化

- 移除 `dependencies` 节点中所有的 `provided` 和 `bundle` 声明，使用 BOM 方式引入 AAR 依赖。

```
    implementation 'com.mpaas.android:push'
    implementation 'com.mpaas.android:nebula'
    implementation 'com.mpaas.android:push-hms5'
    implementation platform("com.mpaas.android:mpaas-baseline:$(latest)")

    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
```

  `$(latest)` 这个变量是 mPaaS 发布的最新基线。如果是标准基线，则不需要更改 `mpaas-baseline` 的值；否则需要改成对应的定制基线的 `artifact` 。

- 移除加载框架与定制。更多关于加载框架和定制的信息，请参见 加载框架与定制 。

通用组件使用变化

如果使用了 `metainfo.xml` 方式定义通用组件，请参见文档 使用 mPaaS 框架通用组件 。

热修复使用变化

如果使用了热修复功能，则需要将 portal 工程中的 `com.alipay.mobile.quinox.LauncherApplication` 替换为 `com.alipay.mobile.framework.quinoxless.QuinoxlessApplication`，并按照在 AAR 接入方式下使用热修复的流程进行初始化，更多信息，请参见 使用热修复功能。如果没有使用热修复，则替换为 `android.app.Application`。

文件变化

不再需要 `slinks` 文件和 `res_slinks` 文件。

可能存在的问题

因为高版本默认禁用了 v1 签名，可能导致在 v1 签名不存在的情况下无线保镖报错，请参见 如何解决运行时出现的 608 错误或 libsgmain 的 native 错误 进行排查。

自查

请参见 检查构建脚本配置 文档进行自查。

升级 mPaaS Inside 接入方式到 AAR 接入方式

插件变化

- 更改 Gradle Wrapper 和 Android Gradle Plugin 到您需要的版本。Gradle 应为 5.0 或以上版本。
- 在所有项目的根目录的 `build.gradle` 文件中，移除 `classpath 'com.alipay.android:android-gradle-plugin'`。
- 移除 portal 中所有的 `com.alipay.portal` 插件
- 移除 portal 中的 `portal {}` DSL 定义，移除 `public.xml` 定义（如果有特别需要，可以保留）。
- 更改 `apply plugin: 'com.alipay.apollo.baseline.update'` 为 `apply plugin: 'com.alipay.apollo.baseline.config'`。

依赖变化

移除 `dependencies` 节点中所有的 `provided` 和 `bundle` 声明，使用 BOM 方式引入 AAR 依赖。

```
    implementation 'com.mpaas.android:push'
    implementation 'com.mpaas.android:nebula'
    implementation 'com.mpaas.android:push-hms5'
    implementation platform("com.mpaas.android:mpaas-baseline:$(latest)")

    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
```

`$(latest)` 这个变量是 mPaaS 发布的最新基线。如果是标准基线，则不需要更改 `mpaas-baseline` 的值；否则需要改成对应的定制基线的 `artifact`。

通用组件使用变化

如果使用了 `metainfo.xml` 方式定义通用组件，请参见文档 使用 mPaaS 框架通用组件。

gradle.properties 变化

不再需要设定 `quinoxless=true`，已有的 `quinoxless=true` 配置可以保留也可以删除。

可能存在的问题

因为高版本默认禁用了 v1 签名，可能导致在 v1 签名不存在的情况下无线保镖报错，请参见 如何解决运行时出现的 608 错误或 libsgmain 的 native 错误 进行排查。

自查

请参见 检查构建脚本配置 文档进行自查。

使用 AAR 接入

1. 将 mPaaS SDK 添加到项目中。
2. 在各个 Module 中添加需要使用的组件 。

2.2.6. 移除指定的 mPaaS 库

在 `build.gradle` 中使用原生的 gradle 语法——exclude 来移除指定的 mPaaS 库。由于可能存在 mPaaS 多个组件引用同一个库的情况，推荐您使用全局配置的方式来移除。例如，移除 mPaaS SDK 中内置的高德 SDK 时，可以参照如下方式：

```
configurations {
    all*.exclude group:'com.mpaas.group.amap', module: 'amap-build'
    all*.exclude group:'com.alipay.android.phone.thirdparty', module: 'amap3dmap-build'
}
```

2.2.7. 隐私权限弹框的使用说明

监管部门要求在用户点击隐私协议弹框中 同意 按钮之前，App 不可以调用相关敏感 API。为应对此监管要求，mPaaS Android 10.1.68 基线的全部版本、10.1.60 基线的 10.1.60.5 及以上版本和 10.1.32 基线的 10.1.32.16 及以上版本对此要求进行了支持。如果您的工程接入了 框架 组件，请参考本文档对工程进行改造。

使用说明

您需要在应用中弹出隐私权限弹窗，并在用户点击同意之后调用框架的接口发送 同意 的广播，框架收到广播后会完成初始化，还会在 `sharedpreference` 中记录用户同意的行为，初始化完成时通过回调的方式通知您。您只有收到回调后，才能正常使用 mPaaS 各组件的能力。

操作步骤

> **重要**
>
> 弹出隐私弹框的 Activity 不可以继承 mPaaS 的 BaseActivity，因为 BaseActivity 会进行埋点数据采集，会导致 App 在同意隐私政策之前采集隐私数据。

1. 在 `meta-data` 中配置隐私权限弹框的开关。该配置的默认状态是关闭。

```
    <meta-data
            android:name="privacy_switcher"
            android:value="true"></meta-data>
```

2. 使用以下接口，发送 同意 的广播。

> **说明**
>
> 只有在单击 同意 后才发送广播。

```
QuinoxlessPrivacyUtil.sendPrivacyAgreedBroadcast(Context context);
```

3. 用户是否已经同意隐私权限的使用。

> **说明**
> 调用此 API 时，请先初始化 mPaaS 框架。

```
QuinoxlessPrivacyUtil.isUserAgreed(Context context);
```

4. 更新用户同意使用隐私权限的标记，可以方便您在特定的场景下再次弹窗。

```
QuinoxlessPrivacyUtil.setUserAgreedState(Context context, **boolean **agreed);
```

5. 框架初始化完成的回调。
   - 使用 `QuinoxApplication`：需要在 `onMPaaSFrameworkInitFinished` 之后使用 mPaaS 的能力。

     > **说明**
     > 如果您需要使用热修复功能则必须使用 `QuinoxApplication`。

   - 未使用 `QuinoxApplication`：需要在 `IInitCallback` 的 `onPostInit` 之后使用 mPaaS 的能力。

```
QuinoxlessFramework.setup(this, new IInitCallback()
{
@Override
  public void onPostInit()
{

}
});
```

6. 如果您使用的是 10.1.68.42 及以上版本基线且需要清除隐私状态，请在以上所有调用之前调用 `QuinoxlessPrivacyUtil.clearPrivacyState(context);`。

2.2.8. 使用 mPaaS 框架通用组件（非必需）

本文档是在组件化接入方式转为原生 AAR 接入方式的情况下，解决原有的 mPaaS 框架通用组件的适配问题。若未使用 mPaaS 框架通用组件，可以跳过阅读本文档。

为了兼容组件化（路由）的方案，在 10.1.60 及以上基线支持使用 apt 的方式来使用以下四个组件：

- ActivityApplication（Application）
- ExternalService（Service）
- BroadcastReceiver
- Pipeline

> **说明** 这四个组件的使用方式与在组件化接入中的使用方式相同，可点击上方组件名查看详情。

使用组件

1. 在 library 和 Application 工程中引入相关的依赖。

```
implementation 'com.mpaas.mobile:metainfo-annotations:1.3.4'
kapt 'com.mpaas.mobile:metainfo-compiler:1.3.4' // kotlin 的 apt 接入方式
annotationProcessor  'com.mpaas.mobile:metainfo-compiler:1.3.4' // java 的 apt 接入方式
```

2. 定义上述四个组件时分别加上特定的 Annotation。Annotation 有如下四种：
   - @Application
   - @Service
   - @BroadcastReceiver
   - @Pipeline

   Annotation 中的参数与 `metainfo.xml` 定义的相同。例如，使用 `@Application` 时，只需要采用如下的方式：

```
@Application(appId = "123123")
public class MicroApplication extends ActivityApplication {
}
```

不使用 library module

若不使用 library module，只需要在 `APP Module` 工程中的任意一个类加上 `@MetaInfoApplication` 即可。如果您搭配使用了 easyconfig 插件（通常会搭配使用），那么还需要开启一个开关。示例如下：

```
@MetaInfoApplication(compatibleWithPlugin=true)
```

使用 library module

若在 `library module` 工程中定义了上述四个组件，则需要执行以下操作：

1. 在任意类中声明一个 `@MetaInfoLibrary`，其中的参数传入 library module 的 packageName。例如：

```
@MetaInfoLibrary(applicationId=BuildConfig.APPLICATION_ID)
```

2. 在 `app module` 工程中的任意一个类加上 `@MetaInfoApplication`，依赖传入 `library module` 中生成的 `MetaInfoConfig.java`。例如：

```
@MetaInfoApplication(dependencies={com.mylibrary.MetaInfoConfig.class})
```

   如果您搭配使用了 easyconfig 插件（通常会搭配使用），那么还需要开启一个开关。整合了开启开关的示例如下：

```
@MetaInfoApplication(dependencies={com.mylibrary.MetaInfoConfig.class},
compatibleWithPlugin = true)
```

混淆相关

把相关的类加入混淆的白名单，特别是 `com.alipay.mobile.core.impl.MetaInfoConfig`。可以使用以下命令：

```
-keep public class com.alipay.mobile.core.impl.MetaInfoConfig
```

2.3. 组件化接入方式（Portal&Bundle）

2.3.1. 关于 Portal 和 Bundle 工程

组件化框架是指 mPaaS 基于 OSGi（Open Service Gateway Initiative，开放服务网关倡议）技术把一个 App 划分成业务独立的一个或多个 Bundle 工程以及一个 Portal 工程的框架。mPaaS 会对每个 Bundle 工程的生命周期和依赖加以管理，使用 Portal 工程把所有的 Bundle 工程包合并成一个可运行的 `.apk` 包。

mPaaS 框架适合团队协同开发 App，并且该框架包含组件初始化、埋点等功能，方便您轻松接入 mPaaS 组件。

Bundle 工程

传统的原生工程由一个主模块或是一个主 module 和若干个子 module 组成，而一个 mPaaS Bundle 工程一般由一个名为 app 的主 module 和若干个子 module 组成。

例如，在支付宝中，一个 Bundle 一般由一个名为 app 的主 module 和以下三个子 module 组成：

- api：纯代码接口，interface 的定义。
- biz：interface 的实现。
- ui：activity，自定义 view 等。

> **说明**
>
> 至少有一个名为 api 的子 module。如果没有子 module， 就打不出 Bundle 的接口包，并且该 Bundle 不能被其他 Bundle 依赖。

通过阅读本文，您将从以下方面了解 Bundle 工程：

- Bundle 与传统工程区别
- Bundle 属性
- Bundle 接口包
- Bundle 工程包

Bundle 与传统工程区别

Bundle 本质上也是一个原生工程，只是在 **工程**、**主 Module**、**子 Module** 的 `build.gradle` 中多了 mPaaS 的 **Apply 插件**，具体差别体现在以下方面：

- 工程根目录 `build.gradle`
- 主 module 的 `build.gradle`
- 子 module 的 `build.gradle`

工程根目录 build.gradle

在工程根目录的 `build.gradle` 中，增加了对 mPaaS 插件的依赖：

> **说明**
>
> 因功能迭代，插件版本可能会不断增加。

```
classpath 'com.alipay.android:android-gradle-plugin:3.0.0.9.13'
```

```
buildscript {
    repositories {
        mavenLocal()
        jcenter()
        // The Following repository is mPaaS address:
        maven {
            credentials {
                username "[illegible]"
                password "[illegible]"
            }
            url "http://mvn.cloud.alipay.com/nexus/content/repositories/r
        }
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.1.3'
        // Following two one is mPaaS dependencies:
        classpath 'com.alipay.android:android-gradle-plugin:2.1.3.2.7'
        // Ref: https://bitbucket.org/hvisser/android-apt
        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
    }
}
```

主 module 的 build.gradle

在主 module 的 `build.gradle` 中，增加了 mPaaS Bundle **Apply 插件** 的声明，表示该工程为 Bundle 工程，Bundle 配置如下：

```
apply plugin: 'com.alipay.bundle'
```

主 module 的 `build.gradle` 中还增加了以下配置：

```
Launcher ~/xunlong.wxl/mpaas/demo
  .gradle
  .idea
  build
  bundle
    build
    hotpatch_resources
    src
    .gitignore
    build.gradle
    bundle.iml
    proguard-rules.pro
  .gitignore
  build.gradle
  gradle.properties
  Launcher.iml
  local.properties
  settings.gradle
External Libraries

Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly.

        versionCode LAUNCHER_VERSION_CODE as int
        versionName LAUNCHER_VERSION_NAME as String
    }
}

// For build the launcher-bundle.
uploadArchives {
    repositories {
        mavenLocal()
    }
}
// The version of launcher-bundle, which configured in the gradle.pro
version = LAUNCHER_VERSION_NAME
// The group of launcher-bundle, which configured in the gradle.prope
group = LAUNCHER_GROUP

// The customized params used in 'com.alipay.bundle' plugin.
bundle {
    exportPackages ''
    initLevel 11110000
    packageId 36
}
// The customized params used in 'android-apt' plugin.
```

其中：

- `version`：该 Bundle 的 version。
- `group`：该 Bundle 的 groupid。

- `exportPackages`：描述当前 Bundle 工程所有的类在哪些包名下面，包名可以取合集。非静态链接的 Bundle 必须填写 `exportPackages`，否则会出现类加载不到的问题。例如，如果所有的代码在 `com.alipay.demo` 和 `com.alipay.bundle` 下，那么在 `exportPackages` 中就可以写 `com.alipay`，也可以写 `com.alipay.demo`、`com.alipay.bundle`。包名不宜过长或过短。
- `initLevel`：框架启动时加载该 Bundle 的时机。时机范围在 0-100，数字越小表示越早加载。其中 11110000 时为使用时加载，即懒加载。
- `packageId`：描述当前 Bundle 的资源的 ID，供 aapt 打包时需要。由于是多 Bundle 架构，每个 Bundle 的 packageId 必须唯一，不可与其它 Bundle 的 packageId 重复。目前 mPaaS 已经使用的 packageId 如下：

Bundle	packageId
com.alipay.android.phone.thirdparty:androidsupportrecyclerview-build	28
com.alipay.android.phone.mobilesdk:framework-build	30
com.alipay.android.phone.rome:pushservice-build	35
com.alipay.android.phone.sync:syncservice-build	38
com.alipay.android.phone.wallet:nebulabiz-build	41
com.alipay.android.phone.mobilecommon:share-build	42
com.alipay.android.phone.wallet:nebulacore-build	66
com.alipay.android.mpaas:scan-build	72
com.alipay.android.phone.wallet:nebula-build	76
com.alipay.android.phone.securitycommon:aliupgrade-build	77

在 `dependencies` 中会添加对 mPaaS 的如下依赖：

```
dependencies {
    compile project(":api")
    apt 'com.alipay.android.tools:androidannotations:2.7.1@jar'
    //mPaaS dependencies
    provided 'com.alipay.android.phone.thirdparty:fastjson-api:1.1.73@jar'
    provided 'com.alipay.android.phone.thirdparty:androidsupport-api:13.23@jar'
}
```

子 module 的 build.gradle

在子 module 的 `build.gradle` 中，增加了 mPaaS **Apply 插件** 的声明，表示该工程为 Bundle 的子 module 工程，最终会打出该 Bundle 的接口包。

```
apply plugin: 'com.alipay.library'
```

在 `dependencies` 中会添加对 mPaaS 的如下依赖：

```
dependencies {
    apt 'com.alipay.android.tools:androidannotations:2.7.1@jar'
    //mPaaS dependencies
    provided "com.alipay.android.phone.thirdparty:utdid-api:1.0.3@jar"
    provided "com.alipay.android.phone.mobilesdk:framework-api:2.1.1@jar"
}
```

Bundle 属性

本框架的 Bundle 属性设计思路参考 OSGi 的 Bundle，但比 OSGi 的 Bundle 更简洁和轻巧。

以下表格列出 Bundle 属性及其解释：

属性	解释
Bundle-Name	Bundle Name，来自于由 `build.gradle` 文件中的 `group` 和 `settings.gradle` 中定义的 `name` 。
Bundle-Version	Bundle Version，来自于 `build.gradle` 文件中的 `version` 。
Init-Level	Bundle 的加载时机，来自于 `build.gradle` 文件中定义的 properties： `init.level` 。
Package-Id	Bundle 资源的 packageid，来自于 `build.gradle` 文件中定义的 properties。
Contains-Dex	是否包含 dex，编译插件自动判断。
Contains-Res	是否包含资源，编译插件自动判断。
Native-Library	包含的 `so` 文件有哪些，编译插件自动判断。
Component-Name	来自于 `AndroidManifest.xml` 文件中定义的 `Activity` ， `Service` ， `BroadcastReceiver` ， `ContentProvider`
exportPackages	该 Bundle 的所有的类所在的包名，参考主 module 的 `build.gradle`

Bundle 接口包

一个 Bundle 有可能包含多个 **子 Module**，如 biz， api， ui。在编译打包 Bundle 的时候，每个子 module 都会分别生成一个格式为 `.jar` 接口包，其中 api 接口包可以被其他 Bundle 使用。

在打包的同时，也会生成一个 Bundle 工程包，这个工程包包含所有的子 module，工程包可以被 Portal 工程使用，工程包在 Portal 中编译，最后生成 `.apk` 包。

- 由 Bundle 的 **子 module** 打出来的接口包，只对外提供定义的 java/kotlin 接口类，不包含其他如 res 下的资源，且这些接口包仅限于来自名称为 **api** 的 module。
- 各 Bundle 工程直接通过 Bundle 的接口包互相依赖，需要在 bundle 的 `build.gradle` 中的 `dependency` 配置依赖 api 接口。例如，Bundle A 依赖 Bundle B 中的 `bapi` 子 module，那么在 Bundle A 相应的子 module 的 `build.gradle` 中的 `dependency` 配置对 `bapi` 的依赖。

```
provided "com.alipay.android.phone:bundleB:1.0.1:bapi@jar"
```

- 依赖中涉及的 `groupId:artifactid:version:classifier` 分别对应 Bundle 中声明的 group，name，version，子 module 的名字。
- Bundle 的 name 默认为主 module 的文件夹名，可以在 `settings.gradle` 中修改，如下代码所示，其中 app 为主 module 的工程名：

```
include ':api', ':xxxx-build'
project(':xxxx-build').projectDir = new File('app')
```

Bundle 工程包

- 由整个 Bundle 工程打出来的 `.jar` 包其实是一个 `.apk` 格式的文件，但是也以 `.jar` 结尾，如 `framework-build.jar` 。
- 如果要在 Portal 中依赖 Bundle，则在 Portal 主 module 的 `build.gradle` 中的 `dependency` 中声明依赖 Bundle 包，如下所示：

```
dependencies {
    bundle "com.alipay.android.phone.mobilesdk:framework-build:version@jar"
    manifest "com.alipay.android.phone.mobilesdk:framework-
build:version:AndroidManifest@xml"
}
```

- 对 Bundle 打包分两种：debug 包和 release 包，在 Portal 依赖 Bundle 的 debug 包时，需要在 debug 包中额外加上 `:raw` 。
  - 当 Portal 依赖 Bundle 的 debug 包时， `bundle "com.alipay.android.phone.mobilesdk:framework-build:version:raw@jar"`
  - 当 Portal 依赖 Bundle 的 release 包时， `bundle "com.alipay.android.phone.mobilesdk:framework-build:version@jar"`

> **说明**
>
> - 打 Portal 包时，需要确定以下内容：
> - 哪些 Bundle 是要打在 app 的主 dex 中。静态链接，有 `ContentProvider` 的 Bundle 必须放在静态链接中。
> - 哪些动态加载。如果 App 不大，建议都在主 dex 中。
> - 如果想把 Bundle 的代码打进主 dex 中，则需要在 Portal 的 `slinks` 文件中配置当前 Bundle。配置内容为：`groupId-artifactId`。如果以 `-build` 结尾，则去掉 -build。例如，groupId 为 `com.mpaas.group`，artifactId 为 `testBundle-build`，则需要在 `slinks` 文件中添加一行：`com.mpaas.group-testBundle`。
> - 静态链接：把 Bundle 的代码打进 `apk` 的 `classes.dex` 或者 `classes1.dex`，`classes2.dex` 等中，以便工程启动时就可以加载 Bundle 中的类。

Portal 工程

Portal 工程把所有的 Bundle 工程包合并成一个可运行的 `.apk` 包。

Portal 与传统工程区别

Portal 和传统开发中的工程的区别体现在 `build.gradle`：

- 工程根目录 `build.gradle`
- 主 module 目录 `build.gradle`

工程根目录 build.gradle

如下图所示，classpath 多了一个 `com.alipay.android:android-gradle-plugin:2.1.3.2.7` 插件：

> **说明**
>
> 因功能迭代，插件版本可能会不断增加。

```
buildscript {
    repositories {
        mavenLocal()
        jcenter()
        // The Following repository is mPaaS address:
        maven {
            credentials {
                username "[illegible]"
                password "[illegible]"
            }
            url "http://mvn.cloud.alipay.com/nexus/content/repositories/r
        }
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.1.3'
        // Following two one is mPaaS dependencies:
        classpath 'com.alipay.android:android-gradle-plugin:2.1.3.2.7'
        // Ref: https://bitbucket.org/hvisser/android-apt
        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
    }
}
```

该插件中包含 Portal 插件，Portal 插件可以在打包过程中把各 Bundle 合并。

- 合并 Bundle 的 `.jar`
- 合并 Bundle 的 `AndroidManifest`

主 module 目录 build.gradle

多了 mPaaS **Apply Portal 插件** 的声明，表示该工程为 Portal 工程。Portal 配置如下：

```
apply plugin: 'com.alipay.portal'
```

同时，在 `dependencies` 中添加相应的对 Bundle 的依赖。`dependencies` 中的语句是 bundle 和 manifest 的声明，用来表示 Portal 依赖了哪些 Bundle 或 manifest：

```
dependencies {
    compile 'com.android.support:appcompat-v7:25.0.0'
    compile(name: 'SecurityGuardSDK-external-release-5.1.38', ext: 'aar')
    // launcher-bundle
    bundle "com.mpaas.demo.launcher:bundle:1.1-SNAPSHOT:raw@jar"
    manifest "com.mpaas.demo.launcher:bundle:1.1-SNAPSHOT:AndroidManifest@xml"
    //adapter-bundle
    bundle "com.mpaas.mpaasadapter:mpaasadapter-build:1.0.0.1701092045@jar"
    manifest "com.mpaas.mpaasadapter:mpaasadapter-build:1.0.0.1701092045:AndroidManifest@xml"

    // basic-bundle -- the component of the basic framework
    bundle "com.alipay.android.phone.mobilesdk:common-build:1.3.0@jar"
    manifest "com.alipay.android.phone.mobilesdk:common-build:1.3.0:AndroidManifest@xml"
    // quinox -- the component of the basic framework
    bundle "com.alipay.android.phone.mobilesdk:quinox-build:2.2.0.161028154501:nolog@jar"
    manifest "com.alipay.android.phone.mobilesdk:quinox-build:2.2.0.161028154501:AndroidManifest@xml"
    // framework -- the component of the basic framework
    bundle "com.alipay.android.phone.mobilesdk:framework-build:2.2.0.161028154723:nolog@jar"
    manifest "com.alipay.android.phone.mobilesdk:framework-build:2.2.0.161028154723:AndroidManifest@xml"
    // android-support-wrapper-bundle
    bundle "com.alipay.android.phone.thirdparty:androidsupport-build:13.23.2.161114144707:nolog@jar"
    manifest "com.alipay.android.phone.thirdparty:androidsupport-build:13.23.2.161114144707:AndroidManifest@xml"
    // rpc-bundle -- the component of the basic framework
```

> **重要**
>
> - 通常 Portal 下面不写代码。
> - 以下几种在 Bundle 工程中使用的资源（style/drawable/string等），必须放在 Portal 工程中，否则会导致编译/运行时找不到资源：
> - `AndroidManifest.xml` 中使用的资源。
> - 传递给 NotificaionManager 使用的资源。
> - 通过 `getResources().getIdentifier()` 方法使用的资源。
> - 引用的第三方 `AAR` 包中如有以上三种情况，也需要解压 `AAR`，将对应的资源复制一份放到 Portal 工程中。

工程依赖

一个 **基于 mPaaS 框架** 的 App 包括 **一个或多个 Bundle** 和 **一个 Portal**。一个 App 有且只能有一个 Portal 工程，但可以有多个 Bundle 工程。

通过 mPaaS 插件，Portal 工程把所有的 Bundle 工程包合并成一个可运行的 `.apk` 包。合并后，该插件把 Bundle 工程部署至指定的仓库地址。该仓库地址在 Bundle 的主 module 中的 `build.gradle` 中定义，如下代码所示：

```
uploadArchives {
    repositories {
        mavenLocal()
    }
}
```

该仓库地址是上传至本地的 `~/.m2` 仓库地址。您也可以添加自定义的仓库地址，如下所示：

```
mavenDeployer {
    mavenLocal()
    repository(url: "${repository_url}") {
        authentication(userName: 'userName', password: 'userName_pwd')
    }
    snapshotRepository(url: "${repository_url}") {
        authentication(userName: 'userName', password: 'userName_pwd')
    }
}
```

上传之后，Bundle 以 `groupid:artifactid:version:classifier@type` 的形式存在指定的仓库中。因此，只要在 Portal 最外层主 module 的 `build.gradle` 中声明 `dependency` 就可指定各 Bundle 依赖，如下代码所示：

```
dependencies {
    bundle 'com.alipay.android.phone.mobilesdk:quinox-
build:2.2.1.161221190158:nolog@jar'
    manifest 'com.alipay.android.phone.mobilesdk:quinox-
build:2.2.1.161221190158:AndroidManifest@xml'
}
```

此外，Bundle 工程之间的相互依赖也需要在 Bundle 的最外层的 `build.gradle` 中声明仓库依赖地址。

> **重要**
>
> 以下配置中的 `username` 和 `password` 不是控制台的登录用户名和密码。您必须 提交工单 获取这两个值。其中：
>
> - `mavenLocal()` 描述依赖的本地仓库地址。
> - `maven{}` 声明依赖的远程仓库地址。

```
allprojects {
    repositories {
        mavenLocal()
        mavenCentral()
        maven {
            credentials {
                username "{username}"
                password "{password}"
            }
            url "http://mvn.cloud.alipay.com/nexus/content/repositories/releases/"
        }
    }
}
```

Bundle 编译打包结果

使用 mPaaS 插件编译打包后，一个 Bundle 会生成一个工程包（是一个 `.jar` 包）。更多信息，请参考 Bundle 工程包 和 Bundle 接口包。

工程包会以 `groupid:artifactid:version:classifier@type` 的形式发布到指定仓库中。发布仓库地址在 Bundle 主 module 中的 `build.gradle` 中定义，示例如下：

```
uploadArchives {
    repositories {
        mavenLocal()
    }
}
```

上述配置指定发布仓库为本地 Maven 仓库（`mavenLocal`）。如需修改本地 Maven 仓库地址（默认 `~/.m2`）或增加发布仓库，请参见 配置发布仓库。

添加 Bundle 依赖

您可以在 Portal 中添加 Bundle 依赖，也可以在 Bundle 中添加其他 Bundle 依赖。只需：

1. 在 Portal 或 Bundle 最外层的 `build.gradle` 中声明依赖仓库地址。依赖仓库应和上文 Bundle 发布仓库相对应。依赖仓库的配置方法，请参见 配置依赖仓库。
2. 在 Portal 或 Bundle 主 module 的 `build.gradle` 中声明 `dependencies` 依赖。如添加 Bundle（`quinox`）依赖的示例如下：

```
dependencies {
    bundle 'com.alipay.android.phone.mobilesdk:quinox-
build:2.2.1.161221190158:nolog@jar'
    manifest 'com.alipay.android.phone.mobilesdk:quinox-
build:2.2.1.161221190158:AndroidManifest@xml'
}
```

相关链接

- mPaaS 插件
- 配置依赖仓库
- 配置发布仓库

2.3.2. 接入流程

如果采用 **组件化接入方式**，您需要完成以下通用步骤以完成接入流程：

1. 配置开发环境
2. 在控制台创建应用
3. 客户端创建新工程
4. 管理组件依赖
5. 构建

客户端创建新工程

本文将以 Windows 开发环境为例，引导您在本地创建一个全新 App，并编译打包，最终获得可运行的 `.apk` 包。

客户端创建新工程之前您需要先完成以下操作：

1. 配置开发环境
2. 在控制台创建应用

创建 Portal 工程

如果您需要组件化方案，可以使用 Portal Bundle 方式，您首先需要创建一个 Portal 工程。

Portal 一般不包含业务代码，仅仅用于将各 Bundle 合并成一个可运行的 `.apk` 包。因此在创建 Portal 时，默认会创建一个后缀名为 `Launcher` 的 Bundle 工程。

具体创建步骤如下：

1. 启动 Android Studio 后，在欢迎页面点击 **Start a new mPaaS project**。
2. 在 **Create New mPaaS Project** 窗口中，选择 **mPaaS Portal**。点击 **Next**。
3. 填写 **项目名称**，在 **控制台配置文件（JSON）路径选择** 中选择从控制台 **代码管理** > **代码配置** 中下载到的 .config 配置文件，mPaaS 插件会根据选择的配置文件自动解析填写 **Package Name** 。点击 **Next**。
4. 选择 mPaaS SDK 版本，并勾选您需要的模块依赖。点击 **Next** 按钮。

> **重要**
>
> - 请按需勾选模块依赖。具体依赖信息，请参考各组件的接入文档。
> - 您也可以只选择框架必选依赖。在完成应用创建之后，再参考各组件的接入文档，使用 mPaaS 插件 > 组件管理 功能添加所需依赖。

5. 确认默认创建的 Bundle 工程的信息。点击 **Finish** 按钮。

至此，您已完成 Portal 工程的创建，并同时获得一个默认创建的 Bundle 工程。

创建新的 Bundle 工程

mPaaS 框架支持多 Bundle，您可以为您的工程添加多个 Bundle 工程。

1. 点击 **File** > **New** > **Start a New mPaaS Project** 菜单。
2. 在 **Create New mPaaS Project** 窗口中，选择 **mPaaS Portal**。点击 **Next**。
3. 填写 **项目名称**，在 **控制台配置文件（JSON）路径选择** 中选择从控制台 **代码管理** > **代码配置** 中下载到的 .config 配置文件，mPaaS 插件会根据选择的配置文件自动解析填写 **Package Name** 。点击 **Next**。
4. 选择 mPaaS SDK 版本，并勾选您需要的模块依赖。点击 **Next** 按钮。
5. 确认默认创建的 Bundle 工程的信息。点击 **Finish** 按钮。

至此，您已完成 Bundle 工程的创建。关于 Bundle 开发的更多信息，请参见 Bundle 工程。

后续步骤

您可以参考各组件的接入文档，接入并使用 mPaaS 组件。

相关链接

组件化接入方式 > 简介：介绍 Portal 和 Bundle 工程的 **代码结构**、**编译打包结果** 以及 **与原生工程的区别**。

管理组件依赖

为了更便捷地升级 mPaaS SDK 基线和管理组件依赖，您需要先升级 Android Studio mPaaS 插件至最新版本。关于更新 mPaaS 插件的更多信息，请参见 更新 mPaaS 插件。

