物联网开发平台

通用功能（应用端）

腾讯云

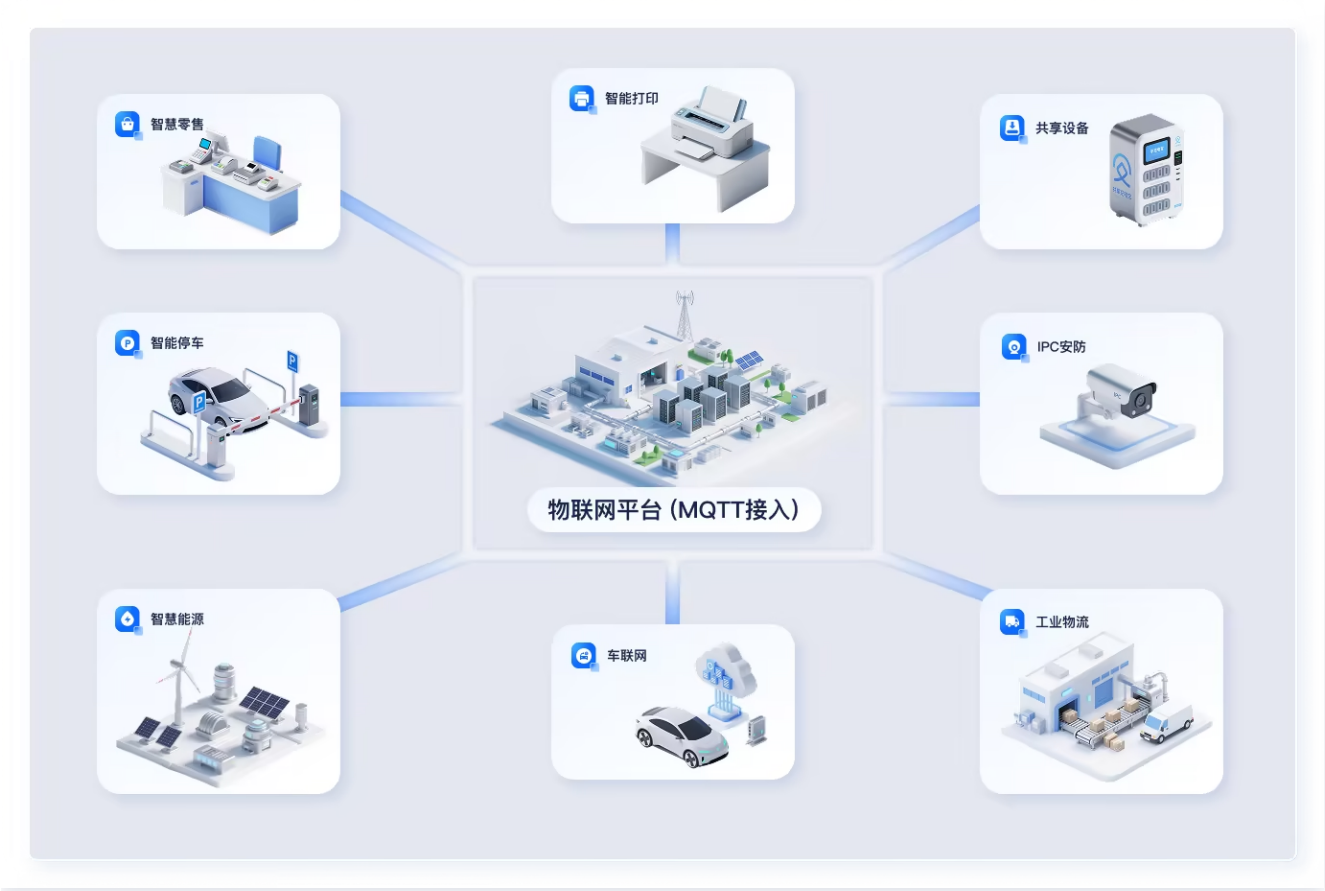
文档目录

通用功能（应用端）
功能介绍

最近更新时间：2026-07-30 12:00:36

功能概述

通用功能（应用端）指平台提供了 Android 和 iOS 应用端 SDK，为智能 IPC、门铃门锁、扫地机、家庭陪伴机器人、大小家电等消费类设备场景，提供用户与设备绑定、家庭成员管理、移动端远程控制绑定设备、设备分享等基础功能。通过集成应用端 SDK，可降低开发者在进行 APP 控制设备、实时监控、视频通话功能对接时的开发成本。



计费说明

平台的通用功能（应用端）主要应用于消费类设备场景，对应的计费方式参考 消费实例计费说明 。

应用场景

消费实例应用场景

应用场景	说明

大小家电	支持各类大小家电的设备接入、管理。如空调、热水器、洗衣机、温控、热泵等各类形态的大小家电设备智能化场景。
全屋智能	支持全屋各类传感器、中控网关、电工照明。
家庭安防	支持各类家庭安防监控 IPC 设备基于 MQTT 协议的设备接入、消息通信、OTA 等能力。

开通服务

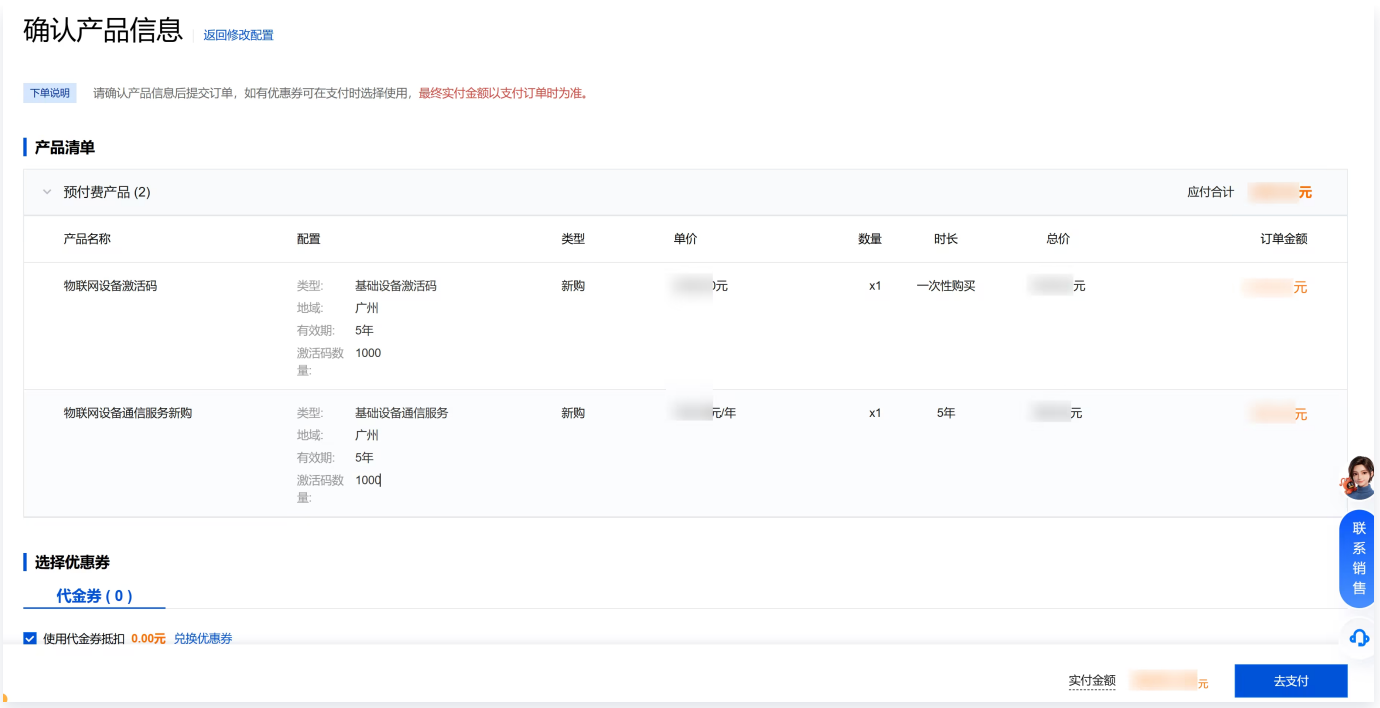
最近更新时间：2026-07-30 12:00:36

开通消费实例正式版

购买消费实例

1. 在购买基础设备接入服务之前，您需要先 注册腾讯云 账号，并完成 实名认证 。
2. 进入 物联网开发平台购买页 ，实例类型选择**消费实例**，服务类型选择“基础设备接入服务”。
3. 生效地域：国内设备选择广州区域，出海设备选择新加坡；有效期默认规格5年。
4. 激活码数量：输入您要量产的设备数量，一个物理设备默认烧录一个激活码。
5. 勾选我已阅读并同意《腾讯云服务协议》，单击**立即购买**。

6. 单击立即购买后进入订单确认，基础设备接入服务分别对应“基础设备激活码”和“基础设备通信服务”。



7. 确认后单击**去支付**进行付款，支付成功后系统将您支付订单的激活码数量累加到对应激活码规格的总可用数量。

查看激活码用量

支付完基础设备接入服务后，用户可进入 物联网开发平台控制台 的**消费实例**，选择左侧导航栏的**用量统计 > 激活码概览**可查看支付成功的订单对应的激活码用量。

创建产品

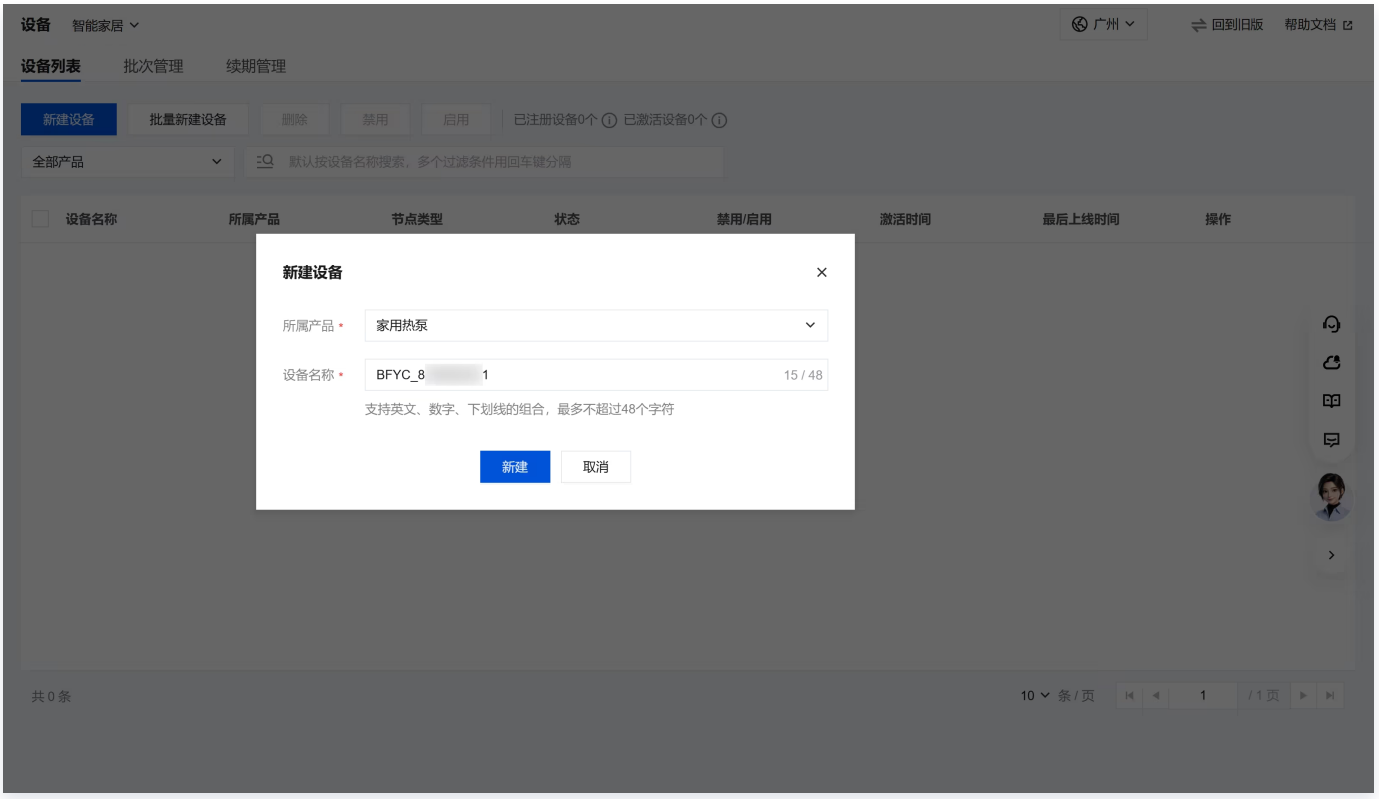
1. 登录 物联网开发平台控制台，进入**实例管理**页面，单击**消费实例**的卡片进入详情页。
2. 选择消费实例左侧导航栏的**产品**，然后单击**创建产品**。

