SDK 用户手册

（Android 版）

发布日期： 2020年05月29日

百度云推送（push.baidu.com）

目录

第1章 简介

百度 Push 服务 Android SDK 是百度官方推出的 Push 服务的 Android 平台开发 SDK，提供给 Android 开发者简单的接口，轻松集成百度 Push 推送服务。

Android Push 服务以后台 service 方式运行。由于 Android 系统限制，从 Push SDK 7.0.0 版本开始，每个应用都会开启一个后台 service，提高消息到达率。

Push service 运行于一个独立进程，不和主进程运行于同一进程，主程序不需要常驻内存，当 Push service 接受到 Push 消息后，会通过 Intent 接口发送给主程序处理。

Push Android SDK 的完整下载包为 Baidu-Push-SDK-Android-L2-VERSION.zip，VERSION 是版本号，如 7.0.0。下载解压后的目录结构如下所示：

- Demo_AS

  Android Studio 格式示例工程，可以快速帮助用户了解如何使用 SDK。
- docs

  版本说明

  升级指南

  用户手册
- libs

  pushservice-VERSION.jar：push SDK 以 jar 的方式提供；

  libbdpush_V3_4.so： Push 服务需要用到的 JNI 资源。请将您要支持的对应体系的 so 文件夹拷贝到您的工程 libs 目录下。
- 产品动态 7.0.0.txt

  Android SDK 7.0.0 版本 Change Log 和升级指南

第2章 阅读对象

本文档面向所有使用该 SDK 的 Android 开发人员、测试人员、合作伙伴以及对此感兴趣的其他用户。

第3章 SDK 功能说明

3.1 框架设计

Push Android SDK 是开发者与 Push 服务器之间的桥梁。可以让用户越过复杂的 Push HTTP/HTTPS API，直接和 Push 服务器进行交互来使用 Push 服务。（框架设计如图 1 所示）

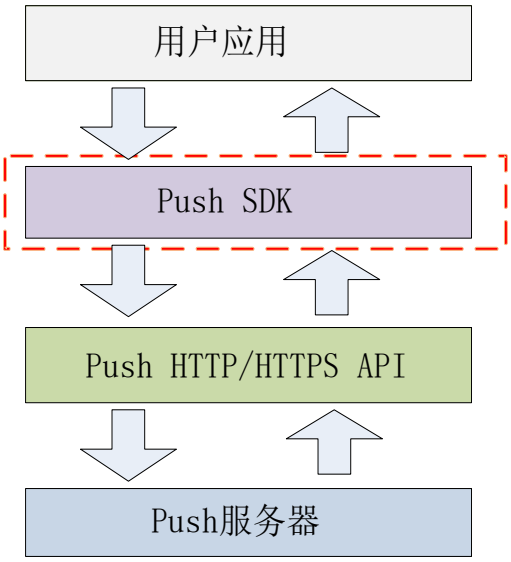

图 1 Push SDK 框架图

3.2 主要功能

本 SDK 主要提供以下功能的接口：

1. Push 服务
   - Push 开启厂商推送代理（包括华为、小米、OPPO、VIVO、魅族）
   - Push 服务初始化及绑定
   - Push 服务停止和恢复
   - Push 免打扰时段设置
2. Tag 管理

   创建、删除、列出标签
   - 创建 Tag
   - 删除 Tag
   - 列举 Tag
3. 通知推送

   接收和展现通知，还提供自定义通知栏样式的功能，包括：
   - 设置自定义通知的 Builder
   - 设置默认的通知的 Builder

4. 推送效果反馈
   - 推送的到达情况统计
5. 服务设置
   - 开启调试模式

第4章 开发前准备

4.1 运行环境

可运行于 Android 2.3（API Level 9）及以上版本。

4.2 参数申请及权限开通

4.2.1 获取应用 ID 及 API Key

开发者需要使用百度账号登录百度云推送官网，注册成为百度开发者，并创建应用，方可获取应用 APP ID、对应的 API KEY 及 SECRET KEY 等信息。详情请参考百度云推送官网中“创建应用”的相关介绍。

其中，应用 APP ID 用于标识开发者创建的应用程序；API KEY 是开发者创建的应用程序的唯一标识，开发者在调用百度 API 时必须传入此参数。

4.2.2 集成厂商推送代理

从 Push SDK 7.0.0 版本开始，Push SDK 支持华为、小米、OPPO、VIVO、魅族厂商代理。如需集成厂商代理请参考厂商接入文档说明。

华为最新 HMS SDK 下载及接入请参考华为推送开发准备

小米最新 PUSH SDK 请从小米开放平台下载

OPPO 最新 PUSH SDK 请从 OPPO PUSH 客户端 SDK 接口文档下载

VIVO 最新 PUSH SDK 请从 VIVO Android PUSH-SDK 集成指南下载

魅族最新 PUSH SDK 请从 Flyme 推送 SDK 下载

4.3 账户支持

4.3.1 无账户登录体系

开发者无需接入百度账户体系，每个终端直接通过 API KEY 向 Push Server 请求设备标识 channelId，此 id 是根据端上的属性生成，具备唯一性，开发者可通过此 id 对应到自己的账户体系，此方式方便灵活。无账户登录体系启动 Android 端 Push 服务的方法如下：

```
PushManager.startWork(getApplicationContext(), PushConstants.LOGIN_TYPE_API_KEY,
"{api_key}");
```

