物联网开发平台

高级功能

腾讯云

文档目录

高级功能
设备认证
设备认证方式概览

最近更新时间：2026-09-11 16:05:33

本文将介绍物联网开发平台支持的设备密钥认证和设备证书认证，帮助您了解两种认证方式的凭证组成、工作机制及适用场景，并根据设备资源、安全要求和凭证管理能力选择合适的认证方式。

认证方式介绍

设备连接物联网开发平台前，需要完成身份认证。平台通过产品 ID、设备名称及设备凭证验证设备身份，认证通过后，设备才能上线并进行消息通信。

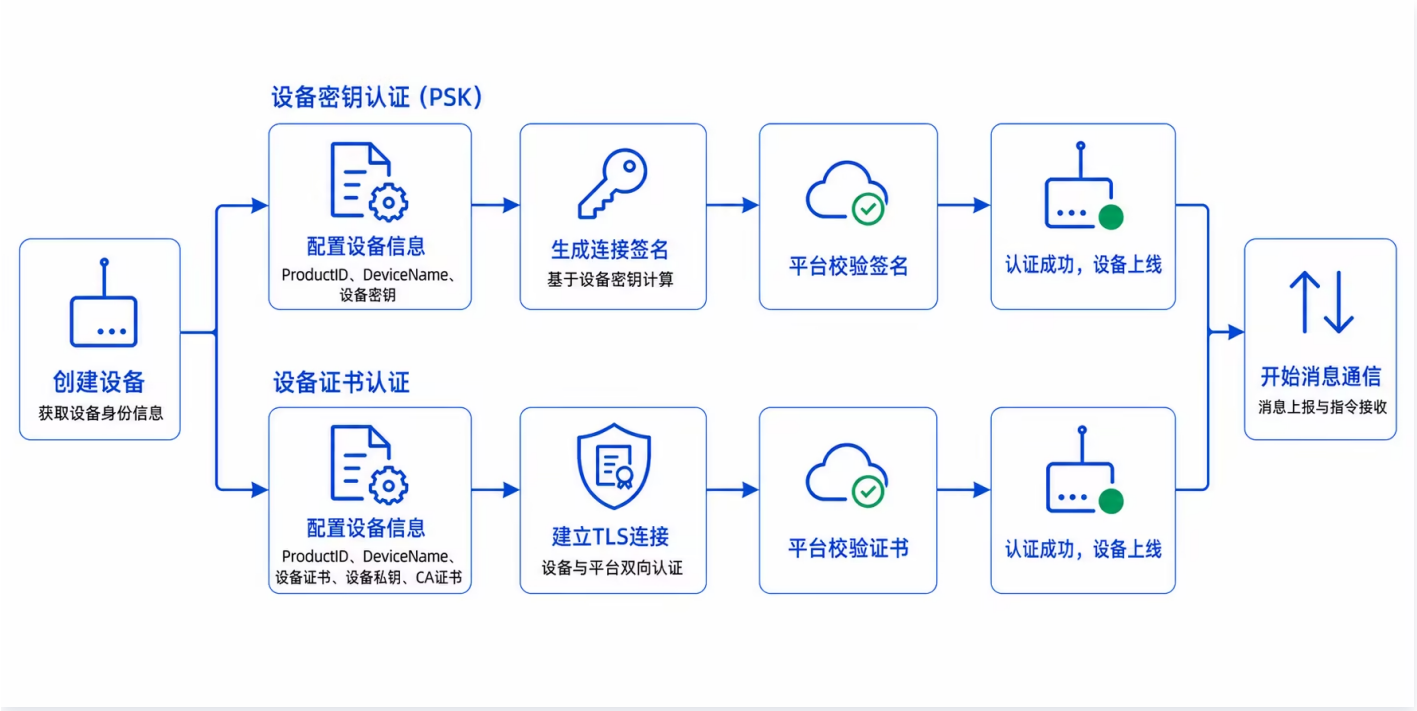
物联网开发平台支持以下两种设备认证方式：

- **设备密钥认证（PSK）**：设备使用独立的设备密钥生成连接签名，平台通过校验签名确认设备身份。
- **设备证书认证**：设备使用独立的设备证书和设备私钥完成身份认证，平台与设备通过证书建立双向可信连接。

认证方式在创建产品时确定，同一产品下的设备使用该产品配置的认证方式。

认证流程

两种认证方式的基本流程如下：



设备认证流程图

两种认证方式的主要区别在于设备使用的凭证及身份校验机制不同。

认证方式对比

对比项	设备密钥认证（PSK）	设备证书认证
设备凭证	设备密钥（DeviceSecret/PSK）。	设备证书和设备私钥。
认证机制	使用设备密钥生成 HMAC 签名。	使用设备证书和私钥完成双向认证。
设备侧准备	ProductID、DeviceName、设备密钥。	ProductID、DeviceName、设备证书、设备私钥及 CA 证书。
实现复杂度	相对较低。	相对较高。
设备资源要求	相对较低。	需要支持证书和私钥的加载及运算。
凭证保护重点	防止设备密钥泄露。	防止设备私钥泄露。
适用场景	资源受限设备、快速接入及具备密钥安全存储能力的设备。	对设备身份安全要求较高，且具备证书和私钥安全存储能力的设备。

> **说明：**
> CA 证书不属于设备身份凭证。设备证书和设备私钥用于证明设备身份，CA 证书用于验证物联网平台服务端身份。

如何选择认证方式

选择认证方式时，建议综合考虑设备资源、安全要求、生产方式及凭证管理能力。

选择设备密钥认证

以下场景可考虑使用设备密钥认证：

- 设备计算或存储资源有限。
- 希望降低设备侧接入复杂度。
- 能够为每台设备安全保存独立设备密钥。
- 现有产线具备设备密钥写入或首次获取能力。

设备密钥属于敏感凭证，不应写入日志、公开代码或由多台设备共用。
前往 设备密钥认证，了解认证参数、连接参数生成方法及设备上线流程。

选择设备证书认证

以下场景可考虑使用设备证书认证：

- 业务对设备身份安全有较高要求。
- 设备能够完成证书和私钥相关运算。
- 设备具备安全芯片或其他私钥安全存储能力。
- 企业具备证书及私钥的生产写入和生命周期管理能力。

设备私钥必须妥善保存。设备证书可以用于标识设备身份，但不能替代对应的设备私钥。

> **说明：**
> 证书认证并非适用于所有生产场景。若设备缺少私钥安全存储能力，仍可能存在凭证泄露风险，应结合实际硬件和生产条件进行选择。

前往 设备证书认证 ，了解认证参数、连接参数生成方法及设备上线流程。

设备密钥认证

最近更新时间：2026-09-11 16:05:33

本文将介绍设备如何使用 ProductID、DeviceName 和设备密钥生成 MQTT 连接参数，并完成设备身份认证和上线。

功能介绍

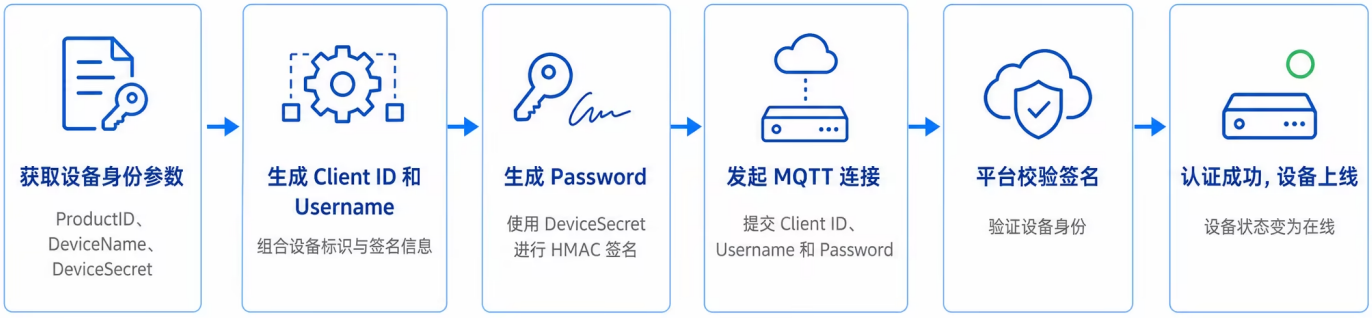
设备密钥认证使用平台为每台设备生成的独立设备密钥（DeviceSecret）验证设备身份。设备使用 DeviceSecret 对 MQTT Username 进行 HMAC 签名，并将签名结果组成 Password；平台校验通过后，设备即可上线。

> ⓘ **注意：**
> 本文中的 DeviceSecret 也称为 PSK。DeviceSecret 属于敏感凭证，请妥善保管。

如果尚未确定使用设备密钥认证还是设备证书认证，请参见 认证方式概览 。

认证流程

设备密钥认证流程参见下图：



认证参数

如需在控制台查看并复制设备身份参数及 MQTT 连接参数，请参见 获取设备三元组 。

设备身份参数（设备三元组）

参数	参数来源	说明	是否保密
ProductID	产品详情页或设备详情页。	标识设备所属产品。	否
DeviceName	设备详情页。	标识产品下的具体设备。	否

DeviceSecret	设备详情页或动态注册。	用于生成 MQTT 认证签名。	是

> ⓘ **注意：**
> ProductSecret 是产品级密钥，主要用于动态注册等产品级操作，不能代替 DeviceSecret 进行设备 MQTT 认证。

MQTT 连接参数

参数	取值或生成方式	说明
Broker Address	• 广州：`${ProductID}.iotcloud.tencentdevices.com` • 曼谷：`${ProductID}.ap-bangkok.iothub.tencentdevices.com`	MQTT 服务器地址。
Broker Port	• `1883`（通用） • `8883`（`TLS+PSK` 加密接入）	密钥认证设备接入端口。
Client ID	`${ProductID}${DeviceName}`	ProductID 与 DeviceName 直接拼接。
Username	`${ClientID};${sdkappid};${connid};${expiry}`	参与设备签名计算。
Password	`${token};${签名算法}`	使用 DeviceSecret 生成。
KeepAlive	0～900秒	MQTT 连接保活时间。

`${}` 表示变量，不是实际的拼接字符。连接域名和端口应以设备所属地域及控制台当前信息为准。
各扩展字段含义如下：

字段	说明
sdkappid	固定填写 `12010126` 。
connid	客户端生成的随机字符串，用于区分连接。
expiry	签名过期时间，使用 Unix 时间戳，单位为秒。
token	使用 DeviceSecret 对完整 Username 进行 HMAC 计算得到的摘要。

签名算法	支持 `hmacsha256` 或 `hmacsha1` ，建议使用 `hmacsha256` 。

使用 Python 连接平台

本节使用 Python 脚本连接物联网开发平台。当脚本返回认证成功且控制台设备状态显示为在线时，即表示设备密钥认证接入完成。
本示例仅验证设备认证和上线，不包含 Topic 发布、订阅及物模型消息通信。

前提条件

开始接入前，请确认已完成以下准备：

- 已创建认证方式为**密钥认证**的产品。
- 已在产品下**创建设备**。
- 已获得设备的 **MQTT 连接参数**。
- **本地环境已安装 Python 3**。
- 本地网络能够访问平台 MQTT 接入域名的 1883 端口。

如果需要设备在产测或首次启动阶段获取 DeviceSecret，请参见 密钥认证设备动态注册 。

获取设备连接参数

在控制台 获取设备三元组 并复制设备身份参数及 MQTT 连接参数。

控制台字段	脚本配置项
MQTT 服务器地址	Broker Address
端口号	Broker Port
Client ID	Client ID
Username	Username
Password	Password

准备运行环境

建议在虚拟环境中安装依赖和运行代码。
安装 Paho MQTT 客户端：

```
# Windows PowerShell
python -m pip install "paho-mqtt>=2.1,<3.0"

# Linux && macOS Shell
```

```
python3 -m pip install "paho-mqtt>=2.1,<3.0"
```

Paho MQTT 是 Python MQTT 客户端库。本示例使用 MQTT 3.1.1及 Callback API VERSION2。
安装完成后，可执行以下命令确认版本：

```
# Windows PowerShell
python -m pip show paho-mqtt

# Linux && macOS Shell
python3 -m pip show paho-mqtt

# 输出中的 Version 应为 2.1.0。
```

创建连接脚本

将以下代码保存为 `mqtt_connect.py`：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import getpass
import sys
from importlib.metadata import PackageNotFoundError, version

try:
    import paho.mqtt.client as mqtt
except ImportError:
    mqtt = None


def required_input(prompt):
    """读取必填参数并清除首尾空格。"""
    value = input(prompt).strip()
    if not value:
        raise ValueError("输入内容不能为空")
    return value


def get_port():
```

```
    """读取并校验MQTT端口号。"""
    port_text = input(
        "端口号（直接回车使用1883）："
    ).strip()

    if not port_text:
        return 1883

    try:
        port = int(port_text)
    except ValueError as exc:
        raise ValueError("端口号必须是整数") from exc

    if not 1 <= port <= 65535:
        raise ValueError("端口号必须在1～65535范围内")

    return port


def check_paho_version():
    """检查Paho MQTT是否安装且支持Callback API VERSION2。"""
    install_command = (
        f'"{sys.executable}" -m pip install "paho-mqtt==2.1.0"'
    )

    if mqtt is None:
        raise ImportError(
            "未安装Paho MQTT客户端。\n"
            f"请使用当前Python环境执行：\n{install_command}"
        )

    try:
        installed_version = version("paho-mqtt")
    except PackageNotFoundError:
        installed_version = "未知"

    if not hasattr(mqtt, "CallbackAPIVersion"):
        raise RuntimeError(
            f"当前Paho MQTT版本为{installed_version}，"
            "不支持Callback API VERSION2。\n"
```

```
            f"请使用当前Python环境执行：\n{install_command}"
        )

    print(f"Paho MQTT版本：{installed_version}")


def main():
    check_paho_version()

    print("\n请输入设备详情页中的MQTT连接参数。")
    print("输入内容仅在本次程序运行期间使用，不会保存到文件。\n")

    broker = required_input("MQTT服务器地址：")

    if "://" in broker:
        raise ValueError(
            "MQTT服务器地址不能包含协议前缀，"
            "请勿输入tcp://或mqtt://"
        )

    if any(char in broker for char in "${}"):
        raise ValueError(
            "MQTT服务器地址不能包含${}，"
            "请将${ProductID}替换为真实ProductID"
        )

    port = get_port()
    client_id = required_input("ClientId：")
    username = required_input("Username：")

    # Password输入时不会显示在终端中。
    password = getpass.getpass(
        "Password（输入内容不会显示）："
    ).strip()

    if not password:
        raise ValueError("Password不能为空")

    def on_connect(
        client,
```

```
        userdata,
        connect_flags,
        reason_code,
        properties,
    ):
        if reason_code == 0:
            print("\n认证成功，设备已连接物联网开发平台。")
            print(f"MQTT服务器：{broker}")
            print(f"端口号：{port}")
            print("请前往控制台查看设备在线状态和上下线日志。")
        else:
            print(
                f"\nMQTT连接失败，返回码：{reason_code}。"
                "\n请检查ClientId、Username、Password及设备状态。"
            )
            client.disconnect()

    def on_connect_fail(client, userdata):
        print(
            "\n网络连接失败，请检查以下内容："
            f"\n- MQTT服务器地址：{broker}"
            f"\n- 端口号：{port}"
            "\n- DNS解析及网络访问状态"
        )

    def on_disconnect(
        client,
        userdata,
        disconnect_flags,
        reason_code,
        properties,
    ):
        if reason_code != 0:
            print(f"\n连接异常断开，返回码：{reason_code}")

    client = mqtt.Client(
        callback_api_version=mqtt.CallbackAPIVersion.VERSION2,
        client_id=client_id,
        clean_session=True,
        protocol=mqtt.MQTTv311,
```

```
    )

    client.username_pw_set(
        username=username,
        password=password,
    )

    client.on_connect = on_connect
    client.on_connect_fail = on_connect_fail
    client.on_disconnect = on_disconnect

    # 不输出Username和Password等认证信息。
    print("\n连接配置：")
    print(f"MQTT服务器：{broker}")
    print(f"端口号：{port}")
    print(f"ClientId：{client_id}")
    print("\n正在建立MQTT连接。")
    print("程序将保持连接，按Ctrl+C退出。")

    client.connect(
        host=broker,
        port=port,
        keepalive=60,
    )

    try:
        client.loop_forever()
    finally:
        client.disconnect()


if __name__ == "__main__":
    try:
        sys.exit(main())
    except KeyboardInterrupt:
        print("\n用户终止连接。")
        sys.exit(0)
    except Exception as exc:
        print(f"\n运行失败：{exc}")
```

```
        sys.exit(1)
```

运行连接脚本

执行：

```
# Windows PowerShell
python mqtt_connect.py

# Linux && macOS Shell
python3 mqtt_connect.py
```

```
# 运行成功后依次输入
MQTT服务器地址：
端口号（直接回车使用1883）：
ClientId：
Username：
Password（输入内容不会显示）：
```

连接成功时，程序输出类似：

```
正在连接：******.iotcloud.tencentdevices.com:1883
ClientId：******
程序将保持连接，按Ctrl+C退出。
认证成功，设备已连接物联网开发平台。
请前往控制台查看设备在线状态。
```

程序会持续运行并保持 MQTT 连接。需要断开连接时，按下 `Ctrl+C` 。

验证设备上线

设备发起 MQTT 连接后，如需在控制台确认设备状态及查看上下线记录，请参见 设备上线和下线 。

问题排查

网络连接失败

检查以下内容：

- Broker Address 是否与设备详情页一致。
- 端口号是否正确。

- 本地 DNS 是否能够解析 MQTT 服务器域名。
- 防火墙或网络策略是否限制目标域名和端口。
- 相关域名是否已加入白名单。

认证失败

检查以下内容：

- ClientId、Username 和 Password 是否属于同一设备。
- 复制参数时是否包含多余空格或换行。
- Username 中的 expiry 是否已过期。
- 是否使用了重新生成前的旧连接参数。
- Broker 中的 ProductID 是否与设备所属产品一致。

出现 HMAC sign error

如果连接参数由设备自主生成，请检查：

- ProductID 和 DeviceName 是否正确。
- DeviceSecret 是否属于当前设备。
- DeviceSecret 是否完成 Base64解码。
- 参与签名的 Username 是否与连接时使用的 Username 完全一致。
- HMAC 算法是否与 Password 中的算法名称一致。
- expiry 是否晚于当前设备时间。

脚本连接成功，但设备未显示在线

检查：

- 是否进入了正确的实例和设备。
- ClientId 是否对应当前 DeviceName。
- 脚本是否仍在运行。
- 上下线日志中是否存在新的连接记录。
- 是否有另一个客户端使用相同 ClientId 建立连接。

设备证书认证

最近更新时间：2026-09-11 16:05:33

本文将介绍设备如何使用 ProductID、DeviceName、设备证书和设备私钥建立 MQTT 安全连接，并完成设备身份认证和上线。

功能介绍

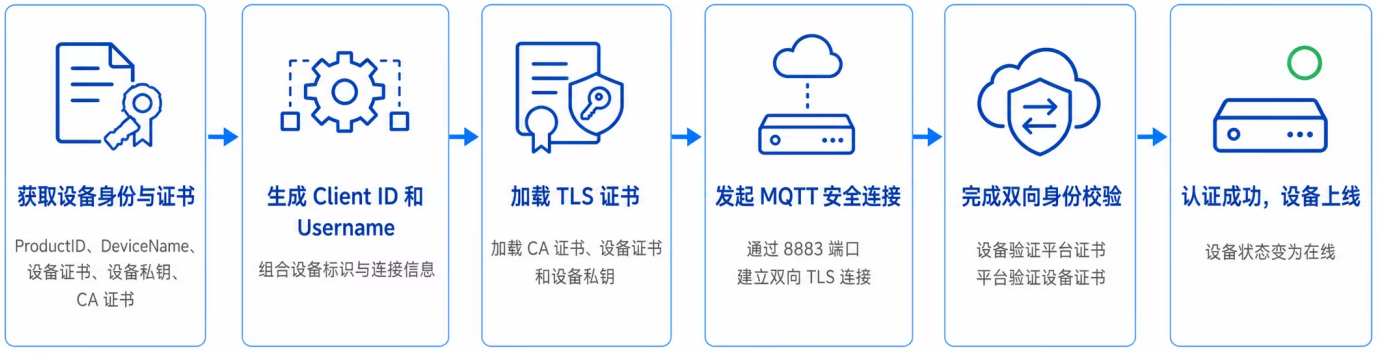
设备证书认证使用平台为每台设备颁发的设备证书和设备私钥验证设备身份。设备连接平台时，通过 CA 证书验证服务器身份，同时向平台提供设备证书并使用设备私钥证明设备身份。双向 TLS 认证通过后，设备即可建立 MQTT 连接并上线。
证书认证采用非对称加密，安全性较高，但要求设备支持 X.509 证书解析及 TLS 连接。

> **注意：**
> 设备私钥属于敏感凭证，请妥善保管。CA 证书和设备证书可以公开分发，但不得泄露设备私钥。

如果尚未确定使用设备证书认证还是设备密钥认证，请参见 认证方式概览 。

认证流程

设备证书认证流程参见下图：



认证参数

设备证书认证涉及“设备身份参数”、“证书文件”和“MQTT 连接参数”三类参数。
如需在控制台创建证书认证设备，并获取设备证书、设备私钥、腾讯云 CA 证书及设备身份信息，请参见 获取设备三元组 。

设备身份参数

参数	参数来源	说明	是否保密

ProductID	产品详情页或设备详情页。	标识设备所属产品。	否
DeviceName	设备详情页。	标识产品下的具体设备。	否

ProductID 与 DeviceName 组合后形成平台中的设备唯一标识。

证书文件

文件	获取方式	作用	是否保密
设备证书	创建设备后由平台颁发。	向平台声明设备身份。	否
设备私钥	创建设备后由平台颁发。	证明设备持有对应设备证书。	是
CA 证书	平台证书下载入口。	验证物联网开发平台服务器身份。	否

设备证书与设备私钥必须配套使用，不能混用不同设备的证书和私钥。

> ⓘ **注意：**
> 请在设备凭证生成后及时下载并安全保存设备私钥。设备私钥不得写入公开代码、日志、截图或普通配置文件。

证书文件参考：

MQTT 连接参数

参数	取值或生成方式	说明
Broker Address	• 广州：`${ProductID}.iotcloud.tencentdevices.com` • 曼谷：`${ProductID}.ap-bangkok.iothub.tencentdevices.com`	MQTT 服务器地址。
Broker Port	`8883`	证书认证设备接入端口。

Client ID	`${ProductID}${DeviceName}`	ProductID 与 DeviceName 直接拼接。
User name	`${ClientID};${sdkappid};${connid};${expiry}`	MQTT 连接用户名。
Password	可填写任意非空值	证书认证不校验 Password 内容。
Keep Alive	0～900秒	MQTT 连接保活时间。

`${}` 表示变量，不是实际的拼接字符。连接域名和端口应以设备所属地域及控制台当前信息为准。
Username 中各字段的含义如下：

字段	说明
sdkappid	固定填写 `12010126` 。
connid	客户端生成的随机字符串，用于区分连接。
expiry	连接参数的过期时间，使用 Unix 时间戳，单位为秒。

证书认证不使用 DeviceSecret，也不需要对 Username 进行 HMAC 签名。Password 不会参与设备证书校验，但仍需按照 MQTT 客户端的要求设置一个非空值。

证书有效期与更新

设备证书和平台 CA 证书均存在有效期。生产环境应建立证书有效期检查及更新机制，避免证书到期导致设备无法连接平台。

平台 CA 证书

平台 CA 证书由物联网开发平台提供，用于设备验证 MQTT 服务器证书，不属于单台设备的身份凭证，可供使用相同平台信任链的设备复用。
平台 CA 证书发生更新时：

- 使用官方设备端 SDK 的设备，应根据平台通知升级至支持新 CA 证书的 SDK 版本，并通过固件升级部署至设备。
- 使用自研 MQTT 客户端的设备，应从物联网开发平台控制台或官方文档获取最新 CA 证书，并通过固件升级或 OTA 更新设备端信任库。
- 平台不会自动将新 CA 证书写入存量设备，用户需要自行完成设备侧证书更新。
- 生产设备应具备远程更新 CA 证书的能力，并根据平台发布的更新时间安排升级，避免存量设备集中断连。

设备证书和 CA 证书均具有有效期。设备建立 TLS 连接时，平台会校验设备证书是否处于有效期内，设备也会通过 CA 证书校验平台服务器证书。

可以使用以下命令查看证书有效期：

请先下载证书文件，并在证书文件的目录下打开终端，执行：

