Tencent Push Notification Service

Android Access

Product Documentation

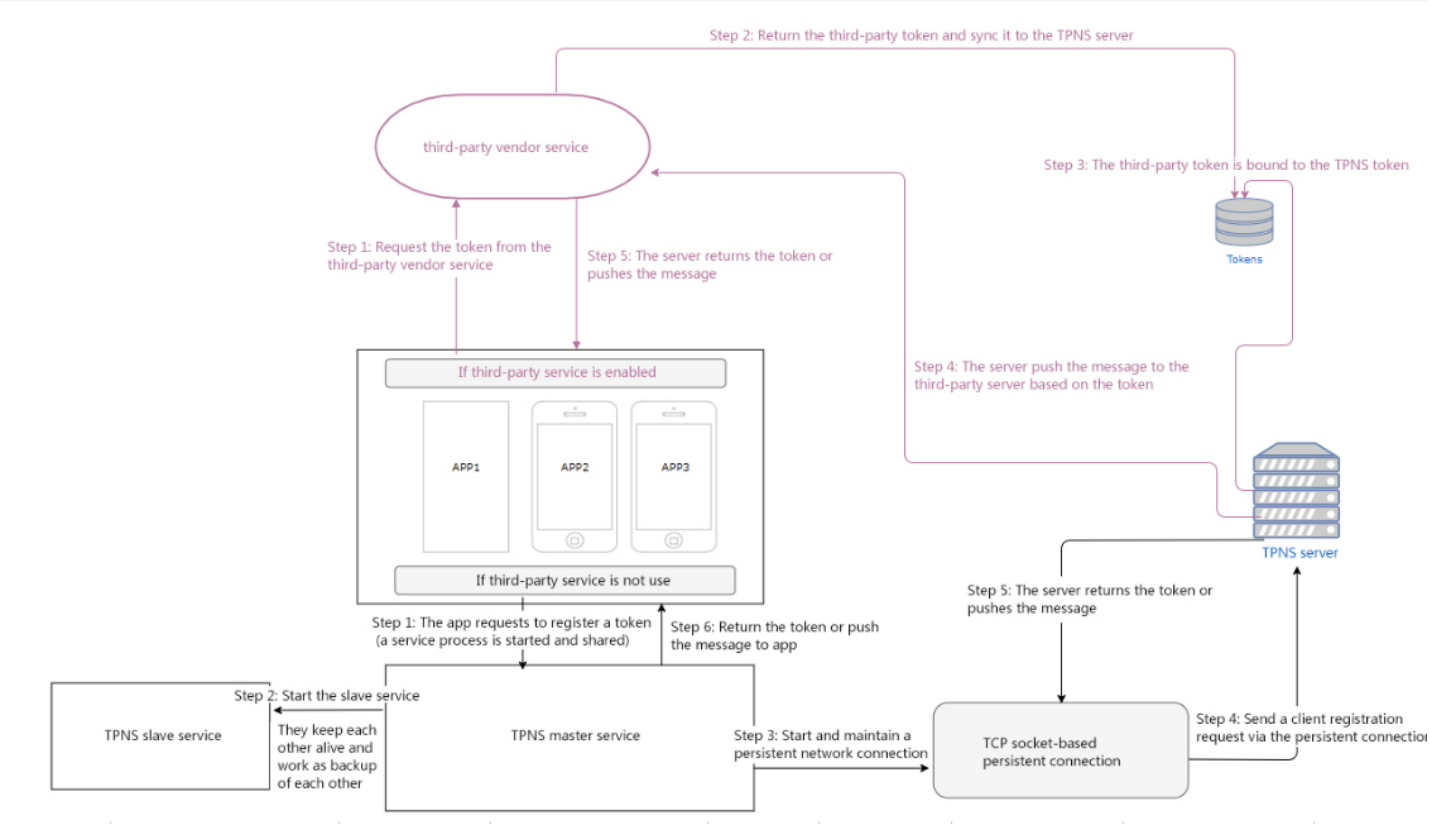
Contents

Android Access
Android Push Service Overview

Last updated：2019-06-26 09:53:57

TPNS is a professional mobile app push platform that can deliver tens of billions of notifications/message in a matter of seconds. It now fully supports both Android and iOS systems. It can be used easily through the embedded SDK, API calls, and web-based visual console to push to specific users, greatly improving user activity and effectively waking up inactive users. In addition, it features display of real-time push effect data.

How TPNS Works



Below are the steps for the Android client to implement the push process (with no vendor-specific channels):

- When the client app starts, it will start a TPNS master service, which is globally unique and shared on one device.

- The TPNS master service randomly starts a slave service in the app accessing TPNS, both of which keep each other alive and work as the backup of each other.
- The TPNS master service establishes a socket-based persistent connection with the TPNS server and maintains the connection through heartbeat and other mechanisms.
- The master service on the client requests a token from the TPNS server via the socket-based persistent connection.
- The TPNS server pushes the message to the master service on the client via the socket-based persistent connection.
- The master service forwards the push message to the corresponding client app.

Below are the steps for the Android client to implement the push process (with vendor-specific channels):

- Send the registration request to the third-party vendor service to request the token.
- Save the third-party token and sync it to the TPNS server.
- Map the third-party token to the TPNS token and save the mapping.
- The TPNS server calls the third-party push API to push the message to the third-party server based on the token mapping.
- The third-party server pushes the message to the client app.