3. 在创建产品页面中，输入“产品名称”，设备类型必须是“**基础设备**”，信息配置好后，单击**创建**。

创建设备

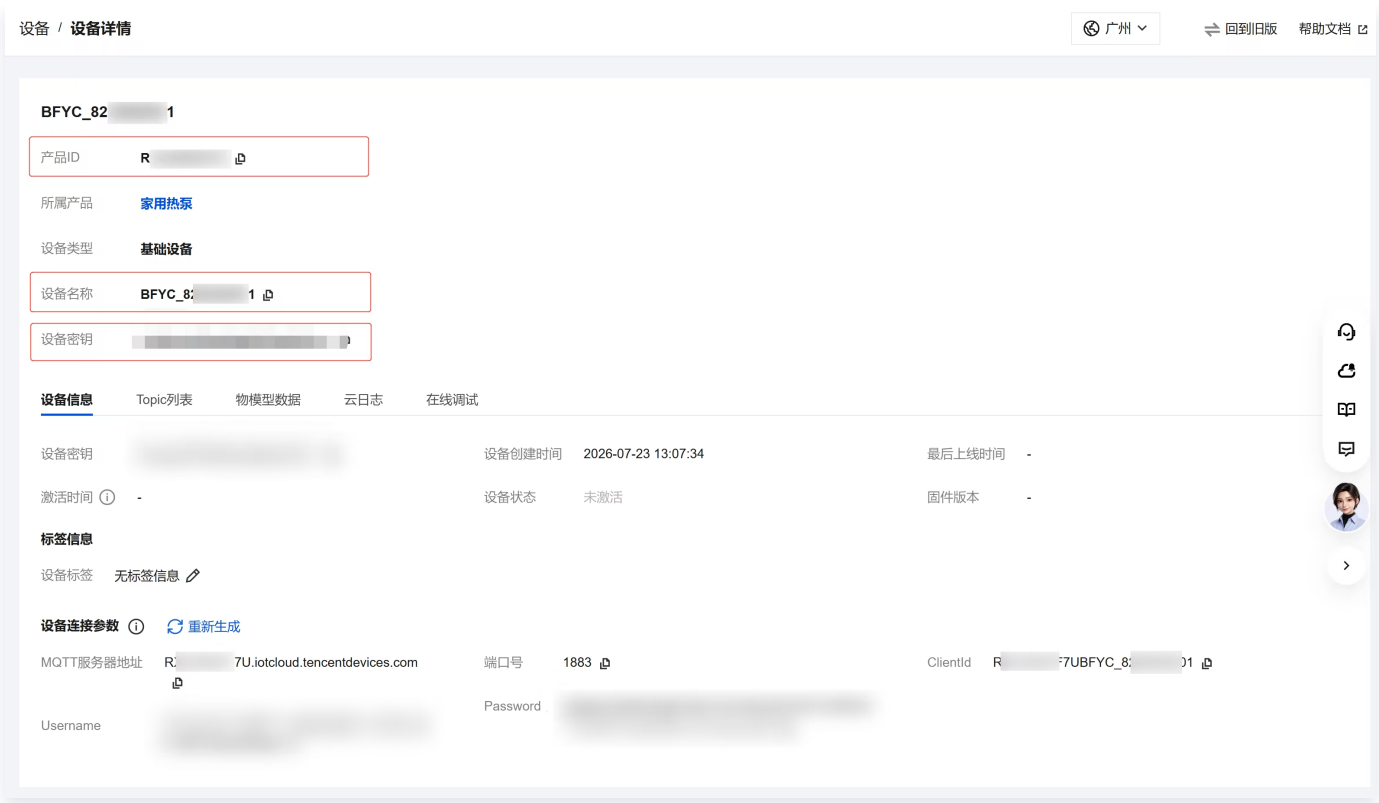
1. 选择消费实例左侧导航栏的**设备**，然后单击**新建设备**。

2. 所属产品选择前述步骤已创建成功的产品，输入设备名称并单击**新建**。



3. 单击创建成功的设备，进入设备详情页获取设备三元组信息。

> **说明：**
> 产品 ID、设备名称、设备密钥三元组信息作为快速接入的设备身份凭证。


设备 / 设备详情
广州
回到旧版
帮助文档
BFYC_82 1
产品ID
所属产品
家用热泵
设备类型
基础设备
设备名称
设备密钥
设备信息
Topic列表
物模型数据
云日志
在线调试
设备密钥
设备创建时间
2026-07-23 13:07:34
最后上线时间
激活时间
设备状态
未激活
固件版本
标签信息
设备标签
无标签信息
设备连接参数
重新生成
MQTT服务器地址
7U.iotcloud.tencentdevices.com
端口号
1883
ClientId
Username
Password

快速接入
Android

最近更新时间：2026-07-30 12:00:38

本文将介绍如何快速完成腾讯云物联网（IoT）应用端 SDK 的接入，在 Android 平台完成登录、家庭管理、设备绑定和设备命令下发等基础能力验证。

前提条件

开通服务

请先按照 开通服务 文档，完成服务开通、设备创建和应用创建，并获取设备信息：

- AppKey
- AppSecret
- ProductId
- DeviceName

定义物模型

请先参考 配置物模型 文档，完成上面 ProductId 对应产品物模型的定义。

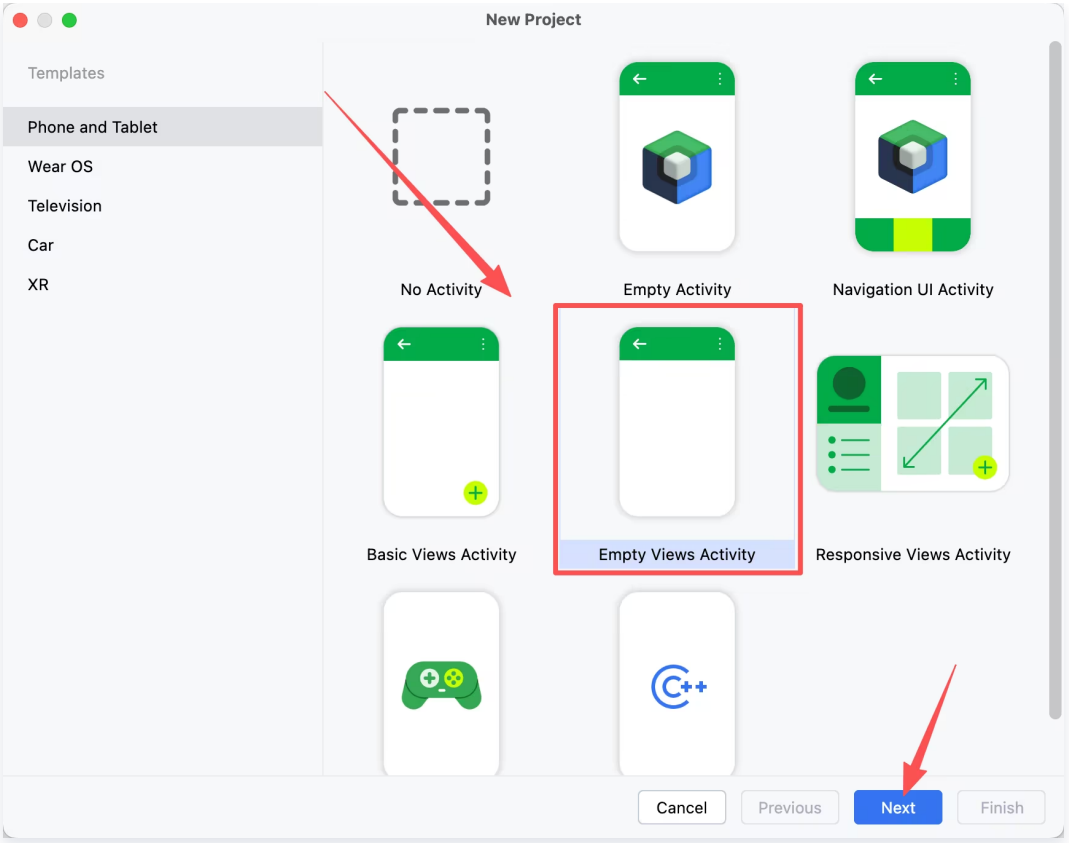
环境准备

- Android Studio 3.5 及以上版本。
- 最低兼容 Android 4.4（SDK API Level 19），建议 Android 5.0（SDK API Level 21）及以上。

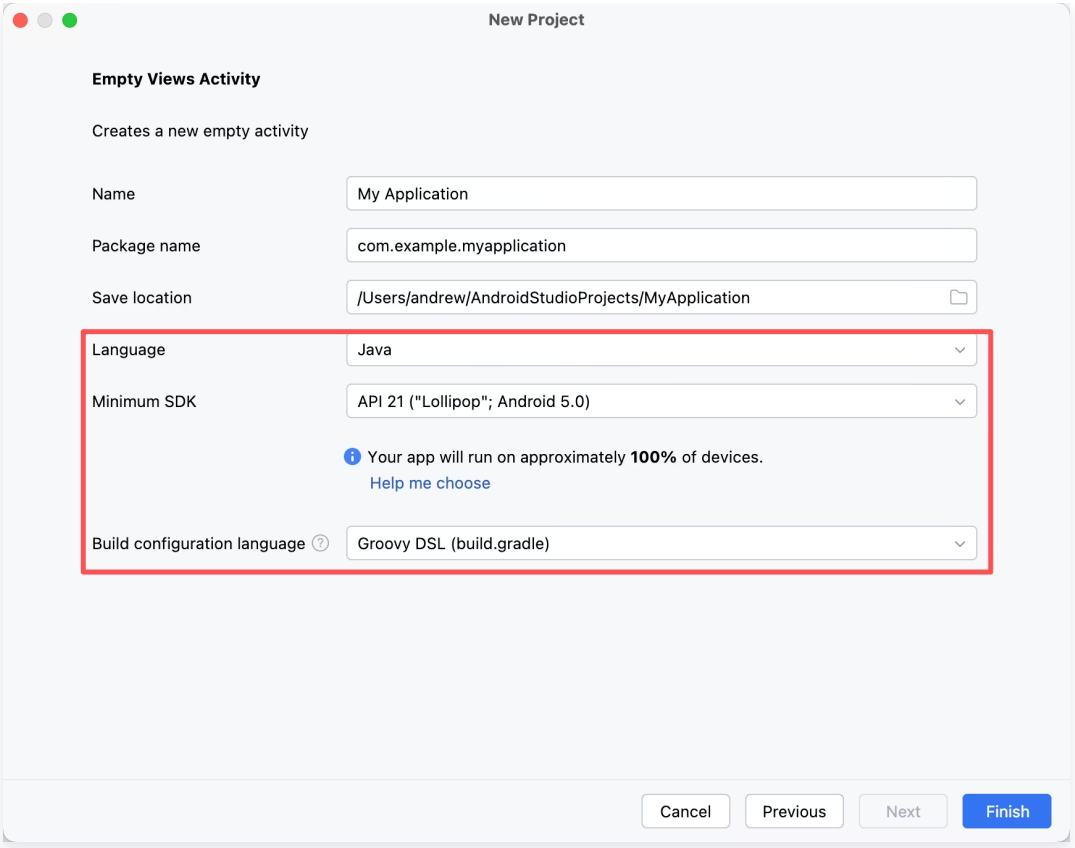
创建项目

参考以下步骤创建一个新的 Android 项目。如果已有项目，可跳过本节。

1. 打开 Android Studio，选择 **New Project**（新建项目）。
2. 选择 **Empty Views Activity** 模板，单击 **Next**。