第5章 使用 SDK 开发应用

5.1 添加 SDK 到 APP 工程 （具体操作可参考 PushDemo_AS 工程）

1. 创建一个 Android Project

注意：**如果您的 Android 工程使用的是 Android API level 21 及以上的版本，您的通知图标背景必须是透明的，否则在 Android5.0 及以上的机器上通知图标可能会变成白色的方块。**

2. 在该工程下创建一个 libs 文件夹
3. 将 pushservice-VERSION.jar 拷贝到 libs 文件夹中，把 SDK 压缩包中的 libs/armeabi 目录下的 libbdpush_V3_4.so 拷贝到工程对应的 jniLibs/armeabi 文件夹下。
   a) 如果你的工程中没有使用其他的.so，建议只复制 armeabi 文件夹。
   b) 如果你的工程中还使用了其他的.so 文件，只需要拷贝云推送对应目录下的.so 文件。
4. AndroidManifest.xml 声明 permission

```
    <!-- Push service 运行需要的权限 -->
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.READ_PHONE_STATE" />
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.VIBRATE" />
    <uses-permission android:name="android.permission.DISABLE_KEYGUARD" />

    <!-- Push service 运行的可选权限 -->
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />

    <!-- 适配 Android N 系统必需的 ContentProvider 写权限声明，写权限包含应用包名-->
    <uses-permission
android:name="baidu.push.permission.WRITE_PUSHINFOPROVIDER.YourPackageName" />
    <permission
            android:name="baidu.push.permission.WRITE_PUSHINFOPROVIDER.YourPackageName"
            android:protectionLevel="signature">
    </permission>
    <!-- Push service 运行需要的权限 END -->

    <!-- 小米代理运行需要的权限 -->
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
    <uses-permission android:name="android.permission.READ_PHONE_STATE" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.GET_TASKS" />
    <uses-permission android:name="android.permission.VIBRATE" />
    <!-- 小米推送权限 -->
```

```
<permission
        android:name="YourPackageName.permission.MIPUSH_RECEIVE"
        android:protectionLevel="signature" />
<uses-permission android:name="YourPackageName.permission.MIPUSH_RECEIVE" />
<!--  小米代理运行需要的权限  END -->

<!-- VIVO 代理运行需要的权限  -->
<uses-permission android:name="android.permission.INTERNET" />
<!-- VIVO 代理运行需要的权限  END -->
```

5. AndroidManifest.xml 注册消息接收 Receiver

客户端需实现自己的 MyPushMessageReceiver，接收 Push 服务的消息，并实现对消息的处理。以下是 AndroidManifest.xml 中的配置代码。

```
<!-- push 应用定义消息 receiver 声明  -->
<receiver android:name="YourPackageName.MyPushMessageReceiver">
    <intent-filter>
        <!--  接收 push 消息  -->
        <action android:name="com.baidu.android.pushservice.action.MESSAGE" />
        <!--  接收 bind、setTags 等 method 的返回结果  -->
        <action android:name="com.baidu.android.pushservice.action.RECEIVE" />
        <!--  可选，接受通知点击事件，和通知自定义内容  -->
        <action android:name="com.baidu.android.pushservice.action.notification.CLICK" />
        <!--  使用华为代理功能必须声明,用于接收华为的透传  -->
        <action android:name="com.huawei.android.push.intent.RECEIVE" />
    </intent-filter>
</receiver>
```

