Alibaba Cloud
IoT Platform

Developer Guide (Devices)

Issue: 20181113

Legal disclaimer

Generic conventions

Table -1: Style conventions

Style	Description	Example
⛔	This warning information indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	⛔ **Danger:** Resetting will result in the loss of user configuration data.
⚠	This warning information indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	⚠ **Warning:** Restarting will cause business interruption. About 10 minutes are required to restore business.
📋	This indicates warning information, supplementary instructions, and other content that the user must understand.	📋 **Note:** Take the necessary precautions to save exported data containing sensitive information.
	This indicates supplemental instructio ns, best practices, tips, and other content that is good to know for the user.	📋 **Note:** You can use **Ctrl** + **A** to select all files.
>	Multi-level menu cascade.	**Settings** > **Network** > **Set network type**
Bold	It is used for buttons, menus, page names, and other UI elements.	Click **OK**.
`Courier font`	It is used for commands.	Run the `cd /d C:/windows` command to enter the Windows system folder.
Italics	It is used for parameters and variables.	`bae log list --instanceid` *`Instance_ID`*
[] or [a|b]	It indicates that it is a optional value, and only one item can be selected.	`ipconfig` *`[-all|-t]`*
{} or {a|b}	It indicates that it is a required value, and only one item can be selected.	`swich` *`{stand | slave}`*

Contents

1 Download device SDKs

IoT Platform provides multiple device SDKs to help you develop your devices and quickly connect them to the Cloud. As an alternative to SDKs, you can also use *Alink Protocol* for development.

Prerequisites

Before developing devices, finish all console configurations, and obtain necessary informations such as the device details and topic information. For more information, see the User Guide.

Device SDKs

Select a device SDK according to the protocol and the features that you require. We recommend that you use C SDK as it provides more features.

Note:

If you have specific development requirements that cannot be met by the current SDKs, you can develop according to the *Alink protocol*.

	C SDK	Java SDK	Android SDK	iOS SDK	HTTP/2 SDK	General protocol
MQTT	√	√	√	√		
CoAP	√					
HTTP/HTTPS	√					
HTTP/2					√	
Other protocols						√
Device certification: unique-certificate-per-device authentication	√	√	√	√	√	√
Device certification: unique-certificate-per-product authentication	√		√			
OTA development	√					
Connecting sub -devices to IoT Platform	√					
Device shadow	√	√	√			

	C SDK	Java SDK	Android SDK	iOS SDK	HTTP/2 SDK	General protocol
Device development based on TSL	√		√			
Remote configuration	√					

Supported platforms

Click *here* to view and query the platforms supported by Alibaba Cloud IoT Platform.

If the platform you want to use is not supported by IoT Platform, please open an issue on the *Github* page.

Download SDKs

- C SDK

Version number	Release date	Development environment	Download link	Updates
V2.2.1	2018/09/03	GNU make on 64-bit Linux	*RELEASED_V2.2.1*	• Added supports for connecting devices to WiFi and using open-source applications to locally control devices. • Added supports for countdown routine before devices go offline. • Added supports for OTA using iTls to download firmware files.
V2.1.0	2018/03/31	GNU make on 64-bit Linux	*RELEASED_V2_10_20180331.zip*	• Added support for CMake: You can use QT or VS2017 on Linux or Windows to open a project and compile software in CMake compiling method. • Added support for TSL model definition on IoT Platform: You can set `FEATURE_CMP_ENABLED = y` and`FEATURE_DM_ENABLED = y` to define TSL models to provide API operations for properties, events, and services. • Added support for unique-certificate-per-product: You can set `FEATURE_SUPPORT_PRODUCT_SECRET = y` to enable

Version number	Release date	Development environment	Download link	Updates
				unique-certificate-per-product authentication and streamline the production queuing process. • Added support for iTLS: You can set `FEATURE_MQTT_DIRECT_NOTLS = y` and `FEATURE_MQTT_DIRECT_NOITLS = n` to enable ID² encryption. You can use iTLS to establish data connections to enhance security and reduce memory consumption. • Added support for remote configuration: You can set `FEATURE_SERVICE_OTA_ENABLED = y` and `FEATURE_SERVICE_COTA_ENABLED = y` to enable the cloud to push configuration information to devices. • Optimized sub-device management of gateways : Added some features.

- Java SDK

Supported protocol	Update history	Download link
MQTT	2017-05-27: Added support for device authentication in the China (Shanghai) region. Added the device shadow demo on the Java client.	*iotx-sdk-mqtt-java*: The Java version that supports MQTT is only a demo of open-source library implementation. It is used only for reference.

Instructions: See *Java SDK*.

- iOS SDK

Download link:

- *https://github.com/CocoaPods/Specs.git*
- *https://github.com/aliyun/aliyun-specs.git*

Instructions:*iOS SDK*

- HTTP/2 SDK

Download link: *iot-http2-sdk-demo*.

- General protocol

Instructions: See *General protocol*.

- Other open-source libraries

  Download link: *https://github.com/mqtt/mqtt.github.io/wiki/libraries*

2 C-SDK

2.1 Authenticate devices

To secure devices, IoT Platform provides certificates for devices, including product certificates (ProductKey and ProductSecret) and device certificates (DeviceName and DeviceSecret). A device certificate is a unique identifier used to authenticate a device. Before a device connects to IoT Hub through a protocol, the device reports the product certificate or the device certificate, depending on the authentication method. The device can connect to IoT Platform only when it passes authentication. IoT Platform supports various authentication methods to meet the requirements of different environments.

IoT Platform supports the following authentication methods:

- Unique-certificate-per-device authentication: Each device has been installed with its own unique device certificate.
- Unique-certificate-per-product authentication: All devices under a product have been installed with the same product certificate.
- Sub-device authentication: This method can be applied to sub-devices that connect to IoT Platform through the gateway.

These methods have their own advantages in terms of accessibility and security. You can choose one according to the security requirements of the device and the actual production conditions. The following table shows the comparison among these methods.

Table 2-1: Comparison of authentication methods

Items	Unique-certificate-per-device authentication	Unique-certificate-per-product authentication	Sub-device authentication
Information written into the device	ProductKey, DeviceName, and DeviceSecret	ProductKey and ProductSecret	ProductKey
Whether to enable authentication in IoT Platform	No. Enabled by default.	Yes. You must enable dynamic register.	Yes. You must enable dynamic register.
DeviceName pre-registration	Yes. You need to make sure that the	Yes. You need to make sure that the	Yes.

Items	Unique-certificate-per-device authentication	Unique-certificate-per-product authentication	Sub-device authentication
	specified DeviceName is unique under a product.	specified DeviceName is unique under a product.	
Certificate installation requirement	Install a unique device certificate on every device. The safety of every device certificate must be guaranteed.	Install the same product certificate on all devices under a product. Make sure that the product certificate is safely kept.	Install the same product certificate into all sub-devices . The security of the gateway must be guaranteed.
Security	High	Medium	Medium
Upper limit for registrations	Yes. A product can have a maximum of 500,000 devices.	Yes. A product can have a maximum of 500,000 devices.	Yes. A maximum of 200 sub-devices can be registered with one gateway.
Other external reliance	None	None	Security of the gateway.

2.1.1 Authenticate devices

To secure devices, IoT Platform provides certificates for devices, including product certificates (ProductKey and ProductSecret) and device certificates (DeviceName and DeviceSecret). A device certificate is a unique identifier used to authenticate a device. Before a device connects to IoT Hub through a protocol, the device reports the product certificate or the device certificate, depending on the authentication method. The device can connect to IoT Platform only when it passes authentication. IoT Platform supports various authentication methods to meet the requirements of different environments.

IoT Platform supports the following authentication methods:

- Unique-certificate-per-device authentication: Each device has been installed with its own unique device certificate.
- Unique-certificate-per-product authentication: All devices under a product have been installed with the same product certificate.
- Sub-device authentication: This method can be applied to sub-devices that connect to IoT Platform through the gateway.

These methods have their own advantages in terms of accessibility and security. You can choose one according to the security requirements of the device and the actual production conditions. The following table shows the comparison among these methods.

Table 2-2: Comparison of authentication methods

Item	**Unique-certificate-per-device authentication**	**Unique-certificate-per-product authentication**	**Sub-device authentication**
Information written into the device	ProductKey, DeviceName, and DeviceSecret	ProductKey and ProductSecret	ProductKey
Whether to enable authentication in IoT Platform	No. Enabled by default .	Yes. You must enable dynamic register.	Yes. You must enable dynamic register.
DeviceName pre-registration	Yes. You need to make sure that the specified DeviceName is unique under a product.	Yes. You need to make sure that the specified DeviceName is unique under a product.	Yes.
Certificate installation requirement	Install a unique device certificate on every device. The safety of every device certificate must be guaranteed.	Install the same product certificate on all devices under a product. Make sure that the product certificate is safely kept.	Install the same product certificate into all sub-devices . The security of the gateway must be guaranteed.
Security	High	Medium	Medium
Upper limit for registrations	Yes. A product can have a maximum of 500,000 devices.	Yes. A product can have a maximum of 500,000 devices.	Yes. A maximum of 1500 sub-devices can be registered with one gateway.
Other external reliance	None	None	Security of the gateway.

2.1.2 Unique-certificate-per-device authentication

This topic describes unique-certificate-per-device authentication and how to use this authentication method.

What is unique-certificate-per-device authentication

IoT Platform uses unique-certificate-per-device authentication by default. The device needs to be installed with a unique device certificate in advance. When connecting to IoT Platform, the device must use ProductKey, DeviceName, and DeviceSecret for authentication. When the device passes the authentication, IoT Platform activates the device. The device and IoT Platform then can transmit data.

Note:

Unique-certificate-per-device authentication is a secure authentication method. We recommend that you use this authentication method.

Workflow of unique-certificate-per-device authentication

Figure 2-1: Unique-certificate-per-device authentication

You can use the following process to authenticate a device with unique-certificate-per-device authentication:

1. Create a product and a device, as shown in "Create products and devices." The device obtains authentication information including ProductKey, DeviceName, and DeviceSecret.
2. Install the ProductKey, DeviceName, and DeviceSecret on the device.
3. Power on and connect the device to IoT Platform. The device will initiate an authentication request to IoT Platform.
4. IoT Platform authenticates the device. If the device passes authentication, IoT Platform establishes a connection to the device, and the device begins to publish or subscribe to topics for data upload and download.

Procedure

1. Use an Alibaba Cloud account to log on to the *IoT Platform console*.
2. Create a product and add a device to the product, as shown in *#unique_12*.
3. From the left-side navigation pane, select **Devices**. Select the device, and click **View** to obtain the ProductKey, DeviceName, and DeviceSecret, as shown in the following figure:

Figure 2-2: ProductKey, DeviceName, and DeviceSecret

4. Download the device SDK, select a connection protocol, and configure the device SDK.
5. Add the ProductKey, DeviceName, and DeviceSecret to the device SDK.
6. Perform the following configurations as needed: OTA settings, sub-device connection, device TSL model development, and device shadow management.
7. Install the device SDK that already includes the ProductKey, DeviceName, and DeviceSecret on the device.

2.1.3 Unique-certificate-per-product authentication

This topic describes the unique-certificate-per-product authentication and how to use this authentication method.

What is unique-certificate-per-product authentication

Unique-certificate-per-product authentication requires that devices under a product are installed with the same firmware. A product certificate (ProductKey and ProductSecret) is installed on the firmware. Using this authentication method can simplify the installation process. After the firmware has been installed, the device dynamically obtains a DeviceSecret from IoT Platform when it is activated.

Note:

- Only the device C SDK supports unique-certificate-per-product authentication.
- Make sure that the device supports unique-certificate-per-product authentication.
- Unique-certificate-per-product authentication has risks of product certificate leakage because all devices under a product are installed with the same firmware. To resolve this issue, go to **Product Information** page. Disable **`Dynamic Register`** to reject authentication requests from any new devices.

Workflow of unique-certificate-per-product authentication

Figure 2-3: Unique-certificate-per-product authentication

You can use the following process to authenticate a device with unique-certificate-per-product authentication:

1. Create a product, as shown in "Create products and devices," and obtain the product certificate .
2. On the Product Information page, enable **`Dynamic Register`**. The system will send an SMS to verify your access to IoT Platform.

   Note:
   During verification, the system will reject dynamic activation requests from any new devices if dynamic registration is disabled. Activated devices will not be affected.
3. On the Devices page, add the device to the product. You can use the MAC address and ID information such as IMEI or SN as DeviceName to pre-register the device with the platform. The device is in **`Inactive`** status.

   Note:
   The system verifies the DeviceName to activate a device. We recommend that you use the device ID information that can be read directly from the device as the DeviceName.
4. Install the same product certificate on all devices under the product, and set `FEATURE_SUPPORT_PRODUCT_SECRET = y` in the device C SDK to enable unique-certificate-per-product authentication.
5. Power on the device and connect the device to the network. The device sends an authentication request to IoT Platform to perform unique-certificate-per-product authentication. If the device passes authentication, IoT Platform dynamically assigns the corresponding DeviceSecret to the device. The device obtains the ProductKey, DeviceName, and DeviceSecret required for connecting to IoT Platform.

   Note:
   In this method, IoT Platform dynamically assigns a DeviceSecret to the device only for the first activation after the device has been added to IoT Platform. To reactivate a device, delete the device from IoT Platform and add it again.
6. The device uses the ProductKey, DeviceName, and DeviceSecret to connect to IoT Platform, and publish or subscribe to topics for data upload and download.

Procedure

1. Use an Alibaba account to log on to *IoT Platform console*.
2. Create a product, as shown in *Create products*.

3. Obtain the product certificate that is generated by IoT Platform, including ProductKey and ProductSecret.
   a. Select the product, and click **View** to go to the **Product Information** page.
   b. Obtain the product certificate, as shown in the following figure:

      Figure 2-4: Product Information

   c. Enable **`Dynamic Registration`**, and enter the SMS verification code to verify your access to IoT Platform.
4. On the **Devices** page, create a device. Use device information such as the MAC address, IMEI, or SN as the DeviceName.
5. Select a connection protocol, download the device C SDK, and configure the C SDK.
6. Add the obtained product certificate (ProductKey and ProductSecret) to the device SDK.
7. In the device SDK, set `FEATURE_SUPPORT_PRODUCT_SECRET = y` to enable unique-certificate-per-product authentication.
8. For more information, see `IOT_CMP_Init` in `/src/cmp/iotx_cmp_api.c`. A code example is as follows:

```
#ifdef SUPPORT_PRODUCT_SECRET
        /* Unique certificate per product */
        if (IOTX_CMP_DEVICE_SECRET_PRODUCT == pparam->secret_type
 && 0 >= HAL_GetDeviceSecret(device_secret)) {
            HAL_GetProductSecret(product_secret);
            if (strlen(product_secret) == 0) {
                CMP_ERR(cmp_log_error_secret_1);
                return FAIL_RETURN;
            }
            /* auth */
            if (FAIL_RETURN == iotx_cmp_auth(product_key, device_nam
e, device_id)) {
                CMP_ERR(cmp_log_error_auth);
                return FAIL_RETURN;
            }
        }
    #endif /**< SUPPORT_PRODUCT_SECRET*/
```

9. Perform the following configurations as needed: OTA settings, sub-device connection, device TSL model development, and device shadow management.
10. Install the device SDK that already includes the product certificate on the device.

2.2 Protocols for connecting devices

2.2.1 Establish MQTT over TCP connections

This topic describes the TCP-based MQTT connection and provides two modes of device authentication.

- MQTT clients directly connect to the specified domain names without providing additional device credential information. We recommend that you use this authentication mode for devices with limited resources.
- After HTTPS authentication, connect to the special value-added services of MQTT, such as distributing communication traffic from devices among clusters.

Note:

When you configure the MQTT CONNECT packet:

- The keepalive value in the Connect command must be larger than 30 seconds, otherwise, the connection will be denied. We recommend that you set the value in the range of 60 to 300 seconds.
- If multiple devices are connected using the same set of ProductKey, DeviceName, and DeviceSecret, some devices will be brought offline.
- The default setting of the MQTT protocol is that open-source SDKs are automatically connected. You can view device behaviors using Log Service.

For more information, see the example of *`\sample\mqtt\mqtt-example.c`* in device SDK code package that you downloaded.

Direct connection to the MQTT client domain

- Without an existing demo

  If you use the open-source MQTT package for access, see the following procedure:

  1. We recommend that you use TLS to encrypt, because it provides higher security. If you are using TLS to encrypt, you need to *download a root certificate*.
  2. If you want to access the server using an MQTT client, see *Open-source MQTT client references*. For more information about the MQTT protocol, see *http://mqtt.org*.

     Note:

     Alibaba Cloud does not provide technical support if you are using third-party code.

3. Instructions for MQTT connection

    - Connection domain: `${YourProductKey}.iot-as-mqtt. ${YourRegionId}.aliyuncs.com:1883`

      Replace ${YourProductKey} with your product ID. Replace ${YourRegionId} with your device region ID. For the expressions of region IDs, see *Regions and zones*.
    - The MQTT Connect packets include the following parameters:

      ```
      mqttClientId: clientId+"|securemode=3,signmethod=hmacsha1,
      timestamp=132323232|"
      mqttUsername: deviceName+"&"+productKey
      mqttPassword: sign_hmac(deviceSecret,content)
      ```

      Sort the following parameters in alphabetical order and then encrypt the parameters based on the specified sign method.

      The value of content is the parameters sent to the server (productKey, deviceName, timestamp, and clientId). Sort these parameters in alphabetical order and then splice the parameters and parameter values.

        - clientId: Indicates the client ID. We recommend that you use the MAC address or the serial number of the device as the client ID. The length of the client ID must be within 64 characters.
        - timestamp: The current time in milliseconds.This parameter is optional.
        - mqttClientId: The expanded parameters are in `||`.
        - signmethod: Specifies a signature algorithm.
        - securemode: The current security mode. Values include: 2 (TLS direct connection) and 3 (TCP direct connection).

    Example: If `clientId = 12345, deviceName = device, productKey = pk, timestamp = 789, signmethod=hmacsha1, deviceSecret=secret`, submit the MQTT parameters over TCP:

    ```
    mqttclientId=12345|securemode=3,signmethod=hmacsha1,timestamp=789|
    username=device&pk
    ```

```
password=hmacsha1("secret","clientId12345deviceNamedevicep
roductKeypktimestamp789").toHexString(); // The last parameter is
a binary-to-hexadecimal string. Its case is insensitive.
```

The result is:

```
FAFD82A3D602B37FB0FA8B7892F24A477F851A14
```

Note:

The three parameters are: mqttClientId, mqttUsername, and mqttPassword of the MQTT Connect logon packets.