3. 在 **Configure your project** 页面配置：
   - **Language** 选择 **Java**。
   - **Minimum SDK** 建议 Android 5.0（API 21）及以上。
   - **Build configuration language** 选择 **Groovy DSL (build.gradle)**。

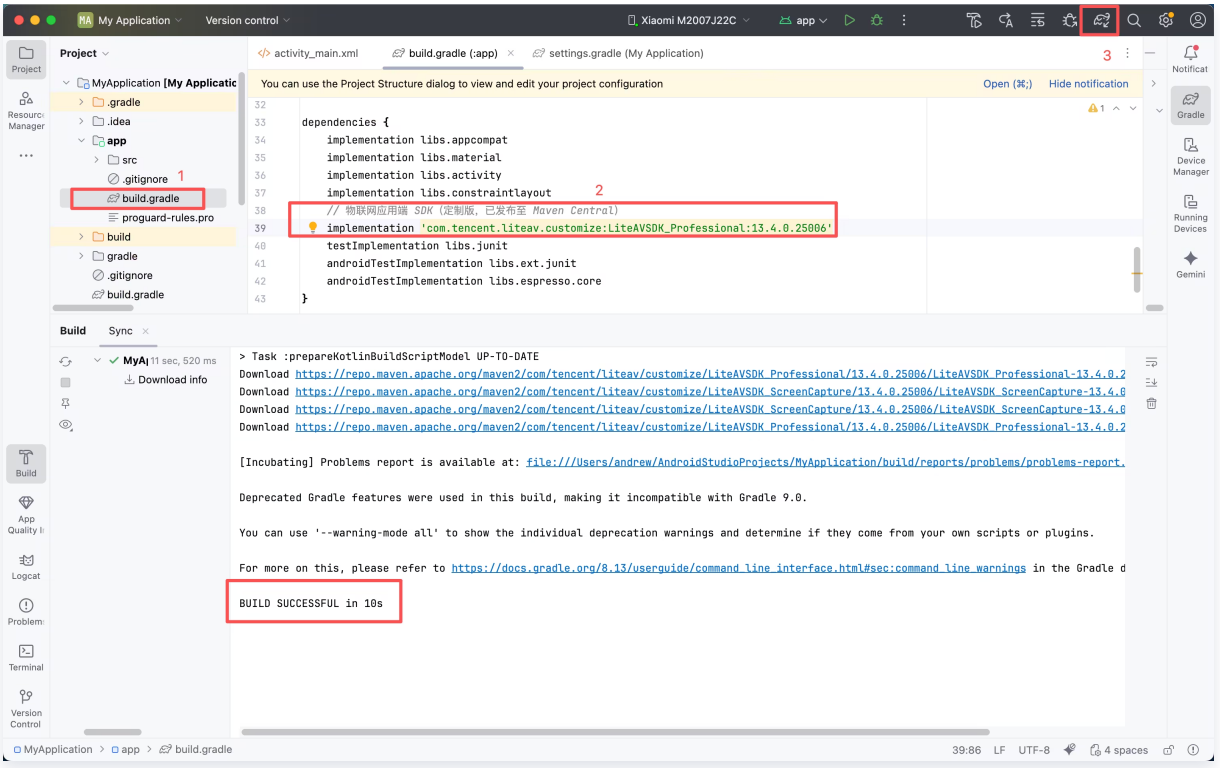


集成 SDK

1. 在 **app/build.gradle**（非根目录的 build.gradle）的 **dependencies** 中添加 IoT SDK 依赖：

```
dependencies {
    // 物联网应用端 SDK
    implementation
'com.tencent.liteav.customize:LiteAVSDK_Professional:13.4.0.25011'
}
```

2. 完成配置后，在 Android Studio 工具栏单击 **Sync Now**，SDK 将自动下载并集成到工程中。出现 Build successful 即表示成功。



接入步骤

步骤 1：创建 SDK 实例

调用 `TXIoTEngine.getInstance` 创建 SDK 实例，注册登录、签名过期和设备状态推送监听。

```
TXIoTEngine iotEngine =
TXIoTEngine.getInstance(getApplicationContext());
iotEngine.addListener(new TXIoTEngine.TXIoTEngineListener() {
    @Override
    public void onLoginSuccess() {
        // 登录成功
    }

    //其他回调省略实现
});
```

步骤 2：计算登录签名

> **说明：**
> 快速接入阶段可直接在客户端计算签名进行调试。正式上线时请将签名计算逻辑放在业务后台，不要将 AppSecret 保存在客户端。

签名参数

参与签名的参数：

参数	说明
`RequestId`	唯一请求 ID，建议使用 UUID。
`Timestamp`	当前 UNIX 秒级时间戳。
`Nonce`	随机正整数，用于和时间戳一起防重放。
`AppKey`	从 开通服务 文档获取的 AppKey。
`OpenID`	用户标识，需与后续登录的 `userId` 保持一致。支持数字、字母、下划线，长度不超过 32 字节。

> ⚠ **注意：**
> OpenID 与 UserID 的关系：签名参数名为 **OpenID**，SDK 登录接口参数名为 **userId**，两者是同一个东西，值必须保持一致。

签名计算规则

1. 去掉值为空的参数。
2. 将参数按参数名的字典序升序排列。
3. 将排序后的参数按 `Key=Value` 格式拼接。
4. 使用 `&` 连接所有参数，得到签名原文。
5. 使用从 开通服务 文档获取的 `AppSecret` 对签名原文进行 HMAC-SHA1 签名。
6. 对签名结果进行 Base64 编码，得到最终 `Signature` 。

签名计算示例

例如，参与签名的参数如下：

```
RequestId=8b8d499bbba1ac28b6da21b4
Timestamp=1546315200
Nonce=71087795
AppKey=your_app_key
OpenID=user_001
```

排序后的签名原文：

```
AppKey=your_app_key&Nonce=71087795&OpenID=user_001&RequestId=8b8d499bbba
1ac28b6da21b4&Timestamp=1546315200
```

参考以下 Python 示例计算签名：

```python
import base64
import hashlib
import hmac

def generate_signature(app_secret, request_id, timestamp, nonce,
app_key, user_id):
    params = {
        "RequestId": request_id,
        "Timestamp": str(timestamp),
        "Nonce": str(nonce),
        "AppKey": app_key,
        "OpenID": user_id,
    }
    # 去掉值为空的参数
    params = {
        key: value
        for key, value in params.items()
        if value is not None and value != ""
    }
    # 按参数名的字典序升序排列
    sorted_items = sorted(params.items(), key=lambda item: item[0])
    # 按 Key=Value 格式拼接，使用 & 连接
    source = "&".join([f"{key}={value}" for key, value in sorted_items])
    # 使用 AppSecret 对签名原文进行 HMAC-SHA1 签名
    digest = hmac.new(
        app_secret.encode("utf-8"),
        source.encode("utf-8"),
        hashlib.sha1,
    ).digest()
    # 对签名结果进行 Base64 编码
    signature = base64.b64encode(digest).decode("utf-8")
    return signature
```

```
signature = generate_signature(
    app_secret="your_app_secret",
    request_id="8b8d499bbba1ac28b6da21b4",
    timestamp=1546315200,
    nonce=71087795,
    app_key="your_app_key",
    user_id="user_001",
)
print(signature)
```

Running Environment

Operating System: Ubuntu 24.04.3 LTS / x86_64

Runtime Version: Python 3.11.1

步骤 3：登录 SDK

调用 `TXIoTEngine.login` 登录 SDK，传入上一步计算得到的签名参数。

参数说明

参数	类型	说明
`appKey`	String	从 开通服务 文档获取的 AppKey。
`userId`	String	用户标识，支持数字、字母、下划线，长度不超过 32 字节。 首次使用会自动注册，已注册则登录原有账号，该账号下的设备绑定关系仍然保留。需与签名参数 `OpenID` 保持一致。
`userSignature.requestId`	String	对应签名参数 `RequestId` 。
`userSignature.timestamp`	long	对应签名参数 `Timestamp` 。
`userSignature.nonce`	int	对应签名参数 `Nonce` 。
`userSignature.signature`	String	签名计算结果。

```
TXIoTEngine.TXIoTUserSignature userSignature = new
TXIoTEngine.TXIoTUserSignature();
userSignature.requestId = requestId;
userSignature.timestamp = timestamp;
userSignature.nonce = nonce;
userSignature.signature = signature;

iotEngine.login(appKey, userId, userSignature);
```

步骤 4：获取家庭列表

登录成功后，调用 `getFamilyManager` 获取家庭管理实例，再调用 `getFamilyList` 获取家庭列表。后续绑定设备时需要 `familyId`。

```
TXIoTFamilyManager familyManager = iotEngine.getFamilyManager();
if (familyManager == null) {
    // SDK 未登录或登录态已失效
    return;
}

familyManager.getFamilyList(new TXIoTCallback<List<TXIoTFamilyInfo>>() {
    @Override
    public void onSuccess(List<TXIoTFamilyInfo> familyList) {
        // 选择一个家庭，记录 familyId
        String familyId = familyList.get(0).familyId;
    }

    @Override
    public void onError(TXIoTErrorCode errorCode, String errorMessage) {
        // 获取家庭列表失败
    }
});
```

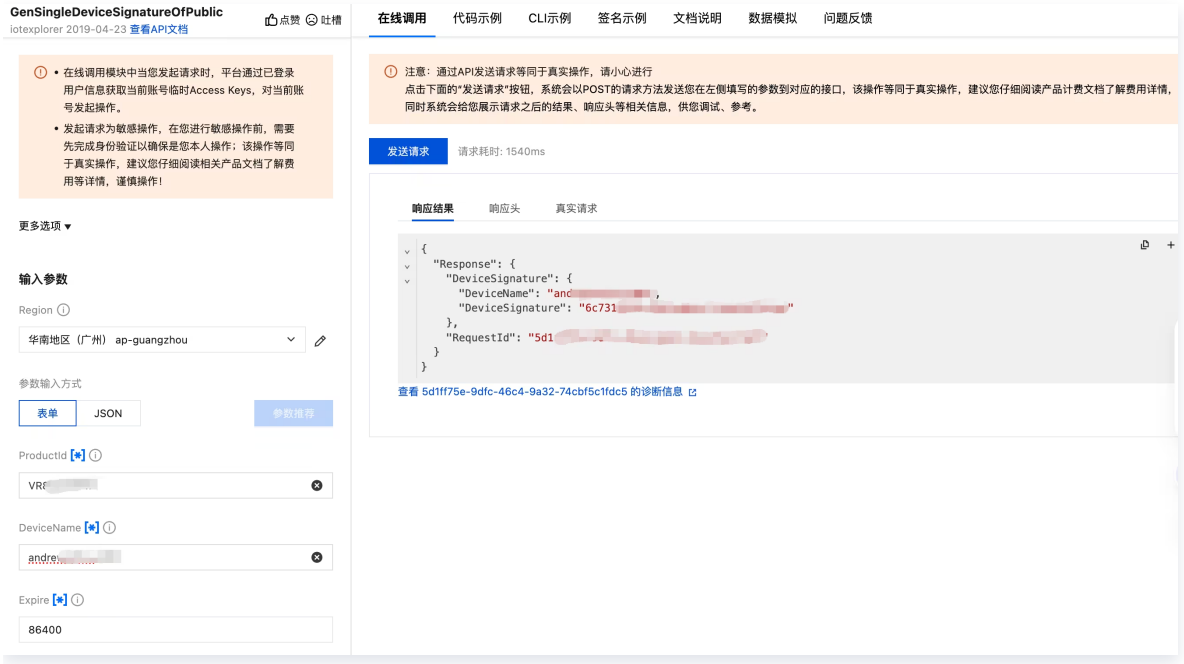
如果当前账号下没有家庭，可调用 `createFamily` 创建：

```
familyManager.createFamily("我的家庭", callback);
```