```
# Windows PowerShell
certutil -dump "证书文件名"

# Linux && macOS Shell
openssl x509 -in "证书文件名" -noout -dates

# 查看输出结果
...
notBefore=     # 证书有效起始时间
notAfter=      # 证书有效结束时间
...
```

```
# 示例
# 腾讯云证书名称：ca.crt
# 设备证书名称：certificate_authentication_device_cert.crt

# Windows PowerShell
certutil -dump ca.crt
certutil -dump certificate_authentication_device_cert.crt

# Linux && macOS Shell
openssl x509 -in ca.crt -noout -dates
openssl x509 -in certificate_authentication_device_cert.crt -noout -
dates

# 查看输出结果
...
notBefore=     # 证书有效起始时间
notAfter=      # 证书有效结束时间
...
```

生成 MQTT 连接参数

设备详情页如已提供完整 MQTT 连接参数，可直接复制使用。也可以根据 ProductID 和 DeviceName 生成 Client ID 与 Username。
生成关系如下：

```
Client ID = ProductID + DeviceName

Username = ClientID + ";12010126;" + connid + ";" + expiry

Password = 任意非空字符串
```

示例：

```
ProductID  = testProductID
DeviceName = testDeviceName
connid     = A1B2C
expiry     = 1786610239

Client ID  = testProductIDtestDeviceName
Username   = testProductIDtestDeviceName;12010126;A1B2C;1786610239
Password   = certificate
```

> **说明：**
> 以上参数仅用于说明拼接规则，不能用于连接真实设备。

安全注意事项

- 每台设备应使用独立的设备证书和设备私钥。
- 不得在日志文件、截图、工单或公开的代码仓库中暴露设备私钥。
- 设备私钥建议保存在安全芯片或受保护的存储区域中。
- 设备证书与设备私钥必须配套使用，不得在不同的设备之间复制使用。
- 不得通过 `tls_insecure_set(True)` 关闭服务器证书校验。
- 设备应维护正确的系统时间，避免证书校验或连接参数校验出现异常。
- 调试完成后，应删除临时复制的设备私钥文件。
- 当设备私钥发生泄露时，应及时禁用设备并重新发放设备凭证。

使用 Python 连接平台

本节使用 Python 脚本连接物联网开发平台。当脚本返回认证成功且控制台设备状态显示为在线时，即表示设备证书认证接入完成。

本示例仅验证设备认证和上线，不包含 Topic 发布、订阅及物模型消息通信。

前提条件

开始接入前，请确认已完成以下准备：

- 已创建认证方式为证书认证的产品。
- 已在产品下创建设备。
- 已获得 **ProductID 和 DeviceName**。
- 已下载当前设备的**设备证书和设备私钥**。
- 已下载平台提供的设备端 **CA 证书**。
- 本地环境已**安装 Python 3**。
- 本地网络能够访问平台 MQTT 接入域名。
- 设备系统时间正确。

准备运行环境

建议在虚拟环境中安装依赖和运行代码。
安装 Paho MQTT 客户端：

```
# Windows powershell
python -m pip install "paho-mqtt>=2.1,<3.0"

# Linux && macOS shell
python3 -m pip install "paho-mqtt>=2.1,<3.0"
```

Paho MQTT 是 Python MQTT 客户端库。本示例使用 MQTT 3.1.1 及 Callback API VERSION2。
安装完成后，可执行以下命令确认版本：

```
# Windows PowerShell
python -m pip show paho-mqtt

# Linux && macOS Shell
python3 -m pip show paho-mqtt

# 输出中的 Version 应为 2.1.0。
```

创建连接脚本

将以下代码保存为 `mqtt_cert_connect.py`：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import secrets
import ssl
import string
import sys
import time
from pathlib import Path

try:
    import paho.mqtt.client as mqtt
except ImportError:
    mqtt = None


SDK_APP_ID = "12010126"

# 凭证只能从脚本同级的cert目录中读取。
SCRIPT_DIR = Path(__file__).resolve().parent
CERT_DIR = (SCRIPT_DIR / "cert").resolve()


def required_input(prompt):
    """读取必填参数并清除首尾空格。"""
    value = input(prompt).strip()
    if not value:
        raise ValueError("输入内容不能为空")
    return value


def get_cert_file(prompt):
    """根据用户输入的文件名，从cert目录中读取凭证文件。"""
    filename = required_input(prompt)

    # 只允许输入文件名，不允许输入目录或绝对路径。
    if (
        Path(filename).is_absolute()
        or "/" in filename
```

```
        or "\\" in filename
        or filename in {".", ".."}
    ):
        raise ValueError(
            "只允许输入cert目录中的文件名，不能包含路径"
        )

    file_path = (CERT_DIR / filename).resolve()

    # 防止通过符号链接等方式访问cert目录外的文件。
    try:
        file_path.relative_to(CERT_DIR)
    except ValueError as exc:
        raise ValueError(
            "凭证文件必须位于脚本同级的cert目录中"
        ) from exc

    if not file_path.is_file():
        raise FileNotFoundError(
            f"在cert目录中未找到文件：{filename}"
        )

    try:
        with file_path.open("rb") as file:
            file.read(1)
    except OSError as exc:
        raise PermissionError(
            f"无法读取凭证文件：{filename}"
        ) from exc

    return file_path


def generate_connid(length=5):
    """生成用于区分MQTT连接的随机字符串。"""
    alphabet = string.ascii_uppercase + string.digits
    return "".join(secrets.choice(alphabet) for _ in range(length))


def main():
```

```
    if mqtt is None:
        raise ImportError(
            '未安装paho-mqtt，请执行：'
            'python3 -m pip install "paho-mqtt>=2.1,<3.0"'
        )

    if not CERT_DIR.is_dir():
        raise FileNotFoundError(
            f"未找到cert目录，请创建：{CERT_DIR}"
        )

    print("请输入cert目录中的凭证文件名。")
    print(f"凭证目录：{CERT_DIR}\n")

    ca_file = get_cert_file("CA证书文件名：")
    device_cert_file = get_cert_file("设备证书文件名：")
    device_key_file = get_cert_file("设备私钥文件名：")

    print("\n已加载以下凭证文件：")
    print(f"- CA证书：{ca_file.name}")
    print(f"- 设备证书：{device_cert_file.name}")
    print(f"- 设备私钥：{device_key_file.name}")

    broker = required_input("\nMQTT服务器地址：")

    port_text = input("端口号（直接回车使用8883）：").strip()
    port = int(port_text) if port_text else 8883

    product_id = required_input("ProductID：")
    device_name = required_input("DeviceName：")

    client_id = f"{product_id}{device_name}"
    connid = generate_connid()
    expiry = int(time.time()) + 60 * 60

    username = (
        f"{client_id};{SDK_APP_ID};{connid};{expiry}"
    )

    # 证书认证不校验Password内容。
```

```
    password = "certificate"

    def on_connect(
        client,
        userdata,
        connect_flags,
        reason_code,
        properties,
    ):
        if reason_code == 0:
            print("\n认证成功，设备已连接物联网开发平台。")
            print("请前往控制台查看设备在线状态。")
        else:
            print(f"\nMQTT连接失败，返回码：{reason_code}")
            client.disconnect()

    def on_connect_fail(client, userdata):
        print(
            "\n网络连接或TLS握手失败，"
            "请检查服务器地址、端口及凭证文件。"
        )

    def on_disconnect(
        client,
        userdata,
        disconnect_flags,
        reason_code,
        properties,
    ):
        if reason_code != 0:
            print(f"\n连接异常断开，返回码：{reason_code}")

    client = mqtt.Client(
        callback_api_version=mqtt.CallbackAPIVersion.VERSION2,
        client_id=client_id,
        clean_session=True,
        protocol=mqtt.MQTTv311,
    )

    client.username_pw_set(
```

```
        username=username,
        password=password,
    )

    client.tls_set(
        ca_certs=str(ca_file),
        certfile=str(device_cert_file),
        keyfile=str(device_key_file),
        cert_reqs=ssl.CERT_REQUIRED,
        tls_version=ssl.PROTOCOL_TLS_CLIENT,
    )

    # 启用服务器证书及域名校验。
    client.tls_insecure_set(False)

    client.on_connect = on_connect
    client.on_connect_fail = on_connect_fail
    client.on_disconnect = on_disconnect

    print(f"\n正在连接：{broker}:{port}")
    print(f"ClientId：{client_id}")
    print("程序将保持连接，按Ctrl+C退出。")

    client.connect(
        host=broker,
        port=port,
        keepalive=60,
    )

    try:
        client.loop_forever()
    finally:
        client.disconnect()


if __name__ == "__main__":
    try:
        sys.exit(main())
    except KeyboardInterrupt:
        print("\n用户终止连接。")
```

```
        sys.exit(0)
    except Exception as exc:
        print(f"\n运行失败：{exc}")
        sys.exit(1)
```

该脚本仅读取证书文件，不会复制或保存证书内容，也不会将认证参数写入配置文件。

准备证书文件

将以下文件放置在 `mqtt_cert_connect.py` 同目录的 `cert` 中

```
├─ mqtt_cert_connect.py
└─ cert/
   ├─ ca.crt
   ├─ device.crt
   └─ device.key
```

其中：

- `ca.crt` 表示平台 CA证书。
- `device.crt` 表示当前设备的设备证书。
- `device.key` 表示与设备证书配套的设备私钥。

> **说明：**
> 以上名称仅为示例，运行脚本时请输入实际文件名。

运行连接脚本

```
# Windows PowerShell
python mqtt_cert_connect.py

# Linux && macOS Shell
python3 mqtt_cert_connect.py
```

运行后依次输入：

```
# 运行示例
凭证目录：D:\test\shell\test_cert\cert

CA证书文件名：ca.crt
```

```
设备证书文件名：certificate_authentication_device_cert.crt
设备私钥文件名：certificate_authentication_device_private.key

已加载以下凭证文件：
- CA证书：ca.crt
- 设备证书：certificate_authentication_device_cert.crt
- 设备私钥：certificate_authentication_device_private.key

MQTT服务器地址：ProductID.iotcloud.tencentdevices.com
端口号（直接回车使用8883）：8883
ProductID：ProductID
DeviceName：certificate_authentication_device
```

连接成功时，程序输出类似：

```
正在连接：******.iotcloud.tencentdevices.com:8883
ClientId：******
程序将保持连接，按Ctrl+C退出。

认证成功，设备已连接物联网开发平台。
请前往控制台查看设备在线状态。
```

程序会持续运行并保持 MQTT 连接。需要断开连接时，按下 `Ctrl+C` 。

验证设备上线

设备发起 MQTT 连接后，如需在控制台确认设备状态及查看上下线记录，请参见 查看设备状态和上下线日志 。

问题排查

网络连接失败

检查以下内容：

- Broker Address 是否与设备详情页或当前地域接入地址一致。
- 端口号是否为证书认证端口 8883。
- 本地 DNS 是否能够解析 MQTT 服务器域名。
- 防火墙或网络策略是否限制对目标域名和端口的访问。
- 相关域名是否已加入访问白名单。

TLS 握手失败

检查以下内容：

- 是否加载了平台提供的正确的 CA 证书。
- 设备证书与设备私钥是否属于同一台设备。
- 设备证书文件或私钥文件是否损坏。
- 证书文件的路径是否正确。
- 设备是否具有读取证书文件和私钥文件的权限。
- 设备的系统时间是否正确。
- 是否错误关闭了服务器证书校验或域名校验。

MQTT 连接失败

检查以下内容：

- ProductID 和 DeviceName 是否属于当前的设备。
- Client ID 是否由 ProductID 和 DeviceName 直接拼接而成。
- Username 中指定的 Client ID 是否与连接时使用的 Client ID 一致。
- Username 中的 sdkappid 是否为 12010126。
- Username 中的 expiry 字段是否已过期。
- 当前产品是否采用了证书认证。
- 设备证书是否已经失效。

证书与私钥不匹配

如果 TLS 握手提示证书错误或私钥错误，请确认：

- 未混用不同设备的证书和私钥。
- 私钥文件的内容完整。
- 文件的格式能够被当前的 TLS 客户端识别。
- 私钥未在复制、换行或编码转换的过程中损坏。

脚本连接成功，但设备未显示在线

检查以下内容：

- 是否进入了正确的实例和正确的设备详情页。
- Client ID 是否对应了当前的 DeviceName。
- 连接脚本是否仍在运行。
- 设备的上下线日志中是否存在新的连接记录。
- 是否有另一个客户端使用相同的 Client ID 建立了连接。

TLS-PSK 密钥认证

最近更新时间：2026-09-11 16:05:32

本文将介绍密钥认证设备如何使用 DeviceSecret 建立 TLS-PSK 连接，并通过8883端口完成设备身份和 MQTT 接入。
如果您尚未了解 ProductID、DeviceName 和 DeviceSecret，请先参见 设备密钥认证 。

方案概述

TLS-PSK 使用设备与平台预先共享的 DeviceSecret 完成 TLS 握手和设备身份认证。
设备将 Base64 编码的 DeviceSecret 解码为原始二进制密钥，并将其作为 TLS 预共享密钥；平台根据 PSK Identity 识别设备，并使用平台保存的 DeviceSecret 验证握手结果。

安全层面	实现方式	作用
设备身份识别	PSK Identity	标识 ProductID 和 DeviceName。
设备身份认证	TLS 预共享密钥	验证设备是否持有正确的 DeviceSecret。
平台身份确认	TLS-PSK 握手	验证对端是否持有相同预共享密钥。
传输链路保护	TLS 1.2	对设备与平台之间的数据进行加密和完整性保护。

TLS-PSK 不使用 X.509证书。平台在握手过程中不提供服务器证书，设备不需要加载平台 CA 证书、设备证书或设备私钥。

> **说明：**
> 本文中的 TLS-PSK 仅指设备端 SDK 支持的预共享密钥接入方式，不等同于“证书型 TLS 加 MQTT HMAC 认证”。

适用场景

以下场景可以使用 TLS-PSK 接入：

- 使用支持 TLS-PSK 的物联网开发平台设备端 SDK。
- 设备采用 DeviceSecret 进行身份认证。
- 需要通过8883端口建立加密连接。
- 能够为每台设备安全保存独立 DeviceSecret。

普通 MQTT 客户端不一定支持 TLS-PSK。使用自研客户端时，需要确认 TLS 库支持 PSK 回调和平台要求的密码套件。
如果业务要求使用 PKI 体系、设备证书或硬件私钥，应选择设备证书认证。

认证流程

TLS-PSK 接入流程参见下图：

TLS-PSK 握手成功后：

- TLS 连接使用预共享密钥完成双端身份确认。
- 平台不会发送服务器证书。
- 设备不执行 CA 证书链和服务器域名校验。
- MQTT CONNECT 报文不设置 Password 字段。
- MQTT 连接成功后，设备状态变为在线。

方案区别

对比项	1883：MQTT 密钥认证	8883：TLS-PSK 密钥认证
传输加密	不使用 TLS。	使用 TLS 1.2。
DeviceSecret 用途	生成 MQTT HMAC Password。	作为 TLS 预共享密钥。
身份认证层级	MQTT 层。	TLS 握手层。
PSK Identity	不涉及。	ProductID 与 DeviceName 拼接。
MQTT Password	需要。	不设置。
服务器证书	不涉及。	不提供。
平台 CA 证书	不需要。	不需要。
客户端要求	支持 MQTT 3.1.1。	同时支持 MQTT 3.1.1和 TLS-PSK。
适用场景	受控环境或基础验证。	设备端 SDK 加密接入。

两种方式使用同一个 DeviceSecret，但认证位置和参数用途不同。TLS-PSK 不是在1883连接参数基础上简单修改端口。

MQTT 连接参数

参数	取值或生成方式	说明
Broker Address	以控制台显示为准。	MQTT 服务器地址。
Broker Port	8883	TLS-PSK 接入端口。
TLS 版本	TLS 1.2	当前验证使用的协议版本。
密码套件	`PSK-AES128-CBC-SHA`	当前 SDK 兼容路径验证结果。
PSK Identity	`${ProductID}${DeviceName}`	ProductID 与 DeviceName 直接拼接。
PSK Key	Base64解码后的 DeviceSecret。	TLS 预共享密钥。
MQTT 版本	MQTT 3.1.1	MQTT 协议版本。
MQTT Client ID	`${ProductID}${DeviceName}`	MQTT 客户端标识。
MQTT Username	按设备端 SDK 规则生成。	用于 MQTT 连接信息。
MQTT Password	不设置。	不应提交 HMAC Password。
KeepAlive	0～900秒	根据设备网络环境设置。

说明：

- `${}` 表示变量，不是实际拼接字符。连接域名应以设备所属地域及控制台显示为准。
- “Password 不设置”表示 MQTT CONNECT 报文中不启用 Password 标志，不建议用空字符串代替未设置。

使用 Python 连接平台

本节使用 Python 脚本连接物联网开发平台。当脚本返回认证成功且控制台设备状态显示为在线时，即表示设备密钥认证接入完成。
本示例仅验证设备认证和上线，不包含 Topic 发布、订阅及物模型消息通信。

前提条件

开始接入前，请确认已完成以下准备：

- 已创建认证方式为**密钥认证**的产品。
- 已在产品下**创建设备**。
- 已获得设备的 **MQTT 连接参数**。
- **本地环境已安装 Python 3.13 或以上版本**。
- 当前 Python 的 OpenSSL 支持 TLS-PSK。
- 本地网络能够访问平台 MQTT 接入域名的 8883 端口。

获取设备身份参数

控制台字段	脚本配置项	说明
产品 ID	ProductID	标识设备所属产品。
设备名称	DeviceName	标识产品下的具体设备。
设备密钥	DeviceSecret	用于生成 TLS-PSK 预共享密钥。
MQTT 服务器地址	Broker Address	设备所属地域的 MQTT 接入域名。

准备运行环境

建议在虚拟环境中安装依赖和运行代码。
Python 标准库从 Python 3.13开始提供 TLS-PSK 客户端回调能力。
运行前检查：

```
python.exe -c "import ssl; print(ssl.OPENSSL_VERSION); print('HAS_PSK
=', ssl.HAS_PSK)"
```

只有输出以下结果时才能运行 TLS-PSK 验证代码：

```
HAS_PSK = True
```

创建连接脚本

将以下代码保存为 `mqtt_tls_psk_connect.py`：

> ⓘ **注意：**
> 该代码仅用于验证平台 TLS-PSK 路径，不建议直接作为生产设备实现。生产设备应使用经过适配和验证的设备端 SDK。

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import base64
import binascii
import getpass
import secrets
import socket
import ssl
import string
import struct
import sys
import time


MIN_PYTHON_VERSION = (3, 13)
DEFAULT_PORT = 8883
DEFAULT_KEEPALIVE = 60
DEFAULT_TIMEOUT = 8
DEFAULT_EXPIRY_SECONDS = 3600
SDK_APP_ID = "12010126"
TLS_PSK_CIPHER = "PSK-AES128-CBC-SHA"


def check_runtime():
    """在读取设备凭证前检查TLS-PSK运行环境。"""
    print("正在检查运行环境。")
    print(f"Python版本：{sys.version.split()[0]}")
    print(f"Python路径：{sys.executable}")
    print(f"OpenSSL版本：{ssl.OPENSSL_VERSION}")

    if sys.version_info < MIN_PYTHON_VERSION:
        raise RuntimeError(
            "当前Python版本不支持TLS-PSK客户端接口。\n"
            "请使用Python 3.13及以上版本运行本程序。\n"
            "例如：D:\\tools\\python313\\python.exe "
            "mqtt_tls_psk_connect.py"
        )
```

```
    if not getattr(ssl, "HAS_PSK", False):
        raise RuntimeError(
            "当前Python所使用的OpenSSL未启用TLS-PSK能力。"
        )

    if not hasattr(ssl.SSLContext, "set_psk_client_callback"):
        raise RuntimeError(
            "当前ssl模块缺少set_psk_client_callback接口。"
        )

    print("TLS-PSK能力：支持")
    print("运行环境检查通过。\n")


def required_input(prompt):
    """读取必填参数并清除首尾空格。"""
    value = input(prompt).strip()
    if not value:
        raise ValueError("输入内容不能为空")
    return value


def get_port():
    """读取并校验MQTT端口。"""
    value = input(
        f"MQTT服务器端口（直接回车使用{DEFAULT_PORT}）："
    ).strip()

    if not value:
        return DEFAULT_PORT

    try:
        port = int(value)
    except ValueError as exc:
        raise ValueError("端口号必须是整数") from exc

    if not 1 <= port <= 65535:
        raise ValueError("端口号必须在1～65535范围内")

    return port
```

```
def decode_device_secret(device_secret):
    """将Base64格式的DeviceSecret解码为TLS PSK。"""
    try:
        psk = base64.b64decode(
            device_secret,
            validate=True,
        )
    except (binascii.Error, ValueError) as exc:
        raise ValueError(
            "DeviceSecret不是有效的Base64字符串，"
            "请检查是否完整复制了控制台中的设备密钥"
        ) from exc

    if not psk:
        raise ValueError("DeviceSecret解码结果为空")

    return psk


def random_connid(length=5):
    """生成MQTT Username使用的随机connid。"""
    alphabet = string.ascii_letters + string.digits
    return "".join(
        secrets.choice(alphabet)
        for _ in range(length)
    )


def encode_utf8(value):
    """编码MQTT UTF-8字符串。"""
    data = value.encode("utf-8")

    if len(data) > 65535:
        raise ValueError("MQTT字符串长度超过65535字节")

    return struct.pack("!H", len(data)) + data
```

```
def encode_remaining_length(length):
    """编码MQTT Remaining Length字段。"""
    result = bytearray()

    while True:
        encoded = length % 128
        length //= 128

        if length > 0:
            encoded |= 0x80

        result.append(encoded)

        if length == 0:
            break

    return bytes(result)


def build_connect_packet(client_id, username, keepalive):
    """
    构造MQTT 3.1.1 CONNECT报文。

    TLS-PSK模式下：
    - MQTT Username仍用于标识设备；
    - MQTT Password不填写；
    - DeviceSecret在TLS握手层作为PSK使用。
    """
    connect_flags = 0x82
    # 0x80：包含Username
    # 0x02：Clean Session
    # 不设置Password标志位

    variable_header = (
        encode_utf8("MQTT")
        + bytes([4])
        + bytes([connect_flags])
        + struct.pack("!H", keepalive)
    )
```