添加组件依赖

为了使用 mPaaS 组件，您需要分别在 Portal 和 Bundle 工程中添加对应组件的依赖：

- 在 Portal 工程中添加，将确保相应依赖在打包时打入您的 APK。
- 在 Bundle 工程中添加，将确保您能在 Bundle 工程中调用对应组件的 API。
- 对于单 Portal 工程模式，您只需在 Portal 工程中添加。
- 如果您在创建 mPaaS 工程时已选择过需要使用的组件，您仍可以按照下文步骤增删组件。

操作步骤

1. 在 Android Studio 中选择 **mPaaS** > **组件化接入**，在弹出的接入面板中，点击 **配置/更新组件** 下的 **开始配置**。
2. 在弹出的组件管理窗口中，点击按钮安装需要的组件。
   - 对于未安装的组件，相应的按钮会显示“未安装”。点击该按钮会安装该组件。
   - 对于已经安装的组件，按钮会显示“已安装”。此时再点击该按钮将会卸载该组件。

后续步骤

如果您此前未使用过 Android Studio mPaaS 插件管理组件依赖，是您首次使用 **组件管理** 功能添加完组件后，您还需要检查或修改以下配置。

1. 检查 Portal/Bundle 工程根目录 `build.gradle` 文件，确保包含以下依赖且不低于 2.8.0 版本：

```
buildscript {
 ...
 dependencies {
     classpath 'com.android.boost.easyconfig:easyconfig:2.8.0'
 }
}
```

2. 检查 Portal 工程主 module 下的 `build.gradle` 文件，确保包含以下内容：

```
apply plugin: 'com.alipay.portal'
portal {
 allSlinks true
 mergeAssets true
}
apply plugin: 'com.alipay.apollo.baseline.update'
mpaascomponents{
 excludeDependencies=[]
}
```

3. 删除旧依赖：

> **重要**
> 强烈建议您在删除以下内容前先进行备份。

   - 对于 Portal & Bundle 模式，您需要在 Portal 工程主 module 下的 `build.gradle` 文件中删除 dependencies 节点下 mPaaS 组件相关依赖（`mpaas-baseresjar` 除外）。
   - 对于单 Portal 工程模式，您需要在主 module 下的 `build.gradle` 文件中删除以下内容：

```
apply from: rootProject.getRootDir().getAbsolutePath() + "/mpaas_bundles.gradle"
apply from: rootProject.getRootDir().getAbsolutePath() + "/mpaas_apis.gradle"
```

     并删除工程根目录下的 `mpaas_bundles.gradle` 和 `mpaas_apis.gradle` 文件，但需要注意，删除 `mpaas_apis.gradle` 文件可能导致编译失败，您需要按照下文在子 module 中修改配置。

4. 如果您需要在子 module 中调用 mPaaS 组件 API：
   - 对于 Portal & Bundle 模式的工程，您需要在 Bundle 工程子 module 下的 `build.gradle` 文件中添加：

```
apply plugin: 'com.alipay.apollo.baseline.update'
```

   - 对于单 Portal 工程模式，您需要在子 module 下的 `build.gradle` 文件中删除：

```
apply from: rootProject.getRootDir().getAbsolutePath() + "/mpaas_apis.gradle"
```

     并添加：

```
apply plugin: 'com.alipay.apollo.baseline.update'
```

5. 如果旧依赖中有为您定制的库，您还需要 添加定制依赖 。
6. 如果由于库冲突导致编译失败，您可以 解决依赖冲突 。

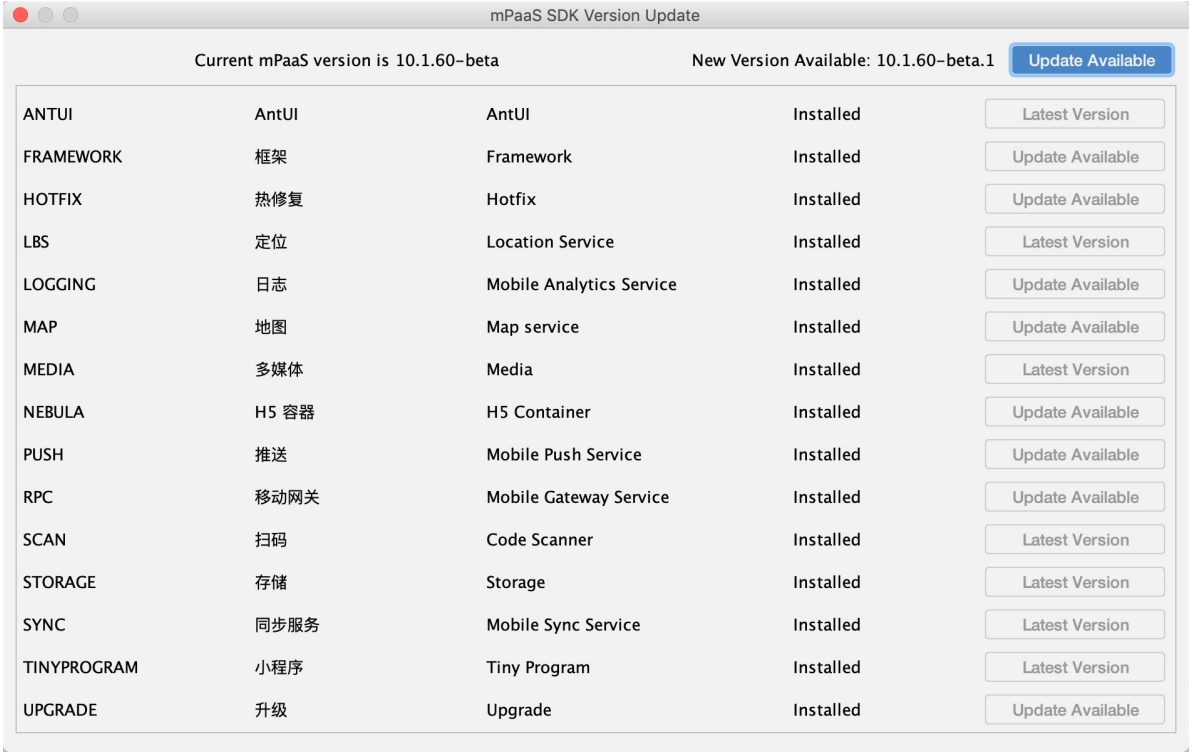
升级基线

1. 在 Android Studio 中点击 **mPaaS** > **组件化接入**，在弹出的接入面板中，点击 **接入/升级基线** 下的 **开始配置**。
2. 点击版本下拉框，选择一个新版本，然后点击 **OK** 按钮，即可升级基线。

升级单个组件

新版

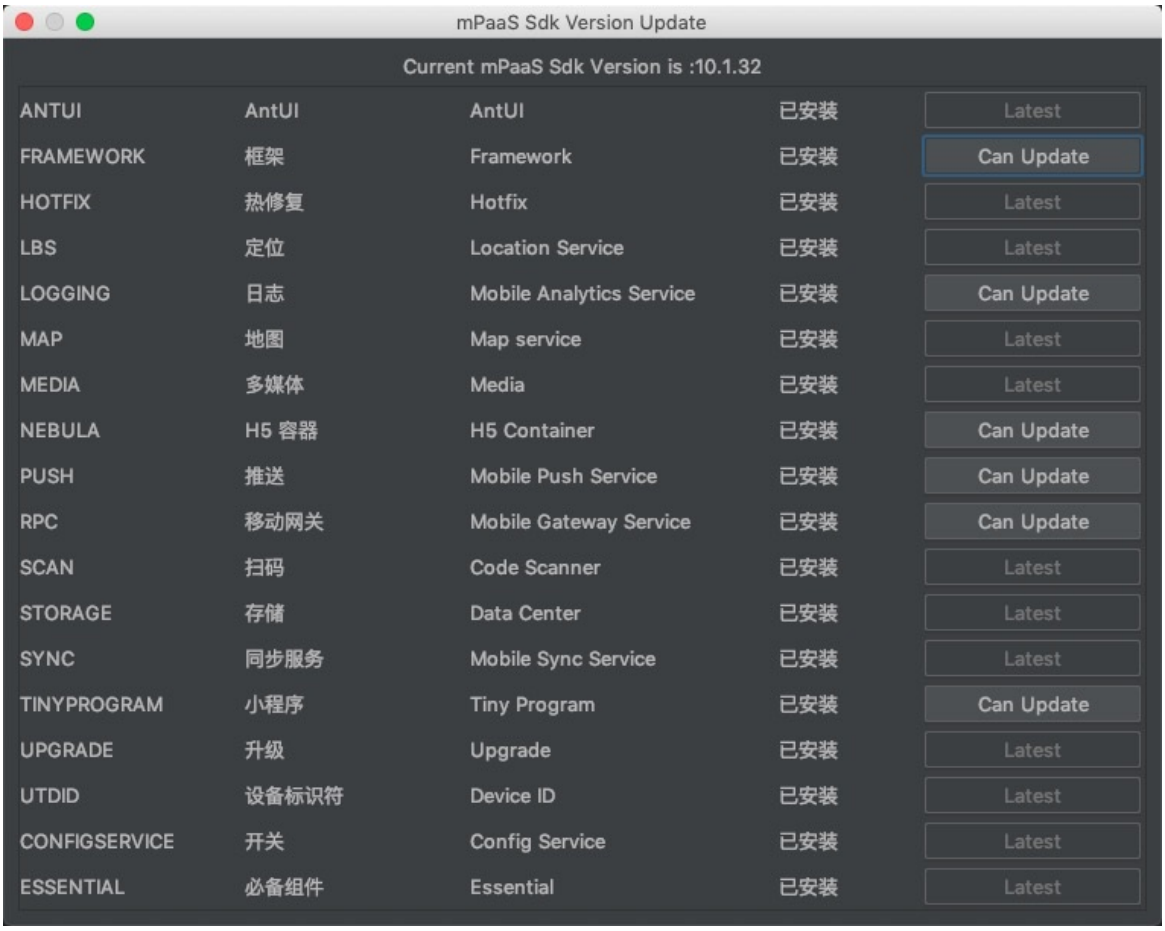
1. 在 Android Studio 中选择 **mPaaS** > **组件升级**，您将看到组件列表。
2. 查看组件状态，进行升级操作，若右上角有提示可更新，那么点击之后就能更新了。



旧版

1. 在 Android Studio 中选择 **mPaaS** > **组件升级**，您将看到组件列表。
2. 查看组件状态，进行升级操作：
   - 若为 **最新版**，则说明该组件无需升级。

- 否则说明该组件有新版本。您可以点击状态按钮，升级该组件。



添加定制依赖

- 如果您首次使用 **组件管理** 管理组件但未升级 SDK，您只需将定制库写在 Portal 工程主 module 下 `build.gradle` 文件中的 dependencies 节点下，例如：

```
bundle 'com.alipay.android.phone.mobilesdk:logging-build:2.0.2.18032216xxxx@jar'
manifest 'com.alipay.android.phone.mobilesdk:logging-
build:2.0.2.18032216xxxx:AndroidManifest@xml'
```

- 如果您首次使用 **组件管理** 管理组件且升级了 SDK，或使用 **基线升级** 升级了 SDK，您的定制库可能需要基于新版本重新定制，请 提交工单 或联系 mPaaS 支持人员确认，重新定制或确认无需重新定制后，您可按照上文添加定制依赖。

构建

单击 **mPaaS** > **构建** 使用 Android Studio mPaaS 插件提供的 **构建** 功能编译工程。

2.3.3. 注册通用组件

模块化是 mPaaS 框架的设计原则之一，业务模块的低耦合与高内聚有利于业务的扩展和维护。

业务模块以 Bundle 的形式存在互不影响，但 Bundle 之间会存在一些关联性，比如跳转到另一个 Bundle 界面，调用另一个 Bundle 中的接口，或者 Bundle 中的一些操作需要在初始化的过程中完成等。

因此，mPaaS 设计了 metainfo 通用组件注册机制，各个 Bundle 将需要注册的组件在 `metainfo.xml` 中声明。

框架目前支持以下组件：

- ActivityApplication（Application）
- ExternalService（Service）
- BroadcastReceiver
- Pipeline

`metainfo.xml` 格式如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<metainfo>
    <broadcastReceiver>
        <className>com.mpaas.demo.broadcastreceiver.TestBroadcastReceiver</className>
        <action>com.mpaas.demo.broadcastreceiver.ACTION_TEST</action>
    </broadcastReceiver>
    <application>
        <className>com.mpaas.demo.activityapplication.MicroAppEntry</className>
        <appId>33330007</appId>
    </application>
</metainfo>
```

Application 组件

ActivityApplication 是 mPaaS 框架设计的组件，起到 Activity 容器的角色。ActivityApplication 组件的主要作用在于管理和组织各个 Activity，专门用于解决跳转到另一个Bundle 界面的问题。因而，调用方只需关心业务方在框架中注册的 ActivityApplication 信息以及约定的参数。

关于此任务

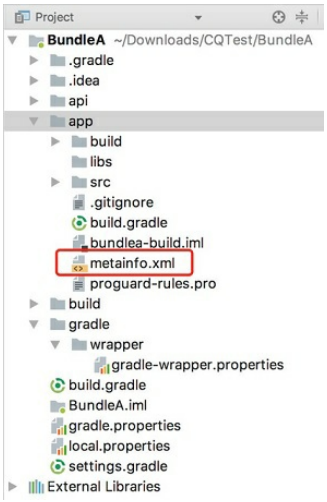
ActivityApplication 的创建、销毁等一系列逻辑完全由 mPaaS 框架来管理。业务方只需要处理其收到的参数并管理自己业务下的 Activity，这样业务方和调用方之间被有效的隔离开来。业务方和调用方只需要协调调用的参数，使得依赖更加轻量。

基于 mPaaS 框架开发的 Android 客户端应用，Activity 须继承自 BaseActivity 或 BaseFragmentActivity，以便能够被 ActivityApplication 类所管理。

> **说明**
>
> 您可以下载代码示例，示例中包含跳转到另一个 Bundle 的 Activity。有关下载地址、使用方法及注意事项，查看 获取代码示例 。

操作步骤

1. 在工程的主 module 中创建 `metainfo.xml` 文件，放在如下图位置：



2. 在 `metainfo.xml` 中写入如下的配置，其中：
   - `className` 配置的类名用于提供跳转的类名称和定义各阶段的行为。具体类的定义，参见步骤 3 的代码。框架通过 `className` 定义的名称加载相应的类，所以该类一定不能被混淆，需要在混淆文件中保留。
   - `appId` ：业务的唯一标识。业务方只需要知道该业务的 appId 就能完成跳转。appId 与 ActivityApplication 的映射由框架层处理。

```
<?xml version="1.0" encoding="UTF-8"?>
<metainfo>
  <application>
      <className>com.mpaas.demo.hotpatch.HotpatchMicroApp</className>
      <appId>33330002</appId>
  </application>
</metainfo>
```

3. 如果 `metainfo` 通过 `className` 指定的类只是完成简单的跳转，使用以下代码实现：

```
 /**
  * 场景一：
  * 若只可能会跳转到某一个Activity界面，那么需要重载getEntryClassName和onRestart，前者返回Activity的classname，后者需要调用getMicroApplicationContext().startActivity(this, getEntryClassName());
  * 场景二：
  * 若需要根据需求跳转到不同的Activity界面那么需要重载onStart和onRestart，根据bundle中的参数跳转到指定的界面
  * Created by mengfei on 2018/7/23.
  */
 public class MicroAppEntry extends ActivityApplication {

     @Override
     public String getEntryClassName() {
         //场景一：只可能跳转到某一个 Activity 在此返回 classname 即可
         //场景二：根据参数跳转到某一界面，需要返回 null
         return MainActivity.class.getName();
     }

     /**
```

```
     * Application被创建时被调用，实现类可以在这里做些初始化的工作
     *
     * @param bundle
     */
    @Override
    protected void onCreate(Bundle bundle) {
        doStartApp(bundle);
    }

    /**
     * 启动Application时被调用
     * 如果Application还没有被创建，会先去执行create方法，然后再执行onStart()回调
     */
    @Override
    protected void onStart() {
    }

    /**
     * 当Application被销毁时，调用此回调
     *
     * @param bundle
     */
    @Override
    protected void onDestroy(Bundle bundle) {

    }

    /**
     * 启动Application时，如果Application已经被start过了，则不调用onStart()而是调用onRestar
t()回调
     *
     * @param bundle
     */
    @Override
    protected void onRestart(Bundle bundle) {
    //针对场景一：需要在此调用getMicroApplicationContext().startActivity(this,
getEntryClassName());
        doStartApp(bundle);
    }

    /**
     * 当一个新的Application被start时，当前的Application将被暂停，此方法被回调
     */
    @Override
    protected void onStop() {

    }

    private void doStartApp(Bundle bundle) {
        String dest = bundle.getString("dest");
        if ("main".equals(dest)) {
            Context ctx =
LauncherApplicationAgent.getInstance().getApplicationContext();
            ctx.startActivity(new Intent(ctx, MainActivity.class));
```

```
            } else if ("second".equals(dest)) {
                Context ctx =
LauncherApplicationAgent.getInstance().getApplicationContext();
                ctx.startActivity(new Intent(ctx, SecondActivity.class));
            }
        }
    }
```

4. 作为调用者，您需要通过框架封装的 `MicroApplicationContext` 中提供的接口进行跳转。`curId` 参数也可以传 null：

```
// 获取 MicroApplicationContext 对象：
MicroApplicationContext context = MPFramework.getMicroApplicationContext();
String curId = "";
ActivityApplication curApp = context.getTopApplication();
 if (null != curApp) {
     curId = curApp.getAppId();
}
String appId = "目标ApplicationActivity的id";
Bundle bundle = new Bundle(); // 附加参数，也可以不传
context.startApp(curId, appId, bundle);
```

Service 组件

mPaaS 设计了 Service 组件解决跨 Bundle 调用接口。Service 组件用于将一些逻辑以服务的形式提供出来，供其他模块使用。

关于此任务

Service 组件的特点如下：

- 没有用户界面的限制。
- 在设计上，遵循接口与实现分离。

原则上只有接口类对调用者可见，所以接口类应定义在接口 module 中（在生成 Bundle project 时，默认会生成一个接口 module，名字是 api），实现定义在主 module 中。

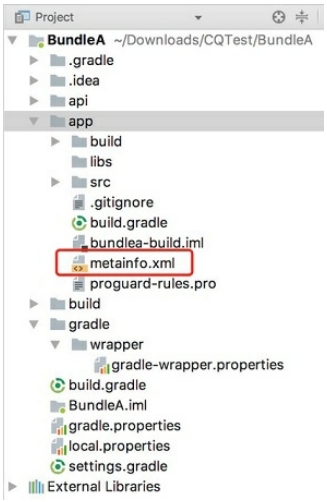
外部调用都通过 `MicroApplicationContext` 的 `findServiceByInterface` 接口，通过 `interfaceName` 获取相应的服务。对于 Bundle 使用来说，只暴露服务抽象接口类，即 interfaceName 中定义的类，抽象接口类会定义在接口包中。

> **说明**
>
> 您可以下载代码示例，示例中包含调用另一个 Bundle 的接口示例。有关下载地址、使用方法及注意事项，查看 获取代码示例 。

操作步骤

通过以下步骤注册 Service 组件：

1. 定义 `metainfo.xml` 位置，如下图所示：



2. 在 `metainfo.xml` 中写入如下的配置。框架将 `interfaceName` 作为 `key` ，`className` 作为 `value` ，记录两者的映射关系。其中，`className` 为具体接口的实现类，`interfaceName` 为抽象接口类：

```
<metainfo>
 <service>
     <className>com.mpaas.cq.bundleb.MyServiceImpl</className>
     <interfaceName>com.mpaas.cq.bundleb.api.MyService</interfaceName>
     <isLazy>true</isLazy>
 </service>
</metainfo>
```

- 抽象接口类定义如下：

```
public abstract class MyService extends ExternalService {
  public abstract String funA();
}
```

- 接口类实现定义如下：

```
public class MyServiceImpl extends MyService {
  @Override
  public String funA() {
      return "这是 BundleB 提供的接口 by service";
  }

  @Override
  protected void onCreate(Bundle bundle) {

  }

  @Override
  protected void onDestroy(Bundle bundle) {

  }
}
```

- 外部调用方式如下：

```
MyService myservice =
LauncherApplicationAgent.getInstance().getMicroApplicationContext().findServiceByInte
ace(MyService.class.getName());
myservice.funA();
```

BroadcastReceiver 组件

BroadcastReceiver 是 `android.content.BroadcastReceiver` 的封装，但区别在于 mPaaS 框架采用了 `android.support.v4.content.LocalBroadcastManager` 来注册和反注册 BroadcastReciever，因此，这些广播仅用于当前应用程序内部，除此之外，mPaas框架内部内置了一系列的广播事件，供使用者监听。

关于此任务

您可以下载包含该通用组件的代码示例。有关下载地址、使用方法及注意事项，查看 获取代码示例 。

mPaaS内置广播事件

mPaaS 定义了多种广播事件，主要用于监听当前应用的状态，注册监听与原生开发没有任何区别，但有一点需要特别注意，这些状态只有在主进程才能监听到。示例代码如下：

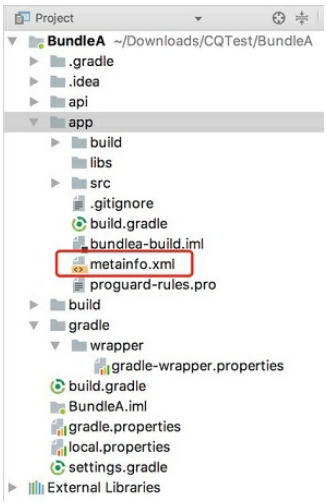
```
IntentFilter filter = new IntentFilter();
filter.addAction(MsgCodeConstants.FRAMEWORK_BROUGHT_TO_FOREGROUND);
filter.addAction(MsgCodeConstants.FRAMEWORK_ACTIVITY_USERLEAVEHINT);
LocalBroadcastManager.getInstance(this).registerReceiver(new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        LoggerFactory.getTraceLogger().debug( s: "demo",  s1: "Received action:" + intent.getAction());
    }
}, filter);
```

内置的广播事件如下：

```
public interface MsgCodeConstants {
    String FRAMEWORK_ACTIVITY_CREATE = "com.alipay.mobile.framework.ACTIVITY_CREATE";
    String FRAMEWORK_ACTIVITY_RESUME = "com.alipay.mobile.framework.ACTIVITY_RESUME";
    String FRAMEWORK_ACTIVITY_PAUSE = "com.alipay.mobile.framework.ACTIVITY_PAUSE";
    // 用户离开的广播，压后台广播
    String FRAMEWORK_ACTIVITY_USERLEAVEHINT =
"com.alipay.mobile.framework.USERLEAVEHINT";
    // 所有 Activity 全都 Stop 的广播，可能代表压后台，但目前没有用相同的判断逻辑
    String FRAMEWORK_ACTIVITY_ALL_STOPPED =
"com.alipay.mobile.framework.ACTIVITY_ALL_STOPPED";
    String FRAMEWORK_WINDOW_FOCUS_CHANGED =
"com.alipay.mobile.framework.WINDOW_FOCUS_CHANGED";
    String FRAMEWORK_ACTIVITY_DESTROY = "com.alipay.mobile.framework.ACTIVITY_DESTROY";
    String FRAMEWORK_ACTIVITY_START = "com.alipay.mobile.framework.ACTIVITY_START";
    String FRAMEWORK_ACTIVITY_DATA = "com.alipay.mobile.framework.ACTIVITY_DATA";
    String FRAMEWORK_APP_DATA = "com.alipay.mobile.framework.APP_DATA";
    String FRAMEWORK_IS_TINY_APP = "com.alipay.mobile.framework.IS_TINY_APP";
    String FRAMEWORK_IS_RN_APP = "com.alipay.mobile.framework.IS_RN_APP";
    // 用户回到前台的广播
    String FRAMEWORK_BROUGHT_TO_FOREGROUND =
"com.alipay.mobile.framework.BROUGHT_TO_FOREGROUND";
}
```

自定义广播事件

1. 定义 `metainfo.xml` 位置，如下图所示：



2. 在 `metainfo.xml` 中写入如下配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<metainfo>
    <broadcastReceiver>

<className>com.mpaas.demo.broadcastreceiver.TestBroadcastReceiver</className>
        <action>com.mpaas.demo.broadcastreceiver.ACTION_TEST</action>
    </broadcastReceiver>
</metainfo>
```

- 自定义 Receiver 实现

```
public class TestBroadcastReceiver extends BroadcastReceiver {
  private static final String ACTION_TEST =
"com.mpaas.demo.broadcastreceiver.ACTION_TEST";

  @Override
  public void onReceive(Context context, Intent intent) {
      String action = intent.getAction();
      if (ACTION_TEST.equals(action)) {
          //TODO
      }
  }
}
```

- 发送广播

```
LocalBroadcastManager.getInstance(LauncherApplicationAgent.getInstance().getApplicati
Context()).sendBroadcast(new
Intent("com.mpaas.demo.broadcastreceiver.ACTION_TEST"));
```

Pipeline 组件

mPaaS 框架有一个比较明显的启动过程，Pipeline 机制允许业务线将自己的运行逻辑封装成 Runnable 放到 Pipeline。框架在适当的阶段启动适当的 Pipeline。

以下为定义的 Pipeline 时机：

- `com.alipay.mobile.framework.INITED`：框架初始化完成。进程在后台启动，框架也会初始化。
- `com.alipay.mobile.client.STARTED`：客户端开始启动。必须等到界面出现，例如，欢迎界面。
- `com.alipay.mobile.TASK_SCHEDULE_SERVICE_IDLE_TASK`：优先级最低，当没有其他高优先级的操作时才会得到执行

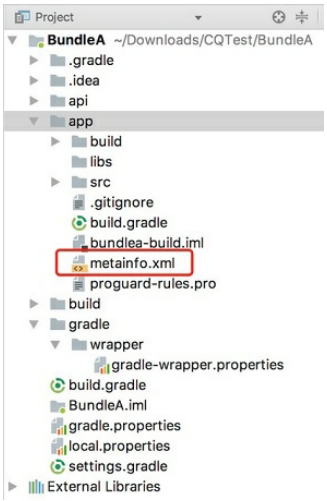
因为 Pipeline 的调用是由框架触发，使用者只需要在 `metainfo` 里指定相应的时机。

关于此任务

您可以下载包含该通用组件的代码示例。有关下载地址、使用方法及注意事项，查看 获取代码示例 。

操作步骤

1. 定义 `metainfo.xml` 位置，如下图所示：



2. 在 `metainfo.xml` 中写入如下的配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<metainfo>
    <valve>
        <className>com.mpaas.demo.pipeline.TestPipeLine</className>
        <!--pipelineName就是用于指定执行所在的阶段-->
        <pipelineName>com.alipay.mobile.client.STARTED</pipelineName>
        <threadName>com.mpaas.demo.pipeline.TestPipeLine</threadName>
        <!--weight指定了操作的优化级，值越小，代表越会优先得到执行-->
        <weight>10</weight>
    </valve>
</metainfo>
```

3. 实现 Pipeline：

```
public class TestPipeLine implements Runnable {
    @Override
    public void run() {
        //....
    }
}
```

2.3.4. 使用 Material Design

本文从 配置工程 和 使用资源 两方面介绍如何使用 Material Design。

配置工程

关于此任务

由于 mPaaS 框架的特殊性，若直接在项目中引入 AppCompat 相关库，编译时会报错，提示资源找不到。为了解决该问题，mPaaS 提供了自定义的 AppCompat 库。要使用 mPaaS 自定义的 AppCompat 库，首先要配置 Portal 工程和 Bundle 工程。

mPaaS AppCompat 库基于原生 Android 23 版本开发，包含以下组件：

- appcompat
- animated-vector-drawable
- cardview
- design
- recyclerview
- support-vector-drawable

由于该自定义的 AppCompat 库是基于原生 Android 23 版本编译，和原生并无区别，只是解决了引入原生库的一系列编译问题。

使用资源的情况主要包括 **使用另一个 Bundle 中的资源** 、**对外提供资源**、**在 AndroidManifest 中使用自定义资源**。由于 mPaaS 框架的特殊性，您需要了解使用资源不同情况下的注意事项。要了解具体内容，查看 使用资源。

操作步骤

1. 配置 Portal 工程。

   在调用 mPaaS AppCompat 库前，完成以下操作配置 Portal 工程：

   i. 运行以下命令将 Gradle 打包插件（Alipay Plugin for Gradle）版本替换为以下版本：

   ```
   classpath 'com.alipay.android:android-gradle-plugin: 3.0.0.9.13'
   ```

   ii. 移除 Gradle 脚本中原先依赖的 AppCompat 库。

   iii. 在 Gradle 脚本中添加以下 AppCompat 依赖：

   ```
   bundle 'com.mpaas.android.res.base:mpaas-baseresjar:1.0.0.180626203034@jar'
   manifest 'com.mpaas.android.res.base:mpaas-
   baseresjar:1.0.0.180626203034:AndroidManifest@xml'
   ```

   iv. 配置完成后，使 Bundle 工程调用 AppCompat 组件，同步 Portal 工程。

2. 配置 Bundle 工程。

   i. 在需要使用 AppCompat 组件的 Bundle 工程中，将 Gradle 打包插件（Alipay Plugin for Gradle）修改为以下版本：

   ```
   classpath 'com.alipay.android:android-gradle-plugin: 3.0.0.9.13'
   ```

   ii. 根据您使用组件的情况，选择需要依赖的子组件。以下为添加 `recyclerview` 的示例语句：

   ```
   provided 'com.mpaas.android.res.base:mpaas-
   baseresjar:1.0.0.180626203034:recyclerview@jar'
   ```

使用资源

Material Design 常见的资源包括 String、Color、Style 等。使用资源的情况主要包括：

- 检测 Package ID 是否重复

- 使用另一个 Bundle 中的资源
- 对外提供资源
- 在 AndroidManifest 中使用自定义资源

检测 Package ID 是否重复

如果按照本文的说明使用资源时，出现资源找不到的情况，您需要查看 Package ID 是否重复。Package ID 定义在 `build.gradle` 中，其 ID 值与生成的资源 ID 有关。重复定义 Package ID 会导致资源找不到的情况。

您可以通过以下两种方式检测 Package ID 是否重复：

方式一：通过 gradle task 自动检测

前置条件

`android-gradle-plugin` 的版本号为 `3.0.0.9.13` 及以上。如：

```
classpath 'com.alipay.android:android-gradle-plugin: 3.0.0.9.13'
```

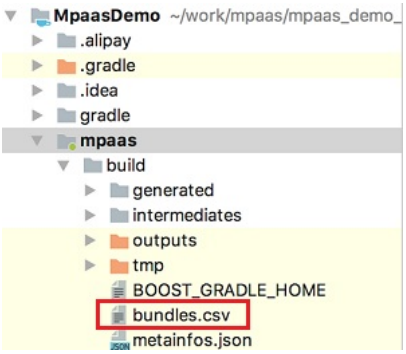
检测步骤

1. 在 Portal 工程根目录下，执行以下命令：
   - Windows 系统：执行 `gradlew.bat checkBundleIds` 。
   - 其它操作系统：执行 `gradlew checkBundleIds` 。
2. 检查输出结果：
   - 若输出结果包含 `No duplicate bundle ids found` ，说明 Package ID 没有重复。
   - 若输出结果包含 `There are duplicate bundle ids` ，说明 Package ID 有重复。您可以根据输出结果进一步判断具体是哪些 Package ID 重复。

方式二：手动检测

手动检测适用于任何情况，具体步骤如下：

1. 在 Portal 工程以下位置打开 `bundles.csv` 纯文本文件。



2. 将 `PackageId` 列进行排序，查看有无重复的 Package ID；确保没有重复的 Package ID 后再重新编译。

使用另一个 Bundle 中的资源

场景示例

使用 `mpaas-baseresjar` 中的资源属于该情况。在使用另一个 Bundle 中的资源时，必须要加上资源的命名空间。命名空间为资源所在 Bundle 的 applicationID，构建 release 包时可能会出现如下图的错误：

