ALIBABA CLOUD

阿里云

移动推送

开发指南

文档版本：20220708

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 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。阿里云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，并在阿里云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阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

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.Android SDK手册

1.1. Android SDK版本说明

本文介绍Android SDK的发布记录和版本关系，并提供最新版本Maven依赖示例和最新版本SDK包文件示例供您参考。

SDK版本关系

> 注意
>
> - 使用移动推送Android SDK 3.2.0或以上版本，需同时将辅助通道SDK版本升级到V3.2.0或以上版本。
> - 移动推送Android SDK 3.2.2版本开始，移动推送SDK下载包中会包含配置使用的辅助通道SDK离线包，请在EMAS控制台下载SDK。从3.2.4版本开始，辅助通道SDK的离线包不在包含华为SDK，华为SDK仅能通过华为仓库获取。从3.6.0版本开始，辅助通道SDK的离线包不在包含魅族SDK，魅族的SDK可以从魅族官网获取。
> - 谷歌辅助通道SDK3.7.7版本开始需要应用开启AndroidX和Java 1.8编译，如果不能开启，可以考虑使用3.7.6版本

移动推送SDK版本说明

版本	说明	时间
3.7.7	1. 升级华为SDK至6.3.0.304 2. 升级谷歌SDK至23.0.3，升级过程中部分参数有变化，请参考Google推送通道集成重新接入SDK。 3. 修复bug	2022-05-07
3.7.6	1. 调整错误码 2. 调整关键日志 3. 修复小米Android12 通知无法打开问题 4. 升级基础库，修复部分bug	2022-04-15
3.7.5	1. 增加部分广播组件权限限制	2022-03-15

版本	说明	时间
3.7.4	1. 升级华为SDK至6.3.0.302 2. 升级小米SDK至4.9.1 3. 升级oppo SDK至3.0.0 4. 升级vivo SDK至3.0.0.4 5. 新增推送通道重连、重置、状态获取监听API 6. 新增channel进程自启动逻辑的开关 7. 完善推送通道异常处理逻辑	2022-03-03
3.7.3	1. 升级vivo SDK 至 3.0.0.3 2. 支持通知样式定制	2021-12-29
3.7.2	1. 适配Android 12 2. 升级小米SDK至 4.8.2	2021-12-16
3.7.1	1. 减少敏感API调用 2. 修复自有通道_ALIYUN_NOTIFICATION_MSG_ID_字段错误 3. 降低厂商通道解析失败概率	2021-11-15
3.7.0	1. 修复大图模式下，LargeIcon显示异常 2. 细化错误码，提供更具体的错误信息 3. 修复连接回调正常，实际连接有可能失败的问题 4. 厂商通道上报，增加厂商通道SDK版本 5. 推送数据增加_ALIYUN_NOTIFICATION_MSG_ID_	2021-11-02
3.6.0	1. 自有通道支持服务侧大文本、大图和inbox模式配置 2. 自有通道支持服务侧notifyId配置 3. SDK侧透出推送msgId 4. 升级华为推送SDK至5.3.0.304 5. 升级小米推送SDK至4.8.1 6. 升级魅族推送SDK至4.1.4	2021-09-27
3.5.0	1. 支持服务下发通知右侧icon配置	2021-09-07

版本	说明	时间
3.4.0	1. 修改部分日志输出 2. 修复个别机型崩溃 3. 适配Android12	2021-05-25
3.3.0	1. 调整部分敏感API调用 2. 支持一些特殊场景	2021-03-19
3.2.5	1. 修复低版本Android禁用channel功能 2. 修复魅族辅助通道先初始化时的异常 3. 增加动态注册的appKey appSecret的接口	2021-01-13
3.2.4	1. 优化SDK启动方式 2. 提供禁用静默通道的能力 3. 优化权限声明 4. 优化组件声明 5. 升级华为通道SDK到5.0.2 6. 升级小米通道SDK到3.8.5 7. 升级魅族通道SDK到4.0.2	2020-12-13
3.2.3	1. 拆分各辅助通道SDK 2. 辅助通道不强制继承activity 3. 升级小米通道SDK版本到3.8.2	2020-11-17
3.2.2	1. 删除不需要的权限声明 2. 升级小米通道SDK版本到3.7.9 3. 升级魅族通道SDK版本到3.9.7 4. 升级OPPO通道SDK版本到2.1.0-fix	2020-09-28
3.2.1	修复本地服务接收数据Bug	2020-08-26
3.2.0	1. 通知默认展示时间 2. 通知默认不会合并展示 3. 提供自定义通知 4. 修复Bug	2020-08-22

版本	说明	时间
3.1.12	修复Bug	2020-07-31
3.1.11	修改Bug，提供设备的唯一性	2020-07-27
3.1.10	修复Bug	2020-07-20

> 说明
>
> 更多更新记录，请登录EMAS控制台，在SDK下载页面单击版本号查看。

辅助通道SDK版本关系说明：

每一行是匹配的版本，一般情况下请使用最新的版本。不同行的版本不能混用。

移动推送	辅助通道	小米通道	华为通道	vivo通道	OPPO通道	魅族通道	Google
3.7.7	3.7.7	3.7.7(4.9.1)	3.7.7(6.3.0.304)	3.7.7(3.0.0.4)	3.7.7(3.0.0)	3.7.7(4.1.4)	3.7.7(23.0.3)
3.7.6	3.7.6	3.7.6(4.9.1)	3.7.6(6.3.0.302)	3.7.6(3.0.0.4)	3.7.6(3.0.0)	3.7.6(4.1.4)	3.7.6(17.6.0)
3.7.5	3.7.4	3.7.4(4.9.1)	3.7.4(6.3.0.302)	3.7.4(3.0.0.4)	3.7.4(3.0.0)	3.7.4(4.1.4)	3.7.4(17.6.0)
3.7.4	3.7.4	3.7.4(4.9.1)	3.7.4(6.3.0.302)	3.7.4(3.0.0.4)	3.7.4(3.0.0)	3.7.4(4.1.4)	3.7.4(17.6.0)
3.7.3	3.7.3	3.7.3(4.8.2)	3.7.3(5.3.0.304)	3.7.3(3.0.0.3)	3.7.3(2.1.0-fix)	3.7.3(4.1.4)	3.7.3(17.6.0)
3.7.2	3.7.2	3.7.2(4.8.2)	3.7.2(5.3.0.304)	3.7.2(2.9.0.1)	3.7.2(2.1.0-fix)	3.7.2(4.1.4)	3.7.2(17.6.0)
3.7.1	3.7.0	3.7.0（4.8.1）	3.7.0（5.3.0.304）	3.7.0(2.9.0.1)	3.7.0（2.1.0-fix）	3.7.0（4.1.4）	3.7.0(17.6.0)

移动推送	辅助通道	小米通道	华为通道	vivo通道	OPPO通道	魅族通道	Google
3.7.0	3.7.0	3.7.0（4.8.1）	3.7.0（5.3.0.304）	3.7.0(2.9.0.1)	3.7.0（2.1.0-fix）	3.7.0（4.1.4）	3.7.0(17.6.0)
3.6.0	3.6.0	3.6.0（4.8.1）	3.6.0（5.3.0.304）	3.4.0(2.9.0.1)	3.6.0（2.1.0-fix）	3.6.0（4.1.4）	3.6.0(17.6.0)
3.5.0	3.4.0	3.4.0（3.8.5）	3.4.0（5.0.4.302）	3.4.0(2.9.0.1)	3.4.0(2.1.0-fix)	3.4.0(4.0.2)	3.4.0(17.6.0)
3.4.0	3.4.0	3.4.0（3.8.5）	3.4.0（5.0.4.302）	3.4.0(2.9.0.1)	3.4.0(2.1.0-fix)	3.4.0(4.0.2)	3.4.0(17.6.0)
3.3.0	3.3.0	3.3.0（3.8.5）	3.3.0（5.0.4.302）	3.3.0(2.9.0.1)	3.3.0(2.1.0-fix)	3.3.0(4.0.2)	3.3.0(17.6.0)
3.2.5	3.2.5	3.2.5（3.8.5）	3.2.5（5.0.2）	3.2.5（2.9.0.1）	3.2.5（2.1.0-fix）	3.2.5（4.0.2）	3.2.5（17.6.0）
3.2.4	3.2.4	3.2.4（3.8.5）	3.2.4（5.0.2）	3.2.4（2.9.0.1）	3.2.4（2.1.0-fix）	3.2.4（4.0.2）	3.2.4（17.6.0）
3.2.3	3.2.3	3.2.3（3.8.2）	3.2.3（2.6.3.305）	3.2.3（2.9.0.1）	3.2.3（2.1.0-fix）	3.2.3（3.9.7）	3.2.3（17.6.0）
3.2.2	3.2.2	3.7.9	2.6.3.305	2.9.0.1	2.1.0-fix	3.9.7	17.6.0
3.2.0~3.2.1	3.2.0下载	辅助通道内置	2.6.3.305	2.9.0.1	辅助通道内置	3.8.7.1	17.6.0
3.1.0~3.1.12	3.1.0下载	辅助通道内置	2.6.3.305	2.9.0.1	辅助通道内置	3.8.7.1	17.6.0

> **说明**
>
> 移动推送 SDK从V3.2.3版本开始，我们拆分了辅助通道SDK，分为辅助通道（比如 com.aliyun.ams:alicloud-android-third-push:3.2.3）和厂商扩展包（比如 com.aliyun.ams:alicloud-android-third-push-huawei:3.2.3），上表中“3.2.3（3.8.2）”表示扩展包的版本号是3.2.3，它所依赖的厂商通道SDK为3.8.2。

最新版本Maven依赖示例

移动推送SDK Maven依赖：

```
compile 'com.aliyun.ams:alicloud-android-push:3.7.7'
```

辅助通道SDK Maven依赖：

```
//华为依赖
compile 'com.aliyun.ams:alicloud-android-third-push-huawei:3.7.7'
//小米依赖
compile 'com.aliyun.ams:alicloud-android-third-push-xiaomi:3.7.7'
//OPPO依赖
compile 'com.aliyun.ams:alicloud-android-third-push-oppo:3.7.7'
//vivo依赖
compile 'com.aliyun.ams:alicloud-android-third-push-vivo:3.7.7'
//魅族依赖
compile 'com.aliyun.ams:alicloud-android-third-push-meizu:3.7.7'
//谷歌依赖
compile 'com.aliyun.ams:alicloud-android-third-push-fcm:3.7.7'
```

最新版本SDK包文件示例

在控制台上下载移动推送SDK包，将包含以下文件：

```
├── alicloud-android-push-3.7.7.aar
├── alicloud-android-accs-4.6.3-emas.aar
├── alicloud-android-agoo-4.6.3-emas.aar
├── alicloud-android-beacon-1.0.7.jar
├── alicloud-android-crashdefend-0.0.6.jar
├── alicloud-android-error-1.1.0.aar
├── alicloud-android-logger-1.2.0.aar
├── alicloud-android-rest-1.6.5-open.aar
├── alicloud-android-sender-1.1.4.aar
├── alicloud-android-tool-1.0.0.jar
├── alicloud-android-utdid-2.6.0.jar
├── networksdk-3.5.8.6-open.jar
├── tnet4android-3.1.14.10-open-fix1.aar
//以下为辅助通道需要的SDK，jar/aar方式引入时使用，具体参考辅助通道配置说明。
//注意华为SDK从5.0.2版本开始不提供离线版本，只能从华为官方仓库获取
//魅族从4.1.4开始提供maven仓库，不在单独提供
├── alicloud-android-third-push-3.7.7.aar
├── alicloud-android-third-push-fcm-3.7.7.aar
├── alicloud-android-third-push-huawei-3.7.7.aar
├── alicloud-android-third-push-meizu-3.7.7.aar
├── alicloud-android-third-push-oppo-3.7.7.aar
├── alicloud-android-third-push-vivo-3.7.7.aar
├── alicloud-android-third-push-xiaomi-3.7.7.aar
├── mipush-4.9.1.jar
├── opush-3.0.0.aar
└── vivo-push-3.0.0.4.aar
```

1.2. Android SDK配置（V3.0.0及以上版本）

本章节介绍移动推送Android SDK V3.0.0及以上版本的集成操作。

前提条件

- 使用手动添加依赖方式需提前下载SDK包，请参见EMAS快速入门中的“下载SDK”。
- 已阅读Android SDK版本说明，获取最新版本对应关系。

样例代码

移动推送服务Android SDK接入工程样例请参见：移动推送Android Demo。

集成步骤

1. **添加依赖**

> 注意
>
> - 使用移动推送Android SDK 3.2.0或以上版本，需同时将辅助通道SDK版本升级到3.2.0或以上版本。
> - Maven库快速集成，配置简单，不容易出问题，后续更新方便，建议开发者采用此方式进行集成。

i. 方式一：Maven快速添加

在Project根目录build.gradle项目文件中进以下配置：

a. 在repositories{}代码段中配置Maven仓库地址。

```
allprojects {
    repositories {
        jcenter()
        maven {
            url 'http://maven.aliyun.com/nexus/content/repositories/releases/'
        }
        // 配置HMS Core SDK的Maven仓地址。
        maven {
            url 'https://developer.huawei.com/repo/'
        }
    }
}
```

b. 在android{}代码段中配置应用包名和NDK。

```
android {
    ......
    defaultConfig {
        applicationId "com.xxx.xxx" //包名
        ......
        ndk {
            //选择要添加的对应cpu类型的.so库。此处仅为示意，推送支持所有主流类型，请根据设备硬件选择
            abiFilters 'armeabi', 'x86'
        }
        ......
    }
    ......
}
```

错误处理：如果在添加以上abFilter配置后Android Studio出现以下提示：

```
NDK integration is deprecated in the current plugin. Consider trying the new experimental plugin.
```

则在Project根目录的gradle.properties文件中添加：

```
android.useDeprecatedNdk=true
```

c. 在dependencies{}代码段中添加SDK依赖。

> **说明**
>
> 以下代码为示例代码，请参考Android SDK版本说明使用最新SDK依赖版本。

```
dependencies {
    ......
    compile 'com.aliyun.ams:alicloud-android-push:3.x.x'
    ......
}
```

> **注意**
>
> 请使用固定版本号集成，不要使用动态版本号。错误示例：3.+或者3.2.+。

ii. 方式二：Android Studio手动集成

解压在控制台下载的SDK包，可将所有SDK拷贝到libs目录下，同时在build.gradle文件中进行如下配置：

> **说明**
>
> 以下代码为示例代码，请参考Android SDK版本说明使用最新SDK依赖版本。

```
android {
...
repositories {
    flatDir {
        dirs 'libs' //this way we can find the .aar file in libs folder
    }
}
dependencies {
compile fileTree(include: ['*.jar'], dir: 'libs')
compile (name:'alicloud-android-push-3.x.x', ext: 'aar')
compile (name:'alicloud-android-accs-4.x.x', ext: 'aar')
// 把所有的aar都添加上
...
// 华为从5.0.2版本开始不在提供离线包
compile 'com.huawei.hms:push:x.x.x.x'
// 魅族从4.1.4版本开始不在提供离线包
compile 'com.meizu.flyme.internet:push-internal:x.x.x'
}
```

2. **AndroidManifest配置**

i. **AppKey、AppSecret配置**

> 说明
>
> - 使用统一接入无需进行此配置。
> - 为避免在日志中泄漏参数 `appkey` / `appsecret` 或App运行过程中产生的数据，建议线上版本关闭SDK调试日志。
> - 由于所有用户使用统一的SDK接入，在接入过程中需要在代码中设置 `appkey` / `appsecret` 参数，而此类参数与计量计费密切相关，为防止恶意反编译获取参数造成信息泄漏，建议您开启混淆，并进行App加固后再发布上线。

在AndroidManifest文件中设置AppKey、AppSecret：

```
<application android:name="******">
    <meta-data android:name="com.alibaba.app.appkey" android:value="******"/> <!--
请填写你自己的- appKey -->
    <meta-data android:name="com.alibaba.app.appsecret" android:value="****"/> <!--
请填写你自己的appSecret -->
</application>
```

`com.alibaba.app.appkey` 和 `com.alibaba.app.appsecret` 为您在EMAS平台上的App对应信息。在EMAS控制台的应用管理中或在下载的配置文件中查看AppKey和AppSecret。

AppKey和AppSecret请务必写在application标签下，否则SDK会报找不到AppKey的错误。

> 说明
>
> 如果您是百川云推送用户，不能直接使用百川平台的AppKey和AppSecret，需要登录EMAS控制台，登录账号为您的百川平台账号，并使用阿里EMAS平台的 `appKey` ， `appSecret` 。

ii. 应用权限配置

> 说明
>
> 使用Maven库快速集成可跳过此步骤。

需要用户动态授权的权限，应用可以在用户签署隐私权限之后，向用户申请。

以下是应用自主添加的权限

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

WRITE_EXTERNAL_STORAGE、READ_EXTERNAL_STORAGE权限，用于存储一些设备标识，提高设备唯一性的判断，应用可以根据需要添加，注意需要应用在用户签署隐私政策之后，动态申请授权。

关于权限合规说明请参考本章节下方的权限合规说明。

iii. 消息接收Receiver配置

创建消息接收Receiver，继承自com.alibaba.sdk.android.push.MessageReceiver，并在对应回调中添加业务处理逻辑，可参考以下代码：

```
public class MyMessageReceiver extends MessageReceiver {
    // 消息接收部分的LOG_TAG
    public static final String REC_TAG = "receiver";
    @Override
    public void onNotification(Context context, String title, String summary, Map<String, String> extraMap) {
        // TODO处理推送通知
        Log.e("MyMessageReceiver", "Receive notification, title: " + title + ", summary: " + summary + ", extraMap: " + extraMap);
    }
    @Override
    public void onMessage(Context context, CPushMessage cPushMessage) {
            Log.e("MyMessageReceiver", "onMessage, messageId: " + cPushMessage.getMessageId() + ", title: " + cPushMessage.getTitle() + ", content:" + cPushMessage.getContent());
    }
    @Override
    public void onNotificationOpened(Context context, String title, String summary, String extraMap) {
        Log.e("MyMessageReceiver", "onNotificationOpened, title: " + title + ", summary: " + summary + ", extraMap:" + extraMap);
    }
    @Override
    protected void onNotificationClickedWithNoAction(Context context, String title, String summary, String extraMap) {
        Log.e("MyMessageReceiver", "onNotificationClickedWithNoAction, title: " + title + ", summary: " + summary + ", extraMap:" + extraMap);
    }
    @Override
    protected void onNotificationReceivedInApp(Context context, String title, String summary, Map<String, String> extraMap, int openType, String openActivity, String openUrl) {
        Log.e("MyMessageReceiver", "onNotificationReceivedInApp, title: " + title + ", summary: " + summary + ", extraMap:" + extraMap + ", openType:" + openType + ", openActivity:" + openActivity + ", openUrl:" + openUrl);
    }
    @Override
    protected void onNotificationRemoved(Context context, String messageId) {
        Log.e("MyMessageReceiver", "onNotificationRemoved");
    }
}
```

将该receiver添加到AndroidManifest.xml文件中：

```
<!-- 消息接收监听器 （用户可自主扩展） -->
<receiver
    android:name=".MyMessageReceiver"
    android:exported="false"> <!-- 为保证receiver安全，建议设置不可导出，如需对其他应用开放可通过android：permission进行限制 -->
    <intent-filter>
        <action android:name="com.alibaba.push2.action.NOTIFICATION_OPENED" />
    </intent-filter>
    <intent-filter>
        <action android:name="com.alibaba.push2.action.NOTIFICATION_REMOVED" />
    </intent-filter>
    <intent-filter>
        <action android:name="com.alibaba.sdk.android.push.RECEIVE" />
    </intent-filter>
</receiver>
```

> **注意**
>
> 旧版本 SDK集成说明：
>
> 如果是从V2.3.7及以下版本升级到V3.0.0及以上版本的用户，需将 `<action android:name="org.agoo.android.intent.action.RECEIVE" />` 改为 `<action android:name="com.alibaba.sdk.android.push.RECEIVE" />`，否则会接收不到推送。

iv. Android 8+适配

请参见Android 8.0以上设备接收不到推送通知。

3. **混淆配置**

如果您的项目中使用Proguard等工具做了代码混淆，请保留以下配置：

```
-keepclasseswithmembernames class ** {
    native <methods>;
}
-keepattributes Signature
-keep class sun.misc.Unsafe { *; }
-keep class com.taobao.** {*;}
-keep class com.alibaba.** {*;}
-keep class com.alipay.** {*;}
-keep class com.ut.** {*;}
-keep class com.ta.** {*;}
-keep class anet.**{*;}
-keep class anetwork.**{*;}
-keep class org.android.spdy.**{*;}
-keep class org.android.agoo.**{*;}
-keep class android.os.**{*;}
-keep class org.json.**{*;}
-dontwarn com.taobao.**
-dontwarn com.alibaba.**
-dontwarn com.alipay.**
-dontwarn anet.**
-dontwarn org.android.spdy.**
-dontwarn org.android.agoo.**
-dontwarn anetwork.**
-dontwarn com.ut.**
-dontwarn com.ta.**
```

4. **SDK初始化配置**

   首先通过PushServiceFactory获取到CloudPushService，然后调用register()初始化并注册推送通道，并确保在Application上下文中进行初始化工作。

   请参照以下代码进行初始化：

```
import android.app.Application;
import android.content.Context;
import android.util.Log;
import com.alibaba.sdk.android.push.CloudPushService;
import com.alibaba.sdk.android.push.CommonCallback;
import com.alibaba.sdk.android.push.noonesdk.PushServiceFactory;
public class MainApplication extends Application {
    private static final String TAG = "Init";
    @Override
    public void onCreate() {
        super.onCreate();
        initCloudChannel(this);
    }
    /**
     * 初始化云推送通道
     * @param applicationContext
     */
    private void initCloudChannel(Context applicationContext) {
        PushServiceFactory.init(applicationContext);
        CloudPushService pushService = PushServiceFactory.getCloudPushService();
        pushService.setLogLevel(CloudPushService.LOG_DEBUG);   //仅适用于Debug包，正式包不需要此行
        pushService.register(applicationContext, new CommonCallback() {
            @Override
            public void onSuccess(String response) {
                Log.d(TAG, "init cloudchannel success");
            }
            @Override
            public void onFailed(String errorCode, String errorMessage) {
                Log.d(TAG, "init cloudchannel failed -- errorcode:" + errorCode + " -- errorMessage:" + errorMessage);
            }
        });
    }
}
```

> **注意**
>
> PushServiceFactory.init必须在Application主线程中，不能放到Activity中执行，也不能异步初始化。移动推送在初始化过程中将启动后台进程channel，必须保证应用进程和channel进程都执行到PushServiceFactory.init。详情请参见Android SDK中CloudPushService应该怎样初始化？

5. **初始化成功验证**

   启动正常确认方法：

   - 回调方法callback.onSuccess()被调用。以上文中接入代码为例，logcat将会打印如下日志：

     ```
     11-24 12:55:51.096  15235-15535/com.alibaba.xxxx D/YourApp:  init cloudchannel success
     ```

   - 确认cloudchannel初始化正常，在logcat日志中，输入awcn关键字：

```
11-24 12:53:51.036  15235-15556/com.alibaba.xxxx E/awcn:  |[seq:AWCN1_1] AUTH httpSta
tusCode: 200
11-24 12:53:51.036  15235-15556/com.alibaba.xxxx E/awcn:  |[seq:AWCN1_1] status:AUTH_
SUCC
```

- 确认DeviceId获取正常：在初始化成功后使用 `cloudPushService.getDeviceId()` 可以成功获取deviceId。
- 如果注册服务器连接失败，则调用callback.onFailed方法，并且自动进行重新注册，直到onSuccess为止（重试规则会由网络切换等时机自动触发）。在onFailed方法中，会由相应的错误码返回，可参考错误处理。

> **说明**
>
> 市场上部分手机，对Log显示做了限制，比如华为手机，屏蔽了Debug和Verbose级别的日志；建议开发时使用Info级别日志。日志级别设置请参考Android SDK API介绍>设置日志等级。

权限合规说明

移动推送使用的权限用途说明

权限	业务用途
ACCESS_NETWORK_STATE ACCESS_WIFI_STATE READ_PHONE_STATE	用于判断网络状态，开启和关闭推送功能，针对区分不同的网络，缓存推送连接参数，使用不同的推送连接
INTERNET	用于连接推送服务，提供推送功能
WAKE_LOCK	用于维持低端手机推送通道的稳定性，提高推送消息的及时性
RECEIVE_BOOT_COMPLETED	用于维持重启手机后推送通道的稳定性，提高推送消息的及时性
READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE	用于存储推送通道的唯一标识，提高推送的准确性
GET_TASKS REORDER_TASKS	用于推送打开方式-打开应用功能，避免应用打开多次

以上是移动推送SDK的权限用途，可以在应用的隐私政策中进行说明，统一告知用户。

如何在用户签署隐私政策之前，避免获取用户敏感信息？

1. 目前最新版本，SDK内部已经在调用相关API之前，判断了相关权限是否授予，如果未授权，不调用，走默认的逻辑。

2. 我们对SDK初始化进行了调整，初始化主要分为两个API调用，一个在启动时调用，另一个可以延迟到用户签署隐私政策之后调用。
   - `PushServiceFactory.init(applicationContext);`：必须在Application的onCreate中执行，不会获取用户信息。
   - CloudPushService的 `void register(Context context, CommonCallback callback);`：可以在用户签署隐私政策之后调用。

   参考代码：

```
public class MyApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
        PushServiceFactory.init(this);

        // 获取隐私政策签署状态
        boolean sign = false;

        if(sign) {
            registerPush();
        } else {
            // 没签，等签署之后再调用registerPush()
        }
    }


    /**
     * 建立推送通道
     */
    public void registerPush() {
        CloudPushService pushService = PushServiceFactory.getCloudPushService();
        pushService.register(applicationContext, new CommonCallback() {
            @Override
            public void onSuccess(String response) {}

            @Override
            public void onFailed(String errorCode, String errorMessage) {}
        });
    }
}
```

SDK延迟初始化说明

为了尽量减少对应用的启动速度影响，我们调整了SDK的API，以便于应用可以将部分逻辑延迟执行，减少对应用启动速度的影响。

必须在Application onCreate中执行的逻辑

```
public class PushServiceFactory {
    /**
     * 初始化推送的变量和基本参数
     * @param appContext
     */
    public static void init(Context appContext) {}
}
```

此API用于初始化一些推送参数，没有启动推送逻辑，必须在Application onCreate中执行。

可以延迟执行的逻辑

```
/**
* PushSDK对外API
*/
public interface CloudPushService {
    /**
     * 初始化用户的SDK，将应用关联到云通道
     */
    void register(Context context, CommonCallback callback);
}
```

此API用于建立推送连接，可以根据业务需要延迟执行。

非手机场景说明

在一些特定设备上，使用场景和手机场景不同，可以进行一些特别的配置，可以分为两类，一类是类似手机应用，当用户使用时应用才会运行，用户不使用时，可能会被系统回收，另一类是应用是系统级别应用，会长时间运行。

类手机应用配置

这种场景，可以开启channel进程心跳，提高通道的稳定性。

```
    /**
     * 初始化云推送通道
     * @param applicationContext
     */
    private void initCloudChannel(Context applicationContext) {
        PushInitConfig.Builder builder = new PushInitConfig.Builder()
                .application(context);
        // 开启channel进程
        builder.disableChannelProcess(false);
        // 开启channel进程心跳
        builder.disableChannelProcessheartbeat(false);
        PushInitConfig config = builder.build();
        PushServiceFactory.init(config);

        CloudPushService pushService = PushServiceFactory.getCloudPushService();
        pushService.setLogLevel(CloudPushService.LOG_DEBUG);   //仅适用于Debug包，正式包不需要
此行
        pushService.register(applicationContext, new CommonCallback() {
            @Override
            public void onSuccess(String response) {
                Log.d(TAG, "init cloudchannel success");
            }
            @Override
            public void onFailed(String errorCode, String errorMessage) {
                Log.d(TAG, "init cloudchannel failed -- errorcode:" + errorCode + " -- erro
rMessage:" + errorMessage);
            }
        });
    }
```

如果系统不支持JobService（Android API版本低于21），还需要添加WAKE_LOCK权限。

```
<!-- 设备 Android API版本低于21 添加WAKE_LOCK权限 -->
<uses-permission android:name="android.permission.WAKE_LOCK"/>
```

长时间运行的系统应用

此类应用因为是一直运行的，根据系统性能要求，可以考虑不使用channel进程推送通道，完全使用应用内推送通道

```
    /**
     * 初始化云推送通道
     * @param applicationContext
     */
    private void initCloudChannel(Context applicationContext) {
        PushInitConfig.Builder builder = new PushInitConfig.Builder()
                .application(context);
        // 根据情况禁止channel进程
        builder.disableChannelProcess(true);
        // 禁止channel进程心跳
        builder.disableChannelProcessheartbeat(true);
        PushInitConfig config = builder.build();
        PushServiceFactory.init(config);

        CloudPushService pushService = PushServiceFactory.getCloudPushService();
        pushService.setLogLevel(CloudPushService.LOG_DEBUG);   //仅适用于Debug包，正式包不需要
此行
        pushService.register(applicationContext, new CommonCallback() {
            @Override
            public void onSuccess(String response) {
                Log.d(TAG, "init cloudchannel success");
            }
            @Override
            public void onFailed(String errorCode, String errorMessage) {
                Log.d(TAG, "init cloudchannel failed -- errorcode:" + errorCode + " -- erro
rMessage:" + errorMessage);
            }
        });
    }
```

长时间运行的应用，需要关注连接是否一直连接，可以注册自己的监听接口，并做一定的检查措施

```
                // 示意代码，请不要直接使用，请根据具体的业务情况使用api
                final Handler handler = new Handler();
                controlService.setConnectionChangeListener(new PushControlService.Connectio
nChangeListener() {
                    @Override
                    public void onConnect() {

                    }

                    @Override
                    public void onDisconnect(final String code, final String msg) {
                        final boolean isNetworkIssue = !isNetworkConnected();
                        // 过一段时间再检查，比如30s
                        handler.postDelayed(new Runnable() {
                            @Override
                            public void run() {
                                if(isNetworkConnected() && !controlService.isConnected()) {
                                    // 如果网络没有问题，而且连接没有恢复
                                    // 此时记录埋点 code 和 msg 信息，一定要记录msg信息
                                    recordDisconnectEvent(code, msg);
                                    if (isNetworkIssue) {
                                        // 刚才没有网络，尝试重连
                                        controlService.reconnect();
                                    } else {
                                        // 网络没有问题，或者重连也不行，进行重置，重新进行初始化
                                        controlService.reset();
                                        initCloudChannel(getContext());
                                    }

                                }
                            }
                        }, 30 * 1000);
                    }
                });
```

常见集成问题

1. UTDID冲突，可参考：阿里云-移动云产品SDK UTDID冲突解决方案
2. 集成Android SDK后运行App报java.lang.NoClassDefFoundError该如何解决？
3. Android端SDK集成失败排查
4. 初推送SDK在初始化时报错
5. Android SDK初始化时报1105、10207报错
6. 在Activity中初始化推送SDK，无法接收到推送通知