Overview of Main Features

The Android SDK contains the APIs provided by TPNS for message push implementation.
It is mainly responsible for:

- Providing two types of push (notification and message) for easy use;
- The binding of account, tag, and device, so that you can implement message pushes to specific user groups for diversified push methods;
- Reporting taps, i.e., the number of times the messages are tapped by users.
- Providing two types of push (notification and message) for easy use;

- Providing multi-vendor channel integration for easy integration with multi-vendor push services.

Android SDK Integration Guide

Last updated：2019-06-26 09:54:06

Automated Integration via Android Studio

Importing Dependencies

In Android Studio, you can automatically access TPNS using jcenter remote repository, without having to import jar packages or so files to the project.
There is no need to configure TPNS-related content in AndroidManifest.xml as jcenter will automatically import.
After the dependencies are imported, modify the app configuration and write the registration code to achieve fast access to TPNS.
The corresponding dependencies are all the latest version at the official website.
For the user-defined receiver, you need to configure related nodes in Androidmanifest.xml.

Configure the following in the `app's build.gradle` file.

```
android {
......
defaultConfig {

// The package name registered at TPNS' official website. Note that the application ID, the current ap
p package name, and the package name of the app registered at TPNS' official website must be the s
ame.
applicationId "your package name"
......

ndk {
// Choose and add the .so libraries corresponding to the cpu type as needed.
abiFilters 'armeabi', 'armeabi-v7a', 'arm64-v8a'
// You can also add 'x86', 'x86_64', 'mips', and 'mips64'.
}

manifestPlaceholders = [

XG_ACCESS_ID:"accessid of the registered app",
XG_ACCESS_KEY : "accesskey of the registered app",
]
......
```

```
}
......
}

dependencies {
......

// jar of the TPNS general version with no vendor-specific channels.
implementation 'com.tencent.xinge:xinge:4.3.2-release'
// jg package
implementation'com.tencent.jg:jg:1.1'
// wup package
implementation 'com.tencent.wup:wup:1.0.0.E-Release'
// mid package, minSdkVersion 14
implementation 'com.tencent.mid:mid:4.0.7-Release'

}
```

Note:

- If Android Studio prompts the following after you add the abiFilter configuration above:

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

Please add the following to the gradle.properties file in the project's root directory:

android.useDeprecatedNdk=true

- If you need to listen to messages, please see the XGBaseReceiver API or the MessageReceiver class of the demo. Inherit XGBaseReceiver and configure the following in the configuration file:

```
<receiver android:name="Complete class name such as: com.qq.xgdemo.receiver.MessageReceiver"
android:exported="true" >
<intent-filter>
<!-- Receive message passthrough -->
<action android:name="com.tencent.android.tpush.action.PUSH_MESSAGE" />
<!-- Listen to handling results such as registration, unregistration, tag setting/deletion, and notificati
on tap ->
<action android:name="com.tencent.android.tpush.action.FEEDBACK" />
</intent-filter>
</receiver>
```

- Versions above 4.X are already compatible with Android P. HTTPS is used by default. If you want to use HTTP, you need to configure it by yourself (Click here to view the configuration method.

Manual Configuration for Integration

Registering and Downloading the SDK

Visit the TPNS console at xg.qq.com, log in with your QQ account number, go to the app registration page, enter the "App name" and "App package name" (which must be the same with the app), select "Operating system" and "Category", and click "Create app".

After the app is created successfully, click "App configuration" to view the app-specific information such as AccessId and AccessKey.

After registration is completed, download the latest version of the Android SDK to your local system and unzip it.

Project Configuration

The steps to import the SDK into the project are as follows:

(1) Create or open an Android project (for more information about how to create an Android project, see the developing environment section).

(2) Copy all the files in the libs directory under the TPNS SDK directory to the libs (or lib) directory of the project.

(3) Select the TPNS jar packages in the libs (or lib) directory, right-click them and select Build Path, and select Add to Build Path to add the SDK to the reference directory of the project.
(4).so files are necessary components of TPNS and support armeabi, armeabi-v7a, misp and x86 platforms. Please add the appropriate .so files according to the platform currently supported.

a) If your project does not use other .so files, it is recommended to copy the four platform directories to your project;