6. AndroidManifest.xml 中其他必须组件配置

```
<!-- Push 服务接收客户端发送的各种请求-->
<receiver android:name="com.baidu.android.pushservice.RegistrationReceiver"
        android:process=":bdservice_v1" >
        <intent-filter>
            <action android:name="com.baidu.android.pushservice.action.METHOD" />
        </intent-filter>
</receiver>

<service android:name="com.baidu.android.pushservice.PushService"
        android:exported="true"
        android:process=":bdservice_v1" >
        <intent-filter >
                <action android:name="com.baidu.android.pushservice.action.PUSH_SERVICE" />
        </intent-filter>
</service>

<!-- 4.4 版本新增的 CommandService 声明，提升小米和魅族手机上的实际推送到达率  -->
<service android:name="com.baidu.android.pushservice.CommandService"
        android:exported="true" />

<!--  适配 Android N 系统必需的 ContentProvider 声明，写权限包含应用包名-->
<provider
        android:name="com.baidu.android.pushservice.PushInfoProvider"
        android:authorities="YourPackageName.bdpush"
        android:writePermission="baidu.push.permission.WRITE_PUSHINFOPROVIDER.YourPack
ageName"
        android:protectionLevel = "signature"
        android:exported="true"
        android:process=":bdservice_v1" />
<!--  可选声明，提升 push 消息送达率  -->
<service
        android:name="com.baidu.android.pushservice.job.PushJobService"
        android:permission="android.permission.BIND_JOB_SERVICE"
        android:process=":bdservice_v1" />

<!-- push 必须的组件声明  END -->
```

```
<!-- 华为代理推送必需组件 -->
<activity
    android:name="com.baidu.android.pushservice.hwproxy.HwNotifyActivity"
    android:exported="true"
    android:launchMode="singleTask"
    android:theme="@android:style/Theme.NoDisplay">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <data
            android:host="bdpush"
            android:path="/hwnotify"
            android:scheme="baidupush" />
    </intent-filter>
</activity>
<!-- 华为 HMS 接入声明 service start -->
<service
    android:name="com.baidu.android.pushservice.HmsPushPatchMessageService"
    android:exported="false">
    <intent-filter>
        <action android:name="com.huawei.push.action.MESSAGING_EVENT" />
    </intent-filter>
</service>
<!-- 华为代理推送必需组件 END -->

<!-- 小米代理推送必需组件 -->
<service
    android:name="com.xiaomi.push.service.XMPushService"
    android:enabled="true"
    android:process=":pushservice" />
<!--注：此 service 必须在 3.0.1 版本以后（包括 3.0.1 版本）加入-->
<service
    android:name="com.xiaomi.push.service.XMJobService"
    android:enabled="true"
    android:exported="false"
    android:permission="android.permission.BIND_JOB_SERVICE"
    android:process=":pushservice" />
```

```
<service
        android:name="com.xiaomi.mipush.sdk.PushMessageHandler"
        android:enabled="true"
        android:exported="true" />

<!--注：此 service 必须在 2.2.5 版本以后（包括 2.2.5 版本）加入-->
<service
        android:name="com.xiaomi.mipush.sdk.MessageHandleService"
        android:enabled="true" />
<receiver
        android:name="com.xiaomi.push.service.receivers.NetworkStatusReceiver"
        android:exported="true">
        <intent-filter>
                <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
                <category android:name="android.intent.category.DEFAULT" />
        </intent-filter>
</receiver>

<receiver
        android:name="com.xiaomi.push.service.receivers.PingReceiver"
        android:exported="false"
        android:process=":pushservice">
        <intent-filter>
                <action android:name="com.xiaomi.push.PING_TIMER" />
        </intent-filter>
</receiver>

<receiver android:name="com.baidu.android.pushservice.PushPatchMessageReceiver">
        <intent-filter>
                <action android:name="com.xiaomi.mipush.RECEIVE_MESSAGE" />
                <action android:name="com.xiaomi.mipush.MESSAGE_ARRIVED" />
                <action android:name="com.xiaomi.mipush.ERROR" />
        </intent-filter>
</receiver>
<!--  小米代理推送必需组件  END -->
```

```
<!-- 魅族代理推送必需组件 -->
<activity
    android:name="com.baidu.android.pushservice.mzproxy.MzNotifyActivity"
    android:launchMode="singleTask"
    android:theme="@android:style/Theme.NoDisplay" />

<receiver
    android:name="com.baidu.android.pushservice.MzPushPatchMessageReceiver">
    <intent-filter>
        <action android:name="com.meizu.flyme.push.intent.MESSAGE" />
        <action android:name="com.meizu.flyme.push.intent.REGISTER.FEEDBACK" />
        <action android:name="com.meizu.flyme.push.intent.UNREGISTER.FEEDBACK" />
    </intent-filter>
</receiver>
<!-- 魅族代理推送必需组件 END -->


<!-- OPPO 代理推送必需组件，注意将 com.baidu.push.example 替换成自己的应用包名 -->
<activity
    android:name="com.baidu.android.pushservice.opproxy.OpNotifyActivity"
    android:configChanges="orientation|keyboardHidden"
    android:exported="true"
    android:launchMode="singleInstance"
    android:theme="@android:style/Theme.NoDisplay">
    <intent-filter>
        <action android:name="com.baidu.push.example.action.RECEIVE_MCS_MESSAGE" />

        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
<!-- OPPO 代理推送必需组件 END -->
```

```
<!-- VIVO 代理推送必需组件  -->
<service
        android:name="com.vivo.push.sdk.service.CommandClientService"
        android:exported="true" />

<receiver android:name="com.baidu.android.pushservice.viproxy.ViPushMessageReceiver">
        <intent-filter>
            <action android:name="com.vivo.pushclient.action.RECEIVE" />
        </intent-filter>
</receiver>

<meta-data
        android:name="com.vivo.push.api_key"
        android:value="您在 VIVO 推送官网申请的 APIKEY" />

<meta-data
        android:name="com.vivo.push.app_id"
        android:value="您在 VIVO 推送官网申请的 APPID" />
<!-- VIVO 代理推送必需组件  END -->
```

请确保 AndroidManifest.xml 中声明了必须的组件以及组件的属性，否则 Push 服务部分功能可能无法正常工作。如需开启厂商代理，请声明对应厂商的推送必需组件。

5.2 调用 API

下面介绍如何调用 SDK 中已封装的 API 完成各项操作：

1. 在主 Activiy 的 OnCreate 方法中，调用接口 startWork，其中 loginValue 是 apiKey。（注意：不要在 Application 的 onCreate 里去做 startWork 的操作，否则可能造成应用循环重启的问题，将严重影响应用的功能和性能。）

```
// 开启华为代理，如需开启，请参考华为代理接入文档
PushManager.enableHuaweiProxy(this, true);
// 开启魅族代理，如需开启，请参考魅族代理接入文档
PushManager.enableMeizuProxy(this, true, "mzAppid", "mzAppKey");
// 开启 OPPO 代理，如需开启，请参考 OPPO 代理接入文档
PushManager.enableOppoProxy(this, true, "oppoAppKey", "oppoAppSecret");
// 开启小米代理，如需开启，请参考小米代理接入文档
PushManager.enableXiaomiProxy(this, true, "xmAppId", "xmAppKey");
// 开启 VIVO 代理，如需开启，请参考 VIVO 代理接入文档
PushManager.enableVivoProxy(this, true);
PushManager.startWork(context, loginType, loginValue)
```