1.3. Android SDK API

1.3.1. 基本设置相关接口

> **说明**
>
> 以下接口调用时，如有回调均为异步执行，且回调不能为空。

SDK初始化

初始化推送SDK配置，必须在application的onCreate方法中执行，主进程和channel进程必须都执行到

方法定义

```
public static void init(Context appContext)
public static void init(PushInitConfig pushInitConfig)

// 初始化配置，针对需要多项配置的情况
public class PushInitConfig {
    public static class Builder {
        // application配置，必填
        public Builder application(Application application)
        // appKey 动态配置，使用manifest配置时 不用设置
        public Builder appKey(String appKey)
        // appSecret 动态配置，使用manifest配置时 不用设置
        public Builder appSecret(String appSecret)
        // 禁用channel进程配置，特殊场景使用，不必须，默认不禁用
        public Builder disableChannelProcess(boolean disableChannelProcess)
        // 禁用channel进程的自启动逻辑，默认禁用
        public Builder disableChannelProcessheartbeat(boolean disableChannelProcessHeartbea
t)
        // 定期循环拉起channel进程配置，特殊场景使用，不必须
        public Builder loopStartChannel(boolean enable)
        // 定期循环拉起channel进程间隔配置 单位ms，特殊场景使用，不必须
        public Builder loopInterval(long interval)
        // 构建初始化配置
        public PushInitConfig build()
    }
}
```

参数说明

参数	类型	是否必填	说明
context	Context	是	应用上下文（需要ApplicationContext）
appKey	String	否	当没有在manifest中配置appKey时，此处必须填写
appSecret	String	否	当没有在manifest中配置appSecret时，此处必须填写

参数	类型	是否必填	说明
enableChannelProcess	boolean	否	是否使用静默通道，默认是。 静默通道用于应用在后台时接收推送，会采取一些提高存活率的措施，如果只需要应用在前台时接收推送，可以不使用静默通道
pushInitConfig	PushInitConfig	是	替代其他init方法，统一所有参数配置到此类。

代码示例

```
// 一般情况
PushInitConfig config = new PushInitConfig.Builder()
        .application(this)
        .build();
PushServiceFactory.init(config);

// 需要代码动态配置appKey和appSecret
PushInitConfig config = new PushInitConfig.Builder()
        .application(this)
        .appKey("填入应用的appKey")
        .appSecret("填入应用的appSecret")
        .build();
PushServiceFactory.init(config);

// 特殊场景 需要禁止channel
PushInitConfig config = new PushInitConfig.Builder()
        .application(this)
        .disableChannelProcess(SpUtils.disableChannel(applicationContext))
        .build();
PushServiceFactory.init(config);

// 特殊场景 需要定时拉起channel
PushInitConfig config = new PushInitConfig.Builder()
        .application(this)
        .loopStartChannel(true)
        .loopInterval(30 * 1000)
        .build();
PushServiceFactory.init(config);
```

SDK注册

注册推送通道，开始接收推送。可以根据需要延迟注册，比如需要用户签署完隐私政策

方法定义

```
void register(Context context, CommonCallback callback);
```

参数说明

参数	类型	是否必填	说明
context	Context	是	应用上下文（需要ApplicationContext）
callback	CommonCallback	是	回调，错误码参见错误处理

代码示例

```
        pushService = PushServiceFactory.getCloudPushService();
        pushService.register(applicationContext, new CommonCallback() {
            @Override
            public void onSuccess(String response) {
                Log.i(TAG, "init cloudchannel success  " + pushService.getDeviceId());
                setConsoleText("init cloudchannel success  " + pushService.getDeviceId());
            }

            @Override
            public void onFailed(String errorCode, String errorMessage) {
                Log.e(TAG, "init cloudchannel failed -- errorcode:" + errorCode + " -- erro
rMessage:" + errorMessage);
                setConsoleText("init cloudchannel failed -- errorcode:" + errorCode + " --
errorMessage:" + errorMessage);
            }
        });
```

SDK动态注册（废弃）

支持动态设置AppKey、AppSecret的注册接口，与SDK注册功能相同。

> **注意**
>
> 3.2.4版本以后不支持此方法调用。如果要在代码中动态注册AppKey和AppSecret，请使用PushServiceFactory.init()带AppKey和AppSecret的初始化方法

方法定义

```
void register(Context context, String appKey, String appSecret, CommonCallback callback);
```

参数说明

参数	类型	是否必填	说明
context	Context	是	应用上下文

参数	类型	是否必填	说明
appKey	String	是	阿里云推送平台的appKey
appSecret	String	是	阿里云推送平台的appSecret
callback	CommonCallback	是	回调，错误码参见错误处理

代码示例

```
        pushService = PushServiceFactory.getCloudPushService();
        pushService.register(applicationContext, "xxxxx", "xxxxxx", new CommonCallback() {
            @Override
            public void onSuccess(String response) {
                Log.i(TAG, "init cloudchannel success  " + pushService.getDeviceId());
                setConsoleText("init cloudchannel success  " + pushService.getDeviceId());
            }

            @Override
            public void onFailed(String errorCode, String errorMessage) {
                Log.e(TAG, "init cloudchannel failed -- errorcode:" + errorCode + " -- erro
rMessage:" + errorMessage);
                setConsoleText("init cloudchannel failed -- errorcode:" + errorCode + " --
errorMessage:" + errorMessage);
            }
        });
```

获取设备标识

获取设备唯一标识，指定设备推送时需要。

方法定义

```
String getDeviceId();
```

代码示例

```
String deviceId = PushServiceFactory.getCloudPushService().getDeviceId();
```

设置日志等级

在通道初始化之前设置日志等级，默认等级为CloudPushService.LOG_DEBUG。

方法定义

```
void setLogLevel(int logLevel);
```

参数说明

参数	类型	是否必填	说明
logLevel	int	是	设置日志等级，支持以下几种类型： CloudPushService.LOG_ERROR CloudPushService.LOG_INFO CloudPushService.LOG_DEBUG CloudPushService.LOG_OFF：关闭Log

代码示例

```
        pushService = PushServiceFactory.getCloudPushService();
        pushService.setLogLevel(CloudPushService.LOG_DEBUG);
```

打开推送通道

> **注意**
>
> - SDK版本V3.0.3及以上版本支持。
> - 用于在程序运行时动态打开推送通道，全量推送场景下，打开推送通道需要24小时生效，其他场景实时生效。

打开推送，推送默认就是打开状态，一般与关闭推送通道配套使用，只有关闭过才需要打开推送

方法定义

```
void turnOnPushChannel(CommonCallback callback);
```

参数说明

参数	类型	是否必填	说明
callback	CommonCallback	是	操作成功与否的回调

代码示例

```
        mPushService = PushServiceFactory.getCloudPushService();
        mPushService.turnOnPushChannel(new CommonCallback() {
             @Override
             public void onSuccess(String s) {
                 tvConsoleText.append("turn on push channel success\n");
             }

             @Override
             public void onFailed(String errorCode, String errorMsg) {
                 tvConsoleText.append("turn on push channel failed." +
                         "errorCode: " + errorCode + ", errorMsg:" + errorMsg + "\n");
             }
         });
```

关闭推送通道

> **注意**
>
> - SDK版本V3.0.3及以上版本支持。
> - 用于在程序运行时动态关闭推送通道，全量推送场景下，关闭推送通道需要24小时生效，其他场景实时生效。

关闭推送，注意此时并不是真正断开推送通道，而是告诉服务这个设备不接收推送了。关闭之后，即使重新初始化SDK也不会打开推送，需要主动调用打开推送，才会重新接收推送

方法定义

```
void turnOffPushChannel(CommonCallback callback);
```

参数说明

参数	类型	是否必填	说明
callback	CommonCallback	是	操作成功与否的回调

代码示例

```
        mPushService = PushServiceFactory.getCloudPushService();
        mPushService.turnOffPushChannel(new CommonCallback() {
             @Override
             public void onSuccess(String s) {
                 tvConsoleText.append("turn off push channel success\n");
             }

             @Override
             public void onFailed(String errorCode, String errorMsg) {
                 tvConsoleText.append("turn off push channel failed." +
                         "errorCode: " + errorCode + ", errorMsg:" + errorMsg + "\n");
             }
         });
```

查询推送通道状态

> **注意**
>
> - SDK版本V3.0.3及以上版本支持。
> - 用于在程序运行时查询当前推送通道状态，如果当前为打开状态，则通过callback.success(String response)回调传入on，反之则传入off。

判断当前推送打开关闭状态

方法定义

```
void checkPushChannelStatus(CommonCallback callback);

public interface CommonCallback {

    void onSuccess(String response);

    void onFailed(String errorCode, String errorMessage);
}
```

参数说明

参数	类型	是否必填	说明
callback	CommonCallback	是	查询结果回调，response为on表示推送通道打开，off表示推送通道关闭

示例代码

```
        mPushService = PushServiceFactory.getCloudPushService();
        mPushService.checkPushChannelStatus(new CommonCallback() {
            @Override
            public void onSuccess(String response) {
                if (response.equals("on")) {
                    // 当前是打开状态
                } else {
                    // 当前是关闭状态
                }
            }

            @Override
            public void onFailed(String errorCode, String errorMessage) {
            }
        });
```

设置消息接收IntentService

> 注意
>
> - SDK版本V3.0.10及以上版本支持。
> - 通过IntentService组件接收消息回调。
> - 设置后消息将通过该组件透出，不在通过MessageReceiver

设置接收推送的服务，服务需要继承AliyunMessageIntentService。默认使用广播的方式接收推送，设置之后会改为使用服务接收推送

方法定义

```
void setPushIntentService(Class messageIntentService);
```

参数说明

参数	类型	是否必填	说明
messageIntentService	Class	否	自定义接收消息IntentService的class，继承AliyunMessageIntentService。 null表示使用广播接收推送

代码示例

```
        mPushService = PushServiceFactory.getCloudPushService();
        mPushService.setPushIntentService(MyMessageIntentService.class);
```

应用内通道控制接口

> 注意
>
> - SDK版本V3.7.4及以上版本支持。
> - 此类接口用于一些需要连接强控制的场景，手机场景不需要

控制接口包括判断应用内通道是否连接、重连、重置、监听连接状态等接口。

监听连接状态接口和判断应用内通道是否连接接口主要用于辅助业务方判断连接的状态。

重连接口用于在业务方发现连接断开一定时间还未重连上时，主动尝试重连。

重置接口用于重置SDK内部的初始化状态，用于在主动重连后，连接也不会恢复时，重置SDK内部状态，然后调用SDK注册接口，重新注册推送通道

接口定义

```
public interface PushControlService {

    /**
     * 判断是否连接
     *
     * @return
     */
    boolean isConnected();

    /**
     * 监听连接状态
     *
     * @param listener
     */
    void setConnectionChangeListener(ConnectionChangeListener listener);

    /**
     * 在 连接断开的情况下，重试重连
     */
    void reconnect();

    /**
     * 重置 本地数据状态，之后可以重新 注册推送
     */
    void reset();

    public static interface ConnectionChangeListener {
        /**
         * 连接建立成功
         */
        void onConnect();

        /**
         * 连接断开
         * 注意如果要记录断开的具体原因，请一定记录msg信息
         * @param code
         * @param msg
         */
        void onDisconnect(String code, String msg);
    }
}
```

通过PushServiceFactory获取接口

```
public class PushServiceFactory {
    public static PushControlService getPushControlService()
}
```

代码示例

```
        final PushControlService controlService = PushServiceFactory.getPushControlService(
);
        controlService.reconnect();
```

接收SDK日志输出

> **注意**
>
> - 如果要输出日志到文件或者上传，注意不要包含debug和info级别日志，避免日志量过大，影响应用性能

注册日志接口，用于接收SDK日志信息，排查问题

接口定义

```
public class AmsLogger {
    public static void addListener(final LoggerListener lisn)
}
```

通过PushServiceFactory获取接口

```
public class PushServiceFactory {
    public static PushControlService getPushControlService()
}
public interface LoggerListener {

    public void d(String TAG, String msg, Throwable tr, int flag);

    public void i(String TAG, String msg, Throwable tr, int flag);

    public void w(String TAG, String msg, Throwable tr, int flag);

    public void e(String TAG, String msg, Throwable tr, int flag);

}
```

代码示例

```
    public static void registerWarnLog() {
        AmsLogger.addListener(new LoggerListener() {
            @Override
            public void d(String TAG, String msg, Throwable tr, int flag) {

            }

            @Override
            public void i(String TAG, String msg, Throwable tr, int flag) {

            }

            @Override
            public void w(String TAG, String msg, Throwable tr, int flag) {
                // 记录 warn日志
                log.w(TAG, msg, tr);
            }

            @Override
            public void e(String TAG, String msg, Throwable tr, int flag) {
                // 记录 error日志
                log.e(TAG, msg, tr);
            }
        });
    }
```

1.3.2. 账号（account）相关接口

> **说明**
>
> 以下接口调用时，如有回调均为异步执行，且回调不能为空。

绑定账号

将应用内账号和推送通道相关联，可以实现按账号的定点消息推送。

> **注意**
>
> - 设备只能绑定一个账号，同一账号可以绑定到多个设备。
> - 同一设备更换绑定账号时无需进行解绑，重新调用绑定账号接口即可生效。
> - 若业务场景需要先解绑后再绑定，在解绑账号成功回调中进行绑定操作，以此保证执行的顺序性。
> - 账号名长度最大支持64字节。

接口定义

```
void bindAccount(String account, CommonCallback callback);
```

参数说明

参数	类型	是否必须	说明
account	String	是	待绑定的账号名。
callback	CommonCallback	是	回调

代码示例

```
        mPushService = PushServiceFactory.getCloudPushService();
        mPushService.bindAccount(account, new CommonCallback() {
            @Override
            public void onSuccess(String s) {
                tvConsoleText.append("bind account " + account + " success\n");
            }

            @Override
            public void onFailed(String errorCode, String errorMsg) {
                tvConsoleText.append("bind account " + account + " failed." +
                        "errorCode: " + errorCode + ", errorMsg:" + errorMsg);
            }
        });
```

解绑账号

将应用内账号和推送通道取消关联。

接口定义

```
void unbindAccount(CommonCallback callback);
```

参数说明

参数	类型	是否必须	说明
callback	CommonCallback	是	成功失败回调

代码示例

```
        mPushService = PushServiceFactory.getCloudPushService();
        mPushService.unbindAccount(new CommonCallback() {
            @Override
            public void onSuccess(String s) {
                tvConsoleText.append("unbind account success\n");
            }

            @Override
            public void onFailed(String errorCode, String errorMsg) {
                tvConsoleText.append("bind account failed." +
                        "errorCode: " + errorCode + ", errorMsg:" + errorMsg + "\n");
            }
        });
```

1.3.3. 标签（tag）相关接口

> **说明**
>
> 以下接口调用时，如有回调均为异步执行，且回调不能为空。

绑定标签

绑定标签到指定目标。

> **注意**
>
> - 支持向设备、账号和别名绑定标签，绑定类型有参数target指定。
> - App最多支持定义1万个标签，单个标签支持的最大长度为129字符。
> - 绑定标签在10分钟内生效。

接口定义

```
void bindTag(int target, String[] tags, String alias, CommonCallback callback);
```

参数说明

参数	类型	是否必须	说明

参数	类型	是否必须	说明
target	int	是	目标类型可选值： • 1：本设备 • 2：本设备绑定的账号 • 3：别名 目标类型可选值（SDK版本V2.3.5及以上版本）： • CloudPushService.DEVICE_TARGET：本设备 • CloudPushService.ACCOUNT_TARGET：本账号 • CloudPushService.ALIAS_TARGET：别名
tags	String[]	是	标签（数组输入）
alias	String	否	别名，仅当target=3时生效
callback	CommonCallback	是	回调

代码示例

```
        mPushService = PushServiceFactory.getCloudPushService();
        mPushService.bindTag(CloudPushService.DEVICE_TARGET, new String[]{tag}, null, new C
ommonCallback() {
            @Override
            public void onSuccess(String s) {
                tvConsoleText.append("bind tag " + tag + " success\n");
            }

            @Override
            public void onFailed(String errorCode, String errorMsg) {
                tvConsoleText.append("bind tag " + tag + " failed." +
                        "errorCode: " + errorCode + ", errorMsg:" + errorMsg + "\n");
            }
        });
```

解绑标签

解绑指定目标的标签。

> 注意
>
> - 支持解绑设备、账号和别名的标签，解绑类型有参数target指定。
> - 解绑标签只是解除设备和标签的绑定关系，不等同于删除标签，即该App下标签依然存在，系统当前不支持删除标签。
> - 解绑标签在10分钟内生效。

接口定义

```
void unbindTag(int target, String[] tags, String alias, CommonCallback callback);
```

参数说明

参数	类型	是否必须	说明
target	int	是	目标类型可选值： • 1：本设备 • 2：本设备绑定的账号 • 3：别名 目标类型可选值（SDK版本V2.3.5及以上版本）： • CloudPushService.DEVICE_TARGET：本设备 • CloudPushService.ACCOUNT_TARGET：本账号 • CloudPushService.ALIAS_TARGET：别名
tags	String[]	是	标签（数组输入）
alias	String	否	别名，仅当target=3时生效
callback	CommonCallback	是	回调

代码示例

```
        mPushService = PushServiceFactory.getCloudPushService();
        mPushService.unbindTag(CloudPushService.DEVICE_TARGET, new String[]{tag}, null, new
CommonCallback() {
            @Override
            public void onSuccess(String s) {
                tvConsoleText.append("unbind tag " + tag + " success\n");
            }

            @Override
            public void onFailed(String errorCode, String errorMsg) {
                tvConsoleText.append("unbind tag " + tag + " failed." +
                        "errorCode: " + errorCode + ", errorMsg:" + errorMsg + "\n");
            }
        });
```

查询标签

查询目标绑定的标签，当前仅支持查询设备标签。

> **注意**
>
> 查询结果可从回调onSuccess(response)的response中获取。

接口定义

```
void listTags(int target, CommonCallback callback);
```

参数说明

参数	类型	是否必须	说明
target	int	是	目标类型可选值： • 1：本设备 目标类型可选值（SDK版本V2.3.5及以上版本）： • CloudPushService.DEVICE_TARGET：本设备
callback	CommonCallback	是	回调

代码示例

```
        mPushService = PushServiceFactory.getCloudPushService();
        mPushService.listTags(CloudPushService.DEVICE_TARGET, new CommonCallback() {
            @Override
            public void onSuccess(String response) {
                tvConsoleText.append("tags:" + response + " \n");
            }

            @Override
            public void onFailed(String errorCode, String errorMsg) {
                tvConsoleText.append("list tags failed. errorCode:" + errorCode + " errorMs
g:" + errorMsg);
            }
        });
```

1.3.4. 别名（alias）相关接口

> **说明**
>
> 以下接口调用时，如有回调均为异步执行，且回调不能为空。

添加别名

为设备添加别名。

> **注意**
>
> - 单个设备最多添加128个别名，同一个别名最多可被添加到128个设备。
> - 别名支持的最大长度为128字节。

接口定义

```
void addAlias(String alias, CommonCallback callback);
```

参数说明

参数	类型	是否必填	说明
alias	String	是	别名
callback	CommonCallback	是	结果回调

代码示例

```
        mPushService = PushServiceFactory.getCloudPushService();
        mPushService.addAlias(alias, new CommonCallback() {
            @Override
            public void onSuccess(String s) {
                tvConsoleText.append("add alias " + alias + " success\n");
            }

            @Override
            public void onFailed(String errorCode, String errorMsg) {
                tvConsoleText.append("add alias " + alias + " failed." +
                        "errorCode: " + errorCode + ", errorMsg:" + errorMsg + "\n");
            }
        });
```

删除别名

删除设备别名。

> **说明**
>
> 支持删除指定别名和删除全部别名。

接口定义

```
void removeAlias(String alias, CommonCallback callback);
```

参数说明

参数	类型	是否必填	说明
alias	String	否	alias = null or alias.length = 0 时，删除设备全部别名。
callback	CommonCallback	是	回调

代码示例

```
        mPushService = PushServiceFactory.getCloudPushService();
        mPushService.removeAlias(alias, new CommonCallback() {
            @Override
            public void onSuccess(String s) {
                tvConsoleText.append("remove alias " + alias + " success\n");
            }

            @Override
            public void onFailed(String errorCode, String errorMsg) {
                tvConsoleText.append("remove alias " + alias + " failed." +
                        "errorCode: " + errorCode + ", errorMsg:" + errorMsg + "\n");
            }
        });
```

查询别名

查询设备别名。

> **注意**
>
> - 查询结果可从回调onSuccess(response)的response中获取。
> - 从SDK版本V3.0.9开始，接口内部有5s短缓存，5s内多次调用只会请求服务端一次。

接口定义

```
void listAliases(CommonCallback callback);
```

参数说明

参数	类型	是否必填	说明
callback	CommonCallback	是	回调

代码示例

```
        mPushService = PushServiceFactory.getCloudPushService();
        mPushService.listAliases(new CommonCallback() {
            @Override
            public void onSuccess(String response) {
                tvConsoleText.append("aliases:" + response + " \n");
            }

            @Override
            public void onFailed(String errorCode, String errorMsg) {
                tvConsoleText.append("list aliases failed. errorCode:" + errorCode + " erro
rMsg:" + errorMsg);
            }
        });
```

1.3.5. 通知设置相关接口

> 说明
>
> 以下接口调用时，如有回调均为异步执行，且回调不能为空。

设置通知声音

设置推送通知声音文件路径。注意android 8.0以上需要使用NotificationChannel配置声音。

> 注意
>
> - 若不调用本接口，默认获取资源ID为 `R.raw.alicloud_notification_sound` 的资源文件。
> - 若没有获取到指定声音文件，取设备设置的消息声音。

接口定义

```
void setNotificationSoundFilePath(String filePath);
```

参数说明

参数	类型	是否必填	说明
filePath	String	是	Uri可以解析的字符串

代码示例

```
        int assignSoundId = getResources().getIdentifier(ASSIGN_NOTIFCE_SOUND, DEFAULT_RES_
SOUND_TYPE, PackageName);
        if (assignSoundId != 0) {
            String defaultSoundPath = DEFAULT_RES_PATH_FREFIX + getPackageName() + "/" + as
signSoundId;
            mPushService = PushServiceFactory.getCloudPushService();
            mPushService.setNotificationSoundFilePath(defaultSoundPath);
            Log.i(SETTING_NOTICE, "Set notification sound res id to R." + DEFAULT_RES_SOUND
_TYPE + "." + ASSIGN_NOTIFCE_SOUND);
            this.appendConsoleText("Set notification sound res id to R." + DEFAULT_RES_SOUN
D_TYPE + "." + ASSIGN_NOTIFCE_SOUND);
        } else {
            Log.e(SETTING_NOTICE, "Set notification sound path error, R."
                    + DEFAULT_RES_SOUND_TYPE + "." + ASSIGN_NOTIFCE_SOUND + " not found.");
            this.appendConsoleText("Set notification sound path error, R."
                    + DEFAULT_RES_SOUND_TYPE + "." + ASSIGN_NOTIFCE_SOUND + " not found.");
        }
```

设置通知栏图标

设置推送通知栏图标资源Bitmap。

> 注意
>
> - 若不调用本接口，默认获取ID为 `R.drawable.alicloud_notification_largeicon` 的资源文件。
> - 若没有获取到指定图标文件，取App启动图标。

接口定义

```
void setNotificationLargeIcon(Bitmap icon);
```

参数说明

参数	类型	是否必填	说明
icon	Bitmap	是	图标资源Bitmap。

代码示例

```
       int assignLargeIconId = getResources().getIdentifier(ASSIGN_NOTIFCE_LARGE_ICON, DEFA
ULT_RES_ICON_TYPE, PackageName);
        if (assignLargeIconId != 0) {
            Drawable drawable = getApplicationContext().getResources().getDrawable(assignLa
rgeIconId);
            if (drawable != null) {
                Bitmap bitmap = ((BitmapDrawable)drawable).getBitmap();
                mPushService = PushServiceFactory.getCloudPushService();
                mPushService.setNotificationLargeIcon(bitmap);
                Log.i(SETTING_NOTICE, "Set notification largeIcon res id to R." + DEFAULT_R
ES_ICON_TYPE + "." + ASSIGN_NOTIFCE_LARGE_ICON);
                this.appendConsoleText("Set notification largeIcon res id to R." + DEFAULT_
RES_ICON_TYPE + "." + ASSIGN_NOTIFCE_LARGE_ICON);
            }
        } else {
            Log.e(SETTING_NOTICE, "Set largeIcon bitmap error, R."
                    + DEFAULT_RES_ICON_TYPE + "." + ASSIGN_NOTIFCE_LARGE_ICON + " not found
.");
            this.appendConsoleText("Set largeIcon bitmap error, R."
                    + DEFAULT_RES_ICON_TYPE + "." + ASSIGN_NOTIFCE_LARGE_ICON + " not found
.");
        }
```

设置状态栏图标

设置推送状态栏图标资源id。

> 注意
>
> - 若不调用本接口，默认获取 `R.drawable.alicloud_notification_smallicon` 的资源文件。
> - 若没有获取到指定资源文件id，取App启动图标。

接口定义

```
void setNotificationSmallIcon(int iconId);
```

参数说明

参数	类型	是否必填	说明
iconId	int	是	图标资源id。

代码示例

```
        int defaultSmallIconId = getResources().getIdentifier(DEFAULT_NOTICE_LARGE_ICON, DE
FAULT_RES_ICON_TYPE, PackageName);
        if (defaultSmallIconId != 0) {
            mPushService = PushServiceFactory.getCloudPushService();
            mPushService.setNotificationSmallIcon(defaultSmallIconId);
            Log.i(SETTING_NOTICE, "Set notification smallIcon res id to R." + DEFAULT_RES_I
CON_TYPE + "." + DEFAULT_NOTICE_SMALL_ICON);
            this.appendConsoleText("Set notification smallIcon res id to R." + DEFAULT_RES_
ICON_TYPE + "." + DEFAULT_NOTICE_SMALL_ICON);
        } else {
            Log.e(SETTING_NOTICE, "Set notification smallIcon error, R." +
                    DEFAULT_RES_ICON_TYPE + "." + DEFAULT_NOTICE_SMALL_ICON + " not found."
);
            this.appendConsoleText("Set notification smallIcon error, R." +
                    DEFAULT_RES_ICON_TYPE + "." + DEFAULT_NOTICE_SMALL_ICON + " not found."
);
        }
```

设置免打扰时段

设置免打扰时间段，过滤所有通知与消息。

> 注意
>
> - 免打扰时段仅支持设置一次，多次调用以最后一次调用设置时段为准。
> - SDK版本V2.3.5以下，设置免打扰时间段为00:00-00:00，可取消免打扰功能。
> - 全天免打扰可以设置为“0:0-23:59”。
> - 免打扰时段设置对小米辅助弹窗通知无效。

接口定义

```
void setDoNotDisturb(int startHour, int startMinute, int endHour, int endMinute, CommonCallback callback);
```

参数说明

参数	类型	是否必须	说明
startHour	int	是	免打扰的起始时间（小时），24小时制，取值范围：0-23。
starMinute	int	是	免打扰起始时间（分钟），取值范围：0-59。
endHour	int	是	免打扰的结束时间（小时），24小时制，取值范围：0-23。
endMinute	int	是	免打扰结束时间（分钟），取值范围：0-59。

代码示例

```
mPushService = PushServiceFactory.getCloudPushService();
       mPushService.setDoNotDisturb(mStartHours, mStartMinutes, mEndHours, mEndMinutes, new CommonCallback() {
            @Override
            public void onSuccess(String s) {
                Log.d(TAG, "设置免打扰成功  " + mStartHours + ":" + mStartMinutes + "~" + mEndHours + ":" + mEndMinutes);
                Toast.makeText(NoDisturbanceActivity.this, "设置免打扰成功", Toast.LENGTH_SHORT).show();
            }

            @Override
            public void onFailed(String s, String s1) {
                Log.d(TAG, "设置免打扰失败-- errorcode:" + s + " -- errorMessage:" + s1);
                Toast.makeText(NoDisturbanceActivity.this, "设置免打扰失败", Toast.LENGTH_SHORT).show();
            }
        });
```

关闭免打扰功能

关闭免打扰功能。

> **注意**
>
> - 免打扰功能是默认关闭的。
> - 没有对应的开发免打扰功能接口，调用设置免打扰功能时间段功能后自动打开免打扰功能。
> - SDK版本V2.3.5及以上版本支持。

接口定义

```
void closeDoNotDisturbMode();
```

代码示例

```
mPushService = PushServiceFactory.getCloudPushService();
mPushService.closeDoNotDisturbMode();
```

删除所有通知

删除推送SDK创建的所有通知。厂商通道的通知无法控制删除。

> **注意**
>
> - 若需要实现精准删除特性通知，可在onNotification回调中获取通知id，自行删除。
> - SDK版本V2.3.7及以上版本支持。

接口定义

```
void clearNotifications();
```

代码示例

```
mPushService = PushServiceFactory.getCloudPushService();
mPushService.clearNotifications();
```

1.3.6. 自建通知统计上报接口

本接口主要针对统计用户自建通知（通过阿里云推送发送透传消息，并在onMessage回调红自定创建通知）的删除/点击事件上报，其相关实现可以参考移动推送Android SDK：透传消息+用户自建通知最佳实践，如果您直接通过阿里云推送通知，无需使用相关接口。

> **注意**
>
> SDK版本V3.0.6及以上版本支持调用当前页面接口进行统计上报。

> **说明**
>
> 以下接口调用时，如有回调均为异步执行，且回调不能为空。

自建通知点击上报接口

上报自建通知的点击事件，请确同一消息仅上报一次。

接口定义

```
void clickMessage(CPushMessage message);
```

参数说明

参数	类型	是否必填	说明
message	CPushMessage	是	要上报点击事件的消息示例。

代码示例

```
    public void onMessage(Context context, CPushMessage cPushMessage) {
        Log.i(REC_TAG, "收到一条推送消息 ： " + cPushMessage.getTitle() + ", content:" + cPus
hMessage.getContent());
        MainApplication.setConsoleText("收到一条推送消息 ： " + cPushMessage.getTitle() + ",
content:" + cPushMessage.getContent());
        PushServiceFactory.getCloudPushService().clickMessage(cPushMessage);
    }
```

自建通知删除上报接口

上报自建通知的删除事件，请确保同一消息仅上报一次。

接口定义

```
void dismissMessage(CPushMessage message);
```

参数说明

参数	类型	是否必填	说明
message	CPushMessage	是	要上报点击事件的消息示例。

代码示例

```
    public void onMessage(Context context, CPushMessage cPushMessage) {
        Log.i(REC_TAG, "收到一条推送消息 ： " + cPushMessage.getTitle() + ", content:" + cPus
hMessage.getContent());
        MainApplication.setConsoleText("收到一条推送消息 ： " + cPushMessage.getTitle() + ",
content:" + cPushMessage.getContent());
        PushServiceFactory.getCloudPushService().dismissMessage(cPushMessage);
    }
```

1.3.7. 短信通知相关接口

为提高信息的到达率和实效性，扩展推送的使用场景，我们推出了推送与短信的融合通知模式。开发者可以设置在一定时间内，如果用户未收到或未点击推送，通过短信补发通知用户。具体方案可参考：短信联动配置。