```
Gradle tasks [clean, buildRelease]
/Users/mengfei/work/mpaas/mpaas_demo_android/MaterialDesign/app/src/main/res/layout/activity_main.xml
    Error: In Gradle projects, always use http://schemas.android.com/apk/res-auto for custom attributes [ResAuto]
/Users/mengfei/work/mpaas/mpaas_demo_android/MaterialDesign/app/src/main/res/layout/activity_recycler_view.xml
    Error: In Gradle projects, always use http://schemas.android.com/apk/res-auto for custom attributes [ResAuto]
/Users/mengfei/work/mpaas/mpaas_demo_android/MaterialDesign/app/src/main/res/layout/activity_scrolling.xml
    Error: In Gradle projects, always use http://schemas.android.com/apk/res-auto for custom attributes [ResAuto]
/Users/mengfei/work/mpaas/mpaas_demo_android/MaterialDesign/app/src/main/res/layout/activity_share_view.xml
    Error: In Gradle projects, always use http://schemas.android.com/apk/res-auto for custom attributes [ResAuto]
/Users/mengfei/work/mpaas/mpaas_demo_android/MaterialDesign/app/src/main/res/layout/app_bar_main.xml
    Error: In Gradle projects, always use http://schemas.android.com/apk/res-auto for custom attributes [ResAuto]
/Users/mengfei/work/mpaas/mpaas_demo_android/MaterialDesign/app/src/main/res/layout/card_main_1.xml
    Error: In Gradle projects, always use http://schemas.android.com/apk/res-auto for custom attributes [ResAuto]
/Users/mengfei/work/mpaas/mpaas_demo_android/MaterialDesign/app/src/main/res/layout/card_main_2.xml
    Error: In Gradle projects, always use http://schemas.android.com/apk/res-auto for custom attributes [ResAuto]
```

解决方法

在 `build.gradle` 下配置 `lintOptions`，配置方法如下图所示：

```
android {
    compileSdkVersion 23
    buildToolsVersion '26.0.2'

    defaultConfig {
        applicationId "com.mpaas.demo.materialdesign"
        minSdkVersion 18
        targetSdkVersion 23
        versionCode 1
        versionName "1.0"
    }
    buildTypes {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
    lintOptions {
        disable 'ResAuto'
    }
}
```

只要引用该 Bundle 中的资源（包括在 layout 中引用、在自定义 style 中引用等），就必须加入命名空间作为前缀。否则，会出现资源找不到的编译错误。

代码示例：在 layout 中引用

以在 layout 中引用另一个 Bundle 中的资源为例，使用的代码示例如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res/com.mpaas.android.res.base"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true">

    <android.support.design.widget.AppBarLayout
        android:id="@+id/app_bar_scrolling"
        android:layout_width="match_parent"
        android:layout_height="@dimen/app_bar_height_image_view"
        android:fitsSystemWindows="true"
        android:theme="@style/AppTheme.AppBarOverlay"
```

```
        android:background="@color/blue">

        <android.support.design.widget.CollapsingToolbarLayout
            android:id="@+id/collapsing_toolbar_layout"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:fitsSystemWindows="true"
            app:contentScrim="?com.mpaas.android.res.base:attr/colorPrimary"
            app:layout_scrollFlags="scroll|exitUntilCollapsed">

            <ImageView
                android:id="@+id/image_scrolling_top"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:fitsSystemWindows="true"
                android:scaleType="fitXY"
                android:src="@drawable/material_design_3"
                app:layout_collapseMode="parallax" />

            <android.support.v7.widget.Toolbar
                android:id="@+id/toolbar"
                android:layout_width="match_parent"
                android:layout_height="?com.mpaas.android.res.base:attr/actionBarSize"
                app:layout_collapseMode="pin"
                app:popupTheme="@style/AppTheme.PopupOverlay" />

        </android.support.design.widget.CollapsingToolbarLayout>
    </android.support.design.widget.AppBarLayout>

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab_scrolling"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/big_activity_fab_margin"
        android:src="@drawable/ic_share_white_24dp"
        app:layout_anchor="@id/app_bar_scrolling"
        app:layout_anchorGravity="bottom|end" />

    <include layout="@layout/content_scrolling" />

</android.support.design.widget.CoordinatorLayout>
```

代码示例：在自定义 style 中引用

以在自定义 style 中使用另一个 Bundle 中的资源为例，使用的代码示例如下所示：

```
    <style name="AppTheme"
parent="@com.mpaas.android.res.base:style/Theme.AppCompat.NoActionBar">
        <!-- Customize your theme here. -->
        <item name="com.mpaas.android.res.base:colorPrimary">@color/colorPrimary</item>
        <item
name="com.mpaas.android.res.base:colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="com.mpaas.android.res.base:colorAccent">@color/colorAccent</item>
    </style>
```

对外提供资源

1. 配置 `Portal` 工程。在 `Portal` 中引入资源 Bundle 的信息：

```
// 引入提供资源的 bundle
bundle "com.mpaas.demo.materialdesign:materialdesign-build:1.0-SNAPSHOT:raw@jar"
manifest "com.mpaas.demo.materialdesign:materialdesign-build:1.0-
SNAPSHOT:AndroidManifest@xml"
// 为了在编译时能找到资源，还需要 provided 该 bundle 的 jar 包
provided 'com.mpaas.demo.materialdesign:materialdesign-build:1.0-SNAPSHOT:raw@jar'
```

2. 定义资源。完成以下步骤定义资源，以使得资源可被其他的 Bundle 或者 Portal 引用：
   i. 将需要对外提供的资源 id 定义在 `public.xml` 中，以达到固定资源 id 的目的。该能力由 Android 提供。资源 id 值可以从 `R.java` 中拷贝，示例代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
 <public name="AppTheme" id="0x1f030000" type="style" />
 <public name="AppTheme.AppBarOverlay" id="0x1f030001" type="style" />
 <public name="AppTheme.NoActionBar" id="0x1f030002" type="style" />
 <public name="AppTheme.NoActionBar.StatusBar" id="0x1f030003" type="style" />
 <public name="AppTheme.PopupOverlay" id="0x1f030004" type="style" />
 <public name="DialogFullscreen" id="0x1f030005" type="style" />
 <public name="DialogFullscreenWithTitle" id="0x1f030006" type="style" />

 <public name="title_activity_login" id="0x1f0c0081" type="string"/>
 <public name="title_activity_recycler_view" id="0x1f0c0082" type="string"/>
 <public name="title_activity_share_view" id="0x1f0c0085" type="string"/>
 <public name="title_activity_scrolling" id="0x1f0c0083" type="string"/>
 <public name="title_activity_settings" id="0x1f0c0084" type="string"/>
 <public name="title_activity_about" id="0x1f0c007f" type="string"/>
 <public name="activity_donate" id="0x1f0c000e" type="string" />
 <public name="activity_my_apps" id="0x1f0c000f" type="string"/>

</resources>
```

   ii. 外部在使用时，在资源前加上包名作为前缀。具体内容，参见 使用另一个 Bundle 中的资源 。

在 AndroidManifest 中使用自定义资源

如果您在 `Bundle` 工程的 `AndroidManifest` 定义了主题，如下代码所示：

```
<activity
            android:name=".activity.MainActivity"
            android:launchMode="singleTop"
            android:theme="@com.mpaas.demo.materialdesign:style/AppTheme.NoActionBar"
            android:windowSoftInputMode="stateHidden|stateUnchanged">
</activity>
```

则您需要：

- 在 `Portal` 工程的主工程路径下添加 `res_slinks` 文件，并且将 Bundle 名，逐行添加到 `res_slinks` 文件中。
- 同时在 `build.gradle` 中去掉该 `Bundle` 的 `manifest` 依赖 。如下代码所示：

```
manifest 'com.mpaas.demo.materialdesign:materialdesign-
build:1.0.0:AndroidManifest@xml'
```

2.3.5. 使用第三方 AAR 资源

本文介绍在使用组件化接入方式（即 Portal&Bundle 接入方式）的场景下如何使用非 com.android.support 的第三方资源。您可以下载并使用本文提供的示例工程，参考下面的使用方法进行体验。

示例工程中包含 SharedResNew、ZHDemo、ZHDemoLauncher 三个工程：

- SharedResNew：需要被共享的 Bundle，其中包含三方 AAR。
- ZHDemoLauncher：使用到三方资源的 Bundle。
- ZHDemo：Portal 工程。

使用第三方资源的过程主要分为以下四步：

1. 引入第三方资源
2. 使用 public.xml 导出资源
3. 验证资源是否导出成功
4. 使用第三方资源

引入第三方资源

在 SharedResNew 中，`com.flyco.tablayout:FlycoTabLayout_Lib:2.1.2@aar` 这个包需要外部使用，因此需要通过SharedResNew 的 `api` 工程使用 `compile` 方式引入该包。不能使用 `implementation` 方式。

```
compile 'com.flyco.tablayout:FlycoTabLayout_Lib:2.1.2@aar'
```

使用 public.xml 导出资源

在 `app` 项目中导出您需要使用的属性。属性将通过 `public.xml` 文件输出，文件路径固定为 `app/src/main/res/values/public.xml` 。

例如，若要导出属性 `tl_bar_color` ，则 `public.xml` 内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <public name="tl_bar_color" id="0x60010027" type="attr" />
</resources>
```

其中：

- `name` ：需所需属性名保持一致。
- `id` ：在第一次 debug 编译（此时没有 `public.xml` 文件）之后，您可以从 `app/build/generated/source/r/debug/\[com/zh/demo\]\(包名文件夹\)/R.java` 中找到 `id` 值，例如：

```
public static final int tl_bar_color=0x60010027;
```

- `type` ：指属性所属的类。以 `tl_bar_color` 为例，其对应的类如下，其 `type` 值为 `attr` 。

```
public static final class attr {
    ....
}
```

验证资源是否导出成功

进行验证资源是否导出成功前，需确保您已成功构建 SharedResNew。若已完成构建，请完成以下操作进行验证。

步骤 1：找到 aapt 路径

找到 aapt，该文件通常在 Android SDK 中。

假设您的电脑用户名为 `username` ，那么在不同的操作系统下，aapt 的路径如下：

- Mac 操作系统

  如果您的 Android SDK 位置为： `/Users/username/Code/android-sdk` 则 aapt 路径一般为： `/Users/username/Code/android-sdk/build-tools/28.0.3/aapt`

- Windows 操作系统

  如果您的 Android SDK 位置为： `C:\Users\用户名\AppData\Local\Android\Sdk` 则 aapt 路径一般为： `C:\Users\用户名\AppData\Local\Android\Sdk\build-tools\28.0.3\aapt.exe`

> **说明**
>
> build 工具版本需为 26.0.0 或以上版本。

步骤 2：找到本地 bundle 包

打开 SharedResNew > app > build.gradle 后，您会看到如下内容：

```
version = "1.0.0-SNAPSHOT"
group = "com.zh.demo.shared.res"
```

其中， `group` 是 maven gav 中最前面的字段； `version` 指版本号。

您打开 Android Studio 时可以看见 app 工程的名称是 `app [sharedresnew-build]` ，那么该 bundle 的本地 gav 就是 `com.zh.demo.shared.res:sharedresnew-build:1.0.0-SNAPSHOT` 。

对应的本地 Maven 仓库目录是：

- Mac 操作系统

  `~/.m2/repositories/com/zh/demo/shared/res/sharedresnew-build/1.0.0-SNAPSHOT/`

- Windows 操作系统

`C:\Users\用户名\.m2\respositories\com\zh\demo\shared\res\sharedresnew-build\1.0.0-SNAPSHOT`

在该目录下，您会看见以下文件：

```
ivy-1.0.0-SNAPSHOT.xml
ivy-1.0.0-SNAPSHOT.xml.sha1
sharedresnew-build-1.0.0-SNAPSHOT-AndroidManifest.xml
sharedresnew-build-1.0.0-SNAPSHOT-AndroidManifest.xml.sha1
sharedresnew-build-1.0.0-SNAPSHOT-api.jar
sharedresnew-build-1.0.0-SNAPSHOT-api.jar.sha1
sharedresnew-build-1.0.0-SNAPSHOT-raw.jar
sharedresnew-build-1.0.0-SNAPSHOT-raw.jar.sha1
```

步骤 3：执行验证命令

使用上述步骤中找到的 aapt 路径，执行以下验证命令：

- Mac 操作系统

```
/Users/username/Code/android-sdk/build-tools/28.0.3/aapt d --values resources ./sharedresnew-build-1.0.0-SNAPSHOT-api.jar > res.txt
```

- Windows 操作系统

```
C:\Users\用户名\AppData\Local\Android\Sdk\build-tools\28.0.3\aapt.exe d --values resources ./sharedresnew-build-1.0.0-SNAPSHOT-api.jar
```

执行命令后，您会得到一个 `res.txt` 文件，用记事本之类的软件打开此文件，里面的内容如下所示：

```
Package Groups (1)
Package Group 0 id=0x60 packageCount=1 name=com.zh.demo
  DynamicRefTable entryCount=22:
    0x3a -> com.alipay.android.liteprocess
    0x7b -> com.alipay.android.multimediaext
    0x6e -> com.alipay.android.phone.falcon.falconlooks
    0x45 -> com.alipay.android.phone.falcon.img
```

在文件中搜索 `tl_bar_color` ，会找到如下内容。如果后面有 `(PUBLIC)` 标记，则表示资源导出成功；若没有，则表示资源导出失败。

```
resource 0x60010027 com.zh.demo:attr/tl_bar_color: <bag> (PUBLIC)
          Parent=0x00000000(Resolved=0x60000000), Count=1
          #0 (Key=0x01000000): (color) #00000010
```

使用第三方资源

在使用资源的地方，例如在 ZHDemoLauncher 项目的任意一个 layout 中，在 xml 顶部增加 xml 命名空间。下面以 `ZHDemoLauncher/app/src/main/res/layout/main.xml` 为例：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:abc="http://schemas.android.com/apk/res/com.zh.demo"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <!-- xxxx -->
</LinearLayout>
```

说明

`xmlns:abc="http://schemas.android.com/apk/res/com.zh.demo"` 这一行中：

- `abc` 为自定义名称，您可以任意命名；
- `http://schemas.android.com/apk/res/` 为固定值；
- `com.zh.demo` 是您在 SharedResNew 的 `AndroidManifest.xml` 中定义的 `package` 值。该包值也可以在使用 aapt 导出的 .txt 文件看到，例如 `resource 0x60010027 com.zh.demo:attr/tl_bar_color`，冒号前面的 `com.zh.demo` 就是需要用的值。

在使用的地方资源加上以下代码：

```
<com.flyco.tablayout.SegmentTabLayout
       ....
       abc:tl_bar_color="#f00" />
```

合起来是：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:abc="http://schemas.android.com/apk/res/com.zh.demo"
    android:layout_width="match_parent"
    android:layout_height="match_parent"

    android:orientation="vertical"
    tools:ignore="ResAuto">
   <com.flyco.tablayout.SegmentTabLayout
       android:id="@+id/myView"
       android:layout_width="wrap_content"
       android:layout_height="32dp"
       android:layout_gravity="center_horizontal"
       android:layout_marginTop="10dp"
       abc:tl_bar_color="#f00"
       tools:visibility="visible" />
</LinearLayout>
```

至此，您已经完成编译。

代码示例

点击 下载示例代码 。

2.3.6. 加载框架与定制

mPaaS Android 框架提供了一整套的加载逻辑。基于此框架，研发团队可以进行多业务线开发。本文描述框架的启动流程以及如何在框架下添加自己的代码以对接启动。

启动流程

Application

传统 Android apk 运行时首先加载 `AndroidManifest` 文件 `application` 节点中 `android:name` 配置的 Application。

由于 mPaaS Android 框架重写了加载流程， `android:name` 中配置 mPaaS Android 框架的 `com.alipay.mobile.quinox.LauncherApplication` 类。

```
<application
    android:name="com.alipay.mobile.quinox.LauncherApplication"
    android:allowBackup="true"
    android:debuggable="true"
    android:hardwareAccelerated="false"
    android:icon="@drawable/appicon"
    android:label="@string/name"
    android:theme="@style/AppThemeNew" >
</application>
```

启动页

由于框架加载 bundle 可能会比较耗时，因此需要一个启动页等待框架启动完成之后再跳转到程序主页，所以在 `AndroidManifest` 文件中配置了框架提供的 `com.alipay.mobile.quinox.LauncherActivity` 应用启动页。

配置如下：

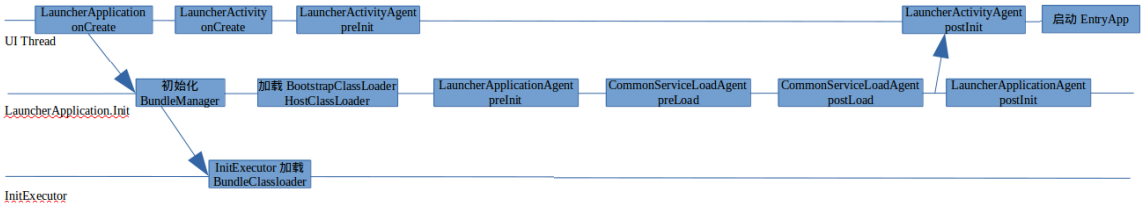
```
<activity
android:name="com.alipay.mobile.quinox.LauncherActivity"
android:configChanges="orientation | keyboardHidden | navigation"
android:screenOrientation="portrait"
android:windowSoftInputMode="stateAlwaysHidden">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
```

为方便开发人员对启动过程的理解，也为了避免启动过程被误改误删、被干扰，mPaaS 的启动过程被适度封装。因此，上述的 `LauncherApplication` 和 `LauncherActivity` 对开发人员完全不可见。

为了让客户端 App 在启动过程中实现自身的初始化逻辑，mPaaS 设计了 `LauncherApplicationAgent` 和 `LauncherActivityAgent` 代理。作为开发人员，您可以通过继承这两个类，在相应的回调中实现自身的初始化逻辑。当您在 bundle 工程中定义了这两个类时，在使用 ProGuard 进行代码混淆的时候则需要对这两个类进行防混淆设置，详情请参见 混淆 Android 文件。

启动流程图

mPaaS Android 框架加载流程如下：



1. 框架启动后主线程会创建启动页 `LauncherActivity` ，然后回调 `LauncherActivityAgent` 的 `preInit` 方法。
2. 框架进行 multidex。过程中会回调 `LauncherApplicationAgent` 的 `preInit` 方法，读取当前 `.apk` 中每个 bundle 的描述文件，并对每个 bundle 创建对应的类加载器，加载其中的资源文件。
3. 初始化完成后，回调 `LauncherActivityAgent` 和 `LauncherApplicationAgent` 的 `postInit` 方法。

定制

实际上，框架已在 Launcher 工程中自动创建了两个类 `MockLauncherApplicationAgent` 和 `MockLauncherActivityAgent` 继承了 `LauncherApplicationAgent` 和 `LauncherActivityAgent` 这两个回调接口。在框架的初始化过程中，它们分别在 `LauncherAppliction` 和 `LauncherActivity` 中被调用。

在 Portal 的 `AndroidManifest.xml` 中配置如下。开发人员也可在 Bundle 中实现这两个代理类，在以上配置中修改对应 `meta-data` 的 `value` 值：

```
     <application
          android:name="com.alipay.mobile.quinox.LauncherApplication" >

         <!-- Application 的回调配置 -->
         <meta-data
            android:name="agent.application"

android:value="com.mpaas.demo.launcher.framework.MockLauncherApplicationAgent"/>

        <!-- Activity 的回调配置 -->
        <meta-data
            android:name="agent.activity"

android:value="com.mpaas.demo.launcher.framework.MockLauncherActivityAgent"/>
        <!-- 启动页的布局配置 -->
        <meta-data
            android:name="agent.activity.layout"
            android:value="layout_splash"/>

     </application>
```

代理类

`agent.application` 配置的是启动过程代理 `ApplicationAgent` ，如下所示：

```
public class MockLauncherApplicationAgent extends LauncherApplicationAgent {
    @Override
    protected void preInit() {
        super.preInit();
        //框架初始化前
    }

    @Override
    protected void postInit() {
        super.postInit();
        //框架初始化后
    }
}
```

客户端 App 可以在 `LauncherApplicationAgent` 的实现类中，进行一些 Application 级别的初始化工作。`preInit()` 回调发生在框架初始化之前，因此不要在这里调用框架（`MicroApplicationContext`）的相关接口。而 `postInit()` 回调发生在框架初始化完成之后，是可以使用的。

`agent.activity` 配置的是启动 Activity 的代理，如下所示：

```
public class MockLauncherActivityAgent extends LauncherActivityAgent {

        @Override
        public void preInit(Activity activity) {
                super.preInit(activity);
                //Launcher Activity 启动前
        }

        @Override
        public void postInit(final Activity activity) {
            super.postInit(activity);
                //Launcher Activity 启动后
                //跳转程序首页的逻辑
                startActivity(activity,YOUR_ACTIVITY);
        }
}
```

同上述的 `LauncherApplicationAgent` 类似，`LauncherActivityAgent` 的两个回调也分别发生在框架初始化之前和之后，使用方法也类似。

修改启动页布局

在 Portal 的 `AndroidManifest.xml` 中还配置了启动页的布局文件，如下所示：

```
<application
android:name="com.alipay.mobile.quinox.LauncherApplication" >
<!-- 启动页的布局配置 -->
<meta-data
android:name="agent.activity.layout"
android:value="layout_splash"/>

</application>
```

修改 value 值为自定义的布局文件名。

> **说明**
> 需要把布局文件和引用的相关资源放置在 Portal 工程里。

相关链接

代码示例

2.3.7. 管理 Gradle 依赖

Gradle 提供配置依赖仓库和配置发布仓库的功能。

配置依赖仓库

mPaaS 常见依赖仓库示例如下：

```
allprojects {
    repositories {
        mavenLocal()
        flatDir {
            dirs 'libs'
        }
        maven {
            url "https://mvn.cloud.alipay.com/nexus/content/repositories/open/"
        }
        maven{url 'http://maven.aliyun.com/nexus/content/groups/public/'}
        maven{url 'http://maven.aliyun.com/nexus/content/repositories/google'}
    }
}
```

- **mavenLocal**：Maven 本地仓库。**本地仓库的路径** 也支持修改。
- **flatDir**：工程 libs 目录下的依赖。
- **maven**：示例中包含蚂蚁科技（`mvn.cloud.alipay.com`）和阿里云（`maven.aliyun.com`）的 Maven 仓库。

您可以在 `repositories` 下 **新增依赖仓库**。

配置发布仓库

本文将简述发布仓库常见示例，帮助您修改本地 Maven 仓库路径（默认 `~/.m2`）、增加自定义发布仓库。

发布仓库示例

一般地，`build.gradle` 文件中有如下配置：

```
uploadArchives {
    repositories {
        mavenLocal()
    }
}
```

这意味着发布仓库为 **本地 Maven 仓库**，即工程打出的 `.jar` 包等会自动发布到本地 Maven 仓库。

修改本地 Maven 仓库路径

本地 Maven 仓库（ `mavenLocal` ）默认路径为 `~/.m2` ，您可以自定义修改。

自定义发布仓库

您可以根据实际情况增加自定义发布仓库，示例如下：

```
uploadArchives {
    mavenDeployer {
        mavenLocal()
        repository(url: "your_repository_url") {
            authentication(userName: '*****', password: '*****')
        }
        snapshotRepository(url: "your_repository_url") {
            authentication(userName: '*****', password: '*****')
        }
    }
}
```

2.3.8. 混淆 Android 文件

mPaaS Android 客户端开发的应用程序是通过 Java 代码编写而成，而 Java 代码易被反编码，因此为了保护 Java 源代码，需要使用 ProGuard 混淆 Android 文件。

ProGuard 是一个压缩、优化和混淆 Java 字节码文件的工具。

- **压缩** 指检测以及删除没有用到的类、字段、方法以及属性。
- **优化** 指分析以及优化方法的字节码。
- **混淆** 指使用无意义的短变量，对类、变量、方法进行重命名。

使用 ProGuard 可以让代码更精简，更高效，也更难被逆向或破解。

前置条件

您已经配置 mPaaS 工程。

关于此任务

采用组件化方案的 mPaaS 工程，每一个 bundle 的编译产物都是一个已经混淆的 `dex` 文件，所以配置混淆文件是以 bundle 工程为单位而进行的。Portal 工程通常没有代码，不需要开启混淆。

代码示例

- **Gradle 配置**

```
android {
  compileSdkVersion 23
  buildToolsVersion "19.1.0"

  defaultConfig {
      applicationId "com.youedata.xionganmaster.launcher"
      minSdkVersion 15
      targetSdkVersion 23
      versionCode 1
      versionName "1.0"
  }
  buildTypes {
      release {
          // 混淆开关，是否混淆
          minifyEnabled true
          // 混淆文件列表，混淆规则配置
          proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-
rules.pro'
      }
  }
  lintOptions {
        checkReleaseBuilds false
        // Or, if you prefer, you can continue to check for errors in release builds,
        // but continue the build even when errors are found:
        abortOnError false
  }
}
```

- **混淆文件示例**

  下列混淆是一个基本示例（如果要添加额外的第三方库，需要加入其它混淆，通常配置文件可在第三方库的官网中找到）：

```
  # Add project specific ProGuard rules here.
  # By default, the flags in this file are appended to flags specified
  # in ${sdk.dir}/tools/proguard/proguard-android.txt
  # You can edit the include path and order by changing the proguardFiles
  # directive in build.gradle.

  # For more details, see [Shrink your code and resources]
(http://developer.android.com/guide/developing/tools/proguard.html)。

  # Add any project specific keep options here:

  # If your project uses WebView with JS, uncomment the following
  # and specify the fully qualified class name to the JavaScript interface
  # class:
  # -keepclassmembers class fqcn.of.javascript.interface.for.webview {
  # public *;
  # }
  -optimizationpasses 5
  -dontusemixedcaseclassnames
  -dontskipnonpubliclibraryclasses
  -dontpreverify
  -verbose
```

```
  -ignorewarnings
  -optimizations !code/simplification/arithmetic,!field/*,!class/merging/*

  -keep public class * extends android.app.Activity
  -keep public class * extends android.app.Application
  -keep public class * extends android.app.Service
  -keep public class * extends android.content.BroadcastReceiver
  -keep public class * extends android.content.ContentProvider
  -keep public class com.android.vending.licensing.ILicensingService
  -keep public class com.alipay.mobile.phonecashier.*
  -keepnames public class *
  -keepattributes SourceFile,LineNumberTable
  -keepattributes *Annotation*

  #-keep public class * extends com.alipay.mobile.framework.LauncherApplicationAgent
{
  #    *;
  #}

  #-keep public class * extends com.alipay.mobile.framework.LauncherActivityAgent {
  #    *;
  #}

  -keepclasseswithmembernames class * {
      native <methods>;
  }

  -keepclasseswithmembernames class * {
      public <init>(android.content.Context, android.util.AttributeSet);
  }

  -keepclasseswithmembernames class * {
      public <init>(android.content.Context, android.util.AttributeSet, int);
  }

  -keepclassmembers enum * {
      public static **[] values();
      public static ** valueOf(java.lang.String);
  }

  -keep class * extends java.lang.annotation.Annotation { *; }
  -keep interface * extends java.lang.annotation.Annotation { *; }

  -keep class * implements android.os.Parcelable {
    public static final android.os.Parcelable$Creator *;
  }

  -keep public class * extends android.view.View{
      !private <fields>;
      !private <methods>;
  }

  -keep class android.util.**{
       public <fields>;
```

```
      public <methods>;
  }

-keep public class  com.squareup.javapoet.**{
    !private <fields>;
      !private <methods>;
}
-keep public class   javax.annotation.**{
        !private <fields>;
        !private <methods>;
  }
-keep public class   javax.inject.**{
     !private <fields>;
     !private <methods>;
  }
-keep interface **{
  !private <fields>;
  !private <methods>;
}
# for dagger
  -keep class * extends dagger.internal.Binding
  -keep class * extends dagger.internal.ModuleAdapter

  -keep class **$$ModuleAdapter
  -keep class **$$InjectAdapter
  -keep class **$$StaticInjection

  -keep class dagger.** { *; }

  -keep class javax.inject.**{ *; }
  -keep class * extends dagger.internal.Binding
  -keep class * extends dagger.internal.ModuleAdapter
  -keep class * extends dagger.internal.StaticInjection

# for butterknife
  -keep class butterknife.* { *; }
  -keep class butterknife.** { *; }
  -dontwarn butterknife.internal.**
  -keep class **$$ViewBinder { *; }

  -keepclasseswithmembernames class * {
      @butterknife.* <fields>;
  }

  -keepclasseswithmembernames class * {
      @butterknife.* <methods>;
  }
```

说明

如果在您的 bundle 工程中定义了框架类 `LauncherApplicationAgent` 和 `LauncherActivityAgent`，请注意进行防混淆设置

- **避免混淆通用组件**

  如果在 `metainfo.xml` 中注册了通用组件，编译时会检查这些组件是否存在，请避免这些组件被混淆，否则会编译失败。例如注册了以下组件：

```
    <metainfo>
    <service>
        <className>com.mpaas.cq.bundleb.MyServiceImpl</className>
        <interfaceName>com.mpaas.cq.bundleb.api.MyService</interfaceName>
        <isLazy>true</isLazy>
    </service>
</metainfo>
```

  请在混淆配置中添加：

```
  -keep class com.mpaas.cq.bundleb.MyServiceImpl
  -keep class com.mpaas.cq.bundleb.api.MyService
```

2.3.9. mPaaS Portal&Bundle 工程使用 MultiDex 的注意事项

不建议您自行在 Portal 和 Bundle 接入方式中接入 MultiDex，除非您是单 Portal 工程，需要使用 `multiDexEnabled true` 。如果您的 Bundle 过大，目前只能使用拆分 Bundle 的方式进行， **不要在 Bundle 中开启 multidex 支持。**

2.3.10. 数据清理白名单

为应对可能发生的连续启动崩溃的情况，mPaaS 建立了数据清理的机制。当 mPaaS 框架启动完成前应用出现卡死或重要线程（例如主线程、multidex.init 线程、ApplicationAgent.init 线程等）发生闪退时，框架可能触发数据清理。该数据清理机制支持定制，通过配置实现在不同情况下对 SharedPreference 、Database 数据库的清理，在极特殊情况下清空整个应用的数据，以保证应用的正常运行。目前该机制已经覆盖了 10.1.32、10.1.60 和 10.1.68 系列基线。

为满足保护重要数据的需求，mPaaS 在数据清理机制中提供了清理白名单功能。通过将目标文件加入清理白名单后即可保护该文件不被清理。

> **说明**
>
> 只有 Portal&Bundle 接入方式 存在数据清理机制。

清理白名单方案 1.0

清理白名单方案 1.0 是在合适的时机调用 `MPFramework` 中的接口来动态设置白名单列表。

支持的基线

清理白名单方案 1.0 支持 10.1.32、10.1.60 和 10.1.68 系列基线。

如果在设置白名单之前已经因为崩溃触发了清理机制，那么清理白名单方案 1.0 将不能生效。如果您使用的是 10.1.32 系列基线，那么建议您升级基线到 10.1.60 或 10.1.68，以使用升级后的清理白名单方案 2.0。更多信息，请参见 清理白名单方案 2.0 。

接入步骤

只需在合适的时机调用设置清理白名单的接口即可。接口如下：

```
/**
     * 设置 SharedPreference 白名单。若之前已经设置过，那么会把之前的数据清空。
     */
    public static void setSPWhiteList(List<String> whiteList);

    /**
     * 添加 SharedPreference 白名单，以追加的方式添加。
     *
     * @param whiteList
     */
    public static void addSPWhiteList(List<String> whiteList);

    /**
     * 获取设置的数据库白名单列表。
     *
     * @return
     */
    public static List<String> getDBWhiteList();

    /**
     * 设置数据库白名单。若之前已经设置过，那么会把之前的数据清空。
     */
    public static void setDBWhiteList(List<String> whiteList) ;

    /**
     * 添加数据库白名单，以追加的方式添加。
     *
     * @param whiteList
     */
    public static void addDBWhiteList(List<String> whiteList);
```