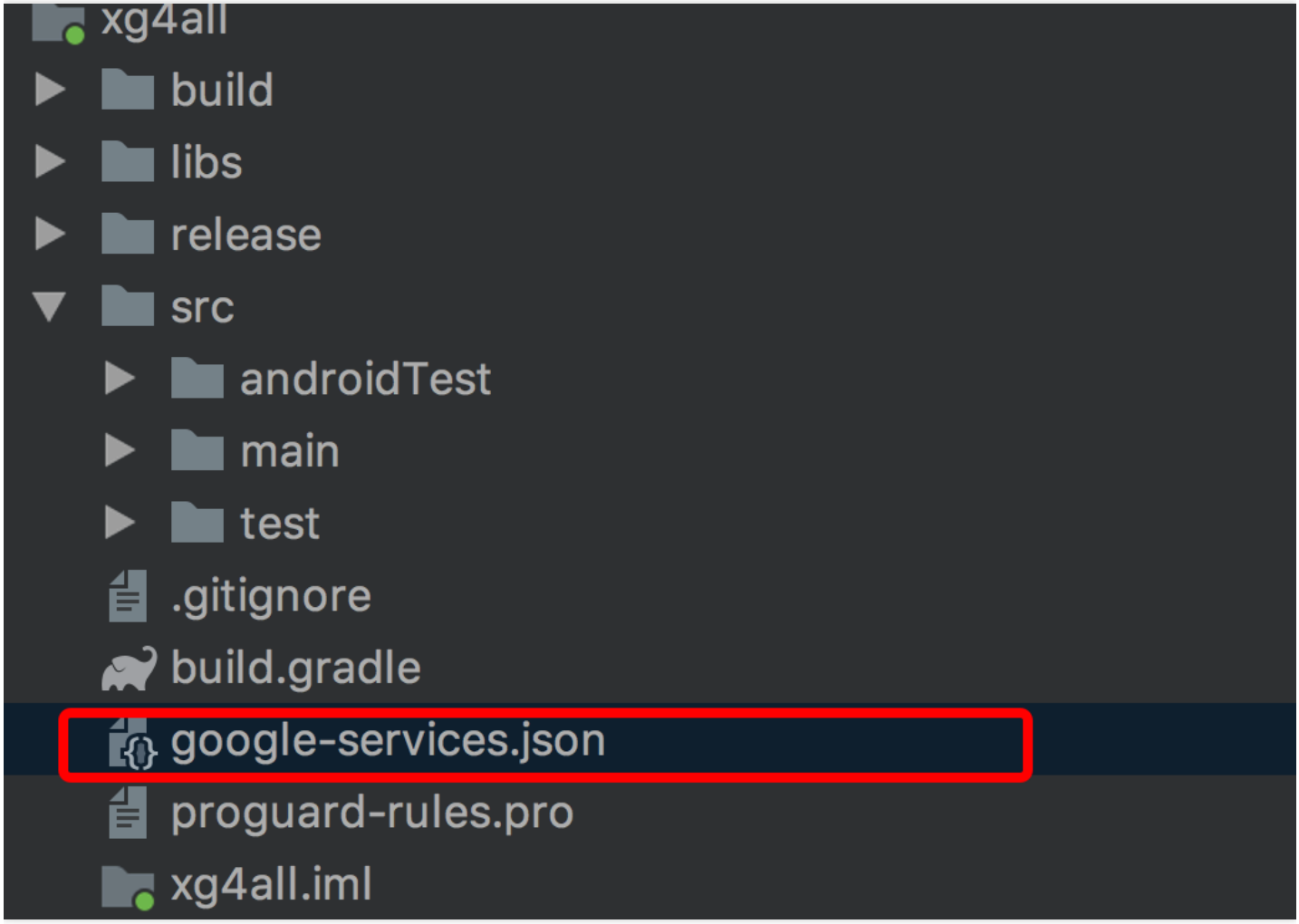
b) If there are already .so files, you only need to copy the files in the corresponding directory of TPNS;

c) If the app is a game with MSDK access, usually only the .so files in the armeabi directory are needed;

d) If the current project already has armeabi, then you only need to add the .so files in the armeabi directory of TPNS but not other directories. In other similar situations,
only add the ones that exist on the current platform;

e) If an error (10004.SOERROR) occurs when you import the .so files to Android Studio, add the file named jniLibs in the main file directory.

Then, copy all the schema files into it, i.e., all the folders under Other-Platform-SO in the SDK directory. See the figure below:



(5) Open Androidmanifest.xml, add the following configuration (it is recommended to see the demo in the downloaded package for modification), where YOUR_ACCESS_ID and YOUR_ACCESS_KEY should be replaced with the accessId and accessKey of the app. Please ensure that the configuration is completed as required; otherwise, the service may fail.

```
<application
<!-- TPNS broadcast receiver, which is **required** -->
<receiver android:name="com.tencent.android.tpush.XGPushReceiver"
android:process=":xg_service_v4" >
<intent-filter android:priority="0x7fffffff" >
<!-- **(Required)** The internal broadcast of the TPNS SDK -->
<action android:name="com.tencent.android.tpush.action.SDK" />
<action android:name="com.tencent.android.tpush.action.INTERNAL_PUSH_MESSAGE" />
<!-- **(Required)** System broadcast: splash screen and network switch -->
<action android:name="android.intent.action.USER_PRESENT" />
```

```
<action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
<!-- **(Optional)** Some commonly used system broadcasts, which enhance the chance of restart of t
he TPNS service. Please choose as needed. You can also add some custom broadcasts of the app to st
art the service -->
<action android:name="android.bluetooth.adapter.action.STATE_CHANGED" />
<action android:name="android.intent.action.ACTION_POWER_CONNECTED" />
<action android:name="android.intent.action.ACTION_POWER_DISCONNECTED" />
</intent-filter>
</receiver>

<!-- **(Optional)** The receiver implemented by the app, which is used to receive the message passth
rough and call back the operation result. Please add as needed -->
<!-- YOUR_PACKAGE_PATH.CustomPushReceiver should be changed to your own receiver: -->
<receiver android:name="com.qq.xgdemo.receiver.MessageReceiver"
android:exported="true" >
<intent-filter>
<!-- Receive message passthrough -->
<action android:name="com.tencent.android.tpush.action.PUSH_MESSAGE" />
<!-- Listen to handling results such as registration, unregistration, tag setting/deletion, and notificati
on tap ->
<action android:name="com.tencent.android.tpush.action.FEEDBACK" />
</intent-filter>
</receiver>

<!-- **Note:** If the start mode of the opened activity is SingleTop, SingleTask, or SingleInstance, plea
se handle it according to the 8th point in the notification troubleshooting self-check list -->
<activity
android:name="com.tencent.android.tpush.XGPushActivity"
android:exported="false" >
<intent-filter>
<!-- If Android Studio is used, please set android:name="android.intent.action"-->
<action android:name="" />
</intent-filter>
</activity>

<!-- **(Required)** TPNS service -->
<service
android:name="com.tencent.android.tpush.service.XGPushServiceV4"
android:exported="true"
android:persistent="true"
android:process=":xg_service_v4" />


<!-- **(Required)** This improves the survival rate of the service -->
<service
```

```
android:name="com.tencent.android.tpush.rpc.XGRemoteService"
android:exported="true">
<intent-filter>
<!-- **(Required)** Please change to the current app package name.PUSH_ACTION, such as the demo
package name: com.qq.xgdemo -->
<action android:name="current app package name.PUSH_ACTION" />
</intent-filter>
</service>


<!-- **(Required)** **Note:** The authorities should be changed to the package name.AUTH_XGPUS
H, such as the demo package name: com.qq.xgdemo -->
<provider
android:name="com.tencent.android.tpush.XGPushProvider"
android:authorities="current app package name.AUTH_XGPUSH"
android:exported="true"/>

<!-- **(Required)** **Note:** The authorities should be changed to the package name.TPUSH_PROVI
DER, such as the demo package name: com.qq.xgdemo -->
<provider
android:name="com.tencent.android.tpush.SettingsContentProvider"
android:authorities="current app package name.TPUSH_PROVIDER"
android:exported="false" />

<!-- **(Required)** **Note:** The authorities should be changed to the package name.TENCENT.MID.
V3, such as the demo package name: com.qq.xgdemo -->
<provider
android:name="com.tencent.mid.api.MidProvider"
android:authorities="current app package name.TENCENT.MID.V3"
android:exported="true" >
</provider>



<!-- **(Required)** Please change YOUR_ACCESS_ID to the AccessId of your app, which is a 10-digit n
umber beginning with "21" and cannot contain spaces -->
<meta-data
android:name="XG_V2_ACCESS_ID"
android:value="YOUR_ACCESS_ID" />
<!-- **(Required)** Please change YOUR_ACCESS_KEY to the AccessKey of your app, which is a 12-cha
racter string beginning with "A" and cannot contain spaces -->
<meta-data
android:name="XG_V2_ACCESS_KEY"
android:value="YOUR_ACCESS_KEY" />
</application>
```

```
<!-- **(Required)** Permissions required by the TPNS SDK -->
<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.WAKE_LOCK" />
<uses-permission android:name="android.permission.VIBRATE" />
<!-- **(Commonly used)** Permissions required by the TPNS SDK-->
<uses-permission android:name="android.permission.RECEIVE_USER_PRESENT" />
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_SETTINGS" />
<!-- **(Optional)** Permissions required by the TPNS SDK-->
<uses-permission android:name="android.permission.RESTART_PACKAGES" />
<uses-permission android:name="android.permission.BROADCAST_STICKY" />
<uses-permission android:name="android.permission.KILL_BACKGROUND_PROCESSES" />
<uses-permission android:name="android.permission.GET_TASKS" />
<uses-permission android:name="android.permission.READ_LOGS" />
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BATTERY_STATS" />
```

Registration and Partial Log Output

1. According to manual access or automatic access, get the TPNS registration log after configuring TPNS (it is recommended to call the registration API with callback during the access process to enable debugging log output of TPNS. For Android Studio, it is recommended to use automatic access via jcenter, without having to configure each node of TPNS in the configuration file as all of them are imported by the dependencies.)

Enable debugging log data

```
XGPushConfig.enableDebug(this,true);
```

token registration

```
XGPushManager.registerPush(this, new XGIOperateCallback() {
@Override
public void onSuccess(Object data, int flag) {
// token may change after the app is uninstalled and then reinstalled on the device
Log.d("TPush", "The registration succeeded, and the device token is: " + data);
```

```
}
@Override
public void onFail(Object data, int errCode, String msg) {
Log.d("TPush", "The registration failed; error code: " + errCode + ", error message: " + msg);
}
})
```

The log of successful registration filtered by "TPush" is as follows:

```
10-09 20:08:46.922 24290-24303/com.qq.xgdemo I/XINGE: [TPush] get RegisterEntity:RegisterEntity [a
ccessId=2100250470, accessKey=null, token=5874b7465d9eead746bd9374559e010b0d1c0bc4, packa
geName=com.qq.xgdemo, state=0, timestamp=1507550766, xgSDKVersion=3.11, appVersion=1.0]
10-09 20:08:47.232 24290-24360/com.qq.xgdemo D/TPush: The registration succeeded, and the devic
e token is: 5874b7465d9eead746bd9374559e010b0d1c0bc4
```

Set an account

```
// Note that TPNS v3.2.2 has upgraded the account binding and unbinding APIs. For details, see the
API documentation.
XGPushManager.bindAccount(getApplicationContext(), "XINGE");
```

The log of successful account registration filtered by "TPush" is as follows:

```
// If the push returns error code 48: account invalid, please confirm whether the account API is succes
sfully called
10-11 15:55:57.810 29299-29299/com.qq.xgdemo D/TPushReceiver: TPushRegisterMessage [accessId=
2100250470, deviceId=853861b6bba92fb1b63a8296a54f439e, account=XINGE, ticket=0, ticketType=0,
token=3f13f775079df2d54e1f82475a28bccd3bfef8c1] successful registration
```

Set a tag

```
XGPushManager.setTag(this,"XINGE");
```

Log of successful tagging:

```
10-09 20:11:42.558 27348-27348/com.qq.xgdemo I/XINGE: [XGPushManager] Action -> setTag with ta
g = XINGE
```