为了实现推送短信融合方案，需要在终端接入绑定/解绑电话号码。

> **注意**
>
> SDK版本V3.0.11及以上版本支持绑定电话号码。

> **说明**
>
> 以下接口调用时，如有回调均为异步执行，且回调不能为空。

绑定电话号码

将设备与电话号码绑定。

接口定义

```
void bindPhoneNumber(String phoneNumber, CommonCallback callback);
```

参数说明

参数	类型	是否必须	说明
phoneNumber	String	是	要绑定的电话号码
callback	CommonCallback	是	回调

代码示例

```
        mPushService = PushServiceFactory.getCloudPushService();
        mPushService.bindPhoneNumber(phoneNumber, new CommonCallback() {
            @Override
            public void onSuccess(String s) {
                tvConsoleText.append("bind phone number " + phoneNumber + " success\n");
            }

            @Override
            public void onFailed(String errorCode, String errorMsg) {
                tvConsoleText.append("bind phone number " + phoneNumber + " failed." +
                        "errorCode: " + errorCode + ", errorMsg:" + errorMsg + "\n");
            }
        });
```

解绑电话号码

解除当前设备与电话号码的绑定关系。

接口定义

```
void unbindPhoneNumber(CommonCallback callback);
```

参数说明

参数	类型	是否必填	说明
callback	CommonCallback	是	结果回调

代码示例

```
mPushService = PushServiceFactory.getCloudPushService();
  mPushService.unbindPhoneNumber(new CommonCallback() {
      @Override
      public void onSuccess(String s) {
          tvConsoleText.append("unbind phone number success\n");
      }

      @Override
      public void onFailed(String errorCode, String errorMsg) {
          tvConsoleText.append("bind phone number " + " failed." +
                  "errorCode: " + errorCode + ", errorMsg:" + errorMsg + "\n");
      }
  });
```

1.3.8. MessageReceiver/AliyunMessageIntentService相关接口

MessageReceiver/AliyunMessageIntentService

通过集成MessageReceiver，可以拦截通知，接收消息，获取推送中的扩展字段。或者在通知打开或删除的时候，切入进行后续处理。

如果调用setPushIntentService，则需集成com.alibaba.sdk.android.push.AliyunMessageIntentService，并覆写相关方法，AliyunMessageIntentService所有消息回调同MessageReceiver一致。

使用方法

- MessageReceiver
  - 继承com.alibaba.sdk.android.push.MessageReceiver;

- 在Manifest中找到原来MessageReceiver的配置，将上边的class替换成你自己的receiver（不要配置多个）。

```
<!--消息接收监听器-->
<receiver android: name="com.alibaba.sdk.android.push.MessageReceiver <-- 把这里替换成你自己的receiver">
            <intent-filter>
                <action android: name="com.alibaba.push2.action.NOTIFICATION_OPENED" />
            </intent-filter>
            <intent-filter>
                <action android: name="com.alibaba.push2.action.NOTIFICATION_REMOVED" />
            </intent-filter>
            <intent-filter>
                <action android: name="com.alibaba.sdk.android.push.RECEIVE" />
            </intent-filter>
</receiver>
```

- AliyunMessageIntentService
  - 继承com.alibaba.sdk.android.push.AliyunMessageIntentService并覆写相关方法
  - 在Manifest中注册该service

```
  <service android: name="MyPushIntentService" >
            <intent-filter>
                <action android: name="com.alibaba.push2.action.NOTIFICATION_OPENED" />
            </intent-filter>
            <intent-filter>
                <action android: name="com.alibaba.push2.action.NOTIFICATION_REMOVED" />
            </intent-filter>
            <intent-filter>
                <action android: name="com.alibaba.sdk.android.push.RECEIVE" />
            </intent-filter>
  </service>
```

消息类型接收回调

用于接收服务端推送的消息，消息不会弹窗，而是回调该方法。

接口定义

```
void onMessage(Context context, CPushMessage message);
```

参数说明

参数	类型	说明
context	Context	android组件上下文。

参数	类型	说明
message	CPushMessage	推送的消息。

通知类型接收回调

用于接收服务端推送的通知，通知会弹窗，通知接收通知以及点击通知等都有回调。

接口定义

- 通知接收回调
  - 客户端接收到通知后，回调该方法。
  - 可获取到并处理通知相关的参数。

```
void onNotification(Context context, String title, String summary, Map<String, String> extraMap);
```

参数说明

参数	类型	说明
context	Context	android组件上下文。
title	String	通知标题。
summary	String	通知内容。
extraMap	Map<String, String>	通知额外参数，包括部分系统自带参数： • `_ALIYUN_NOTIFICATION_ID_`(V2.3.5及以上)：创建通知对应id • `_ALIYUN_NOTIFICATION_PRIORITY_`(V2.3.5及以上)：创建通知对应id。默认不带，需要通过OpenApi设置。

- 通知打开回调
  - 打开通知时会回调该方法，通知打开上报由SDK自动完成。

```
void onNotificationOpened(Context context, String title, String summary, String extraMap);
```

参数说明

参数	类型	说明
context	Context	android组件上下文。
title	String	通知标题。
summary	String	通知内容。
extraMap	Map<String, String>	通知额外参数，包括部分系统自带参数： • `_ALIYUN_NOTIFICATION_ID_`(V2.3.5及以上)：创建通知对应id。 • `_ALIYUN_NOTIFICATION_PRIORITY_`(V2.3.5及以上)：创建通知对应id。默认不带，需要通过OpenApi设置。

- 无跳转逻辑通知打开回调
  - 打开无跳转逻辑（open=4）通知时回调该方法（v2.3.2及以上版本支持），通知打开上报由SDK自动完成。

```
void onNotificationClickedWithNoAction(Context context, String title, String summary, Strin
g extraMap);
```

参数说明

参数	类型	说明
context	Context	android组件上下文。
title	String	通知标题。
summary	String	通知内容。
extraMap	Map<String, String>	通知额外参数，包括部分系统自带参数： • `_ALIYUN_NOTIFICATION_ID_`(V2.3.5及以上)：创建通知对应id。 • `_ALIYUN_NOTIFICATION_PRIORITY_`(V2.3.5及以上)：创建通知对应id。默认不带，需要通过OpenApi设置。

- 通知删除回调
  - 删除通知时回调该方法，通知删除上报由SDK自动完成。

```
void onNotificationRemoved(Context context, String messageId);
```

参数说明

参数	类型	是否必填	说明
context	Context	是	android组件上下文。
messageId	String	是	删除通知的Id。

- 通知在应用内到达回调
  - 当用户创建自定义通知样式，并且设置推送应用内到达不创建通知弹窗时调用该回调，且此时不调用onNotification回调（v2.3.3及以上版本支持）参数。

```
void onNotificationReceivedInApp(Context context, String title, String summary, Map<String, String> extraMap, int openType, String openActivity, String openUrl);
```

参数说明

参数	类型	说明
context	Context	android组件上下文。
title	String	通知标题。
summary	String	通知内容。
extraMap	Map<String, String>	通知额外参数。
openType	int	原本通知打开方式，1：打开APP；2：打开activity；3：打开URL；4：无跳转逻辑。
openActivity	String	所要打开的activity的名称，仅当服务端参数openType=2时有效，其余情况为null。

参数	类型	说明
openUrl	String	所要打开的URL，仅当服务端参数openType=3时有效，其余情况为null。

1.3.9. 自定义通知样式相关接口

1.3.9.1. 概述

移动推送Android SDK支持用户自定义通知样式，用户可以设定自己的通知样式，涉及的内容包括通知的提醒方式（声音、震动、静默），通知在状态栏的显示图标，推送消息应用内到达时是否创建通知以及自定义通知布局文件等。

> 注意
>
> SDK V2.3.3及以上版本支持自定义样式通知。

客户端设置通知样式

用户利用SDK提供的自定义通知样式接口创建自定义样式通知。SDK中有两个自定义样式通知类：

1. `BasicCustomPushNotification`：支持用户设置基础样式，包括提醒方式、状态栏图标以及当推送消息到达时应用处于前台的情况下是否创建该通知等。
2. `AdvancedCustomPushNotification`：是`BasicCustomPushNotification`的子类，继承了`BasicCustomPushNotification`的所有方法，同时还可以设置通知样式布局文件。

每个样式都需要对应一个特定的整数类型ID，如果多个样式设置为同一个ID，则最后设置的样式有效。如果SDK没有找到对应ID的样式则会创建默认样式的通知。

样式只需设置一次，SDK会记住这个设置，在需要使用时加载对应样式。具体使用例子请参考Demo。

`BasicCustomPushNotification` 代码示例

```
BasicCustomPushNotification notification = new BasicCustomPushNotification();
notification.setRemindType(BasicCustomPushNotification.REMIND_TYPE_SOUND);
notification.setStatusBarDrawable(R.drawable.logo_yuanjiao_120);
boolean res = CustomNotificationBuilder.getInstance().setCustomNotification(1, notification
);
```

`AdvancedCustomPushNotification` 代码示例

```
AdvancedCustomPushNotification notification = new AdvancedCustomPushNotification(R.layout.n
otitfication_layout, R.id.m_icon, R.id.m_title, R.id.m_text);
notification.setServerOptionFirst(true);
notification.setBuildWhenAppInForeground(false);
boolean res = CustomNotificationBuilder.getInstance().setCustomNotification(2, notification
);
```

后端推送消息时添加自定义样式ID

客户端设置完成后，服务端在推送通知时需要利用OpenAPI指明对应的自定义样ID。

使用OpenAPI推送消息时设定特定样式的ID。

> **注意**
>
> 服务端不能设置样式，只能指定需要展现的样式ID，指定ID的样式必须在客户端已经进行设置，否则SDK会创建默认样式的通知。

```
final SimpleDateFormat dateFormat = new SimpleDateFormat("MM-dd HH:mm:ss");
final String date = dateFormat.format(new Date());
PushRequest pushRequest = new PushRequest();
// 推送目标
pushRequest.setAppKey(appKey);
pushRequest.setTarget("device"); //推送目标: device:推送给设备; account:推送给指定账号,tag:推送
给自定义标签; all: 推送给全部
pushRequest.setTargetValue("deviceId");
// 推送配置
pushRequest.setType(1); // 0:表示消息(默认为0), 1:表示通知
pushRequest.setTitle(date); // 消息的标题
pushRequest.setBody("PushRequest body"); // 消息的内容
pushRequest.setSummary("PushRequest summary"); // 通知的摘要
pushRequest.setAndroidNotificationBarType(2);//设置的通知样式ID，通知栏自定义样式范围0-100
// 推送配置: Android
pushRequest.setAndroidOpenType("1"); // 点击通知后动作,1:打开应用 2: 打开应用Activity 3:打开 ur
l
pushRequest.setAndroidExtParameters("{\"_NOTIFICATION_BAR_STYLE_\":\"2\"}");
```

1.3.9.2. BasicCustomPushNotification API

默认构造函数

BasicCustomPushNotification的默认构造函数，所有配置采用默认设置：通知方式采用震动+通知；NotificationFlag采用Notification.FLAG_AUTO_CANCEL，状态栏图标用的是android.R.drawable.stat_notify_chat。

接口定义

```
public BasicCustomPushNotification();
```

构造函数

接口定义

```
public BasicCustomPushNotification(int drawable, int flags, int remindType);
```

参数说明

参数	类型	是否必填	说明
drawable	int	是	状态栏图标
flags	int	是	NotificationFlags，支持系统Notification下的Flag参数
remindType	int	是	提醒方式类型，支持以下选择： • BasicCustomPushNotification.REMIND_TYPE_SILENT：静默； • BasicCustomPushNotification.REMIND_TYPE_VIBRATE：震动； • BasicCustomPushNotification.REMIND_TYPE_SOUND：声音； • BasicCustomPushNotification.REMIND_TYPE_VIBRATE_AND_SOUND：声音+震动

代码示例

```
new BasicCustomPushNotification(R.drawable.logo, Notification.FLAG_AUTO_CANCEL, BasicCustomPushNotification.REMIND_TYPE_VIBRATE_AND_SOUND);
```

获取状态栏图标

获取已设置的状态栏图标。

接口定义

```
public int getStatusBarDrawable()
```

代码示例

```
int drawable = (new BasicCustomPushNotification()).getStatusBarDrawable();
```

设置状态栏图标

更改状态栏图标设置。

接口定义

```
public void setStatusBarDrawable(int statusBarDrawable);
```

参数说明

参数	类型	是否必填	说明
statusBarDrawable	int	是	状态栏图标资源ID

代码示例

```
(new BasicCustomPushNotification()).setStatusBarDrawable(R.drawable.logo);
```

获取提醒方式

获取已经设置的提醒方式。

接口定义

```
public int getRemindType();
```

代码示例

```
int type = (new BasicCustomPushNotification()).getRemindType();
```

设置提醒方式

更改自定义通知的提醒方式。

接口定义

```
public void setRemindType(int remindType);
```

参数说明

参数	类型	是否必填	说明
remindType	int	是	提醒方式类型，支持以下选择： • BasicCustomPushNotification.REMIND_TYPE_SILENT：静默； • BasicCustomPushNotification.REMIND_TYPE_VIBRATE：震动； • BasicCustomPushNotification.REMIND_TYPE_SOUND：声音； • BasicCustomPushNotification.REMIND_TYPE_VIBRATE_AND_SOUND：声音+震动

代码示例

```
(new BasicCustomPushNotification()).setRemindType(BasicCustomPushNotification.REMIND_TYPE_VIBRATE_AND_SOUND);
```

获取Notification Flags参数

获取已经设置的notification flags参数。

接口定义

```
public int getNotificationFlags();
```

代码示例

```
int type = (new BasicCustomPushNotification()).getNotificationFlags();
```

设置Notification Flags参数

更改自定义通知的flags参数。

接口定义

```
public void setNotificationFlags(int notificationFlags);
```

参数说明

参数	类型	是否必填	说明
notificationFlags	int	是	支持系统自带的Notification Flag参数

代码示例

```
(new BasicCustomPushNotification()).setNotificationFlags(Notification.FLAG_AUTO_CANCEL);
```

获取是否服务端设置优先

利用OpenAPI或者阿里云推送控制台推送消息都可以设置提醒方式，当后端设置的提醒方式和自定义样式提醒方式冲突时，SDK根据 `serverOptionFirst` 参数来判断提醒方式策略。如果该参数为true，则采用后端设定的提醒方式；如果该参数为false，则采用自定义样式指定的提醒方式。默认为false

接口定义

```
public boolean isServerOptionFirst();
```

代码示例

```
boolean isServerOptionFirst = (new BasicCustomPushNotification()).isServerOptionFirst();
```

设置是否服务端优先

更改自定义通知的serverOptionFirst参数。

接口定义

```
public void setServerOptionFirst(boolean serverOptionFirst);
```

参数说明

参数	类型	是否必填	说明
serverOptionFirst	boolean	是	是否服务器配置优先。 • true：采用后端设定的提醒方式 • false：采用自定义样式指定的提醒方式 默认值为false。

代码示例

```
(new BasicCustomPushNotification()).setServerOptionFirst(true);
```

获取推送前台到达是否创建通知参数

当推送到达时，如果应用处在前台，用户可以用过自定义样式决定是否创建通知。默认是创建通知。

接口定义

```
public boolean isBuildWhenAppInForeground();
```

代码示例

```
boolean isBuildWhenAppInForeground = (new BasicCustomPushNotification()).isBuildWhenAppInFo
reground();
```

设置推送前台到达是否创建通知参数

更改当推送到达时应用处在前台的情况下是否创建通知的设置。

接口定义

```
public void setBuildWhenAppInForeground(boolean buildWhenAppInForeground);
```

参数说明

参数	类型	是否必填	说明
buildWhenAppInForeground	boolean	是	是否创建通知

代码示例

```
(new BasicCustomPushNotification()).setBuildWhenAppInForeground(false);
```

1.3.9.3. AdvanceCustomPushNotification API

`AdvancedCustomPushNotification` 是 `BasicCustomPushNotification` 的子类，继承了 `BasicCustomPushNotification` 的所有方法。

构造函数

`AdvancedCustomPushNotification` 类的构造函数， `AdvancedCustomPushNotification` 没有默认构造函数。

接口定义

```
public AdvancedCustomPushNotification( int view, int iconViewId, int titleViewId, int contentViewId);
```

参数说明

参数	类型	是否必填	说明
view	int	是	自定义通知布局文件ID。 **说明** Notification的自定义布局是RemoteViews，和其他RemoteViews一样，在自定义视图布局文件中，仅支持FrameLayout、LinearLayout、RelativeLayout三种布局。
iconViewId	int	是	自定义布局文件中icon的viewId。
titleViewId	int	是	自定义布局文件中title的viewId。
contentViewId	int	是	自定义布局文件中显示通知正文的viewId。

代码示例

```
AdvancedCustomPushNotification advancedCustomPushNotification = new AdvancedCustomPushNotification(R.layout.demo_notification_cus_notif, R.id.m_icon, R.id.m_title, R.id.m_text);
```

demo_notification_cus_notif.xml示例

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <ImageView
        android:id="@+id/m_icon"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_alignParentLeft="true"
        android:layout_margin="5dp"
        />
    <TextView
        android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:text="Advanced Notification"
        android:layout_alignParentRight="true"
        android:gravity="center_vertical"
        android:textSize="10sp"
        />

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_alignParentTop="true"
        android:layout_margin="5dp"
        android:layout_toRightOf="@id/m_icon"
        android:layout_toLeftOf="@id/text"
        android:orientation="vertical">

        <TextView
            android:id="@+id/m_title"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textColor="#2844DD"
            android:text="title"
            android:textSize="20sp" />

        <TextView
            android:id="@+id/m_text"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textColor="#AA3b5E"
            android:text="text"
            android:textSize="15sp" />
    </LinearLayout>

</RelativeLayout>
```

设置通知图标

设置通知栏中显示的图标，该图标显示在iconViewId所指定的控件中。

接口定义

```
public void setIcon(int icon);
```

参数说明

参数	类型	是否必填	说明
icon	int	是	icon图标资源ID。

代码示例

```
advancedCustomPushNotification.setIcon(R.id.m_icon);
```

获取通知图标

获取设置的通知图标。

接口定义

```
public int getIcon();
```

代码示例

```
int icon = advancedCustomPushNotification.getIcon();
```

1.3.9.4. CustomNotificationBuilder API

CustomNotificationBuilder用于注册用户设定好的自定义样式通知。

获取CustomNotificationBuilder API

CustomNotificationBuilder是单例类，必须通过指定接口来获取实例。

接口定义

```
public static CustomNotificationBuilder getInstance();
```

代码示例

```
CustomNotificationBuilder customNotificationBuilder = CustomNotificationBuilder.getInstance
();
```

注册自定义样式通知

用户创建好自定义样式通知后需要将其注册，并赋予其一个特定的ID。

接口定义

```
public  boolean setCustomNotification(int customNotificationId, BasicCustomPushNotification
notification);
```

参数说明

参数	类型	是否必填	说明
customNotificationId	int	是	所注册的自定义样式通知的ID，ID必须大于0。如果将多个不同的自定义样式通知赋予同一个ID，则最后注册的通知有效，其他的通知样式将会被覆盖。
notification	BasicCustomPushNotification / AdvancedCustomPushNotification 对象	是	创建的通知，该通知可以是BasicCustomPushNotification对象也可以是AdvancedCustomPushNotification对象，但是不能为null。

> **说明**
>
> 该方法会返回一个boolean类型的结果，如果返回true，则注册成功；反之则失败。

代码示例

```
CustomNotificationBuilder.getInstance().setCustomNotification(1, new BasicCustomPushNotific
ation());
```

1.4. 辅助通道集成

1.4.1. 概述

在国内Android生态中，推送通道都是由终端与云端之间的长链接来维持，严重依赖于应用进程的存活状态。如今一些手机厂家会在自家ROM中做系统级别的推送通道，再由系统分发给各个App，以此提高在自家ROM上的推送送达率。

相关概念

辅助通道

移动推送针对小米、华为、VIVO、OPPO、魅族、谷歌等设备管控较严的情况，分别接入了相应的设备厂商推送辅助通道以提高这些设备上的到达率。

> **说明**
>
> 移动推送优先选择自由通道进行推送消息下发，只有在自有通道断连时才会选择辅助通道下发消息。

辅助弹窗

辅助弹窗通过系统通道下发通知，可以在进程被杀死情况下推送成功。由于辅助弹窗通过对应设备上的推送通知实现，因而通过辅助弹窗下发的通知不会触发 `onNotification` 回调。当前移动推送已接入小米、华为、OPPO、VIVO、魅族辅助弹窗。

集成指导

辅助通道集成

- 小米辅助通道集成
- 华为辅助通道集成
- vivo辅助通道集成
- OPPO辅助通道集成
- 魅族辅助通道集成
- Google推送通道集成
- 厂商通道原生SDK集成

辅助弹窗接入

辅助弹窗接入

场景解析

辅助通道集成场景解析

1.4.2. 小米辅助通道集成

本章节介绍如何集成移动推送提供的小米辅助通道SDK。

前提条件

最新的小米开放平台是分开Push功能的，需要先在Push功能区开通/启用推送功能。

获取小米推送密钥

登录小米开发平台，注册您的App，得到注册应用的AppID、AppKey、AppSecret。

控制台配置密钥

登录移动推送控制台，设置您的小米推送密钥（AppSecret），设置方法参见配置厂商通道密钥。

辅助通道集成

> **警告**
>
> - 如果使用辅助通道扩展包V3.2.0及以上版本，需要将推送SDK升级到V3.2.0及以上版本。
> - 3.2.0及以上版辅助通道扩展包以aar形式透出，省却manifest文件配置，减少出错概率。
> - 3.2.0及之前版本，小米通道依赖包已内置在alicloud-android-third-push中，无需单独添加。

1. **准备工作**

   请阅读Android SDK版本说明，下载对应版本SDK或获取最新SDK配置信息。

2. **添加依赖**

   - 方式一（首选）：Maven集成

     项目顶层build.gradle中添加Maven仓库地址：

     ```
     allprojects {
         repositories {
             maven {
                 url 'http://maven.aliyun.com/nexus/content/repositories/releases/'
             }
         }
     }
     ```

     gradle添加依赖：

     ```
     dependencies {
         compile 'com.aliyun.ams:alicloud-android-third-push-xiaomi:x.x.x'
         compile 'com.aliyun.ams:alicloud-android-third-push:x.x.x'
     }
     ```

   - 方式二：手动集成

     解压下载好的辅助通道SDK扩展包，并将之放置到app module的libs路径下，并在app module的build.gradle文件中添加如下配置：

     ```
     repositories {
         flatDir {
             dirs 'libs' //this way we can find the .aar file in libs folder
         }
     }
     ...
     dependencies {
         ......
         //根据具体的版本添加依赖
         compile(name: 'alicloud-android-third-push-3.x.x', ext: 'aar')
         compile(name: 'alicloud-android-third-push-xiaomi-x.x.x', ext: 'aar')
         //3.2.2版本开始，需要单独添加小米sdk依赖，小米sdk是jar形式
         compile fileTree(include: ['*.jar'], dir: 'libs')
     }
     ```

3. **混淆配置**

   如果集成推送SDK的工程开启了代码混淆，需要添加以下辅助通道的Proguard配置。

   ```
   # 小米通道
   -keep class com.xiaomi.** {*;}
   -dontwarn com.xiaomi.**
   ```

4. **初始化**

   > **注意**
   >
   > 辅助通道注册务必在Application中执行且放在推送SDK初始化代码之后，否则可能导致辅助通道注册失败。

将以下代码加入您 `application.onCreate()` 方法中初始化通道。

```
// 注册方法会自动判断是否支持小米系统推送，如不支持会跳过注册。
MiPushRegister.register(applicationContext, "小米AppID", "小米AppKey");
```

本方法会自动判断是否支持小米系统推送，如不支持会跳过注册。

5. **Android 8+适配**

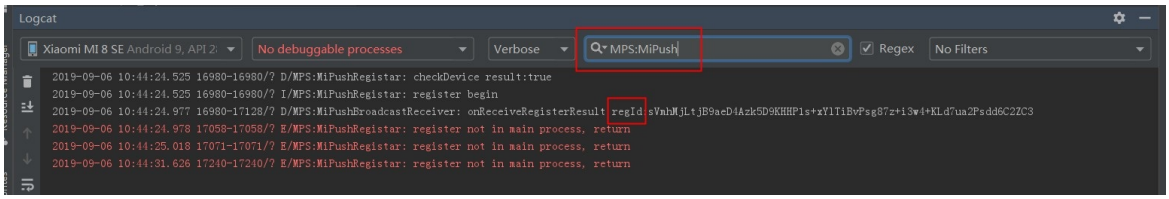
自Android 8.0（API Level 26）起，Android推出了NotificationChannel机制，旨在对通知进行分类管理。如果用户App的 `targetSdkVersion` 大于等于26，且并未设置 `NotificaitonChannel` ，那么创建的通知是不会弹出显示。

请参见Android 8.0以上设备接收不到推送通知进行适配。

6. **初始化成功验证**

> **警告**
>
> 查看初始化日志需将SDK日志等级设置为DEBUG，请参见SDK API介绍中的设置日志等级进行设置。

小米通道初始化成功，可看到以下日志：



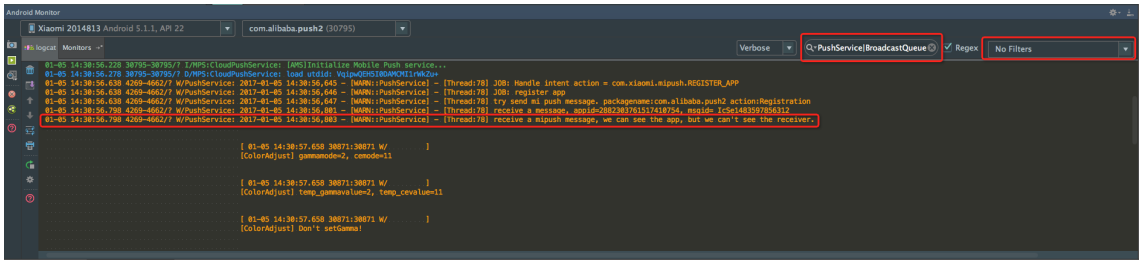
收到小米通道下行的消息：

```
12-09 22:24:34.065 19566-25042/com.xxx D/MPS:MiPushReceiver: onReceiveMessage,msg=[{"f"
:262,"b":"{\"content\"\ ... ... ,"i":"f__-rnje3_OH74gE|VG0g3kwMnGADAGrXZku1FFW5"}]
```

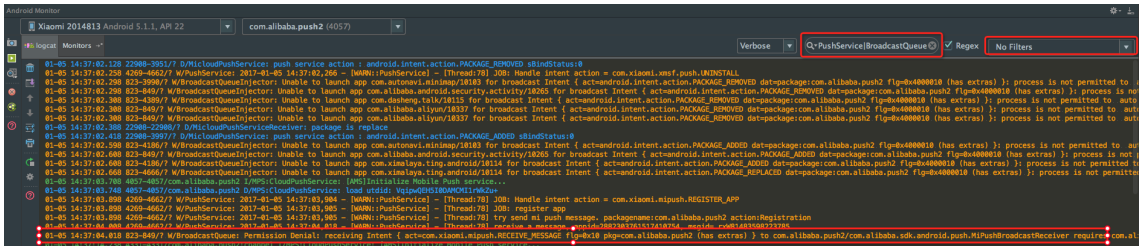
若小米通道注册失败（未看到小米注册成功日志），请查看系统日志（logcat设置为NoFiters）。

以PushService|BroadcastQueue为正则式进行过滤，示例如下图：

- MiPushBroadcastReceiver未配置：



- MIPUSH_RECEIVE权限未配置



使用辅助弹窗

厂商通道，除Google通道外，只能通过辅助弹窗来接收推送数据，详情请参见辅助弹窗接入。

> 注意
>
> - 使用移动推送进行厂商通道推送时（使用移动推送控制台或者OpenAPI进行推送时），服务端请必须参考辅助弹窗文档进行服务端配置，服务端参数不设置，不会给厂商通道进行推送。
> - Android SDK V2.3.0及以上版本支持小米辅助弹窗。

辅助通道常见问题

- Android端辅助通道和弹窗问题的排查步骤
- Android端辅助通道SDK与其他厂商SDK冲突
- Android端阿里云移动推送与其他注册厂商如何同时获取regId
- Android端辅助通道收到推送通知后单击通知无法打开相应Activity
- Android端辅助弹窗启动报解析body异常
- 在集成移动推送辅助通道后显示”register not in main process，return”
- 小米开放平台推送服务常见问题

1.4.3. 华为辅助通道集成

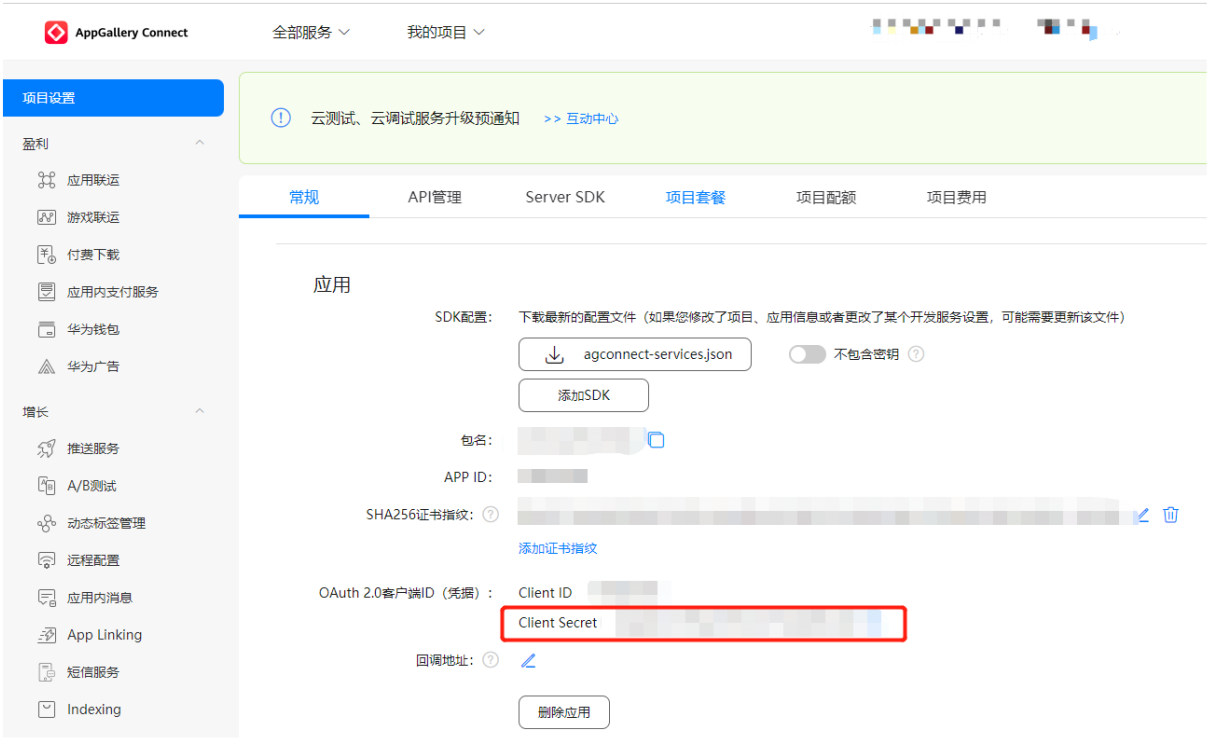
本章节介绍如何集成移动推送提供的华为辅助通道SDK。

获取华为推送密钥

登录华为开发者联盟，注册您的应用，在应用信息中获取APP ID和Secret Key。

> 说明
>
> 您可以在**我的应用**中查看应用APP ID和Secret Key。

如果在上图中未找到SecretKey，您也可以在下图中查看**OAuth2.0客户端ID（凭据）**中的**Client Secret**。



配置SHA256证书指纹

在华为开发者联盟配置SHA256证书指纹。获取及配置请参见华为官方文档配置AppGallery Connect。

设置消息回执

使用辅助弹窗后，华为通道的到达率统计覆盖用户点击华为弹窗推送通知的场景，未点击部分暂未覆盖，您可在华为端设置消息回执，便于移动推送更好的统计推送数据，具体设置方法请参见：消息回执。

开通回执需配置回调地址和HTTPS证书：

- 回调地址：https://amspush-ack.aliyuncs.com/hw/
- HTTPS证书：