清理白名单方案 2.0

清理白名单方案 2.0 是在触发到清理机制时，框架通过反射加载开发者配置的白名单来设置类，优先读取自定义的清理策略。

支持的基线

清理白名单方案 2.0 支持 10.1.60 和 10.1.68 系列基线。其中：

- 10.1.60 基线要求 10.1.60.10 版本及以上。
- 10.1.68 基线要求 10.1.68.4 版本及以上。

接入步骤

1. 继承 `com.mpaas.framework.adapter.api.ClearDataStrategy` ，实现相关接口。

```
public abstract class ClearDataStrategy {
    public ClearDataStrategy() {
    }

    /**
     * 是否开启清理机制。
     * 若返回 false，则什么文件都不清理。
     * 若返回 true，则会执行清理策略。可以通过 getSPWhiteList，getDBWhiteList 返回需要保证的文件列表。
     *
     * @return
     */
    public abstract boolean enableClearDataStrategy();

   /**
     * 若开启了清理机制，通过该接口返回需要保护的 SharedPreference 文件。
     *
     * @return
     */
    public List<String> getSPWhiteList() {
        return null;
    }

   /**
     * 若开启了清理机制，通过该接口返回需要保护的 db 文件。
     *
     * @return
     */
    public List<String> getDBWhiteList() {
        return null;
    }
}
```

2. 在 Portal 的 `AndroidManifest` 中配置策略类的信息。

> **说明**
> 由于需要反射调用 `ClearDataStrategy` ，所以不能混淆 `ClearDataStrategy` 。

```
        <meta-data
            android:name="ClearDataStrategy"
            android:value="com.mpaas.demo.launcher.ClearDataStrategy" />
```

2.3.11. 清理隐私权限

由于 Android 系统的升级变迁和 mPaaS 自身业务的发展等历史原因，在默认的 portal 工程中会存在一部分冗余权限，如下列表所示。这些权限在当前的 mPaaS 版本中已经不再需要，您可以选择删除或按需保留。

高危可清理权限

以下 5 个是高危权限，经过验证可以清除。

```
<uses-permission android:name="android.permission.RECEIVE_SMS" />
<uses-permission android:name="android.permission.READ_SMS" />
<uses-permission android:name="android.permission.READ_LOGS" />
<uses-permission android:name="android.permission.BATTERY_STATS" />
<uses-permission android:name="android.permission.MANAGE_FINGERPRINT" />
```

不必要权限

以下权限非高危隐私权限，但这些权限在 mPaaS 产品对外过程中不需要使用到。如果有特殊需要的话，可以保留，否则都可以去除。

```
<uses-permission android:name="com.alipay.permission.ALIPAY_UPDATE_CREDENTIALS" />
<uses-permission android:name="com.yunos.permission.TYID_SERVICE" />
<uses-permission android:name="com.taobao.permission.USE_CREDENTIALS" />
<uses-permission android:name="com.htc.launcher.permission.READ_SETTINGS" />
<uses-permission android:name="com.majeur.launcher.permission.UPDATE_BADGE" />
<uses-permission android:name="com.aliyun.permission.TYID_SERVICE" />
<uses-permission android:name="com.htc.launcher.permission.UPDATE_SHORTCUT" />
<uses-permission android:name="com.anddoes.launcher.permission.UPDATE_COUNT" />
<uses-permission android:name="com.yunos.permission.STORAGE_SERVICE" />
<uses-permission android:name="com.aliyun.permission.STORAGE_SERVICE" />
<uses-permission android:name="com.alipay.permission.ALIPAY_USE_CREDENTIALS" />
<uses-permission android:name="com.sonyericsson.home.permission.BROADCAST_BADGE" />
<uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW" />
<uses-permission android:name="nxp.permission.ACCESS_WALLET_SERVICE" />
<uses-permission
android:name="com.samsung.android.authservice.permission.READ_CONTENT_PROVIDER" />
<uses-permission android:name="com.taobao.permission.UPDATE_CREDENTIALS" />
<uses-permission android:name="com.yunos.permission.TYID_MGR_SERVICE" />
<uses-permission android:name="com.aliyun.permission.TYID_MGR_SERVICE" />

<uses-permission android:name="com.android.launcher.permission.UNINSTALL_SHORTCUT" />
<uses-permission android:name="com.android.launcher.permission.INSTALL_SHORTCUT" />

<uses-permission android:name="android.permission.AUTHENTICATE_ACCOUNTS" />
<uses-permission android:name="android.permission.USE_CREDENTIALS" />
<uses-permission android:name="android.permission.MANAGE_ACCOUNTS" />
<uses-permission android:name="android.permission.GET_ACCOUNTS" />
<uses-permission android:name="android.permission.WRITE_SETTINGS" />
<uses-permission android:name="android.permission.READ_PROFILE" />
<uses-permission android:name="android.permission.USE_FINGERPRINT" />
```

2.3.12. 隐私权限弹框的使用说明

监管部门要求在用户点击隐私协议弹框中 同意 按钮之前，App 不可以调用相关敏感 API。为应对此监管要求，mPaaS Android 10.1.32.17 以上（32 版本） 和 10.1.60.5 以上（60 版本）的基线提供了支持。如果您采用的是组件化的接入方法，请您根据实际情况参考本文对工程进行改造。

使用方法

> **重要**
>
> 弹出隐私弹框的 Activity 不可以继承 mPaaS 的 BaseActivity，因为 BaseActivity 会进行埋点数据

> 采集，会导致 App 在同意隐私政策之前采集隐私数据。

1. 新建隐私许可弹框回调类。在代码中新建一个类，实现 `PrivacyListener` 接口，类的实现可以参考以下代码：

```
public class MyPrivacyListener implements PrivacyListener {
 // 在本方法内进行隐私许可弹框
 @Override
 public void showPrivacy(final Activity activity, final PrivacyResultCallback privacy
ResultCallback) {
     if(null==privacyResultCallback){
        return;
     }
     if(null!=activity){
         new AlertDialog.Builder(activity)
                 .setTitle("隐私许可弹框")
                 .setMessage("主体内容")
                 .setPositiveButton("同意继续使用", new
DialogInterface.OnClickListener() {
                     @Override
                     public void onClick(DialogInterface dialogInterface, int i) {
                         // 点击确定后，取消弹框
                         dialogInterface.cancel();
                         // 将弹框结果设置为 true
                         privacyResultCallback.onResult(true);
                     }
                 })
                 .setNegativeButton("不同意并退出", new
DialogInterface.OnClickListener() {
                     @Override
                     public void onClick(DialogInterface dialogInterface, int i) {
                         // 点击不同意后，取消弹框
                         dialogInterface.cancel();
                         // 将弹框结果设置为 false
                         privacyResultCallback.onResult(false);
                         // 结束掉当前的 activity，框架会杀掉进程
                         if(null!=activity){
                             activity.finish();
                         }
                     }
                 })
                 .setCancelable(false)
                 .create()
                 .show();
     }else{
         // 如果 activity 是空的话，回调结果设置 false
         privacyResultCallback.onResult(false);
     }
 }
}
```

> **说明**
>
> 如果您使用的是 10.1.68.42 及以上版本基线且需要清除隐私状态，请实现 `PrivacyListener2` 接口，并实现 `shouldClear` 函数。

以下代码是对 `shouldClear` 函数的说明：

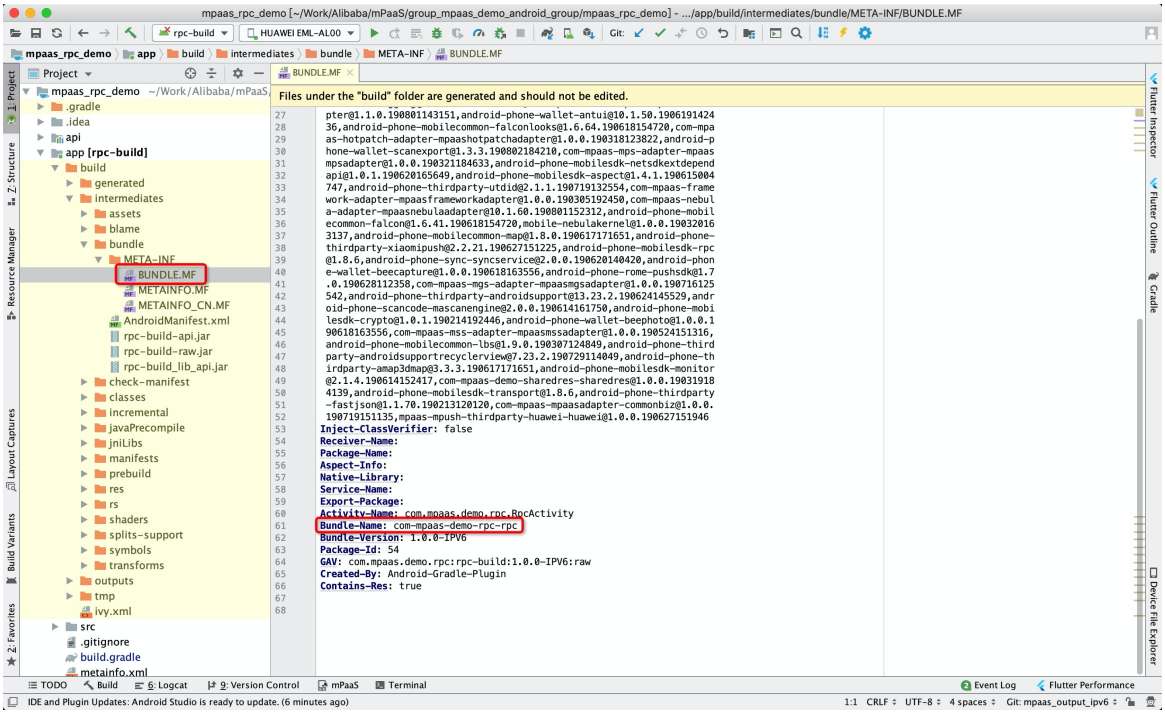
```
@Override
public boolean shouldClear(Context context) {
  //用户没有点同意隐私协议默认用 SharedPreferences 存储 false 设置为 return false。如果需要再次弹窗需 SP 存储 true 设置为return true;
  return false;//return 的值取 SP 里面存的 boolean 值。
}
```

回调时，一定要弹出一个对话框来触发 `windowFocusChange` ，触发后框架才会进行后续的操作。由于该回调类会由系统框架进行反射初始化，且时机非常早，因此请不要添加带方法名的构造函数，以及在构造函数中加入具体逻辑。

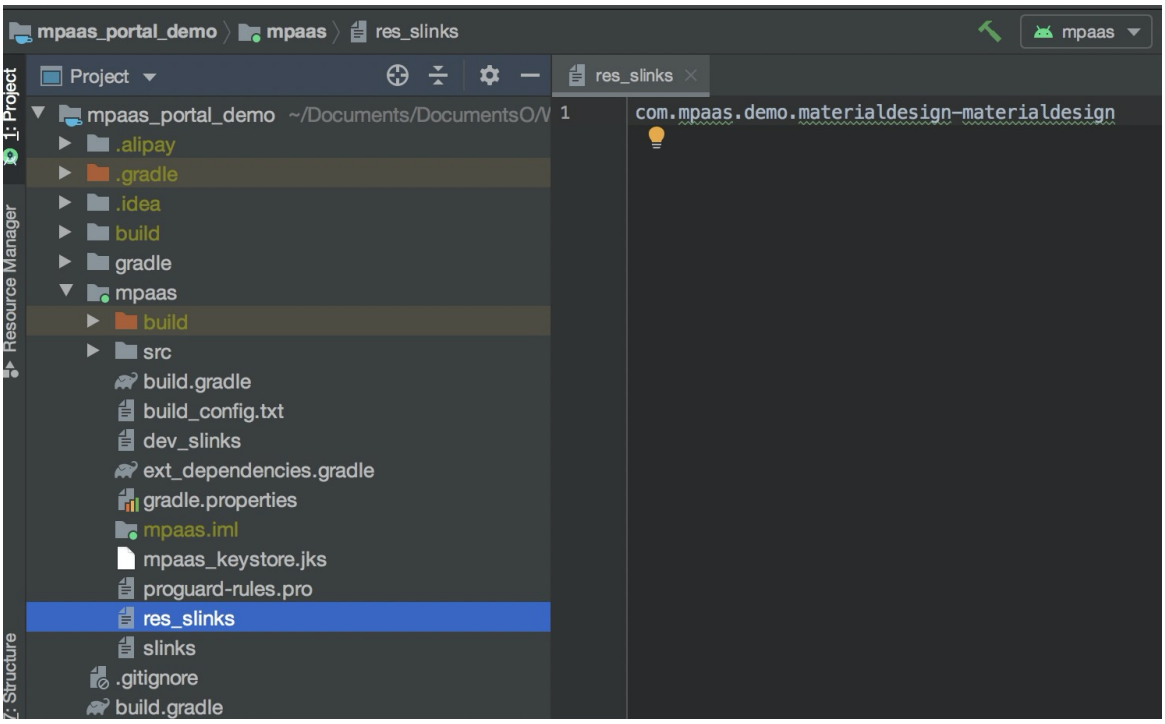
如果您需要在弹出对话框这里使用资源，在不同基线下需要采用不同的方法。

- 在 32 基线下，您需要采用如下方法： `Resources resource = QuinoxAgent.getInstance().getResourcesByBundle("资源所在的bundle的bundlename");`

> **说明**
>
> `bundlename` 可在 Bundle 工程中的主 module 中的 `/build/intermediates/bundle/META-INF/BUNDLE.MF` 中查看。



- 在 60 基线下，您需要在 Portal 工程的主 module 下，建立 `res_slinks` 文件，并将您的资源所在的 bundle 的 `group` 和 `artifact` 按照规则写到 `res_slinks` 文件内。规则为 `group-artifact.split("-")[0]` 。如果组合后内容过长需要换行，请您注意进行换行处理。例如： `group = com.mpaas.demo.materialdesignartifact = materialdesign-build` ，最终写入 `res_slinks` 文件中的配置是 `com.mpaas.demo.materialdesign-materialdesign` 。



完成以上内容后，您可以直接使用 `LayoutInflator.inflate(R.layout.xxx)` 来调用资源。

2. 在 `AndroidManifest` 中注册弹框回调类。在 `portal` 的 `AndroidManifest` 中注册隐私许可弹框回调类， `value` 是刚才实现的隐私许可弹框回调类的全路径。代码如下所示。注意将全路径及类名要替换成您自己的回调类。

```
<!--隐私许可弹框回调-->
<meta-data
 android:name="privacy.listener"
 android:value="com.mpaas.demo.launcher.MyPrivacyListener" />
```

3. 启动弹框拦截。在 `MockLauncherApplicationAgent` 的 `preInit` 中，加入启动弹框拦截。代码如下所示：

```
//检测是否要向用户进行隐私许可弹框
if(!PrivacyUtil.isUserAgreed(getApplicationContext())){
 PermissionGate.getInstance().waitForUserConform(mContext,
getMicroApplicationContext());
}
```

4. 启动第一个 Activity。在 `MockLauncherActivityAgent` 的 `postInit` 中，进行第一个 `Activity` 的跳转。代码如下所示：

```
public class MockLauncherActivityAgent extends LauncherActivityAgent {

    private Handler mUIHandler = new Handler(Looper.getMainLooper());

    @Override
    public void preInit(Activity activity) {
        super.preInit(activity);
    }

    @Override
    public void postInit(final Activity activity) {
        super.postInit(activity);

        if (PrivacyUtil.isUserAgreed(activity)) {
            mUIHandler.postDelayed(new Runnable() {
                public void run() {
                    Intent intent = new Intent(activity, MainActivity.class);
                    activity.startActivity(intent);
                    activity.finish();
                }
            }, 1000);
        } else {

            PermissionGate.getInstance().registerPrivacyCallback(
                    new PrivacyCallback() {
                        @Override
                        public void onTermsOfUseAgreed() {
                            mUIHandler.postDelayed(new Runnable() {
                                public void run() {
                                    Intent intent = new Intent(activity,
MainActivity.class);
                                    activity.startActivity(intent);
                                    activity.finish();
                                }
                            }, 1000);
                        }
                    }
            );
        }
    }

}
```

3.选择基线

3.1. 基线简介

基线是指一系列功能的稳定版本的集合，是进一步开发的基础。而 mPaaS 产品是基于支付宝的某个特定版本开发的，因此对于 mPaaS 而言，基线则是所基于版本的 SDK 的集合。随着 mPaaS 产品的不断升级，会出现多个版本的基线。

10.2.3 基线

基于 10.1.68 基线新增以下特性：

- 从 mPaaS 10.2.3.4 起，支持 targetSdkVersion 31。
- 支持 targetSdkVersion 30。
- CPU 架构仅支持 armeabi-v7a 和 arm64-v8a ，不再支持 armeabi。
- 接入方式不再维护 mPaaS Inside 方式，原 mPaaS Inside 接入如需升级到 10.2.3，请修改为 mPaaS AAR 接入。
- 默认适配 Android 13，升级后无需额外适配工作。

详情请参见 发布说明 10.2.3。

10.1.68 基线

基于 10.1.60 基线新增以下特性：

- 新增了 AAR 接入方式，更贴近原生体验。
- 为单组件提供了更好的支持，提供单组件 demo，更多信息，请参考 获取代码示例。
- 优化单组件 SDK 大小，使整体应用包体积有效降低。
- 对小程序进行更细粒度拆分，用户可根据自身需求进行选择。
- 更新 UC 内核更新至 3.0，提供了更好的性能和更强的稳定性。

详情请参见 发布说明 10.1.68。

10.1.60 基线

基于 10.1.32 基线新增以下特性：

- 增加了 **小程序组件** 正式版。小程序正式版拥有更加完善的 API，且在稳定性、兼容性等方面有了大幅提高。关于小程序升级请参见 小程序升级说明，关于小程序 IDE 新增调试、预览、发布等功能的详情请参见 小程序 IDE。
- 对 **H5 容器** 整体进行大幅优化，提供了更加简化的接入流程，持续补强能力，在兼容性、稳定性等方面有显著提高。关于 H5 容器和离线包升级，请参见 H5 容器升级说明。
- 消息推送组件新增加了对 OPPO 和 vivo 渠道推送的支持。
- 新增加了对 **社交分享** 组件的管理支持，提供了简化的接入流程。
- 新增加 **智能投放** 组件。智能投放提供了在应用内个性化投放广告的能力，支持针对定向人群进行个性化广告投放，帮助 APP 运营人员精准、及时触达用户，详情请参见 智能投放。

详情请参见 发布说明 10.1.60。

10.1.32 基线

10.1.32 版本是现有 mPaaS 对外发布基线中最早的系列，现已进入维护期，除修复漏洞或 bug 外，不再进行需求迭代。详情请参见 发布说明 10.1.32。

选择基线

- 10.1.68 基线正式支持 AAR，如果您需要使用 AAR 方式接入，请选择 10.1.68 基线。
- 10.1.60 基线暂不支持 AAR 方式接入。
- 10.1.32 基线已进入维护期，不会有新增功能，因此不推荐新用户使用。

3.2. mPaaS 10.1.68 升级指南

基于 10.1.60 版本，mPaaS 10.1.68 进行了以下更新：

- 新增了 AAR 接入方式，更贴近原生体验。更多 AAR 接入方式的信息，请参考 原生 AAR 接入方式。
- 为单组件提供了更好的支持，提供单组件 demo，更多信息，请参考 获取代码示例。
- 优化单组件 SDK 大小，使整体应用包体积有效降低。
- 对小程序进行更细粒度拆分，用户可根据自身需求进行选择。
- 更新 UC 内核更新至 3.0，提供了更好的性能和更强的稳定性。

升级指南

AAR 接入方式下的升级指南

如果您已有采用原生 AAR 接入方式的工程，请按照以下步骤完成升级。

1. 环境配置。

```
gradle = 6.5 // 需使用 6.5 及以上版本
com.android.tools.build:gradle:4.0.0 // 需使用 4.0.0 及以上版本
com.android.boost.easyconfig:easyconfig:2.8.0
```

> **重要**
>
> 如您需要设置 `com.android.tools.build:gradle` 为 4.2 或以上，则需要在 `gradle.properties` 文件进行如下配置：`android.enableResourceOptimizations=false`。

2. 参考 更新 mPaaS 插件 文档，升级 Android Studio mPaaS 插件到 2.20031016 或以上。
3. 在 Android Studio 中的当前工程下，点击菜单 **mPaaS** > **基线升级**，选择 **10.1.68**，并点击 **OK**。
4. 升级成功后，查看 `mpaas_packages.json` 文件，如果 `base_line` 字段是 `10.1.68` 即表示升级成功。

Inside 接入方式下的升级指南

如果您已有基于 Inside 接入方式的工程，请按照以下步骤完成升级。

1. 环境配置。

```
gradle = 6.2 // 需使用 6.2 及以上版本
com.android.tools.build:gradle:3.5.3
com.alipay.android:android-gradle-plugin:3.5.18
com.android.boost.easyconfig:easyconfig:2.8.0
```

2. 参考 更新 mPaaS 插件 文档，升级 Android Studio mPaaS 插件到 2.20031016 或以上。
3. 在 Android Studio 中的当前工程下，点击菜单 **mPaaS** > **基线升级**，选择 **10.1.68**，并点击 **OK**。
4. 升级成功后，查看 `mpaas_packages.json` 文件，如果 `base_line` 字段是 `10.1.68` 即表示升级成功。

组件化接入方式（Portal Bundle）下的升级指南

如果您已有基于 Portal&Bundle 接入方式的工程，请按照以下步骤完成升级。

1. 环境配置。

```
gradle = 4.4
com.android.tools.build:gradle:3.0.1
com.alipay.android:android-gradle-plugin:3.0.0.9.13
com.android.boost.easyconfig:easyconfig:2.8.0
```

2. 请参考 更新 mPaaS 插件 文档，升级 Android Studio mPaaS 插件到 2.20031016 或以上。
3. 在 Android Studio 中的当前工程下，点击菜单 **mPaaS** > **基线升级**，选择 **10.1.68**，并点击 **OK**。
4. 升级成功后，查看 `mpaas_packages.json` 文件，如果 `base_line` 字段是 `10.1.68` 即表示升级成功。

升级到最新的 Gradle 插件

目前 Google 官方提供的 Android Gradle Plugin 是 3.5.x 版本。mPaaS 也提供了 3.5.x 版本的插件作为适配，可支持 Google Android Gradle Plugin 3.5.3 和 Gradle 6.0 的 API。您可根据需要，参考 升级到最新的 Gradle 插件 文档升级 Gradle 插件。

组件管理变更

在更新至 10.1.68 之后，以下组件发生了变更，如您之前有选择这些组件，则需要按照以下改动重新操作。更多信息，请参考 组件管理。

- **FRAMEWORK 框架** 已变更为可选项。
- **MAP 地图** 已变更为 **TINYAPP-MAP 小程序地图**。
- **TINYPROGRAM 小程序** 已变更为 **TINYAPP 小程序**。
- **MINIPROGRAM-BLUETOOTH 小程序蓝牙** 已删除，默认合并至 **TINYAPP 、小程序** 中。
- **MINIPROGRAM-MEDIA 小程序多媒体** 已变更为 **TINYAPP-MEDIA 小程序多媒体**。
- **TINYVIDEO 小程序视频** 已删除，目前暂时不提供小程序视频。
- 新增 **UCCORE UC内核**，之前如果您用到 UC 内核，例如用到了 H5 容器或是小程序，请手动添加该组件。

组件使用升级指南

H5 容器

从 10.1.68 基线开始自定义标题栏的使用方法有了变化，更多信息请参见 自定义导航栏（10.1.68）。

UC 内核

在 10.1.68 基线中对 UC 内核进行了升级，请全面回归前端页面内容等相关部分，以免出现兼容性问题。

组件 API 变更

H5 容器

H5TitleView

H5TitleView 新增了部分接口，更多信息请参见 自定义导航栏（10.1.68）。

MPNebula

新增接口，增加 `MicroApplication app` 参数。

```
 /**
 * 启动在线 url
 *
 * @param app micro app
 * @param url 在线地址
 */
public static void startUrl(MicroApplication app, String url)

 /**
 * 启动在线 url
 *
 * @param app   micro app
 * @param url   在线地址
 * @param param 启动参数
 */
public static void startUrl(MicroApplication app, String url, Bundle param)
```

扫一扫

在 Inside 或 AAR 模式下，如未接入框架，需改用 MPScan 以下方法启动扫一扫标准 UI：

```
startMPaasScanActivity(Activity activity, ScanRequest scanRequest, ScanCallback scanCal
lback);
```

参数和原 ScanService 一致。

3.3. mPaaS 10.1.60 升级指南

关于 mPaaS 10.1.60 正式版

- 10.1.60 基线已适配 **Android 10**。
- 10.1.60 基线新增加了 **小程序组件** 正式版。小程序正式版拥有更加完善的 API，且在稳定性、兼容性等方面有了大幅提高。关于小程序升级请参见 小程序升级说明，关于小程序 IDE 新增调试、预览、发布等功能的详情请参见 小程序 IDE。
- 10.1.60 基线对 **H5 容器** 整体进行大幅优化，提供了更加简化的接入流程，持续补强能力，在兼容性、稳定性等方面有显著提高。关于 H5 容器和离线包升级，请参见 H5 容器升级说明。
- 10.1.60 基线中，消息推送新增加了对 OPPO 和 vivo 渠道推送的支持。
- 10.1.60 基线新增加了对 **社交分享** 组件的管理支持，提供了简化的接入流程。关于社交分享的升级，请参见 迁移到 10.1.60 基线。
- 10.1.60 基线新增加 **智能投放** 组件。智能投放提供了在应用内个性化投放广告的能力，支持针对定向人群进行个性化广告投放，帮助 APP 运营人员精准、及时触达用户，详情请参见 智能投放。
- 10.1.60 基线的整体组件的兼容性、稳定性都有了大幅提高，功能也有着显著提升，具体的发布说明请参见 Android SDK 发布说明。

mPaaS 10.1.60 正式版升级指南

操作步骤

1. 升级 Android Studio mPaaS 插件到 v2.19123015 或以上。关于更新 mPaaS 插件，参见 更新 mPaaS 插件。
2. 在 Android Studio 中的当前工程下，点击菜单 **mPaaS** > **基线升级**，选择 **10.1.60**，并点击 **OK**。
3. 升级成功后，查看 mpaas_packages.json 中，”base_line” 字段是 10.1.60 即表示升级成功。

> **说明**
>
> 10.1.60-beta 基线转为正式版也需要按上述操作。

组件使用升级指南

10.1.60 基线中的 H5 容器、小程序和社交分享组件在接入、使用等方面做了大幅调整。如您接入了上述组件，需详细阅读下列说明：

- 阅读 H5 容器升级说明 了解 H5 容器和离线包升级的更多信息。
- 阅读 小程序升级说明 了解小程序升级的更多信息。
- 社交分享 SDK 接入方式升级。阅读 迁移到 10.1.60 基线 了解 **社交分享** 组件升级的更多信息。

  说明：

  - 从 10.1.60 开始，分享 SDK 使用 mPaaS 插件进行管理。如果需要安装分享组件，请参见 迁移到 10.1.60 基线 进行操作。
  - 如未使用插件进行分享 SDK 的接入，则会导致分享 SDK 的升级与问题修复不能得到及时更新。

组件 API 变更

mPaaS 组件从 10.1.32 基线开始起添加了适配层，建议您使用含有适配层的 API，具体可参考以下各组件文档中的旧版本升级注意事项：

- 移动分析：
  - 新增适配器，简化使用，参见 自定义事件日志 。
  - 部分 API 废弃，您需使用新的 API，参见 移动分析，否则可能导致编译失败。
- 移动推送：新增适配器，简化使用，参见 移动推送 。
- 移动同步：新增适配器，简化使用，参见 移动同步 。
- 热修复：新增适配器，简化使用，参见 热修复 SDK。
- 版本升级：新增适配器，简化使用，参见 版本升级 。
- 开关配置：新增适配器，简化使用，参见 开关配置 。
- H5 容器：
  - 新增适配器，简化使用，参见 H5 容器 SDK 10.1.32。
  - 容器配置方法变更，如果升级前为 10.0.18 版本，您需使用新的容器配置方法，参见 容器配置 10.1.32，否则您的容器配置将无法生效。
  - 10.1.60 基线变更参考 升级说明 。
- 小程序：
  - 先进行 H5 容器升级。
  - 升级变更信息 升级说明 。

> **说明**
>
> 强烈建议您修改代码，使用中间层（适配器）方法而非直接使用底层方法，因为某些底层方法可能会在将来的版本中发生变更或废弃。如果您继续使用，在将来的更新中可能需要花费更多的时间进行适配。

定制依赖处理

查看所有 `build.gradle` 中 `dependencies` 的依赖配置，确认是否配置有 mPaaS 组件的 bundle 依赖。若有依赖，且是从低版本 SDK（例如 10.1.32）升级至 10.1.60 版本，您的定制库可能需要基于新版本重新定制，否则可能会出现不兼容等问题，请 提交工单 或联系 mPaaS 支持人员确认。

4.解决依赖冲突

4.1. 解决依赖冲突

由于 mPaaS 的产品依赖部分第三方 SDK，因此在接入 mPaaS 的过程中，可能会遇到项目中已集成的第三方库与 mPaaS SDK 产生冲突的问题。

为应对可能产生的冲突，mPaaS 提供了移除 mPaaS 内部第三方 SDK 的能力，具体移除方法您可以参考：

- 原生 AAR 方式
- 组件化接入方式

mPaaS 选择使用的版本具有较高的稳定性和安全性。如果您移除了 mPaaS 所依赖的第三方库，且您使用的 SDK 与 mPaaS 所使用的第三方 SDK 版本不同，请进行足够和充分的测试，以保证功能稳定。

如果您遇到了依赖冲突，请参考以下解决方案：

- 解决高德定位冲突
- 解决高德地图冲突
- 解决阿里巴巴无线保镖冲突
- 解决阿里巴巴 utdid 冲突
- 解决 wire/okio 冲突
- 解决 fastjson 冲突
- 解决 android support 冲突

4.2. 解决高德定位冲突

mPaaS 内置了高德定位 SDK。如果您的应用因需要在 Google Play 应用市场上线而同时集成高德官方提供的能通过谷歌审核的版本 SDK ，将会遇到高德定位冲突的情况。

> **重要**
>
> 10.1.32 基线不支持开发者集成定位 SDK，不存在此冲突。

解决办法

移除 mPaaS 内置的高德定位 SDK。

操作步骤

1. 确认 mPaaS 所使用的高德定位 SDK 版本，以便您选取相同或相近的过审版本。

```
'com.alipay.android.phone.mobilecommon:AMapSearch:6.1.0_20180330@jar'
'com.alipay.thirdparty.amap:amap-location:4.7.2.20190927@jar'
```

2. 获取 mPaaS 使用的高德定位 SDK 的 `group:artifact` 信息。

```
'com.mpaas.group.amap:amap-build'
```

3. 移除 mPaaS 高德定位 SDK。
   - AAR 方式

```
configurations {
all*.exclude group:'com.mpaas.group.amap', module: 'amap-build'
}
```

- 组件化（Portal & Bundle）

```
mpaascomponents {
excludeDependencies = [
   "com.mpaas.group.amap:amap-build"
]
}
```

4.3. 解决高德地图冲突

冲突说明

mPaaS 内置了高德地图 SDK。如果您的应用因为需要在 Google Play 应用市场上线而同时集成高德官方提供的能通过谷歌审核的版本 SDK 的话，将会出现高德地图冲突的情况。