如需开启厂商代理，请调用对应厂商的开关方法，并将必要参数替换为您在厂商推送官网申请的值，最后将“API Key“手动修改为指定应用的 API Key。接入厂商代理推送请参考厂商接入文档

2. 自定义通知样式（可选）

这是通知推送的高级功能，对于很多应用来说，使用系统默认的通知栏样式就足够了。

SDK 提供了 2 个用于定制通知栏样式的构建类（PushNotificationBuilder 是两者的基类）：

- BasicPushNotificationBuilder

  用于定制 Android Notification 里的 defaults / flags / icon 等基础样式（行为）。

- CustomPushNotificationBuilder

  除了让开发者定制 BasicPushNotificationBuilder 中的基础样式以外，该类可以进一步定制 Notification Layout。

当开发者需要为不同的通知指定不同的通知栏样式（行为）时，则需要调用 ***PushManager.setNotificationBuilder*** 设置多个通知栏构建类。

设置时，开发者自己维护 notificationBuilderId 这个编号，下发通知时使用 notification_builder_id 指定该编号，从而 SDK 会调用开发者应用程序里设置过的指定编号的通知栏构建类，来定制通知栏样式。如果通过管理控制台来推送通知，请在*高级设置* 的 *自定义样式* 栏中指定编号。

这里以 CustomPushNotificationBuilder 为例，代码如下：

```
CustomPushNotificationBuilder cBuilder = new CustomPushNotificationBuilder(layoutId,
        layoutIconId, layoutTitleId, layoutTextId);
cBuilder.setNotificationFlags(Notification.FLAG_AUTO_CANCEL);
cBuilder.setNotificationDefaults(Notification.DEFAULT_SOUND
        |Notification.DEFAULT_VIBRATE);
cBuilder.setStatusbarIcon(statusbarIconId);
cBuilder.setLayoutDrawable(notificationIconId);
cBuilder.setNotificationSound(notificationSoundId);
PushManager.setNotificationBuilder(this, notificationBuilderId, cBuilder);
```

此外，从 SDK 5.8.0 版本开始，提供关于 Android O（8.x）新特性---通知渠道的设置接口。若您的应用需要适配 Android O（8.x）系统，且将目标版本 targetSdkVersion 设置为 26 及以上时，可自定义系统所必需的 channelId 和 channelName 字段，若不指定，SDK 将使用渠道名默认值"云推送"。对于 BasicPushNotificationBuilder，所有的通知消息均使用同一个 channel；对于 CustomPushNotificationBuilder，可根据自定义的 notification_builder_id 编号，来指定不同 channel。调用方法示例如下：

```
cBuilder.setChannelId("your custom ID");
cBuilder.setChannelName("your custom Name");
```

设置过的 channel 都会显示在系统的应用通知列表中，您可以根据需要调用 Android O 系统 API 删除 SDK 默认的 channelId（“com.baidu.android.pushservice.push”）或自定义的 channelId。调用方法示例如下：

```
NotificationManager mNotificationManager =
        (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
// The id of the channel.
String id = "com.baidu.android.pushservice.push";
mNotificationManager.deleteNotificationChannel(id);
```

3. 客户端程序需要自己实现一个 Receiver 来接收 Push 消息、接口调用回调以及通知点击事件，也就是上文提到的 AndrodManifest.xml 中注册的 receiver，**该 Receiver 需要继承 PushMessageReceiver，示例如下：**

```
public class MyPushMessageReceiver extends PushMessageReceiver {
/** TAG to Log */
public static final String TAG = MyPushMessageReceiver.class.getSimpleName();

/**
 * 调用 PushManager.startWork 后，sdk 将对 push server 发起绑定请求，这个过程是异步的。绑定请求的结果通过 onBind 返回。
 */
@Override
public void onBind(Context context, int errorCode, String appid,
                    String userId, String channelId, String requestId) {
        String responseString = "onBind errorCode=" + errorCode + " appid="
                    + appid + " userId=" + userId + " channelId=" + channelId
                    + " requestId=" + requestId;
}

/**
 * 接收透传消息的函数。
 */
@Override
public void onMessage(Context context, String message, String customContentString) {
        String messageString = "透传消息  message=" + message + " customContentString="
                    + customContentString;
        Log.d(TAG, messageString);

        // 自定义内容获取方式，mykey 和 myvalue 对应透传消息推送时自定义内容中设置的键和值
        if (customContentString != null & customContentString != "") {
            JSONObject customJson = null;
            try {
                customJson = new JSONObject(customContentString);
                String myvalue = null;
                if (!customJson.isNull("mykey")) {
                    myvalue = customJson.getString("mykey");
                }
```

```
                } catch (JSONException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
            }
        }

    /**
         * 接收通知点击的函数。
         */
        @Override
        public void onNotificationClicked(Context context, String title,
                String description, String customContentString) {
            String notifyString = "通知点击  title=" + title + " description="
                    + description + " customContent=" + customContentString;
            Log.d(TAG, notifyString);

            // 自定义内容获取方式，mykey 和 myvalue 对应通知推送时自定义内容中设置的
键和值
            if (customContentString != null & customContentString != "") {
                JSONObject customJson = null;
                try {
                    customJson = new JSONObject(customContentString);
                    String myvalue = null;
                    if (!customJson.isNull("mykey")) {
                        myvalue = customJson.getString("mykey");
                    }
                } catch (JSONException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }

        /**
         * 接收通知到达的函数。
         * @param context  上下文
```

```
         * @param title  推送的通知的标题
         * @param description  推送的通知的描述
         * @param customContentString  自定义内容，为空或者 json 字符串
         */
    @Override
        public void onNotificationArrived (Context context, String title,
                                String description, String customContentString) {
            String notifyString = "通知到达  title=" + title + " description="
                                + description + " customContent=" + customContentString;
            Log.d(TAG, notifyString);

            // 自定义内容获取方式，mykey 和 myvalue 对应通知推送时自定义内容中设置的
键和值
            if (customContentString != null & customContentString != "") {
                JSONObject customJson = null;
                try {
                    customJson = new JSONObject(customContentString);
                    String myvalue = null;
                    if (!customJson.isNull("mykey")) {
                        myvalue = customJson.getString("mykey");
                    }
                } catch (JSONException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }

        /**
         * setTags()  的回调函数。
         */
        @Override
        public void onSetTags(Context context, int errorCode,
                                List<String> sucessTags, List<String> failTags, String requestId) {
            String responseString = "onSetTags errorCode=" + errorCode + " sucessTags="
```

```
                                        + sucessTags + " failTags=" + failTags + " requestId="
                                + requestId;
        }

    /**
         * delTags()  的回调函数。
         */
        @Override
        public void onDelTags(Context context, int errorCode,
                                List<String> sucessTags, List<String> failTags, String requestId) {
                String responseString = "onDelTags errorCode=" + errorCode + " sucessTags="
                                + sucessTags + " failTags=" + failTags + " requestId="
                                + requestId;
        }

        /**
         * listTags()  的回调函数。
         */
        @Override
        public void onListTags(Context context, int errorCode,
                                List<String> tags, String requestId) {
                String responseString = "onListTags errorCode=" + errorCode + " tags=" + tags;
        }

        /**
         * PushManager.stopWork()  的回调函数。
         */
        @Override
        public void onUnbind(Context context, int errorCode, String requestId) {
                String responseString = "onUnbind errorCode=" + errorCode
                                + " requestId = " + requestId;
        }

    }
```