```
-----BEGIN CERTIFICATE-----
MIIDdTCCAl2gAwIBAgILBAAAAAABFUtaw5QwDQYJKoZIhvcNAQEFBQAwVzELMAkG
A1UEBhMCQkUxGTAXBgNVBAoTEEdsb2JhbFNpZ24gbnYtc2ExEDAOBgNVBAsTB1Jv
b3QgQ0ExGzAZBgNVBAMTEkdsb2JhbFNpZ24gUm9vdCBDQTAeFw05ODA5MDExMjAw
MDBaFw0yODAxMjgxMjAwMDBaMFcxCzAJBgNVBAYTAkJFMRkwFwYDVQQKExBHbG9i
YWxTaWduIG52LXNhMRAwDgYDVQQLEwdSb290IENBMRswGQYDVQQDExJHbG9iYWxT
aWduIFJvb3QgQ0EwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDaDuaZ
jc6j40+Kfvvxi4Mla+pIH/EqsLmVEQS98GPR4mdmzxzdzxtIK+6NiY6arymAZavp
xy0Sy6scTHAHoT0KMM0VjU/43dSMUBUc71DuxC73/OlS8pF94G3VNTCOXkNz8kHp
1Wrjsok6Vjk4bwY8iGlbKk3Fp1S4bInMm/k8yuX9ifUSPJJ4ltbcdG6TRGHRjcdG
snUOhugZitVtbNV4FpWi6cgKOOvyJBNPc1STE4U6G7weNLWLBYy5d4ux2x8gkasJ
U26Qzns3dLlwR5EiUWMWea6xrkEmCMgZK9FGqkjWZCrXgzT/LCrBbBlDSgeF59N8
9iFo7+ryUp9/k5DPAgMBAAGjQjBAMA4GA1UdDwEB/wQEAwIBBjAPBgNVHRMBAf8E
BTADAQH/MB0GA1UdDgQWBBRge2YaRQ2XyolQL30EzTSo//z9SzANBgkqhkiG9w0B
AQUFAAOCAQEA1nPnfE920I2/7LqivjTFKDK1fPxsnCwrvQmeU79rXqoRSLblCKOz
yj1hTdNGCbM+w6DjY1Ub8rrvrTnhQ7k4o+YviiY776BQVvnGCv04zcQLcFGUl5gE
38NflNUVyRRBnMRddWQVDf9VMOyGj/8N7yy5Y0b2qvzfvGn9LhJIZJrglfCm7ymP
AbEVtQwdpf5pLGkkeB6zpxxxYu7KyJesF12KwvhHhm4qxFYxldBniYUr+WymXUad
DKqC5JlR3XC321Y9YeRq4VzW9v493kHMB65jUr9TU/Qr6cf9tveCX4XSQRjbgbME
HMUfpIBvFSDJ3gyICh3WZlXi/EjJKSZp4A==
-----END CERTIFICATE-----
```

控制台配置密钥

登录移动推送控制台，设置您的华为推送密钥（AppID和AppSecret），设置方法参见配置厂商通道密钥。

通道集成

> **警告**
>
> - 华为推送的2.X版本SDK将于2021年9月30日下线。为了避免推送受到影响，请尽快升级推送SDK及华为辅助通道SDK到3.2.4版本及以上。
> - 如果使用辅助通道扩展包**V3.2.0**及以上版本，需要将推送SDK升级到**V3.2.0**及以上版本。

1. **准备工作**

   请阅读Android SDK版本说明，下载对应版本SDK或获取最新SDK配置信息。

2. **添加依赖**

   > **说明**
   >
   > - 建议使用Maven集成。
   > - 3.2.0及以上版辅助通道扩展包以aar形式透出，省却manifest文件配置，减少出错概率。

   - 方式一：手动集成

     注意3.2.4及以上版本华为SDK不支持手动集成。

     解压下载好的辅助通道SDK扩展包，并将之放置到app module的libs路径下，并在app module的build.gradle文件中添加如下配置：

```
repositories {
    flatDir {
        dirs 'libs' //this way we can find the .aar file in libs folder
    }
    // 配置HMS Core SDK的Maven仓地址。
    maven {
        url 'https://developer.huawei.com/repo/'
    }
}
...
dependencies {
    ......
    compile(name: 'alicloud-android-third-push-x.x.x', ext: 'aar')
    compile(name: 'alicloud-android-third-push-huawei-x.x.x', ext: 'aar')
    compile 'com.huawei.hms:push:5.x.x.x'
}
```

- 方式二：Maven集成

  项目顶层build.gradle中添加Maven仓库地址：

```
allprojects {
    repositories {
        maven {
            url 'http://maven.aliyun.com/nexus/content/repositories/releases/'
        }
        // 配置HMS Core SDK的Maven仓地址。
        maven {
            url 'https://developer.huawei.com/repo/'
        }
    }
}
```

  gradle添加依赖：

```
dependencies {
    compile 'com.aliyun.ams:alicloud-android-third-push-huawei:x.x.x'
}
```

3. **混淆配置**

   如果集成推送SDK的工程开启代码混淆，需要添加以下辅助通道的Proguard配置。

```
# 华为通道
-keep class com.huawei.** {*;}
-dontwarn com.huawei.**
```

4. **初始化**

   i. 在AndroidManifest.xml中配置AppID，其中xxxxx为华为应用的AppID。

```
<meta-data
            android:name="com.huawei.hms.client.appid"
            android:value="appid=xxxxxx" />
```

ii. 将以下代码加入你application.onCreate()方法中初始化通道。

> 注意
>
> 辅助通道注册务必在Application中执行且放在推送SDK初始化代码之后，否则可能导致辅助通道注册失败。

```
// 注册方法会自动判断是否支持华为系统推送，如不支持会跳过注册。
HuaWeiRegister.register(application);
```

本方法会自动判断是否支持华为系统推送，如不支持会跳过注册。

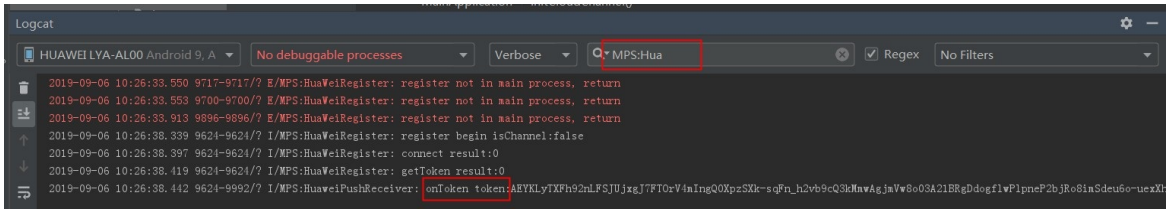
5. **Android 8+配置**

   自Android 8.0（API Level 26）起，Android推出了NotificationChannel机制，旨在对通知进行分类管理。如果用户App的targetSdkVersion大于等于26，且并未设置NotificaitonChannel，那么创建的通知是不会弹出显示。

   请参见Android 8.0以上设备接收不到推送通知进行适配。

6. **初始化成功验证**

   华为通道初始化成功，可以看到以下日志：



使用辅助弹窗

厂商通道，除Google通道外，只能通过辅助弹窗来接收推送数据，详情参考辅助弹窗接入。

> 注意
>
> - 在阿里云这边进行厂商通道推送时（使用移动推送控制台或者OpenAPI进行推送时），服务端请必须参考辅助弹窗文档进行服务端配置，服务端参数不设置，不会给厂商通道进行推送。
> - Android SDK V3.0.8及以上版本支持华为辅助弹窗。
> - 当前辅助弹窗功能在华为系统中仅支持Emotion UI（华为定制ROM）4.1及以上版本的设备。

辅助通道常见问题

- 华为通道注册时返回6003：证书指纹配置不对应，请再次进行“配置SHA256证书指纹”
- Android端辅助通道和弹窗问题的排查步骤
- Android端辅助通道SDK与其他厂商SDK冲突
- Android端阿里云移动推送与其他注册厂商如何同时获取regId
- Android端辅助通道收到推送通知后单击通知无法打开相应Activity
- Android端辅助弹窗启动报解析body异常
- 在集成移动推送辅助通道后显示”register not in main process，return”

1.4.4. vivo辅助通道集成

本章节介绍如何集成移动推送提供的vivo辅助通道SDK。

获取vivo推送密钥

登录vivo开放平台，注册您的应用，在应用信息中获取AppID、AppKey、AppSecret。

控制台配置密钥

登录移动推送控制台，设置您的vivo推送密钥（AppID和AppSecret），设置方法参见配置厂商通道密钥。

通道集成

1. **准备工作**

   请阅读Android SDK版本说明，下载对应版本SDK或获取最新SDK配置信息。

2. **添加依赖**

   > 注意
   > - 如果使用辅助通道扩展包V3.2.0及以上版本，需要将推送SDK升级到V3.2.0及以上版本。
   > - 建议使用Maven集成。
   > - 3.2.0及以上版辅助通道扩展包以aar形式透出，省却manifest文件配置，减少出错概率。

   - 方式一：手动集成

     解压下载好的辅助通道SDK扩展包，并将之放置到app module的libs路径下，并在app module的build.gradle文件中添加如下配置：

     ```
     repositories {
         flatDir {
             dirs 'libs' //this way we can find the .aar file in libs folder
         }
     }
     ...
     dependencies {
         ......
         compile(name: 'alicloud-android-third-push-x.x.x', ext: 'aar')
         compile(name: 'alicloud-android-third-push-vivo-x.x.x', ext: 'aar')
         compile(name: 'vivo-push-3.0.0.3', ext: 'aar')
     }
     ```

   - 方式二：Maven集成

     项目顶层build.gradle中添加Maven仓库地址：

```
allprojects {
    repositories {
        maven {
            url 'http://maven.aliyun.com/nexus/content/repositories/releases/'
        }
    }
}
```

gradle添加依赖：

```
dependencies {
    compile 'com.aliyun.ams:alicloud-android-third-push:x.x.x'
    compile 'com.aliyun.ams:alicloud-android-third-push-vivo:x.x.x'
}
```

3. **混淆配置**

   如果集成推送SDK的工程开启代码混淆，需要添加以下辅助通道的Proguard配置。

```
# VIVO通道
-keep class com.vivo.** {*;}
-dontwarn com.vivo.**
```

4. **初始化**

   i. 需要在AndroidManifest.xml中声明AppID和AppKey，示例如下：

```
<meta-data
    android:name="com.vivo.push.api_key"
    android:value="请填写vivo平台上注册应用的appKey" />
 <meta-data
    android:name="com.vivo.push.app_id"
    android:value="请填写vivo平台上注册应用的appID" />
```

ii. 需要在AndroidManifest.xml中，对辅助弹窗配置scheme，示例如下：

```
<activity
    android:name=".bizactivity.ThirdPushPopupActivity"
    android:exported="true"
    android:label="@string/title_activity_third_push_notice"
    android:launchMode="singleInstance"
    android:screenOrientation="portrait">
    <!-- 假设 ThirdPushPopupActivity 是用于接收厂商通道推送数据的辅助弹窗 -->
    <!-- scheme 配置开始 -->
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />

        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />

        <data
            android:host="${applicationId}"
            android:path="/thirdpush"
            android:scheme="agoo" />
    </intent-filter>
    <!-- scheme 配置结束 -->
</activity>
```

> **说明**
>
> ".bizactivity.ThirdPushPopupActivity" 类可参考辅助弹窗接入获取。

iii. 将以下代码加入您 `application.onCreate()` 方法中初始化通道。

> **注意**
>
> 辅助通道注册务必在Application中执行且放在推送SDK初始化代码之后，否则可能导致辅助通道注册失败。

```
// vivo通道注册
VivoRegister.register(applicationContext);
```

本方法会自动判断是否支持vivo系统推送，如不支持会跳过注册。

5. **Android 8+适配**

自Android 8.0（API Level 26）起，Android推出了NotificationChannel机制，旨在对通知进行分类管理。如果用户App的targetSdkVersion大于等于26，且并未设置NotificaitonChannel，那么创建的通知是不会弹出显示。

请参见Android 8.0以上设备接收不到推送通知进行适配。

6. **初始化成功验证**

vivo通道是否注册成功，可以通过过滤 `MPS:vPush` 关键字查看，注册成功会打印getRegId regId: **相关日志，否则检查参数是否正确填入。

使用辅助弹窗

厂商通道，除Google通道外，只能通过辅助弹窗来接收推送数据，详情参考辅助弹窗接入。

> 注意
>
> 使用移动推送进行厂商通道推送时（使用移动推送控制台或者OpenAPI进行推送时），服务端请必须参考辅助弹窗文档进行服务端配置，服务端参数不设置，不会给厂商通道进行推送。

辅助通道常见问题

vivo设备安装应用后默认不给通知权限，需引导用户手动开启。

Android端辅助通道和弹窗问题的排查步骤

Android端辅助通道SDK与其他厂商SDK冲突

Android端阿里云移动推送与其他注册厂商如何同时获取regId

Android端辅助通道收到推送通知后单击通知无法打开相应Activity

Android端辅助弹窗启动报解析body异常

在集成移动推送辅助通道后显示"register not in main process，return"

1.4.5. OPPO辅助通道集成

本章节介绍如何集成移动推送提供的小米辅助通道SDK。

获取OPPO推送密钥

登录OPPO开放平台，在推送服务中注册您的应用，在配置管理>应用配置中获取AppKey、AppSecret和MasterSecret。

控制台配置密钥

登录移动推送控制台，设置您的OPPO推送密钥（AppKey和MasterSecret），设置方法参见配置厂商通道密钥。

通道集成

1. **准备工作**

   请阅读Android SDK版本说明，下载对应版本SDK或获取最新SDK配置信息。

2. **添加依赖**

   > 注意
   >
   > 如果使用辅助通道扩展包**V3.2.0**及以上版本，需要将推送SDK升级到**V3.2.0**及以上版本。

> 说明
>
> - 建议使用Maven集成。
> - 3.2.0及以上版辅助通道扩展包以aar形式透出，省却manifest文件配置，减少出错概率。
> - 3.2.2版本之前，OPPO通道依赖包已内置在alicloud-android-third-push中，无需单独添加。

- 方式一：手动集成

  解压下载好的辅助通道SDK扩展包，并将之放置到app module的libs路径下，并在app module的build.gradle文件中添加如下配置：

```
repositories {
    flatDir {
        dirs 'libs' //this way we can find the .aar file in libs folder
    }
}
...
dependencies {
    ......
    compile(name: 'alicloud-android-third-push-x.x.x', ext: 'aar')
    compile(name: 'alicloud-android-third-push-oppo-x.x.x', ext: 'aar')
    compile(name: 'opush-2.1.0-fix', ext: 'aar')
}
```

- Maven集成

  项目顶层build.gradle中添加Maven仓库地址：

```
allprojects {
    repositories {
        maven {
            url 'http://maven.aliyun.com/nexus/content/repositories/releases/'
        }
    }
}
```

  gradle添加依赖：

```
dependencies {
    compile 'com.aliyun.ams:alicloud-android-third-push:x.x.x'
    compile 'com.aliyun.ams:alicloud-android-third-push-oppo:x.x.x'
}
```

3. **混淆配置**

   如果集成推送SDK的工程开启代码混淆，需要添加以下辅助通道的Proguard配置。

```
# OPPO通道
-keep public class * extends android.app.Service
```

4. **初始化**

   将以下代码加入您 `application.onCreate()` 方法中初始化通道。

> 注意
>
> 辅助通道注册务必在Application中执行且放在推送SDK初始化代码之后，否则可能导致辅助通道注册失败。

```
// OPPO通道注册
// appKey/appSecret在OPPO开发者平台获取
OppoRegister.register(applicationContext, appKey, appSecret);
```

本方法会自动判断是否支持OPPO系统推送，如不支持会跳过注册。

5. **Android 8+适配**

   自Android 8.0（API Level 26）起，Android推出了NotificationChannel机制，旨在对通知进行分类管理。如果用户App的targetSdkVersion大于等于26，且并未设置NotificaitonChannel，那么创建的通知是不会弹出显示。

   请参见Android 8.0以上设备接收不到推送通知进行适配。

6. **初始化成功验证**

   OPPO通道是否注册成功，可以通过过滤 `MPS:OPush` 关键字查看，注册成功会打印onRegister regid= **相关日志，否则检查参数是否正确填入。

使用辅助弹窗

厂商通道，除Google通道外，只能通过辅助弹窗来接收推送数据，详情参考辅助弹窗接入。

> 注意
>
> 1. 辅助弹窗功能的使用依赖于厂商通道，请确保已集成最近的辅助通道扩展包。
> 2. Android SDK V3.1.4及以上版本支持OPPO辅助弹窗。

辅助通道常见问题

Android端辅助通道和弹窗问题的排查步骤

Android端辅助通道SDK与其他厂商SDK冲突

Android端阿里云移动推送与其他注册厂商如何同时获取regId

Android端辅助通道收到推送通知后单击通知无法打开相应Activity

Android端辅助弹窗启动报解析body异常

在集成移动推送辅助通道后显示"register not in main process，return"

OPPO平台推送常见问题汇总

1.4.6. 魅族辅助通道集成

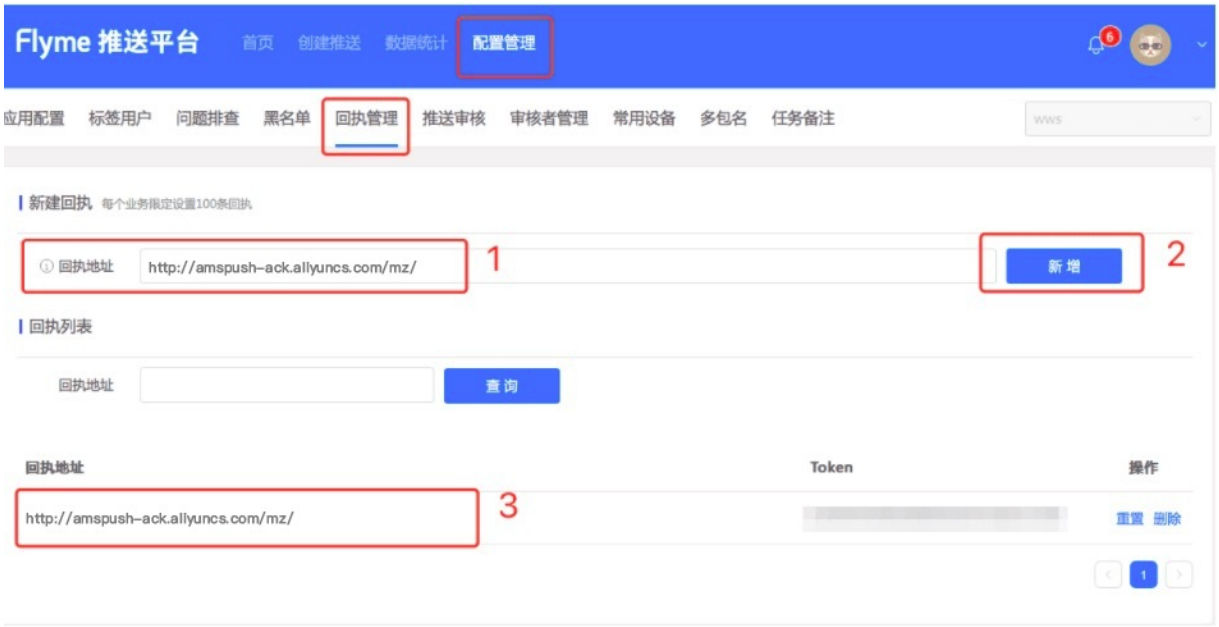
本章节介绍如何集成移动推送提供的魅族辅助通道SDK。

获取魅族推送密钥

登录魅族开放平台，在魅族消息推送服务中注册您的应用，在应用信息中获取AppID和AppSecret。

设置消息回执

在魅族推送平台的应用列表，单击打开应用，进入**配置管理>回执管理**页面。



设置回执地址为：http://amspush-ack.aliyuncs.com/mz/ 及 https://amspush-ack.aliyuncs.com/mz/ 否则可能会收不到推送消息。

控制台配置密钥

登录移动推送控制台，设置您的魅族推送密钥（AppID和AppSecret），设置方法参见配置厂商通道密钥。

通道集成

1. **准备工作**

   请阅读Android SDK版本说明，下载对应版本SDK或获取最新SDK的配置信息。

2. **添加依赖**

   > **注意**
   >
   > 如果使用辅助通道扩展包**V3.2.0**及以上版本，需要将推送SDK升级到**V3.2.0**及以上版本。

   > **说明**
   >
   > - 建议使用Maven集成。
   > - 3.2.0及以上版辅助通道扩展包以aar形式透出，省却manifest文件配置，减少出错概率。

   - 方式一：手动集成

     解压下载好的辅助通道SDK扩展包，并将之放置到app module的libs路径下，并在app module的build.gradle文件中添加如下配置：

```
repositories {
    flatDir {
        dirs 'libs' //this way we can find the .aar file in libs folder
    }
}
...
dependencies {
    ......
    compile(name: 'alicloud-android-third-push-x.x.x', ext: 'aar')
    compile(name: 'alicloud-android-third-push-meizu-x.x.x', ext: 'aar')
    compile(name: 'meizu-push-3.9.7', ext: 'aar')
}
```

- 方式二：Maven集成

  项目顶层build.gradle中添加Maven仓库地址：

```
allprojects {
    repositories {
        maven {
            url 'http://maven.aliyun.com/nexus/content/repositories/releases/'
        }
    }
}
```

  gradle添加依赖：

```
dependencies {
    compile 'com.aliyun.ams:alicloud-android-third-push:x.x.x'
    compile 'com.aliyun.ams:alicloud-android-third-push-meizu:x.x.x'
}
```

3. **混淆配置**

   如果集成推送SDK的工程开启代码混淆，需要添加以下辅助通道的Proguard配置。

```
# 魅族通道
-keep class com.meizu.cloud.** {*;}
-dontwarn com.meizu.cloud.**
```

4. **初始化**

   将以下代码加入您 `application.onCreate()` 方法中初始化通道。

> **注意**
>
> 辅助通道注册务必在Application中执行且放在推送SDK初始化代码之后，否则可能导致辅助通道注册失败。

```
// 魅族通道注册
// appId/appkey在魅族开发者平台获取
MeizuRegister.register(applicationContext, "appId", "appkey");
```

   本方法会自动判断是否主持魅族系统推送，如不支持会跳过注册。

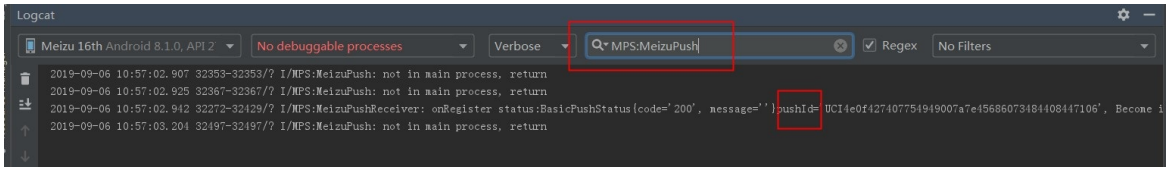
5. **Android 8+适配**

   自Android 8.0（API Level 26）起，Android推出了NotificationChannel机制，旨在对通知进行分类管理。如果用户App的targetSdkVersion大于等于26，且并未设置NotificaitonChannel，那么创建的通知是不会弹出显示。

   请参见Android 8.0以上设备接收不到推送通知进行适配。

6. **初始化成功验证**

   魅族通道初始化成功，可以看到以下日志：



使用辅助弹窗

厂商通道，除Google通道外，只能通过辅助弹窗来接收推送数据，详情参考辅助弹窗接入。

> **注意**
>
> 使用移动推送进行厂商通道推送时（使用移动推送控制台或者OpenAPI进行推送时），服务端请必须参考辅助弹窗文档进行服务端配置，服务端参数不设置，不会给厂商通道进行推送。

辅助通道常见问题

不进行“设置消息回执”，将不会给魅族通道推送。

Android端辅助通道和弹窗问题的排查步骤

Android端辅助通道SDK与其他厂商SDK冲突

Android端阿里云移动推送与其他注册厂商如何同时获取regId

Android端辅助通道收到推送通知后单击通知无法打开相应Activity

Android端辅助弹窗启动报解析body异常

在集成移动推送辅助通道后显示"register not in main process，return"

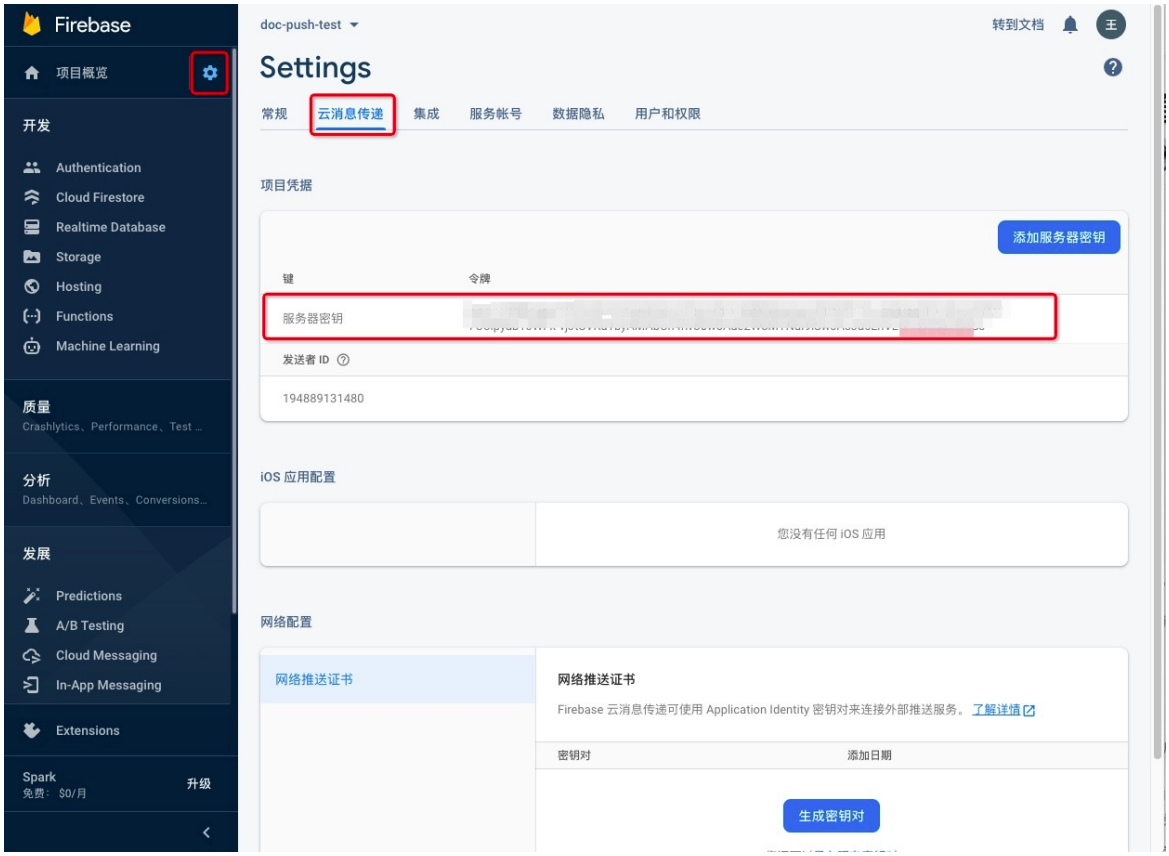
1.4.7. Google推送通道集成

本文介绍如何接入Google推送通道。

获取GCM/FCM服务器密钥

1. 在Firebase创建项目，在左侧导航栏选择Cloud Messaging，进入Google推送服务。
2. 在当前项目下新增App，下载对应App的google-services.json文件。

3. 记录2步骤中下载的JSON文件中的”project_number”，“mobilesdk_app_id”，“project_id”，“current_key”这四个key对应的value, 分别记录为sendId/applicationId/projectId/apiKey, 下文通道初始化过程中需要用到。
4. 在Firebase控制台的项目设置中获取服务器密钥。



控制台配置密钥

登录移动推送控制台，设置您的GCM/FCM服务器密钥，设置方法参见配置厂商通道密钥。

通道集成

1. **准备工作**

   请阅读Android SDK版本说明，下载对应版本SDK或获取SDK配置信息。

2. **添加依赖**

   > **注意**
   >
   > 如果使用辅助通道扩展包V3.2.0及以上版本，需要将推送SDK升级到V3.2.0及以上版本。

   > **说明**
   >
   > - 建议使用Maven集成。
   > - 3.2.0及以上版辅助通道扩展包以aar形式透出，省却manifest文件配置，减少出错概率。

   - 方式一：手动集成

     解压下载好的辅助通道SDK扩展包，并将之放置到app module的libs路径下，并在app module的build.gradle文件中添加如下配置：

```
repositories {
    flatDir {
        dirs 'libs' //this way we can find the .aar file in libs folder
    }
}
...
dependencies {
    ......
    compile(name: 'alicloud-android-third-push-x.x.x', ext: 'aar')
    compile(name: 'alicloud-android-third-push-fcm-x.x.x', ext: 'aar')
  compile ('com.google.firebase:firebase-messaging:23.0.3')
}
```

- 方式二：Maven集成

  项目顶层build.gradle中添加Maven仓库地址：

```
allprojects {
    repositories {
        maven {
            url 'http://maven.aliyun.com/nexus/content/repositories/releases/'
        }
    }
}
```

  gradle添加依赖：

```
dependencies {
    compile 'com.aliyun.ams:alicloud-android-third-push:x.x.x'
    compile 'com.aliyun.ams:alicloud-android-third-push-fcm:x.x.x'
}
```

3. **混淆配置**

   如果集成推送SDK的工程开启代码混淆，需要添加以下辅助通道的Proguard配置。