Receive message log

```
10-16 19:50:01.065 5969-6098/com.qq.xgdemo D/XINGE: [i] Action -> handleRemotePushMessage
10-16 19:50:01.065 5969-6098/com.qq.xgdemo I/XINGE: [i] >> msg from service, @msgId=1 @accId=2
100250470 @timeUs=1508154601660412 @recTime=1508154601076 @msg.date= @msg.busiMsgId=
0 @msg.timestamp=1508154601 @msg.type=1 @msg.multiPkg=0 @msg.serverTime=1508154601000
```

```
@msg.ttl=259200 @expire_time=1508154860200076 @currentTimeMillis=1508154601076
10-16 19:50:01.095 5969-6098/com.qq.xgdemo D/XINGE: [m] Action -> handlerPushMessage
10-16 19:50:01.105 5969-6098/com.qq.xgdemo I/XINGE: [m] Receiver msg from server :PushMessageM
anager [msgId=1, accessId=2100250470, busiMsgId=0, content={"n_id":0,"title":"XGDemo","style_id":
1,"icon_type":0,"builder_id":1,"vibrate":0,"ring_raw":"","content":"token push","lights":1,"clearable":1,"a
ction":{"aty_attr":{"pf":0,"if":0},"action_type":1,"activity":""},"small_icon":"","ring":1,"icon_res":"","custom_
content":{}}, timestamps=1508154601, type=1, intent=Intent { act=com.tencent.android.tpush.action.I
NTERNAL_PUSH_MESSAGE cat=[android.intent.category.BROWSABLE] pkg=com.qq.xgdemo (has extr
as) }, messageHolder=BaseMessageHolder [msgJson={"n_id":0,"title":"XGDemo","style_id":1,"icon_typ
e":0,"builder_id":1,"vibrate":0,"ring_raw":"","content":"token push","lights":1,"clearable":1,"action":{"aty
_attr":{"pf":0,"if":0},"action_type":1,"activity":""},"small_icon":"","ring":1,"icon_res":"","custom_content":
{}}, msgJsonStr={"n_id":0,"title":"XGDemo","style_id":1,"icon_type":0,"builder_id":1,"vibrate":0,"ring_ra
w":"","content":"token push","lights":1,"clearable":1,"action":{"aty_attr":{"pf":0,"if":0},"action_type":1,"ac
tivity":""},"small_icon":"","ring":1,"icon_res":"","custom_content":{}}, title=XGDemo, content=token pus
h, customContent=null, acceptTime=null]]
10-16 19:50:01.105 5969-6098/com.qq.xgdemo V/XINGE: [XGPushManager] Action -> msgAck(com.q
q.xgdemo,1)
10-16 19:50:01.115 5969-6098/com.qq.xgdemo I/XINGE: [TPush] title encry obj:{"cipher":"YZXM+CuPh
qaBn4eK0SE9ApWieHznugNT2uKo0OaXtlDDHLJiY7NlvSL2ZnlSb8E7yd7E7i9JU3g0PlFyYNLjokNp1buJ
uPoMYEHaJ0s6vmUMY+cq0Sv782XHxNzekV4a9mRcJ5xsOccIjH1VoskUmikfZJo3XLhZveWNYGPaoto
="}
10-16 19:50:01.125 5969-6098/com.qq.xgdemo E/XINGE: [MessageInfoManager] delOldShowedCache
Message Error! toDelTime: 1507981801138
10-16 19:50:01.145 5969-6098/com.qq.xgdemo I/XINGE: [MessageHelper] Action -> showNotification
{"n_id":0,"title":"XGDemo","style_id":1,"icon_type":0,"builder_id":1,"vibrate":0,"ring_raw":"","content":"to
ken push","lights":1,"clearable":1,"action":{"aty_attr":{"pf":0,"if":0},"action_type":1,"activity":""},"small_ic
on":"","ring":1,"icon_res":"","custom_content":{}}
```

Code Obfuscation

If your project uses tools such as ProGuard to obfuscate the code, please keep the following options; otherwise, the TPNS service will not be available.

```
-keep public class * extends android.app.Service
-keep public class * extends android.content.BroadcastReceiver
-keep class com.tencent.android.tpush.** {* ;}
-keep class com.tencent.mid.** {* ;}
-keep class com.qq.taf.jce.** {*;}
-keep class com.tencent.bigdata.** {* ;}
```

Android SDK API

Last updated：2019-06-26 09:54:30

API Overview

The package name path prefix for all APIs is: `com.tencent.android.tpush` . There are several important classes that provide APIs, including:

Class name	Description
`XGPushManager`	Push service push
`XGPushConfig`	Push service configuration item API
`XGPushBaseReceiver`	Receiver to receive messages and result feedback, which needs to be statically registered in `AndroidManifest.xml` by yourself

XGPushManager Function Class

`XGPushManager` provides a list of APIs of TPNS. The default method is `public static` type.

Prototype	Function
`void registerPush(Context context)`	Start and register (no registration callback)
`void registerPush ( Context context, final XGIOperateCallback callback )`	Start and register (with registration callback)
`void registerPush( Context context, String account, XGIOperateCallback callback )`	Start and register the app, and bind the account at the same time. Recommended for apps with an account system (Used for versions below 3.2.2; there is registration callback)

Prototype	Function

`void bindAccount(`
`Context context,`
`String account,`
`XGIOperateCallback callback`
`)`
	Start and register the app, and bind the account at the same time. Recommended for apps with an account system
	(Used for version 3.2.2 and higher;
	this API will override the account previously bound to the device;
	and only the currently registered account will take effect)

`void bindAccount(`
`Context context,`
`final String account`
`)`
	Start and register the app, and bind the account at the same time. Recommended for apps with an account system
	(Used for version 3.2.2 and higher; this API will override the account previously bound to the device;
	and only the currently registered account will take effect. There is no registration callback)