5.3 错误码说明

error_code	描述
0	绑定成功
10001	当前网络不可用，请检查网络
10002	服务不可用，连接 server 失败
10003	服务不可用，503 错误
10101	应用集成方式错误，请检查各项声明和权限定义
20001	未知错误
30600	服务端内部出现错误
30601	非法函数请求，请检查您的请求内容
30602	请求参数错误，请检查您的参数
30603	非法构造请求，服务端验证失败
30605	请求的数据在服务端不存在
30608	绑定关系不存在或未找到

5.4 混淆打包说明

如果开发者需要混淆自己的 APK，请在混淆文件（一般默认为 Android 工程下 proguard-project.txt 或者 proguard.cfg）中添加如下说明(VERSION 为版本名称，pushservice-VERSION.jar 为集成的 jar 包名字)：

```
-libraryjars libs/pushservice-VERSION.jar
-dontwarn com.baidu.**
-keep class com.baidu.**{*; }
# 小米代理推送
-dontwarn com.xiaomi.**
-dontwarn org.apache.thrift.**
-keep class com.xiaomi.**{*; }
-keep class org.apache.thrift.**{*; }

# 魅族代理推送
-dontwarn com.meizu.cloud.**
-keep class com.meizu.cloud.**{*; }

# OPPO 代理推送
-dontwarn com.heytap.mcssdk.**
-dontwarn com.mcs.aidl.**
-keep class com.heytap.mcssdk.**{*; }
-keep class com.mcs.aidl.**{*; }

# VIVO 代理推送
-dontwarn com.vivo.push.**
-dontwarn com.vivo.vms.**
-keep class com.vivo.push.**{*; }
-keep class com.vivo.vms.**{*; }

# 华为代理推送
-ignorewarnings
-keepattributes *Annotation*
-keepattributes Exceptions
-keepattributes InnerClasses
-keepattributes Signature
-keepattributes SourceFile,LineNumberTable
-keep class com.hianalytics.android.**{*;}
-keep class com.huawei.updatesdk.**{*;}
-keep class com.huawei.hms.**{*;}
```

第6章 API 说明

本文档提供开放接口的 API 说明。

6.1 类

本 SDK 中有四个重要的开放类，分别为：PushManager、PushSettings、BasicPushNotificationBuilder 和 CustomPushNotificationBuilder，还有常量类 PushConstants。

类	描述
PushManager	PushManager 提供了所有使用 Push 服务的静态方法
BasicPushNotificationBuilder	用于定制 Android Notification 里的 defaults / flags / icon 等基础样式（行为）
CustomPushNotificationBuilder	用于定制 Android Notification 里的 defaults / flags / icon，以及通知栏的 layout、图标和状态栏图标
PushSettings	PushSettings 提供了端上 Push 服务的配置静态方法
PushConstants	SDK 对外的常量定义
PushMessageReceiver	Push 消息处理 Receiver

6.2 API

本 SDK 目前支持以下接口：

分类	功能	API 函数原型
Push 服务接口	提供 Push 服务	enableHuaweiProxy, enableXiaomiProxy, enableMeizuProxy, enableOppoProxy, enableVivoProxy, startWork, stopWork, resumeWork, setJobSchedulerId
Tag 管理接口	Tag 的创建与删除	setTags, delTags, listTags
通知管理接口	自定义通知样式	CustomPushNotificationBuilder, BasicPushNotificationBuilder setNotificationFlags, setNotificationDefaults, setStatusbarIcon, setLayoutDrawable, setNotificationSound, setNotificationBuilder, setChannelId, setChannelName
设置接口	Push 服务设置	enableDebugMode
异步消息处理接口	Push 消息处理 receiver	onBind, onMessage, onNotificationClicked, onNotificationArrived, onSetTags, onDelTags, onListTags, onUnbind