- With an existing demo

1. An MQTT connection over TCP is supported only when you directly connect to a domain. Set the values of FEATURE_MQTT_DIRECT, FEATURE_MQTT_DIRECT, and FEATURE_MQTT_DIRECT_NOTLS in make.settings to **y**.

```
FEATURE_MQTT_DIRECT = y
FEATURE_MQTT_DIRECT = y
FEATURE_MQTT_DIRECT_NOTLS = y
```

2. In the SDK, call IOT_MQTT_Construct to connect to the cloud.

```
pclient = IOT_MQTT_Construct(&mqtt_params);
if (NULL == pclient) {
EXAMPLE_TRACE("MQTT construct failed");
rc = -1;
goto do_exit;
}
```

The function declaration is as follows:

```
/**
* @brief Construct the MQTT client
* This function initialize the data structures, establish MQTT
connection.
*
* @param [in] pInitParams: specify the MQTT client parameter.
*
* @retval NULL : Construct failed.
* @retval NOT_NULL : The handle of MQTT client.
* @see None.
*/
void *IOT_MQTT_Construct(iotx_mqtt_param_t *pInitParams);
```

Connect after the HTTPS authentication

1. Authenticate devices

Use HTTPS for device authentication. The authentication URL is `https://iot-auth. ${YourRegionId}.aliyuncs.com/auth/devicename`. Replace ${YourRegionId} with your device region ID. For the expressions of region IDs, see *Regions and zones*.

- The authentication request parameters are as follows:

Parameter	Required	Description
productKey	Yes	ProductKey. You can view the ProductKey in the IoT Platform console.
deviceName	Yes	DeviceName. You can view the DeviceName in the IoT Platform console.
sign	Yes	Signature. The authentication format is HMAC-MD5 for deviceSecret and content. In the content, all parameters submitted to the server (except for version, sign, resources, and signmethod) are sorted alphabetically. Then, the parameter values are spliced with the parameter names without using any splice character.
signmethod	No	The algorithm type, such as HMAC-MD5 or HMAC-SHA1. The default type is HMAC-MD5.
clientId	Yes	The client ID. Its length must be within 64 characters.
timestamp	No	The timestamp. Serial port validation is not required.
resources	No	The resource description that you want to obtain, such as MQTT. Multiple resource names are separated by commas.

- Response parameters:

Parameter	Required	Description
iotId	Yes	The connection tag that is issued by the server and used to specify a value for username in the MQTT Connect packets.
iotToken	Yes	The token is valid for seven days. It is used to specify a value for password in the MQTT Connect packets.

Parameter	Required	Description
resources	No	The resource information. The expanded information includes the MQTT server address and CA certificate information.

- Request example using x-www-form-urlencoded:

```
POST /auth/devicename HTTP/1.1
Host: iot-auth.cn-shanghai.aliyuncs.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 123
productKey=123&sign=123&timestamp=123&version=default&clientId=123
&resouces=mqtt&deviceName=test
sign = hmac_md5(deviceSecret, clientId123deviceNametestprodu
ctKey123timestamp123)
```

- Response example:

```
HTTP/1.1 200 OK
Server: Tengine
Date: Wed, 29 Mar 2017 13:08:36 GMT
Content-Type: application/json;charset=utf-8
Connection: close
{
     "code" : 200,
     "data" : {
        "iotId" : "42Ze0mk3556498a1AlTP",
        "iotToken" : "0d7fdeb9dc1f4344a2cc0d45edcb0bcb",
        "resources" : {
            "mqtt" : {
               "host" : "xxx.iot-as-mqtt.cn-shanghai.aliyuncs.com
",
               "port" : 1883
        }
      }
      },
      "message" : "success"
}
```

2. Connect to MQTT

   a. Download the *root.crt* file of IoT Platform. We recommend that you use TLS protocol version 1.2.

   b. Connect the device to the Alibaba Cloud MQTT server address and authenticate the returned MQTT address and port.

   c. TLS is used to establish a connection. The client authenticates the server using CA certificates. The server authenticates the client using the "username=iotId, password=iotToken, clientId=custom device identifier (use either the MAC address or the device serial number to specify clientId)" in the MQTT connect packets.

If the iotId or iotToken is invalid, then MQTT Connect fails. The connect ack flag you receive is 3.

The error code descriptions are as follows:

- 401: request auth error. This error code is usually returned when the signature is invalid.
- 460: param error. Parameter error.
- 500: unknown error. Unknown error.
- 5001: meta device not found. The specified device does not exist.
- 6200: auth type mismatch. An unauthorized authentication type error.

If you use an existing demo, set the value of FEATURE_MQTT_DIRECT in make.settings to "n." Then call the IOT_MQTT_Construct function to reconnect after the authentication.

```
FEATURE_MQTT_DIRECT = n
```

The HTTPS authentication process is documented in the iotx_guider_authenticate of \src\system\iotkit-system\src\guider.c.

SDK APIs

- IOT_MQTT_Construct: Call IOT_MQTT_Construct to establish an MQTT connection to the cloud.

  The mqtt-example program automatically quits every time the program sends a message. The following are methods that you can use to keep the device online:

  - When the mqtt-example program is executed, use the `./mqtt-example loop` command to make the device always remain online.
  - Modify the demo code. The demo code of the mqtt-example program contains IOT_MQTT_Destroy. The device goes offline when IOT_MQTT_Destroy is called. To keep the device online, remove IOT_MQTT_Unregister and IOT_MQTT_Destroy. Use while to maintain a persistent connection.

  The code is as follows:

```
while(1)
{
IOT_MQTT_Yield(pclient, 200);
HAL_SleepMs(100);
}
```

The response parameter declaration is as follows:

```
/**
* @brief Construct the MQTT client
* This function initialize the data structures, establish MQTT
connection.
*
* @param [in] pInitParams: specify the MQTT client parameter.
*
* @retval NULL : Construct failed.
* @retval NOT_NULL : The handle of MQTT client.
* @see None.
*/
void *IOT_MQTT_Construct(iotx_mqtt_param_t *pInitParams);
```

- IOT_MQTT_Subscribe: Call IOT_MQTT_Subscribe to subscribe to a topic in the cloud.

  To ensure that callback topic_handle_func can be successfully delivered, keep memory for topic_filter.

  The code is as follows:

```
/* Subscribe the specific topic */
rc = IOT_MQTT_Subscribe(pclient, TOPIC_DATA, IOTX_MQTT_QOS1,
_demo_message_arrive, NULL);
if (rc < 0) {
IOT_MQTT_Destroy(&pclient);
EXAMPLE_TRACE("IOT_MQTT_Subscribe() failed, rc = %d", rc);
rc = -1;
goto do_exit;
}
```

The response parameter declaration is as follows:

```
/**
* @brief Subscribe MQTT topic.
*
* @param [in] handle: specify the MQTT client.
* @param [in] topic_filter: specify the topic filter.
* @param [in] qos: specify the MQTT Requested QoS.
* @param [in] topic_handle_func: specify the topic handle callback-
function.
* @param [in] pcontext: specify context. When call 'topic_hand
le_func', it will be passed back.
*
* @retval -1 : Subscribe failed.
* @retval >=0 : Subscribe successful.
The value is a unique ID of this request.
The ID will be passed back when callback 'iotx_mqtt_param_t:
handle_event'.
* @see None.
*/
int IOT_MQTT_Subscribe(void *handle,
const char *topic_filter,
iotx_mqtt_qos_t qos,
iotx_mqtt_event_handle_func_fpt topic_handle_func,
void *pcontext);
```

- IOT_MQTT_Publish: Call IOT_MQTT_Publish to publish messages to the cloud.

  The code is as follows:

  ```
  /* Initialize topic information */
  memset(&topic_msg, 0x0, sizeof(iotx_mqtt_topic_info_t));
  strcpy(msg_pub, "message: hello! start!") ;
  topic_msg.qos = IOTX_MQTT_QOS1;
  topic_msg.retain = 0;
  topic_msg.dup = 0;
  topic_msg.payload = (void *)msg_pub;
  topic_msg.payload_len = strlen(msg_pub);
  rc = IOT_MQTT_Publish(pclient, TOPIC_DATA, &topic_msg);
  EXAMPLE_TRACE("rc = IOT_MQTT_Publish() = %d", rc);
  ```

  The response parameter declaration is as follows:

  ```
  /**
  * @brief Publish message to specific topic.
  *
  * @param [in] handle: specify the MQTT client.
  * @param [in] topic_name: specify the topic name.
  * @param [in] topic_msg: specify the topic message.
  *
  * @retval -1 : Publish failed.
  * @retval 0 : Publish successful, where QoS is 0.
  * @retval >0 : Publish successful, where QoS is >= 0.
  The value is a unique ID of this request.
  The ID will be passed back when callback 'iotx_mqtt_param_t:
  handle_event'.
  * @see None.
  */
  int IOT_MQTT_Publish(void *handle, const char *topic_name,
  iotx_mqtt_topic_info_pt topic_msg);
  ```

- IOT_MQTT_Unsubscribe: Call IOT_MQTT_Unsubscribe to cancel the subscription in the cloud.

  The code is as follows:

  ```
  IOT_MQTT_Unsubscribe(pclient, TOPIC_DATA);
  ```

  The response parameter declaration is as follows:

  ```
  /**
  * @brief Unsubscribe MQTT topic.
  *
  * @param [in] handle: specify the MQTT client.
  * @param [in] topic_filter: specify the topic filter.
  *
  * @retval -1 : Unsubscribe failed.
  * @retval >=0 : Unsubscribe successful.
  The value is a unique ID of this request.
  The ID will be passed back when callback 'iotx_mqtt_param_t:
  handle_event'.
  * @see None.
  */
  ```

```
int IOT_MQTT_Unsubscribe(void *handle, const char *topic_filter);
```

- IOT_MQTT_Yield: Call IOT_MQTT_Yield to receive data.

  Call this operation wherever data needs to be received. Start a separate thread to perform this operation if the system resources are sufficient.

  The code is as follows:

```
/* handle the MQTT packet received from TCP or SSL connection */
IOT_MQTT_Yield(pclient, 200);
```

  The response parameter declaration is as follows:

```
/**
* @brief Handle MQTT packet from remote server and process timeout
request
* which include the MQTT subscribe, unsubscribe, publish(QOS >= 1),
reconnect, etc..
*
* @param [in] handle: specify the MQTT client.
* @param [in] timeout_ms: specify the timeout in millisecond in this
 loop.
*
* @return status.
* @see None.
*/
int IOT_MQTT_Yield(void *handle, int timeout_ms);
```

- IOT_MQTT_Destroy: Call IOT_MQTT_Destroy to terminate an MQTT connection and release the memory.

  The code is as follows:

```
IOT_MQTT_Destroy(&pclient);
```

  The response parameter declaration is as follows:

```
/**
* @brief Deconstruct the MQTT client
* This function disconnect MQTT connection and release the related
resource.
*
* @param [in] phandle: pointer of handle, specify the MQTT client.
*
* @retval 0 : Deconstruct success.
* @retval -1 : Deconstruct failed.
* @see None.
*/
int IOT_MQTT_Destroy(void **phandle);
```

- IOT_MQTT_CheckStateNormal: Call IOT_MQTT_CheckStateNormal to view the current connection status.

You can use this operation to query the status of an MQTT connection. However, this operation cannot detect a device disconnection in real time. This operation may detect a disconnection only when IoT Platform sends data or performs keepalive checks.

The response parameter declaration is as follows:

```
/**
* @brief check whether MQTT connection is established or not.
*
* @param [in] handle: specify the MQTT client.
*
* @retval true : MQTT in normal state.
* @retval false : MQTT in abnormal state.
* @see None.
*/
int IOT_MQTT_CheckStateNormal(void *handle);
```

MQTT keep alive

The device sends packets at least once in the time interval specified in keepalive_interval_ms. The packets sent include PING requests.

If the server does not receive any packet in the time interval specified in keepalive_interval_ms, IoT Platform will disconnect the device and the device needs to reconnect to the platform.

The value of keepalive_interval_ms can be configured in IOT_MQTT_Construct. IoT Platform uses this value as the heartbeat. The value range of keepalive_interval_ms is 60000-300000.

2.2.2 Establish MQTT over WebSocket connections

Context

IoT Platform supports MQTT over WebSocket. WebSocket is used to establish a connection. The MQTT protocol is used to communicate over the WebSocket connection.

Using WebSocket has the following advantages:

- Allows browser-based applications to establish persistent connections to the server.
- Uses port 433, which allows messages to pass through most firewalls.

Procedure

1. Certificate preparation

   The WebSocket protocol includes WebSocket and WebSocket Secure. Websocket and WebSocket Secure are used for unencrypted and encrypted connections, respectively.

Transport Layser Security (TLS) is used in WebSocket Secure connections. Like a TLS connection, a WebSocket Secure connection requires a *root certificate*.

2. Client selection

   Java clients can directly use the *Official client SDK* by replacing the connect URL in the SDK with a URL that is used by WebSocket. For clients that use other language versions or connections without using the official SDK, see *Open-source MQTT clients*. Make sure that the client supports WebSocket.

3. Connections

   An MQTT over WebSocket connection has a different protocol and port number in the connect URL from an MQTT over TCP connection. MQTT over WebSocket connections have the same parameters as MQTT over TCP connections. The securemode parameter is set to 2 and 3 for WebSocket Secure connections and WebSocket connections, respectively.

   - Connect to the domain name of the China (Shanghai) region: ${productKey}.iot-as-mqtt.cn-shanghai.aliyuncs.com:443

     Replace ${productKey} with your product key.
   - An MQTT Connect packet contains the following parameters:

```
mqttClientId: clientId+"|securemode=3,signmethod=hmacsha1,
timestamp=132323232|"
mqttUsername: deviceName+"&"+productKey
mqttPassword: sign_hmac(deviceSecret,content)sign. Sort the
content parameters in alphabetical order and sign them according
to the signing method.
content=Parameters sent to the server (productKey,deviceName,
timestamp,clientId). Sort these parameters in alphabetical order
and splice the parameters and parameter values.
```

     Where,

     - clientId: Specifies the client ID up to 64 characters. We recommend that you use a MAC address or SN.
     - timestamp: (Optional) Specifies the current time in milliseconds.
     - mqttClientId: Parameters within `||` are extended parameters.
     - signmethod: Specifies a signature algorithm.
     - securemode: Specifies the secure mode. Values include 2 (WebSocket Secure) and 3 (WebSocket).

The following are examples of MQTT Connect packets with predefined parameter values:

```
clientId=12345, deviceName=device, productKey=pk, timestamp=789,
signmethod=hmacsha1, deviceSecret=secret
```

- For a WebSocket connection:
  - Connection domain

    ```
    ws://pk.iot-as-mqtt.cn-shanghai.aliyuncs.com:443
    ```

  - Connection parameters

    ```

    mqttclientId=12345|securemode=3,signmethod=hmacsha1,timestamp=
    789|
    mqttUsername=device&pk
    mqttPasswrod=hmacsha1("secret","clientId12345deviceNamedevicep
    roductKeypktimestamp789").toHexString();
    ```

- For a WebSocket Secure connection:
  - Connection domain

    ```
    wss://pk.iot-as-mqtt.cn-shanghai.aliyuncs.com:443
    ```

  - Connection parameters

    ```

    mqttclientId=12345|securemode=2,signmethod=hmacsha1,timestamp=
    789|
    mqttUsername=device&pk
    mqttPasswrod=hmacsha1("secret","clientId12345deviceNamedevicep
    roductKeypktimestamp789").toHexString();
    ```

2.2.3 CoAP-based connections

Overview

Constrained Application Protocol (CoAP) is applicable to low-power devices that have limited resources, such as Narrowband IoT (NB-IoT) devices. The process for connecting NB-IoT devices to IoT Platform based on CoAP is described in *Figure 2-5: CoAP-based connections*. ￼

Figure 2-5: CoAP-based connections

The CoAP-based connection follows this process:

1. The NB-IoT module integrates an SDK for accessing Alibaba Cloud IoT Platform. The manufacturer requests unique certificates in the console, including ProductKey, DeviceName, and DeviceSecret, and installs them to devices.
2. The NB-IoT device accesses IoT Platform over the Internet service provider's (ISP's) cellular network. You need to contact the local ISP to make sure that the NB-IoT network has covered the region where the device is located.
3. After the device is connected to IoT Platform, the ISP's machine-to-machine (M2M) platform manages service usage and billing for traffic generated by the NB-IoT device.
4. You can report real-time data collected by the device to IoT Platform based on Constrained Application Protocol/User Datagram Protocol (CoAP/UDP). IoT Platform secures connections with more than 100 million NB-IoT devices and manages related data. The system connects with big data services, ApsaraDB, and report systems of Alibaba Cloud to achieve intelligent management.
5. IoT Platform provides data sharing functions and message pushing services to forward data to related service instances and quickly integrate device assets and actual applications.

Use the SDK to connect an NB-IoT device to IoT Platform

Follow these instructions to connect the device to IoT Platform using the C SDK. For more information, see *Download device SDKs*.

CoAP-based connection

The process of connecting devices using the CoAP protocol is described as follows:

1. Use the CoAP endpoint address: `endpoint = ${ProductKey}.iot-as-coap.cn-shanghai.aliyuncs.com:5684`. Replace ProductKey with the product key that you have requested.
2. Download the *root certificate* using the Datagram Transport Layer Security (DTLS)-based secure session.
3. Trigger device authentication to obtain the token for the device before the device sends data.
4. The device includes this token in reported data. If the token has expired, you need to request a new token. The system caches the token locally for 48 hours.

CoAP-based connection

1. Authenticate the device. You can use this function to request the token before the device sends data. You only need to request a token once.

```
POST /auth
Host: ${productKey}.iot-as-coap.cn-shanghai.aliyuncs.com
Port: 5684
Accept: application/json or application/cbor
Content-Format: application/json or application/cbor
payload: {"productKey":"ZG1EvTEa7NN","deviceName":"NlwaSPXsCp
TQuh8FxBGH","clientId":"mylight1000002","sign":"bccb3d2618afe74b3eab
12b94042f87b"}
```

Parameters are described as follows:

- Method: POST, Ony the POST method is supported.
- URL: /auth, URL address.
- Accept: the encoding format for receiving data. Currently, application/json and application/cbor are supported.
- Content-Format: encoding format of upstream data. Currently, application/json and application/cbor are supported.

The payload parameters and JSON data formats are described as follows:

Table 2-3: Payload parameters

Field Name	Required	Description
productKey	Yes	productKey, requested from the IoT Platform console.
deviceName	Required	deviceName, requested from the IoT Platform console.
sign	Required	Signature, format: hmacmd5(deviceSecret,content). The content includes all parameter values that are submitted to the instance, except for version, sign, resources, and signmethod. These parameter values are sorted in alphabetical order and without concatenated symbols.
signmethod	No	Algorithm type, hmacmd5 or hmacsha1.
clientId	Required	Client identifier, a maximum of 64 characters.
timestamp	No	Timestamp, not used in checks.

The response is as follows:

```
 response: {"token":"eyJ0b2tlbiI6IjBkNGUyNjkyZTNjZDQxOGU5MTA4
Njg4ZDdhNWI3MjUxIiwiZXhwIjoxNDk4OTg1MTk1fQ.DeQLSwVX8iBjdazjzNHG
5zcRECWcL49UoQfq1lXrJvI"}
```

The return codes are described as follows:

Table 2-4: Return codes

Code	Description	Payload	Remarks
2.05	Content	Authentication passed: token object	The request is correct.
4.00	Bad Request	Returned error message	The payload in the request is invalid.
4.04	Not Found	404 not found	The requested path does not exist.
4.05	Method Not Allowed	Supported method	The request method is not allowed.
4.06	Not Acceptable	The required Accept parameter	The Accept parameter is not the specified type.
4.15	Unsupported Content-Format	Supported content	The requested content is not the specified type.
5.00	Internal Server Error	Error message	The authentication request is timed out or an error occurs on the authentication server.