```
    payload = (
        encode_utf8(client_id)
        + encode_utf8(username)
    )

    remaining_length = len(variable_header) + len(payload)

    return (
        bytes([0x10])
        + encode_remaining_length(remaining_length)
        + variable_header
        + payload
    )


def recv_exact(sock, length):
    """从Socket读取指定长度的数据。"""
    chunks = bytearray()

    while len(chunks) < length:
        data = sock.recv(length - len(chunks))

        if not data:
            raise ConnectionError("服务器已关闭连接")

        chunks.extend(data)

    return bytes(chunks)


def receive_mqtt_packet(sock):
    """接收一个完整MQTT报文。"""
    first_byte = recv_exact(sock, 1)[0]

    multiplier = 1
    remaining_length = 0

    while True:
        encoded = recv_exact(sock, 1)[0]
        remaining_length += (encoded & 0x7F) * multiplier
```

```
        if encoded & 0x80 == 0:
            break

        multiplier *= 128

        if multiplier > 128 * 128 * 128:
            raise ValueError("MQTT Remaining Length格式错误")

    payload = recv_exact(sock, remaining_length)
    return first_byte, payload


def parse_connack(first_byte, payload):
    """解析MQTT 3.1.1 CONNACK报文。"""
    if first_byte != 0x20 or len(payload) != 2:
        raise ValueError("服务器返回的不是有效CONNACK报文")

    return_code = payload[1]

    messages = {
        0: "连接成功",
        1: "不支持的协议版本",
        2: "ClientId不合法",
        3: "服务不可用",
        4: "Username或认证信息错误",
        5: "未授权",
    }

    return return_code, messages.get(
        return_code,
        f"未知返回码：{return_code}",
    )


def create_tls_psk_context(psk_identity, psk):
    """创建TLS-PSK客户端上下文。"""
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)

    # TLS-PSK路径不使用X.509服务器证书。
```

```
    context.check_hostname = False
    context.verify_mode = ssl.CERT_NONE

    # 当前已验证的平台TLS-PSK路径使用TLS 1.2。
    context.minimum_version = ssl.TLSVersion.TLSv1_2
    context.maximum_version = ssl.TLSVersion.TLSv1_2

    try:
        context.set_ciphers(TLS_PSK_CIPHER)
    except ssl.SSLError as exc:
        raise RuntimeError(
            f"当前OpenSSL不支持密码套件：{TLS_PSK_CIPHER}"
        ) from exc

    def psk_callback(identity_hint):
        return psk_identity, psk

    context.set_psk_client_callback(psk_callback)
    return context


def main():
    # 必须先检查运行环境，再读取设备信息和凭证。
    check_runtime()

    print("密钥认证设备TLS-PSK连接验证工具")
    print("本工具仅验证设备认证和上线。")
    print("输入的DeviceSecret仅保留在程序内存中，"
          "不会显示或写入文件。\n")

    product_id = required_input("ProductID: ")
    device_name = required_input("DeviceName: ")

    default_broker = (
        f"{product_id}.iotcloud.tencentdevices.com"
    )

    broker = input(
        "MQTT服务器地址"
        f"（直接回车使用{default_broker}）: "
```

```
    ).strip() or default_broker

    if any(char in broker for char in "${}/"):
        raise ValueError(
            "MQTT服务器地址应为不包含协议和路径的完整域名"
        )

    port = get_port()

    device_secret = getpass.getpass(
        "DeviceSecret（输入内容不会显示）："
    ).strip()

    if not device_secret:
        raise ValueError("DeviceSecret不能为空")

    psk = decode_device_secret(device_secret)

    client_id = f"{product_id}{device_name}"
    psk_identity = client_id
    connid = random_connid()
    expiry = int(time.time()) + DEFAULT_EXPIRY_SECONDS
    username = (
        f"{client_id};{SDK_APP_ID};{connid};{expiry}"
    )

    print("\n连接配置：")
    print(f"MQTT服务器：{broker}:{port}")
    print(f"ClientId：{client_id}")
    print(f"PSK Identity：{psk_identity}")
    print(f"PSK长度：{len(psk)}字节")
    print(f"TLS密码套件：{TLS_PSK_CIPHER}")
    print("MQTT Username：已自动生成")
    print("MQTT Password：不使用")
    print("DeviceSecret不会输出。")

    confirm = input(
        "\n是否开始TLS-PSK连接？[y/N]："
    ).strip().lower()
```

```
    if confirm != "y":
        print("已取消连接。")
        return 0

    context = create_tls_psk_context(
        psk_identity,
        psk,
    )

    raw_socket = None
    tls_socket = None

    try:
        print(f"\n正在连接：{broker}:{port}")

        raw_socket = socket.create_connection(
            (broker, port),
            timeout=DEFAULT_TIMEOUT,
        )

        tls_socket = context.wrap_socket(
            raw_socket,
            server_hostname=broker,
        )

        print("TLS-PSK握手成功。")
        print(f"TLS版本：{tls_socket.version()}")

        cipher = tls_socket.cipher()
        if cipher:
            print(f"协商密码套件：{cipher[0]}")

        connect_packet = build_connect_packet(
            client_id,
            username,
            DEFAULT_KEEPALIVE,
        )

        tls_socket.sendall(connect_packet)
```

```
        first_byte, payload = receive_mqtt_packet(
            tls_socket
        )

        return_code, message = parse_connack(
            first_byte,
            payload,
        )

        if return_code != 0:
            raise RuntimeError(
                f"MQTT认证失败：{message}"
            )

        print("\nTLS-PSK连接及MQTT认证成功。")
        print("设备已连接物联网开发平台。")
        print("请前往控制台查看设备在线状态。")
        print("程序将保持连接，按Ctrl+C退出。")

        # 每30秒发送一次MQTT PINGREQ，保持设备在线。
        tls_socket.settimeout(40)

        while True:
            time.sleep(30)
            tls_socket.sendall(b"\xC0\x00")

            packet_type, packet_payload = (
                receive_mqtt_packet(tls_socket)
            )

            if packet_type != 0xD0:
                raise RuntimeError(
                    "未收到预期的MQTT PINGRESP"
                )

    finally:
        if tls_socket is not None:
            try:
                # MQTT DISCONNECT
                tls_socket.sendall(b"\xE0\x00")
```

```
            except Exception:
                pass

            try:
                tls_socket.close()
            except Exception:
                pass

        elif raw_socket is not None:
            try:
                raw_socket.close()
            except Exception:
                pass

    return 0


if __name__ == "__main__":
    try:
        sys.exit(main())
    except KeyboardInterrupt:
        print("\n用户终止连接。")
        sys.exit(0)
    except socket.gaierror:
        print("\n运行失败：无法解析MQTT服务器地址。")
        sys.exit(1)
    except socket.timeout:
        print("\n运行失败：连接或响应超时。")
        sys.exit(1)
    except ssl.SSLError as exc:
        print(f"\nTLS-PSK连接失败：{exc}")
        sys.exit(1)
    except Exception as exc:
        print(f"\n运行失败：{exc}")
        sys.exit(1)
```

运行连接脚本

执行：

```
# Windows PowerShell
python mqtt_tls_psk_connect.py

# Linux && macOS Shell
python3 mqtt_tls_psk_connect.py
```

```
# 运行成功后依次输入
ProductID:
DeviceName:
MQTT服务器地址:
MQTT服务器端口（直接回车使用8883）:
DeviceSecret（输入内容不会显示）:
```

连接成功时，程序输出类似：

```
是否开始TLS-PSK连接? [y/N]: y

正在连接: ******.iotcloud.tencentdevices.com:8883
TLS-PSK握手成功。
TLS版本: TLSv1.2
协商密码套件:

TLS-PSK连接及MQTT认证成功。
设备已连接物联网开发平台。
请前往控制台查看设备在线状态。
程序将保持连接，按Ctrl+C退出。
```

程序会持续运行并保持 MQTT 连接。需要断开连接时，按下 `Ctrl+C` 。

验证设备上线

设备发起 MQTT 连接后，如需在控制台确认设备状态及查看上下线记录，请参见 查看设备状态和上下线日志 。

问题排查

无法连接8883端口

检查：

- MQTT 服务器地址是否正确。

- 是否错误保留 `${}` 。
- 防火墙是否允许访问8883端口。
- DNS 是否能够解析服务器域名。
- 产品和设备是否属于当前地域。

TLS-PSK 握手失败

检查：

- 客户端 TLS 库是否支持 PSK。
- 密码套件是否受支持。
- PSK Identity 是否为 ProductID 与 DeviceName 直接拼接。
- DeviceSecret 是否属于当前设备。
- DeviceSecret 是否完成 Base64解码。
- PSK Key 是否以二进制形式传入。

TLS 握手成功但 MQTT 连接失败

检查：

- MQTT 协议是否为3.1.1。
- Client ID 是否正确。
- Username 是否按 SDK 规则生成。
- MQTT CONNECT 是否错误设置 Password。
- Client ID 和 PSK Identity 是否指向同一设备。
- 设备是否已被禁用或删除。

量产设备凭证管理

最近更新时间：2026-09-11 16:05:33

本文将介绍设备凭证的主要发放方式，帮助您根据认证方式、设备规模、产线能力和安全要求，选择产线预置、厂商云分发或动态注册方案。
本文只介绍方案差异和选择建议，不包含控制台操作、动态注册协议及 MQTT 接入步骤。

设备凭证说明

设备接入物联网开发平台前，需要获得平台分配的设备身份信息和设备级凭证。
不同认证方式使用的凭证不同：

认证方式	设备身份信息	设备级凭证
密钥认证	ProductID、DeviceName	DeviceSecret
证书认证	ProductID、DeviceName	设备证书、设备私钥

说明：

- DeviceSecret 和设备私钥均属于敏感凭证，应由每台设备独立保存，不得由多台设备共用。
- 腾讯云 CA 证书用于设备验证平台服务器身份，不属于设备级身份凭证。

如果尚未确定认证方式，请参见 设备认证方式概览 了解情况。

发放方式对比

设备凭证主要可以通过以下三种方式发放：

对比项	产线预置	厂商云分发	动态注册
基本方式	在生产过程中写入设备级凭证。	设备首次联网后向厂商服务器申请凭证。	设备直接向平台申请设备级凭证。
产线写入内容	每台设备不同的设备级凭证。	厂商自定义的初始身份信息。	ProductID、ProductSecret 等产品级信息。
是否需要厂商云服务	否	是	否
是否需要逐台写入设备凭证	是	否	否

设备首次启动是否依赖网络	否	是	是
适用场景	设备数量可控、产线具备安全写入能力。	已有设备管理服务或需要自定义审核流程。	大批量生产，希望减少逐台凭证烧录。

调用服务端 API 不是独立的凭证发放方式。产线系统和厂商云服务均可以通过服务端 API 创建设备并获取设备凭证。

产线预置设备凭证

方案说明

产线预置是指厂商提前在物联网开发平台创建设备，取得每台设备的设备级凭证，然后在生产过程中将对应凭证写入设备。
设备首次联网时，直接使用本地保存的凭证接入物联网开发平台。
基本流程如下：

设备可以通过以下方式创建：

- 在控制台批量创建设备。
- 由产线系统调用服务端 API 创建设备。

注意事项

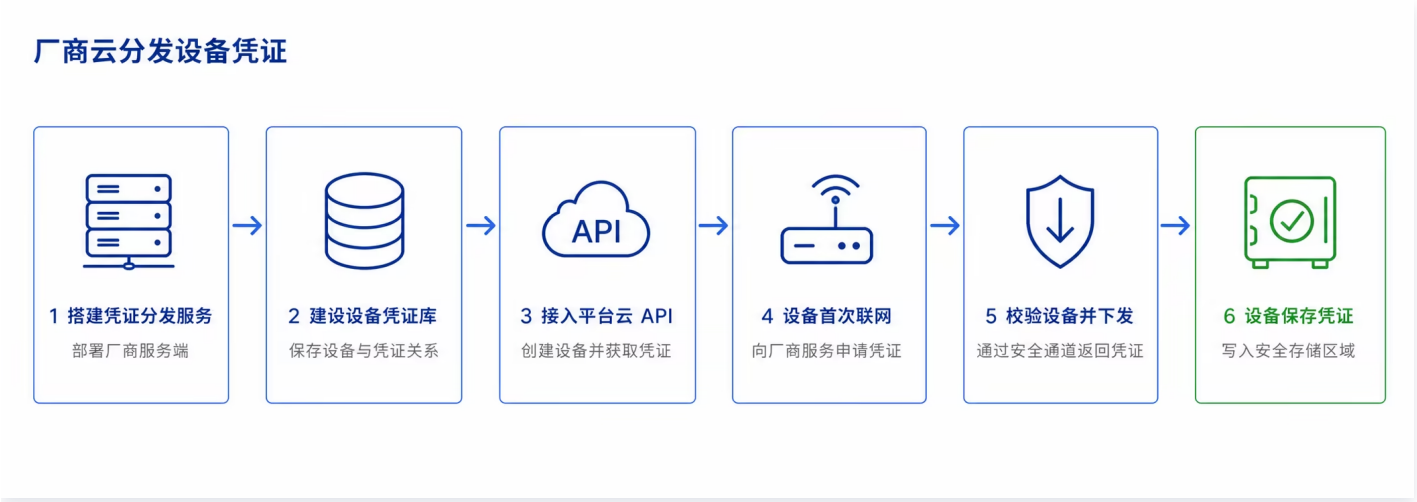
产线系统需要保证设备、设备名称和凭证一一对应，避免出现错烧、漏烧或多台设备共用凭证。
生产日志中不得记录完整 DeviceSecret 或设备私钥。

厂商云分发设备凭证

方案说明

厂商云分发是指设备首次启动后，先连接厂商建设的凭证分发服务，由厂商服务完成设备审核、平台设备创建及凭证下发。

基本流程如下：



这种方式不要求产线逐台写入物联网开发平台设备凭证，但需要设备具备用于访问厂商云服务的初始身份信息。

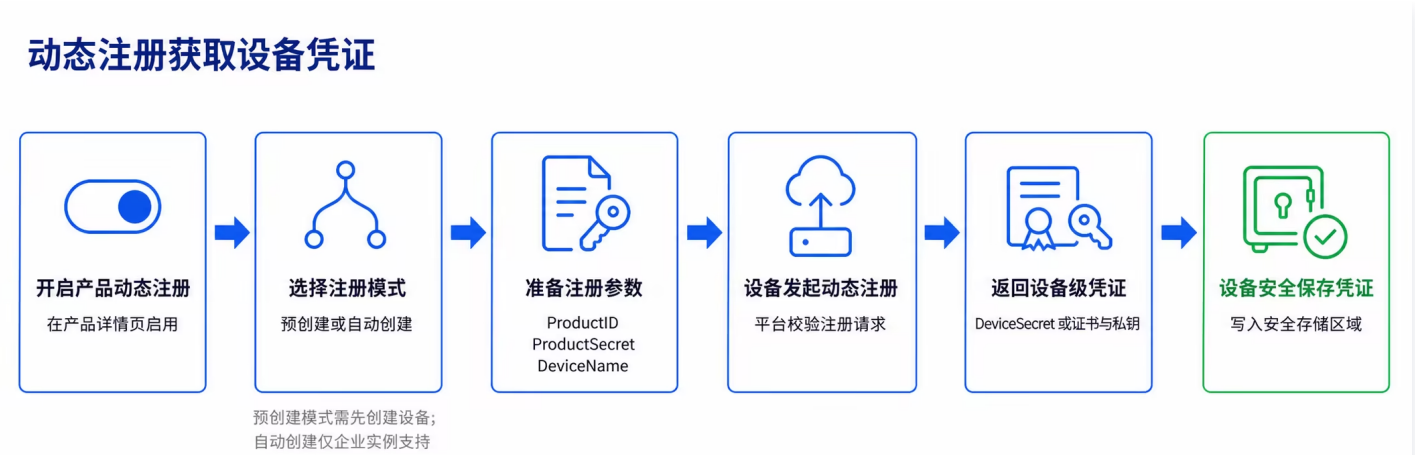
注意事项

- 厂商需要自行建设和维护：
  - 设备初始身份校验机制。
  - 凭证分发接口。
  - 设备与凭证关系数据库。
  - 服务端 API 调用逻辑。
  - 凭证传输及存储保护。
  - 失败重试和异常设备处理机制。
- 设备与厂商服务之间必须使用安全连接，不能通过明文接口传输 DeviceSecret 或设备私钥。

动态注册获取设备凭证

方案说明

动态注册是指设备使用产品级注册信息向物联网开发平台发起注册请求，平台校验成功后返回该设备对应的设备级凭证。
基本流程如下：



根据产品认证方式，动态注册成功后返回：

- 密钥认证：DeviceSecret。
- 证书认证：设备证书和设备私钥。

动态注册只负责获取设备级凭证，不等同于设备已经完成 MQTT 接入。设备还需要使用取得的凭证连接物联网开发平台。

了解更多 动态注册概览 。

注意事项

- ProductSecret 属于产品级敏感凭证。多台设备可能使用同一 ProductSecret 发起注册，一旦泄露，影响范围通常大于单个设备凭证。
- 设备成功取得 DeviceSecret、设备证书或设备私钥后，应立即写入安全的非易失存储，后续启动时直接读取本地凭证，不应将动态注册作为每次启动的固定步骤。
- 未激活设备重复注册可能刷新设备级凭证，使设备此前取得的凭证失效。已激活设备能否再次注册及注册模式限制，请参见对应的动态注册文档。

动态注册
动态注册概览

最近更新时间：2026-09-11 16:05:33

本文将介绍设备动态注册的作用、基本流程及实例支持差异，帮助您选择对应的动态注册方式。

功能介绍

动态注册用于设备首次联网时获取设备级认证凭证，适用于批量生产过程中不便为每台设备分别写入设备密钥、设备证书或设备私钥的场景。
使用动态注册时，同一产品下的设备可以统一预置 ProductID 和 ProductSecret。每台设备还需要提供在当前产品下唯一的 DeviceName。
设备首次联网后，使用上述信息向平台发起动态注册请求。平台校验通过后，根据产品认证方式返回对应的设备级凭证。

> ⓘ **注意：**
> ProductSecret 是产品级凭证，仅用于动态注册，不能代替设备级凭证直接完成 MQTT 设备认证。

基本流程

动态注册流程参见下图：

动态注册和设备认证属于两个不同阶段：

1. 设备通过动态注册获取设备级凭证。
2. 设备保存凭证。
3. 设备使用获得的凭证建立 MQTT 连接。
4. 平台完成设备身份认证后，设备上线。

动态注册成功仅表示设备已经获得凭证，不代表设备已经上线。

实例支持差异

能力	消费实例	企业实例
为预创建设备动态获取 DeviceSecret	支持	支持
根据注册请求自动创建设备	不支持	支持
动态获取设备证书和设备私钥	不支持	支持

消费实例使用动态注册前，需要提前创建设备。
企业实例支持根据动态注册请求自动创建设备，并返回对应的设备级凭证。

> **说明：**
> 部分历史文档将消费实例称为公共实例。本文统一使用当前控制台中的“消费实例”表述。

选择动态注册方式

平台根据产品认证方式返回不同的设备级凭证，了解不同认证方式详见 设备认证方式概览 。

产品认证方式	动态注册结果	后续操作	动态注册文档
密钥认证	DeviceSecret	使用设备密钥完成 MQTT 认证	密钥认证动态注册
证书认证	设备证书和设备私钥	使用设备证书和私钥完成 MQTT 认证	证书认证动态注册

证书认证动态注册不会返回平台 CA 证书。设备建立 MQTT 连接前，还需要按照证书认证接入要求准备平台 CA 证书。

密钥认证动态注册

最近更新时间：2026-09-11 16:05:33

本文将介绍密钥认证设备如何使用 `ProductID` 、 `DeviceName` 和 `ProductSecret` 发起动态注册，获取设备级 `DeviceSecret` 。
本文以成功获取并保存设备凭证为完成标准，不包含后续 MQTT 连接与设备上线。需要连接平台时，请参见 设备密钥认证 。

功能介绍

动态注册用于降低量产阶段逐台写入设备密钥的成本。设备首次联网时，使用产品级注册信息向平台发起请求，平台校验通过后返回加密的设备级凭证。
密钥认证动态注册涉及以下两类密钥：

密钥	作用范围	主要用途	是否保密
`ProductSecret`	产品级	生成动态注册请求签名、解密注册结果。	是
`DeviceSecret`	设备级	生成 MQTT 认证参数，验证单台设备身份。	是

`ProductSecret` 由同一产品下参与动态注册的设备共享，不能代替 `DeviceSecret` 进行 MQTT 身份认证。

动态注册模式

发起动态注册前，需要先根据实例类型和设备创建方式选择注册模式。

注册模式	适用实例	DeviceName 要求	平台处理方式
预创建设备	消费实例、企业实例	必须提前在控制台创建。	为已有且未激活的设备返回 `DeviceSecret` 。
自动创建设备	仅企业实例	无需提前创建。	根据首次注册请求创建设备并返回 `DeviceSecret` 。

设备状态与重复注册

设备是否允许动态注册取决于其激活状态。

设备状态	是否允许动态注册	处理结果

云端已创建，但从未注册、从未接入 MQTT。	允许	生成并返回 DeviceSecret。
已注册获得密钥，但从未使用该密钥接入 MQTT。	允许	重新生成 DeviceSecret，旧密钥失效。
曾使用设备密钥成功接入 MQTT。	不允许	动态注册被拒绝。

说明：

- 控制台可能将前两种情况都显示为“未激活”，但两者的密钥状态不同。
- 设备收到 DeviceSecret 后，应立即保存。未完成保存前，不应再次发起动态注册。

注册流程

密钥认证设备的动态注册流程如下图：

动态注册完成只表示设备获得了后续身份认证所需的 `DeviceSecret`，不表示设备已经连接平台或处于在线状态。

参数说明

动态注册涉及注册模式、设备身份、请求地址、请求头、请求正文和返回结果六类参数。
如需在控制台开启动态注册、选择注册模式并获取 ProductID、ProductSecret 和 DeviceName，请参见 启用动态注册 。

注册输入参数

参数	参数来源	说明	是否保密
动态注册地址	平台文档	地域对应的动态注册服务地址。	否

ProductID	产品详情页	标识设备所属产品。	否
ProductSecret	产品详情页	用于生成注册请求签名和解密返回结果。	是
DeviceName	控制台或设备端	产品下的设备唯一标识。	否
注册模式	产品详情页	预创建设备或自动创建设备。	否

注册模式参数

注册模式用于确定设备是否需要提前创建，但不会作为字段发送至动态注册接口。

模式	对应配置	说明
预创建设备	`RegisterType=1`	设备须提前创建，消费实例和企业实例均可使用。
自动创建设备	`RegisterType=2`	平台按请求创建设备，仅企业实例支持。

`RegisterType` **是产品动态注册配置，不是设备注册请求正文中的参数。**

设备身份参数

参数	参数来源	说明	是否保密
`ProductID`	产品详情页。	标识设备所属产品。	否
`DeviceName`	• 预创建设备模式：控制台预创建设备名称。 • 自动创建设备模式：设备端注册时提供名称。	标识产品下的具体设备。	否
`ProductSecret`	产品详情页。	用于注册签名和结果解密。	是

`ProductSecret` **属于产品级敏感凭证。任何获得该凭证的设备或系统都可能以该产品身份发起注册请求，应限制其分发和读取范围。**

请求地址