`void appendAccount(`
`Context context,`
`String account,`
`XGIOperateCallback callback`
`)`
	Start and register the app, and bind the account at the same time. Recommended for apps with an account system
	(Used for version 3.2.2 and higher; this API will retain the previous account and only perform an addition operation.
	A maximum of 10 accounts is allowed for one token; if this limit is exceeded, the previously bound account will be automatically replaced. There is registration callback)

`void appendAccount(`
`Context context,`
`final String account`
`)`
	Start and register the app, and bind the account at the same time. Recommended for apps with an account system
	(Used for version 3.2.2 and higher;
	this API will retain the previous account and only perform an addition operation.
	A maximum of 10 accounts is allowed for one token; if this limit is exceeded, the previously bound account will be automatically replaced. There is no registration callback)

`void delAccount(`
`Context context,`
`final String account,`
`XGIOperateCallback callback`
`)`
	Unbind the specified account (used for version 3.2.2 and higher; there is registration callback)

`void delAccount(`
`Context context,`
`final String account`
`)`
	Unbind the specified account (used for version 3.2.2 and higher; there is no registration callback)

Prototype	Function

`void registerPush(`
`Context context,`
`String account,`
`String ticket,`
`int ticketType,`
`String qua,`
`final XGIOperateCallback callback`
`)`
	Same as above, only for use by businesses containing an login state

`void unregisterPush(Context context)`	Unregister; it is recommended to call when there is no need to receive pushes

`void setTag(`
`Context context,`
`String tagName`
`)`
	Set a tag

`void setTags(`
`Context context,`
`String operateName,`
`Set<String> tags`
`)`
	Set multiple tags, which will override the tags previously set for this device (used for version 4.0.3 and higher)

`void addTags(`
`Context context,`
`String operateName,`
`Set<String> tags`
`)`
	Add multiple tags (used for version 4.0.3 and higher)

`void deleteTag(`
`Context context,`
`String tagName`
`)`
	Delete a tag

`void deleteTags(`
`Context context,`
`String operateName,`
`Set<String> tags`
`)`
	Delete multiple tags (used for version 4.0.3 and higher)

Prototype	Function
`void cleanTags(` `Context context,` `String operateName` `)`	Clear all tags (used for version 4.0.3 and higher)
`XGPushClickedResult onActivityStarted(Activity activity)`	Statistics of effects when Activity is opened; get the delivered custom key-value
`void onActivityStoped(` `Activity activity` `)`	Statistics of effects when Activity is opened
`void setPushNotificationBuilder(` `Context context,` `int notificationBulderId,` `XGPushNotificationBuilder notifiBuilder` `)`	Customize local notification style
`long addLocalNotification(` `Context context,` `XGLocalMessage msg` `)`	Local notification
`boolean isNotificationOpened(` `Context context` `)`	Check whether the notification bar is closed
`void cancelNotifaction(Context context, int id)`	Clear a single notification
`void cancelAllNotifaction(Context context)`	Clear all notifications in the notification bar

XGPushConfig Configuration Class

`XGPushConfig` provides a list of TPNS external configuration APIs. The default method is `public static` type. The `set` and `enable` methods provided for this class can take effect only if called before the `XGPushManager` API.

Prototype	Function
`void enableDebug(Context context, boolean debugMode)`	Whether to enable debugging mode, i.e. outputting the logcat log; important note: in order to ensure data security, it must be set to false before publishing
boolean setAccessId(Context context,long accessId)	Configure accessId
boolean setAccessKey(Context context,String accessKey)	Configure accessKey
String getToken(Context context)	Get token of the device; a normal result can be obtained only if the registration is successful
void setReportNotificationStatusEnable(final Context context,final boolean debugMode)	Set whether to report the notification bar status; enabled by default
void setReportApplistEnable(final Context context,final boolean debugMode)	Set the whether to report the app list for smart push; enabled by default
void enableOtherPush(Context context, boolean flag)	Set whether to support third-party vendor push
void setMiPushAppId(Context context, String appid)	Set Mi push APPID
void setMiPushAppKey(Context context, String appkey)	Set Mi push APPKEY
void setMzPushAppId(Context context, String appid)	Set Meizu push APPID
void setMzPushAppKey(Context context, String appkey)	Set Meizu push APPKEY
void setHuaweiDebug(boolean isHuaweiDebug)	Set Huawei log mode for troubleshooting

XGPushBaseReceiver Broadcast Class

The XGPushBaseReceiver class enables receipt of passthrough messages and feedback on operation results. You must inherit this class and overload related methods.

In addition, you need to register it statically in AndroidManifest.xml (please note that if it is dynamically registered in the code, the current app can receive the messages only when it is running).

Prototype	Function
void onTextMessage(Context context,XGPushTextMessage message)	Callback of the in-app message
void onRegisterResult(Context context,int errorCode,XGPushRegisterResult registerMessage)	Register the callback
void onUnregisterResult(Context context, int errorCode)	Unregister the callback
void onSetTagResult(Context context,int errorCode,String tagName)	Set tag callback
void onDeleteTagResult(Context context, int errorCode,String tagName)	Delete tag callback
void onNotifactionShowedResult(Context context, XGPushShowedResult notifiShowedRlt)	Callback triggered by the display of the notification, where the notification received by the app can be retained
void onNotifactionClickedResult(Context context, XGPushClickedResult message)	Callback triggered by the tap on the notification

Starting and Registering

The app can only use the TPNS service after completing the registration and startup of TPNS. Please ensure that the AccessI and AccessKey are configured before this.