步骤 5：绑定设备

绑定设备时需要**设备绑定签名（DeviceSignature）**，通过 API Explorer 在线调试获取。

打开 **GenSingleDeviceSignatureOfPublic 接口**，填入从 **开通服务** 文档获取的 **ProductId**、**DeviceName**（Expire 自己选合适的就行），单击**在线调试**，从返回结果的 `Response.DeviceSignature` 中获取签名值。



调用 `getDeviceManager` 获取设备管理实例，再调用 `bindDevice` 将设备绑定到指定家庭。

```
TXIoTDeviceManager deviceManager = iotEngine.getDeviceManager();
if (deviceManager == null) {
    // SDK 未登录或登录态已失效
    return;
}

deviceManager.bindDevice(
    familyId,
    deviceBindSignature,
    new TXIoTCallback<TXIoTDeviceInfo>() {
        @Override
        public void onSuccess(TXIoTDeviceInfo deviceInfo) {
            // 绑定成功，记录 deviceId 用于后续控制设备
            TXIoTDeviceId deviceId = deviceInfo.deviceId;
        }

        @Override
        public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
```

```
            // 绑定失败
        }
    }
);
```

步骤 6：发送设备命令

调用 `sendCommand` 向设备发送物模型命令。`jsonData` 需与本文 **前提条件** 章节产品物模型定义保持一致。

`sendCommand` 的 `deviceId` 参数有以下两种来源，二者选其一即可：

- **方式一（推荐，需先完成步骤5的设备绑定）**：从绑定成功回调的 `deviceInfo.deviceId` 中获取。
- **方式二**：使用控制台获取的 ProductId 和 DeviceName 手动构造（仍需完成步骤5的设备绑定）。

```
// 获取 deviceId（以下两种方式任选其一）：

// 【方式一：推荐】从步骤5绑定成功回调中获取
// TXIoTEngineDef.TXIoTDeviceId deviceId = deviceInfo.deviceId;

// 【方式二】使用 ProductId + DeviceName 手动构造
// ProductId 和 DeviceName 来自前提条件章节从控制台获取的值
// TXIoTEngineDef.TXIoTDeviceId deviceId = new
TXIoTEngineDef.TXIoTDeviceId();
// deviceId.productId = "your_product_id";
// deviceId.deviceName = "your_device_name";

String jsonData = "{\"power_switch\":1}"; // 需要与产品物模型定义保持一致

deviceManager.sendCommand(
    deviceId,  // 传入方式一或方式二获取的 deviceId
    jsonData,
    new TXIoTCallback<String>() {
        @Override
        public void onSuccess(String result) {
            // 命令发送成功
        }

        @Override
        public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
            // 命令发送失败
        }
```

```
    }
);
```

常见问题

为什么 `getFamilyManager` 或 `getDeviceManager` 返回 `null`？

SDK 尚未登录或登录态已失效。请先调用 `TXIoTEngine.login`，并在收到 `onLoginSuccess` 后再获取对应 Manager。

登录签名过期后如何处理?

当收到 `onUserSignatureExpired` 回调时，请重新计算登录签名，然后再次调用 `TXIoTEngine.login`。

绑定设备失败如何排查?

请检查 `familyId` 是否属于当前登录用户，设备绑定签名是否正确，以及设备是否已被其他家庭绑定。SDK 会在 `onError` 中返回具体错误码和错误信息。

下一步

完成基础接入后，您还可以继续接入以下能力：

- 绑定与分享
- 远程控制
- 实时监控
- 视频通话

iOS

最近更新时间：2026-07-30 12:00:38

本文将介绍如何快速完成腾讯云物联网（IoT）应用端 SDK 的接入，在 iOS 平台完成登录、家庭管理、设备绑定和设备命令下发等基础能力验证。

前提条件

开通服务

请先按照 开通服务 文档，完成服务开通、设备创建和应用创建，并获取设备信息：

- AppKey
- AppSecret
- ProductId
- DeviceName

定义物模型

请先参考 配置物模型 文档，完成上面 ProductId 对应产品物模型的定义。

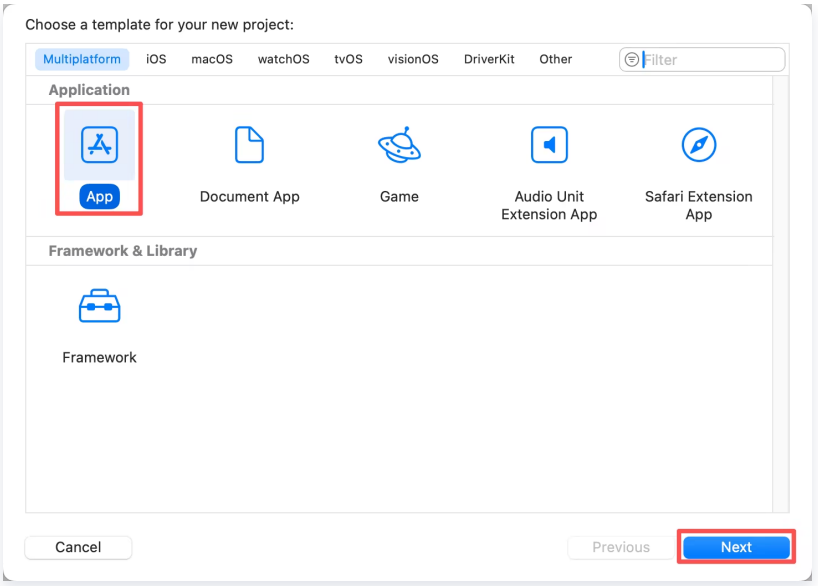
环境准备

- Xcode 11.0 及以上版本。
- 确保项目已设置有效的开发者签名。
- 一台 iOS 9.0 及以上真机。

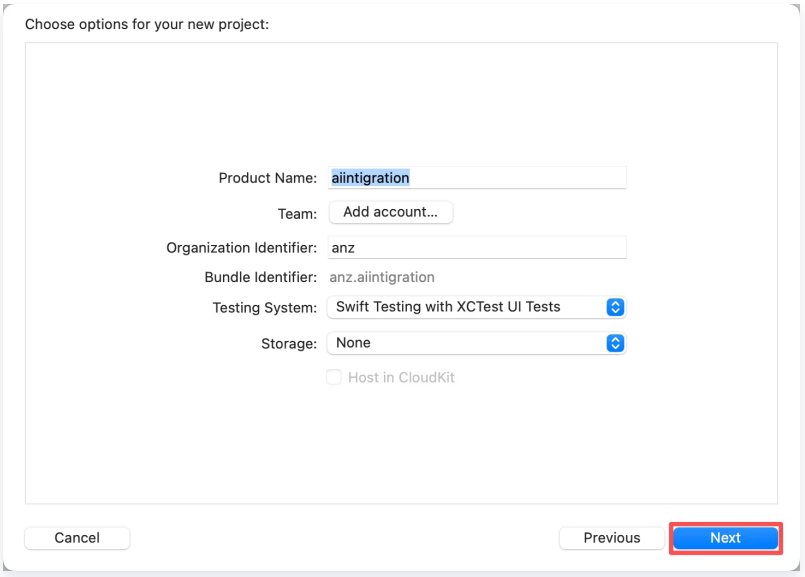
创建项目

参考以下步骤创建一个新的 iOS 项目。如果已有项目，可跳过本节。

1. 打开 Xcode，选择 **New Project**（新建项目）。
2. 在平台标签中选择 **iOS**，然后选择 **App** 模板，单击 **Next**。



3. 在 **Choose options for your new project** 页面设置 **Product Name**、**Team**、**Organization Identifier** 等项目基本信息，单击 **Next** 并选择保存路径，最后单击 **Create** 完成创建。



集成 SDK

1. 在项目根目录的 `Podfile` 中添加 IoT SDK 依赖：

> **说明：**
> `Podfile` 是 iOS 项目的 CocoaPods 依赖配置文件，位于项目根目录（与 `.xcodeproj` 同级），用文本编辑器编辑即可。如果目录下没有 `Podfile`，先在终端进入项目目录执行 `pod init` 生成。

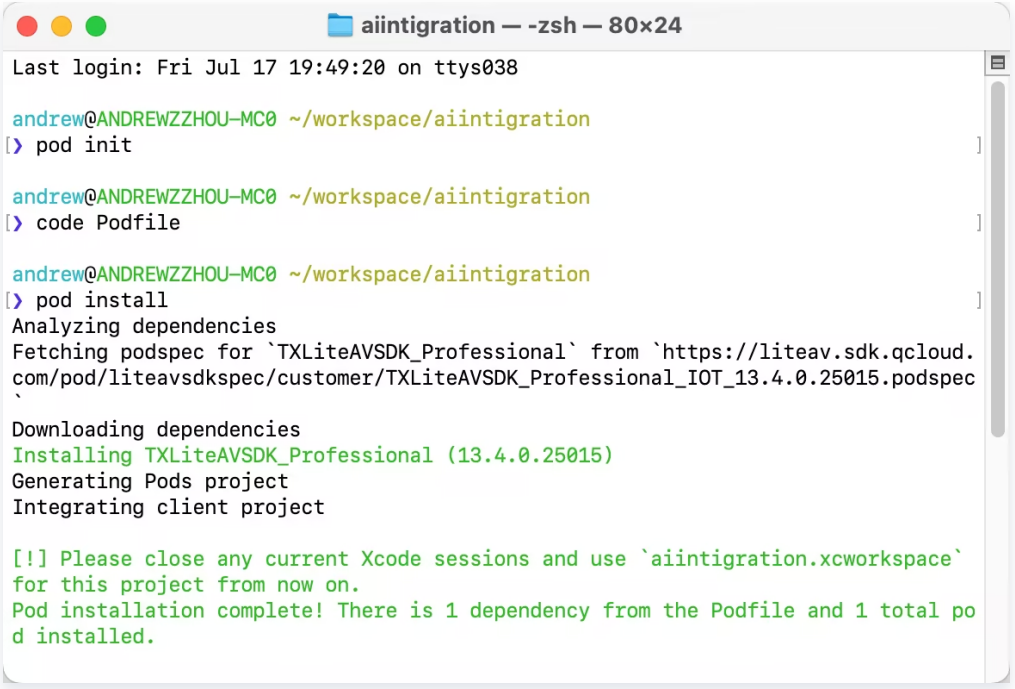
```
# 将下方的 'YourApp' 替换为您 Xcode 项目中实际的 Target 名称
platform :ios, '12.0'
```

```
target 'YourApp' do
  # 物联网应用端 SDK（IOT 定制版）
  pod 'TXLiteAVSDK_Professional',:podspec
=>'https://liteav.sdk.qcloud.com/pod/liteavsdkspec/customer/TXLiteAVSD
K_Professional_IOT_13.4.0.25016.podspec'
end
```

Xcode 16 兼容性问题：若使用 Xcode 16 及以上版本，`pod install` 后可能报部署目标版本错误。请在 `Podfile` 末尾添加以下配置解决：

```
post_install do |installer|
  installer.pods_project.build_configurations.each do |config|
    config.build_settings['IPHONEOS_DEPLOYMENT_TARGET'] = '12.0'
  end
  installer.pods_project.targets.each do |target|
    target.build_configurations.each do |config|
      config.build_settings['IPHONEOS_DEPLOYMENT_TARGET'] = '12.0'
    end
  end
end
```

2. **完成配置后，在终端执行 `pod install`，SDK 将自动下载并集成到工程中。**



3. **`pod install` 完成后，在 Finder 中找到项目目录下的 .xcworkspace 文件并双击打开（请打开 `.xcworkspace` 而非 `.xcodeproj`）。**

4. 在 Xcode 的 **Build Settings** 中搜索 **User Script Sandboxing**，将其值设置为 **No**。
5. 在 **Signing & Capabilities** 中确认已配置有效的开发者签名。