```
# 广州地域
https://ap-guangzhou.gateway.tencentdevices.com/device/register
```

```
# 曼谷地域
https://ap-bangkok.gateway.tencentdevices.com/device/register
```

参数	示例值	说明
请求协议	`HTTPS`	建议使用 HTTPS。
Host	• 广州地域：`ap-guangzhou.gateway.tencentdevices.com` • 曼谷地域：`ap-bangkok.gateway.tencentdevices.com/device/register`	动态注册服务域名。
URI	`/device/register`	动态注册接口路径。
请求方法	`POST`	固定值。
Content-Type	`application/json; charset=utf-8`	固定使用 JSON。

请求头参数

请求头	是否必选	说明
`Content-Type`	是	固定为 `application/json; charset=utf-8` 。
`Host`	是	动态注册服务域名。
`X-TC-Algorithm`	是	签名算法，填写 `hmacsha256` 。
`X-TC-Timestamp`	是	当前 Unix 时间戳，单位为秒。
`X-TC-Nonce`	是	客户端生成的随机数。
`X-TC-Signature`	是	使用 `ProductSecret` 生成的请求签名。

请求正文参数

参数	是否必选	类型	说明
`ProductId`	是	String	产品唯一标识。
`DeviceName`	是	String	设备在产品下的唯一名称。

请求示例：

```
{
  "ProductId": "您的ProductID",
  "DeviceName": "您的DeviceName"
}
```

参数名称区分大小写。请求正文使用 `ProductId`，不能写成 `ProductID`。

签名参数

待签名字符串按照以下顺序拼接：

```
POST
${Host}
/device/register

hmacsha256
${Timestamp}
${Nonce}
${RequestHash}
```

参数	生成方式
`Host`	动态注册服务域名。
`Timestamp`	当前 Unix 时间戳，单位为秒。
`Nonce`	客户端生成的随机数。
`RequestHash`	对实际请求正文进行 SHA-256 计算后得到的小写十六进制字符串。
`Signature`	使用 `ProductSecret` 对待签名字符串进行 HMAC-SHA256 计算，再进行 Base64 编码。

签名公式：

```
RequestHash = LowercaseHex(SHA256(RequestBody))

Signature = Base64(
    HMAC-SHA256(ProductSecret, StringToSign)
)
```

用于生成 `RequestHash` 的请求正文必须与实际发送的字节内容完全一致。

平台返回参数

平台正常处理请求后，主要返回以下参数：

参数	类型	说明
`RequestId`	String	请求唯一标识，用于问题定位。
`Len`	Int64	加密 `Payload` 的长度。
`Payload`	String	Base64 编码的加密设备凭证。
`State`	Int64	`0` 表示新增设备，`1` 表示设备已激活。

返回示例：

```
{
  "Response": {
    "Len": 53,
    "Payload": "加密后的设备凭证",
    "RequestId": "请求唯一标识",
    "State": 1
  }
}
```

`Payload` **不能直接作为设备密钥使用，需要先进行解密。**

Payload 解密参数

参数	取值
Base64	先对 `Payload` 进行 Base64 解码。

AES 模式	AES-128-CBC
AES 密钥	`ProductSecret` 的前16个字节。
初始化向量	16个字符 `0` 。
数据清理	去除解密结果末尾的零字节。
结果格式	JSON

密钥认证设备解密后的内容类似：

```
{
  "encryptionType": 2,
  "psk": "设备级DeviceSecret"
}
```

参数	说明
`encryptionType`	凭证类型；密钥认证应为 `2` 。
`PSK`	当前设备最新的 `DeviceSecret` 。

使用 Python 验证动态注册

以下示例用于验证：

```
选择注册模式
→ 输入注册参数
→ 生成请求签名
→ 发起动态注册
→ 解密Payload
→ 脱敏显示设备凭证
```

示例以成功接收并解密 `DeviceSecret` 作为运行成功标准，不负责 MQTT 接入。

> ⓘ **注意：**
> 本示例仅用于测试设备验证。未激活设备重复运行脚本可能重新生成 `DeviceSecret` ，使此前获得的旧密钥失效。示例只显示脱敏凭证，不保存完整凭证，不应直接用于生产量产。

准备运行环境

安装加解密依赖：

```
# Windows PowerShell
python -m pip install "pycryptodome==3.23.0"

# Linux或macOS
python3 -m pip install "pycryptodome==3.23.0"
```

如果 Linux 提示 Python 环境由系统管理，请在虚拟环境中安装依赖。

准备设备参数

请在控制台开启动态注册、选择注册模式并获取 ProductID、ProductSecret 和 DeviceName，详情请参见 启用动态注册 。

创建注册脚本

将以下代码保存为 `dynamic_register_psk.py`：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import base64
import getpass
import hashlib
import hmac
import json
import secrets
import sys
import time
import urllib.error
import urllib.request
from urllib.parse import urlsplit, urlunsplit

try:
    from Crypto.Cipher import AES
except ImportError:
    AES = None


DEFAULT_REGISTER_URL = (
```

```
    "https://ap-guangzhou.gateway.tencentdevices.com/device/register"
)


def required_input(prompt):
    """读取必填参数并清除首尾空格。"""
    value = input(prompt).strip()

    if not value:
        raise ValueError("输入内容不能为空")

    return value


def select_register_mode():
    """选择控制台中已配置的动态注册模式。"""
    print("请选择动态注册模式：")
    print("1. 预创建设备：消费实例和企业实例均支持")
    print("2. 自动创建设备：仅企业实例支持")

    choice = required_input("请输入1或2：")

    if choice == "1":
        register_mode = "预创建设备"
        print(f"\n已选择：{register_mode}")
        print("请确认目标设备已在控制台创建且处于未激活状态。")
        return register_mode

    if choice == "2":
        register_mode = "自动创建设备"
        print(f"\n已选择：{register_mode}")
        print("请确认当前使用企业实例，并已开启自动创建设备。")
        return register_mode

    raise ValueError("注册模式只能输入1或2")


def get_register_endpoint():
    """读取并校验完整动态注册地址。"""
    print("\n请输入动态注册参数。")
```

```
    print("输入内容仅在本次运行期间使用，不会保存到文件。\n")

    register_url = input(
        "动态注册地址"
        f"（直接回车使用{DEFAULT_REGISTER_URL}）："
    ).strip()

    if not register_url:
        register_url = DEFAULT_REGISTER_URL

    if any(char in register_url for char in "${}"):
        raise ValueError(
            "动态注册地址不能包含变量符号，"
            "请填写实际地域的完整地址"
        )

    parsed = urlsplit(register_url)

    if parsed.scheme.lower() != "https":
        raise ValueError(
            "动态注册地址必须以https://开头"
        )

    if not parsed.hostname:
        raise ValueError(
            "动态注册地址中缺少有效的服务域名"
        )

    if not parsed.hostname.endswith(
        ".gateway.tencentdevices.com"
    ):
        raise ValueError(
            "动态注册地址不是有效的腾讯云设备注册服务地址"
        )

    if parsed.username or parsed.password:
        raise ValueError(
            "动态注册地址不能包含用户名或密码"
        )
```

```
    if parsed.query or parsed.fragment:
        raise ValueError(
            "动态注册地址不能包含查询参数或片段"
        )

    # 兼容地址末尾多输入一个斜杠。
    request_path = parsed.path.rstrip("/")

    if request_path != "/device/register":
        raise ValueError(
            "动态注册地址的路径必须为/device/register"
        )

    # 生成规范化URL，确保请求路径和签名路径一致。
    normalized_url = urlunsplit(
        (
            "https",
            parsed.netloc,
            request_path,
            "",
            "",
        )
    )

    return {
        "url": normalized_url,
        "host": parsed.netloc,
        "path": request_path,
    }


def get_device_name(register_mode):
    """根据注册模式提示用户输入DeviceName。"""
    print("\n设备名称填写说明：")

    if register_mode == "预创建设备":
        print(
            "请输入已在控制台创建且当前处于未激活状态的"
            "设备名称，必须与控制台中的名称完全一致。"
        )
```

```
    else:
        print(
            "请输入计划使用的设备名称。若该名称不存在，"
            "平台将按此名称自动创建设备。"
        )
        print(
            "设备名称必须符合平台命名规则，"
            "并在当前产品下保持唯一。"
        )

    return required_input("DeviceName: ")


def build_request(
    register_endpoint,
    product_id,
    product_secret,
    device_name,
):
    """生成动态注册请求正文和鉴权请求头。"""
    request_body = json.dumps(
        {
            "ProductId": product_id,
            "DeviceName": device_name,
        },
        separators=(",", ":"),
        ensure_ascii=False,
    )

    timestamp = str(int(time.time()))

    # X-TC-Nonce必须是随机正整数。
    nonce = str(
        secrets.randbelow(2147483646) + 1
    )

    payload_hash = hashlib.sha256(
        request_body.encode("utf-8")
    ).hexdigest()
```

```
    # 第4行是空查询字符串，必须保留。
    string_to_sign = "\n".join(
        [
            "POST",
            register_endpoint["host"],
            register_endpoint["path"],
            "",
            "hmacsha256",
            timestamp,
            nonce,
            payload_hash,
        ]
    )

    signature_bytes = hmac.new(
        product_secret.encode("utf-8"),
        string_to_sign.encode("utf-8"),
        hashlib.sha256,
    ).digest()

    signature = base64.b64encode(
        signature_bytes
    ).decode("utf-8")

    headers = {
        "Content-Type": "application/json",
        "Host": register_endpoint["host"],
        "X-TC-Algorithm": "hmacsha256",
        "X-TC-Timestamp": timestamp,
        "X-TC-Nonce": nonce,
        "X-TC-Signature": signature,
    }

    return request_body.encode("utf-8"), headers


def send_register_request(
    register_endpoint,
    request_body,
    headers,
```

```
):
    """使用Python标准库发送动态注册请求。"""
    request = urllib.request.Request(
        url=register_endpoint["url"],
        data=request_body,
        headers=headers,
        method="POST",
    )

    try:
        with urllib.request.urlopen(
            request,
            timeout=15,
        ) as response:
            response_body = response.read().decode("utf-8")

    except urllib.error.HTTPError as exc:
        try:
            error_body = exc.read().decode(
                "utf-8",
                errors="replace",
            )
        except Exception:
            error_body = ""

        message = f"动态注册请求返回HTTP {exc.code}"

        if error_body:
            message += f"，响应内容：{error_body}"

        raise RuntimeError(message) from exc

    except urllib.error.URLError as exc:
        raise ConnectionError(
            f"无法连接动态注册服务：{exc.reason}"
        ) from exc

    except TimeoutError as exc:
        raise TimeoutError(
            "动态注册请求超时，请检查网络和注册地址"
```

```
        ) from exc

    try:
        return json.loads(response_body)
    except json.JSONDecodeError as exc:
        raise ValueError(
            "平台注册接口未返回有效的JSON数据："
            f"{response_body}"
        ) from exc


def decrypt_payload(payload, product_secret):
    """使用ProductSecret解密动态注册返回的Payload。"""
    product_secret_bytes = product_secret.encode("utf-8")

    if len(product_secret_bytes) < 16:
        raise ValueError(
            "ProductSecret长度不足，无法生成AES密钥"
        )

    # AES-128密钥取ProductSecret的前16字节。
    aes_key = product_secret_bytes[:16]

    # 动态注册协议使用16个字符0作为初始向量。
    aes_iv = b"0" * 16

    try:
        encrypted_payload = base64.b64decode(
            payload,
            validate=True,
        )
    except Exception as exc:
        raise ValueError(
            "响应中的Payload不是有效的Base64数据"
        ) from exc

    if len(encrypted_payload) % AES.block_size != 0:
        raise ValueError(
            "Payload长度不符合AES-CBC解密要求"
        )
```

```
    cipher = AES.new(
        aes_key,
        AES.MODE_CBC,
        aes_iv,
    )

    decrypted_bytes = cipher.decrypt(
        encrypted_payload
    ).rstrip(b"\x00")

    try:
        decrypted_text = decrypted_bytes.decode("utf-8")
        return json.loads(decrypted_text)
    except Exception as exc:
        raise ValueError(
            "Payload解密成功，但无法解析为JSON"
        ) from exc


def parse_register_response(
    response_data,
    product_secret,
):
    """检查动态注册响应并解析设备凭证。"""
    response = response_data.get("Response")

    if not isinstance(response, dict):
        raise ValueError(
            f"动态注册响应格式异常：{response_data}"
        )

    error = response.get("Error")

    if error:
        error_code = error.get("Code", "Unknown")
        error_message = error.get("Message", "Unknown")

        raise RuntimeError(
            f"动态注册失败："
```

```
            f"{error_code} - {error_message}"
        )

    payload = response.get("Payload")

    if not payload:
        raise ValueError(
            f"动态注册响应中未找到Payload：{response}"
        )

    credential = decrypt_payload(
        payload=payload,
        product_secret=product_secret,
    )

    return response, credential


def mask_secret(secret):
    """脱敏显示DeviceSecret。"""
    if len(secret) <= 8:
        return "*" * len(secret)

    return f"{secret[:4]}********{secret[-4:]}"


def main():
    if AES is None:
        raise ImportError(
            "当前Python环境缺少pycryptodome，请执行：\n"
            f'{sys.executable} -m pip install '
            '"pycryptodome==3.23.0"'
        )

    print("密钥认证设备动态注册验证工具")
    print(
        "未激活设备重复注册可能刷新DeviceSecret，"
        "请仅使用测试设备验证。\n"
    )
```

```
    register_mode = select_register_mode()
    register_endpoint = get_register_endpoint()

    product_id = required_input("ProductID：")
    device_name = get_device_name(register_mode)

    product_secret = getpass.getpass(
        "ProductSecret（输入内容不会显示）："
    ).strip()

    if not product_secret:
        raise ValueError("ProductSecret不能为空")

    print("\n即将发起动态注册。")
    print(f"注册模式：{register_mode}")
    print(f"动态注册地址：{register_endpoint['url']}")
    print(f"ProductID：{product_id}")
    print(f"DeviceName：{device_name}")
    print(
        "如果设备仍处于未激活状态，本次注册可能生成新的"
        "DeviceSecret，并使旧密钥失效。"
    )

    confirmation = input(
        "是否开始动态注册？[y/N]："
    ).strip().lower()

    if confirmation not in {"y", "yes"}:
        print("已取消动态注册。")
        return 0

    request_body, headers = build_request(
        register_endpoint=register_endpoint,
        product_id=product_id,
        product_secret=product_secret,
        device_name=device_name,
    )

    print(
        f"\n正在请求：{register_endpoint['url']}"
```

```
    )

    response_data = send_register_request(
        register_endpoint=register_endpoint,
        request_body=request_body,
        headers=headers,
    )

    response, credential = parse_register_response(
        response_data=response_data,
        product_secret=product_secret,
    )

    encryption_type = credential.get("encryptionType")
    device_secret = credential.get("psk")

    if encryption_type != 2 or not device_secret:
        raise ValueError(
            "平台未返回有效的密钥认证设备凭证"
        )

    state = response.get("State")

    state_text = {
        0: "新建设备",
        1: "已有设备",
        "0": "新建设备",
        "1": "已有设备",
    }.get(state, "未返回或未知")

    print("\n动态注册运行成功。")
    print(f"注册模式：{register_mode}")
    print(f"ProductID：{product_id}")
    print(f"DeviceName：{device_name}")
    print(f"设备处理状态：{state_text}")
    print("凭证类型：密钥认证")
    print(f"encryptionType：{encryption_type}")
    print(
        f"DeviceSecret（脱敏）："
        f"{mask_secret(device_secret)}"
```

```
    )
    print(f"DeviceSecret长度：{len(device_secret)}")
    print(
        f"RequestId："
        f"{response.get('RequestId', '未返回')}"
    )

    print(
        "\n已成功接收设备凭证。"
        "完整DeviceSecret不会显示或保存。"
    )
    print(
        "如用于生产设备，请在程序中接入安全存储，"
        "并在注册成功后立即保存完整DeviceSecret。"
    )

    return 0


if __name__ == "__main__":
    try:
        sys.exit(main())
    except KeyboardInterrupt:
        print("\n用户终止操作。")
        sys.exit(0)
    except Exception as exc:
        print(f"\n运行失败：{exc}")
        sys.exit(1)
```

运行脚本

```
# Windows PowerShell
python dynamic_register_psk.py

# Linux或macOS
python3 dynamic_register_psk.py
```

程序将依次要求输入：

```
注册模式
动态注册地址
ProductID
DeviceName
ProductSecret
```

ProductSecret 输入时不会显示在终端中，也不会写入文件。

运行成功结果

成功接收并解密设备凭证后，输出类似：

```
动态注册运行成功。
注册模式：预创建设备
ProductID：ASJXXXXXXX
DeviceName：device_001
设备处理状态：已有设备
凭证类型：密钥认证
encryptionType：2
DeviceSecret（脱敏）：lDZ6********s7Q==
DeviceSecret长度：24
RequestId：xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx

已成功接收设备凭证。完整DeviceSecret不会显示或保存。
```

可在控制台设备页面查看注册成功的设备信息。

常见问题

预创建设备模式提示设备不存在

检查：

- 是否已在当前产品下创建目标设备。
- `DeviceName` 是否与控制台完全一致。
- 产品和设备是否属于当前实例。
- 请求地址是否与产品所在地域一致。

自动创建设备模式注册失败

检查：

- 是否使用企业实例。

- 是否开启自动创建设备。
- 动态注册设备数量是否达到配置上限。
- 同名设备是否已经激活。

重复注册后原密钥无法使用

未激活设备重复注册可能重新生成 `DeviceSecret` 。注册成功后，应以最新返回的 `PSK` 为准，之前获得的旧密钥可能已经失效。

已激活设备无法再次注册

设备曾使用 `DeviceSecret` 成功接入 MQTT 后即进入已激活状态。已激活设备不能通过动态注册重新获取同一 `DeviceName` 的设备密钥。

Payload 解密失败

检查：

- 是否使用当前产品的 `ProductSecret` 。
- 是否先对 `Payload` 进行 Base64 解码。
- 是否使用 `ProductSecret` 前16个字节作为 AES 密钥。
- 是否使用 AES-128-CBC。
- 初始化向量是否为16个字符 `0` 。
- 是否正确清理解密结果末尾的零字节。

恢复出厂后无法重新注册

如果设备恢复出厂时删除了 `DeviceSecret` ，但云端设备已经激活，则不能通过同一设备名称再次注册获取设备凭证。
恢复出厂时建议保留设备身份凭证。

证书认证动态注册

最近更新时间：2026-09-11 16:05:33

本文将介绍证书认证设备如何使用 `ProductID` 、 `DeviceName` 和 `ProductSecret` 发起动态注册，获取并保存设备证书和设备私钥。
本文以成功获取并保存设备凭证为完成标准，不包含后续 MQTT 连接与设备上线。需要连接平台时，请参见 设备证书认证 。

功能介绍

证书认证设备动态注册用于减少生产过程中逐台烧录设备证书和设备私钥的工作。
设备首次启动后，使用产品级身份信息向平台发起注册请求。平台校验通过后，返回该设备对应的：

- 设备证书 `clientCert` 。
- 设备私钥 `clientKey` 。

设备端解密注册结果后，应立即将设备证书和设备私钥写入安全存储，后续使用已保存的凭证完成设备认证。

> ⓘ **注意：**
> 动态注册不会返回平台 CA 证书。平台 CA 证书应通过官方 SDK、控制台或产品文档单独获取。

动态注册模式

发起动态注册前，需要先根据实例类型和设备创建方式选择注册模式。

注册模式	设备创建要求	DeviceName 要求	平台处理方式
预创建设备	注册前必须在控制台创建设备。	必须与控制台中的设备名称完全一致。	校验已有设备并返回设备证书和设备私钥。
自动创建设备	无需提前创建设备。	由设备端提供，同一产品下不能重复。	平台自动创建设备并返回设备证书和设备私钥。

自动创建设备应设置合理的创建数量上限，避免因固件异常或 ProductSecret 泄露产生非预期设备。

设备状态与重复注册

设备是否允许动态注册取决于其激活状态。

设备状态	是否允许动态注册	处理结果
云端已创建，但从未注册、从未接入 MQTT。	允许	生成并返回 clientCert 和 clientKey。

已注册获得设备证书与私钥，但从未使用接入 MQTT。	允许	重新生成 clientCert 和 clientKey，旧证书和私钥失效。
曾使用设备证书与私钥成功接入 MQTT。	不允许	动态注册被拒绝。

说明：

- 控制台可能将前两种情况都显示为“未激活”，但两者的密钥状态不同。
- 设备收到 DeviceSecret 后，应立即保存。未完成保存前，不应再次发起动态注册。

注册流程

证书认证动态注册流程参见下图：

动态注册完成只表示设备获得了后续身份认证所需的设备证书与私钥，不表示设备已经连接平台或处于在线状态。

参数说明

动态注册涉及注册模式、设备身份、请求地址、请求头、请求正文和返回结果六类参数。
如需在控制台开启动态注册、选择注册模式并获取 ProductID、ProductSecret 和 DeviceName，请参见 启用动态注册 。

注册输入参数

参数	参数来源	说明	是否保密
动态注册地址	平台文档。	地域对应的动态注册服务地址。	否
ProductID	产品详情页。	标识设备所属产品。	否

ProductSecret	产品详情页。	用于生成注册请求签名和解密返回结果。	是
DeviceName	控制台或设备端。	产品下的设备唯一标识。	否
注册模式	产品详情页。	预创建设备或自动创建设备。	否

注册模式参数

注册模式用于确定设备是否需要提前创建，但不会作为字段发送至动态注册接口。

模式	对应配置	说明
预创建设备	`RegisterType=1`	设备须提前创建。
自动创建设备	`RegisterType=2`	平台按请求创建设备。

`RegisterType` 是产品动态注册配置，不是设备注册请求正文中的参数。

设备身份参数

参数	参数来源	说明	是否保密
`ProductID`	产品详情页。	标识设备所属产品。	否
`DeviceName`	• 预创建设备模式：控制台预创建设备名称。 • 自动创建设备模式：设备端注册时提供名称。	标识产品下的具体设备。	否
`ProductSecret`	产品详情页。	用于注册签名和结果解密。	是

`ProductSecret` 属于产品级敏感凭证。任何获得该凭证的设备或系统都可能以该产品身份发起注册请求，应限制其分发和读取范围。

请求地址

```
# 广州地域
https://ap-guangzhou.gateway.tencentdevices.com/device/register

# 曼谷地域
https://ap-bangkok.gateway.tencentdevices.com/device/register
```

参数	示例值	说明
请求协议	`HTTPS`	建议使用 HTTPS。
Host	`ap-guangzhou.gateway.tencentdevices.com`	动态注册服务域名。
URI	`/device/register`	动态注册接口路径。
请求方法	`POST`	固定值。
Content-Type	`application/json; charset=utf-8`	固定使用 JSON。

请求头参数

请求头	是否必选	说明
`Content-Type`	是	固定为 `application/json; charset=utf-8` 。
`Host`	是	动态注册服务域名。
`X-TC-Algorithm`	是	签名算法，填写 `hmacsha256` 。
`X-TC-Timestamp`	是	当前 Unix 时间戳，单位为秒。
`X-TC-Nonce`	是	客户端生成的随机数。
`X-TC-Signature`	是	使用 `ProductSecret` 生成的请求签名。

请求正文参数

参数	是否必选	类型	说明
`ProductId`	是	String	产品唯一标识。
`DeviceName`	是	String	设备在产品下的唯一名称。

请求示例：

```
{
```

```
  "ProductId": "您的ProductID",
  "DeviceName": "您的DeviceName"
}
```