The SDK provides IOT_CoAP_Init and IOT_CoAP_DeviceNameAuth for building CoAP-based authentication on IoT Platform

Example code:

```
iotx_coap_context_t *p_ctx = NULL;
p_ctx = IOT_CoAP_Init(&config);
if (NULL ! = p_ctx) {
  IOT_CoAP_DeviceNameAuth(p_ctx);
  do {
    count ++;
    if (count == 11) {
      count = 1;
    }
    IOT_CoAP_Yield(p_ctx);
  } while (m_coap_client_running);
  IOT_CoAP_Deinit(&p_ctx);
} else {
  HAL_Printf("IoTx CoAP init failed\r\n");
}
```

The function statement is as follows:

```
/**
* @brief Initialize the CoAP client.
* This function is used to initialize the data structure and network
,
* and create the DTLS session.
*
* @param [in] p_config: Specify the CoAP client parameter.
*
* @retval NULL: The initialization failed.
* @retval NOT_NULL: The contex of CoAP client.
* @see None.
*/
iotx_coap_context_t *IOT_CoAP_Init(iotx_coap_config_t *p_config);

/**
* @brief Device name handle for authentication by the remote server.
*
* @param [in] p_context: Contex pointer to specify the CoAP client.
*
* @retval IOTX_SUCCESS: The authentication is passed.
* @retval IOTX_ERR_SEND_MSG_FAILED: Sending the authentication
message failed.
* @retval IOTX_ERR_AUTH_FAILED: The authentication failed or timed
out.
* @see iotx_ret_code_t.
*/
int IOT_CoAP_DeviceNameAuth(iotx_coap_context_t *p_context);
```

2. Set upstream data (${endpoint}/topic/${topic}).

   This is used to send data to a specified topic. To set ${topic}, choose**Products** > **Notifications**in the IoT Platform console.

   To send data to topic`/productkey/${deviceName}/pub`, use URL address `${productKey}.iot-as-coap.cn-shanghai.aliyuncs.com:5684/topic/productkey/device/pub` to report data if the current device name is device. You can only use a topic that has the publishing permission to report data.

   Example code:

```
POST /topic/${topic}
Host: ${productKey}.iot-as-coap.cn-shanghai.aliyuncs.com
Port: 5683
Accept: application/json or application/cbor
Content-Format: application/json or application/cbor
payload: ${your_data}
CustomOptions: number:61(token)
```

   Parameter description:

- Method: POST. The POST method is supported.
- URL: /topic/${topic}. Replace ${topic} with the topic of the current device.
- Accept: Received data encoding methods. Currently, application/json and application/cbor are supported.
- Content-Format: Upstream data encoding format. The service does not check this format.
- CustomOptions: Indicates the token that the device has obtained after authentication. Option Number: 61.

3. The SDK provides IOT_CoAP_SendMessage for sending data, and IOT_CoAP_GetMessagePayload and IOT_CoAP_GetMessageCode for receiving data.

   Example code:

```
/* send data */
static void iotx_post_data_to_server(void *param)
{
  char path[IOTX_URI_MAX_LEN + 1] = {0};
  iotx_message_t message;
  iotx_deviceinfo_t devinfo;
  message.p_payload = (unsigned char *)"{\"name\":\"hello world\"}";
  message.payload_len = strlen("{\"name\":\"hello world\"}");
  message.resp_callback = iotx_response_handler;
  message.msg_type = IOTX_MESSAGE_CON;
  message.content_type = IOTX_CONTENT_TYPE_JSON;
  iotx_coap_context_t *p_ctx = (iotx_coap_context_t *)param;
  iotx_set_devinfo(&devinfo);
  snprintf(path, IOTX_URI_MAX_LEN, "/topic/%s/%s/update/", (char *)
devinfo.product_key,
  (char *)devinfo.device_name);
  IOT_CoAP_SendMessage(p_ctx, path, &message);
}

/* receive data */
static void iotx_response_handler(void *arg, void *p_response)
{
  int len = 0;
  unsigned char *p_payload = NULL;
  iotx_coap_resp_code_t resp_code;
  IOT_CoAP_GetMessageCode(p_response, &resp_code);
  IOT_CoAP_GetMessagePayload(p_response, &p_payload, &len);
  Hal_printf ("[appl]: Message response code: 0x % x \ r \ n ",
resp_code );
  Hal_printf ("[appl]: Len: % d, payload: % s, \ r \ n ", Len, Fargo
 payload );
}
```

```
/**
* @ Brief send a message using the specific path to the server.
* The client must pass the authentication by the server before
sending messages.
*
* @param [in] p_context: Contex pointer to specify the CoAP client.
* @param [in] p_path: Specify the path name.
* @param [in] p_message: The message to be sent.
*
* @retval IOTX_SUCCESS: The message has been sent.
* @retval IOTX_ERR_MSG_TOO_LOOG: The message is too long.
* @retval IOTX_ERR_NOT_AUTHED: The client has not passed the
authentication by the server.
* @see iotx_ret_code_t.
*/
int IOT_CoAP_SendMessage(iotx_coap_context_t *p_context, char *
p_path, iotx_message_t *p_message);

/**
* @brief Retrieves the length and payload pointer of the specified
message.
*
* @param [in] p_message: The pointer to the message to get the
payload. This should not be NULL.
* @param [out] pp_payload: The pointer to the payload.
* @param [out] p_len: The size of the payload.
*
* @retval IOTX_SUCCESS: The payload has been obtained.
* @retval IOTX_ERR_INVALID_PARAM: The payload cannot be obtained due
 to invalid parameters.
* @see iotx_ret_code_t.
**/
int IOT_CoAP_GetMessagePayload(void *p_message, unsigned char **
pp_payload, int *p_len);

/**
* @brief Get the response code from a CoAP message.
*
* @param [in] p_message: The pointer to the message to add the
address information to.
* This should not be null.
* @param [out] p_resp_code: The response code.
*
* @retval IOTX_SUCCESS: The response code to the message has been
obtained.
* @retval IOTX_ERR_INVALID_PARAM: The pointer to the message is NULL
.
* @see iotx_ret_code_t.
**/
int IOT_CoAP_GetMessageCode(void *p_message, iotx_coap_resp_code_t *
p_resp_code);
```

Restrictions

- The topics follow the Message Queuing Telemetry Transport (MQTT) topic standard. The `coap://host:port/topic/${topic}` operation in CoAP can be used for all ${topic} topics and MQTT-based topics. You cannot specify parameters in the format of `` `? query_String=xxx` ``.
- The client locally caches the requested token that has been transmitted over DTLS and included in the response.

- The transmitted data size depends on the maximum transmission unit (MTU).The MTU less than 1KB is recommended.
- Only China (Shanghai) region supports CoAP-based connections.

Descriptions of other functions in the C SDK

- Use IOT_CoAP_Yield to receive data.

  Call this function to receive data. You can run this function in a single thread if the system allows.

```
/**
* @brief Handle CoAP response packet from remote server,
* and process timeout requests etc..
*
* @param [in] p_context: Contex pointer to specify the CoAP client.
*
* @return status.
* @see iotx_ret_code_t.
*/
int IOT_CoAP_Yield(iotx_coap_context_t *p_context);
```

- Use IOT_CoAP_Deinit to free up the memory.

```
/**
* @brief Deinitialize the CoAP client.
* This function is used to release the CoAP DTLS session
* and release the related resource. *
* @param [in] p_context: Contex pointer to specify the CoAP client.
*
* @return None.
* @see None.
*/
void IOT_CoAP_Deinit(iotx_coap_context_t **p_context);
```

For more information about how to use the C SDK, see *`\sample\coap\coap-example.c`*.

2.2.4 Establish communication over the HTTP protocol

Descriptions:

The description of communication over the HTTP protocol is as follows:

- The HTTP server endpoint = https://iot-as-http.cn-shanghai.aliyuncs.com.
- Only the HTTPS protocol is supported.
- Before transferring data, the device initializes authentication to obtain an access token.

- Each time a device publishes data to IoT Platform, the access token is required. If the token is invalid, the device must go through the authentication process again in order to obtain a new access token. Tokens can be cached locally for 48 hours.

Device authentication (${endpoint}/auth)

You call this operation to obtain an access token before transmitting data. You only need to perform this operation once unless the token becomes invalid.

```
POST /auth HTTP/1.1
Host: iot-as-http.cn-shanghai.aliyuncs.com
Content-Type: application/json
body: {"version":"default","clientId":"mylight1000002","signmethod
":"hmacsha1","sign":"4870141D4067227128CBB4377906C3731CAC221C","
productKey":"ZG1EvTEa7NN","deviceName":"NlwaSPXsCpTQuh8FxBGH","
timestamp":"1501668289957"}
```

Parameters:

- Method: POST. The POST method is supported.
- URL: /auth. This is the URL address. This address only supports the HTTPS protocol.
- Content-Type: Currently only application/json is supported.

The JSON data format has the following properties:

Table 2-5: Request parameters

Field name	Required	Description
ProductKey	Required	You can retrieve it in the IoT Platform console.
DeviceName	Required	You can retrieve it in the IoT Platform console.
ClientId	Required	The Client ID can be up to 64 characters. We recommend that you use either the device MAC address or serial number as the Client ID.
TimeStamp	Optional	The timestamp, which is used to verify that the request is valid within 15 minutes.
Sign	Required	The signature, the format is hmacmd5 (deviceSecret, content), where the content

Field name	Required	Description
		is all parameters (except version, sign, and signmethod) in alphabetical order, and the parameters are in listed together in sequence without splicing symbols.
SignMethod	Optional	The algorithm type, set the value to hmacmd5 or hmacsha1. The default value is hmacmd5.
Version	Optional	If you do not set the version, the value is set to default.

The output is as follows:

```
Body:
{
  "code": 0, // the status code
  "message": "success", // the message
  "Info ":{
    "token": "eyJ0eXBlIjoiSldUIiwiYWxnIjoiaG1hY3NoYTEifQ.eyJleHBpcm
UiOjE1MDI1MzE1MDc0NzcsInRva2VuIjoiODA0ZmFjYTBiZTE3NGUxNjliZjY0ODVlNWNi
NDg3MTkifQ.OjMwu29F0CY2YR_6oOyiOLXz0c8"
  }
}
```

Status codes

Table 2-6: Descriptions

Code	Message	Description
10000	common error	Unknown errors.
10001	param error	An exception occurred while requesting the parameter.
20000	auth check error	An error occurred while authorizing the device.
20004	update session error	An error occurred while updating the session.
40000	request too many	The throttling policy limits the number of requests.

SDKs use the IOT_HTTP_Init function and the IOT_HTTP_DeviceNameAuth function to authenticate devices.

```
handle = IOT_HTTP_Init(&http_param);
if (NULL ! = handle) {
  IOT_HTTP_DeviceNameAuth(handle);
  HAL_Printf("IoTx HTTP Message Sent\r\n");
} else {
  HAL_Printf("IoTx HTTP init failed\r\n");
  return 0;
}
```

Function declaration:

```
/**
* @ Initializes the HTTP client
* This function initializes data.
*
* @ param [in] pInitParams: Specify the init param information.
*
* @ retval NULL: Initialization failed.
* @ Retval not_null: The context of HTTP client.
* @ see None.
*/
Void * IOT_HTTP_Init (iotx_http_param_t * pinitparams );
/**
* @ brief handle Device Name authentication with remote server.
*
* @ param [in] handle: Pointer to context, specify the HTTP client.
*
* @ retval 0: Authentication successful.
* @retval -1 : Authentication failed.
* @see iotx_err_t.
*/
int IOT_HTTP_DeviceNameAuth(void *handle);
```

Uploaded data (${endpoint}/topic/${topic })

Transferring data to a specified topic, you can set ${topic} in the IoT Platform console by selecting**Products** > **Communication.**For example, for ${topic} `/productkey/${deviceName}/pub`, if the device name is device123, the device can report data through `https://iot-as-http.cn-shanghai.aliyuncs.com/topic/productkey/device123/pub` .

- Example:

```
POST /topic/${topic} HTTP/1.1
Host: iot-as-http.cn-shanghai.aliyuncs.com
Password: ${token}
Content-Type: application/octet-stream
Body: ${your_data}
```

Parameters:

  - Method: POST. The POST method is supported.
  - URL: /topic/${topic}. Replace the ${topic} placeholder with the name of the specific device topic. This address only supports the HTTPS protocol.
  - Content-Type: Currently only application/octet-stream is supported.
  - Password: The ${token} access token, which is returned after device authentication. This parameter is placed in the header.
- Body: The content sent to ${topic}, which is in binary byte with UTF-8 encoding.
- Return value:

```
Body:
{
  "code": 0, // the status code
  "message": "success", // the message
  "Info ":{
    "MessageId": 892687627916247040,
    "Data": byte [] // It is UTF-8 encoding, can be empty.
  }
}
```

Status codes:

Table 2-7: Descriptions

Code	Message	Description
10000	common error	Unknown errors.
10001	param error	An exception occurred while requesting the parameter.
20001	token is expired	The access token is invalid. You need to obtain a new token by calling the auth operation for authentication.
20002	token is null	The request header has no tokens.
20003	check token error	An error occurred during authentication. You need to obtain a new token by calling the auth operation for authentication.
30001	publish message error	An error occurred while uploading data.
40000	request too many	The throttling policy limits the number of requests.

C SDK

The SDK uses the IOT_HTTP_SendMessag function to send and receive data.

```
static int iotx_post_data_to_server(void *handle)
{
  Int ret =-1;
  char path[IOTX_URI_MAX_LEN + 1] = {0};
  char rsp_buf[1024];
  iotx_http_t *iotx_http_context = (iotx_http_t *)handle;
  iotx_device_info_t *p_devinfo = iotx_http_context->p_devinfo;
  iotx_http_message_param_t msg_param;
  msg_param.request_payload = (char *)"{\"name\":\"hello world\"}";
  msg_param.response_payload = rsp_buf;
  msg_param.timeout_ms = iotx_http_context->timeout_ms;
  msg_param.request_payload_len = strlen(msg_param.request_payload) +
1;
  msg_param.response_payload_len = 1024;
  msg_param.topic_path = path;
  HAL_Snprintf(msg_param.topic_path, IOTX_URI_MAX_LEN, "/topic/%s/%s/
update",
  p_devinfo->product_key, p_devinfo->device_name);
  if (0 == (ret = IOT_HTTP_SendMessage(iotx_http_context, &msg_param
))) {
    HAL_Printf("message response is %s\r\n", msg_param.response_p
ayload);
  } else {
    HAL_Printf("error\r\n");
  }
  return ret;
}
```

```
/**
* @brief Send a message with specific path to the server.
* Client must be authenticated by the server before sending message.
*
* @param [in] handle: pointer to context, specifies the HTTP client.
* @param [in] msg_param: Specifies the topic path and http payload
configuration.
*
* @retval 0 : Successful.
* @retval -1 : Failed.
* @see iotx_err_t.
*/
int IOT_HTTP_SendMessage(void *handle, iotx_http_message_param_t *
msg_param);
```

Additional functions in the C SDK.

The IOT_HTTP_Disconnect function close the HTTP connection to clear the cache.

```
/**
* @brief Closes the TCP connection between the client and server.
*
* @param [in] handle: pointer to context, specifies the HTTP client.
* @return None.
```

```
 * @ see None.
 */
void IOT_HTTP_Disconnect(void *handle);
```

Restrictions

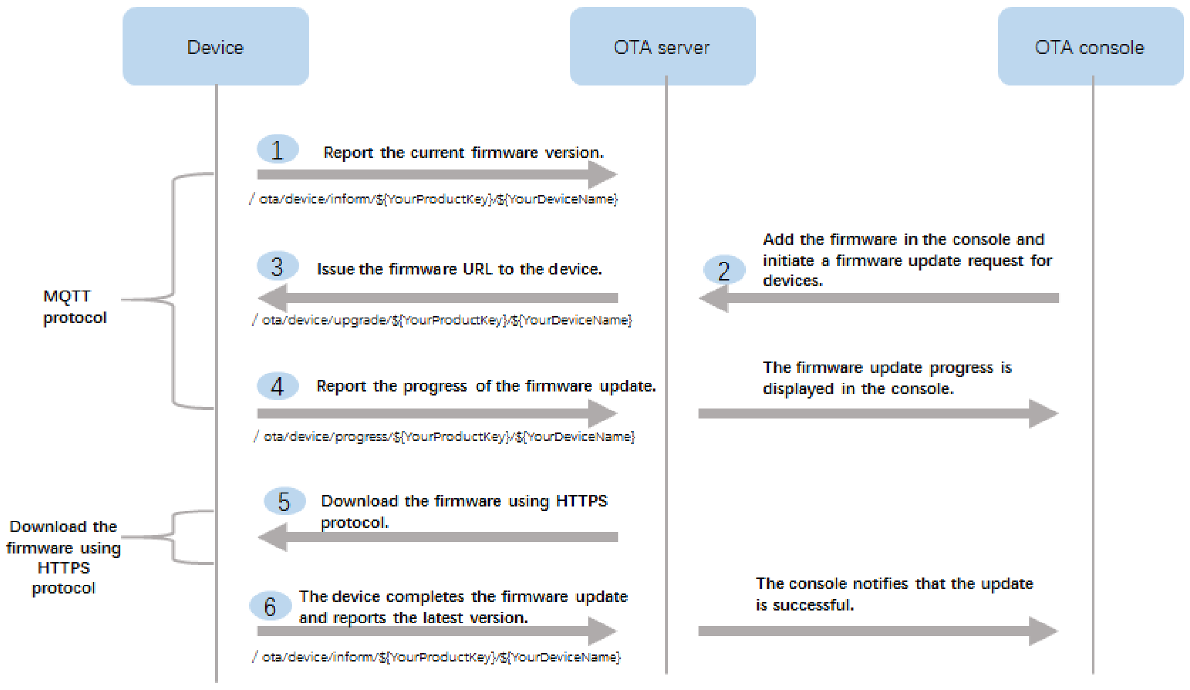
- The specifications of topics based on the HTTP protocol and MQTT protocols are same. The operation `https://iot-as-http.cn-shanghai.aliyuncs.com/topic/${topic}` can be reused for all topics based on the HTTP protocol and MQTT protocols. The operation can not set parameters using `? query_String=xxx`.
- The client caches the token locally. When the token expires, you need to obtain a new token. The new token will then be cached again.
- The size of data transferred by the upload operation is limited to 128 KB.
- Only China (Shanghai) region supports HTTP protocol communication.

2.3 OTA Development

Update firmware

In this example, IoT Platform uses the MQTT protocol to update the firmware. *Figure 2-6: Firmware update* shows the update process as follows:

Figure 2-6: Firmware update


Device
OTA server
OTA console
1 Report the current firmware version.
/ ota/device/inform/${YourProductKey}/${YourDeviceName}
2 Add the firmware in the console and initiate a firmware update request for devices.
3 Issue the firmware URL to the device.
/ ota/device/upgrade/${YourProductKey}/${YourDeviceName}
MQTT protocol
4 Report the progress of the firmware update.
/ ota/device/progress/${YourProductKey}/${YourDeviceName}
The firmware update progress is displayed in the console.
5 Download the firmware using HTTPS protocol.
Download the firmware using HTTPS protocol
6 The device completes the firmware update and reports the latest version.
/ ota/device/inform/${YourProductKey}/${YourDeviceName}
The console notifies that the update is successful.