接入步骤

步骤 1：获取 SDK 实例

调用 `TXIoTEngine.getInstance` 获取 SDK 实例，并注册 `TXIoTEngineDelegate` 监听登录、签名过期和设备状态推送。

```
// 获取单例实例
TXIoTEngine *iotEngine = [TXIoTEngine getInstance];
// 注册监听（self 需遵循 TXIoTEngineDelegate 协议）
[iotEngine addDelegate:self];
```

在监听对象中实现 `TXIoTEngineDelegate` 协议方法：

```
#pragma mark - TXIoTEngineDelegate

- (void)onLoginSuccess {
    // 登录成功
}

- (void)onLoginFailure:(TXIoTErrorCode)errCode errMsg:(NSString *)errMsg
{
    // 登录失败
}

- (void)onUserSignatureExpired {
    // 登录签名过期，请重新计算签名后再次调用 login
}

- (void)onReceivePushMessage:(TXIoTPushMessage *)pushMessage {
    // 接收设备在线、离线等状态变更
}
```

步骤 2：计算登录签名

> **说明：**

> 快速接入阶段可直接在客户端计算签名进行调试。正式上线时请将签名计算逻辑放在业务后台，不要将 AppSecret 保存在客户端。

签名参数

参与签名的参数：

参数	说明
`RequestId`	唯一请求 ID，建议使用 UUID。
`Timestamp`	当前 UNIX 秒级时间戳。
`Nonce`	随机正整数，用于和时间戳一起防重放。
`AppKey`	控制台应用详情中的 AppKey。
`OpenID`	用户标识，需与后续登录的 `userId` 保持一致。支持数字、字母、下划线，长度不超过 32 字节。

> ⚠ **注意：**
> **OpenID 与 UserID 的关系：**签名参数名为 **OpenID**，SDK 登录接口参数名为 **userId**，两者是同一个东西，值必须保持一致。

签名计算规则

1. 去掉值为空的参数。
2. 将参数按参数名的字典序升序排列。
3. 将排序后的参数按 `Key=Value` 格式拼接。
4. 使用 `&` 连接所有参数，得到签名原文。
5. 使用从 开通服务 文档获取的 `AppSecret` 对签名原文进行 HMAC-SHA1 签名。
6. 对签名结果进行 Base64 编码，得到最终 `Signature` 。

签名计算示例

例如，参与签名的参数如下：

```
RequestId=8b8d499bbba1ac28b6da21b4
Timestamp=1546315200
Nonce=71087795
```

```
AppKey=your_app_key
OpenID=user_001
```

排序后的签名原文：

```
AppKey=your_app_key&Nonce=71087795&OpenID=user_001&RequestId=8b8d499bbba
1ac28b6da21b4&Timestamp=1546315200
```

参考以下 Python 示例计算签名：

```python
import base64
import hashlib
import hmac

def generate_signature(app_secret, request_id, timestamp, nonce,
app_key, user_id):
    params = {
        "RequestId": request_id,
        "Timestamp": str(timestamp),
        "Nonce": str(nonce),
        "AppKey": app_key,
        "OpenID": user_id,
    }
    # 去掉值为空的参数
    params = {
        key: value
        for key, value in params.items()
        if value is not None and value != ""
    }
    # 按参数名的字典序升序排列
    sorted_items = sorted(params.items(), key=lambda item: item[0])
    # 按 Key=Value 格式拼接，使用 & 连接
    source = "&".join([f"{key}={value}" for key, value in sorted_items])
    # 使用 AppSecret 对签名原文进行 HMAC-SHA1 签名
    digest = hmac.new(
        app_secret.encode("utf-8"),
        source.encode("utf-8"),
        hashlib.sha1,
    ).digest()
```

```
    # 对签名结果进行 Base64 编码
    signature = base64.b64encode(digest).decode("utf-8")
    return signature

signature = generate_signature(
    app_secret="your_app_secret",
    request_id="8b8d499bbba1ac28b6da21b4",
    timestamp=1546315200,
    nonce=71087795,
    app_key="your_app_key",
    user_id="user_001",
)
print(signature)
```

Running Environment

Operating System: Ubuntu 24.04.3 LTS / x86_64

Runtime Version: Python 3.11.1

步骤 3：登录 SDK

调用 `TXIoTEngine.login` 登录 SDK，传入上一步计算得到的签名参数。

参数说明

参数	类型	说明
`appKey`	NSString	控制台应用详情中的 AppKey。
`userId`	NSString	用户标识，支持数字、字母、下划线，长度不超过 32 字节。首次使用会自动注册，已注册则登录原有账号，该账号下的设备绑定关系仍然保留。需与签名参数 `OpenID` 保持一致。
`userSignature.requestId`	NSString	对应签名参数 `RequestId` 。
`userSignature.timestamp`	int64_t	对应签名参数 `Timestamp` 。
`userSignature.nonce`	NSInteger	对应签名参数 `Nonce` 。

`userSignature.signature`	NSString	签名计算结果。

```
TXIoTUserSignature *userSignature = [[TXIoTUserSignature alloc] init];
userSignature.requestId = requestId;
userSignature.timestamp = timestamp;
userSignature.nonce = nonce;
userSignature.signature = signature;

[iotEngine login:appKey userId:userId userSignature:userSignature];
```

步骤 4：获取家庭列表

登录成功后，调用 `getFamilyManager` 获取家庭管理实例，再调用 `getFamilyList` 获取家庭列表。后续绑定设备时需要 `familyId`。

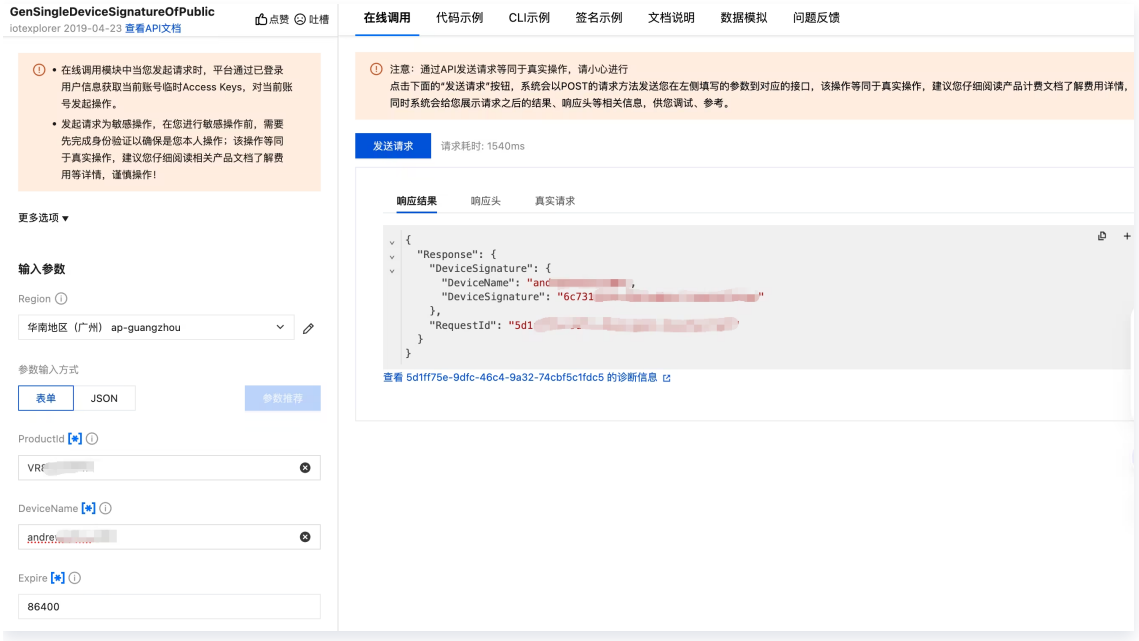
```
TXIoTFamilyManager *familyManager = [iotEngine getFamilyManager];
if (familyManager == nil) {
    // SDK 未登录或登录态已失效
    return;
}

TXIoTCallback<NSArray<TXIoTFamilyInfo *> *> *callback = [[TXIoTCallback
alloc] init];
callback.onSuccess = ^(NSArray<TXIoTFamilyInfo *> * _Nullable
familyList) {
    // 选择一个家庭，记录 familyId
    NSString *familyId = familyList.firstObject.familyId;
};
callback.onError = ^(TXIoTErrorCode errorCode, NSString * _Nullable
errorMessage) {
    // 获取家庭列表失败
};
[familyManager getFamilyList:callback];
```

如果当前账号下没有家庭，可调用 `createFamily` 创建：

```
TXIoTCallback<TXIoTFamilyInfo *> *callback = [[TXIoTCallback alloc]
init];
callback.onSuccess = ^(TXIoTFamilyInfo * _Nullable familyInfo) {
    // 创建成功，记录 familyId
};
callback.onError = ^(TXIoTErrorCode errorCode, NSString * _Nullable
errorMessage) {
    // 创建家庭失败
};
[familyManager createFamily:@"我的家庭" callback:callback];
```

步骤 5：绑定设备

绑定设备时需要设备绑定签名（DeviceSignature），通过 API Explorer 在线调试获取。
打开 GenSingleDeviceSignatureOfPublic 接口，填入从 开通服务 文档获取的 ProductId、DeviceName（Expire 自己选合适的就行），单击在线调试，从返回结果的 `Response.DeviceSignature` 中获取签名值。



调用 `getDeviceManager` 获取设备管理实例，再调用 `bindDevice` 将设备绑定到指定家庭。

```
TXIoTDeviceManager *deviceManager = [iotEngine getDeviceManager];
if (deviceManager == nil) {
    // SDK 未登录或登录态已失效
    return;
}
```

```
TXIoTCallback<TXIoTDeviceInfo *> *callback = [[TXIoTCallback alloc]
init];
callback.onSuccess = ^(TXIoTDeviceInfo * _Nullable deviceInfo) {
    // 绑定成功，记录 deviceId 用于后续控制设备
    TXIoTDeviceId *deviceId = deviceInfo.deviceId;
};
callback.onError = ^(TXIoTErrorCode errorCode, NSString * _Nullable
errorMessage) {
    // 绑定失败
};
[deviceManager bindDevice:familyId deviceBindSignature:@"您的签名"
callback:callback];
```

步骤 6：发送设备命令

调用 `sendCommand` 向设备发送物模型命令。`jsonData` 需与本文 前提条件 章节产品物模型定义保持一致。`sendCommand` 的 `deviceId` 参数有以下两种来源，二者选其一即可：

- **方式一（推荐，需先完成步骤5的设备绑定）**：从绑定成功回调的 `deviceInfo.deviceId` 中获取。
- **方式二**：使用控制台获取的 ProductId 和 DeviceName 手动构造（仍需完成步骤5的设备绑定）。

```
// 获取 deviceId（以下两种方式任选其一）：

// 【方式一：推荐】从步骤5绑定成功回调中获取
// TXIoTDeviceId *deviceId = deviceInfo.deviceId;

// 【方式二】使用 ProductId + DeviceName 手动构造
// ProductId 和 DeviceName 来自前提条件章节从控制台获取的值
// TXIoTDeviceId *deviceId = [[TXIoTDeviceId alloc] init];
// deviceId.productId = @"your_product_id";
// deviceId.deviceName = @"your_device_name";

NSString *jsonData = @"{\"power_switch\":1}"; // 需要与产品物模型定义保持一致
TXIoTCallback<NSString *> *callback = [[TXIoTCallback alloc] init];
callback.onSuccess = ^(NSString * _Nullable result) {
    // 命令发送成功
};
callback.onError = ^(TXIoTErrorCode errorCode, NSString * _Nullable
errorMessage) {
    // 命令发送失败
};
```

```
[deviceManager sendCommand:deviceId jsonData:jsonData
callback:callback];  // 传入方式一或方式二获取的 deviceId
```

常见问题

为什么 `getFamilyManager` 或 `getDeviceManager` 返回 `nil`？

SDK 尚未登录或登录态已失效。请先调用 `TXIoTEngine.login`，并在收到 `onLoginSuccess` 后再获取对应 Manager。

登录签名过期后如何处理?

当收到 `onUserSignatureExpired` 回调时，请重新计算登录签名，然后再次调用 `TXIoTEngine.login`。

绑定设备失败如何排查?