解决办法

移除 mPaaS 内置的高德地图 SDK。

操作步骤

1. 确认 mPaaS 所使用的高德地图 SDK 版本，以便您选取相同或相近的过审版本。

```
'com.alipay.android.phone.mobilecommon:AMap-2DMap:5.2.1_20190114@jar'
```

2. 获取 mPaaS 使用的高德地图 SDK 的 `group:artifact` 信息。

```
'om.alipay.android.phone.thirdparty:amap3dmap-build'
```

3. 移除 mPaaS 高德地图 SDK。
   - AAR 方式：

```
configurations {
all*.exclude group:'com.alipay.android.phone.thirdparty', module: 'amap3dmap-build'
}
```

   - 组件化（Portal & Bundle）：

```
mpaascomponents {
excludeDependencies = [
   "com.alipay.android.phone.thirdparty:amap3dmap-build"
]
}
```

4.4. 解决无线保镖冲突

冲突说明

如果在使用 mPaaS 的同时也使用了其他阿里系 SDK，那么可能会出现存在无线保镖冲突（SecurityGuardSDK）的情况。

解决办法

mPaaS 提供移除 mPaaS 无线保镖库，使用其他阿里系 SDK 提供的保镖库。

操作步骤

1. 确认当前 mPaaS 所使用的无线保镖 SDK 的版本，以便选取相同或相近的其他阿里系保镖库。

```
'SecurityGuardSDK-without-resources-5.4.2009'
```

2. 确认 mPaaS 使用的无线保镖 SDK 的 `group:artifact` 信息。

```
'com.alipay.android.phone.thirdparty:securityguard-build'
```

3. 移除 mPaaS 无线保镖。
   - AAR 方式

```
configurations {
all*.exclude group:'com.alipay.android.phone.thirdparty', module: 'securityguard-bu
ild'
}
```

   - mPaaS Inside & 组件化（Portal & Bundle）

```
mpaascomponents {
excludeDependencies = [
   "com.alipay.android.phone.thirdparty:securityguard-build"
]
}
```

4. 解决图片冲突。
   i. config 中增加图片后缀并编译。在 config 文件中加入 `"authCode": "1234"` ，其中， `1234` 可以为任意字符串，建议使用 4 位数字。

```
{
"appId":"xxx",
"appKey":"xxx",
"base64Code":"xxx",
"packageName":"xxx",
"rootPath":"xxx",
"workspaceId":"xxx",
"rpcGW":"xxx",
"mpaasapi":"xxx",
"pushPort":"xxx",
"pushGW":"xxx",
"logGW":"xxx",
"syncport":"xxx",
"syncserver":"xxx",
"authCode": "1234"
}
```

ii. 验证图片后缀是否生效。通过反编译，查看生成的 apk 中是否在 drawable 中存在 `yw_1222_1234.jpg` 图片，以及在 AndroidManifest 中是否含有如下信息。

```
<meta-data
android:name="security_guard_auth_code"
android:value="1234" />
```

> **说明**
>
> 解决图片冲突仅支持 10.1.32.7 及以上、10.1.60（beta 版需要 beta.7 及以上）和 10.1.68 基线版本。

4.5. 解决 utdid 冲突

冲突说明

如果您在使用了 mPaaS 的同时也使用了阿里系 SDK，可能会遇到 utdid 冲突。当您遇到此种情况，请参考以下解决方案。

解决办法

移除 mPaaS utdid 库，使用其他阿里系 SDK 提供的 utdid。

操作步骤

1. 确认 mPaaS 所使用的 utdid SDK 的版本，以便您选取相同或相近的版本。

```
'com.taobao.android:utdid4all:1.5.1.3@jar'
```

2. 获取 mPaaS 所使用的 utdid SDK 的 `group:artifact` 信息。

```
'com.alipay.android.phone.thirdparty:utdid-build'
```

3. 移除 mPaaS utdid SDK。
   - AAR 方式

```
configurations {
all*.exclude group:'com.alipay.android.phone.thirdparty', module: 'utdid-build'
}
```

   - 组件化（Portal & Bundle）

```
mpaascomponents {
excludeDependencies = [
   "com.alipay.android.phone.thirdparty:utdid-build"
]
}
```

4. 加入接口包。
   - 10.1.68.8 及以下基线如果您使用了 utdid 相关的 API，请下载 jar 包 utdid-build-1.1.5.3-api.jar.zip，并引入 (compile/implementation) 到工程参与编译。
   - 10.1.68.9 及以上基线无需任何操作。

4.6. 解决支付宝支付 SDK 冲突

冲突说明

如果您在使用 mPaaS 的同时也使用了支付宝支付 SDK，在部分情况下会出现库冲突的情况。

解决办法

如您遇到支付宝支付 SDK 冲突。

- 如果您的基线版本是 10.2.3.6 及以上，请添加如下配置。

```
configurations {
 all*.exclude group:"com.mpaas.android.anotations", module:"anotations-build"
}
```

- 如果是其他基线版本，请使用以下解冲突支付 SDK 版本。

```
dependencies {
  ...
 implementation 'com.alipay.sdk.android:alipaysdk-mpaas:15.8.03.210526122749'
  ...
}
```

4.7. 解决 wire/okio 冲突

冲突说明

由于 mPaaS 使用 wire/okio 来进行 RPC 网络连接，而 okhttp 也需要引用 okio，所以当您在使用 mPaaS 的同时使用了 okhttp，那就可能出现 wire/okio 冲突。

解决方法

10.1.68 基线

移除 mPaaS 的 wire/okio 依赖，并对 移动网关 功能进行回归测试以确保功能正常。操作步骤如下：

1. 确认 mPaaS 所使用的 wire/okio 版本。

```
'com.squareup.okio:okio:1.7.0@jar'
'com.squareup.wire:wire-lite-runtime:1.5.3.4@jar'
```

2. 获取 mPaaS 第三方 SDK 的 `group:artifact` 信息。

```
'com.alipay.android.phone.thirdparty:wire-build'
```

3. 移除 mPaaS 库。
   - 如果您采用原生 AAR 方式接入 mPaaS，gradle 的依赖传递会自动使用较高的版本，无需主动移除。通常来说 mPaaS 选择使用的版本具有较高的稳定性和安全性，建议使用 mPaaS 提供的版本。如果版本不一致，请在上线前对 mPaaS 功能进行测试以保证稳定性。
   - 如果您采用 mPaaS Inside 或组件化（Portal & Bundle）方式接入 mPaaS，需执行如下操作：

```
mpaascomponents {
excludeDependencies = [
  "com.alipay.android.phone.thirdparty:wire-build"
]
}
```

4. 加回 wire 或 okio（使用公网的 wire/okio，原生 AAR 方式接入方式无需关注）。因为 mPaaS 把 wire 和 okio 的依赖，都写在 `com.alipay.android.phone.thirdparty:wire-build` 库内，所以您需要根据实际情况，选择性加回。
   - 如果只是 okio 冲突，但不存在 wire 冲突，需要加回 wire。

     ```
     implementation 'com.squareup.wire:wire-lite-runtime:1.5.3.4@jar'
     ```

   - 如果只是 wire 冲突，但不存在 okio 冲突，需要加回 okio。

     ```
     'com.squareup.okio:okio:1.7.0@jar'
     ```

10.2.3 基线

完全移除 mPaaS 的版本依赖，使用业务本身需要的版本。解决 wire/okio 冲突，需要执行以下几步操作：

1. 去除 mPaaS 中的 wire，目前 mPaaS 内部不强依赖 wire。
   - 原生 AAR 项目中需执行如下操作：

     ```
     configurations {
      all*.exclude group: 'com.alipay.android.phone.thirdparty', module: 'wire-build'
     }
     ```

   - mPaaS Inside & 组件化（Portal & Bundle）项目中需执行如下操作：

     ```
     mpaascomponents {
             excludeDependencies = [
                     "com.alipay.android.phone.thirdparty:wire-build"
                 ]
     }
     ```

2. 所有业务方 rpc 的 pb 类继承 `com.squareup.wire.Message`，需要改成继承 `com.mpaas.thirdparty.squareup.wire.Message`。

   以下组件功能需要回归：
   - 移动网关
   - 消息推送
   - 数据同步
   - 热修复
   - 开关配置管理
   - 智能投放

4.8. 解决 fastjson 冲突

冲突说明

mPaaS 使用 fastjson 来进行 JSON 解析，如果您在项目中也使用了 fastjson，就会出现 fastjson 冲突。

解决办法

移除 mPaaS 中的 fastjson-build。

操作步骤

1. 确认当前 mPaaS 所使用的 fastjson 版本。

```
'com.alibaba:fastjson:1.x.x.android@jar'
```

2. 获取 mPaaS 使用的第三方 SDK 的 `group:artifact` 信息。

```
'com.alipay.android.phone.thirdparty:fastjson-build'
```

3. 移除 mPaaS 库。
   - AAR 方式如果您是原生 AAR 方式接入 mPaaS，则无需主动移除，gradle 依赖传递会自动使用较高的版本。mPaaS 选择使用的版本具有较高的稳定性和安全性，建议使用 mPaaS 提供的版本。如果版本不一致，请在上线前对 mPaaS 功能进行测试以保证稳定性。
   - 组件化（Portal & Bundle）

```
mpaascomponents {
excludeDependencies = [
  "com.alipay.android.phone.thirdparty:fastjson-build"
]
}
```

4.9. 解决 android support 冲突

组件化（Portal&Bundle）和 mPaaS Inside 接入方式下的 android support 冲突

冲突说明

mPaaS 内置了基于 23.2.1 版本的 support 库，同时添加了 Fragment 切面逻辑进行自动化页面埋点。如果在使用 mPaaS 的同时也添加了官方版本的 android support 库的话，会出现 android support 冲突。

解决方法

移除 androidsupport-build，直接替换为官方版本。如果还需要使用 mPaaS 提供的 Fragment 自动化日志功能，您需要手动添加 监控逻辑。

> **说明**
>
> 原生 AAR 方式并没有内置 support 库，因此您无需做任何处理。但如果您需要使用 mPaaS 提供的 Fragment 自动化日志功能，请手动添加 监控逻辑。

操作步骤

1. 确认当前 mPaaS 所使用的 android support 版本。

```
'com.android.support:support-v4'
'com.android.support:appcompat-v7'
```

2. 获取 mPaaS 第三方 SDK 的 `group:artifact` 信息。

```
'com.alipay.android.phone.thirdparty:androidsupport-build'
'com.alipay.android.phone.thirdparty:androidsupportrecyclerview-build'
```

3. 移除 mPaaS 库。
   - AAR 方式如果您是原生 AAR 方式接入 mPaaS，无需主动移除。
   - mPaaS Inside & 组件化（Portal & Bundle）

```
mpaascomponents {
excludeDependencies = [
   "com.alipay.android.phone.thirdparty:androidsupport-build"
]
}
```

原生 AAR 接入方式下的 Android support 冲突

冲突说明

原生 AAR 接入方式使用了基于 23.4.0 版本的 support-v4 库。从 24.2.0 起，Google 更改了代码组织方式，不再以全家桶的方式提供 support-v4 库的所有模块，而 appcompat-v7 采用了全家桶的方式引入库的所有模块，更多详情请参见 支持库软件包 。因此，当您的工程使用了 appcompat-v7 包时，会和原生 AAR 所基于的 support-v4 库产生入 AAR 依赖冲突。

解决方法

手动引入高版本 support-v4，同时引入您需要的 appcompat-v7。

操作步骤

1. 主动引入高版本 support-v4。

```
implementation 'com.android.support:support-v4:(您使用的版本，比如 28.0.0)'
```

2. 引入您需要的 appcompat-v7。

```
implementation 'com.android.support:appcompat-v7:(您使用的版本，比如 28.0.0)'
```

4.10. 解决 libc++_shared.so 冲突

冲突说明

mPaaS 部分组件依赖了 `libc++_shared.so` ，如果您集成的其他三方 SDK 也包含该 so 时会产生冲突。

解决办法

移除 mPaaS 内置的 `libc++_shared.so` 。

操作步骤

- AAR 接入方式

```
configurations {
  all*.exclude group:'com.mpaas.commonlib', module: 'libcshared-build'
}
```

- 组件化（Portal & Bundle）接入方式

```
mpaascomponents {
  excludeDependencies = [
      "com.mpaas.commonlib:libcshared-build"
  ]
}
```

4.11. 解决 libstlport_shared.so 冲突

冲突说明

mPaaS 部分组件依赖了 `libstlport_shared.so` ，如果您集成的其他三方 SDK 也包含该 so 时会产生冲突。

解决办法

移除 mPaaS 内置的 `libstlport_shared.so` 。

操作步骤

- AAR 接入方式

```
configurations {
  all*.exclude group:'com.alipay.android.phone.wallet', module: 'basicstl-build'
}
```

- 组件化（Portal & Bundle）接入方式

```
mpaascomponents {
  excludeDependencies = [
      "com.alipay.android.phone.wallet:basicstl-build"
  ]
}
```

4.12. 解决 libcrashsdk.so 冲突

冲突说明

如果在使用 mPaaS 的同时也使用了其他第三方的 SDK，如友盟 SDK，会导致 libcrashsdk.so 冲突。

```
[ERROR] :more than one file named : libcrashsdk.so in below files
C:\Users\Administrator\.gradle\caches\modules-2\files-2.1\com.mpaas.uc.crash\uccrash-bu
ild\1.0.0.201221171651\d347c79b8091adc68c33e1ca04b702b1c85888ca\uccrash-build-1.0.0.201
221171651.jar
C:\Users\Administrator\.m2\repository\com\xinmei\etrust\bundleone\bundleone-build\1.0.0
\bundleone-build-1.0.0-raw.jar
```

解决办法

移除 mPaaS UC 内核里的 libcrashsdk.so。

操作步骤

1. 确认 mPaaS 所使用的 UC 内核里 libcrashsdk 的版本，以便您选取相同或相近的过审版本。

```
'com.mpaas.uc.crash:uccrash-build:10.1.60a.00001878@jar'
```

2. 获取 mPaaS 使用的 SDK 的 **`group:artifact`** 信息。

```
'com.mpaas.uc.crash:uccrash-build'
```

3. 移除 mPaaS UC 内核里的 libcrashsdk。
   - 使用 AAR 的接入方式时：

```
configurations {
    all*.exclude group:'com.mpaas.uc.crash', module: 'uccrash-build'
}
```

- 使用 mPaaS Inside 或组件化（Portal&Bundle）的接入方式时：

```
mpaascomponents {
    excludeDependencies = [
       "com.mpaas.uc.crash:uccrash-build"
    ]
}
```

4.13. 解决 libcrashsdk.so 冲突

冲突说明

如果在使用 mPaaS 的同时也使用了其他第三方的 SDK，如友盟 SDK，会导致 libcrashsdk.so 冲突。

```
[ERROR] :more than one file named : libcrashsdk.so in below files
C:\Users\Administrator\.gradle\caches\modules-2\files-2.1\com.mpaas.uc.crash\uccrash-bu
ild\1.0.0.201221171651\d347c79b8091adc68c33e1ca04b702b1c85888ca\uccrash-build-1.0.0.201
221171651.jar
C:\Users\Administrator\.m2\repository\com\xinmei\etrust\bundleone\bundleone-build\1.0.0
\bundleone-build-1.0.0-raw.jar
```

解决办法

移除 mPaaS UC 内核里的 libcrashsdk.so。

操作步骤

1. 确认 mPaaS 所使用的 UC 内核里 libcrashsdk 的版本，以便您选取相同或相近的过审版本。

```
 'com.mpaas.uc.crash:uccrash-build:10.1.60a.00001878@jar'
```

2. 获取 mPaaS 使用的 SDK 的 `group:artifact` 信息。

```
'com.mpaas.uc.crash:uccrash-build'
```

3. 移除 mPaaS UC 内核里的 libcrashsdk。
   - 使用 AAR 的接入方式时：

```
configurations {
    all*.exclude group:'com.mpaas.uc.crash', module: 'uccrash-build'
}
```

   - 使用 mPaaS Inside 或组件化（Portal&Bundle）的接入方式时：

```
mpaascomponents {
    excludeDependencies = [
       "com.mpaas.uc.crash:uccrash-build"
    ]
}
```

5.开发者工具

5.1. Android Studio mPaaS 插件

5.1.1. 关于 mPaaS 插件

mPaaS 插件是一个具有图形化界面的插件工具，该工具提供了编译打包、管理组件依赖、热修复、加密图片等功能，用于帮助开发者能够快速接入 mPaaS 并辅助进行开发工作。安装 mPaaS 成功后我们会在 Android Studio 的顶部菜单栏看到 **mPaaS** 菜单 。

mPaaS 插件提供多种开发辅助功能，包括：

菜单		功能
原生 AAR 接入		协助通过原生 AAR 接入方式将工程接入 mPaaS。
组件化接入		协助通过组件化接入方式将工程接入 mPaaS。
基础工具	热修复	为已经添加了热修复的组件生成热修补丁。
	生成加密图片（专有云配置文件）	生成包含了加解密密钥信息的加密图片。
	生成蓝盾图片	mPaaS 提供蓝盾能力作为无线保镖的替代方案。
	生成控制台用签名 APK	在只输入签名相关的参数的前提下，生成签名后的 APK，用来在 mPaaS 控制台中获取配置文件。
	生成 UC Key 签名信息	为申请 UC SDK 的 Key 生成签名信息。
	日志诊断工具	分析 Android Studio 的日志，快速定位编译问题。
	常见问题	跳转至 文档中心，查看接入 Android 过程中的常见问题。
帮助	查看文档	转至 mPaaS 文档中心。
构建		构建工程。

相关链接

- 安装 mPaaS 插件：在Android Studio 中如何安装 mPaaS 插件。
- 使用 mPaaS 插件：mPaaS 插件的各个功能的使用简介。

- 更新及卸载 mPaaS 插件：对 mPaaS 插件进行更新或卸载。

5.1.2. 安装 mPaaS 插件

mPaaS 插件提供多种开发辅助功能，包括：新建 mPaaS 工程，添加、删除和升级 mPaaS 组件，构建工程等。为方便您使用以上功能，本文将向您介绍 mPaaS 插件的安装过程。

mPaaS 插件的安装有 **在线安装** 和 **离线安装** 两种安装方式。

- 如果您的 Android Studio 为 4.0 及之后的版本，可通过 **在线安装** 或 **离线安装** 最新版本的 mPaaS 插件。
- 在 mPaaS 的 JetBrains 主页 上，您可以获取 mPaaS 插件的部分离线安装包。

在线安装

操作步骤

1. 在 Android Studio 中，通过 **Android Studio** > **Preferences** 打开偏好窗口。如您使用的是 Windows 系统，则通过 **File** > **Settings** 打开设置对话框。
2. 点击左侧 **Plugins**，然后点击窗口顶部的 **Marketplace**。
3. 以 **mPaaS** 为关键字进行搜索，在搜索结果中，点击搜索到的 mPaaS 插件下方的 **Install** 按钮安装插件。
4. 安装结束后，根据提示重启 Android Studio，即可在菜单栏看到 mPaaS 菜单。

离线安装

前提条件

您已经获得了 mPaaS 插件的安装包。

操作步骤

> **说明**
>
> **mPaaS 插件的安装包为压缩包文件形式，在安装前无需解压。**

1. 在 Android Studio 中，通过 **Android Studio** > **Preference** 打开设置对话框。
2. 点击左侧 **Plugins**，然后点击右上方的 ⚙，并在下拉菜单中选择 **Install Plugin from Disk**。
3. 选择您已经下载到的 mPaaS 插件安装包，点击 **确认** 即可安装。安装完成后，根据提示重启 Android Studio，您就可以开始使用 mPaaS 插件了。

5.1.3. 使用 mPaaS 插件

mPaaS 插件通过图形化界面，帮助您快速地接入 mPaaS 并便捷地使用 mPaaS 的功能。

mPaaS 插件的功能主要包括 原生 AAR 方式、组件化接入、**基础工具**、**帮助** 和 **构建**。

- **原生 AAR 接入**、**组件化接入** 分别提供了接入面板，在接入面板中的接入向导会协助您将 mPaaS 以不同的接入方式接入到您的工程中。在完成接入后，您还可以在接入面板进行 **基线升级** 和 **组件管理**。
- 在 **基础工具** 中，mPaaS 提供了 **热修复**、**生成加密图片（专有云配置文件）**、**生成 UC Key 签名信息** 和 **生成蓝盾图片** 的功能，方便快速完成使用 mPaaS 功能前的信息准备。
- 在 **帮助** 中，mPaaS 提供了 **日志诊断工具**、**常见问题**、和 **查看文档** 的功能，方便在使用 mPaaS 过程中遇到问题时快速获得支持。
- **构建**，完成接入 mPaaS 后构建工程。

添加配置文件

接入过程的主要工作是将配置文件添加到工程中，mPaaS 插件支持 **手动导入** 的方式添加配置文件。手动导入需要在控制台下载配置文件后，再通过 mPaaS 插件手动添加到工程里。

手动导入

前提条件

- 已创建了蚂蚁或阿里云账号并开通了 mPaaS 服务。
- 已在 mPaaS 控制台创建应用。更多关于创建应用的信息，请参见 在控制台创建 mPaaS 应用 。
- 已有一个Android 开发工程。

操作步骤

1. 在 Android Studio 中打开已有工程，单击 **mPaaS** > **原生 AAR 接入** 或 **组件化接入**。在弹出的接入面板中，单击 **导入 App 配置** 下的 **开始导入**。
2. 选择 **我已经从控制台上下载配置文件（Ant-mPaaS-xxxx.config），准备导入到工程** ，单击 **Next**。
3. 选择配置文件后，单击 Finish，即完成了配置文件的导入。导入成功后，将会收到导入配置文件成功的提示信息。

AAR 接入

操作步骤

1. 在 Android Studio 中打开已有工程，单击 **mPaaS** > **原生 AAR 接入**。
2. 导入 App 配置。在接入面板中，单击 **开始导入**，使用 手动导入 的方式完成配置文件的添加。

后续步骤

1. 接入/升级基线
2. 配置/更新组件

组件化接入

操作步骤

1. 在 Android Studio 中打开已有工程，单击 **mPaaS** > **组件化接入**。
2. 导入 App 配置。在接入面板中，单击 **开始导入**，使用 手动导入 的方式完成配置文件的添加。
3. 转换工程。如果您的工程是原生 Android 工程，还需要对工程进行转换。在接入面板中，单击 **安装 mPaaS Portal**。在安装 mPaaS Portal 窗口中，分别选择原始工程的位置和配置文件，单击 **OK**。

后续步骤

1. 接入/升级基线
2. 配置/更新组件

接入/升级基线

升级到常规基线

操作步骤

1. 单击 **mPaaS** > **原生 AAR 接入** 或 **组件化接入**，在弹出的接入面板中，单击接入/升级基线下的 **开始配置**。
2. 选择需要升级的基线版本，单击 **OK**。 升级成功后，您将看到 **基线升级成功** 的提示。

后续步骤

单击接入面板中的升级基线，在选择基线窗口中将会看到您的基线版本号。

升级到自定义基线

通常情况下，我们提供的基线面向所有客户，如 10.1.32、10.1.60、10.1.68。当您需要定制 mPaaS 的功能时，您可以向和您对接的 mPaaS 的工作人员提出需求，我们会按照您的需求为您定制基线。在交付时，mPaaS 的工作人员会向您提供定制基线的 ID，您只需要在 mPaaS 插件中填写该 ID，即可获得此定制基线。

前提条件

确认您的 Android Studio mPaaS 版本为 V2.19111217 或以上。您可以参考 更新 mPaaS 插件 以了解当前的 mPaaS 插件版本和如何升级 mPaaS 插件。

操作步骤

1. 删除您 Android Studio 工程里已经存在的 mpaas_package.json 文件。
2. 单击 **mPaaS** > **原生 AAR 接入** 或 **组件化接入**，在弹出的接入面板中，单击接入/升级基线下的 **开始配置**。
3. 在基线升级对话框中勾选 **自定义基线** 并输入您得到的定制基线 ID。
4. 单击 **OK**，即完成自定义基线的引入。

配置/更新组件

mPaaS组件管理（AAR）

前提条件

您已完成基线升级。

操作步骤

1. 单击 **mPaaS** > **原生 AAR 接入**，在弹出的接入面板中，单击配置/更新组件下的 **开始配置**。
2. 在弹出的管理窗口中，单击 **mPaaS 组件管理**，选择要进行管理的 module，勾选要添加的组件，单击 **OK**。如果您的工程中有多个 module，您可以在选择不同的 module 后，分别为其添加组件。
3. 组件添加完成后，单击 **OK**。

组件管理

操作步骤

1. 单击 **mPaaS** > **组件化接入**，在弹出的接入面板中，单击配置/更新组件下的 **开始配置**。
2. 在弹出的组件管理窗口中，单击按钮安装需要的组件。

基础工具

基础工具中包含 **热修复**、**生成加密图片（专有云配置文件）**、**生成控制台用签名 APK**、**生成 UC Key 签名信息**、**生成蓝盾图片** 等功能。

热修复

使用热修复能力前，首先要让 App 具备热修复的能力。更多详情，请参见 热修复管理：接入 Android——热修复。

生成热修复补丁

使用 mPaaS 插件的 **生成热修复补丁**，通过以下步骤生成热修复包：

1. 针对不同的 mPaaS 集成方式，选择对应的包，通过 mPaaS 插件的 **生成热修复补丁** 生成热修复包。
   - 如果是 **原生 AAR 工程**，需要准备有 bug 的线上 `APK` 包和修复后的 `APK` 包。mPaaS 插件会根据代码的不同，生成热修复包。

提示：

- 在 **New bundle** 栏，填写修复后的 `APK` 包的本地地址。
- 在 **Old bundle** 栏， 填写有 bug 的 `APK` 包的本地地址。
- 在 **白名单** 一栏输入白名单。

- 如果是 **组件化（Portal&Bundle）工程**，需要准备 **正在使用的有 bug 的 bundle 包** 和 **修复后的 bundle 包**。

  提示：

  - bundle 的输出路径为 bundle 的主 module 目录下的 `build/intermediates/bundle/xxxx-raw.jar` 。如果是 release 包则没有 `-raw` 。
  - **New bundle**：选择修复 bug 的包路径。
  - **Old bundle**：选择有 bug 的包路径。
  - **白名单**：选择用于指定修复的类的 `.txt` 格式的配置文件。该配置文件的编写规则见下文 **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**：输出的 patch 包路径。
  - **是否使用 dexPatch**：选择是否使用 dexPatch 热修复方式。mPaaS 插件的 **生成热修复补丁** 功能支持 Andfix 和 dexPatch 两种热修复方式。不论使用哪种热修复方案，在发版本前都需要进行验证，查看热修复包是否能生效。
    - 不勾选时，会生成 Andfix 热修复包。Andfix 热修复包能够立即生效，不需要重启应用；但因机型问题，修复的场景限制较多。
    - 勾选时，会生成 dexPatch 热修复包。dexPatch 热修复包不能立即生效，需要杀掉进程后才能生效；但其能够修复的场景比 Andfix 热修复包多，且机型适配问题少。

2. 输入签名信息生成热修复包。

> **重要**
>
> 生成热修复包所需要的签名文件必须和运行的 `APK` 的签名文件保持一致，并且签名文件和生成图片选择的 `APK` 的签名文件也要保持一致。生成的图片需要放在 Portal 工程的 `res/drawable` 文件夹下面，命名为 `yw_1222.jpg` 。

> **说明**
>
> **白名单配置文件编写规则**
>
> 打热修复包时用于指定修复的类的配置文件为 `.txt` 格式，该配置文件应包含并按顺序包含以下信息：
>
> i. 需要 Patch 的类。以 `L` 开头，后跟以混淆后真实类名。如果多个类，每行只可写一个。示例： `Lxxx.xxx.clazzX`
>
> ii. 设置 Patch 类型为 dexpatch。示例： `PatchType: dexpatch`
>
> iii. 设置是否是静态 Bundle。默认为 `true` 。示例： `HostDex: true`
>
> iv. 适配 Android 11。示例： `android-phone-mobilesdk-quinox-Configs: ForceEnableQSecondDex-true` 。 `android-phone-mobilesdk-quinox` 为 Bundle 名称。如果采用的是组件化 Portal&Bundle 接入方式，请按照 Bundle 名称更新； AAR 接入方式下没有 Bundle 工程，值就是 `android-phone-mobilesdk-quinox` 。

合并补丁

一个 Android App 版本最多只能有一个热修复包在运行。如果客户端某版本有两个 bug，那需要先在本地使用 **合并补丁** 功能将修正两个 bug 的热修复包合成一个热修复包。例如，针对某一个版本的 App 已经发过热修复包 A， 之后在这个版本上又发现了另外一个问题，这时可在本地生成另外一个热修复包 B，然后合并 A 与 B 两个热修复包，最后下发到客户端。

> **说明**
>
> 本节仅针对使用组件化（Portal&Bundle）工程的用户，使用原生 AAR 工程的用户可以在原先修复的基础上，相对于最原始未修复的包再打出一个补丁发布。

操作步骤

1. 填写热修复包文件夹地址。
   - **Merge dir**：选择合并的 hotpatch 包目录。文件夹下面是所有的需要合并的包，包名需要以 `.jar` 或者 `.apk` 结尾。
   - **Patch file dir**：选择合并之后输出的热修复包目录。
2. 配置签名信息。

合并完成后，将会收到合并成功的提醒。

生成加密图片（专有云配置文件）

为了安全，mPaaS 某些组件访问网络时需要对内容进行加密。

- 具有特殊名称 `yw_1222.jpg` 的图片为加解密提供密钥信息。mPaaS 组件自动使用该图片进行加解密，无需额外操作。
- 由于公有云环境已弃用该加密图片，故公有云用户可忽略本节内容。

生成并使用加密图片 `yw_1222.jpg` 的详情如下。

准备

加密图片与 APK 的签名文件有绑定关系。因此，需要准备 Portal 工程签名之后的 APK。具体的签名步骤，请参考 Android 官方网站：对应用进行签名 。

> **说明**
>
> - 此处的 APK 应和 **发布版本** 的 APK 使用相同的签名文件。
> - 生成的加密图片只能用在该 APK 工程中。

生成

您可以通过 **mPaaS 插件** 生成加密图片。

1. 在 Android Studio 中，单击 **mPaaS** > **基础工具** > **生成加密图片（专有云配置文件）** 。
2. 在 **Release Apk**， 选择 Portal 工程签名之后的 APK 文件， **RSA** 会自动填充。
3. 在 **mPaaS Config File**， 选择 Portal 工程的 `.config` 文件， **workSpaceId**、**appId** 和 **packageName** 会自动填充。如果没有，可以根据工程的 `.config` 文件中的配置填写到对应输入框中。
4. 填写 **appsecret** 。

   注意：作为服务端管理人员，您可以从控制台中查询 **appid** 所对应的 **appsecret**。
5. 在 **jpg Version** 栏，填写对应的无线保镖图片版本号。

> **说明**
> 查看 Portal 工程主 module 下 `build.gradle` 文件中 `securityguard` 版本，低于 5.4 的填 **4**（例如基线中给出的 `securityguard-build:5.1.38.180402194514` ），其余填 **5** 。

6. 在 **outPath**， 选择无线保镖图片 `yw_1222.jpg` 的输出路径，即加密图片生成的本地路径。
7. 单击 **OK** 生成加密图片。