```
# GCM/FCM通道
-keep class com.google.firebase.**{*;}
-dontwarn com.google.firebase.**
```

4. **初始化**

> 注意
>
> - 接入前手机必须安装Google Paly Services，否则注册不成功，大部分国内Android手机的谷歌服务已被剥离。
> - 辅助通道注册务必在Application中执行且放在推送SDK初始化代码之后，否则可能导致辅助通道注册失败。
> - 移动推送SDK版本为3.7.7之前的用户，在升级过程中，需更新代码中的sendId、applicationId、projectId、apiKey参数，参数获取方式请参见：获取GCM/FCM服务器密钥。

将以下代码加入你application.onCreate()方法中初始通道。

```
//GCM/FCM辅助通道注册
GcmRegister.register(this, sendId, applicationId, projectId, apiKey); //sendId/applicat
ionId/projectId/apiKey为已获得的参数
```

本方法不会自动判断是否支持Google系统推送，需自行判断；或者在其他厂商通道初始化都返回false的情况下再执行。

5. **Android 8+适配**

   自Android 8.0（API Level 26）起，Android推出了NotificationChannel机制，旨在对通知进行分类管理。如果用户App的targetSdkVersion大于等于26，且并未设置NotificaitonChannel，那么创建的通知是不会弹出显示。

   请参见Android 8.0以上设备接收不到推送通知进行适配。

6. **初始化成功验证**

   GCM/FCM通道初始化成功，可以看到以下日志：

```
05-19 19:18:44.530 19153-19177/com.xxx D/MPS:GcmRegister: fcm token is eWIXLYCNP0Q:APA9
1bFUAgxj6XYf5okyoCBnRPw1UwITndzXrvPDgbdI2N44PYm17hFEBiNXNQJrJ8bOG_xjw3c3UPDAhzNMTLNjlAK
cjUanKyLA6E3k4wEmgZuhgUT02UMmMvH2LVA1L2Z4-l-cT_Ug
```

   收到GCM/FCM通道下发的消息：

```
05-19 19:20:04.900 19153-20391/com.alibaba.push2 D/MPS:GcmRegister: onReceiveMessage pa
yload msg:[......]
```

GCM/FCM场景说明

使用原生Android系统

- 常规使用时，应用前台运行，或者多任务管理中关闭应用（杀进程），或者应用后台运行，都可以使用GCM/FCM通道推送到。
- 在**设置**的**应用管理**中**强制关闭应用**，应用将无法自启动，所以此时推送无法收到，可以看到如下日志提示，意思是强制关闭的应用不能唤起。

```
GCM: broadcast intent callback: result=CANCELLED forIntent { act=com.google.android.c2dm.
intent.RECEIVE flg=0x10000000 pkg=com.aliyun.emas.pocdemo (has extras) }
```

- 开启**电池性能优化**时，可以设置是否允许应用后台运行：允许后台运行时，推送通道不受影响，应用是否杀进程都可以收到推送。不允许后台运行时，杀进程操作，系统会当成**强制关闭应用**来处理应用的运行状态，此时可能无法收到推送。

使用设备厂商的Android系统

国内部分厂商定制了Android系统，在这些系统中会把杀进程操作，当成**强制关闭应用**来处理，最终会导致GCM/FCM通道无法送达。此时建议接入对应厂商通道来提高应用杀进程后的推送到到达率，比如华为、小米等。

辅助通道常见问题

Android端辅助通道和弹窗问题的排查步骤

Android端辅助通道SDK与其他厂商SDK冲突

Android端阿里云移动推送与其他注册厂商如何同时获取regId

Android端辅助通道收到推送通知后单击通知无法打开相应Activity

Android端辅助弹窗启动报解析body异常

在集成移动推送辅助通道后显示“register not in main process，return”

1.4.8. 厂商通道原生SDK集成

本章节介绍当您自行接入厂商通道原生SDK时，如果使用移动推送需要的配置。

简介

厂商通道原生SDK集成，由您自行接入厂商提供的通道SDK和自行进行初始化。

> 说明
>
> 此方式不限制厂商通道SDK版本，不需要集成移动推送提供的厂商SDK。

该接入方式，不要求厂商通道SDK版本，不要求使用阿里云移动推送已经集成好的厂商SDK，需要自行接入厂商通道SDK和自行进行初始化，然后在厂商通道初始化成功后，将厂商的设备ID通过接口上传即可，主要针对以下场景：

- 使用厂商通道用于其他业务需求。
- 与其他已经集成好厂商通道的SDK发生依赖冲突，比如其他推送或即时通讯产品等。

通道集成

1. **准备工作**

   请阅读Android SDK版本说明，获取最新SDK配置信息。

2. **添加依赖**

   > 说明
   >
   > 本文档接入方式，下面添加的SDK最低使用3.2.3版本。

   - 方式一：手动集成

     将辅助通道扩展包放置到app module的libs路径下，并在app module的build.gradle文件中添加如下配置：

     ```
     repositories {
         flatDir {
             dirs 'libs' //this way we can find the .aar file in libs folder
         }
     }
     ...
     dependencies {
         ......
          //根据具体的版本添加依赖
         compile(name: 'alicloud-android-third-push-x.x.x', ext: 'aar')
         compile fileTree(include: ['*.jar'], dir: 'libs')
     }
     ```

   - 方式二：Maven配置

项目顶层build.gradle中添加Maven仓库地址：

```
allprojects {
    repositories {
        maven {
            url 'http://maven.aliyun.com/nexus/content/repositories/releases/'
        }
    }
}
```

gradle添加依赖：

```
dependencies {
    compile 'com.aliyun.ams:alicloud-android-third-push:x.x.x@aar'
}
```

3. **上传厂商设备ID**

在厂商通道初始化成功后，调用 `ThirdPushManager.reportToken()` 上传厂商设备ID：

```
/**
* 以华为为例
*
* @param  context 上下文对象，建议传ApplicationContext
* @param  thirdTokenKeyword 厂商设备ID的标识，下面做详细介绍
* @param  token 厂商设备ID，厂商设备ID叫法不同，此处以token为统称
*/
ThirdPushManager.reportToken(context, ThirdPushReportKeyword.HUAWEI.thirdTokenKeyword,
token);
```

`ThirdPushReportKeyword` 类介绍：

管理厂商关键字的枚举类，用于上报厂商设备ID以及收到消息类型的推送时选择关键字使用，目前支持并定义了华为、小米、OPPO、vivo、魅族、Google通道的关键字。

```
public static enum ThirdPushReportKeyword {
    HUAWEI("HW_TOKEN", "huawei"),
    XIAOMI("MI_TOKEN", "xiaomi"),
    OPPO("OPPO_TOKEN", "oppo"),
    VIVO("VIVO_TOKEN", "vivo"),
    MEIZU("MZ_TOKEN", "meizu"),
    FCM("gcm", "gcm");

    public String thirdTokenKeyword;//厂商的设备ID标识
    public String thirdMsgKeyword;//厂商的消息标识

    private ThirdPushReportKeyword(String thirdTokenKeyword, String thirdMsgKeyword) {
        this.thirdTokenKeyword = thirdTokenKeyword;
        this.thirdMsgKeyword = thirdMsgKeyword;
    }
}
```

4. **厂商通道通知类型-辅助弹窗**

厂商通道，除Google通道外，只能通过辅助弹窗来接收推送数据，详情参考辅助弹窗接入文档，并注意：

- 在阿里云这边进行厂商通道推送时（使用移动推送控制台或者OpenAPI进行推送时），服务端请必须参考辅助弹窗文档进行服务端配置，服务端参数不设置，不会给厂商通道进行推送。
- 使用辅助弹窗接收数据，请务必先执行如下操作，否则会解析数据失败。初始化厂商通道解码器，必须在Application中，并且在厂商通道初始化之前调用，Google通道不需要：

```
//此处以华为为例
ThirdPushManager.registerImpl(new HuaweiMsgParseImpl());

//MsgParseImpl 目前有 HuaweiMsgParseImpl、XiaoMiMsgParseImpl、OppoMsgParseImpl、VivoMsg
ParseImpl、MeizuMsgParseImpl
```

5. **厂商通道消息类型**

   接收到消息类型需要统一回调处理，可以使用 `ThirdPushManager.onPushMsg` 把消息内容传到阿里云推送SDK。

```
/**
* 以华为为例
*
* @param  context 上下文对象，建议传ApplicationContext
* @param  thirdTokenKeyword 厂商消息的标识
* @param  token 厂商的消息内容
*/
ThirdPushManager.onPushMsg(context, ThirdPushReportKeyword.HUAWEI.thirdMsgKeyword, msgC
ontent);
```

1.4.9. 辅助弹窗接入

本章节介绍接入厂商通道后如何使用辅助弹窗。

概述

接入推送功能的App进程在后台被清理后会收不到推送通知；使用辅助弹窗后，可以借助设备的系统通道，保证在App后台被清理的情况下，仍能收到推送通知。辅助弹窗的通知展示效果和普通通知相同。

使用辅助弹窗后，通知将由设备系统托管弹出，单击通知栏将转跳到指定的Activity。

> 注意
>
> 辅助弹窗功能的使用依赖于厂商通道，请确保已集成最新的厂商通道SDK。

客户端配置

集成AndroidPopupActivity

- 服务端指定辅助弹窗通道推送时，一定要指定通知单击后要打开的Activity，该Activity需继承自抽象类 `AndroidPopupActivity` ，否则无法获取到通知的相关信息，并且会影响通知到达率的统计；

> 说明
>
> `MiPushSystemNotificationActivity` 已废弃，小米、华为、OPPO等厂商通道弹窗统计集成 `AndroidPopupActivity` 。

- AndroidPopupActivity中提供抽象方法 `onSysNoticeOpened()` ，实现该方法后可获取到辅助弹窗通知的标题、内容和额外参数，在通知单击时触发，原本的通知回调 `onNotification()` 和 `onNotificationOpened()` 不适用于辅助弹窗；
- 指定打开的托管弹窗Activity在AndroidManifest.xml中注册时需要声明属性： `android:exported=true` 。

接入如下所示：

```
import com.alibaba.sdk.android.push.AndroidPopupActivity;
public class PopupPushActivity extends AndroidPopupActivity {
    static final String TAG = "PopupPushActivity";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
    /**
     * 实现通知打开回调方法，获取通知相关信息
     * @param title     标题
     * @param summary   内容
     * @param extMap    额外参数
     */
    @Override
    protected void onSysNoticeOpened(String title, String summary, Map<String, String> extM
ap) {
        Log.d("OnMiPushSysNoticeOpened, title: " + title + ", content: " + summary + ", ext
Map: " + extMap);
    }
}
```

任意Activity

自辅助通道SDK 3.2.3版本后，新增支持任意Activity都可以拿到厂商通知单击时的数据，实现过程如下。创建 `PopupNotifyClick` 实例，同时传入 `PopupNotifyClickListener` 接口，在 `PopupNotifyClick` 实例的 `onCreate` 方法传入 `context` 和单击通知指定启动 `Activity` 的 `Intent` ，解析完成后，会在 `PopupNotifyClickListener` 接口的 `onSysNoticeOpened` 回调标题、内容和额外参数。如示例：

```
public class XXXActivity extends Activity {

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        (new PopupNotifyClick(new PopupNotifyClickListener() {
            public void onSysNoticeOpened(String title, String summary, Map<String, String>
extMap) {
                Log.d("xxx", "Receive notification, title: " + title + ", content: " + cont
ent + ", extraMap: " + extraMap);
            }
        })).onCreate(this, this.getIntent());
    }
}
```

辅助弹窗Activity启动模式说明

由于各个厂商系统的特性，设置不同启动模式，单击通知时的效果也不完全相同，有些厂商设备上部分启动模式也不符合原有启动效果。

首先说明，应用结束进程时，单击第一条离线通知，是必然会正常创建启动辅助弹窗Activity，所以无论使用AndroidPopupActivity还是自行使用PopupNotifyClick，均可以正常获取到通知数据。当单击一条离线通知启动了应用后，再次单击第二条离线通知，各启动模式和各厂商设备表现不完全相同，下面按照启动模式进行关键信息说说明：

- Standard：如果已经启动的辅助弹窗Activity不finish，单击第二条通知，不会再次创建辅助弹窗Activity，部分厂商也不会触发已有辅助弹窗Activity的onNewInten方法，所以也无法使用PopupNotifyClick解析onNewInten方法的intent。如果辅助弹窗Activity finish了，单击第二条通知，均会重新创建辅助弹窗Activity，可以正常拿到通知数据。所以使用Standard启动模式时，建议及时将辅助弹窗Activity finish。
- SingleTop：如果已经启动的辅助弹窗Activity不finish，单击第二条通知时刚好界面展示的就是辅助弹窗Activity，此时会触发onNewInten方法，可以使用PopupNotifyClick解析onNewInten方法的intent。其他场景，与Standard启动模式效果一致。所以使用SingleTop启动模式时，也建议及时将辅助弹窗Activity finish。
- SingleTask：各个场景，均符合SingleTask启动模式原有效果，即：辅助弹窗Activity 未finish，单击第二条通知会显示辅助弹窗Activity并触发onNewInten方法；辅助弹窗Activity finish，单击第二条通知会重新创建辅助弹窗Activity。所以使用SingleTask启动模式时，在原有辅助弹窗Activity，增加使用PopupNotifyClick解析onNewInten方法的intent即可。
- SingleInstance：各个场景，也是均符合SingleInstance启动模式原有效果。所以使用SingleInstance启动模式时，在原有辅助弹窗Activity，增加使用PopupNotifyClick解析onNewInten方法的intent即可。

服务端配置

服务端可以使用OpenAPI，也可以使用阿里云控制台。

> **注意**
>
> 使用移动推送进行厂商通道推送时，服务端必须进行以下配置，服务端参数不设置，不会给厂商通道进行推送。

1、OpenAPI推送配置

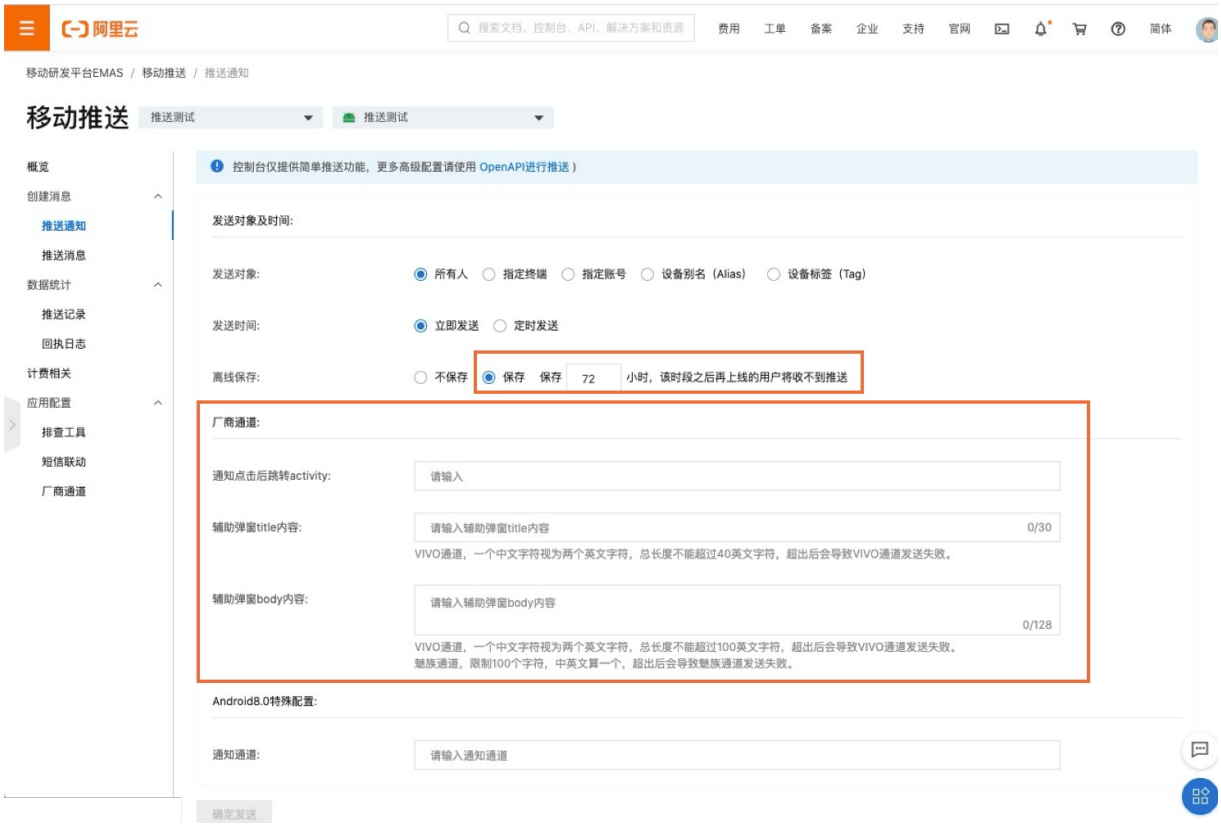
- OpenAPI 2.0的推送高级接口提供了 `AndroidPopupActivity` 、 `AndroidPopupTitle` 和 `AndroidPopupBody` 三个参数，分别用于设置辅助弹窗通知打开时跳转的Activity、通知标题以及通知内容（注意： `StoreOffline` 参数也需要设置为true）；
- 辅助弹窗仅在当前厂商通道设备的应用后台进程被清理时生效，对非接入厂商通道设备和在线的设备不生效。
- 当辅助弹窗生效时，推送接口的Title、Body、AndroidActivity以及额外参数设置中的功能性设置（如声音、震动等）都不起作用；

服务端配置示例：

```
PushRequest pushRequest = new PushRequest();
// 其余设置省略
// 通知
pushRequest.setPushType("NOTICE");
// 标题
pushRequest.setTitle(dateFormat.format(new Date()));
// 内容
pushRequest.setBody("PushRequest body");
// 额外参数
pushRequest.setAndroidExtParameters("{\"k1\":\"android\",\"k2\":\"v2\"}");
// 设置辅助弹窗打开Activity，填写Activity类名，需包名+类名
pushRequest.setAndroidPopupActivity("*****");
// 设置辅助弹窗通知标题
pushRequest.setAndroidPopupTitle("*****");
// 设置辅助弹窗通知内容
pushRequest.setAndroidPopupBody("*****");
// 72小时后消息失效，不会再发送
String expireTime = ParameterHelper.getISO8601Time(new Date(System.currentTimeMillis() + 72
* 3600 * 1000));
pushRequest.setExpireTime(expireTime);
// 离线消息是否保存,若保存，在推送时候，用户即使不在线，下一次上线则会收到
pushRequest.setStoreOffline(true);
//推送消息类型时，设置true，设备离线时会自动把消息转成辅助通道的通知
pushRequest.setAndroidRemind(true);
```

2、阿里云控制台推送配置

登录移动研发平台EMAS，选择**移动推送**后选择对应的应用，在左侧导航栏选择**创建消息>推送通知**，在**高级设置（选填）**中，设置厂商通道必要参数，如下图所示。



1.4.10. 辅助通道集成场景解析

普通推送结合辅助弹窗推送的场景，帮助您快速理解辅助弹窗功能的配置。

客户端配置

客户端有Main、Second两个Activity，MainActivity为App打开主页面，SecondActivity extends AndroidPopupActivity；

- 普通通知回调配置

```
public class MyMessageReceiver extends MessageReceiver {
  /**
  * 推送通知的回调方法
  * @param context
  * @param title
  * @param summary
  * @param extraMap
  */
  @Override
  public void onNotification(Context context, String title, String summary, Map<String, S
tring> extraMap) {
    Log.d(TAG, "Receive notification, title: " + title + ", content: " + summary + ", ext
raMap: " + extraMap);
  }
}
```

- MainActivity定义：

```
package com.alibaba.push.testdemo;
public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    Log.d(TAG, "Main");
  }
}
```

- SecondActivity定义：

```
package com.alibaba.push.testdemo;
import com.alibaba.sdk.android.push.AndroidPopupActivity;
public class SecondActivity extends AndroidPopupActivity {
  /**
   * 辅助弹窗指定打开Activity回调
   * @param title     标题
   * @param content   内容
   * @param extraMap  额外参数
   */
  @Override
  protected void onSysNoticeOpened(String title, String content, Map<String, String> extr
aMap) {
      Log.d(TAG, "Receive XiaoMi notification, title: " + title + ", content: " + content
+ ", extraMap: " + extraMap);
  }
}
```

场景：普通推送+辅助弹窗

服务端配置

服务端配置如下：

```
PushRequest pushRequest = new PushRequest();
// 其余设置省略
// 通知
pushRequest.setPushType("NOTICE");
// 标题
pushRequest.setTitle("hello");
// 内容
pushRequest.setBody("PushRequest body");
// 点击通知后动作 "APPLICATION" : 打开应用 "ACTIVITY" : 打开AndroidActivity "URL" : 打开URL "NO
NE" : 无跳转
pushRequest.setAndroidOpenType("APPLICATION");
// 设定通知打开的activity，仅当AndroidOpenType="Activity"有效
pushRequest.setAndroidActivity("com.alibaba.push.testdemo.MainActivity");
// 设置辅助弹窗打开Activity
pushRequest.setAndroidPopupActivity("com.alibaba.push.testdemo.SecondActivity");
// 设置辅助弹窗通知标题
pushRequest.setAndroidPopupTitle("hello2");
// 设置辅助弹窗通知内容
pushRequest.setAndroidPopupBody("PushRequest body2");
// 设定android类型设备通知的扩展属性
pushRequest.setAndroidExtParameters("{\"k1\":\"android\",\"k2\":\"v2\"}");
```

推送结果

非厂商通道设备和在线设备：

- 收到普通推送通道弹出的通知，点击后打开App，进入首页MainActivity，如果设备在前台，保持当前界面不变；
- onNotification()回调输出Receive notification, title: hello, content: PushRequest body, extraMap: {k1=android, k2=v2}；

清理进程后的小米、华为等厂商通道设备：

- 辅助弹窗通道弹出通知，点击通知后跳转到SecondActivity；
- onSysNoticeOpened()回调输出Receive XiaoMi notification, title: hello, content: PushRequest body, extraMap: {k1=android, k2=v2}；

场景总结

服务端参数

- Title、Body、AndroidExtparameters全场景生效。
- AndroidOpenType、AndroidActivity作用于在线设备和非小米、华为等厂商通道设备。
- AndroidPopupTitle、AndroidPopupBody、AndroidPopupActivity仅作用于辅助通道。

客户端数据接收

- 在线设备和非小米、华为等厂商通道设备，数据通过自定义接收器MessageReceiver接收数据。
- 辅助通道，通过辅助弹窗AndroidPopupActivity接收数据。

> **说明**
>
> 辅助通道送达的通知，通知中标题和内容是服务端设置的 `AndroidPopupTitle` 、 `AndroidPopupBody` ，点击通知唤起辅助弹窗 `AndroidPopupActivity` ，在 `onSysNoticeOpened` 中回调的是服务端设置的 `Title` 、 `Body` 、 `AndroidExtParameters` 。

1.5. 错误处理

本文介绍移动推送的错误码列表供您参考。

调用 `CloudPushService` 的相关接口时，如果发生错误，可以在 `CommonCallback` 的 `onFailed()` 回调中可以获取到 `errorCode` 和 `errorMessage` 。

错误码 （V3.7.6版本及以上）

- **推送接口层错误码**

错误码	错误描述	备注
PUSH_00000	success	-

错误码	错误描述	备注
PUSH_10101	参数缺失	请检查请求参数是否正确。
PUSH_10102	参数无效	请检查请求参数是否正确。
PUSH_10103	服务端签名与客户端不匹配	请检查推送配置是否正确。
PUSH_10104	Tag相关错误	请根据具体错误信息排查，如果不能解决，请联系阿里云技术支持。
PUSH_10105	Alias相关错误	请根据具体错误信息排查，如果不能解决，请联系阿里云技术支持。
PUSH_10106	服务端内部错误	请根据具体错误信息联系阿里云技术支持。
PUSH_10107	网络IO错误	1. 请检查网络是否可用 2. 请根据具体错误信息排查，如果不能解决，请联系阿里云技术支持。
PUSH_10108	返回结果解析错误	请保留具体错误信息，联系阿里云技术支持排查。
PUSH_10109	网络连接失败,请检查网络配置	请检查网络是否可用。
PUSH_10114	内部错误	请保留具体错误信息，联系阿里云技术支持排查。
PUSH_10115	通道注册状态异常	请保留具体错误信息，联系阿里云技术支持排查。
PUSH_10118	其它接口错误	请根据具体错误信息联系阿里云技术支持。
PUSH_10119	非主进程不用初始化	在非主进程执行初始化时触发，可以忽略。

错误码	错误描述	备注
PUSH_10120	推送注册超时	请保留具体错误信息，联系阿里云技术支持排查。
PUSH_10121	网络请求失败，请检查网络是否可用	1. 请检查网络是否可用。 2. 请根据具体错误信息排查，如果不能解决，请联系阿里云技术支持。
PUSH_20101	参数输入非法	请检查请求的输入参数是否正确。
PUSH_20102	静默连接进程名设置错误,进程名不能为空且必须与manifest文件配置相符。系统自动设置为manifest所配置进程名	开启debug会检查此错误，目前进程名不支持修改，请不要修改组件进程配置。
PUSH_20103	appversion参数错误,请检查您的版本号,版本号不能为null或长度不能超过32位	开启debug会检查此错误，请检查应用版本号是否过长。
PUSH_20105	ChannelService未设置辅助进程	开启debug会检查此错误，如果不是特殊场景，请检查是否修改了推送组件的进程配置。
PUSH_20106	核心组件未配置	开启debug会检查此错误，请检查是否删除了推送组件的声明。
PUSH_20107	连续crash，推送服务关闭	1. 应用初始化推送后崩溃，会在下次启动关闭推送服务。请检查应用的崩溃记录。 2. 开发测试场景下，人为触发的，请清除应用数据恢复。 3. 线上场景会尝试自动恢复，如果仍然崩溃，需要升级应用版本才会恢复。
PUSH_20108	未初始化，请先调用PushServiceFactory的init方法	请确认是否正常初始化。

错误码	错误描述	备注
PUSH_20109	废弃接口	请查看CHANP文档，使用合适的API。
PUSH_20110	已经调用注册，重复调用无效	1. register方法如果失败了，会自动重试，一般情况下不需要重复调用。 2. 如果希望内部重试失败的情况，由外部重新调用register，请至少在上一次register失败回调两次（确认内部重试还是失败）的情况下，先调用PushControlService的reset方法，然后再调用下一次register方法。
PUSH_ACCS_123	accs错误信息	格式ACCS_123, 123为accs错误码，请结合accs错误码排查。
PUSH_xxx	agoo错误信息	格式AGOO_xxx, xxx为agoo错误码，请结合agoo错误码排查。

- **推送协议层（AGOO）错误码**

错误码	错误描述	备注
EAGOO_SDK_success	success	-
EAGOO_SDK_remove_alias_fail_no_token	移除别名失败，本地没有别名记录	1. 请检查输入的别名是否正确。 2. 低版本推送有概率出现，添加别名后，应用的数据被清除，导致SDK内部存储的别名信息丢失，无法移除。
EAGOO_SDK_remove_alias_fail_no_alias	移除别名失败，本地没有别名记录	请检查输入的别名是否正确2. 低版本推送有概率出现，添加别名后，应用的数据被清除，导致SDK内部存储的别名信息丢失，无法移除
EAGOO_SDK_invalid_arg	请求参数错误	请检查输入参数。

错误码	错误描述	备注
EAGOO_SDK_accs_disabled	accs检查不通过	1. 请检查初始化是否成功。 2. 请检查配置是否正确。 3. 请检查请求是否是在主进程。
EAGOO_SDK_agoo_not_bind	请先注册初始化agoo	请检查初始化是否成功。
EAGOO_ACCS_123	accs 错误信息	1. 推送底层通道错误，需要根据错误码排查。 2. 格式EAGOO_ACCS_123, 123为推送底层通道错误码，请结合推送底层通道错误码排查。
EAGOO_SERVER_XXX	服务错误信息	1. 推送服务报错，请联系技术支持排查。 2. 格式EAGOO_SERVER_XXX, XXX为推送服务错误码，请联系阿里云技术支持排查。

- **推送底层通道（ACCS）错误码**

错误码	错误描述	备注
200	成功	-
300	通道未建立	请先初始化bindApp，再调用其它API。
-1	静默连接中断，无法发送消息	内部会重试，如果一直失败，需要排查下静默通道是否正常。
-2	参数错误,发送的msg为null	请检查发起请求的参数是否正确。
-3	服务返回数据异常	请关注错误信息中的服务返回数据，并联系阿里云技术支持同学确认原因。
-4	单次发送数据过大	请减少一次发送的数据量，封装之后总的数据量要小于16KB。

错误码	错误描述	备注
-5	发送服务地址为null	请检查下初始化配置是否正确
-6	静默通道长连接认证参数错误	请检查初始化参数配置是否正确
-7	静默通道长连接认证异常	请查看错误信息，确认具体异常信息。
-8	发送数据异常	请查看错误信息，确认具体异常信息。
-9	发送消息超时	需要结合具体是查看为什么超时。
-10	静默通道长连接断连	断连需要查看之前的日志。
-11	应用内长连接断开	一般为长连接建连失败造成，需要看日志分析。
-12	静默通道长连接ping超时	-
-13	无网络	请检查网络连接。
-14	appKey不存在	请检查初始化配置是否正确。
-15	appSecret不存在	请检查初始化配置是否正确。
70008	长连接发送队列已满	请确认是否有高并发发送消息，如果有，请限制发送频次。
70020	低级别限流	请和部署同学确认限流策略。
70021	高级别限流,不发送	请和部署同学确认限流策略。
70023	防刷解封后触发的限流，不发送	请和部署同学确认限流策略。

错误码	错误描述	备注
102	设备无效	如果是测试时发现的，请清除应用数据重新尝试。
302	设备无效	如果是测试时发现的，请清除应用数据重新尝试。
303	appkey配置错误	请检查AppKey配置是否正确。
304	包名错误	请检查appKey和应用包名是否匹配。
-20	服务返回错误	请关注下错误信息中的服务返回的错误码，并联系阿里云技术支持同学确认原因。
-22	底层sdk连接关闭	请关注下错误信息中的底层sdk返回的错误信息，并联系阿里云技术支持同学确认原因。
-23	发送数据返回错误	请关注下错误信息中的底层sdk返回的错误信息，并联系阿里云技术支持同学确认原因。
-25	不应该发生的错误	请关注下错误信息，检查初始化是否存在错误。
-26	建连参数错误	请检查初始化配置是否正确。
-27	建连超时	1. 请查看具体错误信息排查。 2. 请检查网络是否正常。
-28	建连失败	1. 请查看具体错误信息排查。 2. 请检查网络是否正常。
-29	连接地址不存在	1. 当前网络下无法解析长链接地址2. 请检查网络是否正常。
-30	建连异常	请查看具体错误信息排查。

错误码	错误描述	备注
-10000	底层网络库信息	小于-10000时，加上10000是底层网络库对应的错误码，请接口底层网络库错误码信息排查。

常见错误码 （V2.3.5版本及以上）

错误名称	错误码（Error Code）	错误描述和解决办法（Error Message）
MISSING_PARAM	10101	参数缺失，根据详细错误信息补充参数。
INVALID_PARAM	10102	参数无效，查看详细错误信息。
SIGN_NOT_MATCH	10103	服务端与客户端签名不匹配，检查AppKey，AppSecret。
TAG_ERROR	10104	Tag请求相关错误，查看详细错误信息。
ALIAS_ERROR	10105	Alias相关错误，查看详细错误信息。
INTERNAL_ERROR	10106	服务端内部错误。
IO_ERROR	10107	网络I/O错误。
RESPONSE_PARSE_ERROR	10108	返回结果解析错误。
CONNECTION_FAIL	10109	网络连接失败,请检查网络配置。
SYSTEM_ERROR	10110	系统错误。
UNKNOWN_ERROR	10111	未知错误。
NO_NETWORK	10201	网络不可用

错误名称	错误码（Error Code）	错误描述和解决办法（Error Message）
APPKEY_NULL	10202	无效AppKey。
APPSECRET_NULL	10203	无效AppSecret。
APPRECEIVER_NULL	10204	回调函数为空。
REG_TIME_OUT	10205	请求超时，请查看tag为awcn的error级别日志。
CONN_INVALID	10206	当前连接异常。
NO_CONNECTION	10207	无网络连接,请查看tag为awcn的error级别日志。建议检查一下相应so包是否添加，可参考出现“1105，网络不稳定或连接异常错误”怎么解决?
TAIR_ERROR	10209	服务器错误。
INVALID_DEVICEID	10210	无效deviceid。
INVALID_PACKAGE	10211	包名与配置不符。
ACCS_CHANNEL_INIT_FAIL	10212	静默连接进程（默认为channel进程）未初始化，参考Android推送失败排查步骤中“已经接入成功，突然出现异常”下第二小节。
API_INVALID_INPUT	20101	参数非法，详见具体错误信息。
APP_VERSION_INVALID	20103	appversion参数错误,请检查您的版本号,版本号不能为null且长度不能超过32位。
CHANNEL_PROCESS_NULL	20105	ChannelService未设置辅助进程。

错误名称	错误码（Error Code）	错误描述和解决办法（Error Message）
REQUIRED_COMPONENT_NOT_EXISTS	20106	核心组件未配置，详见具体错误信息。
CONTINOUS_CRASH	20107	连续crash，推送服务关闭。

常见错误码（V2.3.4版本及以下）

错误名称	错误码（Error Code）
NO_NETWORK	1101
REG_FAIL	1056
INVAILD_APPKEY	1052
INVAILD_PACKAGENAME	1053
INVAILD_APPSECRET	1054
NETWORK_UNSTABLE	1105
INVAILD_SERVER_RETRUN	1115
SYSTEM_UNKNOWN_ERROR	1108

2.iOS SDK手册

2.1. iOS SDK配置

本章节介绍移动推送iOS SDK的集成操作。

前提条件

- 使用手动添加依赖方式需提前下载SDK包，请参见EMAS快速入门>下载SDK。
- 使用统一接入添加依赖需提前下载配置文件，请参见EMAS快速入门>下载配置文件。
- 已配置推送证书，请参见配置推送证书。
- 已阅读移动推送产品的使用限制。

样例代码

移动推送服务iOS SDK接入工程样例参见移动推送iOS Demo。

集成步骤

1. **添加依赖**

   i. 方式一：手动添加

   a. 解压下载好的SDK包，在Xcode中，把SDK包目录中的framework拖入对应Target下即可，在弹出框勾选**Copy items if needed**。