6.2.1 Push 是否启用华为代理模式 -- enableHuaweiProxy

- 函数原型

```
public static void enableHuaweiProxy(Context context, boolean proxyEnable);
```

- 功能

  PushManager 类定义的静态方法，是否启用华为代理模式。

- 参数

  context：当前执行 Context

  proxyEnable：是否开启华为代理模式，默认关闭

- 返回结果

  无

6.2.2 Push 是否启用小米代理模式 -- enableXiaomiProxy

- 函数原型

```
public static void enableXiaomiProxy(Context context,boolean proxyEnable, String appId,
String appKey);
```

- 功能

  PushManager 类定义的静态方法，是否启用小米代理模式。

- 参数

  context：当前执行 Context

  proxyEnable：是否开启小米代理模式，默认关闭

  appId：应用在小米推送平台的 appId

  appKey：应用在小米推送平台的 appKey

- 返回结果

  无

6.2.3 Push 是否启用魅族代理模式 -- enableMeizuProxy

- 函数原型

```
public static void enableMeizuProxy(Context context, boolean proxyEnable, String appId,
String appKey);
```

- 功能

  PushManager 类定义的静态方法，是否启用魅族代理模式。

- 参数

  context：当前执行 Context

  proxyEnable：是否开启魅族代理模式，默认关闭

  appId：应用在魅族推送平台的 appId

appKey：应用在魅族推送平台的 appKey

- 返回结果

  无

6.2.4 Push 是否启用 OPPO 代理模式 -- enableOppoProxy

- 函数原型

```
public static void enableOppoProxy(Context context, boolean proxyEnable, String appKey, String appSecret);
```

- 功能

  PushManager 类定义的静态方法，是否启用 OPPO 代理模式。

- 参数

  context：当前执行 Context

  proxyEnable：是否开启 OPPO 代理模式，默认关闭

  appKey：应用在 OPPO 推送平台的 appKey

  appSecret：应用在 OPPO 推送平台的 appSecret

- 返回结果

  无

6.2.5 Push 是否启用 VIVO 代理模式 -- enableVivoProxy

- 函数原型

```
public static void enableVivoProxy(Context context, boolean proxyEnable);
```

- 功能

  PushManager 类定义的静态方法，是否启用 VIVO 代理模式。

- 参数

  context：当前执行 Context

  proxyEnable：是否开启 VIVO 代理模式，默认关闭

- 返回结果

  无

6.2.6 Push 服务初始化及绑定-- startWork

- 函数原型

```
public static void startWork(Context context, int loginType, String loginValue)
```

- 功能

  PushManager 类定义的静态方法，完成 Push 服务的初始化， 并且自动完成 bind 工作。

- 参数

context：当前执行 Context

loginType：PushConstants.LOGIN_TYPE_API_KEY

loginValue：API Key

- 返回结果

通过自定义的 Receiver 类里 onBind 方法返回结果，详见 6.2.26 节 onBind

6.2.7 停止和恢复 Push 服务-- stopWork、resumeWork

- 函数原型

```
public static void stopWork(Context  context)
```

- 功能

PushManager 类定义的静态方法，停止本应用 Push 服务进程，并且完成 unbind 工作。startWork 和 resumeWork 都会重新开启本应用 Push 功能。

- 参数

context：当前执行 Context

- 返回结果

通过自定义的 Receiver 类里 onUnbind 方法返回结果，详见 6.2.33 节 onUnbind

- 函数原型

```
public static void resumeWork(Context  context)
```

- 功能

PushManager 类定义的静态方法，恢复本应用 Push 服务， 并且再次完成 bind 工作。

- 参数

context：当前执行 Context

- 返回结果

通过自定义的 Receiver 类里 onBind 方法返回结果，详见 6.2.26 节 onBind

6.2.8 设置 PushJobService 的标识符 -- setJobSchedulerId

- 函数原型

```
public static void setJobSchedulerId(Context context, int id)
```

- 功能

PushManager 类定义的静态方法，设置 PushJobService 的 jobId。JobScheduler 在调度 PushJobService 时默认 jobId 为 1，可以通过此方法修改 jobId 默认值。

- 参数

context：当前执行 Context

id：PushJobService 任务的 jobId，默认为 1

- 返回结果

  无

6.2.9 查询 push 是否被停止的接口-- isPushEnabled

- 函数原型

```
public static boolean isPushEnabled(Context context)
```

- 功能

  PushManager 类定义的静态方法，查询 push 是否已经被停止。

- 参数

  context：当前执行 Context

- 返回结果

  true：已开启 push 服务

  false：未开启 push 服务

6.2.10 设置免打扰时段-- setNoDisturbMode

- 函数原型

```
PushManager.setNoDisturbMode(Context context, int startHour, int startMinute, int endHour, int endMinute)
```

- 功能

  PushManager 类定义的静态方法，设置免打扰模式的具体时段，该时间内处于免打扰模式，通知到达时去除通知的提示音、振动以及提示灯闪烁。

  注意：如果开始时间小于结束时间，免打扰时段为当天的起始时间到结束时间；如果开始时间大于结束时间，免打扰时段为第一天起始时间到第二天结束时间；如果开始时间和结束时间的设置均为 00:00 时，取消免打扰时段功能。如果没有调用此接口，默认无免打扰时段。