使用

加密图片的使用步骤如下：

1. 将加密图片 `yw_1222.jpg` 存放到 Portal 工程的 `res/drawable` 文件夹中。
2. 如使用 ProGuard，需避免加密图片被混淆。
   i. 检查 `build.gradle` 中是否配置了如下内容：

```
minifyEnabled true
shrinkResources true
```

   ii. 若有如上配置，为了避免加密图片被混淆，需要在 `res/raw` 下创建 `keep.xml` 文件。文件内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<resources xmlns:tools="http://schemas.android.com/tools"
tools:keep="@drawable/yw_1222*" /><!--tools:discard="@layout/unused2"-->
```

生成蓝盾图片

> **说明**
> 如果您从 mPaaS 控制台下载的 `.config` 文件中的 `absBase64Code` 值为空，则需进行下面 **生成蓝盾图片** 的操作（适用于私有云场景）。

单击 mPaaS > 基础工具 > 生成蓝盾加密图片，输入相关信息，即可生成蓝盾图片。

重点输入项说明：

- **Release Apk：** 接入 mPaaS 的工程打包出的 release apk 包，需要进行签名。
- **MD5:** release apk 包上传之后会自动获取填入，即 apk 包的 `public md5 key` 。
- **mPaaS config File:** mPaaS 控制台点击下载配置即可下 `.config` 文件并传入。
- **appSecret:** mPaaS 控制台查看。
- 其他项 **appId**、**packageName**、**outPath** 传入以上信息后会自动识别填入。

最后将生成的图片添加到工程的 `assets` 目录下。

生成控制台用签名 APK

在 mPaaS 控制台中获取配置文件时，需要上传签名后的 APK 文件。但在未创建工程或未编译出签名后的 APK 时，获取配置文件的过程就会受阻而无法进行。为解决此问题，mPaaS 已将此过程简化为 Android Studio mPaaS 插件的 **生成控制台用签名 APK** 功能。该功能可在只需输入签名相关的参数的前提下，生成签名后的 APK。

生成

1. 单击 **mPaaS** > **基础工具** > **生成控制台用签名 APK**，进入 **构造签名 APK** 页面。

2. 在 **构造签名 APK** 页面，填写相关配置信息。
3. 单击 **OK** 即可生成签名后的 APK 文件。
4. 单击 **Reveal in Finder** 即可找到上述步骤生成的 APK 文件，文件名为 `mpaas-signed.apk` 。至此，签名后的 APK 生成成功。

   打开该 APK 文件查看，会发现该文件很小，且已经被签名。

生成 UC Key 签名信息

在 Android 应用中接入 UC SDK 能够有效解决各种厂商浏览器的兼容性问题。为添加 UC SDK，您需要先申请 UC SDK 的授权。该功能能够帮助您快速获得授权，下文介绍了申请 UC SDK 授权的全流程操作。

> **说明**
>
> 该功能自 V2.20062211 版本起加入。更多信息，请参见 V2.20062211 发布说明 。

操作步骤

1. 在工程中，添加 **UC 内核（UCCORE）** 依赖。
2. 提供应用的 Android native package 名称（package name）。
3. 单击 **mPaaS** > **基础工具** > **生成 UC Key 签名信息** ，进入 **查询签名信息** 页面。
4. 在 **查询签名信息** 页面，填写相关配置信息。单击 **Next**。
5. 复制获得的 SHA1 信息。
6. 填写 UC key 申请表 并提交。

   > **说明**
   >
   > 由于产品策略变更，UC 不再全面开放申请，从2022.12.01起不支持公开申请 UC Key。需要填写表单相关信息，工作人员会进行审核并反馈申请结果。

7. 将获取的 Key 填入 Portal 项目的 `AndroidManifest.xml` 文件中： `<meta-data android:name="UCSDKAppKey" android:value="您申请获得的 key"/>` 。

   > **说明**
   >
   > UC SDK 的授权信息与 APK 的 **包名** 以及 **签名** 绑定。因此，如果 UCWebView 没有生效，请检查签名和包名与申请时使用的信息是否一致。

帮助

日志诊断工具

1. 单击 **mPaaS** > **帮助** > **日志诊断工具** 。
2. 将您需要分析的日志信息 copy 到输入框，单击 **Next**。
3. 等待分析结果。
4. 查看分析结果。 分析结果中包含 **定位原因** 和 **解决方案** 的信息，您可以根据具体的定位信息和解决方案来修改您的代码。
5. 修改完成后单击 **Finish** 关闭弹窗。

常见问题

单击 **mPaaS** > **帮助** > **常见问题**，即可跳转至 接入 Android 常见问题，查看接入过程中的常见问题。

查看文档

单击 **mPaaS** > **帮助** > **查看文档**，即可跳转至 mPaaS 文档中心，查看各组件的使用文档。

构建

在 Android Studio 中选择 **mPaaS** > **构建**。即可构造工程。

5.1.4. 更新及卸载 mPaaS 插件

本文向您介绍如何更新及卸载 mPaaS 插件。

本文档中的截图是从 Windows 环境下获得，macOS 和 Linux 环境下操作流程与 Windows 环境下相似，此处不再赘述。

更新 mPaaS 插件

1. 打开 Android Studio，点击 **File** > **Settings**。
2. 在打开的 **Settings** 窗口的左侧导航栏，选择 **Plugins**。
3. 在窗口右侧，选择 **Updates** 选项卡，搜索并找到 mPaaS 插件，点击其右侧的 **Update**。
4. 在弹出的第三方插件隐私说明中，点击 **Accept**。
5. 随后，Android Studio 会自动下载 mPaaS 插件。
6. 更新完毕后，点击 **Restart IDE**。并在弹出的确认窗口中点击 **Restart** 重启 Android Studio。
7. 重启 Android Studio 后，重新进入 **File** > **Settings** > **Plugins**，您即可看到 mPaaS 插件已更新至最新版本。

卸载 mPaaS 插件

1. 打开 Android Studio，点击 **File** > **Settings**。
2. 在打开的 **Settings** 窗口的左侧导航栏，选择 **Plugins**。
3. 在窗口右侧的 **Installed** 选项卡中，搜索并找到 mPaaS 插件，点击插件名，进入插件详情页。
4. 在 mPaaS 插件详情页，点击窗口右上方 **Disable** 右侧的下拉箭头，选择 **Uninstall**。
5. 在弹出的确认窗口中，点击 **Yes** 即可卸载 mPaaS 插件。

6.Android 适配说明

6.1. mPaaS 10.1.68 适配 Android 12

本文介绍了用户在使用 mPaaS 10.1.68 版本基线时，需要为 Android 12 进行的适配工作。

谷歌已于 2021 年 10 月 4 日发布 Android 12 正式版。mPaaS 作为基础库，已在 10.1.68 基线上进行了适配。 10.1.68.37 及之后的版本已经完成了对 Android 12 的适配。在 mPaaS 适配之前，在 Android 12 设备上，mPaaS SDK 受到的影响为：**H5 容器无法启动 UC 内核** 。

升级 SDK

使用 接入/升级基线 来升级 mPaaS SDK 。

- 如果基线版本为 10.1.68，只需升级到最新版本即可。可参考 10.1.68 发布说明。
- 如果基线版本为 10.1.60 或以下版本，请升级至 10.1.68，并更新至最新版本。

启动 UC 内核

在 Android 12 系统上需要使用特定版本的 UC 内核，并添加配置来开启 UC 内核。若不进行以下适配，在 Android 12 系统上 H5 容器将默认启用系统 WebView。

使用特定版本的 UC 内核

在主 module （Protal&Bundle 接入方式下在 Portal 工程中）的 `build.gradle` 中的 `dependencies` 节点下添加依赖：

```
implementation('com.alipay.android.phone.wallet:nebulaucsdk-
build:999.3.22.2.30.211011154625@aar') {
    force = true
}
```

Protal&Bundle 接入方式还需要移除 SDK 中原本的 UC 内核，在主 module （Protal&Bundle 接入方式在 Portal 工程中）的 `build.gradle` 中添加以下内容：

```
mpaascomponents {
    excludeDependencies = [
            "com.alipay.android.phone.wallet:nebulaucsdk-build"
    ]
}
```

添加配置在 Android 12 上开启 UC 内核

在 assets 中 config 目录下创建 `custom_config.json` 文件，并在文件中添加以下内容：

```
[
  {
    "value":"{\"h5_enableExternalWebView\":\"YES\",\"h5_externalWebViewSdkVersion\":{\"
min\":11,\"max\":31}}",
    "key":"h5_webViewConfig"
  }
]
```

回归测试

升级 UC 内核可能会伴随部分浏览器特性而发生改动，请对使用 UC 浏览器的相关业务进行回归测试。

定制库处理

10.1.68 版本各组件并入了定制化的需求，如果您的依赖中包含定制库，则需要按以下情况处理：

- 如果您是从低版本 SDK（例如 10.1.60）升级至 10.1.68 版本，您的定制库可能需要基于新版本进行重新定制，请搜索群号 41708565 加入钉钉群咨询 mPaaS 支持人员。
- 如果您已使用 10.1.68 版本，则只需更新部分组件。参见下文的 适配 Android 12 更新的库清单，检查您的定制库是否包含在其中。
  - 如果不包含，您可继续使用该定制库。
  - 如果包含，您的定制库可能需要重新定制，请搜索群号 41708565 加入钉钉群咨询 mPaaS 支持人员。

适配 Android 12 更新的库清单

- nebulauc
- multimediabiz

6.2. mPaaS 10.1.68 适配 Android 11

背景

谷歌已于 2020 年 9 月 9 日正式发布 Android 11 系统。mPaaS 作为基础库，已在 10.1.68 基线上进行了适配。 10.1.68.14 及之后的版本已经完成了对 Android 11 系统的适配。在 mPaaS 适配之前，在 Android 11 设备上，mPaaS SDK 受到的影响为：**H5 容器无法启动 UC 内核** 。

升级 SDK/组件

使用 mPaaS 插件 来升级 mPaaS SDK 或组件。

- 如果使用的基线版本已经是 10.1.68，只需升级到最新版本即可。可参考 10.1.68 发布说明。
- 如果使用的基线版本为 10.1.60 或以下版本，请升级至 10.1.68，并更新至最新版本。目前暂无计划在 mPaaS 10.1.60 及以下版本中适配 Android 11 系统。

更新热修复配置

如果您在 10.1.68.14 之前版本的基线中已经使用了热修复，那么您需要更新热修复的白名单配置以适配 Android 11。详情请参见 白名单配置文件编写规则 。

定制库处理

10.1.68 版本各组件合入了定制化的需求，但是为了稳妥起见，如果此前您的依赖中包含定制库，则需要按以下情况处理：

- 如果您是从低版本 SDK（例如 10.1.60）升级至 10.1.68 版本，您的定制库可能需要基于新版本重新定制，请搜索群号 41708565 加入钉钉群进行咨询。
- 如果您已使用 10.1.68 版本，则只需更新部分组件。参见下文的 适配 Android 11 更新的库清单，检查您的定制库是否包含在其中。
  - 如果不包含，您可继续使用该定制库。
  - 如果包含，您的定制库可能需要重新定制，请搜索群号 41708565 加入钉钉群进行咨询。

适配 Android 11 更新的库清单

- nebulaappproxy
- nebulauc

6.3. mPaaS 支持多 CPU 架构

在 mPaaS 旧版基线中，SDK 使用的动态库（.so 文件）仅支持 armeabi 架构。但部分用户还有对其他 CPU 架构支持的需求，例如使用 armeabi-v7a 架构，或应用上架 Google Play 需支持 arm64-v8a 架构等。mPaaS 从 10.1.68.21 开始，增加了对 armeabi-v7a、arm64-v8a 架构的支持。如果您的应用需要支持 armeabi 以外的其他架构，请 使用 mPaaS 插件 将 SDK 更新到 10.1.68.21 或以上版本 ，并按照下文更新配置和回归相关功能。

如果您的应用不需要支持 armeabi 以外的其他架构，您仍然可以正常更新 SDK 到 10.1.68.21 或以上版本，并且不需要进行其他修改。

更新配置

整体兼容性

- 支持原生 AAR 和 Portal&Bundle 接入方式
- 支持 armeabi、armeabi-v7a、arm64-v8a 架构
- 支持 targetSdkVersion 26 - 29
- 支持 Android 11 系统

在 Google Play 发布

如果您的应用使用了 mPaaS 的定位组件或小程序中的地图功能，并需要在 Google Play 发布，那么您需要移除 mPaaS 内置的高德 SDK，并改用高德官方提供的能够通过 Google 审核的版本。请参照下文修改：

- 解决高德定位冲突
- 解决高德地图冲突

更新 Gradle 配置

原生 AAR

更新 Gradle 版本，推荐版本为 6.2，最低支持版本 5.0。如最新版本编译失败请使用推荐版本 6.2。

```
distributionUrl=https\://services.gradle.org/distributions/gradle-6.2-all.zip
```

Portal&Bundle

更新 Gradle 版本，推荐版本为 6.2，最低支持版本 5.0。如最新版本编译失败请使用推荐版本 6.2。

```
distributionUrl=https\://services.gradle.org/distributions/gradle-6.2-all.zip
```

更新 agp 版本：

对于 Portal&Bundle，在 Portal 工程和所有 Bundle 工程根目录 `build.gradle` 中修改。

```
classpath 'com.alipay.android:android-gradle-plugin:3.5.18'
classpath 'com.android.tools.build:gradle:3.5.3' // 最低 3.5.0
```

生成 APK

设置 CPU 架构

- 对于原生 AAR，在工程主 module 的 `build.gradle` 中设置。
- 对于 Portal&Bundle，若生成 APK 就在 Portal 工程主 module 的 `build.gradle` 中设置；若生成 Bundle 就在 Bundle 工程主 module `build.gradle` 中设置。

按照原生方式设置 abiFilters 即可：

```
ndk {
    abiFilters "armeabi", "armeabi-v7a", "arm64-v8a"
}
```

编译

无特殊配置，正常编译即可。

回归测试

您需要分别对不同架构的 APK 做全量回归测试。回归测试中您需要重点关注以下组件功能（如果使用）：

组件	验证项目
移动网关	• 开启签名校验（更多关于签名校验的详细信息，请参见 网关管理功能介绍）后，RPC 调用是否成功。 • 开启 数据加密 后，RPC 调用是否成功。
热修复	• 热修复 是否能够生效。
扫一扫	• 标准 UI 扫码是否成功。 • 标准 UI 打开手机相册、拍照及预览是否正常。 • 自定义 UI 扫码是否成功，如自定义 UI，需要 适配部分新接口 。
统一存储	• 数据库加密存储 是否正常。 • 文件加密存储 是否正常。
分享	• 分享到新浪微博、QQ 是否成功。
OCR	• OCR 识别相关内容是否正常。
音视频	• 音视频通话 功能是否正常。

6.4. mPaaS 适配 targetSdkVersion 30

mPaaS 主基线对 targetSdkVersion 最高仅支持到 29。如果您的应用需要将 targetSdkVersion 升级到 30，请通过 使用 mPaaS 插件将 SDK 更新到基线 `10.2.3` ，并按照下文进行适配和回归相关功能。

前置条件

已完成对 targetSdkVersion 28、29 的适配。更多信息，请先见 mPaaS 适配 targetSdkVersion 28、mPaaS 适配 targetSdkVersion 29。

操作步骤

1. 修改 targetSdkVersion。

- AAR 接入方式

  在工程主 module 下的 build.gradle 文件中修改属性 targetSdkVersion 为 30。

- Portal&Bundle 接入方式

  在 Portal 工程主 module 下的 build.gradle 文件中修改属性 targetSdkVersion 30。在 Bundle 工程中的 targetSdkVersion 可不修改，但不得高于 Portal 工程。

2. 通用配置。

   在工程（Portal&Bundle 接入方式为 Portal 工程）主 module 下的 build.gradle 文件中显式开启 v2 和 v1 签名：

```
android {
  ...
  signingConfigs {
    release {
      storeFile file("myreleasekey.keystore")
      storePassword "password"
      keyAlias "MyReleaseKey"
      keyPassword "password"
      v2SigningEnabled true // 开启 v2 签名
      v1SigningEnabled true // 开启 v1 签名
    }
  }
}
```

3. （可选）使用小程序视频播放功能。

   如果您接入了小程序并需要使用视频播放功能，同时您的应用需要支持 64位 CPU 架构，请修改主工程 AndroidManifest.xml，在 application 节点下添加以下属性：

```
android:allowNativeHeapPointerTagging="false"
```

4. 回归测试。

   全量回归测试的设备中必须包含 Android 11 或以上版本的设备。

   回归测试中您需要重点关注以下组件功能（如果使用）：

组件	验证项目
H5容器	离线包下载更新是否正常
移动分析	各类监控日志写入本地和上报是否正常
小程序	小程序包下载更新是否正常 图片-拍照 API 是否正常 视频播放、录制 API 是否正常 地图 API 是否正常
OCR	识别功能是否正常

定位	定位功能是否正常
分享	分享到各平台是否正常
设备标识	设备标识功能是否正常

6.5. mPaaS 适配 targetSdkVersion 29

mPaaS 旧版基线对 targetSdkVersion 最高仅支持到 26。从 10.1.68.21 开始，mPaaS 增加了对 targetSdkVersion 29 的支持。如果您的应用需要将 targetSdkVersion 升级到 29，请使用 mPaaS 插件 将 SDK 更新到 10.1.68.21 或以上版本，并按照下文进行适配和回归相关功能。

适配 targetSdkVersion 29

前置条件

请先参考 mPaaS 适配 targetSdkVersion 28 完成 targetSdkVersion 28 的相关适配。

修改 targetSdkVersion

AAR 接入方式

在工程主 module 下的 `build.gradle` 文件中修改属性 targetSdkVersion 29。

Portal&Bundle 接入方式

- 在 Portal 工程主 module 下的 `build.gradle` 文件中修改属性 targetSdkVersion 29。
- 在 Bundle 工程中的 targetSdkVersion 可不修改，但不得高于 Portal 工程。

通用配置

修改工程 `AndroidManifest.xml`，在 application 节点下添加以下属性：

```
<application
    android:requestLegacyExternalStorage="true"
    ... >
```

后台使用定位功能

如果您的应用需要在后台时使用定位功能，需添加、申请以下权限：

- 在 `AndroidManifest.xml` 中添加权限：

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

- 调用定位 API 前确保申请了该权限：

```
String[] permissions;
if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.Q) {
  permissions = new String[]{
      Manifest.permission.ACCESS_FINE_LOCATION,
      Manifest.permission.ACCESS_COARSE_LOCATION,
      Manifest.permission.ACCESS_BACKGROUND_LOCATION
      };
} else {
  permissions = new String[]{
      Manifest.permission.ACCESS_FINE_LOCATION,
      Manifest.permission.ACCESS_COARSE_LOCATION
      };
}
ActivityCompat.requestPermissions(this, permissions, 101);
```

使用小程序蓝牙功能

如果您的应用需要在小程序中使用蓝牙相关 API，需添加、申请以下权限。

- 在 `AndroidManifest.xml` 中添加权限：

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

- 调用蓝牙 API 前确保申请了该权限：

```
String[] permissions = new String[]{
      Manifest.permission.ACCESS_FINE_LOCATION,
      };
ActivityCompat.requestPermissions(this, permissions, 101);
```

回归测试

全量回归测试的设备中必须包含 Android 10.0 或以上版本的设备。

回归测试中您需要重点关注以下组件功能（如果使用）：

组件	验证项目
统一存储	数据库加密存储 是否正常。
热修复	热修复 是否能够生效。
移动分析	移动分析 卡顿监控是否正常。
小程序	• 小程序文件 API 是否正常。 • 小程序蓝牙 API 是否正常。 • 小程序地图组件是否正常。
定位	定位 是否正常。

6.6. mPaaS 适配 targetSdkVersion 28

mPaaS 旧版基线对 targetSdkVersion 最高仅支持到 26。从 10.1.68.21 开始，mPaaS 增加了对 targetSdkVersion 28 的支持。如果您的应用需要将 targetSdkVersion 升级到 28，请使用 mPaaS 插件 将 SDK 更新到 10.1.68.21 或以上版本，并按照下文进行适配和回归相关功能。

适配 targetSdkVersion 28

修改 targetSdkVersion

AAR 接入方式

在工程主 module 下的 `build.gradle` 文件中修改属性 targetSdkVersion 28。

Portal & Bundle 接入方式

- 在 Portal 工程主 module 下的 `build.gradle` 文件中修改属性 targetSdkVersion 28。
- 在 Bundle 工程中的 targetSdkVersion 可不修改，但不得高于 Portal 工程。

通用配置

AAR 接入方式

修改工程 `AndroidManifest.xml`，在 application 节点下添加如下代码：

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

Portal & Bundle 接入方式

修改 Portal 工程 `AndroidManifest.xml`：

- 在 application 节点下添加如下代码：

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

- 从 LauncherActivity 删除以下属性（SDK 已改为通过代码设置）：

  ```
  android:screenOrientation="portrait"
  ```

其他配置

允许 HTTP 请求

Android 9.0 的网络配置默认禁止了 HTTP 请求，只允许 HTTPS 请求，设置 targetSdkVersion 28 将在 9.0+ 设备上启用 9.0 的网络配置。如果您仍然需要发送 HTTP 请求（包括小程序中），可通过配置 `networkSecurityConfig` 开启。

- 在工程（Portal&Bundle 为 Portal 工程）的 `res/xml` 目录下创建 `network_security_config.xml` 文件，内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <base-config cleartextTrafficPermitted="true">
      <trust-anchors>
          <certificates src="system" />
      </trust-anchors>
  </base-config>
</network-security-config>
```

- 在工程（Portal & Bundle 为 Portal 工程）的 `AndroidManifest.xml` 中的 `application` 节点添加属性：

```
android:networkSecurityConfig="@xml/network_security_config"
```

更多配置可参考 谷歌官方文档 。

透明背景 Activity 设置屏幕方向 crash

该适配点为 Android 8.0 系统 bug。在 8.0 设备上，当应用 targetSdkVersion > 26 时，透明背景的 Activity 如果设置了屏幕方向，打开该 Activity 就会触发 crash。触发具体条件为：

- Activity 使用的 theme 中 `windowIsTranslucent` 或 `windowIsFloating` 属性为 `true` 。
- 在 `AndroidManifest.xml` 中设置了 `screenOrientation` 属性，或调用了 `setRequestedOrientation` 方法。

您需要检查所有 Activity 是否满足触发条件，同时除了您自定义的 style 外，请注意部分常用的系统 theme 也满足条件，例如：

```
@android:style/Theme.Translucent.NoTitleBar
@android:style/Theme.Dialog
```

推荐适配方式：

1. 对于 theme 满足条件的 Activity，删除 `AndroidManifest.xml` 中的 `screenOrientation` 属性，改为调用 `setRequestedOrientation` 方法。
2. 在对应 Activity 或父类中重写 `setRequestedOrientation` 方法，`try catch` `super.setRequestedOrientation()` 兜底：

```
 @Override
 public void setRequestedOrientation(int requestedOrientation) {
     try {
         super.setRequestedOrientation(requestedOrientation);
     } catch (Exception ignore) {

     }
 }
```

3. mPaaS 提供的 `BaseActivity` 、 `BaseFragmentActivity` 、 `BaseAppCompatActivtiy` 均已重写 `setRequestedOrientation` 方法兜底。
4. 完成上述适配后，虽可避免 crash，但仍可能出现在 Android 8.0 设备上锁定方向失效的情况，请确保您的 Activity 不会因旋转屏幕发生异常（例如重走生命周期导致某些成员变量为空）。

Android 8.0 系统相关源码：

```java
@MainThread
@CallSuper
protected void onCreate(@Nullable Bundle savedInstanceState) {
    if (DEBUG_LIFECYCLE) Slog.v(TAG, "onCreate " + this + ": " + savedInstanceState);

    if (getApplicationInfo().targetSdkVersion > 0 && mActivityInfo.isFixedOrientation()) {
        final TypedArray ta = obtainStyledAttributes(com.android.internal.R.styleable.Window);
        final boolean isTranslucentOrFloating = ActivityInfo.isTranslucentOrFloating(ta);
        ta.recycle();

        if (isTranslucentOrFloating) {
            throw new IllegalStateException(
                    "Only fullscreen opaque activities can request orientation");
        }
    }

    if (mLastNonConfigurationInstances != null) {
        mFragments.restoreLoaderNonConfig(mLastNonConfigurationInstances.loaders);
    }
    if (mActivityInfo.parentActivityName != null) {
        if (mActionBar == null) {
            mEnableDefaultActionBarUp = true;
        } else {
            mActionBar.setDefaultDisplayHomeAsUpEnabled(true);
        }
    }
```

回归测试

全量回归测试的设备中必须包含 Android 9.0 以上版本的设备，同时对于透明背景 Activity 设置屏幕方向 crash 问题，请在 Android 8.0 机型上专项测试。

回归测试中您需要重点关注以下组件功能（如果使用）：

组件	验证项目
移动网关	• 开启 签名校验 后，RPC 调用是否成功。 • 开启 数据加密 后，RPC调用是否成功。
热修复	热修复 是否能够生效。
扫一扫	• 标准 UI 扫码是否成功。 • 标准 UI 打开手机相册、拍照及预览是否正常。 • 自定义 UI 扫码是否成功，如自定义 UI，需要 适配部分新接口 。
统一存储	• 数据库加密存储 是否正常。 • 文件加密存储 是否正常。
分享	分享到新浪微博、QQ 是否成功。

7.参考

7.1. 组件化接入方式下的环境配置

本文介绍了在不同系统下开发客户端应用前所需要完成的环境配置。

在进行客户端开发之前，您首先需要配置开发环境：

- 配置 Windows 开发环境
- 配置 macOS 开发环境
- 配置 Linux 开发环境

配置 Windows 开发环境

参考以下说明配置 Windows 开发环境。

配置 Java 8 环境

mPaaS 框架只支持 **JDK 8 及以上版本** 。

1. 下载并安装 JDK 8。
2. 配置 `JAVA_HOME` 环境变量，并将 `JAVA_HOME` 下的 bin 路径添加到 `PATH` 环境变量中。
3. 正确配置后，在命令行执行 `java -version` 命令，您将看到 JDK 版本等信息：

配置 Gradle 4.4 环境

mPaaS 框架只支持 **Gradle 4.4**。

使用 Gradle Wrapper（推荐）

1. 如果您的项目原先已经使用 Gradle Wrapper 进行构建，则建议在项目目录 `/gradle/wrapper/gradle.properties` 下把版本号修改为 **4.4**。
2. 如果您的项目没有使用 Gradle Wrapper，建议先使用全局的 Gradle（4.4）版本，然后调用 `gradle wrapper --gradle-version=4.4` 安装一个 gradle wrapper。完成上述步骤后，您只需要使用 `./gradlew` 的形式构建，这样的方式能最小的影响您的开发环境。

使用独立 gradle

1. 下载 。
2. 解压 `.zip` 包，然后将解压路径配置为 `GRADLE_HOME` 环境变量，并将 `GRADLE_HOME` 下的 bin 路径添加到 `PATH` 环境变量中。
3. 正确配置后，在命令行执行 `gradle -v` 命令，您将看到 Gradle 版本等信息：

安装并配置 Android Studio

安装 Android Studio

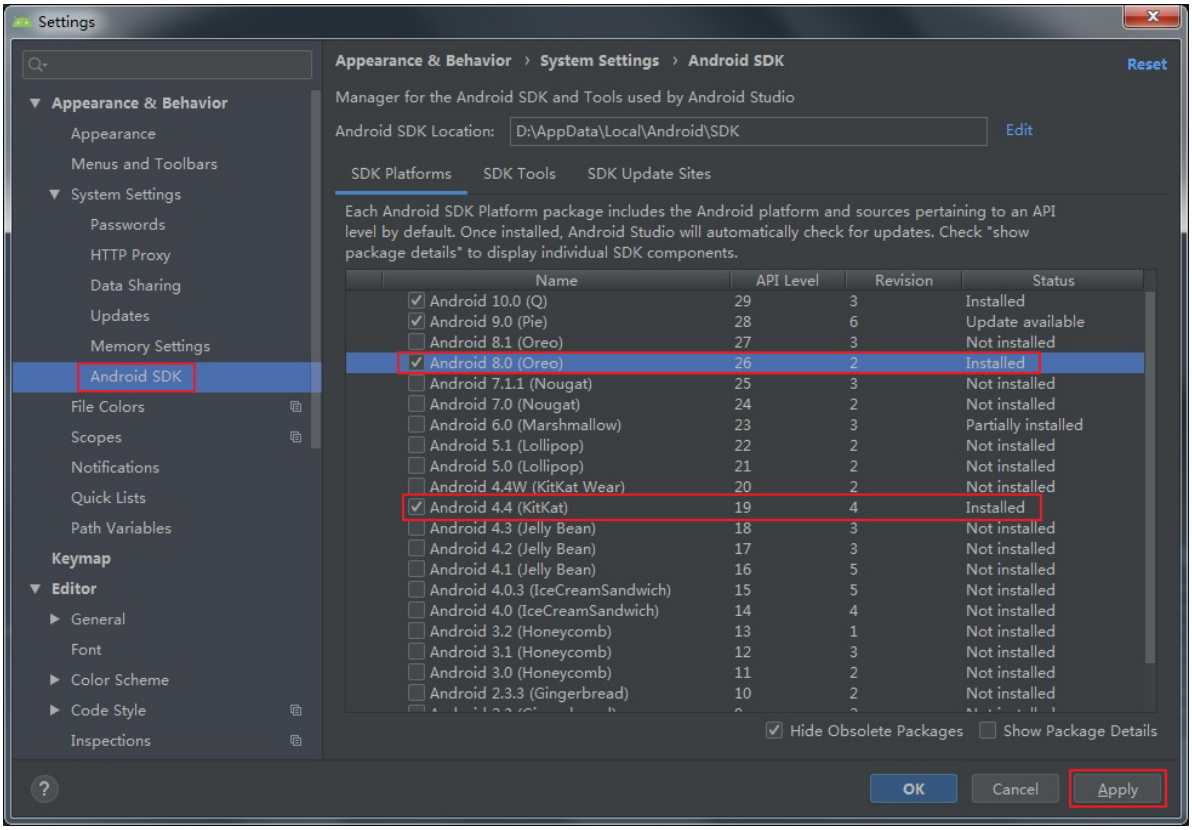
最新版 mPaaS 插件仅支持 **4.0** 及以上版本的 Android Studio。

- 关于 Android Studio 下载，请参见 Android Developers。
- 安装指南。
- 如果您之前使用的是低版本 Android Studio 并已经安装了 mPaaS 插件，那么在您从低版本 Android Studio 升级至 4.0 或更新版本的 Android Studio 之后，您只需要升级 mPaaS 插件至最新版本即可。详情请参见 更新 mPaaS 插件。
- 如果您需要支持 4.0 之前版本 Android Studio 的 mPaaS 插件，请在下载离线安装包后，采用离线安装形式安装。更多关于离线安装的指导说明，请参考 离线安装 mPaaS 插件。

安装 Android SDK

您需要安装 API Level 为 **19** 和 **26** 的 Android SDK。

1. 在 Android Studio 中，通过 **File** > **Settings** 打开设置对话框。
2. 如下图所示，在 **Android SDK** 对话框中，勾选 API Level 为 **19** 和 **26** 的 SDK，然后点击 **Apply** 按钮进行安装：



安装 mPaaS 插件

关于更多安装 mPaaS 插件的信息，请参见 安装 mPaaS 插件。

配置 Gradle 构建工具

您需要确保工程构建时使用 **Gradle Wrapper**。

1. 在 Android Studio 中，通过 **File** > **Settings** 打开设置对话框。
2. 在 **Gradle** 对话框中，勾选 **Use default gradle wrapper**，然后点击 **Apply** 按钮。

配置 macOS 开发环境

按照以下描述配置 macOS 开发环境。

配置 Java 8 环境

mPaaS 框架只支持 **JDK 8 及以上版本** 。

1. 下载并安装 JDK 8。
2. 配置 `JAVA_HOME` 环境变量，并将 `JAVA_HOME` 下的 bin 路径添加到 `PATH` 环境变量中。
3. 正确配置后，在命令行执行 `java -version` 命令，您将看到 JDK 版本等信息。