请检查 `familyId` 是否属于当前登录用户，设备绑定签名是否正确，以及设备是否已被其他家庭绑定。SDK 会在 `onError` 中返回具体错误码和错误信息。

下一步

完成基础接入后，您还可以继续接入以下能力：

- 绑定与分享
- 远程控制
- 实时监控
- 视频通话

远程控制

最近更新时间：2026-07-30 12:00:37

本文介绍腾讯云物联网（IoT）应用端 SDK 中设备管理（`TXIoTDeviceManager`）的远程控制能力：通过 `sendCommand` 向设备下发命令、通过 `getProperties` 查询设备当前属性。完成快速接入、登录 SDK 后，您需确保目标设备已完成 绑定与分享，取得 `deviceId` 后方可对其进行远程控制。

> **说明：**
> 本文仅介绍设备远程控制（`sendCommand` / `getProperties`）相关能力，不包含 `TXIoTMonitorSession` 和 `TXIoTCallSession` 等音视频功能。Android 和 iOS 接口语义基本一致，下文以选项卡形式分别给出关键调用方式。

前提条件

在远程控制设备前，请确保已完成以下准备工作：

- 已开通腾讯云物联网相关服务，并在控制台完成实例、应用和设备准备。可参见 开通服务 。
- 已在客户端工程中集成 IoT 应用端 SDK。
- 已通过业务后台生成登录签名，并调用 `TXIoTEngine` 完成登录。
- 设备已完成绑定或分享：仅当用户本人拥有该设备、或家庭成员拥有该设备、或用户被分享了该设备时，才具备远程控制的权限。若尚未完成设备绑定或分享，请先参考 绑定与分享 。

如果尚未完成 SDK 集成与登录，请先参考 Android 快速接入 或 iOS 快速接入 。

APP 操作过程

获取管理对象

设备管理能力通过 `TXIoTEngine` 获取。在 Android 和 iOS 上，登录成功后，均可通过 `TXIoTEngine` 单例实例调用 `getDeviceManager()` 获取设备管理对象。

> **说明：**
> 如果获取到的 `TXIoTDeviceManager` 为 `null`（iOS 为 `nil`），通常是因为 SDK 尚未登录或登录已过期，请先完成登录后再使用相关能力。

下发命令和查询属性

设备命令和设备属性均使用 JSON 字符串。`sendCommand` 用于向设备下发命令，`getProperties` 用于查询设备当前属性。

> **注意：**

> `jsonData` 的字段需要与设备物模型或业务约定保持一致。设备离线、设备不存在或没有权限时，接口会通过 `onError` 返回错误码和错误信息。

方法	说明
`sendCommand`	向设备下发命令，参数 `jsonData` 为设备命令 JSON 字符串。
`getProperties`	查询设备当前属性，返回设备属性 JSON 字符串。

Android

```
String jsonData = "{\"power_switch\":1}";
deviceManager.sendCommand(deviceId, jsonData, new
TXIoTEngineDef.TXIoTCallback<String>() {
    @Override
    public void onSuccess(String result) {
        // 命令下发成功，result 为服务端返回的 JSON 字符串。
    }

    @Override
    public void onError(TXIoTEngineDef.TXIoTErrorCode errorCode,
String errorMessage) {
        // 命令下发失败，请根据错误码和错误信息处理。
    }
});

deviceManager.getProperties(deviceId, new
TXIoTEngineDef.TXIoTCallback<String>() {
    @Override
    public void onSuccess(String result) {
        // 查询成功，result 为设备属性 JSON 字符串。
    }

    @Override
    public void onError(TXIoTEngineDef.TXIoTErrorCode errorCode,
String errorMessage) {
        // 查询属性失败，请根据错误码和错误信息处理。
    }
```

```
});
```

iOS

```
NSString *jsonData = @"{\"power_switch\":1}";
TXIoTCallback<NSString *> *commandCallback = [TXIoTCallback new];
commandCallback.onSuccess = ^(NSString *result) {
    // 命令下发成功，result 为服务端返回的 JSON 字符串。
};
commandCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 命令下发失败，请根据错误码和错误信息处理。
};
[deviceManager sendCommand:deviceId jsonData:jsonData
callback:commandCallback];

TXIoTCallback<NSString *> *propertyCallback = [TXIoTCallback new];
propertyCallback.onSuccess = ^(NSString *result) {
    // 查询成功，result 为设备属性 JSON 字符串。
};
propertyCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 查询属性失败，请根据错误码和错误信息处理。
};
[deviceManager getProperties:deviceId callback:propertyCallback];
```

设备端接收远程控制

APP 发送远程控制命令，对应设备端接收属性变更、设备端接收行为调用并回复结果。本节仅作简要介绍，详情可参见 物模型通信 。

步骤 1：设备初始化物模型模块

调用 `tc_iot_data_model_init` 设置回调。

```
static void on_receive_property_changed_cb(char *property_id,
```

```
                                                tc_iot_data_model_property_t
*data_model_property);

static int  on_receive_new_action_cb(char *action_id,
                                     tc_iot_data_model_action_t
*data_model_action);


tc_iot_data_model_callback_t callback;
memset(&callback, 0, sizeof(callback));
callback.on_receive_property_changed_cb =
on_receive_property_changed_cb;
callback.on_receive_new_action_cb       = on_receive_new_action_cb;

tc_iot_error_e dm_rc = tc_iot_data_model_init(callback);
if (dm_rc != TC_IOT_ERR_SUCCESS) {
    printf("tc_iot_data_model_init failed: %d\n", dm_rc);
}
```

步骤 2：设备接收属性变更

当 APP 发送控制命令修改了设备的某个属性时，SDK 会通过 `on_receive_property_changed_cb` 回调通知用户层。
根据回调参数 `tc_iot_data_model_property_t` 确定属性名称，属性数据类型，属性数据值。

```
static void on_receive_property_changed_cb(char *property_id,
                                            tc_iot_data_model_property_t
*data_model_property) {
    if (property_id == NULL || data_model_property == NULL) {
        return;
    }

    data_model_data_item_t *item = &data_model_property->property;
    printf("[demo] property changed: id=%s, type=%d\n", item->data_id,
(int)item->data_type);

    switch (item->data_type) {
        case DATA_MODEL_DATA_TYPE_BOOL:
            // 根据 item->data_value.value_bool 调整设备状态
            break;
```

```
        case DATA_MODEL_DATA_TYPE_INT:
            // 根据 item->data_value.value_int 调整设备状态
            break;
        case DATA_MODEL_DATA_TYPE_FLOAT:
            // 根据 item->data_value.value_float 调整设备状态
            break;
        case DATA_MODEL_DATA_TYPE_STRING:
            // 根据 item->data_value.value_string 调整设备状态
            break;
        case DATA_MODEL_DATA_TYPE_ENUM:
            // 根据 item->data_value.value_enum 调整设备状态
            break;
        case DATA_MODEL_DATA_TYPE_TIME:
            // 根据 item->data_value.value_time 调整设备状态
            break;
        default:
            break;
    }
}
```

> **说明：**
> on_receive_property_changed_cb 触发后，设备如果修改了属性值，可以调用 tc_iot_data_model_report_property 将最新的属性值上报云端。

步骤 3：设备接收行为调用

当 APP 发送控制命令调用设备的行为时，SDK 会通过 `on_receive_new_action_cb` 回调通知用户层。
回调返回值为 int：0 表示执行成功，非0表示执行失败。
如果控制台物模型行为定义了输出参数，回调中给 `action_output_data_list / action_output_data_num` 赋值，SDK 会把结果回传给云端。

```
static int on_receive_new_action_cb(char *action_id,
tc_iot_data_model_action_t *data_model_action) {
    if (action_id == NULL || data_model_action == NULL) {
        return -1;
    }
```

```
    printf("[demo] receive new action: id=%s, token=%s, timestamp=%u,
input_num=%d\n",
           action_id,
           data_model_action->token ? data_model_action->token : "",
           (unsigned)data_model_action->timestamp,
           data_model_action->action_input_data_num);

    // 1. 解析输入参数
    for (int i = 0; i < data_model_action->action_input_data_num; i++) {
        data_model_data_item_t *input = &data_model_action-
>action_input_data_list[i];
        printf("    input[%d]: id=%s, type=%d\n", i, input->data_id,
(int)input->data_type);
        // 根据业务需要读取 input->data_value ...
    }

    // 2. 执行行为逻辑（同步执行，避免在回调里做长时间阻塞）

    // 3. 填充输出参数（按云端定义的输出参数顺序和类型）
    int output_num = data_model_action->action_input_data_num;  // 示例:
与输入参数个数保持一致
    data_model_action->action_output_data_num = output_num;
    data_model_action->action_output_data_list = NULL;
    if (output_num > 0) {
        data_model_action->action_output_data_list =
            (data_model_data_item_t *)calloc(output_num,
sizeof(data_model_data_item_t));
        if (data_model_action->action_output_data_list == NULL) {
            return -2;  // 内存不足，回复云端失败
        }
        for (int i = 0; i < output_num; i++) {
            data_model_data_item_t *input  = &data_model_action-
>action_input_data_list[i];
            data_model_data_item_t *output = &data_model_action-
>action_output_data_list[i];

            output->data_id = input->data_id;       // 与云端约定好的输出参
数标识
            output->data_type = input->data_type;   // 与云端约定好的输出参
数类型
```

```
            if (input->data_type == DATA_MODEL_DATA_TYPE_INT) {
                output->data_value.value_int = input-
>data_value.value_int + 1;
            } else {
                output->data_value = input->data_value;
            }
        }
    }

    // 4. 返回 0：执行成功；返回非 0：执行失败，云端会收到对应错误码
    return 0;
}
```

说明：

- 如果行为没有输出参数，把 action_output_data_num 设置为 0、action_output_data_list 设置为 NULL 即可。
- 返回0表示执行成功，云端收到的 action_reply 消息中 code=0、status=success；返回非0时 code=返回值、status=action execution failed。

绑定与分享

最近更新时间：2026-07-30 12:00:38

本文介绍腾讯云物联网（IoT）应用端 SDK 中家庭管理（`TXIoTFamilyManager`）和设备管理（`TXIoTDeviceManager`）相关 API 的使用方法。完成快速接入并登录 SDK 后，您可以通过本文完成家庭创建、房间管理、成员邀请、设备绑定、设备列表查询、设备分享等功能。

前提条件

在调用本文 API 前，请确保已完成以下准备工作：

- 已开通腾讯云物联网相关服务，并在控制台完成实例、应用和设备的准备工作。可参见 开通服务 。
- 已在客户端工程中集成 IoT 应用端 SDK。
- 已通过业务后台生成登录签名，并调用 `TXIoTEngine` 完成登录。

如果尚未完成 SDK 集成与登录，请先参考 Android 快速接入 或 iOS 快速接入 。

获取管理对象

在 Android 和 iOS 上，登录成功后，均可通过 `TXIoTEngine` 单例调用 `getFamilyManager()` 和 `getDeviceManager()` 获取对应的管理对象。

> **说明：**
> 如果获取到的 `TXIoTFamilyManager` 或 `TXIoTDeviceManager` 为 `null`（iOS 为 `nil`），通常是因为 SDK 尚未登录或登录已过期，请先完成登录后再使用相关能力。

家庭管理

家庭是设备和成员管理的基础容器。设备绑定、设备列表查询、房间归属、家庭成员邀请等操作都需要使用 `familyId` 。

查询或创建家庭

客户端通常先查询家庭列表。如果当前用户没有家庭，可创建一个默认家庭。创建成功后，接口返回 TXIoTFamilyInfo 结构体，请从中取出 `familyId` 并保存，供后续绑定设备或查询设备使用。

方法	说明
getFamilyList	查询当前用户的家庭列表，返回 TXIoTFamilyInfo 列表。
createFamily	创建家庭，返回 TXIoTFamilyInfo（含 `familyId`）。