- 参数

  context：当前执行 Context

  startHour，startMinute：起始时间，24 小时制，取值范围 0~23，0~59

  endHour，endMinute：结束时间，24 小时制，取值范围 0~23，0~59

- 返回结果

  无

6.2.11 设置 Tag-- setTags

- 函数原型

```
public static void setTags(Context  context,  List<String>  tags)
```

- 功能

PushManager 类定义的静态方法，用于设置标签；成功设置后，可以从管理控制台或您的服务后台，向指定的设置了该 tag 的一群用户进行推送。

注意：tag 设置的前提是已绑定的端，也就是应用有运行过 startWork 或 bind，且在 onBind 回调中返回成功。

- 返回结果

  通过自定义的 Receiver 类里 onSetTags 方法返回结果，详见 6.2.30 节 onSetTags

6.2.12 删除 Tag-- delTags

- 函数原型

```
public static void delTags(Context  context,  List<String>  tags)
```

- 功能

  PushManager 类定义的静态方法，用于删除标签。

- 返回结果

  通过自定义的 Receiver 类里 onDelTags 方法返回结果，详见 6.2.31 节 onDelTags

6.2.13 列举 Tag-- listTags

- 函数原型

```
public static void listTags(Context  context)
```

- 功能

  PushManager 类定义的静态方法，用于列出本机绑定的标签。

- 返回结果

  通过自定义的 Receiver 类里 onListTags 方法返回结果，详见 6.2.32 节 onListTags

6.2.14 设置通知的 Builder -- setNotificationBuilder

- 函数原型

```
public static void setNotificationBuilder(Context  context,  int  id,
                                          PushNotificationBuilder  notificationBuilder)
```

- 功能

PushManager 类定义的静态方法，设置通知栏样式，并为样式指定编号。在管理控制台或您的服务后台中，您可以指定相应的编号，让客户端显示预先设定好的样式。

- 参数

  context ： android app 运行上下文

  id ： notificationBuilder 编号，开发者自己维护

  notificationBuilde ： 通知栏构建类

6.2.15 设置默认的通知 Builder-- setDefaultNotificationBuilder

- 函数原型

```
public static void setDefaultNotificationBuilder(Context  context,
                                                  PushNotificationBuilder  notificationBuilder)
```

- 功能

PushManager 类定义的静态方法，设置默认的通知栏样式；如果推送通知时不指定指定 id 的样式，都将显示该默认样式。

- 参数

Context ： android app 运行上下文
notificationBuilder ： 通知栏构建类

6.2.16 自定义通知 Builder-- BasicPushNotificationBuilder

- 函数原型

```
BasicPushNotificationBuilder ()
```

- 功能

自定义通知状态栏构建类构造函数(定制通知栏基础样式) 。

6.2.17 自定义通知 Builder-- CustomPushNotificationBuilder

- 函数原型

```
CustomPushNotificationBuilder(layoutId,  layoutIconId,  layoutTitleId,  layoutTextId)
```

- 功能

自定义通知状态栏构建类构造函数(定制通知栏基础样式及 layout)。

- 参数

layoutId ： 自定义 layout 资源 id
layoutIconId ： 自定义 layout 中显示 icon 的 view id
layoutTitleId ： 自定义 layout 中显示标题的 view id
layoutTextId ： 自定义 layout 中显示内容的 view id

6.2.18 设置通知 flags-- setNotificationFlags

- 函数原型

```
public void setNotificationFlags (int  flags)
```

- 功能

基类 PushNotificationBuilder 定义的方法，定制 Android Notification 里的 flags。

- 参数

flags ： Android Notification flags

6.2.19 设置通知 defaults-- setNotificationDefaults

- 函数原型

```
public void setNotificationDefaults (int    defaults)
```

- 功能

基类 PushNotificationBuilder 定义的方法，定制 Android Notification 里的 defaults。

- 参数

defaults ： Android Notification defaults

6.2.20 设置通知状态栏 icon-- setStatusbarIcon

- 函数原型

```
public void setStatusbarIcon (int    icon)
```

- 功能

基类 PushNotificationBuilder 类定义的方法，定制 Android Notification 通知状态栏的 icon 图标。

- 参数

icon：图标资源 id

6.2.21 设置通知样式图片-- setLayoutDrawable

- 函数原型

```
public void setLayoutDrawable(int    drawableId)
```

- 功能

CustomPushNotificationBuilder 类定义的方法，定制自定义 layout 中显示的图片。

- 参数

drawableId：图标资源 id

6.2.22 设置通知声音-- setNotificationSound

- 函数原型

```
public void setNotificationSound (String    soundId)
```

- 功能

CustomPushNotificationBuilder 类定义的方法，自定义推送的声音。

- 参数

soundId：声音资源路径

6.2.23 设置通知渠道 ID-- setChannelId

- 函数原型

```
public void setChannelId(String channelId)
```

- 功能

基类 PushNotificationBuilder 类定义的方法，在应用 targetSdkVersion 大于等于 26 时，自定义通知渠道 ID。（SDK 5.8.0 新增 API）