The new version of the SDK has integrated TPNS startup and app registration into the registration API, that is, the startup and registration operations are completed by default by simply calling the registration API.

After successful registration, the device token will be returned. The token is used to identify the uniqueness of the device and is also the unique identifier of the connection between TPNS and the backend. For more information about how to get the token, see the "Getting Token" section.

The registration API usually provides a compact version and a version with callback. Please choose an appropriate version according to your business needs.

Binding Device Registration

Ordinary registration only registers the current device, and the backend can send push messages to different device tokens. There are two versions of the API.

Note: This registration method does not support push to account.

(1) Prototype

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

*** (2) Parameters***

context: The context object of the current app, which cannot be null

(3) Sample

```
java XGPushManager.registerPush(this);
```

In addition, in order to make it easier for you to know whether the registration succeeds, a version with callback is provided.

(1) Prototype

```
java public static void registerPush(Context context, final XGIOperateCallback callback)
```

*** (2) Parameters***

context: The context object of the current app, which cannot be null

callback: The callback of the operation, which include callbacks for success and failure and cannot be null

(3) Sample

```
java XGPushManager.registerPush(this, new XGIOperateCallback() { @Override public void onSuccess(Object data, int flag) { Log.d("TPush", "The registration succeeded, and the device token is: " + data); } @Override public void onFail(Object data, int errCode, String msg) { Log.d("TPush", "The registration failed; error code: " + errCode + ", error message: " + msg); } })
```

Binding Account Registration

Binding account registration is using specified account to register app on the basis of binding device registration. One account cannot be logged into multiple devices. In this way, backend can send push messages to specified accounts. This API has two versions.

Note: Accounts can be email, QQ account number, mobile number, username, etc.

(1) Prototype

```
java public static void registerPush(Context context, String account)
```

*** (2) Parameters***

context: The context object of the current app, which cannot be null

account: This is the bound account. Push messages can be sent to the account after bound. The account cannot be a single character such as "2" or "a".

If you want to push by alias, you need to set the alias in the account field of the registration request when calling the registration API. Only one account alias is allowed for one device.

(3) Sample

`java
XGPushManager.registerPush(this, "UserAccount")

In addition, in order **to make** it easier **for** you **to** know whether the registration succeeds, **a version** with callback **is** provided.

(1) Prototype

```java
public static void registerPush(Context context, String account,
final XGIOperateCallback callback)
```

*** (2) Parameters***

context: The context object of the current app, which cannot be null

account: This is the bound account. Push messages can be sent to the account after bound.

If you want to push by alias, you need to set the alias in the account field of the registration request when calling the registration API. Only one account alias is allowed for one device. If multiple devices log in to the same account, only the last bound device is valid. This cannot be null

callback: The callback of the operation, which include callbacks for success and failure and cannot be null

Note:
Up to 10 accounts are allowed for one token, and up to 100 tokens are allowed for one account.

To bind an account in TPNS v3.2.2 beta and higher, you need to call the new API.

Start and register the app, and bind the account at the same time. Recommended **for** apps with an account **system** (used **for** versions below 3.2.2; there is registration callback).
void registerPush(Context context, String account, XGIOperateCallback callback)

Start and register the app, and bind the account at the same time. Recommended **for** apps with an a

```
ccount system. (Used for version 3.2.2 and higher; this API will override the account previously bound
to the device; and only the currently registered account will take effect.)
void bindAccount(Context context, String account, XGIOperateCallback callback)

Start and register the app, and bind the account at the same time. Recommended for apps with an a
ccount system. (Used for version 3.2.2 and higher; this API will override the account previously bound
to the device; and only the currently registered account will take effect. There is no registration callba
ck.)
void bindAccount(Context context, final String account)

Start and register the app, and bind the account at the same time. Recommended for apps with an a
ccount system. (Used for version 3.2.2 and higher; this API will retain the previous account and only p
erform an addition operation. A maximum of 10 accounts is allowed for one token; if this limit is exc
eeded, the previously bound account will be automatically replaced. There is registration callback.)
void appendAccount(Context context, String account, XGIOperateCallback callback)

Start and register the app, and bind the account at the same time. Recommended for apps with an a
ccount system. (Used for version 3.2.2 and higher; this API will retain the previous account and only p
erform an addition operation. A maximum of 10 accounts is allowed for one token; if this limit is exc
eeded, the previously bound account will be automatically replaced. There is no registration callbac
k.)
void appendAccount(Context context, final String account)
```

(3) Sample

```
XGPushManager.registerPush(this, "UserAccount",
new XGIOperateCallback() {
@Override
public void onSuccess(Object data, int flag) {
Log.d("TPush", "The registration succeeded, and the device token is: " + data);
}

@Override
public void onFail(Object data, int errCode, String msg) {
Log.d("TPush", "The registration failed; error code: " + errCode + ", error message: " + msg);
}
});
```

Unbinding Account

If the app was bound to an account by calling registerPush(context, account) and now needs to be unbound (such as when the user exits), the following method can be called.

Call

`java registerPush(context, "*") or registerPush(context, "*", xGIOperateCallback )`

That is, setting account="*" means unbinding the previous account.

To unbind an account in TPNS v3.2.2 beta and higher, you need to call the new API:

```
// Unbind the specified account (used for version 3.2.2 and higher; there is registration callback)

void delAccount(Context context, final String account, XGIOperateCallback callback)

// Unbind the specified account (used for version 3.2.2 and higher; there is no registration callback)

void delAccount(Context context, final String account  )
```

Note:
Account unbinding just removes the association between the token and the app account. If full/tag/token push is used, the notification/message can still be received.