配置 Gradle 4.4 环境

mPaaS 框架只支持 **Gradle 4.4**。

使用 Gradle Wrapper（推荐）

1. 如果您的项目原先已经使用 Gradle Wrapper 进行构建，则建议在项目目录 `/gradle/wrapper/gradle.properties` 下把版本号修改为 **4.4**。
2. 如果您的项目没有使用 Gradle Wrapper，建议先使用全局的 Gradle（4.4）版本，然后调用 `gradle wrapper --gradle-version=4.4` 安装一个 gradle wrapper。完成上述步骤后，您只需要使用 `./gradlew` 的形式构建，这样的方式能最小的影响您的开发环境。

使用独立 Gradle

1. 下载 。
2. 解压 `.zip` 包，然后将解压路径配置为 `GRADLE_HOME` 环境变量，并将 `GRADLE_HOME` 下的 bin 路径添加到 `PATH` 环境变量中。
3. 正确配置后，在命令行执行 `gradle -v` 命令，您将看到 Gradle 版本等信息。

安装并配置 Android Studio

安装 Android Studio

最新版 mPaaS 插件仅支持 **4.0** 及以上版本的 Android Studio。

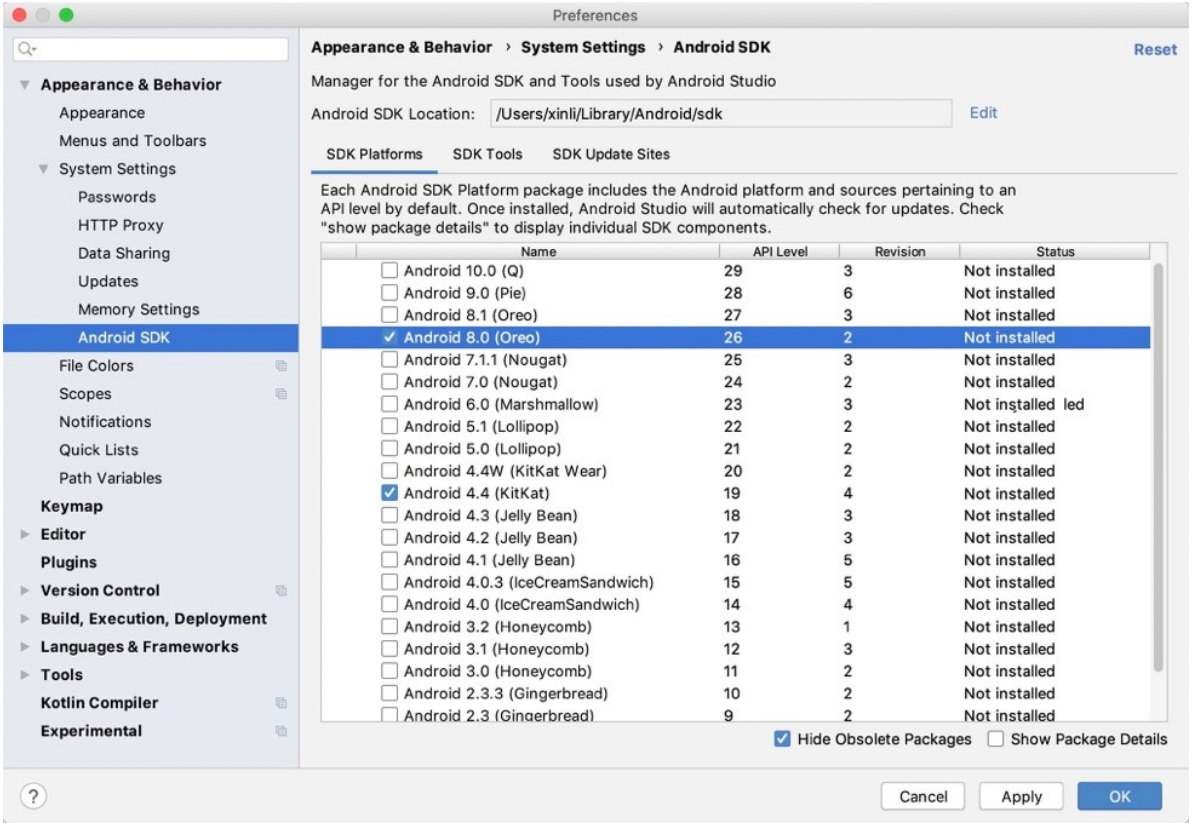
- 关于 Android Studio 下载，请参见 Android Developers。
- 安装指南。
- 如果您之前使用的是低版本 Android Studio 并已经安装了 mPaaS 插件，那么在您从低版本 Android Studio 升级至 4.0 或更新版本的 Android Studio 之后，您只需要升级 mPaaS 插件至最新版本即可。详情请参见 更新 mPaaS 插件。
- 如果您需要支持 4.0 之前版本 Android Studio 的 mPaaS 插件，请在下载离线安装包后，采用离线安装形式安装。更多关于离线安装的指导说明，请参考 离线安装 mPaaS 插件。

安装 Android SDK

您需要安装 API Level 为 **19** 和 **26** 的 Android SDK。

1. 在 Android Studio 中，通过 **Android Studio** > **Preferences** 打开设置对话框。

2. 在 **Android SDK** 对话框中，勾选 API Level 为 **19** 和 **26** 的 SDK，然后点击 **Apply** 按钮进行安装。



安装 mPaaS 插件

关于更多安装 mPaaS 插件的信息，请参见 安装 mPaaS 插件。

配置 Gradle 构建工具

您需要确保工程构建时使用 **Gradle Wrapper**。

1. 在 Android Studio 中打开任意一个 Android 工程。
2. 打开设置对话框。
3. 在 **Gradle** 对话框中，勾选 **Use default gradle wrapper**，然后点击 **Apply**。

配置 Linux 开发环境

按照以下说明配置 Linux 开发环境。说明：Linux 操作系统版本较多，本文仅适用于 CentOS 和 Ubuntu 版本。

配置 Java 8 环境

mPaaS 框架只支持 **JDK 8 及以上版本** 。

1. 下载并安装 JDK 8。
2. 配置 `JAVA_HOME` 环境变量，并将 `JAVA_HOME` 下的 bin 路径添加到 `PATH` 环境变量中。
3. 正确配置后，在命令行执行 `java -version` 命令，您将看到 JDK 版本等信息。

配置 Gradle 4.4 环境

使用 Gradle Wrapper（推荐）

1. 如果您的项目原先已经使用 Gradle Wrapper 进行构建，则建议在项目目录 `/gradle/wrapper/gradle.properties` 下把版本号修改为 **4.4**。

2. 如果您的项目没有使用 Gradle Wrapper，建议先使用全局的 Gradle（4.4）版本，然后调用 `gradle wrapper --gradle-version=4.4` 安装一个 gradle wrapper。完成上述步骤后，您只需要使用 `./gradlew` 的形式构建，这样的方式能最小的影响您的开发环境。

使用独立 gradle

1. 下载 。
2. 解压 `.zip` 包，然后将解压路径配置为 `GRADLE_HOME` 环境变量，并将 `GRADLE_HOME` 下的 bin 路径添加到 `PATH` 环境变量中。
3. 正确配置后，在命令行执行 `gradle -v` 命令，您将看到 Gradle 版本等信息。

安装 32 位兼容库

CentOS 6、CentOS 7、Ubuntu 等 Linux 发行版默认去除 `ia32-lib` 。所有 64 位 Linux 系统都需安装 32 位兼容库。参照 android-sdk 中的安装方法。

```
Ubuntu:
sudo apt-get install zlib1g:i386

CentOS:
yum install libstdc++.i686
```

安装并配置 Android Studio

安装 Android Studio

最新版 mPaaS 插件仅支持 **4.0** 及以上版本的 Android Studio。

- 关于 Android Studio 下载，请参见 Android Developers。
- 安装指南。
- 如果您之前使用的是低版本 Android Studio 并已经安装了 mPaaS 插件，那么在您从低版本 Android Studio 升级至 4.0 或更新版本的 Android Studio 之后，您只需要升级 mPaaS 插件至最新版本即可。详情请参见 更新 mPaaS 插件。
- 如果您需要支持 4.0 之前版本 Android Studio 的 mPaaS 插件，请在下载离线安装包后，采用离线安装形式安装。更多关于离线安装的指导说明，请参考 离线安装 mPaaS 插件。

安装 Android SDK

您需要安装 API Level 为 **19** 和 **26** 的 Android SDK。

1. 在 Android Studio 中，通过 **File** > **Settings** 打开设置对话框。
2. 在 **Android SDK** 对话框中，勾选 API Level 为 **19** 和 **26** 的 SDK，然后点击 **Apply** 按钮进行安装。

安装 mPaaS 插件

关于更多安装 mPaaS 插件的信息，请参见 安装 mPaaS 插件。

配置 Gradle 构建工具

您需要确保工程构建时使用 **Gradle Wrapper**。

1. 在 Android Studio 中打开任意一个 Android 工程。
2. 打开设置对话框。
3. 在 **Gradle** 对话框中，勾选 **Use default gradle wrapper**，然后点击 **Apply** 按钮。

7.2. 切换工作空间（Workspace）

应用开发过程中，常会有更换应用环境信息或多套环境（即工作空间，Workspace）并行研发的需求。

mPaaS 提供的工具可帮助您在开发过程中方便地进行环境切换。根据切换环境的需求不同，分为以下两种方式：

- 静态环境切换
- 动态环境切换

静态环境切换

前置条件

您已有基于 mPaaS 框架开发的 App。更多信息参见 基于 mPaaS 框架 > 快速开始 。

在进行静态环境切换时，需要使用到 `easyconfig` ， `easyconfig` 的工作原理如下：

- 能够修改 `AndroidManifest workspace` 相关的 `meta` 属性。
- 修改 `assets` 下的 `mpaas.properties` 文件。
- 如果 `mPaaS` 工程配置文件中包涵 `base64` 属性且属性不为空，会生成无线保镖加密图片 `yw_1222.jpg` 。

公有云

在公有云环境中，切换工作空间的步骤如下：

1. 确保工程根目录 `build.gradle` 文件中，有如下依赖：

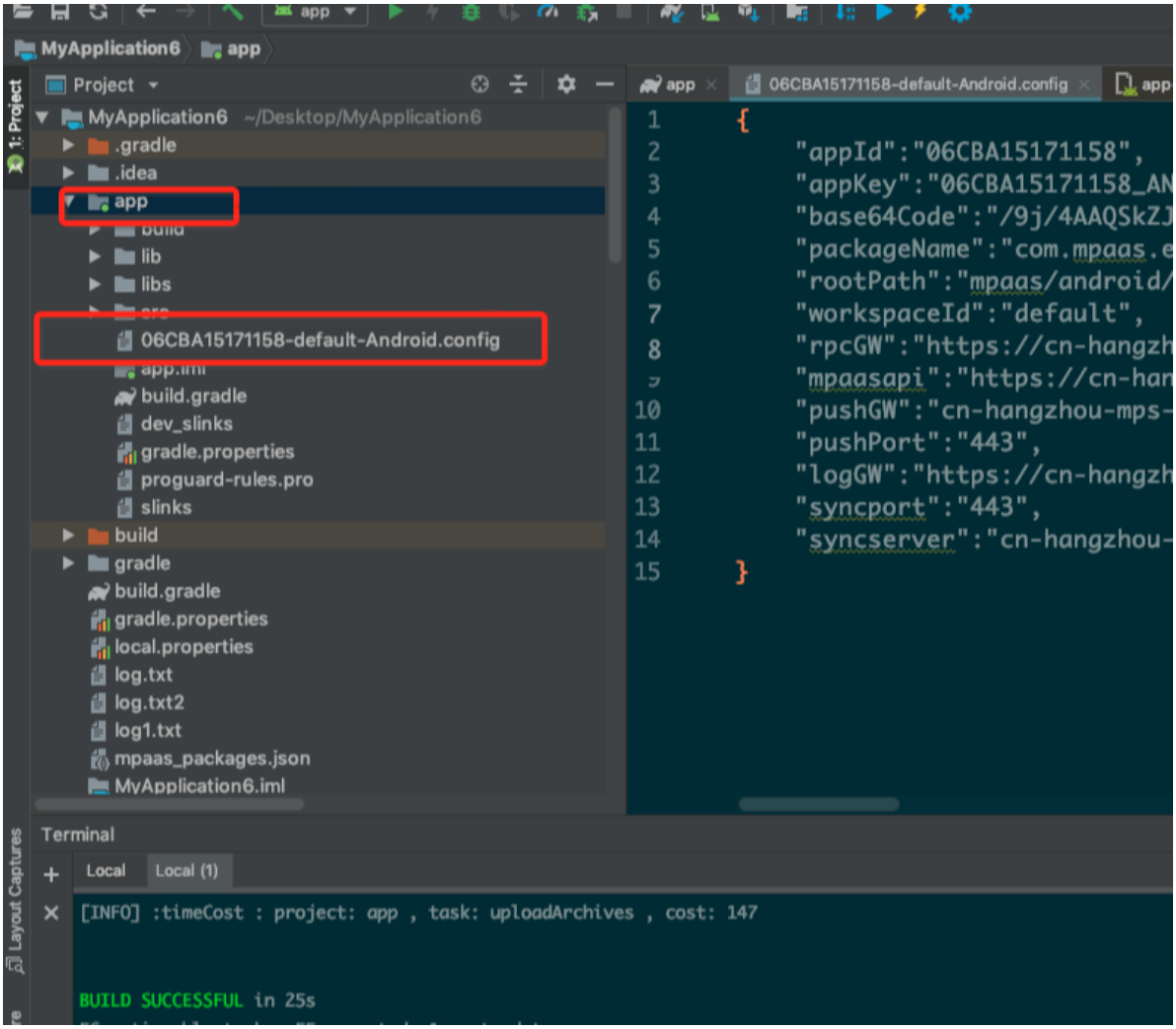
   > **说明**
   >
   > 因功能迭代，以下依赖的版本可能会不断增大。

   ```
   classpath 'com.alipay.android:android-gradle-plugin:3.0.0.9.13'
   // 版本号必须大于 2.8.0
   classpath 'com.android.boost.easyconfig:easyconfig:2.8.0'
   ```

2. 确保主工程（ `android main module` ）的 `build.gradle` 中有如下配置（请注意顺序）：

   ```
   apply plugin: 'com.alipay.portal'
   //位于 com.alipay.portal 之后即可
   apply plugin: 'com.alipay.apollo.baseline.update'
   ```

3. 从控制台下载对应工作空间（Workspace）的 `.config` 配置文件。更多信息，请参考 在控制台创建应用 > 下载配置文件。
4. 将下载的 `.config` 配置文件添加到主工程（ `android main module` ）路径下。如下图所示：



> **重要**
>
> 仅保留对应工作空间的配置文件即可。

专有云

在专有环境中，切换工作空间的步骤如下：

1. **确保工程根目录** `build.gradle` **文件中，有如下依赖：**

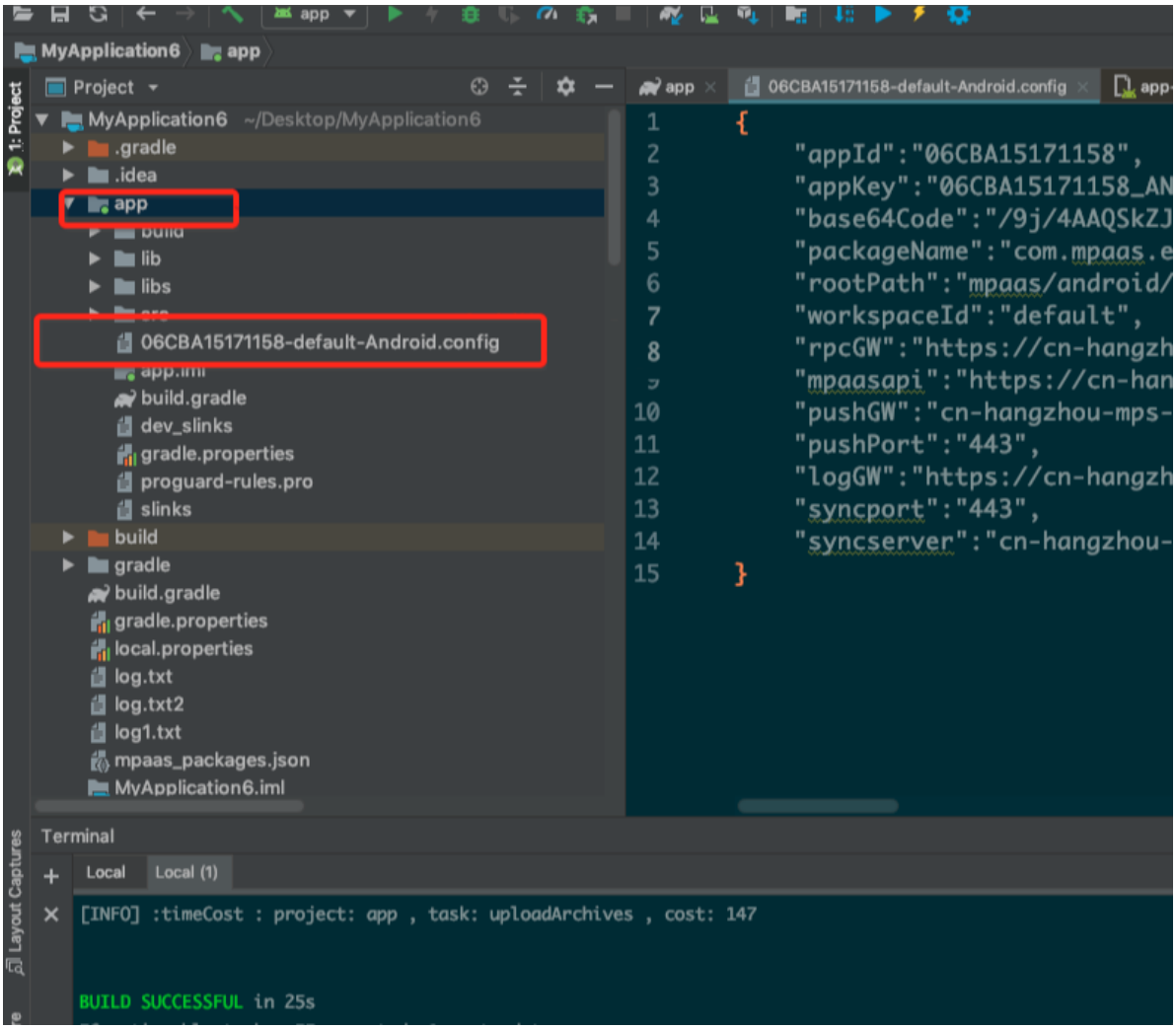
> **说明**
>
> 因功能迭代，以下依赖的版本可能会不断增大。

```
classpath 'com.alipay.android:android-gradle-plugin:3.0.0.9.13'
// 版本号必须大于 2.8.0
classpath 'com.android.boost.easyconfig:easyconfig:2.8.0'
```

2. **确保主工程（** `android main module` **）的** `build.gradle` **中有如下配置（需注意顺序）：**

```
apply plugin: 'com.alipay.portal'
//位于 com.alipay.portal 之后即可
apply plugin: 'com.alipay.apollo.baseline.update'
```

3. 从控制台下载对应工作空间的 `.config` 配置文件。更多信息参考 在控制台创建应用 > 下载配置文件。
4. 将下载的 `.config` 配置文件添加到主工程（`android main module`）路径下。如下图所示：



> **重要**
>
> 仅保留对应工作空间的配置文件即可。

5. 使用 mPaaS 插件生成 `yw_1222.jpg` 加密图片。更多信息参见 加密图片（专有云）。

动态环境切换

动态环境切换指客户端不重新打包的情况下，通过修改手机设置中环境选项，动态修改应用的环境信息。

> **说明**
>
> - 动态环境切换功能仅支持在专有云环境下使用。
> - 动态环境切换适用于开发阶段多套环境并存且频繁切换的场景。
> - 采用动态环境切换时需要向应用写入新环境的环境配置文件。因此在采用该方式时，您需要为应用申请文件存储权限。

由于 mPaaS 安全验签机制的限制，更新环境配置信息会修改无线保镖验签 `yw_1222.jpg` 图片，因此动态切换环境有两个限制：

- 仅适用于开发阶段，上线前请注意删除对应的配置，否则 Release 包会报 `RuntimeException` 异常。
- mPaaS 控制台需关闭网络请求验签开关，否则会因验签图片信息不对导致请求失败。

添加动态环境切换 SDK

1. 添加依赖。
   - 原生 AAR 接入方式下，在工程主 module 下面的 `build.gradle` 配置文件中的 `dependencies` 中添加以下依赖：

     ```
     dependencies {
     ...
     implementation 'com.mpaas.mocksettings:mocksettings-build:10.1.60a.1575@aar'
     ...
     }
     ```

   - Portal&Bundle 接入方式下，在 Portal 工程主 module 下面的 `build.gradle` 配置文件中的 `dependencies` 中添加以下依赖：

     ```
     dependencies {
     ...
     bundle 'com.mpaas.mocksettings:mocksettings-build:1.0.0.200421111458@jar'
     manifest 'com.mpaas.mocksettings:mocksettings-
     build:1.0.0.200421111458:AndroidManifest@xml'
     ...
     }
     ```

2. 使用 SDK。
   - 原生 AAR 接入方式下，重写 Application 的 `getPackageManager` 方法，替换 `PackageManager` 为 `MockSettingsPackageManager` 。

```
private MockSettingsPackageManager mockSettingsPackageManager;

@Override
public PackageManager getPackageManager() {
    if (mockSettingsPackageManager == null) {
        mockSettingsPackageManager = new MockSettingsPackageManager(this,
super.getPackageManager());
    }
    return mockSettingsPackageManager;
}
```

- Portal&Bundle 接入方式下，将 Portal 工程主 module 下的 `AndroidManifest.xml` 中的 `application` 修改为：

```
<application
     android:name="com.alipay.mobile.quinox.MockSettingsLauncherApplication"
     ...
     >
     ...
</application>
```

3. 添加以下权限并确保运行时已动态申请：

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

4. 编译 debug 包或在 `AndroidManifest.xml` 中打开 debug 设置：

```
<application
     android:debuggable="true"
     ...
     >
     ...
</application>
```

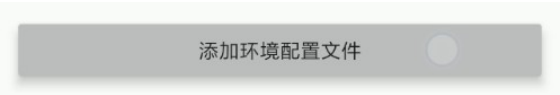
动态切换

1. 扫描二维码下载 **mPaaS 设置** App。

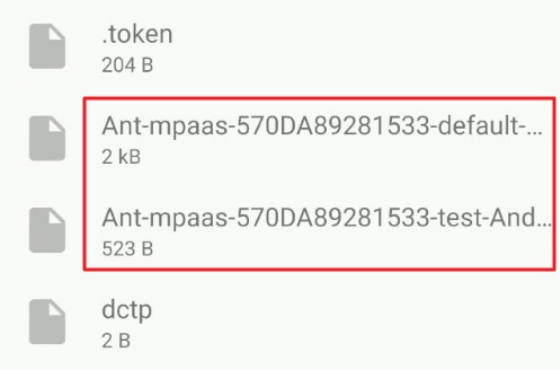
   安装完成后，显示 **mPaaS 设置** App 的图标如下：

2. 将 mPaaS 控制台下载的 `config` 文件放入手机 SD 卡或内部存储中。

3. 添加环境。通过 **mPaaS 设置** App 将 `config` 文件添加到列表中。
   i. 打开 **mPaaS 设置** App。
   ii. 点击 **环境列表** 页面底部的 **添加环境配置文件**。



   iii. 找到要添加的环境配置文件。



   iv. 依次将两个文件（正式环境和测试环境）添加至环境列表。

4. 切换环境。
   i. 选择上图中的一个环境，点击 **切换**，将其选中为当前环境。

   ii. 然后启动该环境所对应的 App，测试请求可正常发送，说明环境切换成功。

此时若切换到另一个环境，再重启前一个环境所对应的 App，由于新的环境内没有对应的 operationType，所以系统会报 3000 异常，这是已成功切换到另一个环境后的正常结果。

7.3. DSA 证书加密工具说明

由于 Android App 通常以 RSA 方式加密，mPaaS 控制台目前暂时仅支持为以 RSA 方式加密的 App 获取签名。如果您需要使用 DSA 方式对 App 进行签名，则需要按照以下步骤进行加签。

具体的加签步骤如下：

1. 进入 mPaaS 控制台 > **代码管理** > **代码配置** > **Android** 标签页下载配置文件。
   - 下载配置文件前，不要上传签名后的 APK。

- 如果存在 base64Code 的值，需清空，如下图所示。


```
{
    "appId":"1A8947[redacted]"
    "appKey":"1A894[redacted]DROID",
    "base64Code":"",
    "packageName":"com.example.nativeapplication",
    "rootPath":"mpaas/android/1A89474101135-default",
    "workspaceId":"default",
    "rpcGW":"https://cn-hangzhou-mgs-gw.cloud.alipay.com/mgw.htm",
    "mpaasapi":"https://cn-hangzhou-component-gw.cloud.alipay.com/mgw.htm",
    "pushPort":"443",
    "pushGW":"cn-hangzhou-mps-link.cloud.alipay.com",
    "logGW":"https://cn-hangzhou-mas-log.cloud.alipay.com",
    "syncport":"443",
    "syncserver":"cn-hangzhou-mss-link.cloud.alipay.com"
}
```


2. 使用 mPaaS 插件生成加密图片。从 Android Studio 顶部导航栏进入 **mPaaS** > **基础工具** > **生成加密图片（专有云配置文件）** 页面，填写相关配置信息后，点击 **OK** 生成加密图片。

appSecret 可从 mPaaS 控制台的 **代码管理** > **代码配置** > **Android** 标签页获取，如下图所示。

3. 进行正常的 RPC 调用，查看是否调用成功。具体参考 调用 RPC 操作说明。

8.常见问题

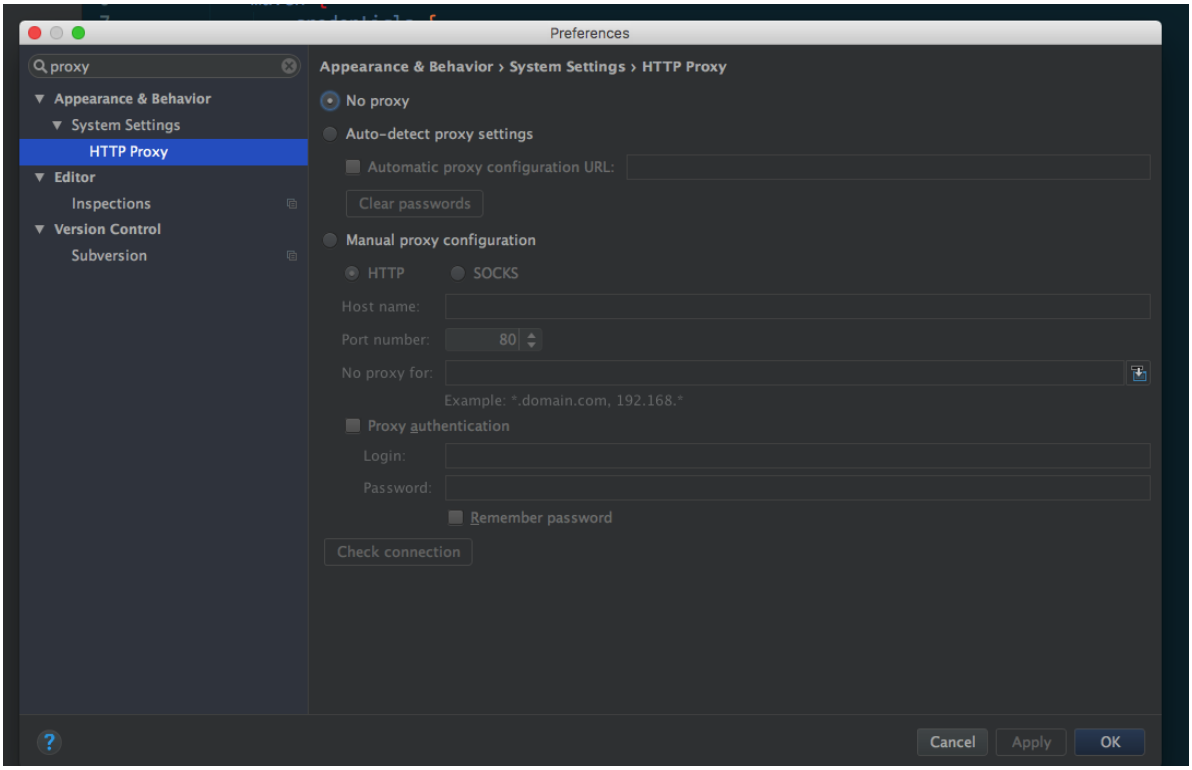
查看以下常见问题列表，单击具体的问题即可查看相应解答。

- 编译时无网络连接
- 程序编译失败
- 编译过程中出现卡顿
- 编译不通过且出现 NullPointerException
- 如何调试应用
- mPaaS Portal、Bundle 工程使用 MultiDex 的注意事项
- 如何清除 Gradle 缓存
- 如何升级到最新的 Gradle 插件
- 在华为 EMUI 10 系统中 input file 标签无法打开相机
- 如何在 Library 中使用/依赖 mPaaS
- 如何解决运行时出现的 608 错或 libsgmain 的 native 错误

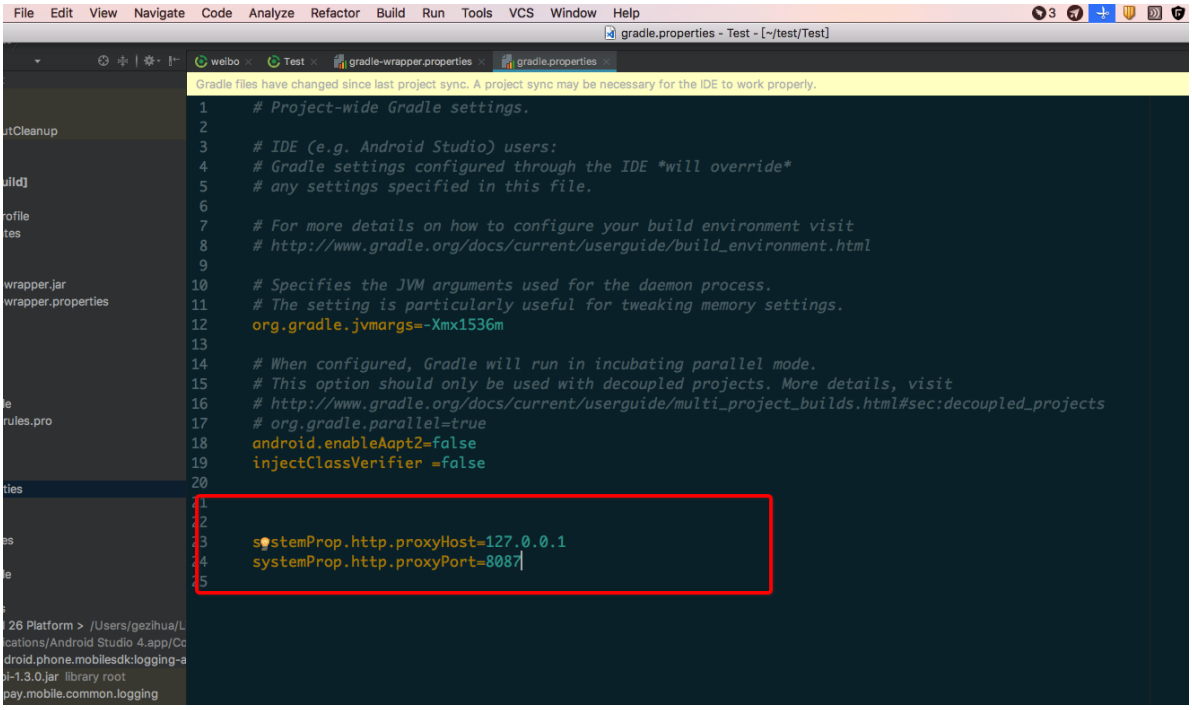
编译时无网络连接

在编译文件时，如果没有网络，很有可能造成编译失败。通过以下步骤，确认编译环境的网络已连接。

1. 确认已连接到互联网。
2. 确认未连接网络代理，包括浏览器代理设置、第三方网络代理软件等。
3. 确认未设置 IDE 代理。



4. 在 `gradle.properties` 文件中，确认未设置 Gradle 代理，即未设置 `systemProp.http.proxyHost` 和 `systemProp.http.proxyPort` 属性。如果有设置，删除相关属性即可。