Android

```
// 获取家庭列表
familyManager.getFamilyList(new TXIoTCallback<List<TXIoTFamilyInfo>>
() {
    @Override
    public void onSuccess(List<TXIoTFamilyInfo> result) {
        if (result != null && !result.isEmpty()) {
            String familyId = result.get(0).familyId;
            // 使用 familyId 继续管理房间、成员和设备。
        }
    }

    @Override
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 查询家庭列表失败，请根据错误码和错误信息处理。
    }
});

// 创建家庭
familyManager.createFamily("我的家庭", new
TXIoTCallback<TXIoTFamilyInfo>() {
    @Override
    public void onSuccess(TXIoTFamilyInfo familyInfo) {
        String familyId = familyInfo.familyId;
        // 创建成功后，可继续创建房间或绑定设备。
    }

    @Override
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 创建家庭失败，请根据错误码和错误信息处理。
    }
});
```

iOS

```
// 查询家庭
TXIoTCallback<NSArray<TXIoTFamilyInfo *> *> *familyListCallback =
[TXIoTCallback new];
familyListCallback.onSuccess = ^(NSArray<TXIoTFamilyInfo *> *result)
{
    if (result.count > 0) {
        NSString *familyId = result.firstObject.familyId;
        // 使用 familyId 继续管理房间、成员和设备。
        return;
    }
};
familyListCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 查询家庭列表失败，请根据错误码和错误信息处理。
};
[familyManager getFamilyList:familyListCallback];

// 创建家庭
TXIoTCallback<TXIoTFamilyInfo *> *createCallback = [TXIoTCallback
new];
createCallback.onSuccess = ^(TXIoTFamilyInfo *familyInfo) {
    NSString *familyId = familyInfo.familyId;
    // 创建成功后，可继续创建房间或绑定设备。
};
createCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 创建家庭失败，请根据错误码和错误信息处理。
};
[familyManager createFamily:@"我的家庭" callback:createCallback];
```

管理房间

房间用于对家庭下的设备进行分组。**房间是可选的**：设备刚绑定时默认不属于任何房间，加入房间只是为了在 App 中对设备做分组展示，不影响设备的绑定状态和功能使用。创建房间后，您可以通过设备管理接口将设备加入或移出房间。

> ⚠ **注意：**

如果您的业务不需要对设备做分组，可以跳过房间相关接口，直接绑定设备并使用设备列表。

方法	说明
createRoom	在指定家庭下创建房间，返回 TXIoTRoomInfo（含 `roomId`）。

Android

```
familyManager.createRoom(familyId, "客厅", new
TXIoTCallback<TXIoTRoomInfo>() {
    @Override
    public void onSuccess(TXIoTRoomInfo roomInfo) {
        String roomId = roomInfo.roomId;
        // 后续可调用 addDeviceToRoom 将设备加入该房间。
    }

    @Override
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 创建房间失败，请根据错误码和错误信息处理。
    }
});
```

iOS

```
TXIoTCallback<TXIoTRoomInfo *> *roomCallback = [TXIoTCallback new];
roomCallback.onSuccess = ^(TXIoTRoomInfo *roomInfo) {
    NSString *roomId = roomInfo.roomId;
    // 后续可调用 addDeviceToRoom 将设备加入该房间。
};
roomCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 创建房间失败，请根据错误码和错误信息处理。
};
```

```
[familyManager createRoom:familyId name:@"客厅"
callback:roomCallback];
```

邀请成员加入家庭

家庭管理员可创建邀请 Token，并将邀请 Token 通过业务自有渠道发送给其他用户。被邀请用户登录 SDK 后，调用加入家庭接口即可成为家庭成员。

> ⚠ **注意：**
> 邀请 Token 属于敏感凭据，请通过可信渠道发送给目标用户，并避免在日志中明文输出。

方法	说明
createFamilyInviteToken	家庭管理员创建邀请 Token。
joinFamilyAsMember	被邀请用户使用邀请 Token 加入家庭。

Android

```
// 家庭管理员创建邀请 Token。
familyManager.createFamilyInviteToken(familyId, new
TXIoTCallback<String>() {
    @Override
    public void onSuccess(String inviteToken) {
        // 将 inviteToken 通过业务自有渠道发送给被邀请用户。
    }

    @Override
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 创建邀请 Token 失败，请根据错误码和错误信息处理。
    }
});

// 被邀请用户使用邀请 Token 加入家庭。
familyManager.joinFamilyAsMember(inviteToken, new
TXIoTCallback<Void>() {
```

```
    @Override
    public void onSuccess(Void result) {
        // 加入家庭成功。
    }

    @Override
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 加入家庭失败，请根据错误码和错误信息处理。
    }
});
```

iOS

```
// 家庭管理员创建邀请 Token。
TXIoTCallback<NSString *> *tokenCallback = [TXIoTCallback new];
tokenCallback.onSuccess = ^(NSString *inviteToken) {
    // 将 inviteToken 通过业务自有渠道发送给被邀请用户。
};
tokenCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 创建邀请 Token 失败，请根据错误码和错误信息处理。
};
[familyManager createFamilyInviteToken:familyId
callback:tokenCallback];

// 被邀请用户使用邀请 Token 加入家庭。
TXIoTVoidCallback *joinCallback = [TXIoTVoidCallback new];
joinCallback.onSuccess = ^{
    // 加入家庭成功。
};
joinCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 加入家庭失败，请根据错误码和错误信息处理。
};
```

```
[familyManager joinFamilyAsMember:inviteToken
callback:joinCallback];
```

设备管理

`TXIoTDeviceManager` 用于完成设备绑定、解绑、列表查询、房间归属、设备分享和别名修改。

设备上下线通知

SDK 登录后，可通过 `TXIoTEngine` 的推送回调 `onReceivePushMessage` 接收设备在线、离线状态变更。收到通知后，您可以根据 `deviceId` 刷新本地设备列表或设备详情。

Android

```
iotEngine.addListener(new TXIoTEngineListener() {
    @Override
    public void onReceivePushMessage(TXIoTPushMessage pushMessage) {
        if (pushMessage.type == TXIoTPushMessageType.STATUS_CHANGE)
{
            TXIoTDeviceId deviceId = pushMessage.deviceId;
            if (pushMessage.subType ==
TXIoTPushMessageSubType.ONLINE) {
                // 设备上线。
            } else if (pushMessage.subType ==
TXIoTPushMessageSubType.OFFLINE) {
                // 设备离线。
            }
        }
    }
});
```

iOS

```
- (void)onReceivePushMessage:(TXIoTPushMessage *)pushMessage {
    if (pushMessage.type == TXIoTPushMessageTypeStatusChange) {
```

```
        TXIoTDeviceId *deviceId = pushMessage.deviceId;
        if (pushMessage.subType == TXIoTPushMessageSubTypeOnline) {
            // 设备上线。
        } else if (pushMessage.subType ==
TXIoTPushMessageSubTypeOffline) {
            // 设备离线。
        }
    }
}
```

绑定与解绑设备

设备绑定需要提供 `familyId` 和 `deviceBindSignature` 。绑定成功后，SDK 返回 TXIoTDeviceInfo ，其中包含设备标识、所属家庭 ID、所属房间 ID、设备状态等信息。当用户不再使用设备或需要将设备迁移到其他家庭时，可调用解绑接口，解绑后该设备会从指定家庭中移除。

> **说明：**
> `deviceBindSignature` 由业务后台生成，用于校验设备绑定权限，请通过您的业务服务端获取后传入。

方法	说明
bindDevice	绑定设备，需 `familyId` 与 `deviceBindSignature` ，返回 TXIoTDeviceInfo 。
unbindDevice	解绑设备，将其从指定家庭中移除， `deviceId` 为 TXIoTDeviceId 。

Android

```
// 绑定设备。
deviceManager.bindDevice(familyId, deviceBindSignature, new
TXIoTCallback<TXIoTDeviceInfo>() {
    @Override
    public void onSuccess(TXIoTDeviceInfo deviceInfo) {
        TXIoTDeviceId deviceId = deviceInfo.deviceId;
        // 绑定成功后，可查询设备列表、下发命令或查询属性。
    }
```

```
    @Override
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 绑定失败，请检查设备绑定签名是否正确、是否过期，以及设备是否已绑定。
    }
});

// 解绑设备。
deviceManager.unbindDevice(familyId, deviceId, new
TXIoTCallback<Void>() {
    @Override
    public void onSuccess(Void result) {
        // 解绑成功。
    }

    @Override
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 解绑失败，请根据错误码和错误信息处理。
    }
});
```

iOS

```
// 绑定设备。
TXIoTCallback<TXIoTDeviceInfo *> *bindCallback = [TXIoTCallback
new];
bindCallback.onSuccess = ^(TXIoTDeviceInfo *deviceInfo) {
    TXIoTDeviceId *deviceId = deviceInfo.deviceId;
    // 绑定成功后，可查询设备列表、下发命令或查询属性。
};
bindCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 绑定失败，请检查设备绑定签名是否正确、是否过期，以及设备是否已绑定。
};
```

```
[deviceManager bindDevice:familyId
deviceBindSignature:deviceBindSignature callback:bindCallback];

// 解绑设备。
TXIoTVoidCallback *unbindCallback = [TXIoTVoidCallback new];
unbindCallback.onSuccess = ^{
    // 解绑成功。
};
unbindCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 解绑失败，请根据错误码和错误信息处理。
};
[deviceManager unbindDevice:familyId deviceId:deviceId
callback:unbindCallback];
```

查询设备列表

设备列表接口为分页接口。首次查询时传入空字符串，返回结果中的 `nextPageToken` 为空字符串时表示没有更多数据。

方法	说明
getDeviceList	分页查询家庭下的设备列表，返回 TXIoTPageResult<TXIoTDeviceInfo>。

Android

```
deviceManager.getDeviceList(familyId, "", new
TXIoTCallback<TXIoTPageResult<TXIoTDeviceInfo>>() {
    @Override
    public void onSuccess(TXIoTPageResult<TXIoTDeviceInfo> result) {
        List<TXIoTDeviceInfo> deviceList = result.dataList;
        String nextPageToken = result.nextPageToken;
        // nextPageToken 为 "" 时表示没有更多数据，否则可传入下一次
getDeviceList 继续分页查询。
    }
```

```
    @Override
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 查询设备列表失败，请根据错误码和错误信息处理。
    }
});
```

iOS

```
TXIoTCallback<TXIoTPageResult<TXIoTDeviceInfo *> *> *listCallback =
[TXIoTCallback new];
listCallback.onSuccess = ^(TXIoTPageResult<TXIoTDeviceInfo *>
*result) {
    NSArray<TXIoTDeviceInfo *> *deviceList = result.dataList;
    NSString *nextPageToken = result.nextPageToken;
    // nextPageToken 为 @"" 时表示没有更多数据，否则可传入下一次
getDeviceList 继续分页查询。
};
listCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 查询设备列表失败，请根据错误码和错误信息处理。
};
[deviceManager getDeviceList:familyId nextPageToken:@""
callback:listCallback];
```

将设备加入房间

绑定设备后，可调用 addDeviceToRoom 将设备加入指定房间。若需要从房间移除设备，可调用 removeDeviceFromRoom 。房间仅用于分组，因此将设备加入房间是可选操作。

方法	说明
addDeviceToRoom	将设备加入指定房间，`deviceId` 为 TXIoTDeviceId 类型。
removeDeviceFromRoom	将设备从房间移除，`deviceId` 为 TXIoTDeviceId 类型。

Android

```
deviceManager.addDeviceToRoom(deviceId, familyId, roomId, new
TXIoTCallback<Void>() {
    @Override
    public void onSuccess(Void result) {
        // 添加设备到房间成功。
    }

    @Override
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 添加设备到房间失败，请根据错误码和错误信息处理。
    }
});
```

iOS

```
TXIoTVoidCallback *addCallback = [TXIoTVoidCallback new];
addCallback.onSuccess = ^{
    // 添加设备到房间成功。
};
addCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 添加设备到房间失败，请根据错误码和错误信息处理。
};
[deviceManager addDeviceToRoom:deviceId familyId:familyId
roomId:roomId callback:addCallback];
```