```
CloudPushSDK.framework
AlicloudUtils.framework
UTDID.framework
UTMini.framework
AlicloudSender.framework
EMASRest.framework
```

   b. 在工程项目中（Build Phases -> Link Binary With Libraries）添加以下库文件。

```
libz.tbd
libresolv.tbd
CoreTelephony.framework
SystemConfiguration.framework
libsqlite3.tbd
```

ii. 方式二：Pod集成

a. 在Podfile中添加source，指定Master仓库和阿里云仓库。

```
source 'https://github.com/CocoaPods/Specs.git'
source 'https://github.com/aliyun/aliyun-specs.git'
```

b. 在终端执行 `pod repo add` 命令，拉取阿里云Pod仓库到本地 。

```
pod repo add AliyunRepo https://github.com/aliyun/aliyun-specs.git
```

或手动拉取Pod仓库工程到CocoaPods仓库（默认为 `~/.cocoapods/repos` ）。

```
git clone https://github.com/aliyun/aliyun-specs.git ~/.cocoapods/repos/
```

> **注意**
>
> - 可执行 `pod repo list` 查看本地 Pod 仓库信息。
> - pod repo list或git clone失败一般为GitHub账号publicKey配置问题，请仔细核对错误信息，Pod命令使用参考官方文档。

c. 在您的工程中添加以下系统依赖库。

```
pod 'AlicloudPush', '~> 1.9.9'
```

> **说明**
>
> ~>为模糊指定版本号方式，~> 1.9.9表明引用版本位于 `1.9.9 <= version < 2.0` 之间的最新版本SDK，用户可参考Podfile Syntax Reference，根据项目需要指定SDK版本。

2. **引入头文件**

```
#import <CloudPushSDK/CloudPushSDK.h>
```

3. **Objc配置**

iOS端集成SDK时需要做 `-ObjC` 配置，即应用的 TARGETS -> Build Settings -> Linking -> Other Linker Flags，需添加上 `-ObjC` 这个属性，否则推送服务无法正常使用 。

`Other Linker Flags` 中设定链接器参数 `-ObjC` ，加载二进制文件时，会将 Objective-C 类和 Category 一并载入，若工程依赖多个三方库，将所有 Category 一并加载后可能发生冲突，可以使用 `-force_load` 单独载入指定二进制文件，配置如下：

```
-force_load<framework_path>/CloudPushSDK.framework/CloudPushSDK
```

> **注意**
>
> `-force_load` 指定的是二进制文件路径，而不是 Framework 目录。若指定为 Framework 目录，-force_load ...path/xxx.framework，Xcode 会报出错误 ld: can't map file, errno=22 file '...path/xxx.framework' for architecture armv7。

4. **SDK初始化配置**

在Xcode中把下载好的配置文件（AliyunEmasServices-Info.plist）拖入对应App Target下即可，在弹出框勾选**Copy items if needed**。

> **说明**
>
> - 配置文件aliyunEmasServices-Info.plist中包含SDK初始化所需的配置信息，用户只需要调用autoInit接口进行初始化，无需手动输入配置信息。
> - 下载配置文件的具体操作参考**移动研发平台 EMAS** > 快速入门 > **下载配置文件**。

请参照以下代码完成SDK的初始化。

> **说明**
>
> - 您可在下载好的配置文件中获取Appkey、AppSecret，或参考EMAS快速入门。
> - 百川云推送迁移过来的用户，不要使用百川平台获取到的Appkey、AppSecret，请使用EMAS平台获取的AppKey及AppSecret，否则会发生鉴权错误。

```
- (void)initCloudPush {
    // SDK初始化
    [CloudPushSDK asyncInit:@"*****" appSecret:@"*****" callback:^(CloudPushCallbackRes
ult *res) {
        if (res.success) {
            NSLog(@"Push SDK init success, deviceId: %@.", [CloudPushSDK getDeviceId]);
        } else {
            NSLog(@"Push SDK init failed, error: %@", res.error);
        }
    }];
}
```

向苹果APNs注册获取deviceToken，并上报到阿里云推送服务器：

```
/**
 *    注册苹果推送，获取deviceToken用于推送
 *
 *    @param     application
 */
- (void)registerAPNS:(UIApplication *)application {
    if ([[[UIDevice currentDevice] systemVersion] floatValue] >= 8.0) {
        // iOS 8 Notifications
        [application registerUserNotificationSettings:
         [UIUserNotificationSettings settingsForTypes:
          (UIUserNotificationTypeSound | UIUserNotificationTypeAlert | UIUserNotificati
onTypeBadge)
                                           categories:nil]];
        [application registerForRemoteNotifications];
    }
    else {
        // iOS < 8 Notifications
        [[UIApplication sharedApplication] registerForRemoteNotificationTypes:
         (UIRemoteNotificationTypeAlert | UIRemoteNotificationTypeBadge | UIRemoteNotif
icationTypeSound)];
    }
}
/*
 *  苹果推送注册成功回调，将苹果返回的deviceToken上传到CloudPush服务器
 */
- (void)application:(UIApplication *)application didRegisterForRemoteNotificationsWithD
eviceToken:(NSData *)deviceToken {
    [CloudPushSDK registerDevice:deviceToken withCallback:^(CloudPushCallbackResult *re
s) {
        if (res.success) {
            NSLog(@"Register deviceToken success.");
        } else {
            NSLog(@"Register deviceToken failed, error: %@", res.error);
        }
    }];
}
/*
 *  苹果推送注册失败回调
 */
- (void)application:(UIApplication *)application didFailToRegisterForRemoteNotification
sWithError:(NSError *)error {
    NSLog(@"didFailToRegisterForRemoteNotificationsWithError %@", error);
}
```

推送消息到来监听：

```
/**
 *    注册推送消息到来监听
 */
- (void)registerMessageReceive {
    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(onMessageReceived:)
                                                 name:@"CCPDidReceiveMessageNotificatio
n"
                                               object:nil];
}
/**
 *    处理到来推送消息
 *
 *    @param     notification
 */
- (void)onMessageReceived:(NSNotification *)notification {
    CCPSysMessage *message = [notification object];
    NSString *title = [[NSString alloc] initWithData:message.title encoding:NSUTF8Strin
gEncoding];
    NSString *body = [[NSString alloc] initWithData:message.body encoding:NSUTF8StringE
ncoding];
    NSLog(@"Receive message title: %@, content: %@.", title, body);
}
```

通知打开监听：

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    // 点击通知将App从关闭状态启动时，将通知打开回执上报
    // [CloudPushSDK handleLaunching:launchOptions];(Deprecated from v1.8.1)
    [CloudPushSDK sendNotificationAck:launchOptions];
    return YES;
}
/*
 *  App处于启动状态时，通知打开回调
 */
- (void)application:(UIApplication*)application didReceiveRemoteNotification:(NSDictionary*)userInfo {
    NSLog(@"Receive one notification.");
    // 取得APNS通知内容
    NSDictionary *aps = [userInfo valueForKey:@"aps"];
    // 内容
    NSString *content = [aps valueForKey:@"alert"];
    // badge数量
    NSInteger badge = [[aps valueForKey:@"badge"] integerValue];
    // 播放声音
    NSString *sound = [aps valueForKey:@"sound"];
    // 取得Extras字段内容
    NSString *Extras = [userInfo valueForKey:@"Extras"]; //服务端中Extras字段，key是自己定义的
    NSLog(@"content = [%@], badge = [%ld], sound = [%@], Extras = [%@]", content, (long)badge, sound, Extras);
    // iOS badge 清0
    application.applicationIconBadgeNumber = 0;
    // 通知打开回执上报
    // [CloudPushSDK handleReceiveRemoteNotification:userInfo];(Deprecated from v1.8.1)
    [CloudPushSDK sendNotificationAck:userInfo];
}
```

注意

- iOS 10系统的设备需注意，基于工信部的要求，国行手机首次安装App时，会弹出一个询问用户“是否允许应用访问数据”的弹框。在用户点击 `允许` 前，或点击 `不允许` ，App的网络环境是不通的，会导致推送SDK的初始化失败，推送服务不能正常使用。推送各接口内部是有 `重试机制` 的，但是，建议用户在业务层，也要捕获处理 `SDK接口错误回调` ，确保正确获知SDK接口调用状态。
- 移动推送iOS SDK已经完成ATS适配，请求都以HTTPS发出，无需在Info.plist中进行ATS配置。
- 若SDK集成过程中出现 `UTDID冲突` ，请参考 阿里云-移动云产品SDK UTDID冲突解决方案。

5. **初始化成功验证**

   根据SDK初始化回调结果查看是否成功。

```
if (res.success) {
            NSLog(@"Push SDK init success, deviceId: %@.", [CloudPushSDK getDeviceId]);
        } else {
            NSLog(@"Push SDK init failed, error: %@", res.error);
        }
```

常见集成问题

1. iOS SDK集成出错排查步骤
2. iOS端推送失败排查步骤
3. iOS端获取deviceToken的问题

2.2. iOS SDK API

2.2.1. 基本设置相关接口

> 说明
>
> 以下接口中提供回调的接口均为异步执行。

SDK自动初始化

无需配置appKey和appSecret，结合AliyunEmasServices-Info.plist配置文件使用。

接口定义

```
+ (void)autoInit:(CallbackHandler)callback;
```

参数说明

参数	类型	是否必填	说明
callback	Block	否	注册状态回调

SDK手动初始化

输入appKey和appSecret初始化推送SDK。

接口定义

```
+ (void)asyncInit:(NSString *)appKey
        appSecret:(NSString *)appSecret
         callback:(CallbackHandler)callback;
```

参数说明

参数	类型	是否必填	说明
appKey	NSString	是	appKey
appSecret	NSString	是	appSecret
callback	Block	否	注册状态回调

打开调试日志

打开推送SDK日志。

> **注意**
>
> 测试时可打开调试查看日志，应用上线后建议关闭。

接口定义

```
+(void)turnOnDebug;
```

获取SDK版本号

版本号也可以在CloudPushSDK.h中查看。

接口定义

```
+ (NSString *)getVersion;
```

获取推送消息通道状态

查询推送应用内通道状态。

接口定义

```
+ (BOOL)isChannelOpened;
```

关闭推送消息通道

关闭推送消息通道（长链），需要保证在SDK初始化之前调用。

接口定义

```
+ (void)closeCCPChannel;
```

获取设备deviceId

DeviceID为阿里云移动推送过程中对设备的唯一标识（并不是设备UUID/UDID），SDK初始化成功后，调用如下接口获取deviceId。

接口定义

```
+ (NSString *)getDeviceId;
```

2.2.2. 账号（account）相关接口

> **说明**
>
> 以下接口中提供回调的接口均为异步执行。

绑定账号

将应用内账号和推送通道相关联，可以实现按账号的定点消息推送。

> **注意**
>
> - 设备只能绑定一个账号，同一个账号可以绑定到多个设备。
> - 同一设备更换绑定账号时无需进行解绑，重新调用绑定账号接口即可生效。
> - 若业务场景需要先解绑后再绑定，在解绑账号成功回调中进行解绑操作，以此保证执行的顺序性。
> - 账号名长度最大支持64字节。

接口定义

```
+ (void)bindAccount:(NSString *)account
       withCallback:(CallbackHandler)callback;
```

参数说明

参数	类型	是否必须	说明
account	NSString	是	待绑定的账号名。
callback	Block	否	回调

解绑账号

将应用内账号和推送通道取消关联。

接口定义

```
+ (void)unbindAccount:(CallbackHandler)callback;
```

参数说明

参数	类型	是否必须	说明
callback	Block	否	回调

2.2.3. 标签（tag）相关接口

> 说明
>
> 以下接口中提供回调的接口均为异步执行。

绑定标签

绑定标签到指定目标。

> 注意
>
> - 支持向设备、账号和别名绑定标签，绑定类型有参数target指定。
> - App最多支持定义1万个标签，单个标签支持的最大长度为129字符。
> - 绑定标签在10分钟内生效。

接口定义

```
+ (void)bindTag:(int)target
        withTags:(NSArray *)tags
       withAlias:(NSString *)alias
    withCallback:(CallbackHandler)callback;
```

参数说明

参数	类型	是否必须	说明
target	int	是	目标类型可选值： • 1：本设备 • 2：本设备绑定的账号 • 3：别名
tags	NSArray	是	标签（数组输入）
alisa	NSString	否	别名，仅当target=3时生效
callback	Block	否	回调

参数	类型	是否必须	说明

解绑标签

解绑指定目标的标签。

> 注意
>
> - 支持解绑设备、账号和别名的标签，解绑类型有参数target指定。
> - 解绑标签只是解除设备和标签的绑定关系，不等同于删除标签，即该App下标签依然存在，系统当前不支持删除标签。
> - 解绑标签在10分钟内生效。

接口定义

```
+ (void)unbindTag:(int)target
         withTags:(NSArray *)tags
        withAlias:(NSString *)alias
     withCallback:(CallbackHandler)callback;
```

参数说明

参数	类型	是否必须	说明
target	int	是	目标类型可选值： • 1：本设备 • 2：本设备绑定的账号 • 3：别名
tags	NSArray	是	标签（数组输入）
alias	NSString	否	别名，仅当target=3时生效
callback	Block	否	回调

查询标签

查询目标绑定的标签，当前仅支持查询设备标签。

> **注意**
>
> 查询结果可从回调的data中获取。

接口定义

```
+ (void)listTags:(int)target
    withCallback:(CallbackHandler)callback;
```

参数说明

参数	类型	是否必须	说明
target	int	是	目标类型可选值： • 1：本设备
callback	Block	否	回调

2.2.4. 别名（alias）相关接口

> **说明**
>
> 以下接口中提供回调的接口均为异步执行。

添加别名

为设备添加别名。

> **注意**
>
> • 单个设备最多添加128个别名，同一个别名最多可被添加到128个设备。
> • 别名支持的最大长度为128字节。

接口定义

```
+ (void)addAlias:(NSString *)alias
    withCallback:(CallbackHandler)callback;
```

参数说明

参数	类型	是否必填	说明
alias	NSString	是	别名

参数	类型	是否必填	说明
callback	Block	否	回调

删除别名

删除设备别名。

> 说明
>
> 支持删除指定别名和删除全部别名。

接口定义

```
+ (void)removeAlias:(NSString *)alias
       withCallback:(CallbackHandler)callback;
```

参数说明

参数	类型	是否必填	说明
alias	NSString	是	alias = null or alias.length = 0 时，删除设备全部别名。
callback	Block	否	回调

查询别名

查询设备别名。

> 注意
>
> 查询结果可从回调的data中获取。

接口定义

```
+ (void)listAliases:(CallbackHandler)callback;
```

参数说明

参数	类型	是否必填	说明
callback	Block	否	回调

2.2.5. 设备DeviceToken相关接口

> 说明
>
> 以下接口中提供回调的接口均为异步执行。

获取设备deviceToken

返回获取APNs返回的deviceToken，调用registerDevice()接口后可获取。

接口定义

```
+ (NSString *)getApnsDeviceToken;
```

上报设备deviceToken

向阿里云移动推送注册该设备的deviceToken，可在APNs注册成功回调中调用该接口。

接口定义

```
+ (void)registerDevice:(NSData *)deviceToken
          withCallback:(CallbackHandler)callback;
```

参数说明

参数	类型	是否必选	说明
deviceToken	NSData	是	苹果APNs服务器返回的deviceToken。
callback	Block	否	回调

2.2.6. 通知上报相关接口

> 说明
>
> 以下接口中提供回调的接口均为异步执行。

上报通知点击事件

上报通知点击事件ACK到推送服务器。

> 注意
>
> 用于替换SDK V1.8.1之前的 `handleLaunching:` 和 `handleReceiveRemoteNotification:` 上报接口。

接口定义

```
+ (void)sendNotificationAck:(NSDictionary *)userInfo;
```

参数说明

参数	类型	是否必填	说明
userInfo	NSDictionary	是	通知payload。

上报通知删除事件

上报通知删除事件ACK到推送服务器，详细使用参见iOS通知删除上报配置。

> **注意**
>
> - SDK V1.9.9.2及以上版本支持。
> - iOS 10及以上系统版本支持。

接口定义

```
+ (void)sendDeleteNotificationAck:(NSDictionary *)userInfo;
```

参数说明

参数	类型	是否必选	说明
userInfo	NSDictionary	是	通知payload。

上报通知点击事件（App处于关闭状态）

上报通知点击事件到移动推送服务器。

> **注意**
>
> - 点击通知将App从关闭状态拉起时，在 `didFinishLaunchingWithOptions` 回调中调用该接口。
> - SDK V1.8.1以下版本支持。

接口定义

```
+ (void)handleLaunching:(NSDictionary *)launchOptions;
```

参数说明

参数	类型	是否必填	说明
launchOptions	NSDictionary	是	`didFinishLaunchingWithOptions` 回调中的 `launchOptions` 参数。

上报通知点击事件（App处于打开状态）

上报通知点击事件到移动推送服务器。

> **注意**
>
> - SDK V1.8.1以下版本支持。
> - App处于打开状态（前台或后台），在 `didReceiveRemoteNotification` 回调中调用该接口。
> - App处于前台，通知不弹窗，直接触发回调；App处于后台，通知弹窗并触发回调。

接口定义

```
+ (void)handleReceiveRemoteNotification:(NSDictionary *)userInfo;
```

参数说明

参数	类型	是否必填	说明
userInfo	NSDictionary	是	`didReceiveRemoteNotification` 回调中的参数 `userInfo` 。

2.2.7. 角标同步接口

> **说明**
>
> 以下接口中提供回调的接口均为异步执行。

同步角标数到服务端

同步设备当前角标数到推送服务端，配合角标自增功能（参考 OpenAPI 2.0 高级推送接口，搜索 `iOSBadgeAutoIncrement` ）使用。

> 注意
>
> SDK V1.9.5及以上版本支持。

接口定义

```
+ (void)syncBadgeNum:(NSUInteger)num
        withCallback:(CallbackHandler)callback;
```

参数说明

参数	类型	是否必填	说明
num	NSUInteger	是	角标数，取值范围[0,99999]
callback	Block	否	回调

2.2.8. 推送通道监听接口

监听推送消息通道建立

通知中心注册事件名为 `CCPDidChannelConnectedSuccess` 的广播监听；

推送通道成功建立后，发出事件名为 `CCPDidChannelConnectedSuccess` 的广播通知。

代码示例

```
- (void)listenerOnChannelOpened {
    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(onChannelOpened:)
                                                 name:@"CCPDidChannelConnectedSuccess"
                                               object:nil];
}
// 通道打开通知
- (void)onChannelOpened:(NSNotification *)notification {
}
```

消息接收监听

通知中心注册事件名为 `CCPDidReceiveMessageNotification` 的广播监听；

成功接收消息后，发出事件名为 `CCPDidReceiveMessageNotification` 的广播通知。

代码示例

```
- (void) registerMessageReceive {
    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(onMessageReceived:)
                                                 name:@"CCPDidReceiveMessageNotification"
                                               object:nil];
}
- (void)onMessageReceived:(NSNotification *)notification {
    CCPSysMessage *message = [notification object];
    NSString *title = [[NSString alloc] initWithData:message.title encoding:NSUTF8StringEnc
oding];
    NSString *body = [[NSString alloc] initWithData:message.body encoding:NSUTF8StringEncod
ing];
    NSLog(@"Receive message title: %@, content: %@.", title, body);
}
```

通知打开监听

iOS 10+

- App处于前台收到通知

```
/**
 *  App处于前台时收到通知(iOS 10+)
 */
- (void)userNotificationCenter:(UNUserNotificationCenter *)center willPresentNotification
:(UNNotification *)notification withCompletionHandler:(void (^)(UNNotificationPresentatio
nOptions))completionHandler {
    NSLog(@"Receive a notification in foregound.");
    // 处理iOS 10通知相关字段信息
    [self handleiOS10Notification:notification];
    // 通知不弹出
    //completionHandler(UNNotificationPresentationOptionNone);
    // 通知弹出，且带有声音、内容和角标（App处于前台时不建议弹出通知）
    completionHandler(UNNotificationPresentationOptionSound | UNNotificationPresentationO
ptionAlert | UNNotificationPresentationOptionBadge);
}
```

- 触发通知动作时回调，比如点击、删除通知和点击自定义action（iOS 10+）。

```
/**
 *  触发通知动作时回调，比如点击、删除通知和点击自定义action(iOS 10+)
 */
- (void)userNotificationCenter:(UNUserNotificationCenter *)center didReceiveNotificationR
esponse:(UNNotificationResponse *)response withCompletionHandler:(void (^)())completionHa
ndler {
    NSString *userAction = response.actionIdentifier;
    // 点击通知打开
    if ([userAction isEqualToString:UNNotificationDefaultActionIdentifier]) {
        NSLog(@"User opened the notification.");
        // 处理iOS 10通知，并上报通知打开回执
        [self handleiOS10Notification:response.notification];
    }
    // 通知dismiss，category创建时传入UNNotificationCategoryOptionCustomDismissAction才可以
触发
```

```
    if ([userAction isEqualToString:UNNotificationDismissActionIdentifier]) {
        NSLog(@"User dismissed the notification.");
    }
    NSString *customAction1 = @"action1";
    NSString *customAction2 = @"action2";
    // 点击用户自定义Action1
    if ([userAction isEqualToString:customAction1]) {
        NSLog(@"User custom action1.");
    }
    // 点击用户自定义Action2
    if ([userAction isEqualToString:customAction2]) {
        NSLog(@"User custom action2.");
    }
    completionHandler();
}

/**
 *  处理iOS 10通知(iOS 10+)
 */
- (void)handleiOS10Notification:(UNNotification *)notification {
    UNNotificationRequest *request = notification.request;
    UNNotificationContent *content = request.content;
    NSDictionary *userInfo = content.userInfo;
    // 通知时间
    NSDate *noticeDate = notification.date;
    // 标题
    NSString *title = content.title;
    // 副标题
    NSString *subtitle = content.subtitle;
    // 内容
    NSString *body = content.body;
    // 角标
    int badge = [content.badge intValue];
    // 取得通知自定义字段内容，例：获取key为"Extras"的内容
    NSString *extras = [userInfo valueForKey:@"Extras"];
    // 通知打开回执上报
    [CloudPushSDK sendNotificationAck:userInfo];
    NSLog(@"Notification, date: %@, title: %@, subtitle: %@, body: %@, badge: %d, extras:
%@.", noticeDate, title, subtitle, body, badge, extras);
}
```

iOS 10以下版本

- App处于关闭状态，点击打开通知；

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDiction
ary *)launchOptions {
    // 点击通知将App从关闭状态启动时，将通知打开回执上报
    // [CloudPushSDK handleLaunching:launchOptions];(Deprecated from v1.8.1)
    [CloudPushSDK sendNotificationAck:launchOptions];
    return YES;
}
```

- App处于打开状态时，点击打开通知；