Topics for firmware update

- The device publishes a message to this topic to report the firmware version to IoT Platform.

  ```
  /ota/device/inform/${productKey}/${deviceName}
  ```
- The device subscribes to this topic to receive a notification of the firmware update from IoT Platform.

  ```
  /ota/device/upgrade/${productKey}/${deviceName}
  ```
- The device publishes a message to this topic to report the progress of the firmware update to IoT Platform.

  ```
  /ota/device/progress/${productKey}/${deviceName}
  ```

Note:

- The device does not periodically send the firmware version to IoT Platform. Instead, the device sends the firmware version to IoT Platform only when the device starts.
- You can view the firmware version to check if the OTA update is successful.
- After you have configured the firmware update for multiple devices in the console of an OTA server, the update status of each device becomes Pending.

  When the OTA system receives the update progress from the device, the update status of the device changes to Updating.
- An offline device cannot receive any update notifications from the OTA server.

  When the device comes online again, the device notifies the OTA server that it is online. When the server receives the online notification, the server determines whether the device requires an update. If an update is required, the server sends an update notification to the device.

OTA code description

1. Install the firmware on a device, and start the device.

   The initialization code for OTA is as follows:

   ```
   h_ota = IOT_OTA_Init(PRODUCT_KEY, DEVICE_NAME, pclient);
   if (NULL == h_ota) {
     rc = -1;
     printf("initialize OTA failed\n");
   }
   ```

   Note:

The MQTT connection (the obtained MQTT client handle pclient) is used to initialize the OTA module.

The function is declared as follows:

```
/**
* @brief Initialize OTA module, and return handle.
* You must construct the MQTT client before you canll this interface
.
*
* @param [in] product_key: specify the product key.
* @param [in] device_name: specify the device name.
* @param [in] ch_signal: specify the signal channel.
*
* @retval 0 : Successful.
* @retval -1 : Failed.
* @see None.
*/
void *IOT_OTA_Init(const char *product_key, const char *device_name
, void *ch_signal);
/**
* @brief Report firmware version information to OTA server (optional
).
* NOTE: please
*
* @param [in] handle: specify the OTA module.
* @param [in] version: specify the firmware version in string format
.
*
* @retval 0 : Successful.
* @retval < 0 : Failed, the value is error code.
* @see None.
*/
int IOT_OTA_ReportVersion(void *handle, const char *version);
```

2. The device downloads the firmware from the received URL.

- IOT_OTA_IsFetching(): Identifies whether firmware is available for download.
- IOT_OTA_FetchYield(): Downloads a firmware package.
- IOT_OTA_IsFetchFinish(): Identifies whether the download has completed or not.

An example code is as follows:

```
// Identifies whether firmware is available for download.
if (IOT_OTA_IsFetching(h_ota)) {
  unsigned char buf_ota[OTA_BUF_LEN];
  uint32_t len, size_downloaded, size_file;
  do {
    //Iteratively downloads firmware.
    len = IOT_OTA_FetchYield(h_ota, buf_ota, OTA_BUF_LEN, 1);
    if (len > 0) {
      //Writes the firmware into the storage such as the flash.
    }
  } while (! IOT_OTA_IsFetchFinish(h_ota)); //Identifies whether the
 firmware download has completed or not.
}
exit: Ctrl ↩
/**
* @brief Check whether is on fetching state
*
* @param [in] handle: specify the OTA module.
*
* @retval 1 : Yes.
* @retval 0 : No.
* @see None.
*/
int IOT_OTA_IsFetching(void *handle);
/**
* @ Brief fetch firmware from remote server with specific timeout
value.
* NOTE: If you want to download more faster, the bigger 'buf' should
 be given.
*
* @param [in] handle: specify the OTA module.
* @param [out] buf: specify the space for storing firmware data.
* @param [in] buf_len: specify the length of 'buf' in bytes.
* @param [in] timeout_s: specify the timeout value in second.
*
* @retval < 0 : Error occur..
* @retval 0 : No any data be downloaded in 'timeout_s' timeout
period.
* @retval (0, len] : The length of data be downloaded in 'timeout_s
' timeout period in bytes.
* @see None.
*/
int IOT_OTA_FetchYield(void *handle, char *buf, uint32_t buf_len,
uint32_t timeout_s);
/**
* @brief Check whether is on end-of-fetch state.
*
* @param [in] handle: specify the OTA module.
*
* @retval 1 : Yes.
* @retval 0 : False.
* @see None.
*/
int IOT_OTA_IsFetchFinish(void *handle);
```

Note:

If you have insufficient device memory, you need to write the firmware into the system OTA partition while downloading the firmware.

3. Call IOT_OTA_ReportProgress() to report the download status.

   Example code:

```
if (percent - last_percent > 0) {
  IOT_OTA_ReportProgress(h_ota, percent, NULL);
}
```

```
IOT_MQTT_Yield(pclient, 100); //
```

You can upload the update progress to IoT Platform. The update progress (1% to 100%) is displayed in real time in the progress column of the updating list in the console.

You can also upload the following error codes:

- -1: Failed to update the firmware.
- -2: Failed to download the firmware.
- -3: Failed to verify the firmware.
- -4: Failed to write the firmware into flash.

4. Call IOT_OTA_Ioctl() to identify whether the downloaded firmware is valid. If the firmware is valid, the device will run with the new firmware at the next startup.

   Example code:

```
int32_t firmware_valid;
IOT_OTA_Ioctl(h_ota, IOT_OTAG_CHECK_FIRMWARE, &firmware_valid, 4);
  if (0 == firmware_valid) {
  printf("The firmware is invalid\n");
} else {
  printf("The firmware is valid\n");
}
```

   If the firmware is valid, modify the system boot parameters to make the hardware system run with the new firmware at the next startup. The modification method varies by hardware system.

```
/**
* @brief Get OTA information specified by 'type'.
* By this interface, you can get information like state, size of
file, md5 of file, etc.
*
* @param [in] handle: handle of the specific OTA
* @param [in] type: specify what information you want, see detail '
IOT_OTA_CmdType_t'
* @param [out] buf: specify buffer for data exchange
* @param [in] buf_len: specify the length of 'buf' in byte.
* @return
@verbatim
NOTE:
1) When type is IOT_OTAG_FETCHED_SIZE, 'buf' should be pointer of
uint32_t, and 'buf_len' should be 4.
2) When type is IOT_OTAG_FILE_SIZE, 'buf' should be pointer of
uint32_t, and 'buf_len' should be 4.
3) When type is IOT_OTAG_MD5SUM, 'buf' should be a buffer, and '
buf_len' should be 33.
4) When type is IOT_OTAG_VERSION, 'buf' should be a buffer, and '
buf_len' should be OTA_VERSION_LEN_MAX.
5) When type is IOT_OTAG_CHECK_FIRMWARE, 'buf' should be pointer of
uint32_t, and 'buf_len' should be 4.
0, firmware is invalid; 1, firmware is valid.
```

```
@endverbatim
*
* @retval 0 : Successful.
* @retval < 0 : Failed, the value is error code.
* @see None.
*/
int IOT_OTA_Ioctl(void *handle, IOT_OTA_CmdType_t type, void *buf,
size_t buf_len);
```

5. Call IOT_OTA_Deinit to terminate a connection and release the memory.

```
/**
* @brief Deinitialize OTA module specified by the 'handle', and
release the related resource.
* You must call this operation to release resource if reboot is not
invoked after downloading.
*
* @param [in] handle: specify the OTA module.
*
* @retval 0 : Successful.
* @retval < 0 : Failed, the value is error code.
* @see None.
*/
int IOT_OTA_Deinit(void *handle);
```

6. After the device restarts, the device runs with the new firmware and reports the new firmware version to IoT Platform. After the OTA module is initialized, call IOT_OTA_ReportVersion() to report the current firmware version. The code is as follows:

```
if (0 ! = IOT_OTA_ReportVersion(h_ota, "version2.0")) {
  rc = -1;
  printf("report OTA version failed\n");
}
```

2.4 Connect sub-devices to the cloud

This topic describes how to connect sub-devices to the cloud.

IoT Platform currently supports two node types, device and gateway.

- Device: refers to a device to which sub-devices cannot be mounted. Devices can connect directly to the IoT Hub. Alternatively, devices can connect as sub-devices mounted to gateways that are connected to the IoT Hub.
- Gateway: refers to a device to which sub-devices can be mounted. A gateway connects sub-devices to IoT Platform. Gateways can manage sub-devices, maintain their topological relationships with sub-devices, and synchronize these topological relationships to the cloud.

2.4.1 Connect sub-devices to IoT Platform

*Gateway and sud-device*Each device, either a gateway or a sub-device, works as a unique device on IoT Platform. Devices can use unique certificates for authentication when communicating with the cloud. You need to install the unique certificates to each device, including ProductKey, DeviceName, and DeviceSecret. Some sub-devices, such as Bluetooth devices and Zigbee devices, have high requirements for installing these unique certificates. You can select dynamic registration for authentication. In this way, you only need to register sub-devices in the cloud by providing ProductKey and DeviceName.

Prerequisites

The gateway has connected to the cloud by using *Unique-certificate-per-device authentication*.

Context

The ProductKey and DeviceName of the sub-device must be provided on IoT Platform before dynamic registration. When a gateway registers its sub-device, IoT Platform verifies DeviceName of this sub-device. After the DeviceName is verified, IoT Platform issues the DeviceSecret.

Follow these steps:

Procedure

1. Log on to the *IoT Platform console* .
2. Configure the gateway SDK.

   Note:

   A gateway can register its sub-devices, bring its sub-devices online or offline, maintain the topological relationship between the gateway and its sub-devices, and relay the communication between the sub-devices and IoT Platform. The manufacturer of the gateway device develops application features based on this SDK, such as connecting sub-devices to IoT Platform, receiving messages from sub-devices, publishing messages to sub-device topics to report status, subscribing to sub-device topics to obtain commands from IoT Platform, and routing messages to sub-devices.

   a) Download the SDK. For more information, see *Download device SDKs*. This section takes a C SDK for example.
   b) Log on to the Linux virtual machine (VM) and configure unique certificates of the gateway.
   c) Enable the feature of the gateway and sub-devices in this SDK.

You can configure this SDK by using code in `iotx-sdk-c\src\subdev` and following the demo that is provided in `sample\subdev`.

The example code consists of the following parts:

- Use the function in subdev to configure this SDK.

  ```
  demo_gateway_function(msg_buf, msg_readbuf);
  ```
- Examples of using the functions provided in the subdev_example_api.h file (encapsulation of topics) to develop code for gateways.

  ```
  demo_thing_function(msg_buf, msg_readbuf);
  ```
- Examples of using the functions provided in the subdev_example_api.h file (encapsulation of topics) to develop code for devices.

  ```
  demo_only_one_device(msg_buf, msg_readbuf);
  ```

Add sub-devices to a gateway:

- To enable unique-certificate-per-device authentication, register your sub-devices in IoT Platform and the gateway must have the ProductKey, DeviceName, and DeviceSecret values of the sub-devices. The gateway then uses IOT_Thing_Register/IOT_Subdevice_Register to register the sub-devices (registered type: IOTX_Thing_REGISTER_TYPE_STATIC).
- To enable dynamic authentication, you need to register sub-devices on IoT Platform. The gateway uses IOT_Thing_Register/IOT_Subdevice_Register to dynamically register the sub-devices (registered type: IOTX_Thing_REGISTER_TYPE_DYNAMIC).

  For more information about dynamic registration, see demo_gateway_function.
- The functions provided in the example/subdev_example_api.c/.h file encapsulate topics for properties, events, and services. You can use these functions directly without working with the corresponding topics.
- To specify the features of the gateway and sub-devices, you need to define `FEATURE_SUBDEVICE_ENABLED = y` in make.settings.

  To specify the features of the gateway or a sub-device, you need to define `FEATURE_SUBDEVICE_STATUS = subdevice` in make.settings.

3. The gateway establishes Message Queuing Telemetry Transport (MQTT) connections with IoT Platform.

4. Register a sub-device.

   The gateway obtains the ProductKey and DeviceName of the sub-device, and uses dynamic registration to obtain DeviceSecret from IoT Platform. We recommend that you use a global unique identifier (GUID), such as the medium access control (MAC) address, as the DeviceDame.

   - Request topic: /sys/{gw_productKey}/{gw_deviceName}/thing/sub/register

     Request format:

     ```
     {"id" : 123,"version":"1.0","params" : [{ "deviceName" : "
     deviceName1234", "productKey" : "1234556554"}],"method":"thing.sub
     .register"}
     ```
   - Response topic: /sys/{gw_productKey}/{gw_deviceName}/thing/sub/register_reply

     Response format:

     ```
     {"id":123,"code":200,"data":[{ "iotId":"12344", "productKey":"xxx
     ", "deviceName":"xxx", "deviceSecret":"xxx"}]}
     ```
   - The ProductKey and DeviceName values in the JSON object cannot be the same as the ProductKey and DeviceName values of the gateway.
   - The gateway and the sub-device send messages based on Quality of Service 0 (QoS 0).
   - The corresponding function is IOT_Subdevice_Register in this SDK. In this function, register_type supports static registration and dynamic registration. For more information about how to use this function and signing computing, see sample\subdev\subdev-example .c.
   - If you set register_type to IOTX_SUBDEV_REGISTER_TYPE_DYNAMIC, you can see an offline sub-device added to the gateway in the console after you use this function.
   - If you set register_type to IOTX_SUBDEV_REGISTER_TYPE_DYNAMIC, you only need to call this function once. If you call this function again, IoT Platform reports that the device already exists.
   - The current version of the SDK has a limitation. The system saves the device_secret value generated during dynamic registration in a global variable of the device. Therefore, the device_secret value is not persistent. The DeviceSecret is lost after you restart this device.

     If you need to use this function, modify iotx_subdevice_parse_register_reply in the iotx-sdk -c\src\subdev\iotx_subdev_api.c file to write device_secret to a module that supports data persistence.

- If you set register_type to IOTX_SUBDEV_REGISTER_TYPE_STATIC, after you use this function, you can see an existing sub-device in the console added to the gateway as an offline sub-device.

  ```
  /**
  * @brief Device register
  * This function is used to register a device and add topology.
  *
  * @param handle pointer to specify the gateway construction.
  * @param register type.
  * IOTX_SUBDEV_REGISTER_TYPE_STATIC
  * IOTX_SUBDEV_REGISTER_TYPE_DYNAMIC
  * @param product key.
  * @param device name.
  * @param timestamp. [if type = dynamic, must be NULL ]
  * @param client_id. [if type = dynamic, must be NULL ]
  * @param sign. [if type = dynamic, must be NULL ]
  * @param sign_method.
  * IOTX_SUBDEV_SIGN_METHOD_TYPE_SHA
  * IOTX_SUBDEV_SIGN_METHOD_TYPE_MD5
  *
  * @return 0, Logout success; -1, Logout fail.
  */
  int IOT_Subdevice_Register(void* handle,
  iotx_subdev_register_types_t type,
  const char* product_key,
  const char* device_name,
  Char * timestamp,
  char* client_id,
  char* sign,
  iotx_subdev_sign_method_types_t sign_type);
  ```

5. Build the topological relationship between the gateway and the sub-device in the cloud.

- Add a topological relationship.

  — Request topic: /sys/{gw_productKey}/{gw_deviceName}/thing/topo/add

    Request format:

    ```
    {
    "id" : "123",
    "version":"1.0",
    "params" : [{
    "deviceName" : "deviceName1234",
    "productKey" : "1234556554",
    "sign":"",
    "signmethod":"hmacSha1" //Supports hmacSha1, hmacSha256, and
    hmacMd5.
    "timestamp":"xxxx",
    "clientId":"xxx",//Indicates a local identifier, and can be
    identical with productKey&deviceName.
    }],
    "method":"thing.topo.add"
    ```

```
}
```

- Response topic: /sys/{gw_productKey}/{gw_deviceName}/thing/topo/add_reply

  Request format:

```
{
"id":"123",
"code":200,
"data":{}
}
```

- The ProductKey and DeviceName values in the JSON object cannot be the same as the ProductKey and DeviceName values of the gateway.
- The gateway and the sub-device send messages based on QoS 0.
- The IOT_Subdevice_Register function has encapsulated this feature of the SDK. You do not need to use other functions.

- Delete the topological relationship.
  - Request topic: /sys/{gw_productKey}/{gw_deviceName}/thing/topo/delete

    Request format:

```
{
"id" : 123,
"version":"1.0",
"params" : [{
"deviceName" : "deviceName1234",
"productKey" : "1234556554"
}],
"method":"thing.topo.delete"
}
```

  - Response topic: /sys/{gw_productKey}/{gw_deviceName}/thing/topo/delete_reply

```
{
"id":123,
"code":200,
"data":{}
}
```

  - The ProductKey and DeviceName values in the JSON object cannot be the same as the ProductKey and DeviceName of the gateway.
  - The gateway and the sub-device send messages based on QoS 0.
  - The IOT_Subdevice_Unregister function has encapsulated this feature of the SDK. You do not need to use other functions.

- Obtain the topological relationship.
  - Request topic: /sys/{gw_productKey}/{gw_deviceName}/thing/topo/get

    Request format:

    ```
    {
    "id" : 123,
    "version":"1.0",
    "params" : {},
    "method":"thing.topo.get"
    }
    ```
  - Response topic: /sys/{gw_productKey}/{gw_deviceName}/thing/topo/get_reply

    ```
    {
    "id":123,
    "code":200,
    "data": [{
    "deviceName" : "deviceName1234",
    "productKey" : "1234556554"
    }]
    }
    ```
  - The ProductKey and DeviceName values in the JSON object cannot be the same as the ProductKey and DeviceName of the gateway.
  - The gateway and the sub-device send messages based on QoS 0.
  - The corresponding function is IOT_Gateway_Get_TOPO in this SDK.
  - You can obtain the information about all sub-devices of this gateway, including their ProductKey, DeviceName, and DeviceSecret certificates, by using this function.

    Response parameter get_topo_reply is in JSON format. You need to parse the response.