参数名称区分大小写。请求正文使用 `ProductId`，不能写成 `ProductID`。

签名参数

待签名字符串按照以下顺序拼接：

```
POST
${Host}
/device/register

hmacsha256
${Timestamp}
${Nonce}
${RequestHash}
```

参数	生成方式
`Host`	动态注册服务域名。
`Timestamp`	当前 Unix 时间戳，单位为秒。
`Nonce`	客户端生成的随机数。
`RequestHash`	对实际请求正文进行 SHA-256 计算后得到的小写十六进制字符串。
`Signature`	使用 `ProductSecret` 对待签名字符串进行 HMAC-SHA256 计算，再进行 Base64 编码。

签名公式：

```
RequestHash = LowercaseHex(SHA256(RequestBody))

Signature = Base64(
    HMAC-SHA256(ProductSecret, StringToSign)
)
```

用于生成 `RequestHash` 的请求正文必须与实际发送的字节内容完全一致。

平台返回参数

参数	类型	说明
RequestId	String	本次请求的唯一标识。
Len	Int64	加密 Payload 的长度。
Payload	String	加密后的设备注册结果。
`State`	Int64	`0` 表示新增设备，`1` 表示设备已激活。

Payload 解密参数

字段	说明
encryptionType	证书认证固定为 `1` 。
clientCert	设备证书文件内容。
clientKey	与设备证书对应的设备私钥内容。

> **说明：**
> `clientCert` 和 `clientKey` 是一一对应的设备级凭证。动态注册结果中不包含平台 CA 证书。

凭证存储要求

设备证书

- 与 ProductID 和 DeviceName 对应保存。
- 可以作为设备身份标识使用。
- 不应与其他设备的私钥组合使用。
- 更新证书时，应同步更新对应的设备私钥。

设备私钥

- 必须写入非易失性存储。
- 不得输出至运行日志。
- 不得提交到代码仓库。
- 不得由多台设备共用。
- 建议存储在安全芯片、可信执行环境或受保护的 Flash 区域。

- 凭证损坏或丢失后，应按照设备生命周期管理方案重新发放。

平台 CA 证书

平台 CA 证书不通过动态注册返回。设备后续建立 MQTT 安全连接时，应：

- 使用官方 SDK 中内置或托管的 CA 证书，或从控制台及官方文档获取平台 CA 证书。
- 将 CA 证书随固件或安全配置预置到设备。
- 建立 CA 证书更新和 OTA 替换机制。

使用 Python 获取设备凭证

以下示例用于验证：

```
选择注册模式
→ 输入注册参数
→ 生成请求签名
→ 发起动态注册
→ 解密Payload
→ 脱敏显示设备凭证
```

示例以成功接收并解密设备私钥与设备证书作为运行成功标准，不负责 MQTT 接入。

> ⓘ **注意：**
> 本示例仅用于测试设备验证。示例只显示脱敏凭证，不保存完整凭证，不应直接用于生产量产。

准备运行环境

安装 AES 解密依赖：

```
# Windows PowerShell
python -m pip install "pycryptodome==3.23.0"

# Linux或macOS
python3 -m pip install "pycryptodome==3.23.0"
```

如果 Linux 提示 Python 环境由系统管理，请在虚拟环境中安装依赖。

准备设备参数

请在控制台开启动态注册、选择注册模式并获取 ProductID、ProductSecret 和 DeviceName，详情请参见 启用动态注册 。

创建注册脚本

将以下代码保存为 `dynamic_register_cert.py`：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import base64
import getpass
import hashlib
import hmac
import json
import random
import ssl
import sys
import time
import urllib.error
import urllib.request
from urllib.parse import urlparse

try:
    from Crypto.Cipher import AES
except ImportError:
    AES = None


DEFAULT_REGISTER_URL = (
    "https://ap-guangzhou.gateway.tencentdevices.com"
    "/device/register"
)

ALGORITHM = "hmacsha256"
REQUEST_TIMEOUT = 20


def required_input(prompt):
    """读取必填参数并清除首尾空格。"""
    value = input(prompt).strip()
    if not value:
        raise ValueError("输入内容不能为空")
```

```
    return value


def select_register_mode():
    """选择控制台中已经启用的动态注册模式。"""
    print("请选择控制台中已配置的动态注册模式：")
    print("1. 预创建设备")
    print("2. 自动创建设备（仅企业实例支持）")

    choice = required_input("请输入1或2：")

    if choice == "1":
        return "precreated", "预创建设备"

    if choice == "2":
        return "automatic", "自动创建设备"

    raise ValueError("动态注册模式只能输入1或2")


def get_register_url():
    """读取并校验完整的动态注册地址。"""
    prompt = (
        "动态注册地址（直接回车使用"
        f"{DEFAULT_REGISTER_URL}）："
    )
    register_url = input(prompt).strip() or DEFAULT_REGISTER_URL

    parsed = urlparse(register_url)

    if parsed.scheme.lower() != "https":
        raise ValueError(
            "动态注册地址必须使用HTTPS，并以https://开头"
        )

    if not parsed.hostname:
        raise ValueError("动态注册地址缺少有效域名")

    if parsed.path.rstrip("/") != "/device/register":
        raise ValueError(
```

```
            "动态注册地址的路径必须为/device/register"
        )

    if parsed.query or parsed.fragment:
        raise ValueError(
            "动态注册地址不能包含查询参数或锚点"
        )

    return register_url, parsed.hostname, "/device/register"


def get_device_name(register_mode):
    """根据动态注册模式提示用户填写设备名称。"""
    print("\n设备名称填写说明：")

    if register_mode == "precreated":
        print(
            "请输入已在控制台创建且处于未激活状态的设备名称，"
            "必须与控制台中的名称完全一致。"
        )
    else:
        print(
            "请输入需要注册的设备名称。平台将按照该名称自动"
            "创建设备，请确保名称符合设备命名规则且未被使用。"
        )

    return required_input("DeviceName：")


def build_request_body(product_id, device_name):
    """生成紧凑JSON请求正文。"""
    request_data = {
        "ProductId": product_id,
        "DeviceName": device_name,
    }

    return json.dumps(
        request_data,
        ensure_ascii=False,
        separators=(",", ":"),
```

```
    )


def create_signature(
    product_secret,
    hostname,
    canonical_uri,
    request_body,
    timestamp,
    nonce,
):
    """按照腾讯云动态注册签名规则生成请求签名。"""
    request_hash = hashlib.sha256(
        request_body.encode("utf-8")
    ).hexdigest()

    string_to_sign = "\n".join(
        [
            "POST",
            hostname,
            canonical_uri,
            "",
            ALGORITHM,
            str(timestamp),
            str(nonce),
            request_hash,
        ]
    )

    signature_bytes = hmac.new(
        product_secret.encode("utf-8"),
        string_to_sign.encode("utf-8"),
        hashlib.sha256,
    ).digest()

    return base64.b64encode(signature_bytes).decode("ascii")


def parse_platform_error(response_data):
    """提取平台返回的错误信息。"""
```

```
    if not isinstance(response_data, dict):
        return "Unknown", "平台返回格式异常", None

    response = response_data.get("Response", response_data)
    request_id = response.get("RequestId")

    error = response.get("Error")
    if isinstance(error, dict):
        code = error.get("Code", "Unknown")
        message = error.get("Message", "平台未返回错误详情")
        return code, message, request_id

    code = (
        response.get("Code")
        or response.get("code")
        or "Unknown"
    )
    message = (
        response.get("Message")
        or response.get("message")
        or "平台未返回错误详情"
    )

    return code, message, request_id


def send_register_request(
    register_url,
    hostname,
    canonical_uri,
    product_id,
    device_name,
    product_secret,
):
    """发送动态注册请求并返回平台Response对象。"""
    request_body = build_request_body(
        product_id,
        device_name,
    )
```

```
    timestamp = int(time.time())
    nonce = random.SystemRandom().randint(
        1,
        2147483647,
    )

    signature = create_signature(
        product_secret=product_secret,
        hostname=hostname,
        canonical_uri=canonical_uri,
        request_body=request_body,
        timestamp=timestamp,
        nonce=nonce,
    )

    headers = {
        "Content-Type": "application/json",
        "Host": hostname,
        "X-TC-Algorithm": ALGORITHM,
        "X-TC-Timestamp": str(timestamp),
        "X-TC-Nonce": str(nonce),
        "X-TC-Signature": signature,
    }

    request = urllib.request.Request(
        url=register_url,
        data=request_body.encode("utf-8"),
        headers=headers,
        method="POST",
    )

    tls_context = ssl.create_default_context()

    try:
        with urllib.request.urlopen(
            request,
            timeout=REQUEST_TIMEOUT,
            context=tls_context,
        ) as response:
            response_text = response.read().decode("utf-8")
```

```
    except urllib.error.HTTPError as exc:
        error_body = exc.read().decode(
            "utf-8",
            errors="replace",
        )

        try:
            error_data = json.loads(error_body)
            code, message, request_id = parse_platform_error(
                error_data
            )
        except json.JSONDecodeError:
            code = f"HTTP {exc.code}"
            message = error_body or str(exc.reason)
            request_id = None

        detail = f"{code} - {message}"
        if request_id:
            detail += f", RequestId: {request_id}"

        raise RuntimeError(
            f"动态注册请求失败: {detail}"
        ) from exc

    except urllib.error.URLError as exc:
        raise RuntimeError(
            f"无法连接动态注册服务: {exc.reason}"
        ) from exc

    except TimeoutError as exc:
        raise RuntimeError(
            "动态注册请求超时，请检查网络连接"
        ) from exc

    try:
        response_data = json.loads(response_text)
    except json.JSONDecodeError as exc:
        raise RuntimeError(
            "平台返回的数据不是有效的JSON"
```

```
        ) from exc

    response_object = response_data.get(
        "Response",
        response_data,
    )

    if "Error" in response_object:
        code, message, request_id = parse_platform_error(
            response_data
        )

        detail = f"{code} - {message}"
        if request_id:
            detail += f", RequestId: {request_id}"

        raise RuntimeError(
            f"动态注册失败: {detail}"
        )

    payload = response_object.get("Payload")
    if not payload:
        code, message, request_id = parse_platform_error(
            response_data
        )

        detail = f"{code} - {message}"
        if request_id:
            detail += f", RequestId: {request_id}"

        raise RuntimeError(
            "平台未返回Payload。"
            f"返回信息: {detail}"
        )

    return response_object


def decrypt_payload(payload, product_secret):
    """
```

```
    解密动态注册Payload。

    腾讯云动态注册协议使用：
    - AES-128-CBC
    - ProductSecret前16字节作为密钥
    - 16个ASCII字符“0”作为IV
    - 0x00字节填充
    """
    secret_bytes = product_secret.encode("utf-8")

    if len(secret_bytes) < 16:
        raise ValueError(
            "ProductSecret长度不足，无法生成AES-128密钥"
        )

    try:
        encrypted_data = base64.b64decode(
            payload,
            validate=True,
        )
    except Exception as exc:
        raise ValueError(
            "Payload不是有效的Base64数据"
        ) from exc

    if not encrypted_data:
        raise ValueError("Payload解码结果为空")

    if len(encrypted_data) % AES.block_size != 0:
        raise ValueError(
            "Payload密文长度不是AES块大小的整数倍"
        )

    # ProductSecret UTF-8编码后的前16字节。
    aes_key = secret_bytes[:16]

    # 注意：这里是16个ASCII字符“0”，不是16个0x00。
    iv = b"0" * 16

    try:
```

```
        cipher = AES.new(
            aes_key,
            AES.MODE_CBC,
            iv=iv,
        )
        decrypted_padded = cipher.decrypt(encrypted_data)
    except Exception as exc:
        raise ValueError(
            "Payload执行AES-CBC解密失败"
        ) from exc

    # 官方协议示例按首个0x00字节截断，不使用PKCS#7。
    zero_position = decrypted_padded.find(b"\x00")

    if zero_position >= 0:
        decrypted_data = decrypted_padded[:zero_position]
    else:
        decrypted_data = decrypted_padded

    if not decrypted_data:
        raise ValueError(
            "Payload解密结果为空，请检查ProductSecret"
        )

    try:
        decrypted_text = decrypted_data.decode("utf-8")
    except UnicodeDecodeError as exc:
        raise ValueError(
            "Payload解密结果不是有效的UTF-8文本。"
            "如果平台已返回Payload，请检查AES解密配置"
        ) from exc

    try:
        credential = json.loads(decrypted_text)
    except json.JSONDecodeError as exc:
        raise ValueError(
            "Payload解密成功，但结果不是有效的JSON数据"
        ) from exc

    if not isinstance(credential, dict):
```

```
        raise ValueError(
            "Payload解密后的数据格式不正确"
        )

    return credential


def get_first_line(value):
    """获取PEM内容首行，不显示正文。"""
    if not isinstance(value, str):
        return "-"

    lines = value.strip().splitlines()
    return lines[0] if lines else "-"


def show_certificate_result(
    credential,
    response_object,
    register_mode_name,
    product_id,
    device_name,
):
    """验证证书凭证并仅输出非敏感摘要。"""
    encryption_type = credential.get("encryptionType")

    try:
        encryption_type = int(encryption_type)
    except (TypeError, ValueError):
        raise ValueError(
            "返回结果缺少有效的encryptionType"
        )

    if encryption_type != 1:
        raise ValueError(
            "平台返回的不是证书认证凭证。"
            f"encryptionType：{encryption_type}，"
            "请检查产品认证方式"
        )
```

```
    client_cert = credential.get("clientCert")
    client_key = credential.get("clientKey")

    if not isinstance(client_cert, str) or not client_cert.strip():
        raise ValueError(
            "返回结果缺少有效的clientCert"
        )

    if not isinstance(client_key, str) or not client_key.strip():
        raise ValueError(
            "返回结果缺少有效的clientKey"
        )

    request_id = response_object.get("RequestId")
    state = response_object.get("State")
    payload_len = response_object.get("Len")

    print("\n证书认证设备动态注册成功。")
    print(f"注册模式：{register_mode_name}")
    print(f"ProductID：{product_id}")
    print(f"DeviceName：{device_name}")
    print("凭证类型：证书认证")
    print(f"encryptionType：{encryption_type}")

    if state is not None:
        state_text = {
            0: "新建设备",
            1: "已有设备",
            "0": "新建设备",
            "1": "已有设备",
        }.get(state, f"未知状态（{state}）")
        print(f"设备处理状态：{state_text}")

    if payload_len is not None:
        print(f"Payload长度：{payload_len}")

    print(
        "设备证书：已获取并成功解密"
        f"（{len(client_cert.encode('utf-8'))}字节，"
        f"首行：{get_first_line(client_cert)}）"
```

```
    )
    print(
        "设备私钥：已获取并成功解密"
        f"（{len(client_key.encode('utf-8'))}字节，"
        "内容不显示）"
    )

    if request_id:
        print(f"RequestId：{request_id}")

    print(
        "\n完整设备证书和设备私钥不会显示，"
        "也不会保存到文件。"
    )
    print(
        "本脚本仅用于验证动态注册及Payload解密流程。"
    )
    print(
        "生产环境必须在收到凭证后立即写入安全存储，"
        "不得依赖重复动态注册获取凭证。"
    )


def main():
    if AES is None:
        raise ImportError(
            "缺少运行依赖，请执行：\n"
            'python -m pip install "pycryptodome==3.23.0"'
        )

    print("证书认证设备动态注册验证工具")
    print(
        "本工具仅验证能否获取并解密设备证书和设备私钥。"
    )
    print(
        "输入及返回的完整凭证不会显示，也不会保存到文件。\n"
    )

    register_mode, register_mode_name = (
        select_register_mode()
```

```
    )

    print(f"\n已选择：{register_mode_name}")

    if register_mode == "precreated":
        print(
            "请确认目标设备已在企业实例中预先创建，"
            "且当前处于未激活状态。"
        )
    else:
        print(
            "请确认当前使用企业实例，并已在产品中"
            "启用自动创建设备。"
        )

    register_url, hostname, canonical_uri = (
        get_register_url()
    )

    product_id = required_input("ProductID：")
    device_name = get_device_name(register_mode)

    product_secret = getpass.getpass(
        "ProductSecret（输入内容不会显示）："
    ).strip()

    if not product_secret:
        raise ValueError("ProductSecret不能为空")

    print("\n即将发起动态注册。")
    print(f"注册模式：{register_mode_name}")
    print(f"动态注册地址：{register_url}")
    print(f"ProductID：{product_id}")
    print(f"DeviceName：{device_name}")
    print(
        "解密后的完整设备证书和设备私钥仅保留在"
        "本次程序运行内存中。"
    )
    print(
        "未激活设备重复注册可能刷新设备凭证，"
```

```
        "并使此前取得的凭证失效。"
    )

    confirm = input(
        "是否开始动态注册？[y/N]："
    ).strip().lower()

    if confirm != "y":
        print("已取消动态注册。")
        return 0

    print(f"\n正在请求：{register_url}")

    response_object = send_register_request(
        register_url=register_url,
        hostname=hostname,
        canonical_uri=canonical_uri,
        product_id=product_id,
        device_name=device_name,
        product_secret=product_secret,
    )

    credential = decrypt_payload(
        payload=response_object["Payload"],
        product_secret=product_secret,
    )

    show_certificate_result(
        credential=credential,
        response_object=response_object,
        register_mode_name=register_mode_name,
        product_id=product_id,
        device_name=device_name,
    )

    # 不执行任何文件写入。
    # 函数结束后，凭证仅等待随进程退出释放。
    return 0
```

```
if __name__ == "__main__":
    try:
        sys.exit(main())
    except KeyboardInterrupt:
        print("\n用户终止运行。")
        sys.exit(0)
    except Exception as exc:
        print(f"\n运行失败：{exc}")
        sys.exit(1)
```

运行脚本

```
# Windows PowerShell
python dynamic_register_cert.py

# Linux或macOS
python3 dynamic_register_cert.py
```

程序将依次要求输入：

```
注册模式
动态注册地址
ProductID
DeviceName
ProductSecret
```

ProductSecret 输入时不会显示在终端中，也不会写入文件。

运行成功结果

成功接收并解密设备凭证后，输出类似：

```
正在请求：https://ap-guangzhou.gateway.tencentdevices.com/device/register

证书认证设备动态注册成功。
注册模式：预创建设备
ProductID：*******
DeviceName：*******
凭证类型：证书认证
```

```
encryptionType: *
设备处理状态：已有设备
Payload长度：3007
设备证书：已获取并成功解密（1204字节，首行：-----BEGIN CERTIFICATE-----）
设备私钥：已获取并成功解密（1704字节，内容不显示）
RequestId: *******