修改设备别名

设备别名用于在 App 中展示设备名称，不会修改设备的产品 ID 或设备名称。

方法	说明
modifyAliasName	修改设备别名，`deviceId` 为 TXIoTDeviceId 类型。别名仅 App 展示名称，不影响产品 ID 或设备名。

Android

```
deviceManager.modifyAliasName(deviceId, "客厅摄像头", new
TXIoTCallback<Void>() {
    @Override
    public void onSuccess(Void result) {
        // 修改设备别名成功。
    }

    @Override
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 修改设备别名失败，请根据错误码和错误信息处理。
    }
});
```

iOS

```
TXIoTVoidCallback *aliasCallback = [TXIoTVoidCallback new];
aliasCallback.onSuccess = ^{
    // 修改设备别名成功。
};
aliasCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 修改设备别名失败，请根据错误码和错误信息处理。
};
[deviceManager modifyAliasName:deviceId aliasName:@"客厅摄像头"
callback:aliasCallback];
```

设备分享

设备分享适用于设备拥有者将单个设备授权给其他用户使用的场景。设备拥有者创建分享 Token，被分享用户使用该 Token 绑定共享设备。

> ⚠ **注意：**
> 设备分享 Token 与家庭邀请 Token 不同。家庭邀请 Token 用于加入家庭，设备分享 Token 仅用于绑定指定共享设备。

分享设备与接受分享

设备拥有者调用 createDeviceSharingToken 生成分享 Token，并通过业务自有渠道发送给被分享用户。被分享用户登录 SDK 后，调用 bindDeviceSharedWithMe 并传入该 Token，即可绑定这台共享设备。此外，被分享用户还可以调用 getDeviceListSharedWithMe 分页查询所有被分享给自己的设备列表。

方法	说明
createDeviceSharingToken	设备拥有者生成设备分享 Token，`deviceId` 为 TXIoTDeviceId 类型。
bindDeviceSharedWithMe	被分享用户使用 Token 绑定共享设备，`deviceId` 为 TXIoTDeviceId 类型。
getDeviceListSharedWithMe	查询所有别人分享给我的设备列表。

Android

```
// 设备拥有者：创建设备分享 Token 并发送给被分享用户。
deviceManager.createDeviceSharingToken(familyId, deviceId, new
TXIoTCallback<String>() {
    @Override
    public void onSuccess(String shareToken) {
        // 将 shareToken 通过业务自有渠道发送给被分享用户。
    }

    @Override
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 创建设备分享 Token 失败，请根据错误码和错误信息处理。
```

```
    }
});

// 被分享用户：使用分享 Token 绑定共享设备。
deviceManager.bindDeviceSharedWithMe(deviceId, shareToken, new
TXIoTCallback<Void>() {
    @Override
    public void onSuccess(Void result) {
        // 绑定共享设备成功。
    }

    @Override
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 绑定共享设备失败，请根据错误码和错误信息处理。
    }
});
```

iOS

```
// 设备拥有者：创建设备分享 Token 并发送给被分享用户。
TXIoTCallback<NSString *> *shareCallback = [TXIoTCallback new];
shareCallback.onSuccess = ^(NSString *shareToken) {
    // 将 shareToken 通过业务自有渠道发送给被分享用户。
};
shareCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 创建设备分享 Token 失败，请根据错误码和错误信息处理。
};
[deviceManager createDeviceSharingToken:familyId deviceId:deviceId
callback:shareCallback];

// 被分享用户：使用分享 Token 绑定共享设备。
TXIoTVoidCallback *bindSharedCallback = [TXIoTVoidCallback new];
bindSharedCallback.onSuccess = ^{
    // 绑定共享设备成功。
};
```

```
bindSharedCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 绑定共享设备失败，请根据错误码和错误信息处理。
};
[deviceManager bindDeviceSharedWithMe:deviceId shareToken:shareToken
callback:bindSharedCallback];
```

管理共享关系

设备拥有者可以查询设备已分享用户列表，也可以移除指定共享用户。被分享用户可查询共享给自己的设备列表，或取消绑定共享设备。

方法	说明
getDeviceSharedUsers	设备拥有者查询已分享给哪些用户，`deviceId` 为 TXIoTDeviceId，异步回调 TXIoTUserInfo 列表。
removeDeviceSharedUser	设备拥有者移除指定共享用户，`deviceId` 为 TXIoTDeviceId，被移除的用户将无法再访问该设备。
getDeviceListSharedWithMe	查询别人分享给我的设备列表，异步回调 TXIoTPageResult<TXIoTDeviceInfo>，支持分页。首次查询传空字符串，`nextPageToken` 为空字符串时表示没有更多数据。
unbindDeviceSharedWithMe	取消绑定别人分享给我的设备，`deviceId` 为 TXIoTDeviceId，取消绑定后不再拥有该设备的访问权限。

Android

```
// 设备拥有者：查询已分享用户列表
deviceManager.getDeviceSharedUsers(deviceId, new
TXIoTCallback<List<TXIoTUserInfo>>() {
    @Override
    public void onSuccess(List<TXIoTUserInfo> userList) {
        // 查询成功，userList 为已分享的用户列表。
    }

    @Override
```

```
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 查询失败，请根据错误码和错误信息处理。
    }
});

// 设备拥有者：移除指定共享用户
deviceManager.removeDeviceSharedUser(deviceId, userId, new
TXIoTCallback<Void>() {
    @Override
    public void onSuccess(Void result) {
        // 移除成功。
    }

    @Override
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 移除失败，请根据错误码和错误信息处理。
    }
});

// 被分享用户：查询共享设备列表（首次传空字符串）
deviceManager.getDeviceListSharedWithMe("", new
TXIoTCallback<TXIoTPageResult<TXIoTDeviceInfo>>() {
    @Override
    public void onSuccess(TXIoTPageResult<TXIoTDeviceInfo>
pageResult) {
        // 查询成功，pageResult.data 为设备列表。
pageResult.nextPageToken 为空字符串时表示没有更多数据。
    }

    @Override
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 查询失败，请根据错误码和错误信息处理。
    }
});

// 被分享用户：取消绑定共享设备
```

```
deviceManager.unbindDeviceSharedWithMe(deviceId, new
TXIoTCallback<Void>() {
    @Override
    public void onSuccess(Void result) {
        // 取消成功。
    }

    @Override
    public void onError(TXIoTErrorCode errorCode, String
errorMessage) {
        // 取消失败，请根据错误码和错误信息处理。
    }
});
```

iOS

```
// 设备拥有者：查询已分享用户列表
TXIoTCallback<NSArray<TXIoTUserInfo*>*> *sharedUsersCallback =
[TXIoTCallback new];
sharedUsersCallback.onSuccess = ^(NSArray<TXIoTUserInfo*> *userList)
{
    // 查询成功，userList 为已分享的用户列表。
};
sharedUsersCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 查询失败，请根据错误码和错误信息处理。
};
[deviceManager getDeviceSharedUsers:deviceId
callback:sharedUsersCallback];

// 设备拥有者：移除指定共享用户
TXIoTVoidCallback *removeCallback = [TXIoTVoidCallback new];
removeCallback.onSuccess = ^{
    // 移除成功。
};
removeCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
```

```
    // 移除失败，请根据错误码和错误信息处理。
};
[deviceManager removeDeviceSharedUser:deviceId userId:userId
callback:removeCallback];

// 被分享用户：查询共享设备列表（首次传空字符串）
TXIoTCallback<TXIoTPageResult<TXIoTDeviceInfo*>*> *sharedDevCallback
= [TXIoTCallback new];
sharedDevCallback.onSuccess = ^(TXIoTPageResult<TXIoTDeviceInfo*>
*pageResult) {
    // 查询成功，pageResult.data 为设备列表。pageResult.nextPageToken 为
空字符串时表示没有更多数据。
};
sharedDevCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 查询失败，请根据错误码和错误信息处理。
};
[deviceManager getDeviceListSharedWithMe:@""
callback:sharedDevCallback];

// 被分享用户：取消绑定共享设备
TXIoTVoidCallback *unbindCallback = [TXIoTVoidCallback new];
unbindCallback.onSuccess = ^{
    // 取消成功。
};
unbindCallback.onError = ^(TXIoTErrorCode errorCode, NSString
*errorMessage) {
    // 取消失败，请根据错误码和错误信息处理。
};
[deviceManager unbindDeviceSharedWithMe:deviceId
callback:unbindCallback];
```

错误处理

家庭和设备管理常见错误码如下，完整错误码定义见 TXIoTErrorCode 。

Android

错误码	说明	建议处理方式
`ERR_INVALID_PARAMETER`	参数不合法。	检查 `familyId`、`deviceId`、Token 或 JSON 字符串是否为空或格式错误。
`ERR_INVALID_ACCESS_TOKEN`	登录凭证无效。	重新登录 SDK。
`ERR_RATE_LIMITED`	请求被限频。	降低调用频率后重试。
`ERR_UNAUTHORIZED_OPERATION`	当前用户无权限。	检查当前用户是否为家庭管理员或是否具备设备操作权限。
`ERR_DEVICE_BOUND`	设备已绑定。	检查设备是否已绑定到当前或其他家庭。
`ERR_BIND_TOKEN_NOT_EXIST`	绑定 Token 不存在。	重新获取设备绑定签名。
`ERR_BIND_TOKEN_IS_EXPIRED`	绑定 Token 已过期。	重新获取设备绑定签名后再次绑定。
`ERR_BIND_TOKEN_NO_PAIR_WITH_DEVICE`	绑定 Token 与设备不匹配。	确认设备绑定签名对应的设备与当前设备一致。
`ERR_FAMILY_DEVICE_LIMIT_EXCEEDED`	家庭设备数量达到上限。	清理无用设备或调整家庭设备数量。
`ERR_CAN_NOT_BIND_SAME_FAMILY`	不能重复绑定到同一家庭。	避免重复绑定同一设备。
`ERR_FAMILY_NOT_EXIST`	家庭不存在。	重新查询家庭列表，确认 `familyId` 是否有效。
`ERR_PRODUCT_NOT_EXIST`	产品不存在。	检查 `productId` 是否正确。
`ERR_DEVICE_NOT_EXIST`	设备不存在。	检查 `deviceName` 和 `productId` 是否正确。
`ERR_DEVICE_OFFLINE`	设备离线。	引导用户检查设备网络状态后重试。

iOS

错误码	说明	建议处理方式
`TXIoTErrorCodeInvalidParameter`	参数不合法。	检查 `familyId`、`deviceId`、Token 或 JSON 字符串是否为空或格式错误。
`TXIoTErrorCodeInvalidAccessToken`	登录凭证无效。	重新登录 SDK。
`TXIoTErrorCodeRateLimited`	请求被限频。	降低调用频率后重试。
`TXIoTErrorCodeUnauthorizedOperation`	当前用户无权限。	检查当前用户是否为家庭管理员或是否具备设备操作权限。
`TXIoTErrorCodeDeviceBound`	设备已绑定。	检查设备是否已绑定到当前或其他家庭。
`TXIoTErrorCodeBindTokenNotExist`	绑定 Token 不存在。	重新获取设备绑定签名。
`TXIoTErrorCodeBindTokenIsExpired`	绑定 Token 已过期。	重新获取设备绑定签名后再次绑定。
`TXIoTErrorCodeBindTokenNoPairWithDevice`	绑定 Token 与设备不匹配。	确认设备绑定签名对应的设备与当前设备一致。
`TXIoTErrorCodeFamilyDeviceLimitExceeded`	家庭设备数量达到上限。	清理无用设备或调整家庭设备数量。
`TXIoTErrorCodeCanNotBindSameFamily`	不能重复绑定到同一家庭。	避免重复绑定同一设备。
`TXIoTErrorCodeFamilyNotExist`	家庭不存在。	重新查询家庭列表，确认 `familyId` 是否有效。
`TXIoTErrorCodeProductNotExist`	产品不存在。	检查 `productId` 是否正确。
`TXIoTErrorCodeDeviceNotExist`	设备不存在。	检查 `deviceName` 和 `productId` 是否正确。

`TXIoTErrorCodeDeviceOffline`	设备离线。	引导用户检查设备网络状态后重试。