    ```
    /**
    * @brief Gateway get topo
    * This function is used to publish a packet with topo/get topic
    , and wait for the reply (with TOPO_GET_REPLY topic).
    *
    * @param handle pointer to specify the Gateway.
    * @param get_topo_reply.
    * @param length [in/out]. in -- get_topo_reply buffer length,
    out -- reply length
    *
    * @return 0, logout success; -1, logout failed.
    */
    int IOT_Gateway_Get_TOPO(void* handle,
    char* get_topo_reply,
    uint32_t* length);
    ```

6. Connect the sub-device to IoT Platform.

- Request topic: /ext/session/{gw_productKey}/{gw_deviceName}/combine/login

  Request format:

```
{
"id":"123",
"params":{
"productKey":"xxxxx",// Sub-device ProductKey
"deviceName":"xxxx",//Sub-device DeviceName
"clientId":"xxxx",
"timestamp":"xxxx",
"signMethod":"hmacmd5 or hmacsha1 or hmacsha256",
"sign":"xxxxx", //Sub-device signature
"cleanSession":"true or false" // If this parameter is set to true
, the system clears all QoS 1- or 2-based messages missed by the
offline sub-device.
}
}
//Sub-devices with ProductKey, DeviceName and DeviceSecret follow
the same signature rules as the gateway.
sign = hmac_md5(deviceSecret, clientId123deviceNametestprodu
ctKey123timestamp123)
```

- Response topic: /ext/session/{gw_productKey}/{gw_deviceName}/combine/login_reply

  Response format:

```
{
"id":"123",
"code":200,
"message":"success"
}
```

- The ProductKey and DeviceName values in the JSON object cannot be the same as the ProductKey and DeviceName of the gateway.
- The gateway and the sub-device send messages based on QoS 0.
- The corresponding function is IOT_Subdevice_Login in this SDK. For more information about how to use this function, see sample\subdev\subdev-example.c.
- You can see the sub-device in online status in the console after you call this function.

```
/**
* @brief Subdevice login
* This function is used to publish a packet with the LOGIN topic,
wait for the reply (with
* LOGIN_REPLY topic), and then subscribe to some subdevice topics.
*
* @param handle pointer to specify the Gateway.
* @param product key.
* @param device name.
* @ Param timestamp. [If aster_type = dynamic, must be null]
* @param client_id. [if register_type = dynamic, must be NULL ]
* @param sign. [if register_type = dynamic, must be NULL ]
* @param sign method, HmacSha1 or HmacMd5.
* @param clean session, true or false.
*
* @return 0, login success; -1, login failed.
*/
int IOT_Subdevice_Login(void* handle,
const char* product_key,
const char* device_name,
const char* timestamp,
const char* client_id,
const char* sign,
iotx_subdev_sign_method_types_t sign_method_type,
iotx_subdev_clean_session_types_t clean_session_type);
```

7. Interact with the sub-device.

    - Request topic: You can use the topic in the format of /${productKey}/${deviceName}/xxx specified on IoT Platform, or in the format of /sys/${productKey}/${deviceName}/thing/xxx.
    - The gateway publishes data to the topic of the sub-device. In the topic, /${productKey}/${ deviceName}/ corresponds to ProductKey and DeviceName of the sub-device.
    - The format of the MQTT payload is unrestricted.
    - This SDK provides three functions: IOT_Gateway_Subscribe, IOT_Gateway_Unsubscribe , and IOT_Gateway_Publish, to subscribe to and publish messages. For more information about how to use this function, see sample\subdev\subdev-example.c.

```

/**
* @brief Gateway Subscribe
* This function is used to subscribe to some topics.
*
* @param handle pointer to specify the Gateway.
* @param topic list.
* @param QoS.
* @param function to receive data.
* @param topic_handle_func's userdata.
*
* @return 0, Subscribe success; -1, Subscribe fail.
*/
int IOT_Gateway_Subscribe(void* handle,
const char *topic_filter,
int qos,
iotx_subdev_event_handle_func_fpt topic_handle_func,
void *pcontext);
/**
* @brief Gateway Unsubscribe
* This function is used to unsubscribe from some topics.
*
* @param handle pointer to specify the Gateway.
* @param topic list.
*
* @return 0, Unsubscribe success; -1, Unsubscribe fail.
*/
int IOT_Gateway_Unsubscribe(void* handle, const char* topic_filter);
/**
```

```
* @brief Gateway Publish
* This function is used to publish some packets.
*
* @param handle pointer to specify the Gateway.
* @param topic.
* @param mqtt packet.
*
* @return 0, Publish success; -1, Publish fail.
*/
int IOT_Gateway_Publish(void* handle,
const char *topic_name,
iotx_mqtt_topic_info_pt topic_msg);
```

8. Disconnect the sub-device from IoT Platform.

- Request topic: /ext/session/{gw_productKey}/{gw_deviceName}/combine/logout

  Request format:

  ```
  {
  "id":123,
  "params":{
  "productKey":"xxxxx",//ProductKey of the sub-device
  "deviceName":"xxxxx",//DeviceName of the sub-device
  }
  }
  ```
- Response topic: /ext/session/{gw_productKey}/{gw_deviceName}/combine/logout_reply

  ```
  {
  "id":"123",
  "code":200,
  "message":"success"
  }
  ```
- The ProductKey and DeviceName values in the JSON object cannot be the same as the ProductKey and DeviceName values of the gateway.
- The gateway and the sub-device send messages based on QoS 0.
- The corresponding function is IOT_Subdevice_Logout in this SDK. For more information about how to use this function, see sample\subdev\subdev-example.c.
- You can see the sub-device in offline status in the console after you use this function.

```
/**
* @brief Subdevice logout
* This function is used to unsubscribe from some subdevice topics,
publish a packet with the
* LOGOUT topic, and then wait for the reply (with LOGOUT_REPLY topic
).
*
* @param handle pointer to specify the Gateway.
* @param product key.
* @param device name.
```

```
*
* @return 0, logout success; -1, logout failed.
*/
int IOT_Subdevice_Logout(void* handle,
const char* product_key,
const char* device_name);
```

9. Unregister the sub-device dynamically.

- Request topic: /sys/{gw_productKey}/{gw_deviceName}/thing/sub/unregister

  Request format:

```
{
"id" : 123,
"version":"1.0",
"params" : [{
"deviceName" : "deviceName1234",
"productKey" : "1234556554"
}],
"method":"thing.sub.unregister"
}
```

- Response topic: /sys/{gw_productKey}/{gw_deviceName}/thing/sub/unregister_reply

```
{
"id":123,
"code":200,
"data":{}
}
```

- The ProductKey and DeviceName values in the JSON object cannot be the same as the ProductKey and DeviceName of the gateway.
- The gateway and the sub-device send messages based on QoS 0.
- The corresponding function is IOT_Subdevice_Unregister in this SDK. For more information about how to use this function, see sample\subdev\subdev-example.c.
- The sub-device is destroyed and cannot be used after you call this function. Do not call this function if you still need the sub-device.

```
/**
* @brief Device unregister
* This function is used to delete the topological relationship and
unregister the device.
* The device must dynamically register again if you want to use this
 device after unregistration.
*
* @param handle pointer to specify the gateway construction.
* @param product key.
* @param device name.
*
* @ Return 0, unregister success;-1, unregister fail.
```

```
*/
int IOT_Subdevice_Unregister(void* handle,
const char* product_key,
const char* device_name);
```

Note:

- gw_productKey indicates ProductKey of the gateway.
- gw_deviceName indicates DeviceName of the gateway.

For more information about other functions in this SDK, use the subdev-example code.

2.5 Device shadows

2.5.1 Device shadow JSON format

Format of the device shadow JSON file

The format is as follows:

```
{
"state": {
"desired": {
"attribute1": integer2,
"attribute2": "string2",
...
"attributeN": boolean2
},
"reported": {
"attribute1": integer1,
"attribute2": "string1",
...
"attributeN": boolean1
}
},
"metadata": {
"desired": {
"attribute1": {
"timestamp": timestamp
},
"attribute2": {
"timestamp": timestamp
},
...
"attributeN": {
"timestamp": timestamp
}
},
"reported": {
"attribute1": {
"timestamp": timestamp
},
"attribute2": {
"timestamp": timestamp
},
...
"attributeN": {
"timestamp": timestamp
}
}
},
"timestamp": timestamp,
"version": version
}
```

The JSON properties are described in *Table 2-8: JSON property*.

Table 2-8: JSON property

Property	Description
desired	The desired status of the device. The application writes the desired property of the device, without accessing the device.
reported	The status that the device has reported. The device writes data to the reported property to report its latest status. The application obtains the status of the device by reading this property.
metadata	The device shadow service automatically updates metadata according to the updates in the device shadow JSON file. State metadata in the device shadow JSON file contains the timestamp of each property. The timestamp is represented as epoch time to obtain exact update time.
timestamp	The latest update time of the device shadow JSON file.
version	When you request updating the version of the device shadow, the device shadow checks whether the requested version is later than the current version. If the requested version is later than the current one, the device shadow updates to the requested version. If not, the device shadow rejects the request. The version number is increased according to the version update to ensure the latest device shadow JSON file version.

Example of the device shadow JSON file:

```

{
"state" : {
"desired" : {
"color" : "RED",
"sequence" : [ "RED", "GREEN", "BLUE" ]
},
"reported" : {
"color" : "GREEN"
}
},
"metadata" : {
"desired" : {
"color" : {
"timestamp" : 1469564492
},
"sequence" : {
"timestamp" : 1469564492
}
},
"reported" : {
"color" : {
"timestamp" : 1469564492
}
}
},
"timestamp" : 1469564492,
"version" : 1
}
```

Empty properties

- The device shadow JSON file contains the desired property only when you have specified the desired status. The following device shadow JSON file, which does not contain the desired property, is also effective:

```
{
"state" : {
"reported" : {
"color" : "red",
}
},
"metadata" : {
"reported" : {
"color" : {
"timestamp" : 1469564492
}
}
},
"timestamp" : 1469564492,
"version" : 1
}
```

- The following device shadow JSON file, which does not contain the reported property, is also effective:

```
{
"state" : {
"desired" : {
"color" : "red",
}
},
"metadata" : {
"desired" : {
"color" : {
```

```
"timestamp" : 1469564492
}
}
},
"timestamp" : 1469564492,
"version" : 1
}
```

Array

The device shadow JSON file can use an array, and must update this array as a whole when the update is required.

- Initial status:

```
{
"reported" : { "colors" : ["RED", "GREEN", "BLUE" ] }
}
```

- Update:

```
{
"reported" : { "colors" : ["RED"] }
}
```

- Final status:

```
{
"reported" : { "colors" : ["RED"] }
}
```

2.5.2 Device shadow data stream

IoT Platform predefines two topics for each device to enable data transmission. The predefined topics have fixed formats.

- topic*`/shadow/update/${productKey}/${deviceName}`*

  The device and application publish messages to this topic. IoT Platform updates the status to the device shadow after receiving messages from this topic.

- topic*`/shadow/get/${productKey}/${deviceName}`*

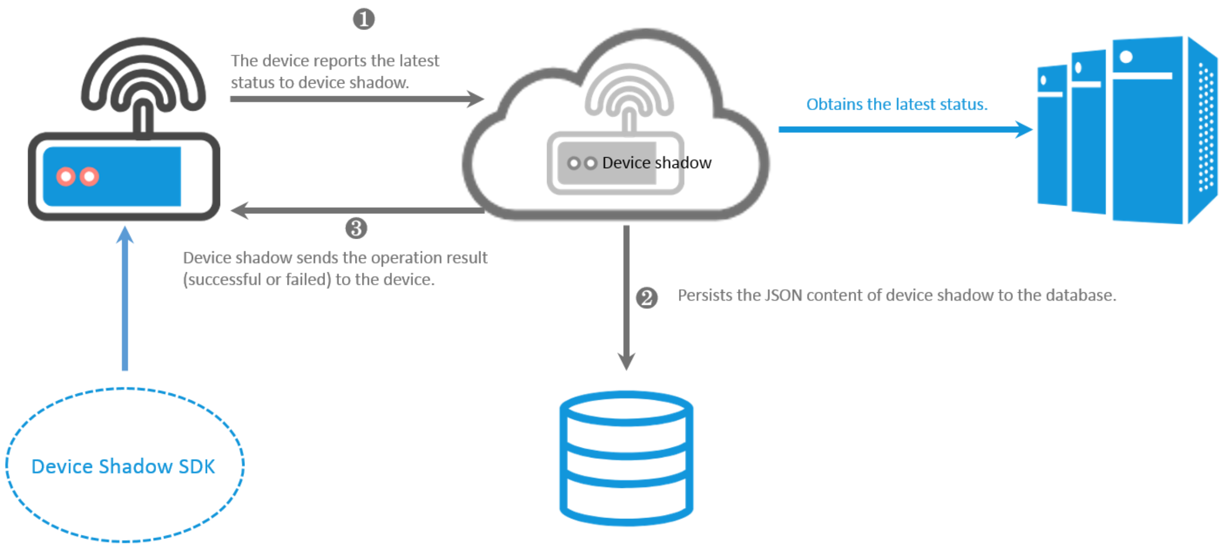
  The device shadow updates the status to this topic, and the device subscribes to the messages from this topic.

The following section takes the lightbulb (productkey:10000 and deviceName:lightbulb) as an example and demonstrates the communication among the device, device shadow, and the application. The device publishes and subscribes to the predefined topics with QoS 1.

Device reports status automatically

The flow chart is as shown in *Figure 2-7: Device reports status automatically*.

Figure 2-7: Device reports status automatically


❶
The device reports the latest status to device shadow.
Device shadow
Obtains the latest status.
❸
Device shadow sends the operation result (successful or failed) to the device.
❷
Persists the JSON content of device shadow to the database.
Device Shadow SDK

1. When the lightbulb is online, the device uses topic */shadow/update/10000/lightbulb* to report the latest status to the device shadow.

   Format of the JSON message:

   ```
   {
   "method": "update",
   "state": {
   "reported": {
   "color": "red"
   }
   },
   "version": 1
   }
   ```

   The JSON parameters are described in *Table 2-9: JSON parameters*.

   Table 2-9: JSON parameters

Parameter	Description
method	Represents the operation type when the device or application requests the device shadow. This parameter must be set to "update" when you perform updates.
state	Represents the status information that the device sends to the device shadow.

Parameter	Description
	The reported field is required. The status information is synchronized to the reported field of the device shadow.
version	Represents the version information contained in the request. The device shadow only accepts the request and updates to the specified version when the new version is later than the current version.

2. When the device shadow accepts the status reported by the lightbulb, the device shadow JSON file is successfully updated.

```
{
"state" : {
"reported" : {
"color" : "red"
}
},
"metadata" : {
"reported" : {
"color" : {
"timestamp" : 1469564492
}
}
},
"timestamp" : 1469564492
"version" : 1
}
```

3. After the update, the device shadow sends the result of the update to the device by sending a message to topic /shadow/get/10000/lightbulb. The device is subscribed to this topic.

   - If the update is successful, the message is as follows:

   ```
   {
   "method":"reply",
   "payload": {
   "status":"success",
   "version": 1
   },
   "timestamp": 1469564576
   }
   ```

   - If an error occurred during the update, the message is as follows:

   ```
   {
   "method":"reply",
   "payload": {
   "status":"error",
   "content": {
   "errorcode": "${errorcode}",
   "errormessage": "${errormessage}"
   }
   },
   ```

```
"timestamp": 1469564576
}
```

The meaning of the error codes is described in *Table 2-10: Error codes*.

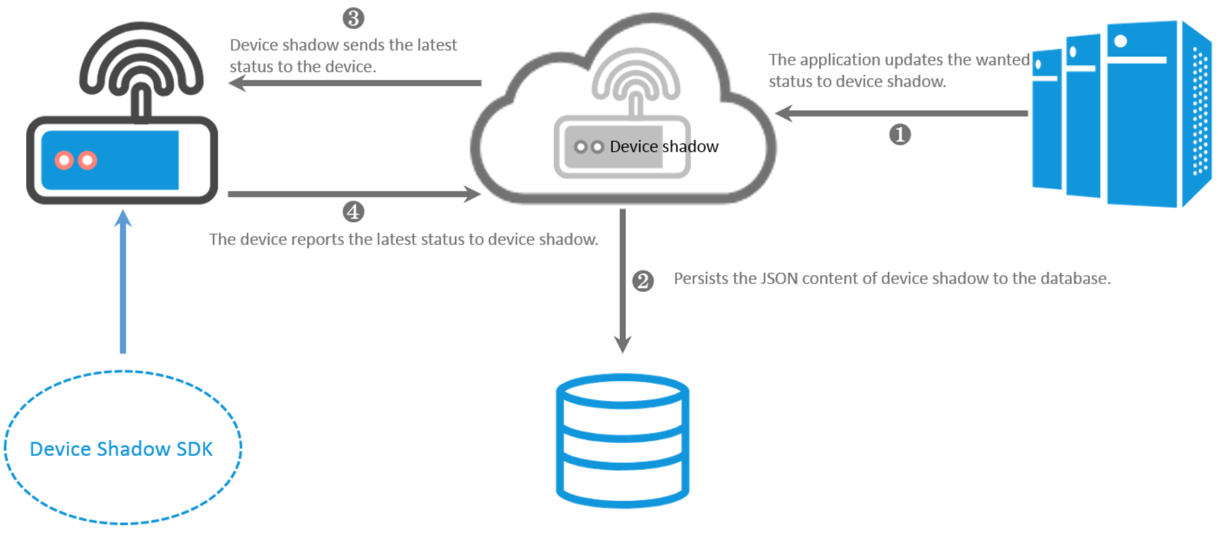
Table 2-10: Error codes

errorCode	errorMessage
400	Invalid JSON format.
401	The method field is missing.
402	The state field is missing.
403	Invalid version field.
404	The reported field is missing.
405	The reported field is blank.
406	Invalid method field.
407	The JSON file is empty.
408	The reported field contains more than 128 attributes.
409	Version conflict.
500	Server exception.

Application changes device status

The flow chart is as shown in *Figure 2-8: Application changes device status*.

Figure 2-8: Application changes device status


❸
Device shadow sends the latest status to the device.
Device shadow
The application updates the wanted status to device shadow.
❶
❹
The device reports the latest status to device shadow.
❷
Persists the JSON content of device shadow to the database.
Device Shadow SDK

1. The application sends a command to the device shadow to change the status of the lightbulb.

   The application sends a message to topic `/shadow/update/10000/lightbulb/`. The message is as follows:

   ```
   {
   "method": "update",
   "state": {
   "desired": {
   "color": "green"
   }
   },
   "version": 2
   }
   ```

2. The application sends an update request to update the device shadow JSON file. The device shadow JSON file is changed to:

   ```
   {
   "state" : {
   "reported" : {
   "color" : "red"
   },
   "desired" : {
   "color" : "green"
   }
   },
   "metadata" : {
   "reported" : {
   "color" : {
   "timestamp" : 1469564492
   }
   },
   "desired" : {
   "color" : {
   "timestamp" : 1469564576
   }
   }
   },
   "timestamp" : 1469564576,
   "version" : 2
   }
   ```

3. After the update, the device shadow sends a message to topic `/shadow/get/10000/lightbulb` and returns the result of update to the device. The result message is created by the device shadow.

   ```
   {
   "method":"control",
   "payload": {
   "status":"success",
   "state": {
   "reported": {
   "color": "red"
   ```

```
},
"desired": {
"color": "green"
}
},
"metadata": {
"reported": {
"color": {
"timestamp": 1469564492
}
},
"desired" : {
"color" : {
"timestamp" : 1469564576
}
}
}
},
"version": 2,
"timestamp": 1469564576
}
```

4. When the lightbulb is online and subscribed to topic `/shadow/get/10000/lightbulb`, the lightbulb receives the message and changes its color to green based on the desired field in the request file.

   If the timestamp shows that the command has expired, you can choose to not update the lightbulb status.