完整设备证书和设备私钥不会显示，也不会保存到文件。
本脚本仅用于验证动态注册及Payload解密流程。
生产环境必须在收到凭证后立即写入安全存储，不得依赖重复动态注册获取凭证。
```

可在控制台设备页面查看注册成功的设备信息。

常见问题

平台返回 signature error

检查：

- ProductSecret 是否属于当前 ProductID。
- 签名是否基于实际发送的请求正文计算。
- Host 和请求路径是否与动态注册地址一致。
- X-TC-Algorithm 是否为 `hmacsha256`。
- 签名结果是否经过 Base64 编码。
- 请求正文中是否增加了空格、换行或其他字段。

平台返回 X-TC-Nonce error

每次请求均应生成新的随机正整数，不应使用空值、负数或重复的固定值。

注册请求被拒绝

检查：

- 当前是否为企业实例。
- 产品认证方式是否为证书认证。
- 产品是否已开启动态注册。
- 控制台配置的注册模式是否与当前场景一致。
- 预创建设备是否存在并处于未激活状态。
- 自动创建设备的 DeviceName 是否已经存在。
- 自动创建设备数量是否达到控制台限制。

解密 Payload 失败

检查：

- 是否使用当前产品的 ProductSecret。
- AES 密钥是否取 ProductSecret 的前16个字节。
- 是否使用 AES-128-CBC。
- IV 是否为16个字符 `0` 。
- 是否先对 Payload 进行 Base64 解码。
- 是否正确处理 PKCS#7 填充。

返回密钥认证凭证

如果 `encryptionType=2` ，说明当前产品为密钥认证产品。请确认 ProductID 是否正确，并改用 密钥认证设备动态注册 文档。

获取凭证后无法再次注册

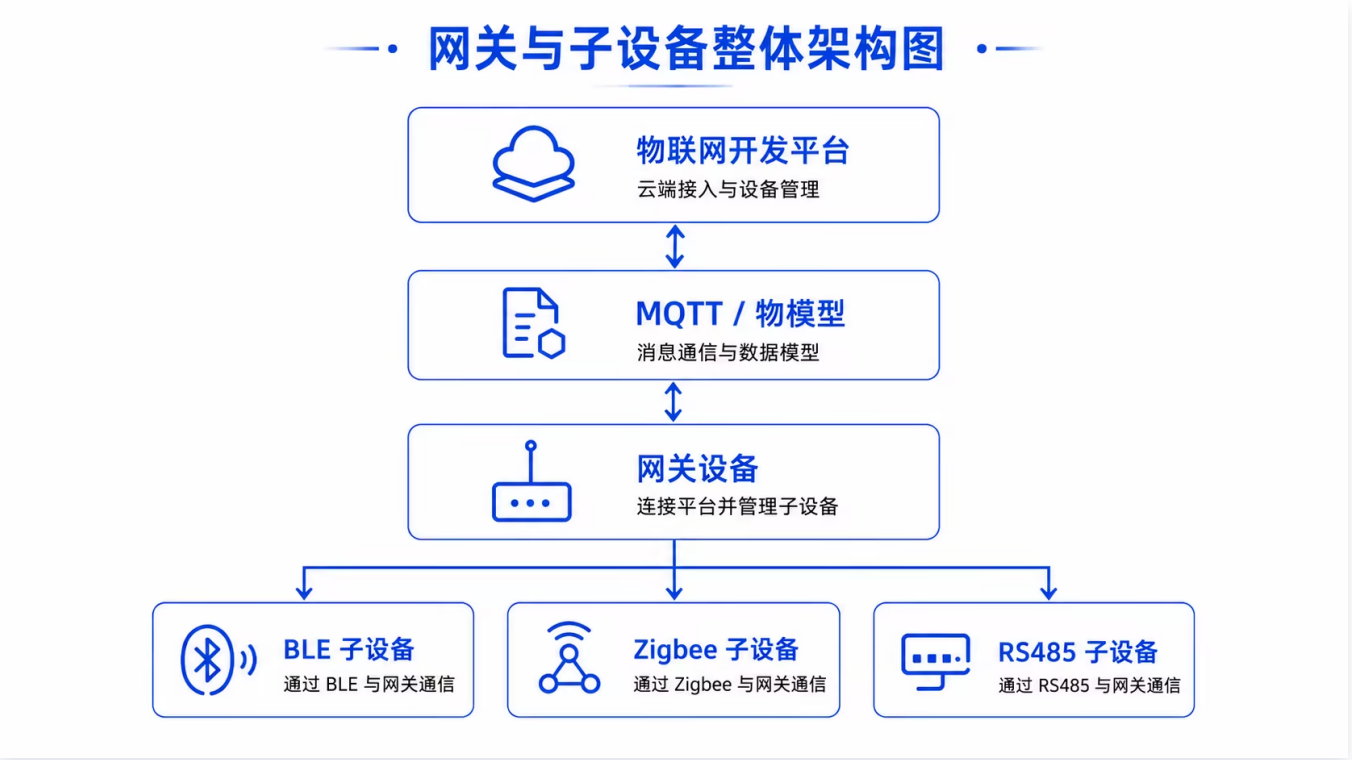
设备激活后，平台会拒绝再次动态注册。设备端应在首次注册成功后立即持久化设备证书和设备私钥，后续启动时直接读取本地凭证。

网关与子设备

最近更新时间：2026-09-11 16:05:32

功能概述

对于 BLE、Zigbee、RS485、433 等无法直接访问互联网的设备，可以通过网关接入物联网开发平台。
网关设备负责与物联网开发平台建立连接，并管理局域网中的子设备。子设备通过网关完成身份关联、上下线和消息通信，实现与云端的间接数据交互。腾讯云当前将设备类型划分为普通设备、网关和子设备，其中子设备必须依托网关与物联网开发平台进行通信。
典型架构如下：



其中，网关承担两个方向的通信

通信方向	说明
北向通信	网关与物联网开发平台进行连接和数据交互。
南向通信	网关通过 BLE、Zigbee、RS485 或其他有线、无线方式连接和管理子设备。

腾讯云现有定义中，网关设备具备北向云端通信和南向子设备管理能力；子设备则通过网关代理与云端交互。

> **说明：**
> 子设备不直接连接物联网开发平台。子设备与云端之间的消息均由网关进行代理。

设备类型

使用网关与子设备能力前，需要根据设备的联网能力和业务角色选择对应的设备类型。

设备类型	是否直接连接平台	主要作用
普通设备	是	直接连接物联网开发平台并进行消息通信，不挂载子设备。
网关设备	是	连接物联网开发平台，同时管理和代理子设备通信。
子设备	否	通过网关间接连接物联网开发平台。

例如，具备 Wi-Fi 或蜂窝网络能力的家庭网关可以创建为**网关设备**；通过 BLE 与网关通信的温湿度传感器可以创建为**子设备**。

网关与子设备的拓扑关系

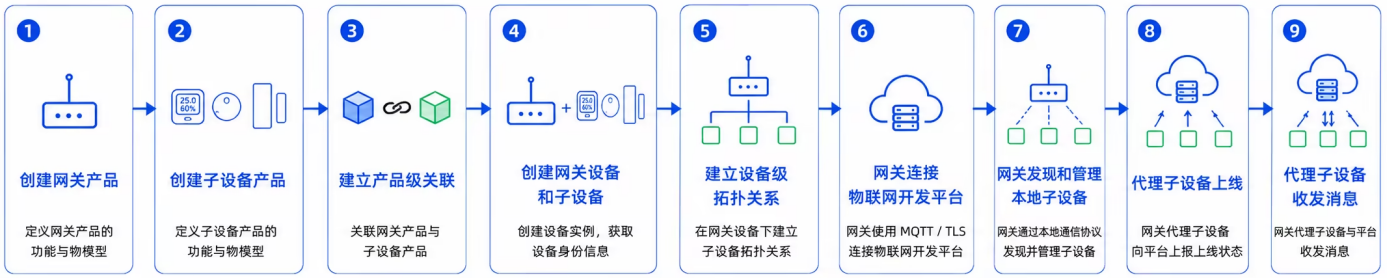
网关和子设备之间的关联关系称为**拓扑关系**。
只有当网关与子设备建立拓扑关系后，网关才能代理该子设备完成上下线和消息通信。
拓扑关系包括两个层级。

- **产品级关联**
- **设备级关联**

设备级关联可以预先在控制台完成，也可以由网关在运行过程中动态建立。动态绑定的前提是对应的网关产品和子设备产品已经完成产品级关联。

接入流程

网关与子设备的整体接入过程如下：



整个流程可以分为三个阶段：

阶段	主要工作
云端配置	创建网关产品、子设备产品和对应设备，并建立产品之间的关联关系。
设备接入	网关连接物联网开发平台，并建立或获取与子设备之间的拓扑关系。
子设备通信	网关代理子设备上线，并代理子设备与云端进行消息收发。

网关只需要建立一条 MQTT 连接。子设备不单独与物联网开发平台建立 MQTT 连接，而是复用网关已经建立的连接，由网关代理完成上下线和业务消息收发。

前提条件

开始网关与子设备接入前，请确认：

- 已创建网关产品和子设备产品，详见 创建和删除产品 ，并建立产品级关联关系。
- 已创建网关设备和对应子设备，详见 创建设备 。
- 已获取网关设备的 ProductID、DeviceName 及 MQTT 认证信息。
- 网关已经能够使用自身身份连接物联网开发平台。
- 已确定设备级拓扑关系的建立方式：提前在平台建立设备级拓扑关系，详见 网关与子设备绑定 ，或由网关运行时动态绑定子设备。
- 网关能够通过 BLE、Zigbee、RS485、UART 等本地协议识别并与实际子设备通信。

如果子设备尚未获得平台设备身份，并需要在首次接入时创建设备，请先完成子设备动态注册。

> ⓘ **说明：**

- 动态注册用于建立子设备的平台身份；拓扑绑定用于建立子设备与网关之间的管理关系，两者属于不同流程。
- 子设备不单独建立 MQTT 连接，后续拓扑管理、上下线和业务消息均复用网关已经建立的 MQTT 连接。

网关管理 Topic

网关通过以下 Topic 完成子设备拓扑查询、绑定、解绑和上下线等管理操作：

Topic	方向	QoS	用途
`$gateway/operation/{GatewayProductID}/{GatewayDeviceName}`	网关 → 平台	0	发送管理请求及应答。
`$gateway/operation/result/{GatewayProductID}/{GatewayDeviceName}`	平台 → 网关	1	接收操作结果和平台通知。

其中，Topic 中的 ProductID 和 DeviceName 均为**网关设备自身身份**；被管理的子设备身份通过 Payload 中的 `product_id` 和 `device_name` 指定。
网关连接成功后，应订阅操作结果 Topic：

```
gateway_result_topic = (
    f"$gateway/operation/result/"
    f"{gateway_product_id}/{gateway_device_name}"
)

client.subscribe(gateway_result_topic, qos=1)
```

ⓘ **注意：**

- 网关管理请求使用 QoS 0。
- MQTT Publish 成功只表示请求已发送，不表示绑定、上线等业务操作已经完成；最终结果应以操作结果 Topic 返回的 `result` 为准。

使用注意事项

- 子设备不会单独建立 MQTT 连接，业务通信复用网关 MQTT 连接。
- 网关管理 Topic 使用网关自身 ProductID 和 DeviceName，子设备身份通过 Payload 指定。
- 建立拓扑关系不代表子设备在线，仍需要执行代理上线。
- MQTT 管理请求发送成功不代表业务操作成功，应以结果 Topic 返回的 `result` 为准。

- BLE、Zigbee、RS485、UART 等南向协议和设备发现逻辑由开发者自行实现。
- 子设备业务消息应使用子设备自身的物模型或自定义 Topic，不应使用 `$gateway/operation/...`。
- 网关应及时处理平台拓扑变化以及实际子设备上下线，保持平台状态和本地状态一致。

接入步骤

步骤 1：网关连接平台

网关首先使用自身 ProductID、DeviceName 和认证信息建立 MQTT 连接。
连接成功后：

1. 订阅网关操作结果 Topic。
2. 初始化本地子设备信息。
3. 查询或恢复已有拓扑关系。
4. 根据实际子设备状态执行绑定、上线和业务通信。

步骤 2：维护子设备身份并同步拓扑

网关需要维护实际子设备与平台子设备身份之间的映射关系。

实际设备标识	示例	平台侧身份
BLE MAC	`AA:BB:CC:DD:EE:FF`	`SENSOR_PRODUCT / sensor_01`
Zigbee 地址	`0x1234`	`SENSOR_PRODUCT / sensor_02`
RS485 地址	`01`	`METER_PRODUCT / meter_01`
其他私有设备 ID	`device-1001`	对应 ProductID / DeviceName

网关启动后，可以查询平台当前已经绑定的子设备：

```
{
  "type": "describe_sub_devices"
}
```

平台返回：

```
{
  "type": "describe_sub_devices",
  "payload": {
    "devices": [
      {
```

```
        "product_id": "SENSOR_PRODUCT",
        "device_name": "sensor_01"
      }
    ]
  }
}
```

根据查询结果：

状态	后续处理
子设备已绑定	可进入代理上线流程。
子设备未绑定	提前建立拓扑关系或执行动态绑定。
本地与平台记录不一致	根据业务策略同步本地状态。

步骤 3：动态绑定子设备（可选）

如果需要在网关运行过程中添加新的子设备，可以发送 `bind` 请求建立设备级拓扑关系。
动态绑定支持设备密钥认证和设备证书认证。
需要准备：

参数	说明
`ProductID`	子设备所属产品 ID。
`DeviceName`	子设备名称。
认证信息	DeviceSecret 或证书相关认证信息。
`random`	随机数。
`timestamp`	Unix 时间戳，单位为秒。
`signature`	子设备身份签名。

以设备密钥认证为例，签名原文为：
`{ProductID}{DeviceName};{random};{timestamp}`
核心实现：

```
random_value = random.randint(1, 2147483647)
timestamp = int(time.time())
```

```
sign_text = (
    f"{sub_product_id}{sub_device_name};"
    f"{random_value};{timestamp}"
)

digest_hex = hmac.new(
    sub_device_secret.encode(),
    sign_text.encode(),
    hashlib.sha1,
).hexdigest()

signature = base64.b64encode(
    digest_hex.encode()
).decode()
```

绑定请求：

```
{
  "type": "bind",
  "payload": {
    "devices": [{
      "product_id": "SENSOR_PRODUCT",
      "device_name": "sensor_01",
      "signature": "******",
      "random": 12345,
      "timestamp": 1786610239,
      "signmethod": "hmacsha1",
      "authtype": "psk"
    }]
  }
}
```

平台返回对应 `bind` 结果， `result = 0` 表示绑定成功。

> **说明：**
> 动态绑定只建立设备级拓扑关系，不代表子设备已经在线。

步骤 4：代理子设备上线

当实际子设备已经接入网关，并且与当前网关存在拓扑关系后，网关发送 `online` 请求：

```
{
  "type": "online",
  "payload": {
    "devices": [{
      "product_id": "SENSOR_PRODUCT",
      "device_name": "sensor_01"
    }]
  }
}
```

平台返回：

```
{
  "type": "online",
  "payload": {
    "devices": [{
      "product_id": "SENSOR_PRODUCT",
      "device_name": "sensor_01",
      "result": 0
    }]
  }
}
```

`result = 0` **表示代理上线成功。**

操作	含义
绑定	建立网关与子设备的管理关系。
上线	表示子设备当前已经通过网关接入平台。

步骤 5：代理子设备业务通信

子设备上线后，可以复用网关已经建立的 MQTT 连接进行业务通信。

> ⓘ **说明：**
> **复用的是网关 MQTT 连接，不是网关管理 Topic。**
> `$gateway/operation/...` **只用于子设备管理，业务数据仍使用子设备自身对应的业务 Topic。**

根据业务需要，可以选择：

通信方式	适用场景
物模型	属性、事件、行为等标准化设备能力。
自定义 Topic	需要自行定义 Topic 和 Payload 的业务消息。

使用物模型

网关将 BLE、Zigbee、RS485 等南向数据解析后，转换成对应子设备的物模型数据，并通过网关 MQTT 连接发送。
云端下发物模型消息时，网关根据目标 ProductID 和 DeviceName 找到实际子设备，再转换成本地协议发送。

使用自定义 Topic

网关也可以使用子设备自身的自定义 Topic 代理消息：

```
topic = (
    f"{sub_product_id}/"
    f"{sub_device_name}/custom/report"
)

client.publish(
    topic,
    payload=payload,
    qos=1,
)
```

接收下行消息时订阅对应子设备 Topic：

```
topic = (
    f"{sub_product_id}/"
    f"{sub_device_name}/custom/control"
)

client.subscribe(topic, qos=1)
```

收到消息后，根据 Topic 中的 ProductID 和 DeviceName 定位实际子设备，并转换成本地协议发送。

步骤 6：同步云端拓扑变化

网关与子设备的拓扑关系也可能通过 App、业务后台等云端操作发生变化。网关需要处理平台主动下发的拓扑通知，并同步本地状态。

拓扑变化

平台下发：

```
{
  "type": "change",
  "payload": {
    "status": 1,
    "devices": [{
      "product_id": "SENSOR_PRODUCT",
      "device_name": "sensor_01"
    }]
  }
}
```

其中：

status	含义
1	增加绑定关系。
0	移除绑定关系。

网关更新本地拓扑后，应返回对应处理结果：

```
{
  "type": "change",
  "payload": {
    "status": 1,
    "devices": [{
      "product_id": "SENSOR_PRODUCT",
      "device_name": "sensor_01",
      "result": 0
    }]
  }
}
```

全部解绑

平台可能下发：

```
{
  "type": "unbind_all"
}
```

网关应清理本地绑定和相关会话状态，并返回：

```
{
  "type": "unbind_all",
  "payload": {
    "result": 0
  }
}
```

> **说明：**
> 对于 `bind`、`online`、`offline`、`unbind` 等请求，应根据返回消息中的 `type`、`product_id` 和 `device_name` 匹配对应操作，以 `result` 判断业务结果。

步骤 7：代理子设备下线和解绑

当网关检测到实际子设备离线时，应向平台同步子设备状态。
不同南向协议可以根据实际机制判断离线，例如：

南向协议	常见离线判断方式
BLE	连接断开。
Zigbee	离线检测或心跳超时。
RS485	连续通信超时或失败。

发送下线请求：

```
{
  "type": "offline",
  "payload": {
    "devices": [{
      "product_id": "SENSOR_PRODUCT",
      "device_name": "sensor_01"
    }]
  }
```

```
}
```

平台返回对应 `offline` 结果，`result = 0` 表示下线成功。
如果同时需要解除该子设备与网关的拓扑关系，再发送：

```
{
  "type": "unbind",
  "payload": {
    "devices": [{
      "product_id": "SENSOR_PRODUCT",
      "device_name": "sensor_01"
    }]
  }
}
```

`unbind` 返回 `result = 0` 后，表示设备级拓扑关系解除。

常见问题

现象	排查建议
网关已连接，但子设备无法上线。	检查是否已建立设备级拓扑关系，并确认 `online` 返回的 `result`。
动态绑定失败。	检查子设备 ProductID、DeviceName、认证信息、时间戳和签名。
Publish 成功但子设备仍离线。	MQTT Publish 成功不代表上线完成，应等待 `online` 业务结果。
子设备业务消息无法上报。	检查是否使用子设备自身的业务 Topic，以及 ProductID、DeviceName 是否正确。
云端下发后实际设备未收到。	检查子设备 Topic 订阅、本地身份映射及南向协议转发。
云端解绑后网关仍保留旧设备。	检查是否处理 `change` 或 `unbind_all` 并同步本地状态。
网关重启后拓扑状态不确定。	重新发送 `describe_sub_devices` 查询平台当前绑定关系。

消息通信
物模型通信

最近更新时间：2026-09-11 16:05:32

本文将介绍设备连接物联网开发平台后，如何通过物模型 Topic 上报设备属性和事件，以及接收云端属性控制和行为调用。
物模型用于将设备能力以统一的数据结构进行描述。产品定义物模型后，平台可以按照属性、事件和行为理解设备数据，并对设备上报的数据格式进行校验。

功能介绍

物模型（Thing Model）是设备在云端的数字化描述，通过属性、事件和行为三个维度定义产品能力。

类型	含义	典型场景
属性 Property	设备当前状态或可配置参数。	开关、温度、亮度。
事件 Event	设备主动上报的业务事件。	告警、故障、信息。
行为 Action	云端调用设备执行具体动作。	开锁、调整音量、执行任务。

其中，属性可由设备主动上报；对于可写属性，云端也可以向设备下发设置请求。事件由设备主动上报，事件类型包括 `alert`、`fault` 和 `info`。行为由云端发起调用，设备执行后返回结果。
物模型通信使用平台规定的 Topic 和 JSON 协议。设备上报的数据需要与产品中定义的属性、事件和行为保持一致，平台会对物模型数据进行校验，不符合物模型定义的数据不会被正常存储和展示。

物模型 Topic

物模型主要使用以下 Topic：

能力	上行 Topic	下行 Topic
属性	`$thing/up/property/{ProductID}/{DeviceName}`	`$thing/down/property/{ProductID}/{DeviceName}`
事件	`$thing/up/event/{ProductID}/{DeviceName}`	`$thing/down/event/{ProductID}/{DeviceName}`
行为	`$thing/up/action/{ProductID}/{DeviceName}`	`$thing/down/action/{ProductID}/{DeviceName}`

不同 Topic 对应不同物模型语义：

- 属性 Topic 用于属性上报、属性控制及相关响应。
- 事件 Topic 用于事件上报及事件响应。
- 行为 Topic 用于行为调用及行为执行结果返回。

通信流程

物模型通信主要包含两类链路：

物模型通信基于平台定义的 Topic 和 JSON 协议进行交互

前提条件

使用物模型通信前，请确认：

- 已创建产品和设备，详见 创建产品 和 创建设备 。
- 已在产品中完成物模型定义。
- 设备已完成 MQTT 身份认证并成功连接物联网开发平台。
- 设备端使用的属性标识符、事件标识符、行为标识符及对应参数，与产品物模型定义保持一致。

物模型协议使用这些标识符描述具体设备能力，因此设备端不能自行修改字段名称或数据类型。

使用限制

单设备 MQTT 上行消息存在速率限制：

QoS	单设备最大上行速率
QoS 0	30 条/秒

QoS 1	10 条/秒

超过限制时，可在设备行为日志中看到：

```
Fail, reach max limit
```

使用注意事项

- 物模型 Topic 由平台定义，不需要像自定义 Topic 一样创建 Topic 类。
- 设备上报和接收的数据必须遵循物模型协议规定的 JSON 结构。
- 属性、事件、行为标识符以及参数必须与产品物模型定义保持一致。
- 属性下行 Topic 中可能同时包含 `report_reply` 、 `control` 、 `get_status_reply` 等消息，应结合 `method` 判断具体消息类型。
- 属性控制完成后，应返回 `control_reply` ；行为执行完成后，应返回 `action_reply` 。
- `clientToken` 用于关联请求和响应，设备回复时应保持一致。
- 事件的 `type` 应使用 `alert` 、 `fault` 或 `info` 。
- `timestamp` 统一使用毫秒级 Unix 时间戳；如果无特殊时间需求，可以不填写，由平台使用当前系统时间。
- `timestamp` 与服务端当前时间相差超过24小时时，平台会返回 `405` 。
- 行为调用后，设备应在**5秒内**返回 `action_reply` ，否则平台会判定调用超时。
- 物模型数据不符合定义时，平台不会正常存储对应数据。
- 单设备消息上报应遵守 QoS 对应的速率限制。

操作步骤

步骤 1：订阅物模型下行 Topic

设备连接平台后，需要根据实际业务订阅对应的物模型下行 Topic，用于接收平台响应、属性控制及行为调用。以下代码以 Python Paho MQTT 为例，假设 MQTT Client 已经完成身份认证并成功连接平台。

```
property_down_topic = (
    f"$thing/down/property/{product_id}/{device_name}"
)

event_down_topic = (
    f"$thing/down/event/{product_id}/{device_name}"
)

action_down_topic = (
    f"$thing/down/action/{product_id}/{device_name}"
```

```
)

client.subscribe(property_down_topic, qos=1)
client.subscribe(event_down_topic, qos=1)
client.subscribe(action_down_topic, qos=1)
```

设备需要保持 MQTT 消息循环运行：

```
client.loop_forever()
```

实际项目只需要订阅业务使用到的 Topic，不要求同时使用属性、事件和行为。

步骤 2：上报设备属性

设备属性用于描述设备当前状态，例如开关状态、温度和亮度。
属性上报 Topic：

```
$thing/up/property/{ProductID}/{DeviceName}
```

设备按照物模型属性协议构造消息，例如：

```
{
  "method": "report",
  "clientToken": "123",
  "params": {
    "power_switch": 1,
    "color": 1,
    "brightness": 32
  }
}
```

主要字段：

字段	说明
`method`	属性上报固定为 `report` 。
`clientToken`	消息 ID，用于关联请求和响应。
`timestamp`	Unix 时间戳，单位为毫秒，可选；不填写时由平台使用当前系统时间。

`params`	本次上报的属性键值。

如果无特殊时间需求，可以不填写 `timestamp` 。如需填写，应使用当前毫秒级 Unix 时间戳；上报时间与服务端当前时间相差超过24小时时，平台会返回 `405` 。

Python 示例：

```
import json
import time

topic = (
    f"$thing/up/property/"
    f"{product_id}/{device_name}"
)

payload = {
    "method": "report",
    "clientToken": "property-001",
    "timestamp": int(time.time() * 1000),
    "params": {
        "power_switch": 1,
        "brightness": 32
    }
}

client.publish(
    topic,
    json.dumps(payload),
    qos=1,
)
```

`params` 中的字段必须已在产品物模型中定义，并符合对应数据类型和取值范围。

平台处理完成后，会通过属性下行 Topic 返回 `report_reply` ：

```
{
  "method": "report_reply",
  "clientToken": "123",
  "code": 0,
  "status": ""
```

```
}
```

其中，`clientToken` 与属性上报请求保持一致，`code = 0` 表示平台成功处理本次属性上报。

步骤 3：接收并处理属性控制

对于可写属性，平台可以通过以下 Topic 向设备下发控制消息：

```
$thing/down/property/{ProductID}/{DeviceName}
```

典型控制消息：

```
{
  "method": "control",
  "clientToken": "123",
  "params": {
    "brightness": 66
  }
}
```

设备收到后，需要先判断 `method == "control"`，再根据 `params` 执行业务逻辑。

```
def on_message(client, userdata, msg):
    data = json.loads(
        msg.payload.decode("utf-8")
    )

    if (
        msg.topic == property_down_topic
        and data.get("method") == "control"
    ):
        handle_property_control(data)
```

这里必须判断 `method`，因为属性下行 Topic 中还可能收到 `report_reply`、`get_status_reply` 等消息，不能把所有属性下行消息都当成控制指令。
设备完成控制后，需要向属性上行 Topic 返回 `control_reply`：

```
{
  "method": "control_reply",
```

```
  "clientToken": "123",
  "code": 0,
  "status": ""
}
```

回复中的 `clientToken` 应与控制请求保持一致，用于平台关联本次请求和响应。

步骤 4：上报设备事件

事件适用于设备主动上报告警、故障和信息类业务状态。
事件上报 Topic：

```
$thing/up/event/{ProductID}/{DeviceName}
```

事件响应 Topic：

```
$thing/down/event/{ProductID}/{DeviceName}
```

事件请求示例：

```
{
  "method": "event_post",
  "clientToken": "123",
  "version": "1.0",
  "eventId": "PowerAlarm",
  "type": "fault",
  "params": {
    "Voltage": 2.8,
    "Percent": 20
  }
}
```

主要字段：

字段	说明
`method`	固定为 `event_post`。
`clientToken`	消息 ID，用于关联请求和响应。

`version`	协议版本，当前默认 `1.0` 。
`eventId`	产品物模型中定义的事件标识。
`type`	事件类型： `alert` 、 `fault` 或 `info` 。
`timestamp`	事件发生时间，Unix 时间戳，单位为毫秒，可选。
`params`	产品物模型中定义的事件参数。

其中， `eventId` 和 `type` 是事件上报的关键字段，设备需要根据产品物模型定义填写对应事件标识和事件类型。
如果需要携带事件发生时间，应填写当前毫秒级 Unix 时间戳；无特殊时间需求时可以省略 `timestamp` 。
平台处理后通过事件下行 Topic 返回：