程序编译失败

如果程序编译失败，可通过以下步骤进行排错与解决。

1. 根据 编译时无网络连接，确认编译环境网络已正常连接。
2. 检查 Gradle 执行记录，确认新增的依赖有效。
3. 检查依赖的 GAV（`group`、`artifact`、`version`）参数设置正确。

```
//引用 debug 包group:artifact:version:raw@jar
bundle "com.app.xxxxx.xxxx:test-build:1.0-SNAPSHOT:raw@jar"
//引用 release 包group:artifact:version@jar
bundle "com.app.xxxxx.xxxx:test-build:1.0-SNAPSHOT@jar"
manifest "com.app.xxxxx.xxxx:test-build:1.0-SNAPSHOT:AndroidManifest@xml"
```

4. 在系统自带的命令行工具中，执行以下命令，导出 Gradle 执行记录：

```
// 执行命令前，确认未定义 productflavor 属性。否则，命令会运行失败。
// 以下命令将执行记录导出至 log.txt 文件中。
gradle buildDebug --info --debug -Plog=true > log.txt
```

5. 查看步骤 4 中导出的记录文件，在最新生成的记录中，会看到类似如下记录，表示新增的依赖不存在。

```
Caused by: org.gradle.internal.resolve.ArtifactNotFoundException: Could not find nebu
lacore-build-AndroidManifest.xml (com.alipay.android.phone.wallet:nebulacore-build:1.
6.0.171211174825).
Searched in the following locations:
http://mvn.cloud.alipay.com/nexus/content/repositories/releases/com/alipay/android/phon
wallet/nebulacore-build/1.6.0.171211174825/nebulacore-build-1.6.0.171211174825-Androi
dManifest.xml
     at
org.gradle.internal.resolve.result.DefaultBuildableArtifactResolveResult.notFound(Defau
BuildableArtifactResolveResult.java:38)
     at
org.gradle.api.internal.artifacts.ivyservice.ivyresolve.CachingModuleComponentRepositor
LocateInCacheRepositoryAccess.resolveArtifactFromCache(CachingModuleComponentRepository
ava:260)
```

6. 访问该记录中的 HTTP 链接（如上一步所列记录中的第 3 行）并登录，查看 Maven 仓库。

> **说明**
> 您可以在 `build.gradle` 文件中查看登录时需要提供的账户名和密码。

7. 执行以下命令刷新 `gradle` 缓存。

```
gradle clean --refresh-dependencies
```

8. 如果 Maven 仓库有对应依赖，删除个人目录下 Gradle 缓存，然后重新编译。删除 Gradle 缓存的方法如下：
   - 在 macOS、Linux、Unix 等系统中运行以下命令。

     ```
     cd ~
     cd .gradle
     cd caches
     rm -rf modules-2
     ```

   - 在 Windows 系统中，默认情况下，路径定位到 `C:\Users\\{用户名}\\.gradle\caches`，删除 `modules-2` 文件夹。

编译过程中出现卡顿

如果编译过程卡顿（等待超过 20 分钟），您可以通过以下步骤提高编译效率。

1. 根据 上文步骤，确认编译环境网络已正常连接。
2. 确认防火墙已关闭。
3. 确认未开启 IntelliJ IDEA 编译器的网络配置。
4. 在代码中，提前加载 Maven 镜像。例如，以下是阿里云加载 Maven 镜像的代码。

```
apply plugin: 'maven'
buildscript {
    repositories {
        mavenLocal()

// 开始先加载 Maven 镜像
        maven{ url 'http://maven.aliyun.com/nexus/content/groups/public/'}

        maven {
            credentials {
                username "请使用已知用户"
                password "请使用已知密码"
            }
            url "http://mvn.cloud.alipay.com/nexus/content/repositories/releases/"
        }
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.1.3'
        classpath 'com.alipay.android:android-gradle-plugin:2.1.3.3.3'
        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
    }
}
allprojects {
    repositories {
        flatDir {
            dirs 'libs'
        }
        mavenLocal()
        maven {
            credentials {
                username "xxxxxxxxx"
                password "xxxxxxx"
            }
            url "http://mvn.cloud.alipay.com/nexus/content/repositories/releases/"
        }
        maven{ url 'http://maven.aliyun.com/nexus/content/groups/public/'}
    }
}
```

编译不通过且出现 NullPointerException

```
Caused by: java.lang.NullPointerException
        at com.alipay.mpaas.codegen.modify.bean.QNameBean.containsAttributeValue(QNameBean.java:38)
        at com.alipay.mpaas.codegen.modify.AddedInterceptor.findElementEquals(AddedInterceptor.java:95)
        at com.alipay.mpaas.codegen.modify.ModifyInterceptor.canHandle(ModifyInterceptor.java:33)
        at com.alipay.mpaas.codegen.modify.ModfiedOrAddedInterceptor.canHandle(ModfiedOrAddedInterceptor.java:26)
        at com.alipay.mpaas.codegen.request.AbsInterceptor.handle(AbsInterceptor.java:26)
        at com.alipay.mpaas.codegen.fastconvert.ModifyAndroidManifestConverter.handleModifyElement(ModifyAndroidManifestConverter.java:134)
        at com.alipay.mpaas.easy.config.GradleModifyAndroidManifestConverter.initManifestObjects(GradleModifyAndroidManifestConverter.java:51)
        at com.alipay.mpaas.easy.config.GradleModifyAndroidManifestConverter.convert(GradleModifyAndroidManifestConverter.java:67)
        at com.alipay.mpaas.easy.config.MPaaSManifestTask.convert(MPaaSManifestTask.java:39)
        at com.alipay.mpaas.easy.config.MPaaSManifestTask.executeTask(MPaaSManifestTask.java:24)
        at org.gradle.internal.reflect.JavaMethod.invoke(JavaMethod.java:73)
        at org.gradle.api.internal.project.taskfactory.StandardTaskAction.doExecute(StandardTaskAction.java:46)
        at org.gradle.api.internal.project.taskfactory.StandardTaskAction.execute(StandardTaskAction.java:39)
        at org.gradle.api.internal.project.taskfactory.StandardTaskAction.execute(StandardTaskAction.java:26)
        at org.gradle.api.internal.AbstractTask$TaskActionWrapper.execute(AbstractTask.java:780)
        at org.gradle.api.internal.AbstractTask$TaskActionWrapper.execute(AbstractTask.java:747)
        at org.gradle.api.internal.tasks.execution.ExecuteActionsTaskExecuter$1.run(ExecuteActionsTaskExecuter.java:121)
        at org.gradle.internal.progress.DefaultBuildOperationExecutor$RunnableBuildOperationWorker.execute(DefaultBuildOperationExecutor.java:336)
        at org.gradle.internal.progress.DefaultBuildOperationExecutor$RunnableBuildOperationWorker.execute(DefaultBuildOperationExecutor.java:328)
        at org.gradle.internal.progress.DefaultBuildOperationExecutor.execute(DefaultBuildOperationExecutor.java:199)
        at org.gradle.internal.progress.DefaultBuildOperationExecutor.run(DefaultBuildOperationExecutor.java:110)
        at org.gradle.api.internal.tasks.execution.ExecuteActionsTaskExecuter.executeAction(ExecuteActionsTaskExecuter.java:110)
        at org.gradle.api.internal.tasks.execution.ExecuteActionsTaskExecuter.executeActions(ExecuteActionsTaskExecuter.java:92)
        ... 103 more
```

在接入专有云时，下载配置文件并接入 mPaaS 后，编译不通过并且出现 `NullPointerException` 。如果遇到此类问题，一般需要对 config 配置文件中的字段进行检查。检查 13 个字段是否有缺少，和公有云下载过来的文件进行对比，确认字段名是否正确。

如何调试应用

开发过程中需要调试代码，本文介绍两种调试方式。

- 以调试模式启动应用
- 应用运行后调试

以调试模式启动应用

- **使用场景**

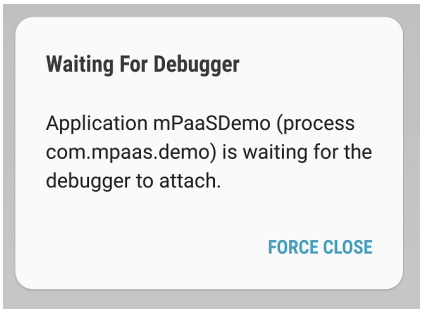
  希望调试应用启动时的最初代码，比如在 `application init` 时初始化代码。

- **操作步骤**

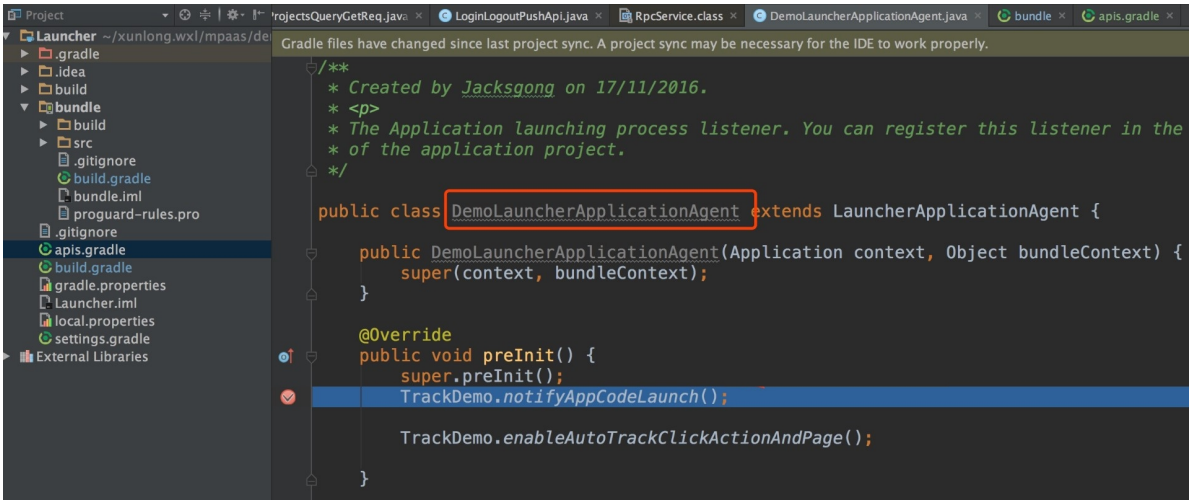
1. 执行命令 `adb shell am start -W -S -D 应用包名/应用第一个启动的页面类名` 。例如，mPaaS Demo 的包名是 `com.mpaas.demo` ，应用第一个启动的页面类名是 `com.alipay.mobile.quinox.LauncherActivity` ，那么可以使用命令行 `adb shell am start -W -S -D com.mpaas.demo/com.alipay.mobile.quinox.LauncherActivity` 以调试模式启动应用。第一个启动的类名如下图所示。

```
<application
    android:name="com.alipay.mobile.quinox.LauncherApplication"
    android:allowBackup="false"
    android:icon="@mipmap/ic_launcher"
    android:label="mPaaSDemo"
    android:theme="@style/AppTheme"
    tools:replace="android:label">
    <activity
        android:name="com.alipay.mobile.quinox.LauncherActivity"
        android:windowSoftInputMode="stateAlwaysHidden">
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
            <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
    </activity>
```

2. 执行命令之后，手机会弹出如下对话框。



3. 对希望调试的代码行设置断点，然后附着到应用所在进程即可，如图。



应用运行后调试

- **使用场景**

  在触发某个事件之后进行调试，比如单击某个按钮或者跳转某个页面才需要调试。

- **操作步骤**

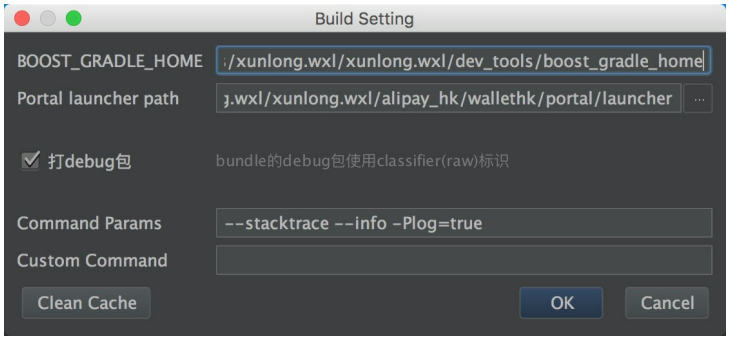
  在应用运行后，单击附着进程（  ）按钮，或者在执行上述命令后，再单击附着按钮开始调试。

在 mPaaS Portal/Bundle 工程中使用 MultiDex 的注意事项

Portal 和 Bundle 不建议介入 MultiDex，除非您是单 portal 工程，需要使用 `multiDexEnabled true` 。如果您的 Bundle 过大，目前只能使用拆分 bundle 的方式进行，不要在 bundle 中开启 multidex 支持。

如何清除 Gradle 缓存

打开 Gradle 插件的设置界面，单击 **Clean Cache** 按钮，即可删除 Gradle 插件的所有缓存数据。



如何升级到最新的 Gradle 插件

> **说明**
>
> 本节内容只适用于 10.1.68 系列基线。更多关于该版本基线的信息，请参见 基线简介 和 10.1.68 系列基线发布说明 。

目前 Google 官方提供的 Android Gradle Plugin 是 3.5.x 版本。

mPaaS 也提供了 3.5.x 版本的插件作为适配，可支持 Google Android Gradle Plugin 3.5.3 和 Gradle 6.2 的 API。

引入方式的变化

1. 您只需要通过添加以下依赖来引入我们的插件，不需要引入 Android Gradle Plugin 官方插件，因为依赖传递的关系，会自动引入。

```
dependencies {
classpath 'com.alipay.android:android-gradle-plugin:3.5.18'
}
```

2. Gradle Wrapper 的版本需要升级到 5.6 以上，推荐使用 6.2。

使用方式的变化

- 不再需要使用 `apply plugin:'com.android.application'`
  - 如果您是 portal 工程，仅需要使用 `apply plugin:'com.alipay.portal'` 。
  - 如果您是 bundle 工程，需要删除 `apply plugin:'com.android.application'` ，仅需要使用 `apply plugin:'com.alipay.bundle'` 。
  - 如果您是 library 工程，需要删除 `apply plugin:'com.alipay.library'` ， 仅需要使用 `apply plugin:'com.android.library` 。
- 如果使用最新稳定版本 Android Studio 3.5 或以上，那么您需要在 `gradle.properties` 里面新增 `android.buildOnlyTargetAbi=false` 。
- 由于我们的无线保镖组件暂不支持 V2 签名，如果您需要使用 Android Studio 调试并安装您的 APK，那么您需要禁用 V2 签名；如果您使用命令行进行构建，且您的 minSdkVersion 大于等于 24，则您也需要禁用 V2 签名。禁用 V2 签名的方式如下：

```
v2SigningEnabled false
```

> **重要**
>
> 清除缓存后需观察确认小程序和 H5 能否正常工作。

在华为 EMUI 10 系统中 input file 标签无法打开相机

由于华为10 系统 URI 的实现和标准 Android 存在部分差异，因此，在华为 10 上可能存在无法打开摄像机的问题。您需要执行以下操作以解决这个问题。

1. 升级基线

- 如果您采用的是 32 系列基线，则需要升级至 10.1.32.18 及以上。
- 如果您采用的是 60 系列基线，则需要升级至 10.1.60.9 及以上。
- 如果您采用的是 68 系列基线，则需要升级至 10.1.68-beta.3 及以上。

2. 配置 FileProvider

您可以复用您现有的 FileProvider，也可以新建一个 FileProvider。

1. 新建 Java 类，继承 FileProvider。

```
import android.support.v4.content.FileProvider;
public class NebulaDemoFileProvider extends FileProvider {
}
```

2. 在 res/xml 中新建 nebula_fileprovider_path.xml。

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">
 <external-path name="external" path="."/>
</paths>
```

3. 在 AndroidManifest 中加入配置。

```
<provider
 android:name="com.mpaas.demo.nebula.NebulaDemoFileProvider"
 android:authorities="com.mpaas.demo.nebula.provider"
 android:exported="false"
 android:grantUriPermissions="true">
 <meta-data
     android:name="android.support.FILE_PROVIDER_PATHS"
     android:resource="@xml/nebula_fileprovider_path" />
</provider>
```

> **说明**
>
> 此处 `android:authorities` 的值 `com.mpaas.demo.nebula.provider` 为 mPaaS 的代码示例信息，您需要根据自己的应用进行设置，并且不能设置为 `com.mpaas.demo.nebula.provider` ，以免与其他 mPaaS 应用产生冲突。

3. 实现 H5NebulaFileProvider

1. 新建 Java 类，实现 `H5NebulaFileProvider` ，实现 `getUriForFile` 方法，在该方法中，调用上面实现的 `FileProvider` 生成 URI。

```
public class H5NebulaFileProviderImpl implements H5NebulaFileProvider {
 private static final String TAG = "H5FileProviderImpl";

 @Override
 public Uri getUriForFile(File file) {
     try {
         return getUriForFileImpl(file);
     } catch (Exception e) {
         H5Log.e(TAG, e);
     }
     return null;
 }

 private static Uri getUriForFileImpl(File file) {
     Uri fileUri = null;
     if (Build.VERSION.SDK_INT >= 24) {
         fileUri =
NebulaDemoFileProvider.getUriForFile(LauncherApplicationAgent.getInstance().getApplicat
nContext(), "com.mpaas.demo.nebula.provider", file);
     } else {
         fileUri = Uri.fromFile(file);
     }
     return fileUri;
 }
}
```

2. 注册 `H5NebulaFileProvider` 。在 mPaaS 初始化完成之后，启动离线包之前，对 `H5NebulaFileProvider` 进行注册，注册一次即可全局生效。

```
H5Utils.setProvider(H5NebulaFileProvider.class.getName(), new
H5NebulaFileProviderImpl());
```

如何在 Library 中使用/依赖 mPaaS

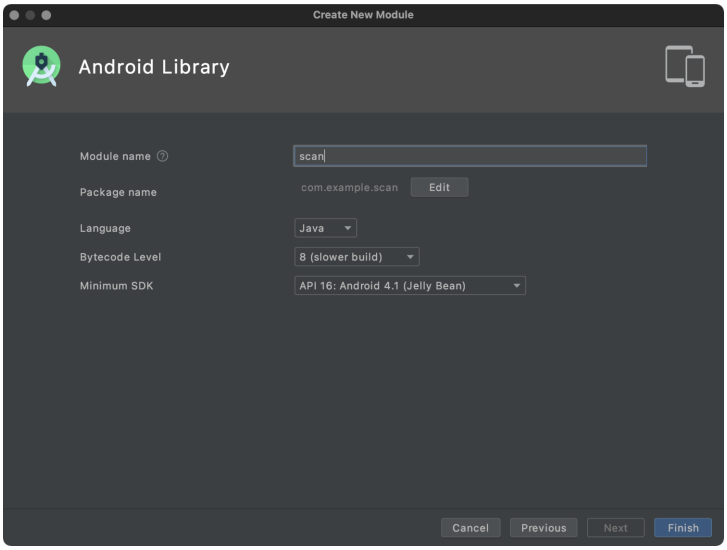
在使用 mPaaS 框架过程中，有时需要复用模块。复用时需要按照使用 Module 依赖的方式添加模块。本文以复用 mPaaS 扫码组件的 Module 为例进行说明。

前提条件

已按照原生 AAR 接入方式将工程接入 mPaaS。

操作步骤

1. 在 Android 工程中创建 Android Library 类型的模块 `scan` 。



2. 在新创建的 scan 模块的 build.gradle 文件中添加 `api platform("com.mpaas.android:$mpaas_artifact:$mpaas_baseline")` 。示例如下：

```
dependencies {
    ......
    //moudle 里使用 mPaaS 组件功能时，必须添加。
    api platform("com.mpaas.android:$mpaas_artifact:$mpaas_baseline")

    ......
 }
```

3. 通过 Android Studio mPaaS 插件为 scan 模块安装扫码组件。具体菜单路径为： **mPaaS > 原生 AAR 接入 > 配置/更新组件 > 开始配置**。安装后，扫码组件会自动加载。
4. 配置 App 主工程。

```
plugins {
    id 'com.android.application'

    ......
    //必须在 app 下的 build.gradle 文件中添加 baseline.config（基线）。
    id 'com.alipay.apollo.baseline.config'
}
```

5. 调用组件模块。在使用扫码组件的地方，导入 scan 模块。

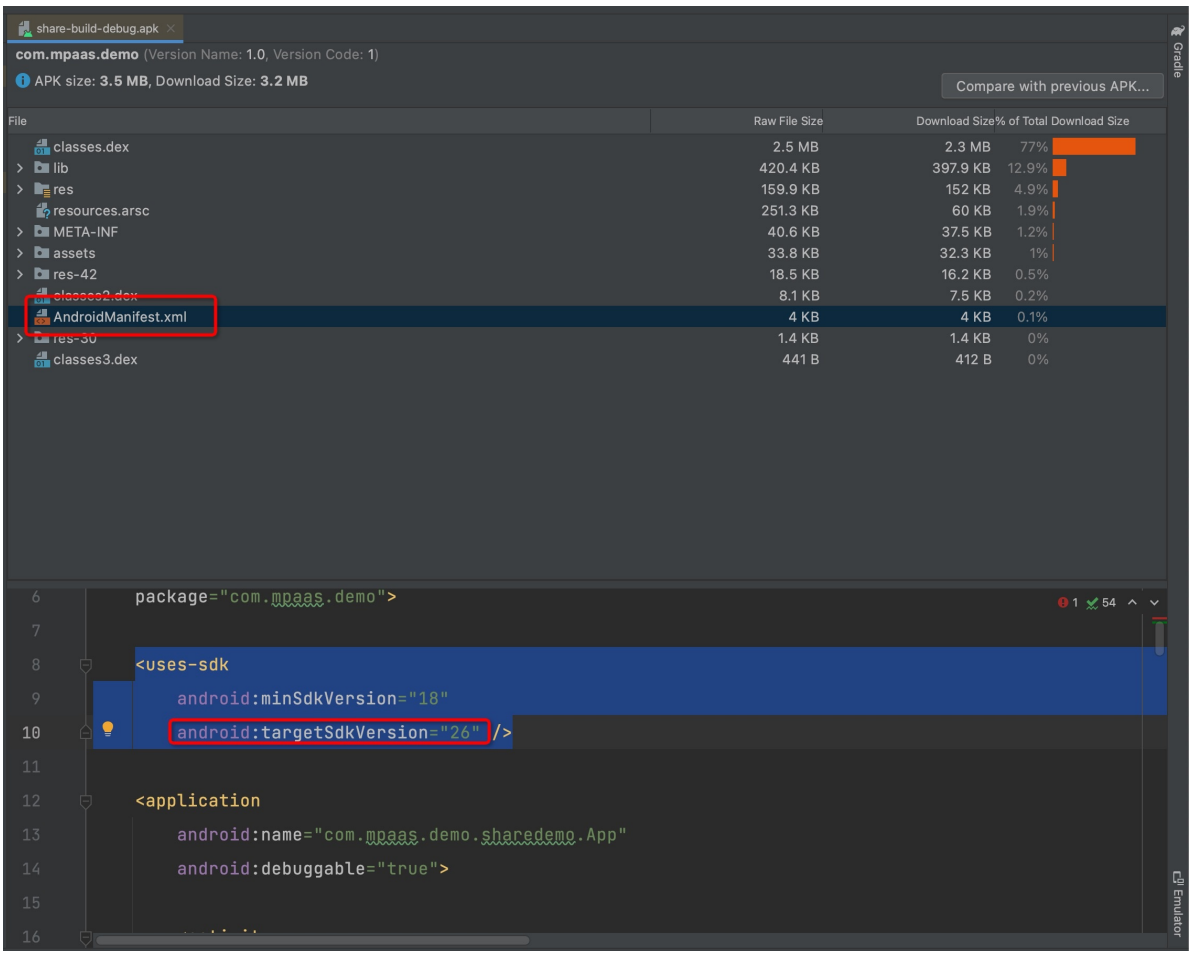
```
dependencies {
    api platform("com.mpaas.android:$mpaas_artifact:$mpaas_baseline")

    ....
    api project(':scan')//扫码组件
}
```

如何解决运行时出现的 608 错误或 libsgmain 的 native 错误

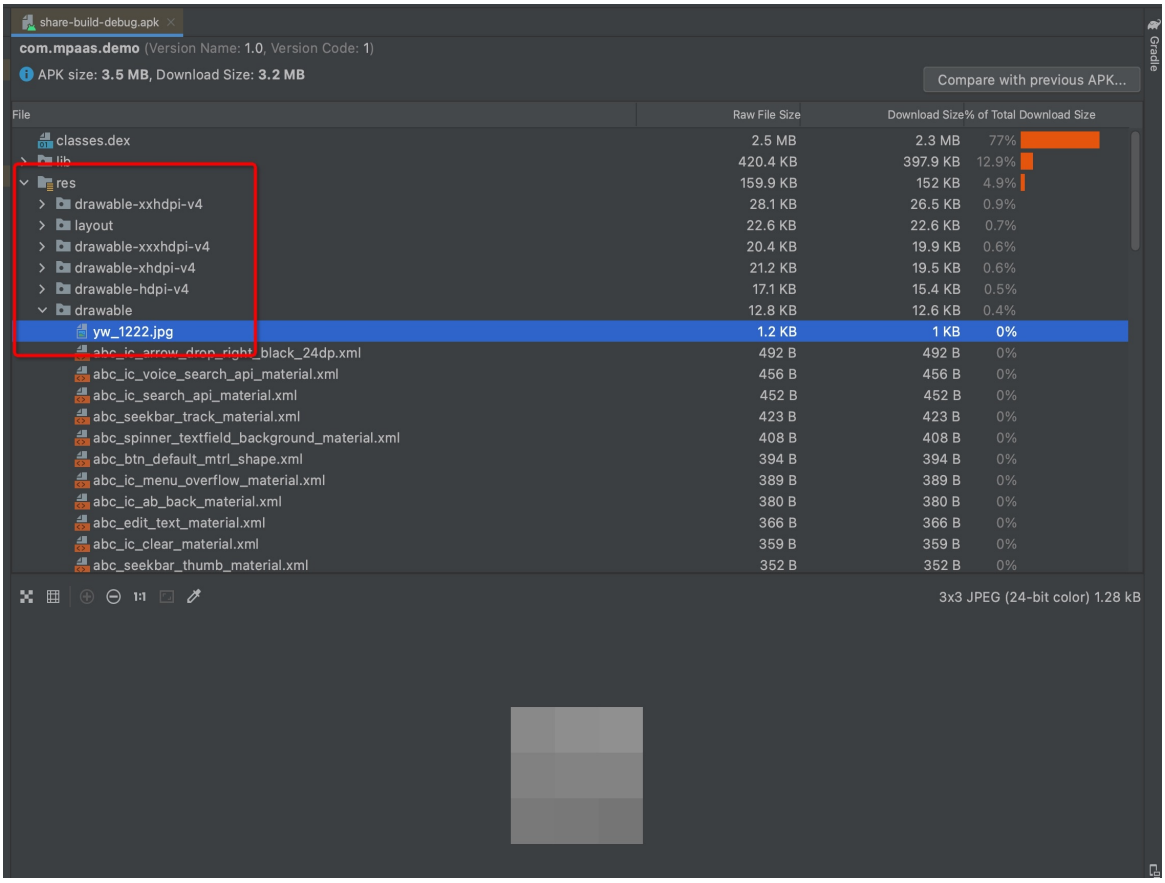
在运行时如果发生异常，在 Android Studio 运行日志中搜索关键字 `SecExcetpion` ，发现有 `608` 错误码或 libsgmain 的 native 错误，可以按照以下步骤进行排查。

1. 将 APK 直接拖放到 Android Studio 中，检查 Androidmenifest 文件中的 targetSdkVersion 是否为 26-28 之间的版本。



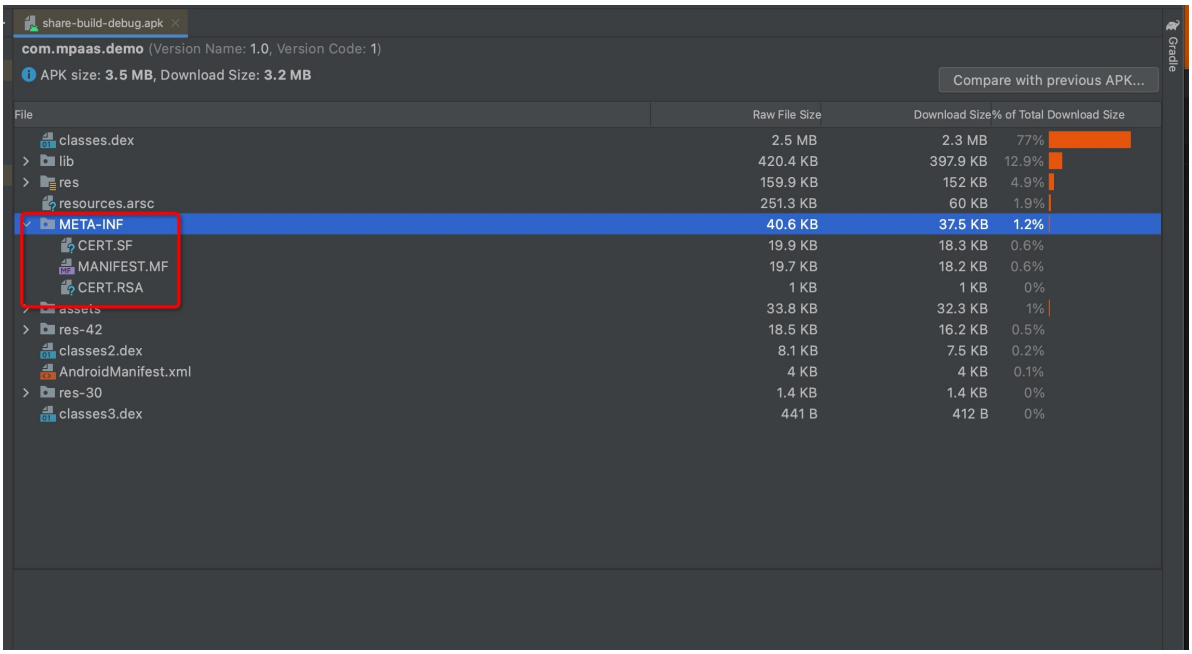
2. 检查是否存在 `res/drawable/yw_1222.jpg` 文件。
   - 检查 config 配置文件中是否有 Base64。

- 检查 Gradle 插件 `baseline.update` 或者 `baseline.config` 是否有应用。



3. 检查 META-INF 是否有 CERT.SF、MANIFEST.MF、和 CERT.RSA 等三个文件.

- 在 `app/build.gradle` 打开 `v1SignEnabled` 。
- 项目根目录下的 `build.gradle` 中是否有 `apply plugin: 'com.alipay.apollo.optimize'` 。



执行以上检查步骤后，且确认结果检查无误，则说明在控制台上传的签过名的 APK 包有问题，如签名错误，需重新上传 APK 包。