5. After the update, the device sends a message to topic `/shadow/update/10000/lightbulb` to report the latest status. The message is as follows:

```
{
"method": "update",
"state": {
"desired":"null"
},
"version": 3
}
```

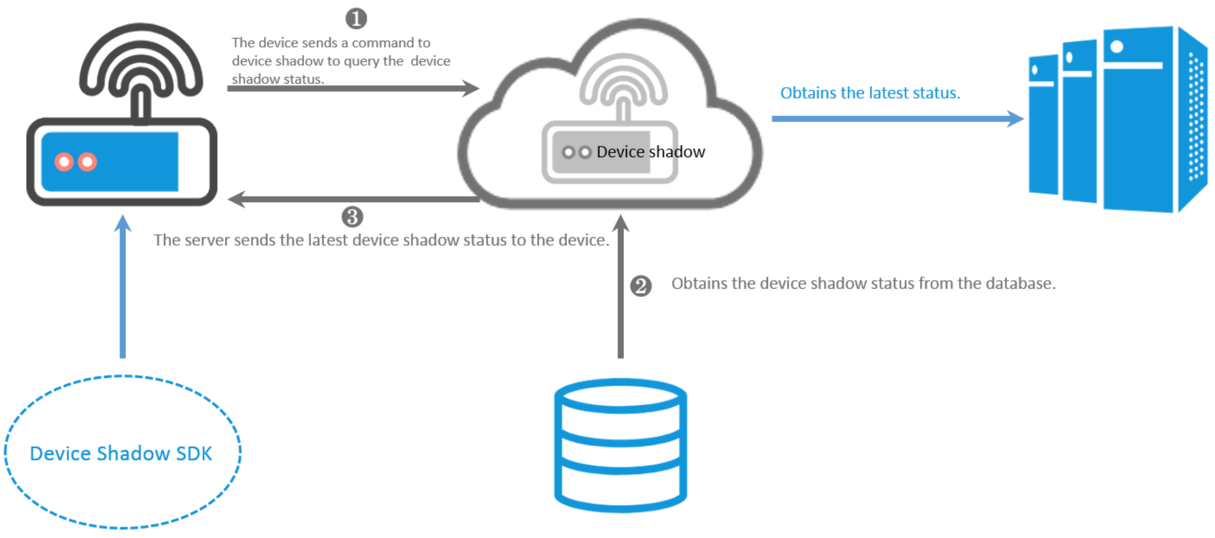
6. After the status has been reported, the device shadow is synchronously updated. The device shadow JSON file is as follows:

```
{
"state" : {
"reported" : {
"color" : "green"
}
},
"metadata" : {
"reported" : {
"color" : {
"timestamp" : 1469564577
}
},
"desired" : {
"timestamp" : 1469564576
}
},
"version" : 3
}
```

Device obtains device shadow

The flow chart is as follows:

Figure 2-9: Device obtains device shadow


❶ The device sends a command to device shadow to query the device shadow status.
Device shadow
Obtains the latest status.
❸ The server sends the latest device shadow status to the device.
❷ Obtains the device shadow status from the database.
Device Shadow SDK

1. The lightbulb sends a message to topic `/shadow/update/10000/lightbulb` and obtains the latest status saved in the device shadow. The message is as follows:

```
{
"method": "get"
}
```

2. When the device shadow receives above message, the device shadow sends a message to topic `/shadow/get/10000/lightbulb`. The lightbulb is subscribed to this topic, and the message is as follows:

```
{
"method":"reply",
"payload": {
"status":"success",
"state": {
"reported": {
"color": "red"
},
"desired": {
"color": "green"
}
},
"metadata": {
"reported": {
"color": {
"timestamp": 1469564492
}
},
"desired": {
"color": {
"timestamp": 1469564492
}
}
}
},
"version": 2,
"timestamp": 1469564576
}
```

3. The SDK uses the IOT_Shadow_Pull operation to obtain the device shadow.

```
IOT_Shadow_Pull(h_shadow);
```

Function prototype:

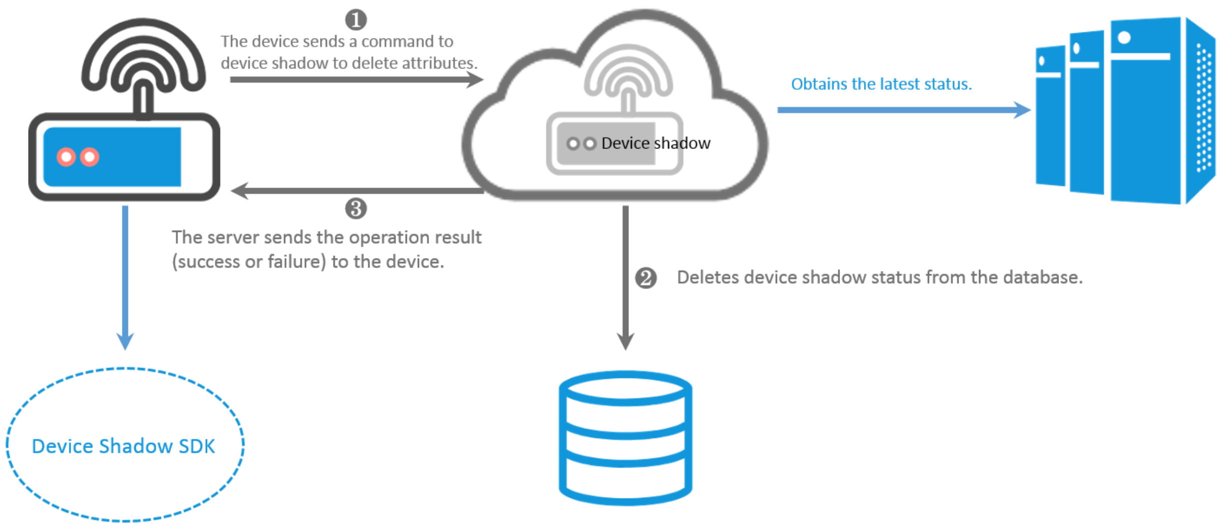
```

/**
* @brief Synchronize device shadow data from cloud.
* It is a synchronous interface.
* @param [in] handle: The handle of device shaodw.
* @retval SUCCESS_RETURN : Success.
* @retval other : See iotx_err_t.
* @see None.
*/
iotx_err_t IOT_Shadow_Pull(void *handle);
```

Device deletes device shadow attributes

The flow chart is as follows:

Figure 2-10: Delete device shadow attributes


❶ The device sends a command to device shadow to delete attributes.
Device shadow
Obtains the latest status.
❸ The server sends the operation result (success or failure) to the device.
❷ Deletes device shadow status from the database.
Device Shadow SDK

1. The lightbulb wants to delete the specified attributes saved in the device shadow. The lightbulb sends a JSON message to topic `/shadow/update/10000/lightbulb`. The message is as follows:

   - The JSON format for deleting the specified attributes:

   ```
   {
   "method": "delete",
   "state": {
   "reported": {
   "color": "null",
   "temperature":"null"
   }
   },
   "version": 1
   }
   ```

   - The JSON format for deleting all attributes:

   ```
   {
   "method": "delete",
   "state": {
   "reported":"null"
   },
   "version": 1
   }
   ```

   To delete attributes, you need to set method to delete and set the values of the attributes to null
   .

2. The SDK uses the IOT_Shadow_DeleteAttribute operation to delete device shadow attributes.

   The following parameters are provided:

```
IOT_Shadow_DeleteAttribute(h_shadow, &attr_temperature);
IOT_Shadow_DeleteAttribute(h_shadow, &attr_light);
IOT_Shadow_Destroy(h_shadow);
```

   Function prototype:

```
/**
* @brief Delete the specific attribute.
*
* @param [in] handle: The handle of device shaodw.
* @param [in] pattr: The parameter to be deleted from server.
* @retval SUCCESS_RETURN : Success.
* @retval other : See iotx_err_t.
* @see None.
*/
iotx_err_t IOT_Shadow_DeleteAttribute(void *handle, iotx_shado
w_attr_pt pattr);
```

2.5.3 Use device shadows

This topic describes the communication between devices, device shadows, and applications.

Context

A device shadow is the shadow that is built on IoT Platform based on a special topic for the related device. This device synchronizes status to the cloud using this device shadow. The cloud can quickly obtain the device status from the device shadow even when the device is not connected to IoT Platform.

Procedure

1. The C SDK provides the IOT_Shadow_Construct function to create the device shadow.

   The function is declared as follows:

```
/**
* @brief Construct the device shadow.
* This function is used to initialize the data structures, establish
 MQTT-based connections.
* and subscribe to the topic: "/shadow/get/${product_key}/${
device_name}".
*
* @param [in] pparam: The specific initial parameter.
* @retval NULL : The construction of the shadow failed.
* @retval NOT_NULL : The construction is successful.
* @see None.
*/
```

```
void *IOT_Shadow_Construct(iotx_shadow_para_pt pparam);
```

2. Use the IOT_Shadow_RegisterAttribute function to register the properties of the device shadow.

   The function is declared as follows:

```
/**
* @brief Create a data type registered to the server.
*
* @param [in] handle: The handle of the device shadow.
* @param [in] pattr: The parameter registered to the server.
* @retval SUCCESS_RETURN : Success.
* @retval other : See iotx_err_t.
* @see None.
*/
iotx_err_t IOT_Shadow_RegisterAttribute(void *handle, iotx_shado
w_attr_pt pattr);
```

3. You can use the IOT_Shadow_Pull function in the C SDK to synchronize device status to IoT Platform whenever the device shadow starts.

   The function is declared as follows:

```

/**
* @brief Synchronize device shadow data to the cloud.
* It is a synchronization function.
* @param [in] handle: The handle of the device shadow.
* @retval SUCCESS_RETURN : Success.
* @retval other : See iotx_err_t.
* @see None.
*/
iotx_err_t IOT_Shadow_Pull(void *handle);
```

4. When the device updates its status, you can use IOT_Shadow_PushFormat_Init, IOT_Shadow_PushFormat_Add, and IOT_Shadow_PushFormat_Finalize in the C SDK to update the device status, and use IOT_Shadow_Push in the C SDK to synchronize the status to the cloud.

   The function is declared as follows:

```


/**
* @brief Start processing the structure of the data type format.
*
* @param [in] handle: The handle of the device shadow.
* @param [out] pformat: The format structure of the device shadow.
* @param [in] buf: The buffer that stores the device shadow.
* @param [in] size: The maximum length of the device shadow
attribute.
* @retval SUCCESS_RETURN : Success.
* @retval other : See iotx_err_t.
* @see None.
```

```
*/
iotx_err_t IOT_Shadow_PushFormat_Init(
                    void *handle,
                    format_data_pt pformat,
                    char *buf,
                    uint16_t size);


/**
* @brief The format of the attribute name and value for the update.
*
* @param [in] handle: The handle of the device shadow.
* @param [in] pformat: The format structure of the device shadow.
* @param [in] pattr: The data type format created in the added
member attributes.
* @retval SUCCESS_RETURN : Success.
* @retval other : See iotx_err_t.
* @see None.
*/
iotx_err_t IOT_Shadow_PushFormat_Add(
                    void *handle,
                    format_data_pt pformat,
                    iotx_shadow_attr_pt pattr);


/**
* @ Brief Complete processing the structure of the data type format.
*
* @param [in] handle: The handle of the device shadow.
* @ Param [in] pformat: The format structure of the device shadow.
* @retval SUCCESS_RETURN : Success.
* @retval other : See iotx_err_t.
* @see None.
*/
iotx_err_t IOT_Shadow_PushFormat_Finalize(void *handle, format_dat
a_pt pformat);
```

5. To disconnect the device from IoT Platform, use IOT_Shadow_DeleteAttribute and IOT_Shadow_Destroy in the C SDK to delete all properties that have been created for this device on IoT Platform, and release the device shadow.

   The function is declared as follows:

```

/**
* @brief Deconstruct the specific device shadow.
*
* @param [in] handle: The handle of the device shadow.
* @retval SUCCESS_RETURN : Success.
* @retval other : See iotx_err_t.
* @see None.
*/
```

```
iotx_err_t IOT_Shadow_Destroy(void *handle);
```

2.6 Configure a TSL-based device

This topic describes how to configure a device based on a TSL model.

Note:

Only IoT Platform Pro supports this feature.

Prerequisites

Create a product, add a device, and define the TSL in the IoT Platform console. A TSL model describes the properties, services, and events of a device, as shown in figure *Figure 2-11: Create devices*.

Figure 2-11: Create devices

Establish a connection to IoT Platform

1. For more information about establishing an MQTT connection to connect a device and IoT Platform, see *Establish MQTT over TCP connections*.
2. Call the linkkit_start operation in the device SDK to establish a connection to IoT Platform and subscribe to topics.

   When you use the device SDK, save a shadow for the device. A shadow is an abstraction of a device, which is used to retrieve the status information of the device. The interaction process between a device and IoT Platform is a synchronization process between the device and shadow and between the shadow and IoT Platform.

   Variable get_tsl_from_cloud is used to synchronize the TSL model from IoT Platform when the device comes online.

   - get_tsl_from_cloud = 0: Indicates that a TSL model has been pre-defined. TSL_STRING is used as the standard TSL model.

     The SDK copies the TSL model that is created in the console, uses the TSL model to define TSL_STRING in linkkit_sample.c, and then calls the linkkit_set_tsl operation to set the pre-defined TSL model.

     Note:

Use the C escape character correctly.

- get_tsl_from_cloud = 1: Indicates that no TSL model has been pre-defined. The SDK must dynamically retrieve the TSL model from IoT Platform.

  Dynamically retrieving a TSL model consumes a large amount of memory and bandwidth. The specific consumption depends on the complexity of the TSL model. A TSL model of 10 KB consumes about 20 KB of memory and 10 KB of bandwidth.

3. Use the linkkit_ops_t parameter to register the callback.

```
linkkit_start(8, get_tsl_from_cloud, linkkit_loglevel_debug, &
alinkops, linkkit_cloud_domain_sh, sample_ctx);
if (! get_tsl_from_cloud) {
   linkkit_set_tsl(TSL_STRING, strlen(TSL_STRING));
}
```

Function implementation:

```
typedef struct _linkkit_ops {
  int (*on_connect)(void *ctx);
  int (*on_disconnect)(void *ctx);
  int (*raw_data_arrived)(void *thing_id, void *data, int len, void
 *ctx);
  int (*thing_create)(void *thing_id, void *ctx);
  int (*thing_enable)(void *thing_id, void *ctx);
  int (*thing_disable)(void *thing_id, void *ctx);
#ifdef RRPC_ENABLED
  int (*thing_call_service)(void *thing_id, char *service, int
request_id, int rrpc, void *ctx);
#else
  int (*thing_call_service)(void *thing_id, char *service, int
request_id, void *ctx);
#endif /* RRPC_ENABLED */
  int (*thing_prop_changed)(void *thing_id, char *property, void *
ctx);
} linkkit_ops_t;
/**
* @brief start linkkit routines, and install callback funstions(
async type for cloud connecting).
*
* @param max_buffered_msg, specify max buffered message size.
* @param ops, callback function struct to be installed.
* @param get_tsl_from_cloud, config if device need to get tsl from
 cloud(! 0) or local(0), if local selected, must invoke linkkit_se
t_tsl to tell tsl to dm after start complete.
* @param log_level, config log level.
* @param user_context, user context pointer.
* @param domain_type, specify the could server domain.
*
* @return int, 0 when success, -1 when fail.
*/
int linkkit_start(int max_buffered_msg, int get_tsl_from_cloud
, linkkit_loglevel_t log_level, linkkit_ops_t *ops, linkkit_cl
oud_domain_type_t domain_type, void *user_context);
/**
```

```
* @brief install user tsl.
*
* @param tsl, tsl string that contains json description for thing
object.
* @param tsl_len, tsl string length.
*
* @return pointer to thing object, NULL when fails.
*/
extern void* linkkit_set_tsl(const char* tsl, int tsl_len);
```

4. After you have connected the device to IoT Platform, log on to the IoT Platform console and verify whether the device has come online.

   Figure 2-12: Device comes online

Send property changes to IoT Platform

1. When the properties of a device change, the device automatically sends the changes to IoT Platform by publishing to topic `/sys/{productKey}/{deviceName}/thing/event/property/post`.

   Request:

```
TOPIC: /sys/{productKey}/{deviceName}/thing/event/property/post
REPLY TOPIC: /sys/{productKey}/{deviceName}/thing/event/property/
post_reply
request
{
"id" : "123",
"version":"1.0",
"params" : {
"PowerSwitch" : 1
},
"method":"thing.event.property.post"
}
response
{
"id":"123",
"code":200,
"data":{}
}
```

2. The SDK calls the linkkit_set_value operation to modify the property of the shadow, and then calls the linkkit_trigger_event operation to synchronize the shadow to IoT Platform.

   Note:

   The device will automatically send the current property of the shadow to IoT Platform.

Function:

```
linkkit_set_value(linkkit_method_set_property_value, sample->thing,
EVENT_PROPERTY_POST_IDENTIFIER, value, value_str); // set value
return linkkit_trigger_event(sample->thing, EVENT_PROPERTY_POST_
IDENTIFIER, NULL); // update value to cloud
```

Function implementation:

```
/**
* @brief set value to property, event output, service output items.
* if identifier is struct type or service output type or event
output type, use '.' as delimeter like "identifier1.ientifier2"
* to point to specific item.
* value and value_str could not be NULL at the same time;
* if value and value_str both as not NULL, value shall be used and
value_str will be ignored.
* if value is NULL, value_str not NULL, value_str will be used.
* in brief, value will be used if not NULL, value_str will be used
only if value is NULL.
*
* @param method_set, specify set value type.
* @param thing_id, pointer to thing object, specify which thing to
set.
* @param identifier, property, event output, service output
identifier.
* @param value, value to set.(input int* if target value is int type
 or enum or bool, float* if float type,
* long long* if date type, char* if text type).
* @param value_str, value to set in string format if value is null.
*
* @return 0 when success, -1 when fail.
*/
extern int linkkit_set_value(linkkit_method_set_t method_set, const
void* thing_id, const char* identifier,
const void* value, const char* value_str);
/**
* @brief trigger a event to post to cloud.
*
* @param thing_id, pointer to thing object.
* @param event_identifier, event identifier to trigger.
* @param property_identifier, used when trigger event with method "
event.property.post", if set, post specified property, if NULL, post
 all.
*
* @return 0 when success, -1 when fail.
*/
extern int linkkit_trigger_event(const void* thing_id, const char*
event_identifier, const char* property_identifier);
```

Get a device property on IoT Platform

1. You can log on to the IoT Platform console and use topic `/sys/{productKey}/{deviceName}/thing/service/property/get` to get a property of a device.

Request:

```
TOPIC: /sys/{productKey}/{deviceName}/thing/service/property/get
REPLY TOPIC: /sys/{productKey}/{deviceName}/thing/service/property/
get_reply
request
{
"id" : "123",
"version":"1.0",
"params" : [
"powerSwitch"
],
"method":"thing.service.property.get"
}
response
{
"id":"123",
"code":200,
"data":{
"powerSwitch":0
}
}
```

2. When the device receives the GET command from IoT Platform, the SDK executes the command to read the property value from the shadow and returns the value to IoT Platform.