```
{
  "method": "event_reply",
  "clientToken": "123",
  "version": "1.0",
  "code": 0,
  "status": "",
  "data": {}
}
```

`code = 0` 表示事件上报成功。

步骤 5：接收并处理行为调用

行为适用于云端调用设备执行具体动作，并实时获取执行结果的场景。
行为调用下行 Topic：

```
$thing/down/action/{ProductID}/{DeviceName}
```

设备执行完成后，通过以下 Topic 返回结果：

```
$thing/up/action/{ProductID}/{DeviceName}
```

例如云端调用设备调整音量：

```
{
```

```
  "method": "action",
  "clientToken": "20a4ccfd-d308-****-86c6-5254008a4f10",
  "actionId": "control_volume",
  "params": {
    "action": "set",
    "volume": 50
  }
}
```

主要字段：

字段	说明
`method`	行为调用固定为 `action` 。
`clientToken`	本次行为调用 ID。
`actionId`	产品物模型中定义的行为标识。
`timestamp`	行为调用时间，Unix 时间戳，单位为毫秒，可选。
`params`	行为输入参数。

如果需要填写 `timestamp` ，应使用当前毫秒级 Unix 时间戳；否则可以省略。
设备根据 `actionId` 和 `params` 执行业务逻辑后，需要返回 `action_reply` ：

```
{
  "method": "action_reply",
  "clientToken": "20a4ccfd-d308-****-86c6-5254008a4f10",
  "code": 0,
  "status": "",
  "response": {
    "volume": 50
  }
}
```

其中：

字段	说明
`method`	行为回复固定为 `action_reply` 。

`clientToken`	与行为调用请求保持一致。
`code`	行为执行结果，`0` 表示成功。
`status`	执行结果或错误信息。
`response`	物模型中定义的行为输出参数。

行为回复中的返回参数必须放在 `response` 中，而不是 `params` 。
设备需要在5秒内返回行为执行结果，否则平台会判定本次行为调用超时。

步骤 6：统一处理下行消息

如果同时使用属性、事件和行为，可以先根据 Topic 将下行消息分发到对应处理逻辑，再根据 `method` 判断具体消息类型。

```
def on_message(client, userdata, msg):
    data = json.loads(
        msg.payload.decode("utf-8")
    )

    method = data.get("method")

    if msg.topic == property_down_topic:
        if method == "control":
            handle_property_control(data)
        elif method == "report_reply":
            handle_property_report_reply(data)
        elif method == "get_status_reply":
            handle_status_reply(data)

    elif msg.topic == event_down_topic:
        if method == "event_reply":
            handle_event_reply(data)

    elif msg.topic == action_down_topic:
        if method == "action":
            handle_action(data)
```

如果处理 `get_status_reply` ，属性数据位于 `data.reported` 下，例如：

```
{
  "method": "get_status_reply",
  "code": 0,
  "clientToken": "123",
  "type": "report",
  "data": {
    "reported": {
      "power_switch": 1,
      "brightness": 66
    }
  }
}
```

因此读取属性值时，应从 `data.reported` 获取，而不是直接从 `data` 根节点读取。
建议业务实现同时结合 `clientToken` 、 `eventId` 、 `actionId` 等字段关联请求和响应。

验证物模型通信

设备成功上报合法物模型数据后，可以在设备详情的物模型数据或相关日志页面查看平台处理结果。
验证时建议依次检查：

- 是否发布到了正确的物模型 Topic。
- Payload 是否为合法 JSON。
- `method` 是否正确。
- 属性、事件和行为标识符是否与产品物模型定义一致。
- 参数类型和取值范围是否符合物模型定义。
- 如填写 `timestamp` ，是否为毫秒级 Unix 时间戳，且与服务端当前时间相差不超过24小时。

常见问题

返回码 / 现象	排查建议
`400`	Payload 不是合法 JSON，检查 JSON 格式。
`403`	检查 `method` 是否正确，或属性、事件、行为标识符是否与产品物模型定义一致。
`405`	时间戳错误。确认 `timestamp` 为毫秒级 Unix 时间戳，并检查与服务端当前时间是否相差超过 24 小时。
`406`	参数不符合物模型定义，检查数据类型、枚举值、数值范围等。例如整型

	枚举不要使用字符串 `"0"` 、 `"1"` 。
`503`	平台服务端处理异常，可结合日志进行重试或进一步排查。
MQTT 已连接，但属性上报后没有数据。	检查属性 Topic、JSON 格式、 `method` 、属性标识符及数据类型。
云端修改属性后设备没有收到。	检查是否订阅 `$thing/down/property/...` ，并确认回调中正确处理 `method = control` 。
属性控制已执行，但平台仍认为失败。	检查是否返回 `control_reply` ，以及 `clientToken` 是否与控制请求一致。
事件上报后没有记录。	检查事件 Topic、 `eventId` 、 `type` 和 `params` 是否与产品物模型定义一致。
行为调用后平台提示超时。	检查设备是否订阅行为下行 Topic，并确认是否在5秒内返回 `action_reply` 。
行为回复后平台无法解析结果。	检查输出参数是否放在 `response` 中，而不是 `params` 。
`get_status_reply` 中读取不到属性值。	属性数据位于 `data.reported` 下，不在 `data` 根节点。
行为日志出现 `Fail, reach max limit` 。	检查设备是否超过单设备 MQTT 上行速率限制。
Payload 为二进制或私有协议。	建议使用自定义透传。
仅需传输自定义业务消息。	企业实例可使用自定义 Topic，具体以当前实例能力为准。

自定义 Topic 通信

最近更新时间：2026-09-11 16:05:33

本文将介绍设备连接物联网开发平台后，如何通过自定义 Topic 发布和订阅业务消息。
当物模型无法满足业务需求，或者需要自行定义 MQTT Payload 数据格式时，可以使用自定义 Topic。

功能介绍

自定义 Topic 是基于 MQTT 发布/订阅机制提供的自定义消息通道。
自定义 Topic 的前两个层级固定为：

```
${ProductID}/${DeviceName}
```

后续层级可以根据业务需要定义，例如：

```
${ProductID}/${DeviceName}/custom/report
${ProductID}/${DeviceName}/custom/control
```

其中：

Topic	用途
`${ProductID}/${DeviceName}/custom/report`	设备向云端上报业务数据。
`${ProductID}/${DeviceName}/custom/control`	设备接收云端下发的业务消息。

自定义 Topic 支持以下设备端操作权限：发布、订阅、发布和订阅。

> **说明：**
> - 自定义 Topic 只定义消息通信通道。Payload 的数据格式、字段和业务语义由设备端与业务系统自行约定。
> - 平台默认为每个设备提供 ${ProductID}/${DeviceName}/data Topic，该 Topic 具有发布和订阅权限，也可以直接用于自定义消息通信。

前提条件

在使用自定义 Topic 进行消息通信前，请确认已完成以下准备：

- 已创建产品和设备，详见 创建和删除产品 和 创建设备 。
- 设备已完成 MQTT 身份认证并成功连接物联网开发平台。

- 已配置需要使用的自定义 Topic。
- 已根据消息方向配置 Topic 的发布或订阅权限。

> **说明：**
> 设备实际发布或订阅的 Topic 必须与产品配置的自定义 Topic 保持一致，其中 `${ProductID}` 和 `${DeviceName}` 应替换为当前设备实际的产品 ID 和设备名称。

注意事项

- 设备实际使用的 Topic 必须与产品配置的自定义 Topic 保持一致。
- Topic 中的 ProductID 和 DeviceName 应与当前连接设备的身份一致。
- Topic 的发布、订阅权限应与实际消息方向一致。
- 自定义 Topic 的 Payload 格式和业务语义由设备端与业务系统自行定义。
- 自定义 Topic 与透传 Topic 是不同的通信能力；自定义 Topic 不触发云端解析。

通信流程

步骤 1：准备自定义 Topic

本示例分别使用一个 Topic 进行设备数据上报和消息下发：

Topic	设备权限	用途

${ProductID}/${DeviceName}/custom/report	发布	设备向平台上报业务数据。
${ProductID}/${DeviceName}/custom/control	订阅	设备接收业务消息。

> ⓘ **注意：**
> Topic 权限从设备端视角定义。“发布”表示设备向平台发送消息；“订阅”表示设备接收平台下发的消息。设备使用无对应权限的 Topic 时，平台会拒绝该操作。

步骤 2：订阅下行 Topic

设备连接平台后，需要订阅用于接收业务消息的自定义 Topic。
以下代码以 Python Paho MQTT 为例。示例假设 MQTT Client 已经完成身份认证并成功连接平台。

```
subscribe_topic = (
    f"{product_id}/{device_name}/custom/control"
)

result, mid = client.subscribe(
    subscribe_topic,
    qos=1
)

if result != 0:
    print(f"订阅请求失败：{result}")
else:
    print(f"正在订阅：{subscribe_topic}")
```

订阅成功后，设备需要保持 MQTT 消息循环运行，才能持续接收平台发送的消息。

```
client.loop_forever()
```

步骤 3：接收并处理下行消息

注册 `on_message` 回调处理已订阅 Topic 的消息：

```
def on_message(client, userdata, msg):
    payload = msg.payload.decode("utf-8")
    print(f"Topic：{msg.topic}")
    print(f"Payload：{payload}")
```

```
    # 根据业务协议解析 Payload，并执行设备逻辑

client.on_message = on_message
```

例如业务下发：

```
{
  "command": "set_mode",
  "mode": "eco"
}
```

设备收到后，可以根据自行约定的 Payload 格式解析 command、mode 等字段并执行对应业务逻辑。

> **说明：**
> 自定义 Topic 不规定 Payload 的字段结构和业务语义。设备端与业务系统需要自行约定消息格式，并确保双方实现一致。

步骤 4：发布上行消息

设备需要上报业务数据时，向具有发布权限的自定义 Topic 发布消息。
例如上报温度和湿度：

```
import json

publish_topic = (
    f"{product_id}/{device_name}/custom/report"
)

payload = {
    "temperature": 26.5,
    "humidity": 60
}

info = client.publish(
    publish_topic,
    payload=json.dumps(payload),
    qos=1
)
```

```
if info.rc != 0:
    print(f"消息发布失败：{info.rc}")
```

如果需要等待 QoS 1 消息发送完成，可以调用：

```
info.wait_for_publish()
```

自定义 Topic 的 Payload 可以根据实际业务自行定义，平台不会按照物模型结构解析其中的业务字段。

自定义 Topic 与其他 Topic 的区别

Topic 类型	主要用途	Payload 处理方式
物模型 Topic	属性、事件、行为等标准设备能力。	按物模型协议处理和校验。
透传 Topic	自定义透传数据与物模型之间转换。	-
自定义 Topic	自定义业务消息通信。	Payload 由业务自行定义和解析。

常见问题

现象	排查建议
MQTT 已连接，但发布消息失败。	检查 Topic 是否与产品配置一致，并确认设备具有发布权限。
设备订阅失败。	检查 Topic 是否正确，并确认设备具有订阅权限。
业务系统下发消息后设备未收到。	确认设备已经成功订阅对应 Topic，并保持 MQTT 消息循环运行。
收到消息但业务解析失败。	检查设备端与业务系统使用的 Payload 格式、字段和编码是否一致。
希望将私有协议数据转换为物模型数据。	不应使用自定义 Topic 的方式处理，请使用自定义透传及云端解析能力。

自定义透传协议

最近更新时间：2026-09-11 16:05:33

本文将介绍设备连接物联网开发平台后，如何通过自定义透传 Topic 收发业务数据。
自定义透传适用于设备已有私有数据协议，或需要使用二进制、非 JSON 等自定义数据格式进行通信的场景。设备与业务系统自行约定 Payload 的数据格式和业务语义，物联网开发平台不解析 Payload 中的业务内容。当前平台“基本概念”也将自定义透传定义为基于 MQTT 自由定义 Payload、平台不进行解析的通信方式。

功能介绍

创建产品时，数据协议默认采用物模型，也可以选择自定义协议进行透传。
自定义透传使用平台规定的上下行透传 Topic：

Topic	方向	用途
`$thing/up/raw/{ProductID}/{DeviceName}`	设备 → 云端	设备上报自定义透传数据。
`$thing/down/raw/{ProductID}/{DeviceName}`	云端 → 设备	设备接收自定义透传数据。

上行数据发布至 `$thing/up/raw/{ProductID}/{DeviceName}`，下行数据通过订阅 `$thing/down/raw/{ProductID}/{DeviceName}` 接收。
自定义透传只定义数据传输方式。Payload 中各字段的含义、长度、编码和业务协议由设备端与业务系统自行约定。

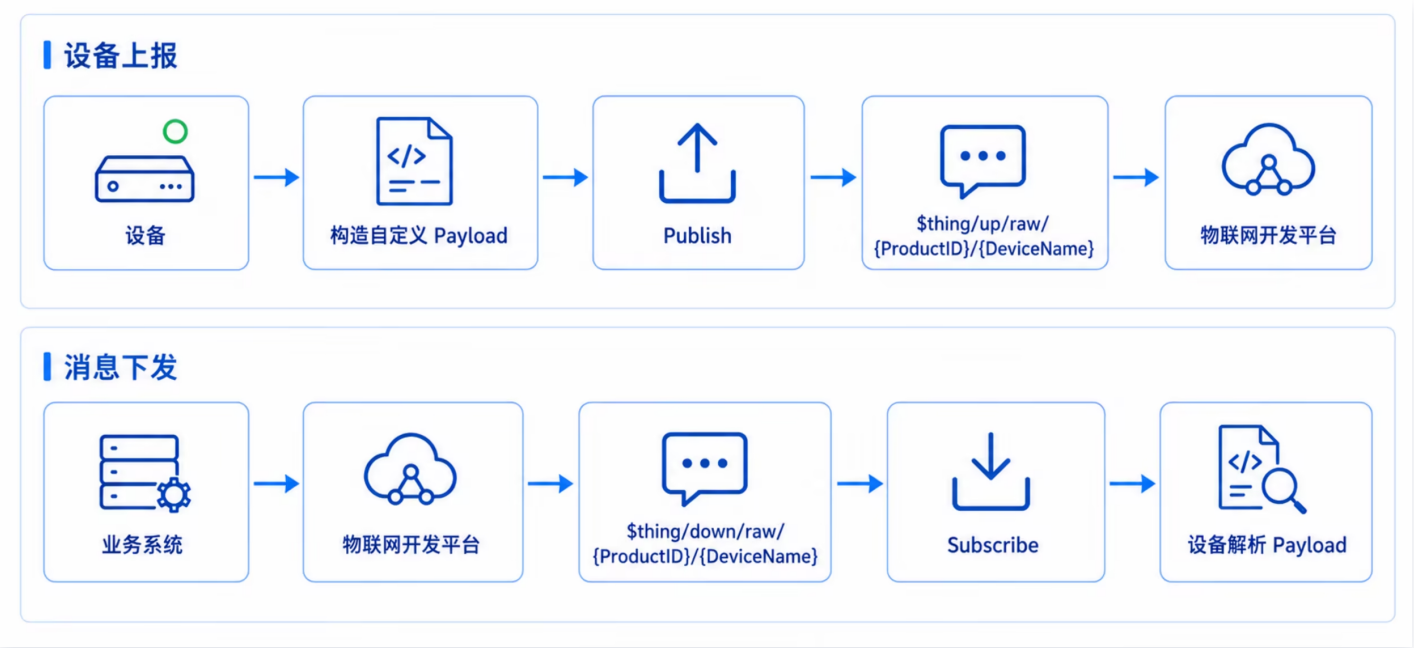
与物模型、自定义 Topic 的区别

对比项	物模型	自定义透传	自定义 Topic
数据语义	属性、事件、行为。	业务自行定义。	业务自行定义。
Topic	平台定义。	平台规定的 Raw Topic。	用户定义后续 Topic 层级。
Payload	遵循物模型协议。	自定义，常用于二进制、非 JSON 或私有协议。	自定义。
平台业务解析	按物模型处理。	不解析。	不解析。
典型场景	标准化设备能力。	已有私有设备协议、二进制协议。	自定义 MQTT 业务消息。

透传 Topic 用于非 JSON 格式业务数据，自定义 Topic 则允许用户自行定义 Topic 后续层级。

通信流程

通信流程参见下图：



前提条件

使用自定义透传进行通信前，请确认：

- 已创建数据协议为**自定义透传**的产品和设备。
- 设备已完成 MQTT 身份认证并成功连接物联网开发平台。
- 已确定设备端与业务系统之间使用的数据协议。
- 已明确 Payload 的字段、长度、数据类型、编码方式及字节序。

如果尚未完成设备 MQTT 连接，请先参见设备密钥认证或设备证书认证相关文档。

使用限制

项目	限制
MQTT 协议版本	MQTT 3.1.1
QoS	支持 QoS 0、QoS 1，不支持 QoS 2。
Retain Message	不支持。
单条 MQTT Publish 消息	最大16KB。

腾讯云当前 MQTT 接入协议明确不支持 QoS 2 和 Retain Message；产品限制规定单条 MQTT 发布消息最大为16KB。

注意事项

- 自定义透传使用平台规定的透传 Topic，不应与自定义 Topic 混用。
- Payload 的数据格式和业务语义由设备端与业务系统自行约定。
- ProductID 和 DeviceName 必须与当前连接设备身份一致。
- 上下行两端应保持字段定义、长度、数据类型和字节序一致。
- 二进制协议应显式完成序列化和反序列化，避免直接依赖设备内存结构。
- 设备需要保持 MQTT 消息循环运行，才能持续接收下行数据。
- 本文使用的12字节二进制协议仅为示例，不是平台强制协议。

操作步骤

步骤 1：定义透传数据格式

自定义透传不限定 Payload 的数据格式。开发者需要根据实际设备能力自行设计通信协议。
以下以一个简单的智能灯二进制协议为例。

上行数据格式

字节偏移	长度	字段	示例值	说明
0	2 Byte	Header	`AA 55`	帧头
2	1 Byte	MessageType	`01`	数据上报
3	1 Byte	Reserved	`00`	保留
4	4 Byte	ClientId	`00 00 00 00`	消息标识
8	1 Byte	PowerSwitch	`01`	开关状态
9	1 Byte	Color	`01`	颜色
10	1 Byte	Brightness	`38`	亮度
11	1 Byte	Reserved	`00`	保留

对应示例 Payload：`AA 55 01 00 00 00 00 00 01 01 38 00`
现有示例程序实际发送的上行数据即采用该字节结构，其中 `0x01` 表示数据上报、`0x20` 表示控制消息。

> **说明：**
> 上述协议仅用于说明自定义透传数据的组织方式，不是物联网开发平台规定的 Payload 格式。实际产品可根据自身协议自行定义。

步骤 2：订阅下行透传 Topic

设备连接平台后，需要订阅下行透传 Topic：

```
down_topic = (
    f"$thing/down/raw/"
    f"{product_id}/{device_name}"
)

client.subscribe(
    down_topic,
    qos=1,
)
```

其中，`ProductID` 和 `DeviceName` 应与当前 MQTT 连接对应的设备身份保持一致。
现有透传示例同样使用 QoS 1 订阅该下行 Topic。

步骤 3：接收并解析下行数据

设备通过 MQTT 消息回调接收下行二进制 Payload。

```
def on_message(client, userdata, msg):
    payload = msg.payload

    print(f"Topic: {msg.topic}")
    print(f"Payload: {payload.hex(' ').upper()}")

    # 根据自定义协议解析 Payload


client.on_message = on_message
```

例如设备收到：`AA 55 20 00 CC CC CC CC FF 01 FF FF`
其中 `0x20` 表示控制消息。现有示例实际接收的下行 Payload 也采用该格式。
例如，可以根据本文示例协议读取相应字段：

```
msg_type = payload[2]
power_switch = payload[8]
color = payload[9]
brightness = payload[10]
```

```
if msg_type == 0x20:
    # 执行设备控制逻辑
    pass
```

实际开发中，应按照设备自身协议约定的字段偏移、长度和编码方式解析 Payload。

步骤 4：发布上行透传数据

设备需要向平台上报数据时，首先按照自定义协议构造 Payload。
例如：

```
payload = bytes([
    0xAA, 0x55,
    0x01,
    0x00,
    0x00, 0x00, 0x00, 0x00,
    0x01,
    0x01,
    0x38,
    0x00,
])
```

然后发布至上行透传 Topic：