- 参数

  channelId：通知渠道 ID

6.2.24 设置通知渠道名-- setChannelName

- 函数原型

```
public void setChannelName(String channelName)
```

- 功能

基类 PushNotificationBuilder 类定义的方法，在应用 targetSdkVersion 大于等于 26 时，自定义通知渠道名。（SDK 5.8.0 新增 API）

- 参数

  channelName：通知渠道名

6.2.25 开启调试模式-- enableDebugMode

- 函数原型

```
public static void enableDebugMode(boolean    debugEnabled)
```

- 功能

PushSettings 类定义的方法，开启调试模式，会输出调试 Log。注：发布应用时，请去除开启调试模式的调用，以免降低 Push 的性能。

6.2.26 获取绑定请求的结果-- onBind

- 函数原型

```
public void onBind(Context context, int errorCode, String appid,
                                    String userId, String channelId, String requestId)
```

- 功能

PushMessageReceiver 的抽象方法，把 receiver 类继承 PushMessageReceiver 可以使用。调用 PushManager.startWork 后，sdk 将对 push server 发起绑定请求，这个过程是异步的。绑定请求的结果通过 onBind 返回。

如果您需要用单播推送，需要把这里获取的 channel id 上传到应用 server 中，再调用 server 接口，用 channel id 给单个手机或者用户推送。

- 参数

  context　BroadcastReceiver 的执行 Context

  errorCode 绑定接口返回值，0 - 成功

  appid　　应用 id。errorCode 非 0 时为 null

userId 应用 user id。errorCode 非 0 时为 null

channelId 应用 channel id。errorCode 非 0 时为 null

requestId 向服务端发起的请求 id。在追查问题时有用；

6.2.27 接收透传消息的函数-- onMessage

- 函数原型

```
public void onMessage(Context context, String message, String customContentString)
```

- 功能

PushMessageReceiver 的抽象方法，把 receiver 类继承 PushMessageReceiver 可以使用。接收透传消息。

- 参数

context 上下文

message 推送的消息

customContentString 自定义内容,为空或者 json 字符串

6.2.28 接收通知点击的函数-- onNotificationClicked

- 函数原型

```
public void onNotificationClicked(Context context, String title,
                                  String description, String customContentString)
```

- 功能

PushMessageReceiver 的抽象方法，把 receiver 类继承 PushMessageReceiver 可以使用。接收通知点击的函数。

- 参数

context 上下文

title 推送的通知的标题

description 推送的通知的描述

customContentString 自定义内容，为空或者 json 字符串

6.2.29 接收通知到达的函数-- onNotificationArrived

- 函数原型

```
public void onNotificationArrived(Context context, String title,
                                  String description, String customContentString)
```

- 功能

PushMessageReceiver 的抽象方法，把 receiver 类继承 PushMessageReceiver 可以使用。接收通知到达的函数。

- 参数

context 上下文

title 推送的通知的标题

description 推送的通知的描述

customContentString 自定义内容，为空或者 json 字符串

6.2.30 setTags 的回调函数-- onSetTags

- 函数原型

```
public void onSetTags(Context context, int errorCode,
                      List<String> sucessTags, List<String> failTags, String requestId)
```

- 功能

PushMessageReceiver 的抽象方法，把 receiver 类继承 PushMessageReceiver 可以使用。setTags() 的回调函数。

- 参数

context 上下文

errorCode 错误码。0 表示某些 tag 已经设置成功；非 0 表示所有 tag 的设置均失败。

successTags 设置成功的 tag

failTags 设置失败的 tag

requestId 分配给对云推送的请求的 id

6.2.31 delTags 的回调函数-- onDelTags

- 函数原型

```
public void onDelTags(Context context, int errorCode,
                      List<String> sucessTags, List<String> failTags, String requestId)
```

- 功能

PushMessageReceiver 的抽象方法，把 receiver 类继承 PushMessageReceiver 可以使用。delTags() 的回调函数。

- 参数

context 上下文

errorCode 错误码。0 表示某些 tag 已经删除成功；非 0 表示所有 tag 均删除失败。

successTags 成功删除的 tag

failTags 删除失败的 tag

requestId 分配给对云推送的请求的 id

6.2.32 listTags 的回调函数-- onListTags

- 函数原型

```
public void onListTags(Context context, int errorCode,
                       List<String> tags, String requestId)
```

- 功能

PushMessageReceiver 的抽象方法，把 receiver 类继承 PushMessageReceiver 可以使用。listTags() 的回调函数。

- 参数

context 上下文

errorCode 错误码。0 表示列举 tag 成功；非 0 表示失败。

tags 当前应用设置的所有 tag。

requestId 分配给对云推送的请求的 id

6.2.33 stopWork 的回调函数-- onUnbind

- 函数原型

```
public void onUnbind(Context context, int errorCode, String requestId)
```

- 功能

PushMessageReceiver 的抽象方法，把 receiver 类继承 PushMessageReceiver 可以使用。PushManager.stopWork() 的回调函数。

- 参数

context 上下文

errorCode 错误码。0 表示从云推送解绑定成功；非 0 表示失败。

requestId 分配给对云推送的请求的 id

6.3 常量说明

Android SDK 的常量定义都在 PushConstants 类中。如下：

✧ LOGIN_TYPE_API_KEY

Push 服务初始化及绑定时标识绑定认证方式（无账号认证方式）。

第7章 联系我们

邮箱：push-support@baidu.com

问题反馈：http://push.baidu.com/issue/list/hot

第8章 缩略语

缩略语	英文全称	说明
SDK	Software Development Kit	软件开发工具包。