Set a device property on IoT Platform

1. You can log on to the IoT Platform console and use topic `/sys/{productKey}/{deviceName}/thing/service/property/set` to set a property of a device client.

   Request:

```
TOPIC: /sys/{productKey}/{deviceName}/thing/service/property/set
REPLY TOPIC: /sys/{productKey}/{deviceName}/thing/service/property/
set_reply
payload:
{
"id" : "123",
"version":"1.0",
"params" : {
"PowerSwitch" : 0,
},
"method":"thing.service.property.set"
}
response
{
"id":"123",
"code":200,
"data":{}
}
```

2. The SDK registers the thing_prop_changed callback function in the linkkit_ops_t parameter of the linkkit_start method to respond to the request sent from IoT Platform for setting device properties.
3. The linkkit_get_value parameter in the callback function is used to get the device property of the shadow, which is the same as the device property that is modified on IoT Platform.
4. After setting the new property value, you can implement the linkkit_answer_service function to return the result to IoT Platform. You can choose whether to perform this task based on your business needs.

   Function implementation:

```
static int thing_prop_changed(void* thing_id, char* property, void*
ctx)
{
char* value_str = NULL;
... ...
linkkit_get_value(linkkit_method_get_property_value, thing_id,
property, NULL, &value_str);
LINKKIT_PRINTF("#### property(%s) new value set: %s ####\n",
property, value_str);
}
/* do user's process logical here. */
linkkit_trigger_event(thing_id, EVENT_PROPERTY_POST_IDENTIFIER,
property);
return 0;
}
```

Callback function:

```
int (*thing_prop_changed)(void *thing_id, char *property, void *ctx);
```

Function implementation:

```
/**
* @brief get value from property, event output, service input/output
items.
* if identifier is struct type or service input/output type or event
output type, use '.' as delimeter like "identifier1.ientifier2"
* to point to specific item.
* value and value_str could not be NULL at the same time;
* if value and value_str both as not NULL, value shall be used and
value_str will be ignored.
* if value is NULL, value_str not NULL, value_str will be used.
* in brief, value will be used if not NULL, value_str will be used
only if value is NULL.
* @param method_get, specify get value type.
* @param thing_id, pointer to thing object, specify which thing to get
.
* @param identifier, property, event output, service input/output
identifier.
```

```
* @param value, value to get(input int* if target value is int type or
 enum or bool, float* if float type,
* long long* if date type, char* if text type).
* @param value_str, value to get in string format. DO NOT modify this
when function returns,
* user should copy to user's own buffer for further process.
* user should NOT free the memory.
*
* @return 0 when success, -1 when fail.
*/
extern int linkkit_get_value(linkkit_method_get_t method_get, const
void* thing_id, const char* identifier,
void* value, char** value_str);
```

Function:

```
linkkit_set_value(linkkit_method_set_service_output_value, thing,
identifier, &sample->service_custom_output_contrastratio, NULL);
linkkit_answer_service(thing, service_identifier, request_id, 200);
```

Function implementation:

```
/**
* @brief set value to property, event output, service output items.
* if identifier is struct type or service output type or event output
type, use '.' as delimeter like "identifier1.ientifier2"
* to point to specific item.
* value and value_str could not be NULL at the same time;
* if value and value_str both as not NULL, value shall be used and
value_str will be ignored.
* if value is NULL, value_str not NULL, value_str will be used.
* in brief, value will be used if not NULL, value_str will be used
only if value is NULL.
*
* @param method_set, specify set value type.
* @param thing_id, pointer to thing object, specify which thing to set
.
* @param identifier, property, event output, service output identifier
.
* @param value, value to set.(input int* if target value is int type
or enum or bool, float* if float type,
* long long* if date type, char* if text type).
* @param value_str, value to set in string format if value is null.
*
* @return 0 when success, -1 when fail.
*/
extern int linkkit_set_value(linkkit_method_set_t method_set, const
void* thing_id, const char* identifier,
const void* value, const char* value_str);
/**
* @brief answer to a service when a service requested by cloud.
*
* @param thing_id, pointer to thing object.
* @param service_identifier, service identifier to answer, user should
 get this identifier from handle_dm_callback_fp_t type callback
* report that "dm_callback_type_service_requested" happened, use this
function to generate response to the service sender.
```

```
* @param response_id, id value in response payload. its value is from
 "dm_callback_type_service_requested" type callback function.
* use the same id as the request to send response as the same
communication session.
* @param code, code value in response payload. for example, 200 when
service is successfully executed, 400 when not successfully executed.
* @param rrpc, specify rrpc service call or not.
*
* @return 0 when success, -1 when fail.
*/
extern int linkkit_answer_service(const void* thing_id, const char*
service_identifier, int response_id, int code);
```

IoT Platform requests a service from the device.

1. IoT Platform uses topic `/sys/{productKey}/{deviceName}/thing/service/{dsl.service.identifer}` to invoke a service from the device. The service is defined in dsl.service.identifer of the standard TSL model.

```
TOPIC: /sys/{productKey}/{deviceName}/thing/service/{dsl.service.
identifer}
REPLY TOPIC:
/sys/{productKey}/{deviceName}/thing/service/{dsl.service.identifer}
_reply
request
{
"id" : "123",
"version":"1.0",
"params" : {
"SprinkleTime" : 50,
"SprinkleVolume" : 600
},
"method":"thing.service.AutoSprinkle"
}
response
{
"id":"123",
"code":200,
"data":{}
}
```

2. The SDK registers the thing_call_service callback function in the linkkit_ops_t parameter of the linkkit_start method, to send a response to the service request.
3. After setting the new property value, you must call the linkkit_answer_service function to send a response to IoT Platform.

   Function:

```
int (*thing_call_service)(void *thing_id, char *service, int
request_id, void *ctx);
```

   Function implementation:

```
static int handle_service_custom(sample_context_t* sample, void*
thing, char* service_identifier, int request_id)
{
char identifier[128] = {0};
/*
* get iutput value.
*/
snprintf(identifier, sizeof(identifier), "%s.%s", service_identifier
, "SprinkleTime");
linkkit_get_value(linkkit_method_get_service_input_value, thing,
identifier, &sample->service_custom_input_transparency, NULL);
/*
* set output value according to user's process result.
*/
snprintf(identifier, sizeof(identifier), "%s.%s", service_identifier
, "SprinkleVolume");
sample->service_custom_output_contrastratio = sample->service_cu
stom_input_transparency >= 0 ? sample->service_custom_input
_transparency : sample->service_custom_input_transparency * -1;
linkkit_set_value(linkkit_method_set_service_output_value, thing,
identifier, &sample->service_custom_output_contrastratio, NULL);
linkkit_answer_service(thing, service_identifier, request_id, 200);
return 0;
}
```

Send events to IoT Platform

1. A device subscribes to topic `/sys/{productKey}/{deviceName}/thing/event/{dsl.event.identifer}/post` to send an event to IoT Platform. The event is defined in dsl.event.identifer of the standard TSL model.

   Request:

```
TOPIC: /sys/{productKey}/{deviceName}/thing/event/{dsl.event.
identifer}/post
REPLY TOPIC: /sys/{productKey}/{deviceName}/thing/event/{dsl.event.
identifer}/post_reply
request
{
"id" : "123",
"version":"1.0",
"params" : {
"ErrorCode" : 0
},
"method":"thing.event.Error.post"
}
response:
{
"id" : "123",
"code":200,
"data" : {}
}
```

2. The SDK calls the linkkit_trigger_event method to send an event to IoT Platform.

Function:

```
static int post_event_error(sample_context_t* sample)
{
char event_output_identifier[64];
snprintf(event_output_identifier, sizeof(event_output_identifier),
 "%s.%s", EVENT_ERROR_IDENTIFIER, EVENT_ERROR_OUTPUT_INFO_IDENTIFIER
);
int errorCode = 0;
linkkit_set_value(linkkit_method_set_event_output_value,
sample->thing,
event_output_identifier,
&errorCode, NULL);
return linkkit_trigger_event(sample->thing, EVENT_ERROR_IDENTIFIER,
NULL);
}
```

Function implementation:

```
/**
* @brief trigger a event to post to cloud.
*
* @param thing_id, pointer to thing object.
* @param event_identifier, event identifier to trigger.
* @param property_identifier, used when trigger event with method "
event.property.post", if set, post specified property, if NULL, post
all.
*
* @return 0 when success, -1 when fail.
*/
extern int linkkit_trigger_event(const void* thing_id, const char*
event_identifier, const char* property_identifier);
```

3 Java SDK

This topic describes how to connect devices to Alibaba Cloud IoT Platform over the MQTT protocol. The Java SDK is used as an example.

Prerequisites

In this demo, a Maven project is used. Install Maven first.

Context

This demo is not made for the Android operating system. If you are using Android, see open-source library *https://github.com/eclipse/paho.mqtt.android*.

Procedure

1. Download the mqttClient SDK at *iotx-sdk-mqtt-java*.
2. Use IntelliJ IDEA or Eclipse to import the demo into a Maven project.
3. Log on to the Alibaba Cloud IoT Platform console, and select **Devices**. Click **View** next to the device to obtain the ProductKey, DeviceName, and DeviceSecret.
4. Modify and run the SimpleClient4IOT.java configuration file.

   a) Configure the parameters.

```
/** Obtain ProductKey, DeviceName, and DeviceSecret from the
console */
private static String productKey = "";
private static String deviceName = "";
private static String deviceSecret = "";
/** The topics used for testing */
private static String subTopic = "/"+productKey+"/"+deviceName+"/
get";
private static String pubTopic = "/"+productKey+"/"+deviceName+"/
pub";
```

   b) Connect to MQTT server.

```
// The client device ID. It can be specified using either MAC
 address or device serial number. It cannot be empty and must
contain no more than 32 characters
String clientId = InetAddress.getLocalHost().getHostAddress();
// Authenticate the device
Map params = new HashMap();
params.put("productKey", productKey); // Specifies the product key
 that the user registered in the console
params.put("deviceName", deviceName); // Specifies the device name
 that the user registered in the console
params.put("clientId", clientId);
String t = System.currentTimeMillis()+"";
params.put("timestamp", t);
```

```
// Specifies the MQTT server. If using the TLS protocol, begin the
 URL with SSL. If using the TCP protocol, begin the URL with TCP
String targetServer = "ssl://"+productKey+".iot-as-mqtt.cn-
shanghai.aliyuncs.com:1883";
// Client ID format:
String mqttclientId = clientId + "|securemode=2,signmethod=
hmacsha1,timestamp="+t+"|"; // Specifies the custom device
identifier. Valid characters include letters and numbers. For more
 information, see Establish MQTT over TCP connections (https://
help.aliyun.com/document_detail/30539.html?spm=a2c4g. 11186623.6.
592. R3LqNT)
String mqttUsername = deviceName+"&"+productKey;// Specifies
username format
String mqttPassword = SignUtil.sign(params, deviceSecret, "
hmacsha1");// Signature
// Code excerpt for connecting over MQTT
MqttClient sampleClient = new MqttClient(url, mqttclientId,
persistence);
MqttConnectOptions connOpts = new MqttConnectOptions();
connOpts.setMqttVersion(4);// MQTT 3.1.1
connOpts.setSocketFactory(socketFactory);
// Configure automatic reconnection
connOpts.setAutomaticReconnect(true);
// If set to true, then all offline messages are cleared. These
messages include all QoS 1 or QoS 2 messages that are not received
connOpts.setCleanSession(false);
connOpts.setUserName(mqttUsername);
connOpts.setPassword(mqttPassword.toCharArray());
connOpts.setKeepAliveInterval(80);// Specifies the heartbeat
interval. We recommend that you set it to 60 seconds or longer
sampleClient.connect(connOpts);
```

c) Send data.

```
String content = "The content of the data to be sent. It can be in
 any format";
MqttMessage message = new MqttMessage(content.getBytes("utf-8"));
message.setQos(0);// Message QoS. 0: At most once. 1: At least
once
sampleClient.publish(topic, message);// Send data to a specified
topic
```

d) Receive data.

```
// Subscribe to a specified topic. When new data is sent to the
topic, the specified callback method is called.
sampleClient.subscribe(topic, new IMqttMessageListener() {
@Override
public void messageArrived(String topic, MqttMessage message)
throws Exception {
// After the device successfully subscribes to a topic, when new
data is sent to the topic, the specified callback method is called
.
// If you subscribe to the same topic again, only the initial
subscription takes effect.
}
});
```

Note:

For more information about MQTT connection parameters, see *Establish MQTT over TCP connections*.

4 iOS SDK

This document describes how to connect your iOS devices to IoT Platform using the iOS SDK.

The SDK uses CocoaPods to manage dependencies. We recommend that you use CocoaPods version 1.1.1 or later.

The iOS SDK has the following features: establish connections using the message queuing telemetry transport (MQTT) protocol, maintain persistent connections, and send MQTT-based upstream and downstream requests.

Integrate the SDK

1. To integrate the SDK, add the following lines to the Podfile in your Xcode project directory.

```
source 'https://github.com/CocoaPods/Specs.git'
source 'https://github.com/aliyun/aliyun-specs.git'
target "necslinkdemo" do
pod 'IMLChannelCore'
pod 'OpenSSL'
end
```

2. Log on to the Alibaba Cloud IoT Platform console, and select **Devices**. Click **View** next to the device to obtain the ProductKey, DeviceName, and DeviceSecret.
3. To develop your code using the SDK, see the following instructions.

Initialize the SDK

Use the ProductKey, DeviceName, and DeviceSecret to establish a secure persistent connection with IoT Platform and configure your server address and port.

```
#import
#import
LKIoTConnectConfig * config = [LKIoTConnectConfig new];
config.productKey = @"your product key";
config.deviceName = @"your device name";
config.deviceSecret = @"your device secret";
config.server = @"www.yourserver.com";// If set to nil, then IoT
Platform is used as the server to connect to.
config.port = 1883,// Your server port. If the server value is set to
nil, then port setting can be skipped.
config.receiveOfflineMsg = NO;// If you want to receive messages when
the client is offline, set it to YES.
[[LKIoTExpress sharedInstance]startConnect:config connectListener:self
];
```

```
// If config.server is set to nil, then the China (Shanghai) node
is connected by default: ${yourproductKey}.iot-as-mqtt.cn-shanghai.
aliyuncs.com:1883`
```

Upstream request

The SDK encapsulates operations such as upstream publish, subscribe, and unsubscribe.

The upstream request can only be used after the SDK is initialized and a connection is establishe d.

```
/**
The remote procedure call (RPC) request API encapsulates the upstream
 request and downstream response of the business logic. The request
 messages containing your business data are encapsulated by the SDK
based on the Alink protocol. A request message resembles the following
:
{
"id":"msgId" // Message ID
"system": {
"version": "1.0", // Message version. Required. The current version is
 1.0
"time": "" // The time (in milliseconds) of the message. Required
},
"request": {
},
"params": {
}
}
@param topic The topic that is requested by RPC, depending on your
business logic. The complete topic is as follows:
/sys/${productKey}/${deviceName}/app/abc/cba
@param opts This is an optional parameter
Example, {"extraParam":{"method":"thing.topo.add"}}
This inserts "method":"thing.topo.add" into the final business data.
The parameter is on the same level as the params in the business data.
@param params The business data parameter.
@param responseHandler The response handler of your business server.
For more information, see LKExpressResponse
*/
-(void)invokeWithTopic:(NSString *)topic opts:(NSDictionary* _Nullable
)opts
params:(NSDictionary*)params respHandler:(LKExpressResponseHandler)
responseHandler;
/**
The data is upstreamed directly. It is not converted to the Alink
protocol.
@param topic The message topic
@param dat The data that is passed through
@param completeCallback The callback of data upstream result
*/
-(void)uploadData:(NSString *)topic data:(NSData *)dat complete:(
LKExpressOnUpstreamResult)completeCallback;
/**
No receipt is issued for upstream data. The SDK encapsulates the
business messages based on the Alink protocol.
@param topic The complete message topic
@param params The business parameter
@param completeCallback The callback of data upstream result
```

```
*/
-(void)publish:(NSString *) topic params:(NSDictionary *)params
complete:(LKExpressOnUpstreamResult)completeCallback;
/**
Subscribe to a topic
@param topic The topic of a subscribed message, depending on your
business logic. A complete topic section must be specified.
/sys/${productKey}/${deviceName}/app/abc/cba
@param completionHandler The callback when the subscription has
ended. If the error field is empty, the subscription is successful.
Otherwise, the subscription has failed.
*/
- (void)subscribe:(NSString *)topic complete: (void (^)(NSError *
_Nullable error))completionHandler;
/**
Unsubscribe from a topic
@param topic The topic of a subscribed message, depending on your
business logic. A complete topic section must be specified.
/sys/${productKey}/${deviceName}/app/abc/cba
@param completionHandler The callback when the subscription has
ended. If the error field is empty, the subscription is successful.
Otherwise, the subscription has failed.
*/
- (void)unsubscribe : (NSString *)topic complete: (void (^)(NSError *
_Nullable error))completionHandler;
```

The differences among the three upstream methods are as follows:

- `-(void)invokeWithTopic:(NSString )topic opts:(NSDictionary _Nullable )opts respHandler:(LKExpressResponseHandler)responseHandler;` : The following is a response model for business request. The server throws back a response in the responseHandler callback when a request is sent out.
- `uploadData:(NSString )topic data:(NSData )dat complete:(LKExpressOnUpstreamResult)completeCallback;`: This is a data passthrough method. The message is directly sent upstream to the cloud. No response is sent back.
- `-(void)publish:(NSString ) topic params:(NSDictionary )params complete:(LKExpressOnUpstreamResult)completeCallback;`: The data is sent upstream after it is encapsulated as business data based on the Alink protocol. For more information about the Alink protocol, see the API Reference.

An example of business requests and responses is as follows:

```

NSString *topic = @"/sys/${productKey}/${deviceName}/account/bind";
NSDictionary *params = @{
@"iotToken": token,
};
[[LKIoTExpress sharedInstance] invokeWithTopic:topic
opts:nil
params:params
respHandler:^(LKExpressResponseHandler * _Nonnull response) {
if (![ response successed]) {
```

```
NSLog(@"Business request failed");
}
}];
// ${productKey} refers to the ProductKey
// ${deviceName} refers to DeviceName
```

An example of calling the operation to subscribe to topic is as follows:

```

NSString *topic = @"/sys/${productKey}/${deviceName}/app/down/event";
[[LKIoTExpress sharedInstance] subscribe:topic complete: ^(NSError *
error) {
if (error ! = nil) {
NSLog(@"Business request failed");
}
}];
```

Downstream data listener

The following is the downstream message listener API pushed from the cloud once you subscribe to a topic.

```
@protocol LKExpressDownListener<NSObject>
-(void)onDownstream:(NSString *) topic data: (id _Nullable) data;///<
topic: The message topic. data: The message content. The data type of
the parameter can be either NSString or NSDictionary

-(BOOL)shouldHandle:(NSString *)topic;///<You can first filter the
 data before you use onDownstream:data: to push the data. If NO is
 returned, then the data is not pushed. If YES is returned, then
onDownstream:data: is used to push the data.
@end

[[LKIoTExpress sharedInstance] addDownstreamListener:LKExpressD
ownListenerTypeGw listener:(id<LKExpressDownListener>)downListener];
// The downListener in this SDK is a weak reference object. Therefore
, the caller needs to maintain its life cycle.
```

Recommendations

We recommend that you decouple ProductKey, DeviceName, and DeviceSecret from the client user account when you establish a channel connection. Your application uses only one set of ProductKey, DeviceName, and DeviceSecret to establish a connection with IoT Platform.