```
up_topic = (
    f"$thing/up/raw/"
    f"{product_id}/{device_name}"
)

client.publish(
    up_topic,
    payload=payload,
    qos=0,
)
```

现有实现也是将二进制 Payload 直接发布至 `$thing/up/raw/{ProductID}/{DeviceName}` 。
本文示例上行使用 QoS 0、下行订阅使用 QoS 1，这不是自定义透传 Topic 的强制要求。物联网开发平台支持 QoS 0 和 QoS 1，不支持 QoS 2。

步骤 5：业务系统下发透传数据

业务系统需要向设备下发数据时，可以通过物联网开发平台提供的**设备透传指令控制**能力向设备订阅的 Topic 发送消息。
对于二进制数据，业务系统可以先将原始 Payload 进行 Base64 编码，并在调用 `PublishMessage` 时设置：

参数	取值
ProductID	目标设备所属产品 ID。
DeviceName	目标设备名称。
Topic	`$thing/down/raw/{ProductID}/{DeviceName}`
Payload	Base64 编码后的二进制数据。
PayloadEncoding	`base64`
QoS	`0` 或 `1`。

当 `PayloadEncoding` 设置为 `base64` 时，平台会将业务系统提交的 Base64 内容转换回二进制数据后发送至设备。
例如设备需要收到：`AA 55 20 00 CC CC CC CC FF 01 FF FF`，业务系统应首先对该二进制数据进行 Base64 编码，再通过透传指令控制接口进行下发。设备收到消息后，按照 步骤 3 中定义的协议解析 Payload 并执行业务逻辑。

二进制协议设计建议

使用自定义透传时，协议本身由业务定义。建议至少明确以下内容：

项目	说明
帧头	用于识别数据帧及协议版本。
消息类型	区分数据上报、控制、应答等消息。
消息标识	用于请求、应答或消息去重。
字段偏移与长度	明确每个字段在 Payload 中的位置。
数据类型	明确整数、字符串、位字段等表示方式。
字节序	多字节整数应明确采用大端或小端。
无效值	明确字段不存在或未设置时的表示方式。

数据长度	可根据协议复杂度增加长度字段。
数据校验	对可靠性要求较高时可增加 CRC、Checksum 等校验字段。

常见问题

现象	排查建议
上行透传数据发送失败。	检查 MQTT 连接状态，以及上行 Topic 中的 ProductID、DeviceName 是否正确。
设备无法收到下行数据。	检查是否已成功订阅 `$thing/down/raw/{ProductID}/{DeviceName}`，并确认 MQTT 消息循环持续运行。
收到数据但解析结果异常。	检查字段偏移、长度、数据类型和字节序是否与业务协议一致。
相同结构在不同芯片上发送结果不同。	检查结构体 Padding 和 CPU 字节序，不建议直接发送结构体内存。
二进制下发后设备收到的数据不一致。	检查业务系统 Base64 编码是否正确，并确认下发接口设置 `PayloadEncoding=base64`。
自定义透传和自定义 Topic 有什么区别。	自定义透传使用平台规定的 Raw Topic；自定义 Topic 的 Topic 后续层级由用户自行定义。

规则引擎

最近更新时间：2026-09-11 16:05:33

本文将介绍规则引擎的作用、数据处理链路、核心能力和典型场景，帮助您判断是否需要使用规则引擎，并选择合适的数据转发目标。

功能介绍

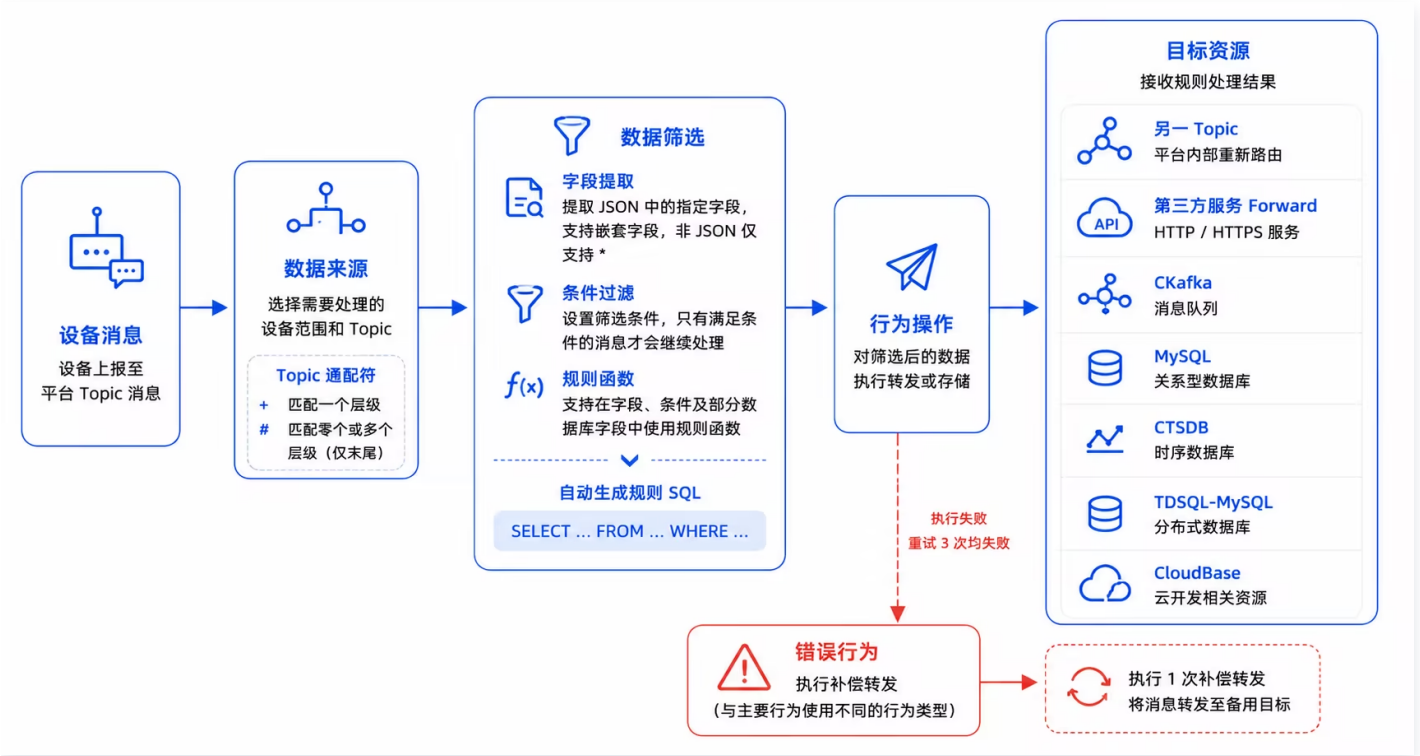
规则引擎用于处理进入物联网开发平台 Topic 的消息。您可以通过规则 SQL 选择消息来源、提取所需字段并设置筛选条件，再将符合条件的数据转发至其他 Topic、腾讯云服务或用户业务系统。
规则引擎主要解决以下问题：

- 哪些设备消息需要进入业务系统。
- 哪些数据满足转发条件。
- 数据需要转发到哪个目标。
- 转发失败后如何进行补偿和排查。

规则引擎位于设备消息通信与业务系统之间，用于完成数据筛选、路由和转发，不负责设备连接、身份认证及 MQTT 消息上报。

数据处理流程

规则引擎按照以下流程处理进入平台的 Topic 消息：
设备消息 → 数据来源 → 数据筛选 → 转发行为 → 目标服务
当主要转发行为执行失败时，可以进入配置的错误行为进行异常处理。



环节	说明
数据来源	指定需要进入规则处理的设备及 Topic。
数据筛选	提取消息字段并设置筛选条件，系统生成对应规则 SQL。
转发行为	将符合条件的数据转发至指定 Topic、云服务或业务系统。
错误行为	主要转发行为多次执行失败后，对异常消息进行补偿处理。

当前控制台的数据筛选配置会根据选择的字段、Topic 和条件自动生成规则 SQL，无需用户直接拼写完整 SQL。

前提条件

创建规则前，请确认：

- 已创建产品和设备，且设备能够正常向物联网开发平台上报消息。
- 已明确需要处理的设备消息及源 Topic。
- 如果需要转发至腾讯云服务或用户业务系统，目标资源已提前创建且状态正常。
- 如果目标资源涉及访问权限、网络或鉴权配置，请提前完成对应配置。

> **说明：**
> 规则引擎只处理已经进入物联网开发平台的消息，不负责设备连接、身份认证及 MQTT 消息上报。

使用限制

使用规则引擎时，需要注意以下限制：

项目	限制
规则数量	一个账号最多100条。
单条规则转发行为数量	最多10个。
非 JSON 数据	不支持字段筛选，只能使用 `*` 转发完整内容。
目标资源	目标服务实例需要处于正常可用状态。
消息去重	同一消息在特定异常或重试情况下可能被重复发送。

当前产品限制明确说明，为确保消息送达，同一条消息可能发生重复发送，因此对于数据库写入、业务接口等场景，建议下游系统根据业务需要实现幂等或去重逻辑。

使用注意事项

- 规则只处理与数据来源匹配的 Topic 消息。

- SQL 中使用的字段必须与设备实际上报 Payload 保持一致。
- 非 JSON 数据无法进行字段和条件筛选，只能整体转发。
- 消息不满足筛选条件时，不会执行转发行为。
- 规则需要启用后才开始处理后续进入平台的消息。
- 转发至云服务或第三方系统前，应确保目标资源状态、网络和权限配置正常。
- 如果业务不能接受重复消息，应在下游系统实现幂等处理。
- 对消息可靠性要求较高的场景，可以配置错误行为作为异常转发出口。

接入步骤

步骤 1：创建规则并选择数据来源

新建规则后，首先配置需要进入规则处理的数据来源。
数据来源主要包括：

配置项	说明
设备	指定产品下全部设备或具体设备。
Topic 类型	指定规则需要处理的消息类型。
Topic	根据产品、设备和消息类型确定实际消息来源。

当前规则引擎支持从不同类型的 Topic 中选择数据来源，例如物模型属性上报、事件上报、行为上报、自定义上报、设备状态变化以及网关子设备拓扑关系消息等。
规则只会处理与数据来源匹配的消息。例如，将数据来源配置为某产品的物模型属性上报 Topic 后，其他 Topic 中的消息不会进入该规则处理流程。

步骤 2：配置数据筛选

选择数据来源后，可以通过**字段**和**条件**设置需要提取和转发的数据，平台会根据配置自动生成对应的规则 SQL。

提取字段

对于 JSON 格式消息，可以指定需要提取的字段。
例如设备上报：

```
{
  "temperature": 32,
  "humidity": 60
}
```

仅需要转发温度和湿度时，可选择：`temperature, humidity`

如果需要转发完整消息，可以使用：`*`
对于嵌套 JSON，可以使用 `.` 访问对应字段。例如：

```
{
  "device_status": {
    "switch": "on"
  }
}
```

可以使用：`device_status.switch`
需要注意：

- 非 JSON 格式的数据不能按字段提取，只能使用 `*` 转发完整消息。
- 当前不支持通过规则 SQL 处理 JSON 数组和子 SQL。

设置筛选条件

对于 JSON 格式消息，可以设置条件，仅让满足条件的数据执行后续转发。
例如：`temperature > 30` 表示只有当设备上报的 `temperature` 大于30时，才执行后续行为。
常用条件包括：

类型	支持示例
比较	`=`、`<>`、`>`、`>=`、`<`、`<=`。
逻辑	`AND`、`OR`。
算术	`+`、`-`、`*`、`/`、`%`。
组合条件	使用 `( )` 组合多个条件。

例如：`temperature > 30 AND humidity > 70` 表示温度大于30且湿度大于70时才执行转发。
当前控制台会根据数据来源、字段和条件自动生成对应 SQL，例如：

```
SELECT temperature, humidity
FROM 'PRODUCT_ID/DEVICE_NAME/data'
WHERE temperature > 30
```

规则 SQL 可以理解为：

SQL 部分	作用

`SELECT`	指定需要提取的数据字段。
`FROM`	指定消息来源 Topic。
`WHERE`	指定消息筛选条件。

步骤 3：配置转发行为

数据满足规则条件后，规则引擎执行配置的转发行为，将处理后的消息发送至目标服务。
当前规则引擎支持以下主要转发目标：

目标类型	适用场景
另一 Topic	在物联网开发平台内部重新路由消息。
第三方服务	转发至 HTTP 或 HTTPS 服务。
CKafka	写入消息队列，供下游业务系统异步消费。
MySQL	将设备数据写入关系型数据库。
CTSDB	存储设备产生的时序数据。
TDSQL-MySQL	将数据写入分布式数据库。
云开发 CloudBase	将数据转发至云开发相关资源。

服务端规则结构当前同样可以看到 Topic 重发布、第三方服务、CKafka、MySQL 等行为类型。
选择转发目标后，根据目标类型配置对应资源。
例如：

- 转发至另一 Topic，需要指定目标 Topic。
- 转发至第三方服务，需要配置目标服务地址。
- 转发至 CKafka，需要选择实例及目标 Topic。
- 转发至数据库，需要选择实例并配置对应的数据字段映射。

> ⓘ **说明：**
> 规则引擎只负责将规则处理结果发送至目标资源。目标资源本身的创建、容量、权限及业务消费逻辑仍需在对应服务中完成。

步骤 4：配置错误行为（可选）

如果主要转发行为失败，可以配置错误行为，对未成功转发的数据进行补偿处理。
处理关系如下：

- **正常转发 → 成功 → 目标服务**

- **正常转发 → 连续失败 → 错误行为**

当主要行为操作执行失败时，平台会按照1s、3s、10s的间隔依次进行3次重试。若3次重试均失败，且已配置错误行为，则平台再执行1次错误行为转发；如果错误行为仍然失败，该消息将被丢弃。错误行为需要使用与主要行为不同的行为类型，可用于目标资源异常时提供备用处理路径。
错误行为应配置为与主要行为不同的转发类型。例如：

主要转发行为	错误行为示例
第三方 HTTPS 服务	CKafka
MySQL	另一 Topic
CKafka	第三方服务

> **说明：**
> 是否配置错误行为可根据业务对消息可靠性的要求决定。

步骤 5：启用并验证规则

完成数据来源、数据筛选和转发行为配置后，启用规则。
建议按照以下方式验证：

1. 使用设备向规则配置的数据来源 Topic 上报一条消息。
2. 确认消息内容满足规则筛选条件。
3. 检查规则是否正常执行。
4. 前往目标资源确认是否收到对应数据。
5. 如果转发失败，根据规则执行结果和错误信息进行排查。

例如规则条件为：`temperature > 30`
分别上报：

```
{
  "temperature": 25,
  "humidity": 60
}
```

该消息不满足条件，不执行转发。
再上报：

```
{
  "temperature": 32,
  "humidity": 60
```

```
}
```

该消息满足条件，规则提取对应字段并执行配置的转发行为。

规则执行与问题定位

规则消息处理可以按照以下三个阶段进行排查：

现象	优先排查
规则未触发。	数据来源、Topic、规则是否启用。
规则触发但没有筛选结果。	Payload 格式、字段名称、字段路径、筛选条件。
数据筛选成功但转发失败。	目标资源状态、配置参数、网络及访问权限。
显示转发成功但业务侧未获取数据。	目标 Topic、队列、数据库或下游消费逻辑。

例如规则日志出现 `Payload_Not_JSON`，表示规则正在按照 JSON 数据进行处理，但收到的 Payload 不是合法 JSON；出现 `Payload_No_Field`，则通常表示实际 Payload 中不存在规则引用的字段。当前官方规则引擎文档提供了对应运行错误码用于定位这类问题。

常见问题

现象	排查建议
设备已经上报消息，但规则没有触发。	检查规则是否启用，以及设备、Topic 类型和实际消息 Topic 是否与数据来源匹配。
规则没有提取到需要的字段。	检查实际 Payload 是否为 JSON，以及字段名称和字段路径是否正确。
设置条件后规则不再转发。	检查设备实际上报值是否满足条件，并确认字段数据类型与表达式匹配。
非 JSON 消息无法设置筛选字段。	非 JSON 数据不支持字段筛选，请使用 `*` 转发完整 Payload。
规则执行成功但目标没有收到数据。	检查目标 Topic、消息队列、数据库等资源配置以及下游消费逻辑。
第三方 HTTP/HTTPS 服务转发失败。	检查服务地址、网络连通性及目标服务响应状态。
CKafka 没有收到消息。	先确认规则是否成功执行，再检查 CKafka 实例、Topic 及下游消费配置。
数据库写入失败。	检查实例状态、数据库及表配置、字段映射和访问权限。

转发失败后没有进入错误行为。	检查是否已经配置错误行为，以及主要转发行为是否已经完成重试。
下游收到重复消息。	规则转发存在重试机制，建议业务端使用消息唯一标识或业务主键实现幂等处理。

CAM权限管理

最近更新时间：2026-09-11 16:05:32

本文将介绍如何使用腾讯云访问管理 CAM（Cloud Access Management）管理物联网开发平台的访问权限，包括系统策略、自定义策略以及基于设备标签的权限控制。

功能介绍

物联网开发平台接入腾讯云访问管理 CAM，可以通过权限策略控制子用户、用户组或角色能够执行哪些操作，以及可以访问哪些 IoT Explorer 资源。
例如，可以为运维人员授予全部只读权限，为开发人员限制部分管理操作，或将支持资源级授权的操作限定在指定产品、设备范围内。对于需要按照地域、团队、客户等维度划分设备的场景，还可以在自定义策略中增加标签条件。

> **注意：**
> 物联网开发平台在 CAM 中的服务简称为 `iotexplorer` 。当前产品整体支持资源级授权和按标签授权，但具体 Action 的授权粒度仍可能是操作级或资源级，配置权限时需要以对应接口的 CAM 定义为准。

权限方案选择

根据需要控制的权限范围，可以选择以下方式：

权限需求	推荐方案	主要控制维度

管理全部 IoT Explorer 资源。	`QcloudIotExplorerFullAccess`	全部管理权限。
只查看平台资源。	`QcloudIotExplorerReadOnlyAccess`	全部只读权限。
只允许部分操作。	自定义策略。	`Action`
限定指定项目、产品或设备。	自定义策略。	`Resource`，仅适用于支持资源级授权的 Action。
禁止删除等敏感操作。	自定义策略。	`Effect = deny`
根据设备标签限制访问范围。	自定义策略 + 标签条件。	`Condition`

其中，系统策略和自定义策略是两类主要授权方式；标签权限本质上是在自定义策略中通过 `Condition` 增加资源标签条件，并不是独立于 CAM 策略之外的另一套权限体系。

权限控制模型

CAM 策略主要通过 `Action`、`Resource`、`Effect` 和可选的 `Condition` 描述权限，可以理解为：**允许或拒绝某个身份，在指定条件下，对哪些资源执行哪些操作。**

策略元素	说明
`Action`	允许或拒绝执行的操作，例如查询设备、删除设备等。
`Resource`	Action 可以作用的资源范围。
`Effect`	`allow` 表示允许，`deny` 表示拒绝。
`Condition`	可选，用于增加标签等附加授权条件。

策略创建完成后，需要关联至对应的子用户、用户组或角色才能生效。现有标签授权文档支持将策略关联至用户、用户组或角色。

操作级与资源级授权

物联网开发平台虽然整体属于资源级授权产品，但并不表示每个 Action 都可以限定到具体产品或设备。当前 CAM 将接口划分为：

授权粒度	说明

操作级	只能控制是否允许调用该 Action，不支持进一步限定具体资源。
资源级	可以通过 `Resource` 将权限限定到接口支持的具体资源。

例如，当前 CAM 接口清单中：

Action	授权粒度	Resource
`DeleteDevice`	操作级	`*`
`DescribeDevice`	操作级	`*`
`DescribeDevices`	资源级	支持设备资源六段式。

对于操作级接口，如果策略将 Action 限定到具体 ProductID、DeviceName 等资源，CAM 会判断该接口不在授权范围，最终表现为无权限，而不是仅对指定资源授权失败。具体 Action 的授权粒度及资源六段式可能随接口调整，应以最新的**物联网开发平台支持 CAM 的业务接口**为准。

前提条件

配置权限前，请确认：

- 已创建需要授权的子用户、用户组或角色。
- 已明确对应账号实际需要执行的 IoT Explorer 操作。
- 如需限制项目、产品或设备范围，已获取对应的 ProjectID、ProductID、DeviceName 等资源标识。
- 如需按标签控制设备范围，已为相关设备配置对应标签。

控制台操作步骤详见 创建子账号 。

使用注意事项

- 遵循最小权限原则，优先授予账号完成实际工作所需的权限，避免普通账号长期持有 FullAccess。
- IoT Explorer 整体支持资源级授权，不代表所有 Action 都支持限制到具体产品或设备。
- 不同 Action 的 Resource 六段式可能不同，应以最新 CAM 接口清单为准。
- CAM 一般遵循显式 Deny 优先于 Allow，但查询资源列表等场景可能存在特殊鉴权行为。
- 使用标签权限时，应关注未打标签资源的实际可见范围，严格隔离场景需要单独验证。
- 编程访问使用的 SecretId、SecretKey 等凭证应妥善保管，不应写入公开代码仓库或客户端程序。

使用系统策略

当账号需要访问全部 IoT Explorer 资源时，可以直接使用平台提供的系统策略。

系统策略	权限范围	适用场景

`QcloudIotExplorerFullAccess`	IoT Explorer 全部管理权限。	平台管理员等需要完整管理能力的账号。
`QcloudIotExplorerReadOnlyAccess`	IoT Explorer 全部只读权限。	运维查看、监控、审计等场景。

`QcloudIotExplorerReadOnlyAccess` 可用于查看主账号下的 IoT Explorer 资源，但不能执行修改操作；授予 `QcloudIotExplorerFullAccess` 后，子账号可以查看、修改和删除主账号可访问的产品、设备等数据，因此应谨慎授予。
如果系统策略权限范围过大，或无法满足指定操作、指定资源的授权需求，应使用自定义策略。

使用自定义策略

自定义策略适用于需要进一步控制操作范围或资源范围的场景。创建策略时，服务选择 `iotexplorer`，再根据实际需求配置 Action、Resource、Effect 和可选 Condition。

配置项	说明
服务	`iotexplorer`
Action	需要允许或拒绝的 IoT Explorer 操作。
Resource	Action 作用的资源范围。
Effect	`allow` 或 `deny`。
Condition	可选，用于增加标签等限制条件。

现有 如何给子账号配置 CAM 权限 支持通过策略生成器选择 `iotexplorer` 服务、指定 API，并选择全部资源或特定资源，也支持直接使用策略语法创建自定义策略。

按操作限制权限

如果只需要开放部分能力，可以只授权业务需要的 Action；如果账号已经拥有较大范围的 Allow 权限，但需要禁止删除等敏感操作，可以增加 Deny 策略。
例如：

```
{
  "version": "2.0",
  "statement": [
    {
      "effect": "deny",
```

```
        "action": [
          "iotexplorer:DeleteStudioProduct",
          "iotexplorer:DeleteDevice"
        ],
        "resource": "*"
      }
    ]
}
```

CAM 一般遵循显式 Deny 优先于 Allow 的策略评估逻辑：若请求匹配 Deny，则拒绝访问；未匹配 Deny 且匹配 Allow 时才允许访问。
需要注意，部分**查询资源列表**等场景存在 Deny 不按通常逻辑生效的特殊情况，因此不建议仅依靠 Deny 实现资源可见性隔离。

按资源限制权限

对于支持资源级授权的 Action，可以进一步通过 `Resource` 限定具体项目、产品或设备。

资源层级	常用资源标识
项目	ProjectID
产品	ProductID
设备	DeviceName

物联网开发平台不同 Action 对应的资源六段式可能不同。例如部分资源包含 Region、UIN、ProjectID、ProductID 和 DeviceName，而部分接口只支持到产品或项目层级。
因此，不建议统一套用固定的 Resource 模板，应根据目标 Action 在 CAM 接口列表中对应的**授权粒度和资源六段式**配置。
产品级权限的策略生成和关联方式可参见 如何给子账号配置 CAM 权限 。

按设备标签限制访问范围

如果需要按照地域、团队、客户等维度进一步划分设备范围，可以为设备配置标签，并在自定义策略中通过 `Condition` 设置资源标签条件。
例如，设备标签为：

配置项	示例
标签键	`location`
标签值	`shenzhen`

Condition Key	`qcs:resource_tag`
条件值	`location&shenzhen`

现有 IoT Explorer 标签权限示例使用：

```
{
  "version": "2.0",
  "statement": [
    {
      "effect": "allow",
      "action": [
        "iotexplorer:GetDeviceList",
        "tag:DescribeResourceTags"
      ],
      "resource": "*",
      "condition": {
        "for_any_value:string_equal": {
          "qcs:resource_tag": [
            "location&shenzhen"
          ]
        }
      }
    }
  ]
}
```

其中标签条件采用 `标签键&标签值` 的格式，例如 `location&shenzhen` 。

> ⓘ **注意：**
>
> - **当前 通过标签设置用户权限 的官方示例中，配置 `location=shenzhen` 后，子用户可以看到匹配该标签的设备，同时也可以看到未设置标签的设备。因此，标签条件不能直接等同于严格的客户、租户或生产环境隔离。对于严格隔离场景，应统一规划标签并实际验证目标 Action 和无标签资源的访问结果。**
> - **另外，业务含义相近的接口可能对应不同 Action，不同 Action 的授权粒度也可能不同。配置标签或资源策略时，应以实际功能触发的 Action 及 CAM 当前接口定义为准，不建议根据接口名称自行替换。**

验证与排查

权限配置完成后，建议使用对应子账号实际验证资源可见范围和操作权限。

现象	优先排查
子账号看不到产品或设备。	查询相关 Action 是否已授权，策略是否已正确关联。
接口返回 `UnauthorizedOperation` 。	Action、授权粒度、Resource、Condition 是否正确。
已限定某个产品，但部分功能仍提示无权限。	对应 Action 是否只支持操作级授权。
已配置 Allow 但仍被拒绝。	是否存在匹配的 Deny 或其他权限策略。
标签设备不可见。	标签 Condition、标签相关权限和实际 Action。
未设置标签的设备仍可见。	当前标签授权示例本身存在该行为，应重新评估隔离策略。

排查时应优先确认**实际调用的 Action**，再检查该 Action 的授权粒度和资源格式。
对于操作级接口，不应尝试通过具体产品或设备 Resource 对其进行资源隔离。