```
- (void)application:(UIApplication*)application didReceiveRemoteNotification:(NSDictionary*)userInfo {
    NSLog(@"Receive one notification.");
    // 取得APNS通知内容
    NSDictionary *aps = [userInfo valueForKey:@"aps"];
    // 内容
    NSString *content = [aps valueForKey:@"alert"];
    // badge数量
    NSInteger badge = [[aps valueForKey:@"badge"] integerValue];
    // 播放声音
    NSString *sound = [aps valueForKey:@"sound"];
    // 取得Extras字段内容
    NSString *Extras = [userInfo valueForKey:@"Extras"]; //服务端中Extras字段，key是自己定义的
    NSLog(@"content = [%@], badge = [%ld], sound = [%@], Extras = [%@]", content, (long)badge, sound, Extras);
    // iOS badge 清0
    application.applicationIconBadgeNumber = 0;
    // 通知打开回执上报
    // [CloudPushSDK handleReceiveRemoteNotification:userInfo];(Deprecated from v1.8.1)
    [CloudPushSDK sendNotificationAck:userInfo];
}
```

2.3. iOS 配置推送证书指南

本文介绍如何获取APNs证书。

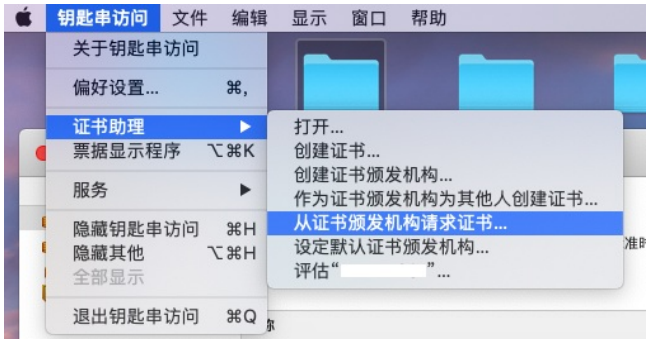
使用iOS推送功能前，请按此文档获取推送证书，并将证书上传到移动推送控制台。

- 获取CSR文件
- 创建App ID
- 创建推送证书
- 上传证书到移动推送控制台
- 证书验证

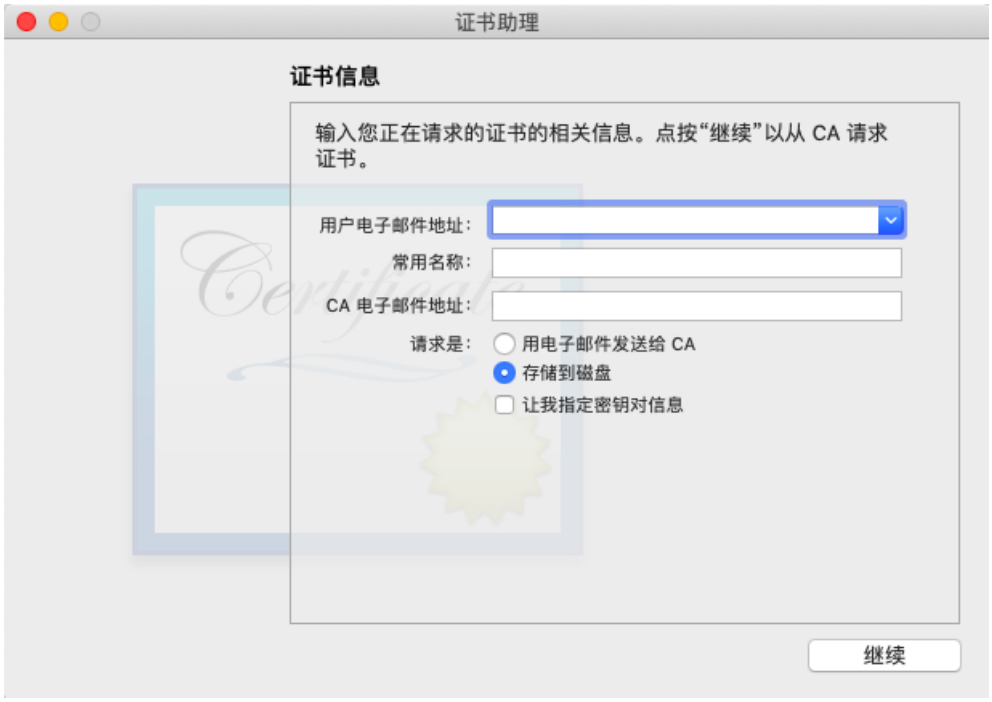
获取CSR文件

> **说明**
>
> 说明：CSR是Cerificate Signing Request的英文缩写，即证书请求文件。证书申请者在申请数字证书时由CSP（加密服务提供者）在生成私钥的同时也生成证书请求文件。证书申请者只要把CSR文件提交给证书颁发机构后，证书颁发机构使用其根证书私钥签名就生成了证书公钥文件，也就是颁发给用户的证书。

1、在Mac电脑的应用程序中打开**钥匙串访问**，在顶部菜单栏中选择**钥匙串访问>证书助理>从证书颁发机构请求证书**。

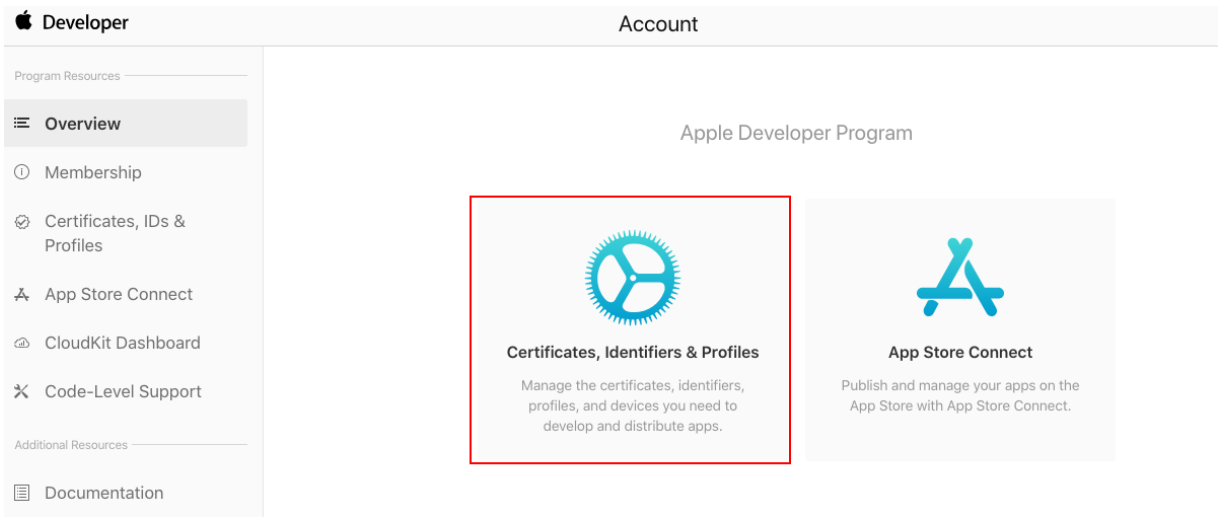


2、在弹出的证书信息中，输入邮箱地址，设置选择储存到磁盘，单击继续将CSR文件存储到本地。

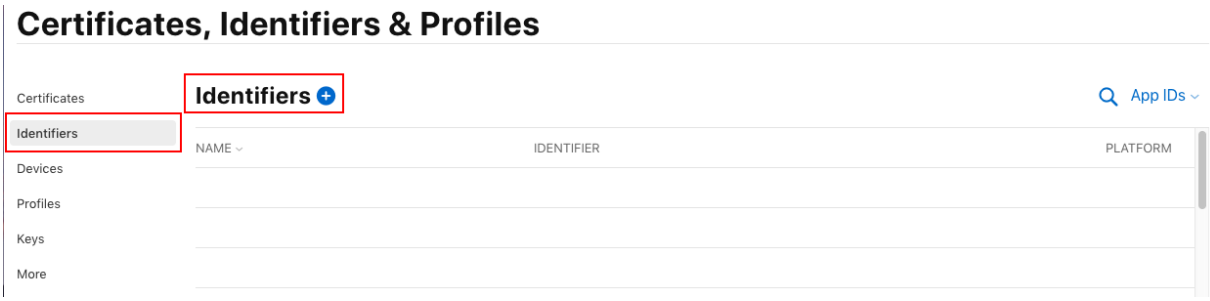


创建App ID

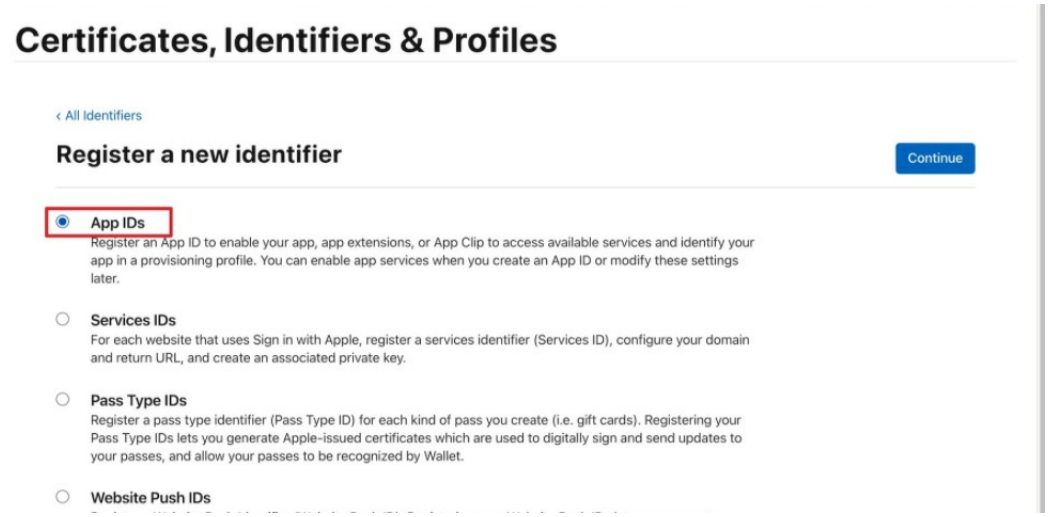
1、登录Apple Member Center，选择 `Certificates，Identifiers & Profiles` 选项。



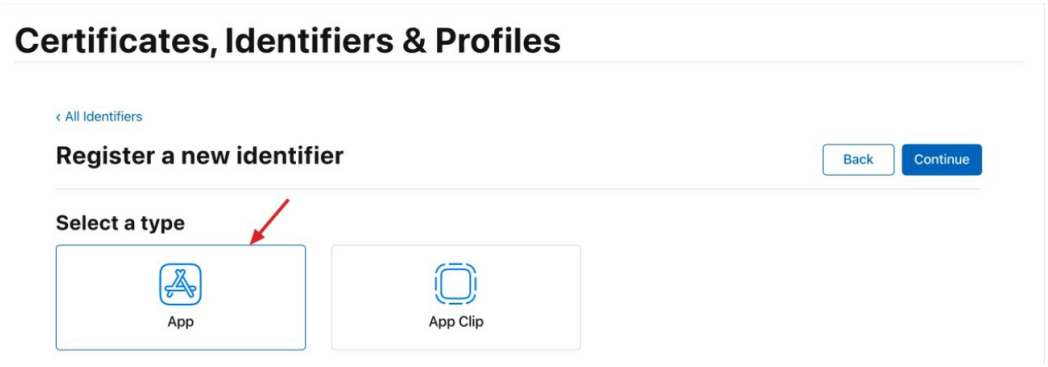
2、选择Identify，单击Identify右侧的 `+` 。



3、选择App IDs，单击 `Continue` 。



4、选择App，单击 `Continue` 。



5、配置Bundle ID等其他信息，下滑页面勾选Push Notifications，开启远程推送服务后再单击 `Continue` 。

图1：配置Bundle ID等信息

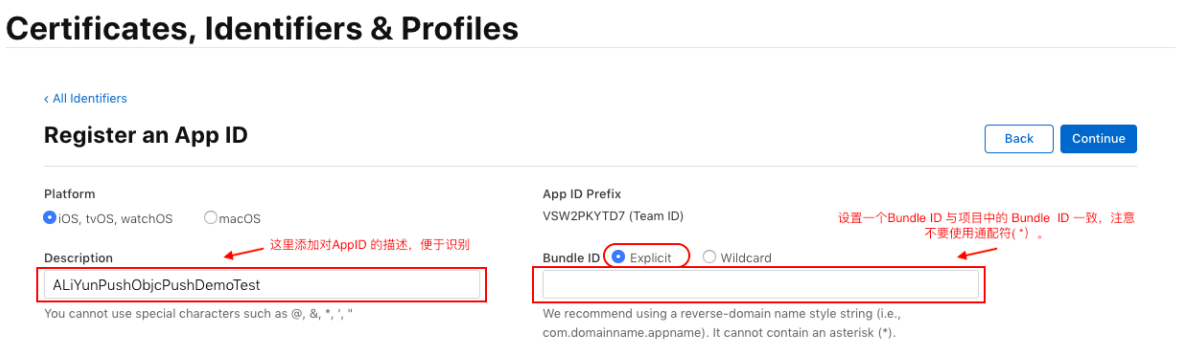
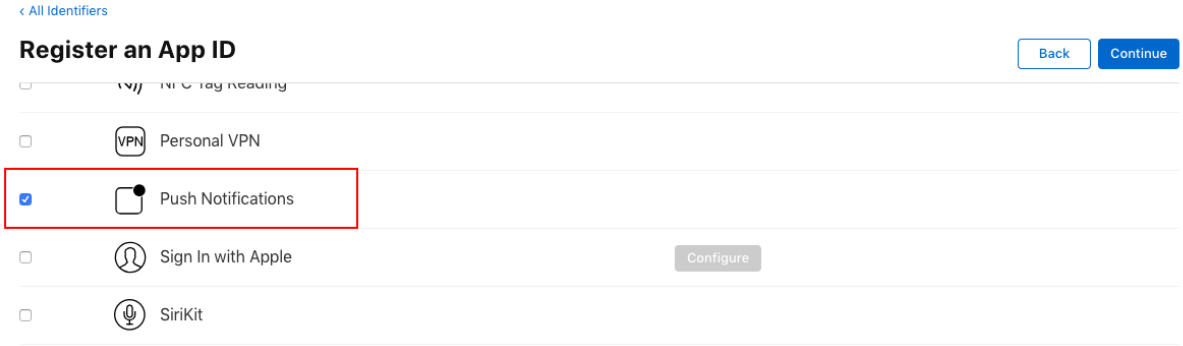


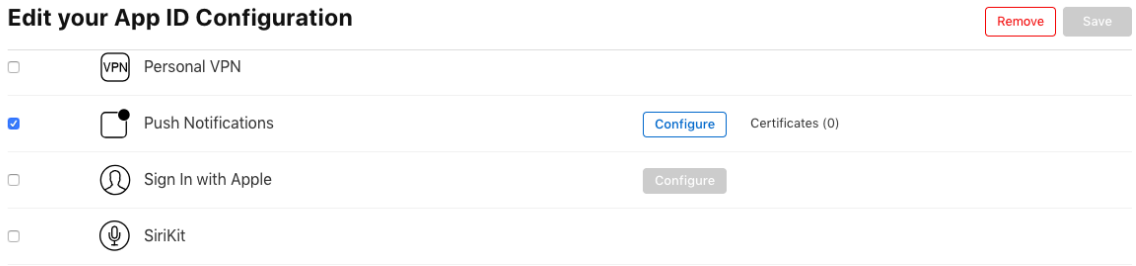
图2：开启远程推送服务



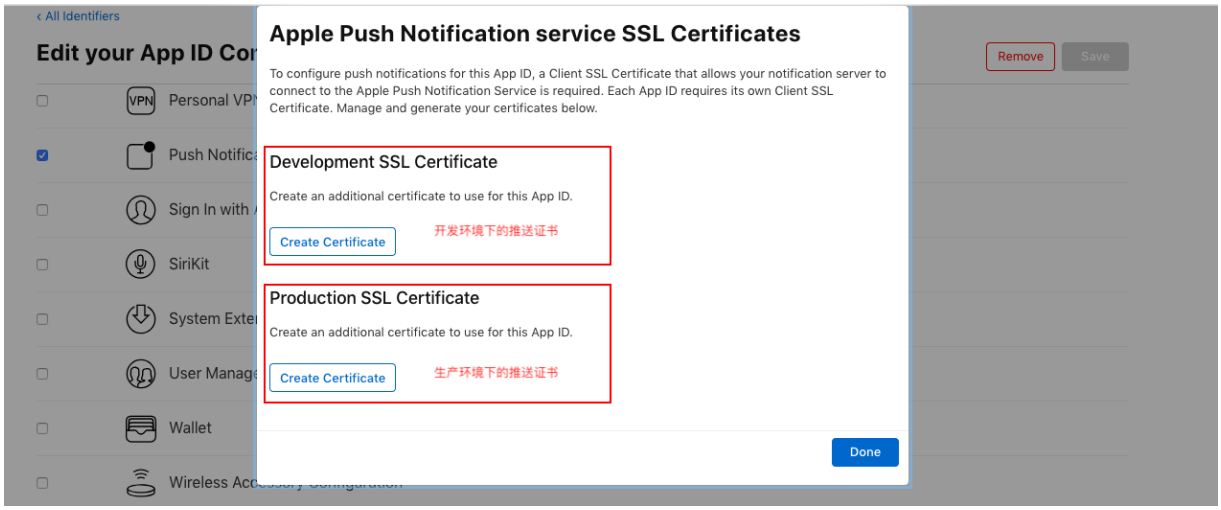
创建推送证书

1、单击您创建好的App ID，进入 `Edit your App ID Configuration` 页面。

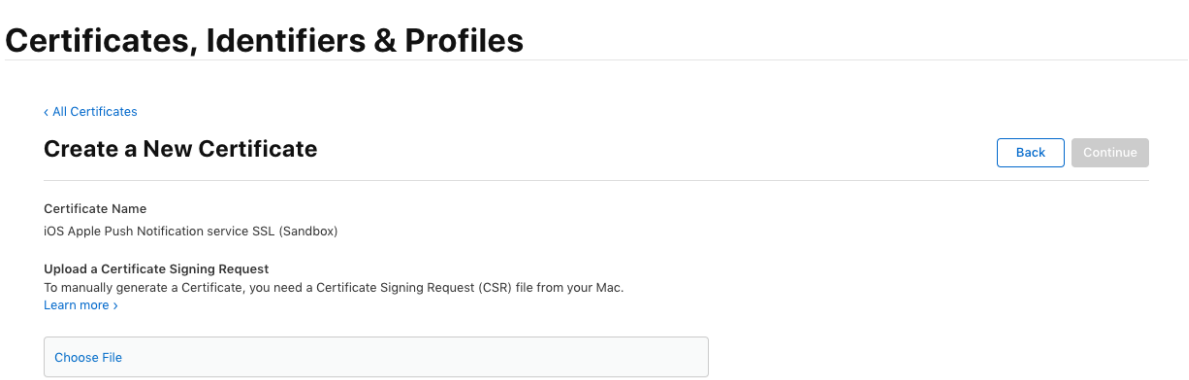
2、下滑页面单击 `Push Notifications` 右侧的Configure。



3、选择开发环境（Development）的Create Certificate进行推送证书配置。

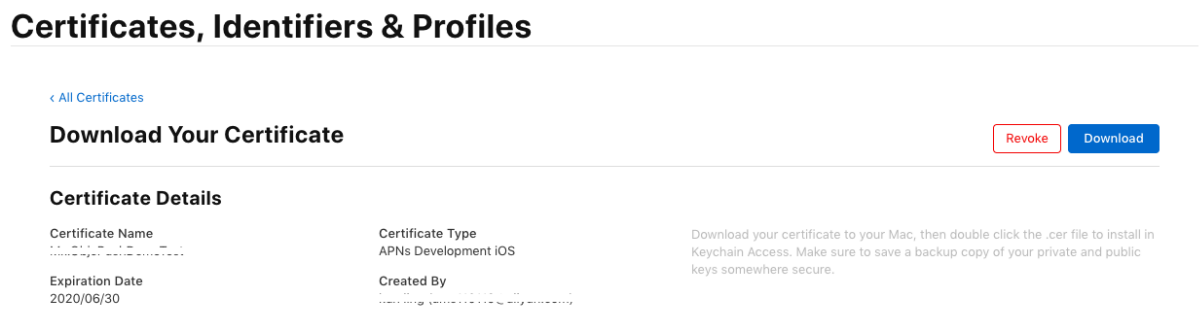


4、单击Choose File上传已获取到的CSR文件。



5、单击 `Continue` ，即可生成开发环境的推送证书。

6、单击 `Download` ，将开发环境的证书下载到本地。



7、重复上述步骤1~6，生成生产环境的证书，并下载到本地。

8、双击打开下载的开发环境和生产环境证书，系统会将其导入钥匙串中。

9、在Mac中打开钥匙串应用，选择**登录>证书**，分别右键导出开发环境和生产环境的.P12证书文件。

> **注意**
>
> 保存.P12文件时请设置密码，密码将在之后移动推送控制台上传证书时使用。

上传证书到移动推送控制台

获取iOS推送证书后需将推送证书上传至移动推送控制台，配置方法参见配置推送证书。

证书验证

SmartPush一款iOS远程推送测试程序。

Mac OS下的APNS工具APP，iOS Push Notification Debug App

地址：https://github.com/shaojiankui/SmartPush

2.4. iOS静默通知

iOS静默通知

iOS静默通知（iOS Silent Notification），属于特殊的远程推送通知，其目的不是为了弹出通知框提醒用户，而是用于后台运行的App和服务端同步数据。例：App在后台放置一段时间，网络已不再活跃，App内数据可能已经过时；服务端可推送一条携带参数的静默通知，处于后台的App可以触发静默通知回调，在后台运行状态下获取对应参数并发起网络请求，获取最新数据更新，整个过程用户无感知。

静默通知限制和注意事项：

- 静默通知主要用于更新和同步数据，用户对其无感知，因此静默通知一般不设置通知内容、声音和角标。
- 静默通知唤醒后台App并执行下载任务时，最多有30秒时间执行。
- App处于前台/后台时均可触发对应通知回调，App关闭后不能触发。
- 静默通知请求在APNs属于低优先级任务，苹果不保证静默通知的到达率。
- 不要利用静默通知对App进行保活，APNs若检测到较高频率的静默通知发送请求，可能会终止其发送（具体策略苹果未公开）。

服务端配置

静默通知一般不设置推送内容、声音和角标，发送到APNs的 `payload` 格式如下，其中需要设定 `content-available` 为 `1`，标记为静默通知。

```
{
"aps":{
"content-available":1
},
"key1":"value1",
"key2":"value2"
}
```

移动推送服务端发送静默通知，需要基于 `OpenAPI 2.0的推送高级接口` ：

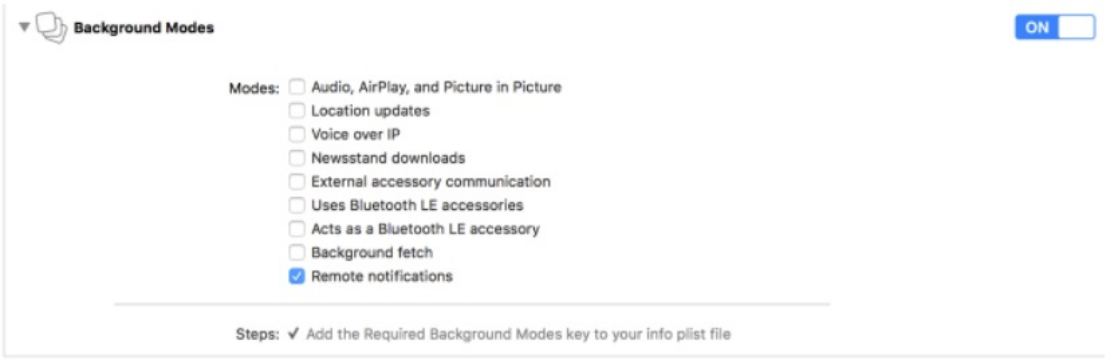
- 接口描述参考OpenAPI 2.0 - API列表 - 推送高级接口。
- `iOSSilentNotification` 设置为 `True` 时，表示使能静默通知。

客户端适配

静默通知回调处理：触发 `didReceiveRemoteNotification` 回调。

App若需要在后台处理静默通知，Xcode工程做以下修改：

- Xcode中选中App对应Project，并选中对应iOS App Target。
- 打开 `Capabilities Tab` ，使能 `Background Modes` ，并勾选 `Remote notifications` 。



2.5. iOS10通知适配

1. iOS 10通知简介

iOS 10系统对推送通知做了较大增强，远程推送通知相关主要体现在以下几点：

- 统一通知相关的API和Framework;
- 通知注册和回调接口修改;
- 通知内容更丰富，支持富媒体（图片、音频、视频等）推送;
- 通知详情自定义UI;

总的来说，iOS 10提供了更简洁易用的通知相关接口，提高了处理通知的自由度。

> **说明**
>
> 阿里云移动推送对iOS 10通知相关feature配置，通过OpenAPI高级推送接口配置，请参考OpenAPI推送高级接口。 阅读本文档时，可参照iOS推送Demo中iOS 10通知相关的代码进行学习和试用。

2. Framework依赖

- `UserNotifications.framework`，iOS 10通知相关类和接口都包含在内；
- 引用如下：

```
#import <UserNotifications/UserNotifications.h>
```

3. 通知字段

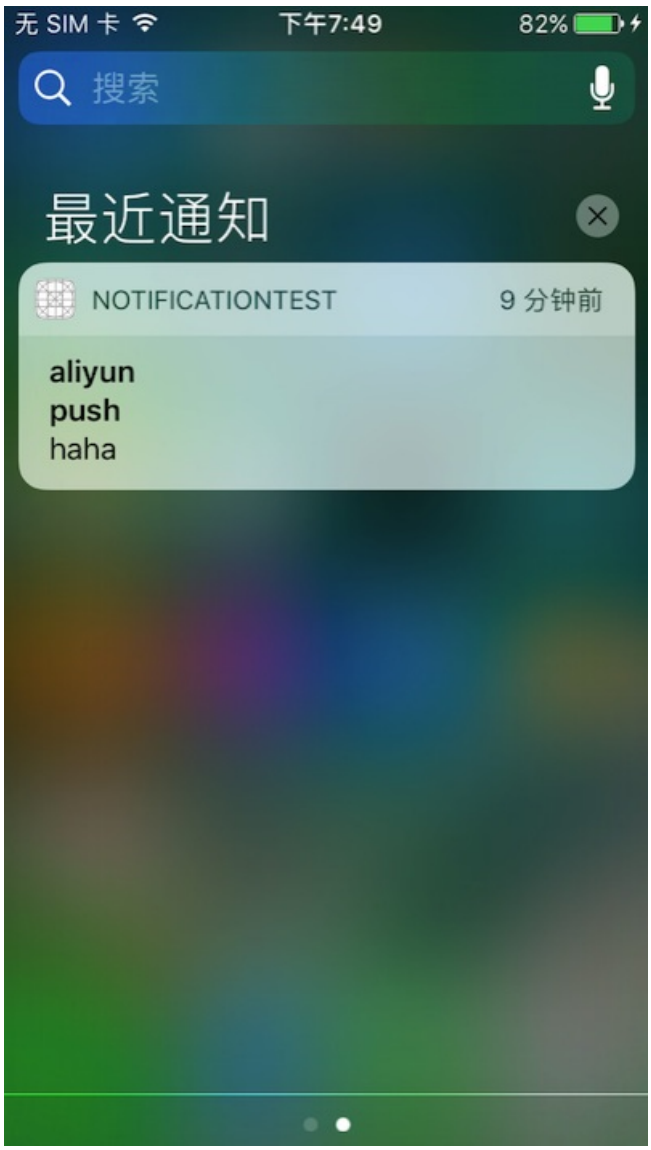
- 原本iOS通知仅支持设置通知内容，可通过OpenAPI的 `Summary` 字段设置，参考OpenAPI推送高级接口字段， **老版本服务端推送通知payload字段** 如下：

```
{
    "aps": {
        "alert": {
        "your notification body",
        },
        "badge": 1,
        "sound": "default",
    },
    "key1":"value1",
    "key2":"value2"
}
```

- iOS 10通知支持设置 **标题(title)** 、 **副标题(subtitle)** 、 **内容(body)** 、 **通知扩展字段(mutable-content)** 、 **通知类别(category)** ，服务端配置参考上述的OpenAPI推送高级接口， **当前服务端推送通知payload字段** 参考如下：

```
{
    "aps": {
        "alert": {
            "title": "title",
            "subtitle": "subtitle",
            "body": "body"
        },
        "badge": 1,
        "sound": "default",
        "category": "test_category",
        "mutable-content": 1
    },
    "key1":"value1",
    "key2":"value2"
}
```

- **【注意】使用OpenAPI推送时，若没有进行iOS 10通知相关配置，通知payload保持老版本不变，已经保证对老版本payload兼容性；若进行iOS 10通知相关配置，请确保客户端业务逻辑对payload相关字段处理的兼容性。**
- iOS 10设备收到通知效果如下，其中 `标题` 为”aliyun”， `副标题` 为”push”， `内容` 为”haha”。 **`title` 设置后，iOS 10+系统显示title如下；【iOS 8.2 <= iOS系统 < iOS 10】，通知应用名称会显示该标题。**



4. 通知中心

4.1 简介

- 基于 `UNUserNotificationCenter` 对象进行通知的调度和管理通知相关的行为，具体如下：
  - 为通知的提醒、声音和角标请求授权；
  - 声明通知类别和可执行动作；
  - 管理通知的弹出；
  - 管理通知在设备通知中心的展示；
  - 获取App通知相关的配置。

- 通知中心对象获取方式如下：

```
UNUserNotificationCenter *center = [UNUserNotificationCenter currentNotificationCenter];
```

4.2 请求授权并向APNs注册

- App使用推送功能前，需要向用户请求推送功能授权，如下所示；
- 第一次调用 `requestAuthorizationWithOptions` 请求授权时，App会弹出如下图所示的授权框，注意App卸载重装前该授权框仅弹出一次，若用户点击“不允许”，需要引导用户到“设置”中打开，推送功能才能正常使用。

- `requestAuthorizationWithOptions` 回调中可捕获用户是否点击授权，在成功授权回调中调用 `registerForRemoteNotifications` ，向APNs注册获取设备的deviceToken，App再次启动时虽然不会弹出授权框，但推送授权请求可获取App推送配置，可触发成功/失败授权回调。
- APNs注册成功/失败回调保持不变，在成功回调中调用阿里云推送SDK接口，将deviceToken上报到阿里云推送服务器。
- 主动调用 `getNotificationSettingsWithCompletionHandler` 接口，回调中可获取App推送授权状态。

```
UNUserNotificationCenter *center = [UNUserNotificationCenter currentNotificationCenter];
if (systemVersionNum >= 10.0) {
    center = [UNUserNotificationCenter currentNotificationCenter];
    [center requestAuthorizationWithOptions:UNAuthorizationOptionAlert | UNAuthorizationOpt
ionBadge | UNAuthorizationOptionSound completionHandler:^(BOOL granted, NSError * _Nullable
error) {
        if (granted) {
            // granted
            NSLog(@"User authored notification.");
            dispatch_async(dispatch_get_main_queue(), ^{
                [application registerForRemoteNotifications];
            });
        } else {
            // not granted
            NSLog(@"User denied notification.");
        }
    }];
}

/*
 *  APNs注册成功回调，将返回的deviceToken上传到CloudPush服务器
 */
- (void)application:(UIApplication *)application didRegisterForRemoteNotificationsWithDevic
eToken:(NSData *)deviceToken {
    NSLog(@"Upload deviceToken to CloudPush server.");
    [CloudPushSDK registerDevice:deviceToken withCallback:^(CloudPushCallbackResult *res) {
        if (res.success) {
            NSLog(@"Register deviceToken success, deviceToken: %@", [CloudPushSDK getApnsDe
```

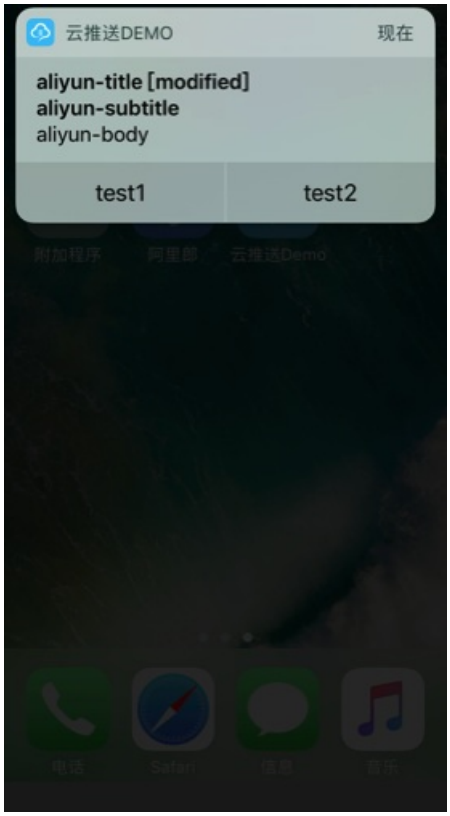
```
viceToken]);
        } else {
            NSLog(@"Register deviceToken failed, error: %@", res.error);
        }
    }];
}

/*
 *  APNs注册失败回调
 */
- (void)application:(UIApplication *)application didFailToRegisterForRemoteNotificationsWit
hError:(NSError *)error {
    NSLog(@"Get deviceToken failed, error: %@", error);
}

// 主动获取设备通知是否授权(iOS 10+)
- (void)getNotificationSettingStatus {
    [center getNotificationSettingsWithCompletionHandler:^(UNNotificationSettings * _Nonnul
l settings) {
        if (settings.authorizationStatus == UNAuthorizationStatusAuthorized) {
            NSLog(@"User authed.");
        } else {
            NSLog(@"User denied.");
        }
    }];
}
```

4.3 Action和Category

- iOS 8以上支持，此处仅讲述iOS 10系统的实现方式。
- 通知支持设置 `Action` 点击动作，即在通知上添加按钮，点击按钮可触发回调以此做出不同的逻辑处理；
- 通知支持 `Category` 分类，可将 `Action` 和 `Category` 进行关联， `Category` 和第6节（通知详情自定义UI）相关。
- 下面代码自定义id为 `action1` 和 `action2` 的通知动作，创建id为 `test_category` 的通知类别后，将两个Action关联到该category，最后注册category到通知中心。
- 使用OpenAPI推送通知时，调用 `setiOSNotificationCategory()` 接口，可指定通知的类别；创建的 `test_category` 类别的通知弹出时如下图所示， `test1` 和 `test2` 按钮分别对应id为 `action1` 和 `action2` 的通知Action；
- 【注意】 `Category` 注册到通知中心需要在推送前完成。




```
/**
 *  创建并注册通知category(iOS 10+)
 */
- (void)createCustomNotificationCategory {
    // 自定义`action1`和`action2`
    UNNotificationAction *action1 = [UNNotificationAction actionWithIdentifier:@"action1" t
itle:@"test1" options: UNNotificationActionOptionNone];
    UNNotificationAction *action2 = [UNNotificationAction actionWithIdentifier:@"action2" t
itle:@"test2" options: UNNotificationActionOptionNone];
    // 创建id为`test_category`的category，并注册两个action到category
    // UNNotificationCategoryOptionCustomDismissAction表明可以触发通知的dismiss回调
    UNNotificationCategory *category = [UNNotificationCategory categoryWithIdentifier:@"tes
t_category" actions:@[action1, action2] intentIdentifiers:@[] options:
                                        UNNotificationCategoryOptionCustomDismissAction];
    // 注册category到通知中心
    [center setNotificationCategories:[NSSet setWithObjects:category, nil]];
}
```

4.4 通知回调

- `UNUserNotificationCenterDelegate` 协议定义了通知相关的回调；

4.4.1 设置代理

- 若要处理通知相关回调，需要实现并指定通知中心对象 `UNUserNotificationCenter` 的代理 `UNUserNotificationCenterDelegate` ，一般是在AppDelegate中实现，如下所示：