Registration with Login State

Taking into account the user's login state, such as in Mobile QQ or Qzone scenarios, we provide a registration API with login state, making it easier for use in such scenarios.

(1) Prototype

```
public static void registerPush(Context context, String account,
String ticket, int ticketType, String qua,
final XGIOperateCallback callback)



*** (2) Parameters***


context: The context object of the current app, which cannot be null

callback: The callback of the operation, which include callbacks for success and failure and cannot b
e null

account: This is the bound account. Push messages can be sent to the account after bound.

If you want to push by alias, you need to set the alias in the account field of the registration request
when calling the registration API. Only one account alias is allowed for one device, and up to 15 devi
ces are allowed for one alias. This cannot be null

ticket: Login state ticket, which cannot be null
```

ticketType: Ticket type

qua: A field dedicated to Qzone, which can be **null if** not needed

***(3) Sample**

```java
XGPushManager.registerPush(this, "UserAccount", "ticket", 1, null,
new XGIOperateCallback() {
@Override
public void onSuccess(Object data, int flag) {
Log.d("TPush", "The registration succeeded, and the device token is: " + data);
}

@Override
public void onFail(Object data, int errCode, String msg) {
Log.d("TPush", "The registration failed; error code: " + errCode + ", error message: " + msg);
}
});
```

Getting Registration Result

There are two ways to check whether the registration succeeds.

(1) Use the callback version of the registration API

The XGIOperateCallback class provides an API to process registration success/failure. Please see the sample in the registration API.

Definition of XGIOperateCallback:

```java
/**
* Operation callback API
*/
public interface XGIOperateCallback {
/**
* Callback when the operation succeeds
* @param data The business data of successful operation, such as the token information when the re
gistration succeeds
* @param flag Flag tag
*/
public void onSuccess(Object data, int flag);
/**
* Callback when the operation fails
* @param data Business data of failed operation
```

```
 * @param errCode Error code
 * @param msg Error message
 */
public void onFail(Object data, int errCode, String msg);
}
```

(2) Overload XGPushBaseReceiver

The result can be obtained by overloading the onRegisterResult method of XGPushBaseReceiver.

(Note: The overloaded XGPushBaseReceiver needs to be configured in AndroidManifest.xml. For more information, see the "Message Configuration" section)

Sample

```
/**
 * Registration result
 *
 * @param context
 * App context object
 * @param errorCode
 * Error code; {@link XGPushBaseReceiver#SUCCESS} indicates success, while others indicate failure
 * @param registerMessage
 * Registration result return
 */
```

Below is the list of methods provided by XGPushRegisterResult:

Method name	Return value	Default value	Description
getToken()	String	""	Token of the device, i.e., the unique ID of the device
getAccessId()	long	0	Get the accessId of the registration
getAccount	String	""	Get the account bound to the registration
getTicket()	String	""	Login state ticket
getTicketType()	short	0	Ticket type

Unregistration

When the user has exited or the app is closed and pushes are no longer needed to be received, the app can be unregistered. (Note: Once the device is unregistered, the device will not receive pushed messages before it is re-registered successfully)

(1) Prototype

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

*** (2) Parameters***

`java
context: The context object of the app.

```
***(3) Sample***

```java
XGPushManager.unregisterPush(this);
```

Unregistration result

The result can be obtained by overloading the onUnregisterResult method of XGPushBaseReceiver.

Sample

`java

```
/**
* Unregistration result
*
* @param context
* App context object
* @param errorCode
* Error code; {@link XGPushBaseReceiver#SUCCESS} indicates success, while others indicate failure
*/
@Override
public void onUnregisterResult(Context context, int errorCode) {

}
```
```

Note

The unregistration operation should not be too frequent; otherwise, it may cause backend synchronization delay.

Switching accounts does not require unregistration, and for multiple registrations, the last registration will take effect.

Notification and Message

TPNS mainly provides two push formats:
"Push notification" and "passthrough message command", which have certain differences.

Push Notification (Displayed in the Notification Bar)

This refers to the content displayed in the notification bar of the device. All operations are performed by the TPNS SDK. The app can listen to taps on notifications,
, the notifications delivered in the frontend do not need to be processed by the app and will be displayed in the notification bar by default.

After the TPNS service is successfully registered, notifications generally can be delivered without any settings made.

In general, combined with custom notification styles, regular notifications can satisfy most business needs, and if you need more flexible pushes, you can consider using messages.

In-app Message Command (Not Displayed in Notification Bar)

This refers to the content delivered to the app by TPNS. The app needs to inherit the XGPushBaseReceiver API implementation and handle all the operations on its own. In other words, the messages delivered are not displayed in the notification bar by default, and TPNS is only responsible for delivering messages from the TPNS server to the app, but not the processing logic of the messages, which should be implemented by the app itself. For details, see MessageReceiver in Demo.

Message refers to the text message delivered by the developer through frontend or backend scripts. TPNS is only responsible for delivering the message to the app, while the APP itself is completely responsible for the handling of the message body.

Message is flexible and highly customizable, making it more suitable for scenarios where the app prefers to handle personalized business needs by itself, such as delivering app configuration information and customizing message retention and display.

For example, if a game operator needs to provide different notifications for different scenarios (such as reminders for user upgrade, version update, and marketing campaigns), they can encapsulate such scenarios in messages in JSON format and deliver them to the app, and then the app can display different reminders based on the scenarios to meet personalized needs.

Message Configuration

To make a message receivable, you need to configure the message receiver by configuring the following information in AndroidManifest.xml, where the value of android:name needs to be changed to the receiver implemented by the app itself.

`xml