```
@interface AppDelegate () <UNUserNotificationCenterDelegate>
@end

@implementation AppDelegate
...
UNUserNotificationCenter *center = [UNUserNotificationCenter currentNotificationCenter];
center.delegate = self;
...
@end
```

4.4.2 回调1：App在前台收到通知

- 当App处于前台，收到通知会触发 `userNotificationCenter:willPresentNotification:withCompletionHandler:` 回调；
- 在回调中可以处理通知相关的字段信息，回调处理结束前需要调用 `completionHandler(UNNotificationPresentationOptions)` ， `UNNotificationPresentationOptions` 的参数含义如下：
  - `UNNotificationPresentationOptionNone` ，通知不提醒；
  - `UNNotificationPresentationOptionSound` ，通知声音提醒；
  - `UNNotificationPresentationOptionAlert` ，通知内容提醒；
  - `UNNotificationPresentationOptionBadge` ，通知角标提醒。
- 基于此，App在前台时也可以将通知弹出。

```
/**
 *  处理iOS 10通知(iOS 10+)
 */
- (void)handleiOS10Notification:(UNNotification *)notification {
    UNNotificationRequest *request = notification.request;
    UNNotificationContent *content = request.content;
    NSDictionary *userInfo = content.userInfo;
    // 通知时间
    NSDate *noticeDate = notification.date;
    // 标题
    NSString *title = content.title;
    // 副标题
    NSString *subtitle = content.subtitle;
    // 内容
    NSString *body = content.body;
    // 角标
    int badge = [content.badge intValue];
    // 取得通知自定义字段内容，例：获取key为"Extras"的内容
    NSString *extras = [userInfo valueForKey:@"Extras"];
    // 通知打开回执上报
    [CloudPushSDK handleReceiveRemoteNotification:userInfo];
    NSLog(@"Notification, date: %@, title: %@, subtitle: %@, body: %@, badge: %d, extras: %
@.", noticeDate, title, subtitle, body, badge, extras);
}

/**
 *  App处于前台时收到通知(iOS 10+)
 */
- (void)userNotificationCenter:(UNUserNotificationCenter *)center willPresentNotification:(
UNNotification *)notification withCompletionHandler:(void (^)(UNNotificationPresentationOpt
ions))completionHandler {
    NSLog(@"Receive a notification in foregound.");
    // 处理iOS 10通知相关字段信息
    [self handleiOS10Notification:notification];
    // 通知不弹出
    //completionHandler(UNNotificationPresentationOptionNone);
    // 通知弹出，且带有声音、内容和角标（App处于前台时不建议弹出通知）
    completionHandler(UNNotificationPresentationOptionSound | UNNotificationPresentationOpt
ionAlert | UNNotificationPresentationOptionBadge);
}
```

4.4.3 回调2：点击/清除通知

- 点击/清除通知时， 可在 `userNotificationCenter:didReceiveNotificationResponse:withCompletionHandler:` 回调里捕获到这些动作，根据 `UNNotificationResponse.actionIdentifier` 可对这些动作进行区分：
  - 点击通知打开App，对应 `UNNotificationDefaultActionIdentifier` ；
  - 左滑删除通知，对应 `UNNotificationDismissActionIdentifier` ，注册 `Category` 时，需传入 `UNNotificationCategoryOptionCustomDismissAction` 才可以捕获到该动作，具体见4.3节的 `Category` 创建和注册；

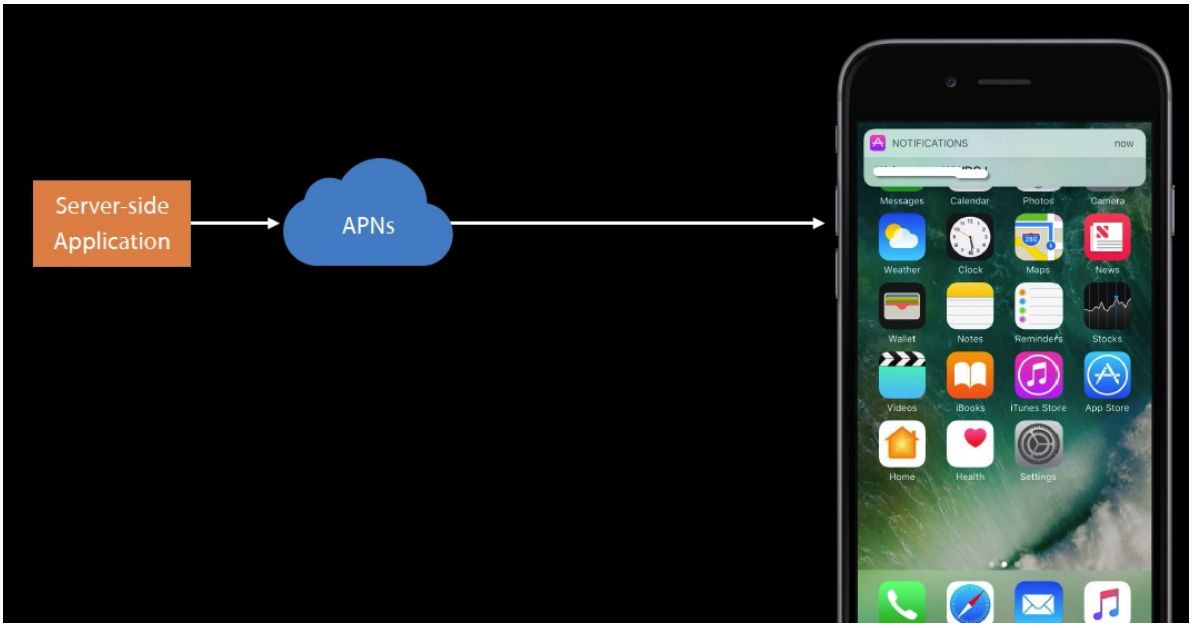
- 点击自定义Action，如点击4.3节创建的id为 `action1` 和 `action2` 的Action，自定义 `Action` 点击动作的捕获的好处在于，即使不进入App同样可完成某些逻辑处理。

- 【注意】两个通知回调是不冲突的，当App处于前台时，收到通知先触发 `userNotificationCenter:willPresentNotification:withCompletionHandler:` 回调；之后若有点击通知动作，再触发 `userNotificationCenter:didReceiveNotificationResponse:withCompletionHandler:` 回调。

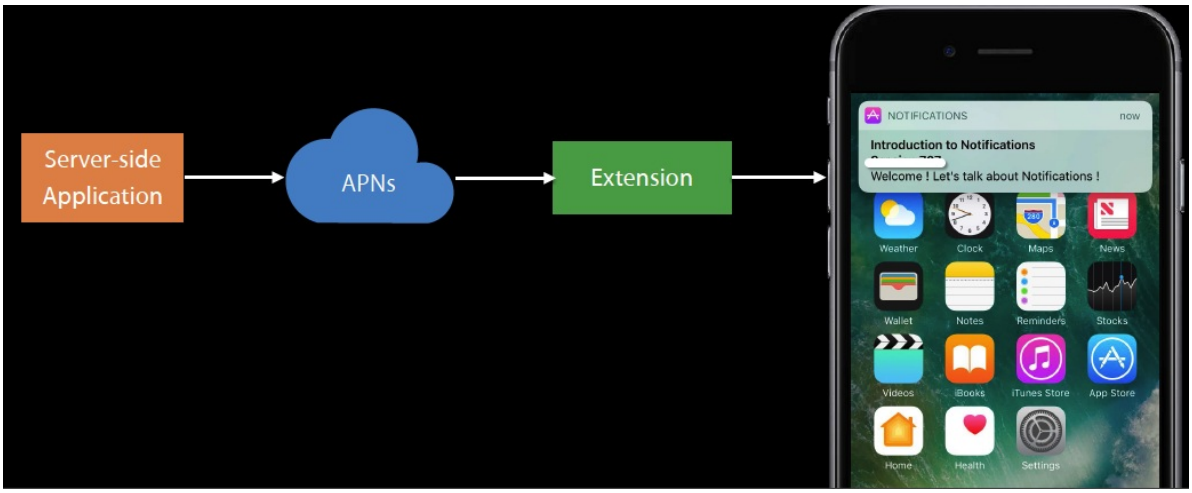
```
/**
 *  触发通知动作时回调，比如点击、删除通知和点击自定义action(iOS 10+)
 */
- (void)userNotificationCenter:(UNUserNotificationCenter *)center didReceiveNotificationResponse:(UNNotificationResponse *)response withCompletionHandler:(void (^)())completionHandler {
    NSString *userAction = response.actionIdentifier;
    // 点击通知打开
    if ([userAction isEqualToString:UNNotificationDefaultActionIdentifier]) {
        NSLog(@"User opened the notification.");
        // 处理iOS 10通知，并上报通知打开回执
        [self handleiOS10Notification:response.notification];
    }
    // 通知dismiss，category创建时传入UNNotificationCategoryOptionCustomDismissAction才可以触发
    if ([userAction isEqualToString:UNNotificationDismissActionIdentifier]) {
        NSLog(@"User dismissed the notification.");
    }
    NSString *customAction1 = @"action1";
    NSString *customAction2 = @"action2";
    // 点击用户自定义Action1
    if ([userAction isEqualToString:customAction1]) {
        NSLog(@"User custom action1.");
    }
    // 点击用户自定义Action2
    if ([userAction isEqualToString:customAction2]) {
        NSLog(@"User custom action2.");
    }
    completionHandler();
}
```

5. 富媒体推送

- iOS 10添加了通知相关的扩展 `Notification Service Extension`，使得通知弹出前可以对通知内容进行修改。
- iOS远程推送过程如下图所示，APNs推送的通知直接在设备上弹出；



- 添加 `Notification Service Extension` 后，如下图所示，APNs推送的通知在弹出前，可先到达 `Extension` 进行处理，【注意】OpenAPI需要调用 `setiOSMutableContent(true)` 接口，这样 `Extension` 才可生效。

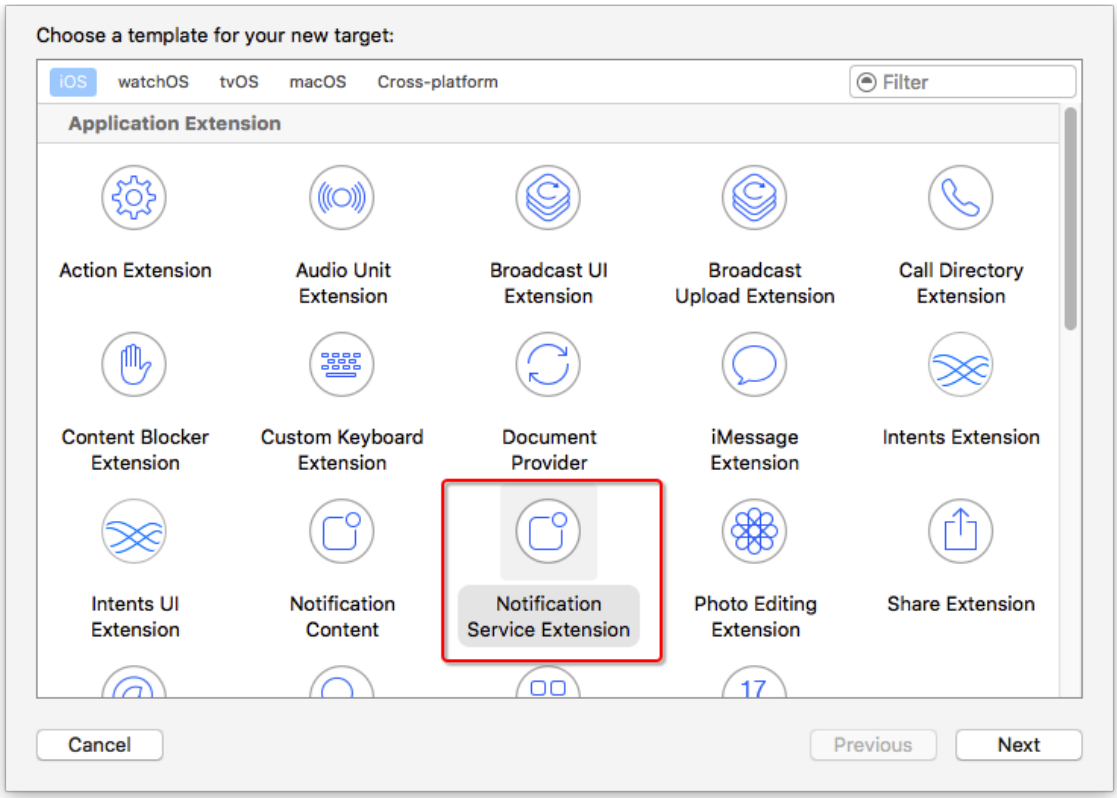


- 在 `Service Extension` 后台预处理阶段，可从远程服务器下载或从本地获取 **富媒体** （图片、音频、视频）资源，将其作为 `attachment` 添加到通知中，富媒体资源类型和大小限制如下：

Attachment	Supported File Types	Maximum Size
Audio	kUTTypeAudioInterchangeFileFormat kUTTypeWaveformAudio kUTTypeMP3 kUTTypeMPEG4Audio	5 MB
Image	kUTTypeJPEG kUTTypeGIF kUTTypePNG	10 MB
Movie	kUTTypeMPEG kUTTypeMPEG2Video kUTTypeMPEG4 kUTTypeAVIMovie	50 MB

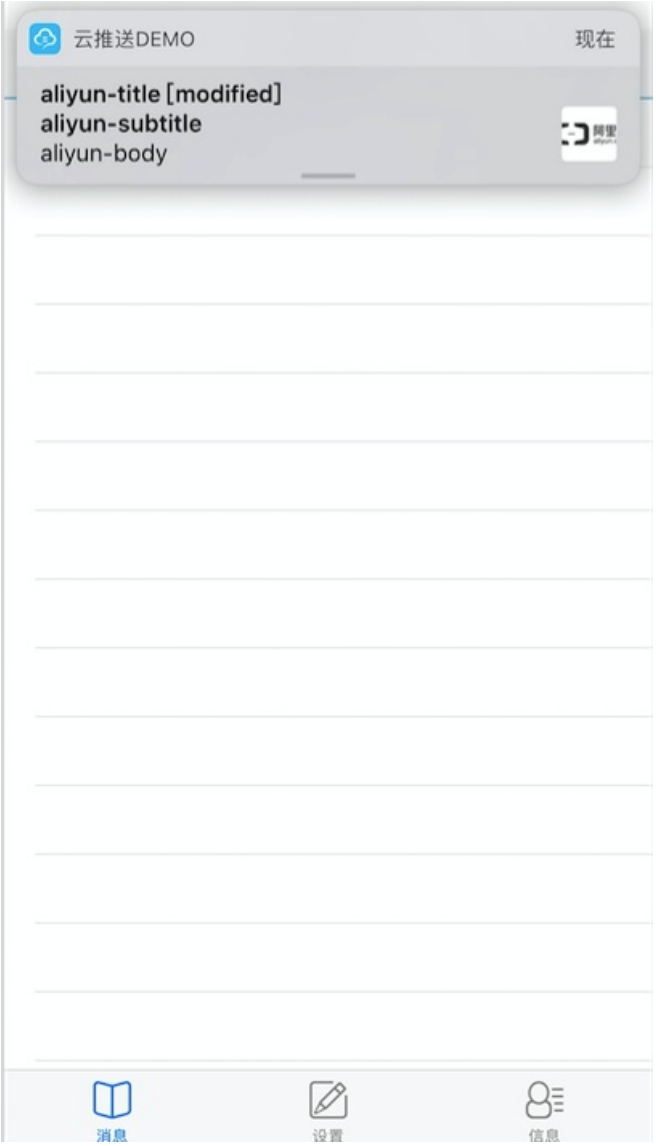
- `Notification Service Extension` 添加步骤：

- Xcode -> File -> New -> Target，选择 `Notification Service Extension` ，如下图所示：



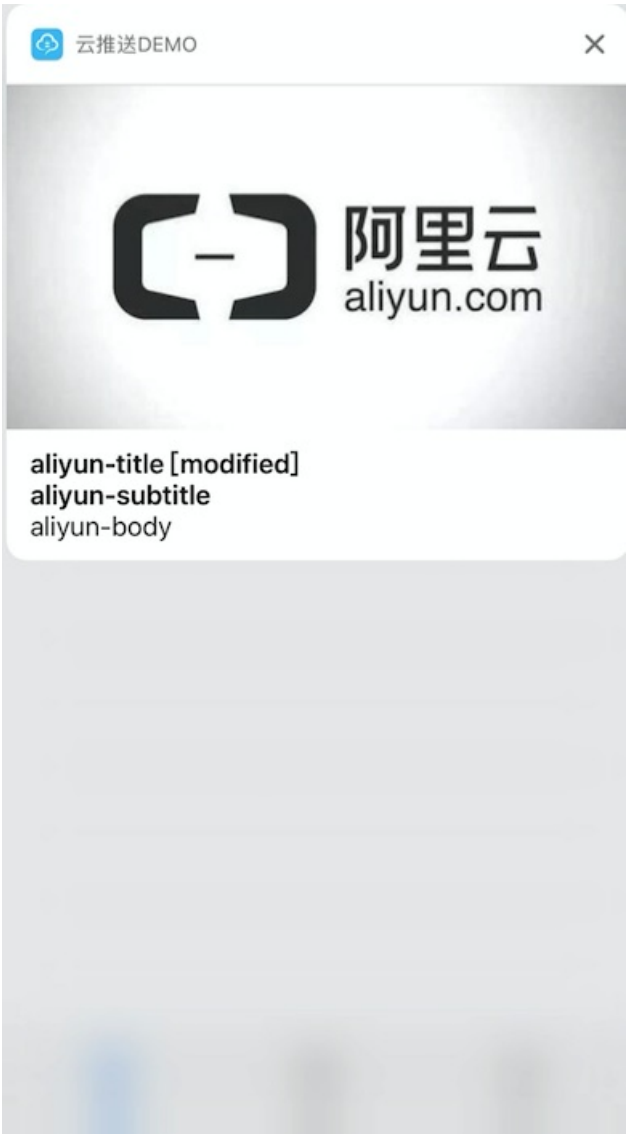
- 输入Target名，创建完成后在目录下Xcode会自动生成 `NotificationService` 的模板，在 `didReceiveNotificationRequest` 回调方法中，处理通知弹出前的动作。

- 可参考iOS Demo Notification Service Extension的实现携带图片的推送通知，从OpenAPI设定的自定义参数 `attachment` 字段中获取图片Url，或者从本地获取图片资源，效果如下图所示。
- 【注意】从远程服务器获取富媒体资源时，同样需遵循App Transport Security (ATS)的原则，若需要请求HTTP资源请参考ATS配置，为 `Service Extension` Target进行配置；建议限制为请求HTTPS资源。



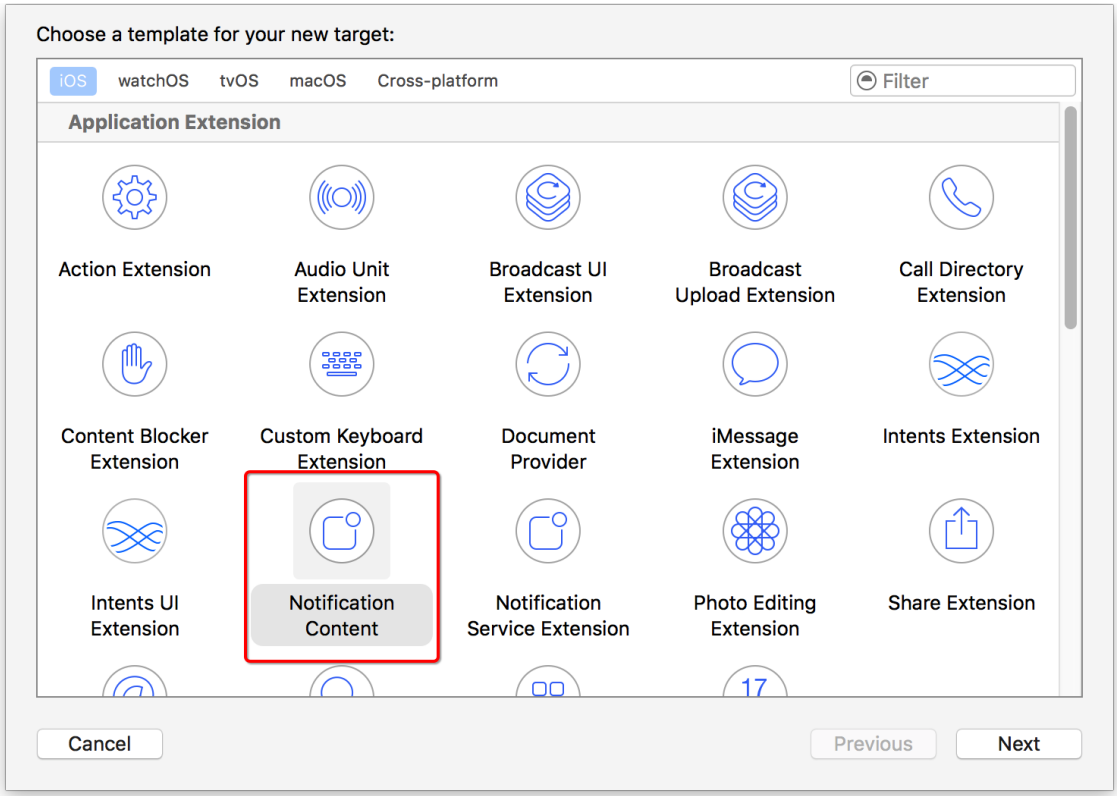
6. 通知详情自定义UI

- 除了 `Notification Service Extension` ，另一个通知相关的Extension为内容扩展 `Content Extension` ，可用于自定义通知详情UI，如修改样式、颜色等。
- iOS 10收到通知后，支持下拉通知（经测试iPhone 5c不支持，建议使用iPhone 6以上手机测试） or 3D touch展开通知详情，携带图片的通知详情样式默认如下图所示，内容扩展可针对通知详情进行定制。


云推送DEMO
阿里云
aliyun.com
aliyun-title [modified]
aliyun-subtitle
aliyun-body

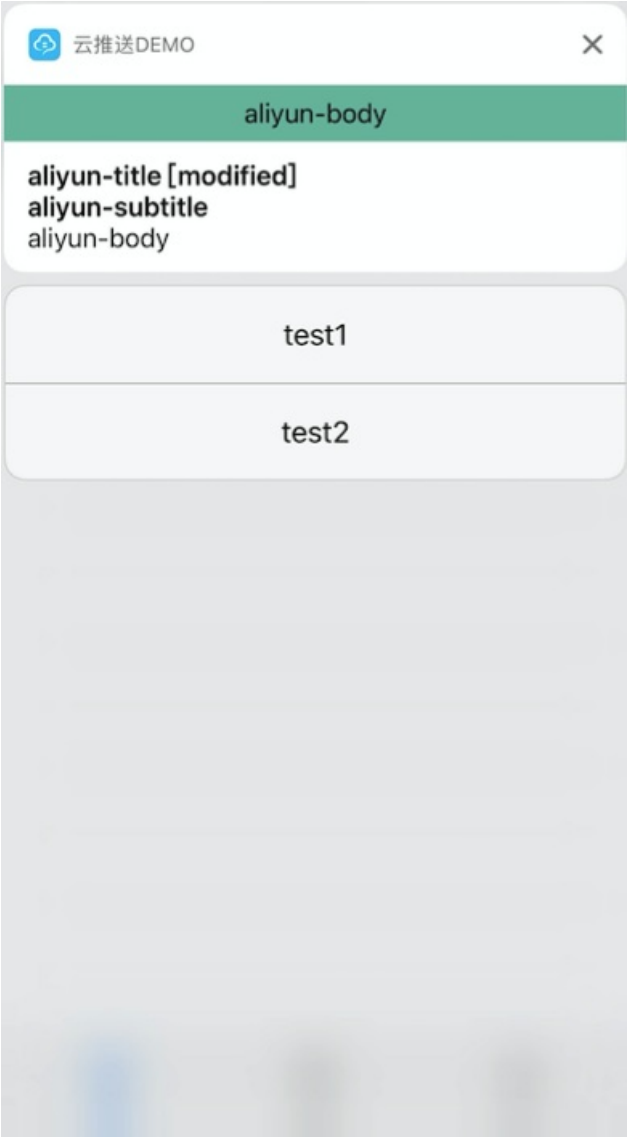
- 内容扩展添加步骤：

- Xcode -> File -> New -> Target，选择 `Notification Content` ，如下图所示：



- 输入Target名，Xcode自动生成 `NotificationViewController` 头文件和源文件， `MainInterface.storyboard` 和 `Info.plist` ，其中 `NotificationViewController` 和 `MainInterface.storyboard` 一起定义了通知详情的UI。
- `Info.plist`中 自动生成 `NSExtension` 相关KV配置，具体含义如下所示：
  - NSExtensionAttributes
    - UNNotificationExtensionCategory，指定自定义通知详情UI适用于哪些category，可为String or Dictionary；（必需）
    - UNNotificationExtensionInitialContentSizeRatio，通知视图长宽比例；（必需）
    - UNNotificationExtensionDefaultContentHidden，原本通知内容是否隐藏，若不指定，默认为NO；（可选）
  - NSExtensionMainStoryboard，storyboard文件名，默认填充为 `MainInterface` ；（必需）
  - NSExtensionPointIdentifier，默认填充为 `com.apple.usernotifications.content-extension` ；（必需）

- OpenAPI推送时，必需通过 `setiOSNotificationCategory` 接口指定通知category，只有指定的category在Info.plist的 `UNNotificationExtensionCategory` 设置，才能保证通知详情自定义UI生效。
- 参考iOS Demo Notification Content Extension的实现方式，进行通知详情自定义UI的设置，如下图所示，绿色的 `aliyun-body` 为自定义的展示UI，字段内容通过拷贝通知内容得来。



2.6. iOS通知删除上报配置

概述

iOS10之后，通知相关类API被统一，通过UserNotifications.framework来统一管理和使用通知。这里主要说明自定义Action结合SDK实现iOS删除上报数据。

相关类

- UNNotificationAction（具体action）
- UNNotificationCategory（action簇集合）
- UNUserNotificationCenter（通知center）

代码实现

1. 创建对应action

```
/**
 *  创建并注册通知category(iOS 10+)
 */
- (void)createCustomNotificationCategory {
    // 自定义`action1`和`action2`
    UNNotificationAction *action1 = [UNNotificationAction actionWithIdentifier:UNNotifi
cationDefaultActionIdentifier title:@"普通_进入前台" options: UNNotificationActionOptionF
oreground];
    UNNotificationAction *action2 = [UNNotificationAction actionWithIdentifier:UNNotifi
cationDismissActionIdentifier title:@"删除" options: UNNotificationActionOptionDestructi
ve];
    // 创建id为`test_category`的category，并注册两个action到category
    // UNNotificationCategoryOptionCustomDismissAction表明可以触发通知的dismiss回调
    UNNotificationCategory *category = [UNNotificationCategory categoryWithIdentifier:@
"test_category" actions:@[action1, action2] intentIdentifiers:@[] options:
                                        UNNotificationCategoryOptionCustomDismissAction
];
    // 注册category到通知中心
    [self.userNotificationCenter setNotificationCategories:[NSSet setWithObjects:catego
ry, nil]];
}
```

2. 通知代理中处理对应action事件

```
- (void)userNotificationCenter:(UNUserNotificationCenter *)center didReceiveNotificatio
nResponse:(UNNotificationResponse *)response withCompletionHandler:(void (^)(void))comp
letionHandler {
    NSString *userAction = response.actionIdentifier;
    if ([userAction isEqualToString:UNNotificationDefaultActionIdentifier]) {
        // 处理iOS 10通知，并上报通知打开回执
        [self handleiOS10Notification:response.notification];
    }
    // 通知dismiss，category创建时传入UNNotificationCategoryOptionCustomDismissAction才可
以触发
    if ([userAction isEqualToString:UNNotificationDismissActionIdentifier]) {
        //处理用户删除通知上报
    // 阿里云推送SDK 支持删除数据上报
    [CloudPushSDK sendDeleteNotificationAck:response.notification.request.content.userI
nfo]
    }
    completionHandler();
}
```

PayLoad

key: category

value: 客户端注册的category ID（本例子中注册id = 'test_category'）

```
{
  "aps":{
    "alert":"自定义Action",
    "sound":"default",
    "badge":1,
    "category":"test_category"
  }
}
```

注意事项

- 在注册UNNotificationCategory时，需要使用UNNotificationCategoryOptionCustomDismissAction类型
- 如果只是单纯监听删除事件，可以注册UNNotificationCategory，创建0个action
- 通知栏中的全部清除无法监听

2.7. 错误处理

- 调用 `CloudPushSDK` 的相关接口时，如果发生错误，可以从 `CallbackHandler` 回调对象中获取错误码和错误描述等信息。
- CallbackHandler定义如下，可从回调处理对象 `res` 中获取
  - `success` （接口调用是否成功）；
  - `data` （调用成功后返回相关数据）；
  - `error` （错误信息描述）。

```
typedef void (^CallbackHandler)(CloudPushCallbackResult *res);
```

常见错误码

错误名称	错误码	错误描述
INIT_INVALID_APPKEY_CODE	1011	appKey配置错误
INIT_INVALID_APPSECRET_CODE	1012	appSecret配置错误
INIT_SESSION_FAILED_CODE	1013	session初始化失败
INIT_AS_ERROR_CODE	1014	连接AS错误，检查网络连接
INIT_SID_ERROR_CODE	1015	sid获取失败
TAG_INPUT_INVALID_CODE	2001	标签输入为空
TAG_APPID_INVALID_CODE	2002	appId错误
TAG_RPC_REQUEST_FAILED_CODE	2003	标签请求错误
ACCOUNT_INVALID_ACCOUNT_CODE	3001	account参数输入错误
ACCOUNT_CHANNEL_CLOSED_CODE	3002	推送通道关闭
ACCOUNT_REQUEST_TIMEOUT_CODE	3003	绑定账号请求超时
ACCOUNT_ENCODER_STATUS_ERROR_CODE	3004	绑定账号状态码错误
ALIAS_INPUT_INVALID_CODE	4001	别名输入为空
VIP_REQ_HTTP_ERROR_CODE	5001	VIP请求状态码错误
VIP_REQ_CONNECTION_ERROR_CODE	5002	VIP请求连接错误

错误名称	错误码	错误描述
VIP_REQ_SERVER_ERROR_CODE	5003	VIP请求服务错误
VIP_REQ_GERNERATE_PARAM_ERROR_CODE	5004	VIP参数生成错误
OTHER_ERROR_INVLIAD_PARA_CODE	6001	其他输入错误

3.客户端集成uni-app插件

EMAS移动推送提供了uni-app插件，帮助uni-app开发者在多端一次性快速集成移动推送功能。uniapp是一个使用Vue.js开发跨平台应用的前端框架。开发者编写一套代码，uni-app可将其编译到iOS、Android平台，保证其正确运行并达到优秀体验。

平台兼容性

- Android：

适用版本区间：4.4-11.0

支持CPU类型：armeabi-v7a，arm64-v8a，x86

- iOS：

适用版本区间：9-15

官方维护

官方维护的版本放在【DCloud插件市场】上以开源的形式发布。如果需要下载打包版本，请访问如下链接：

移动推送uniapp插件包下载及操作指导