The client user account can be switched by rebinding the ProductKey, DeviceName, and DeviceSecret of the device with a different client user account.

Two different client user accounts can use the same channel, if ProductKey, DeviceName, and DeviceSecret are rebound. This means that the binding between the channel and the client user account can be changed dynamically.

5 HTTP/2 SDK

You can use the HTTP/2 protocol to establish communication between your devices and IoT Platform. The following is an HTTP/2 SDK development demo that you can use as reference to help you develop your own HTTP/2 SDK.

Prerequisites

In this demo, a Maven project is used. You must have installed Maven before you begin.

Procedure

1. Download the HTTP/2 SDK at the following link: *iot-http2-sdk-demo*.
2. Use IntelliJ IDEA or Eclipse to import the demo into a Maven project.
3. Obtain the device information (ProductKey, DeviceName, and DeviceSecret) on the IoT Platform console. For more information, see **User Guide**>**Create products and devices**.
4. Modify the H2Client.java configuration file.

    a. Configure the parameters.

```
//Obtain the ProductKey, DeviceName, and DeviceSecret of the
device from the IoT Platform console.
String productKey = "";
String deviceName = "";
String deviceSecret = "";

// The topics that are used to receive messages.
String subTopic = "/" + productKey + "/" + deviceName + "/get";
String pubTopic = "/" + productKey + "/" + deviceName + "/update";
```

    b. Connect to the HTTP/2 server to receive data.

```
// endPoint: https://${uid}.iot-as-http2.${region}.aliyuncs.com
String endPoint = "https://" + productKey + ".iot-as-http2.cn-
shanghai.aliyuncs.com";

// The unique identifier of the client device.
String clientId = InetAddress.getLocalHost().getHostAddress();

// Configure for device connection
Profile profile = Profile.getDeviceProfile(endPoint, productKey,
deviceName, deviceSecret, clientId);

//A value of true indicates that all offline messages will be
cleared. Offline messages refer to all messages that were not
received when messages were sent with QoS 1 or QoS 2.
profile.setCleanSession(false);

//Construct the client
MessageClient client = MessageClientFactory.messageClient(profile
);

// Receive data
```

```
client.connect(messageToken -> {
Message m = messageToken.getMessage();
System.out.println("receive message from " + m);
return MessageCallback.Action.CommitSuccess;
});
```

c. Subscribe to topics.

```
// Subscribe to topics. After the topic subscription is successful
, you can receive messages by using the callback interface when
the device is connected.
CompletableFuture subFuture = client.subscribe(subTopic);
System.out.println("sub result : " + subFuture.get());
```

d. Send data.

```
//Send messages.
MessageToken messageToken = client.publish(pubTopic, new Message("
hello iot".getBytes(), 0));
System.out.println("publish success, messageId: " + messageToken.
getPublishFuture().get().getMessageId());
```

5. Run the code after you have modified the configuration file.

Interface description

- Identity authentication interface

  When you connect devices to IoT Platform, you can use Profile to configure identity information of the devices. The interface parameters are as follows:

```
Profile profile = Profile.getDeviceProfile(endPoint, productKey,
deviceName, deviceSecret, clientId);
MessageClient client = MessageClientFactory.messageClient(profile);
client.connect(messageToken -> {
    Message m = messageToken.getMessage();
    System.out.println("receive message from " + m);
    return MessageCallback.Action.CommitSuccess;
});
```

Description of parameters in Profile.

Name	Type	Required	Description
endPoint	String	Yes	The endpoint address. The endpoint format is: https://${productKey}.iot-as-http2.${regionId}.aliyuncs.com. For example, https://al2sdABC1234.iot-as-http2.cn-shanghai.aliyuncs.com
productKey	String	Yes	The ID of the product to which the device belongs. You can get this information on the IoT Platform console.

Name	Type	Required	Description
deviceName	String	Yes	The name of the device. You can get this information on the IoT Platform console.
deviceSecret	String	Yes	The secret of the device. You can get this information on the IoT Platform console.
clientId	String	Yes	The unique identifier of the client device.
cleanSession	Boolean	No	Determines whether or not to clear the messages that were not received when they were sent with QoS 1 or QoS 2.
heartBeatInterval	Long	No	The interval of heartbeat, in milliseconds .
heartBeatTimeOut	Long	No	The timeout time of a heartbeat, in milliseconds.
multiConnection	Boolean	No	Determines whether or not to use multiple connections. If the device productKey and deviceName are used for device connection, set this parameter value as false.
callbackThreadCorePoolSize	Integer	No	The size of the core callback thread pool .
callbackThreadMaximumPoolSize	Integer	No	The maximum size of the callback thread pool.
callbackThreadBlockingQueueSize	Integer	No	The blocking queue of the callback thread.
authParams	Map	No	Custom authentication parameters.

- Connect to IoT Platform

  After the MessageClient value is received, you can connect a device to IoT Platform. IoT Platform will then authenticate the device identity and, if authentication is successful, the messages will be sent and received. When you are configuring the connection, you need to configure a message receiving interface that is able to handle messages without setting a callback interface. Then, the server can push messages that have been subscribed to the SDK . The configuration for the connection is as follows:

```
void connect(MessageCallback messageCallback);
```

- Message subscription

```
/**
* Subscribe to a topic
* @param topic                       topic
* @return completableFuture for subscribe result
*/

CompletableFuture subscribe(String topic);

/**
* Subscribe to a topic and define the callback topic
* @param topic                      topic
* @param messageCallback callback when message received on this
topic
* @return completableFuture for subscribe result
*/
CompletableFuture subscribe(String topic, MessageCallback messageCal
lback);

/**
* unsubscribe a topic
*
* @param topic                       topic
* @return completableFuture for unsubscribe result
*/
CompletableFuture unsubscribe(String topic);
```

- Receive messages

  When you set configurations for receiving messages, you must configure the callback interface MessageCallback. Additionally, you must set configurations for the connection and for topic subscription by including the interface MessageClient. The configuration for the message receiving interface is as follows:

```
/**
*
* @param messageToken  message token
* @return Action                   action after consuming
*/
Action consume(final MessageToken messageToken);
```

  When messages are received by using MessageToken.getMessage, this method will be called . Because the interfaces are called in the thread pool, note the following security issues. The returned values of this method will decide whether or not to return ACKs for messages sent with QoS 1 and QoS2.

Return value	Description
Action.CommitSuccess	ACK will be returned.
Action.CommitFailure	ACK will not be returned, and the message will be received later.
Action.CommitAckManually	ACK will not be returned automatically. Call MessageClient.ack() to return ACK manually.

- Publish messages

```
/**
* Publish a message to a specified topic
*
* @param topic                      topic
* @param message                 message entity
* @return completableFuture for publish result
*/
MessageToken publish(String topic, Message message);
```

6 General protocols

6.1 Overview

The Alibaba Cloud IoT Platform already supports MQTT, CoAP, HTTP and other common protocols, yet fire protection agreement GB/T 26875.3-2011, Modbus and JT808 is not supported, and in some specialized cases, devices may not be able to connect to IoT Platform. At this point, you need to use general protocol SDK to quickly build a bridge between your devices and platform to Alibaba Cloud IoT Platform, allowing two-way data communication.

General protocol SDK

The general protocol SDK is a protocol self-adaptive framework, using for high-efficiency bi-directional communication between your devices or platform to IoT Platform. The SDK architecture is shown below:

General protocol provides two SDKs: Core SDK and Server SDK.

- **General protocol core SDK**

  Core SDK abstracts abilities like session and configuration management. It acts like a net bridge between devices and IoT Platform and communicates with the Platform in representation of devices. This greatly simplifies the development of IoT Platform. Its main features include:

  - provides non-persistent session management capabilities.
  - provides configuration management capabilities based on configuration files.
  - provides connection management capabilities.
  - provides upstream communication capability.
  - provides downstream communication capabilities.
  - supports device authentication.

  If your devices are already connected to the internet and you want to build a bridge between IoT Platform and your devices or existing platform, choose core SDK.

- **General protocol server SDK**

  Server SDK provides device connection service on the basis of core SDK function. Its main features include:

- supports any protocol that is based on TCP/UDP.
- supports TLS/SSL encryption for transmission.
- supports horizontal expansion of the capacity of device connection.
- provides Netty-based communication service.
- provides automated and customizable device connection and management capability.

If you want to build the connection service from scratch, choose server SDK which provides socket for communication.

Development and deployment

Create products and devices in IoT console

Create products and devices in console. See *Create a product (Pro Edition)* for more information. Acquire the ProductKey, DeviceName and DeviceSecret of the net bridge device you've just created.

Note:
Net bridge is a virtual concept, and you can use the information of any device as device information of the net bridge.

SDK dependency

General protocol SDKs are currently only available in Java, and supports JDK 1.8 and later versions. Maven dependencies:

```
<! -- Core SDK -->
<dependency>
    <groupId>com.aliyun.openservices</groupId>
    <artifactId>iot-as-bridge-sdk-core</artifactId>
    <version>1.0.0</version>
</dependency>

<! -- Server SDK -->
<dependency>
    <groupId>com.aliyun.openservices</groupId>
    <artifactId>iot-as-bridge-sdk-server</artifactId>
    <version>1.0.0</version>
</dependency>
```

Develop SDK

*Develop Core SDK*and *Server SDK* briefly introduces the development process. For detailed implementation, refer to javadoc.

Deployment

The completed bridge connection service can be deployed on Alibaba Cloud using services like *ECS* and *SLB*, or deployed in local environment to guarantee communication security.

The whole process (if using Alibaba Cloud ECS to deploy) is shown below:

6.2 Develop Core SDK

You can integrate the IoT Platform bridge service with existing connection services or platforms that use the general protocol core SDK to allow devices or servers to quickly access Alibaba Cloud IoT Platform.

Prerequisites

For information about the concepts, functions, and Maven dependencies of the general protocol core SDK, see *Overview*.

Configuration management

The general protocol core SDK uses file-based configuration management by default. For information about customized configurations, see *Custom components > Configuration management*. The general protocol core SDK supports:

- Java Properties, JSON, and *HOCON* formats.
- Structured configuration to simplify maintenance.
- The override of file configurations with Java system properties, such as java -Dmyapp.foo.bar=10.
- Configuration file separation and nested references.

Table 6-1: application.conf

Net bridge is a virtual concept. You can use the **`productKey`**, **`deviceName`**, and **`deviceSecret`** of any device as the information of the net bridge.

Parameter	Required	Description
`productKey`	Yes	The product ID of the net bridge product.
`deviceName`	No	The device name of the net bridge device. The default value is the ECS instance MAC address.
`deviceSecret`	No	The device secret of the net bridge device.
`http2Endpoint`	Yes	HTTP/2 gateway service address.

Parameter	Required	Description
		The address format is `${UID}.iot-as-http2.${RegionId}.aliyuncs.com:443`. where: • `${UID}` indicates your account ID. To view your account ID, log on to the Alibaba Cloud console, hover your mouse over your account image, and click **Security Settings**. You are then directed to the **Account Management** page that displays your account ID. • `${RegionId}` indicates the region ID where your service is located. For example, if the region is Shanghai, the HTTP/2 gateway service address is `123456789.iot-as-http2.cn-shanghai.aliyuncs.com:443`. For information about RegionId expressions, see *Regions and zones*.
`authEndpoint`	Yes	Device authentication service address Device authentication service address: `https://iot-auth. ${RegionId}.aliyuncs.com/auth/bridge`. `${RegionId}` indicates the region ID where your service is located. For example, if the region is Shanghai, the device authentication service address is `https://iot-auth.cn-shanghai.aliyuncs.com/auth/bridge`. For information about RegionId expressions, see *Regions and zones*.
`popClientProfile`	Yes	Call APIs to configure the client. For details, see the *API client configuration*.

Table 6-2: API client configuration

Parameter	Required	Description
accessKey	Yes	The access key of the API caller.
accessSecret	Yes	The secret key of the API caller.
name	Yes	The region name of the API.
region	Yes	The region ID of the API.

Parameter	Required	Description
product	Yes	The name of the product. Set it to `Iot` if not specified.
endpoint	Yes	The endpoint of the API. Endpoint structure: `iot.${RegionId}.aliyuncs.com`. `${RegionId}` indicates the region ID of your service. For example, If the region is Shanghai, the endpoint is `iot.cn-shanghai.aliyuncs.com`. For information about RegionId expressions, see *Regions and zones*.

devices.conf

Configure the ProductKey, DeviceName, and DeviceSecret of the device. For information about customizing configuration files, see *Custom components > Configuration management*.

```
XXXX //  Original identifiers of the device
{
    "productKey": "123",
    deviceName: "",
    deviceSecret: ""
}
```

Interfaces

Initialization

`com.aliyun.iot.as.bridge.core.BridgeBootstrap` initializes the communication between the device and Alibaba Cloud IoT Platform. After the BridgeBootstrap instance is created, the *Basic configurations* component of the gateway will be initialized. For information about customizing configurations, see *Custom components > Configuration management*.

Complete the initialization using one of the following interfaces:

- `bootstrap()`: initialization without downstream messaging.
- `bootstrap(DownlinkChannelHandler handler)`: initialize using **`DownlinkChannelHandler`** specified by the developer.

Sample code:

```
BridgeBootstrap bootstrap = new BridgeBootstrap();
// Do not implement downstream messaging
bootstrap.bootstrap();
```

Connect devices to IoT Platform

Only devices that are online can establish a connection with or send connection requests to IoT Platform. There are two methods that can enable devices to get online: local session initialization and device authentication.

1. Session initialization

   The general protocol SDK provides non-persistent local session management. See *Custom components > Session management* for information on customization.

   Interfaces for creating new instances:

   - `com.aliyun.iot.as.bridge.core.model.Session.newInstance(String originalIdentity, Object channel)`
   - `com.aliyun.iot.as.bridge.core.model.Session.newInstance(String originalIdentity, Object channel, int heartBeatInterval)`
   - `com.aliyun.iot.as.bridge.core.model.Session.newInstance(String originalIdentity, Object channel, int heartBeatInterval, int heartBeatProbes)`

   `originalIdentity` indicates the unique device identifier and has the same function as SN in the original protocol. **`channel`** is the communication channel between devices and bridge service, and has the same function as a channel in Netty. **`heartBeatInterval`** and **`heartBeatProbes`** are used for heartbeat monitoring. The unit of heartBeatInterval is seconds. heartBeatProbes indicates the maximum number of undetected heartbeats that is allowed. If this number is exceeded, a heartbeat timeout event will be sent. To handle a timeout event, register `com.aliyun.iot.as.bridge.core.session.SessionListener`.

2. Authenticate devices

   After the initialization of local device session, use `com.aliyun.iot.as.bridge.core.handler.UplinkChannelHandler.doOnline(Session newSession, String originalIdentity, String... credentials)` to complete local device authentication and Alibaba Cloud IoT Platform online authentication. The device will then either be allowed to communicate or will be disconnected according to the authentication result. SDK provides online authentication for IoT Platform. By default, local authentication is disabled. If you need to set up local authentication, see *Customized components > Connection authentication*.

   Sample code:

```
UplinkChannelHandler uplinkHandler = new UplinkChannelHandler();
Session session = session. newinstance (device, Channel );
boolean success = uplinkHandler.doOnline(session, originalIdentity);
```

```
if (success) {
    // Successfully got online, and will accept communication
requests.
} else {
    // Failed to get online, and will reject communication requests
and disconnect (if connected).
}
```

Device Offline

When a device disconnects or detects that it needs to disconnect, a device offline operation must be initiated. Use `com.aliyun.iot.as.bridge.core.handler.UplinkChannelHandler.doOffline(String originalIdentity)` to bring a device offline.

Sample code:

```
UplinkChannelHandler uplinkHandler = new UplinkChannelHandler();
Uplinkhandler. dooffline (originalidentity );
```

Report Data

You can use `com.aliyun.iot.as.bridge.core.handler.UplinkChannelHandler` to report data to Alibaba Cloud IoT Platform. Data reporting involves three key steps: identify the device that is going to report data, locate the corresponding session for this device, and report data to IoT Platform. Use the following interfaces to report data.

Note:
Make sure that the data report has been managed and security issues have been handled.

- `CompletableFuture doPublishAsync(String originalIdentity, String topic, byte[] payload, int qos)`: send data asynchronously and return immediately. You can then obtain the sending result using future.
- `CompletableFuture doPublishAsync(String originalIdentity, ProtocolMessage protocolMsg)`: send data asynchronously and return immediately. You can then obtain the sending result using future.
- `boolean doPublish(String originalIdentity, ProtocolMessage protocolMsg, int timeout)`: send data asynchronously and wait for the response.
- `boolean doPublish(String originalIdentity, String topic, byte[] payload, int qos, int timeout)`: send data asynchronously and wait for the response.

Sample code:

```
UplinkChannelHandler uplinkHandler = new UplinkChannelHandler();
DeviceIdentity identity = ConfigFactory.getDeviceConfigManager().
getDeviviceIdentity(device);
```

```
if (identity == null) {
    // Devices are not mapped with those registered on IoT Platform,
and messages are dropped.
    return;
}
Session session = SessionManagerFactory.getInstance().getSession(
device);
if (session == null) {
    // The device is not online. You can either get the device online
or drop messages. Make sure devices are online before reporting data
to IoT Platform.
}
boolean success = uplinkHandler.doPublish(session, topic, payload, 0,
10);
if(success) {
    // Data is successfully reported to Alibaba Cloud IoT Platform.
} else {
    // Failed to report data to IoT Platform
}
```

Downstream Messaging

The general protocol SDK provides `com.aliyun.iot.as.bridge.core.handler.DownlinkChannelHandler` as the downstream data distribution processor. It supports unicast and broadcast (if the message sent from the cloud does not include specific device information).

Sample code:

```
public class SampleDownlinkHandler implements DownlinkChannelHandler {
    @Override
    public boolean pushToDevice(Session session, String topic, byte[]
payload) {
        // Process messages pushed to the device
    }

    @Override
    public boolean broadcast(String topic, byte[] payload) {
        // Process broadcast
    }
}
```

Custom components

You can customize the device connection authentication, session management, and configurat ion management components. You must complete the initialization and substitution of those components before calling BridgeBootstrap intialization.

Connection authentication

To customize the device connection authentication, implement `com.aliyun.iot.as.bridge.core.auth.AuthProvider` and then, before initializing BridgeBootstrapcall, call `com.aliyun.iot.as.bridge.core.auth.AuthProviderFactory.init(AuthProvider customizedProvider)` to replace the original authentication component with the customized component.

Session management

To customize the session management, implement `com.aliyun.iot.as.bridge.core.session.SessionManager` and then, before initializing BridgeBootstrapcall, call `com.aliyun.iot.as.bridge.core.session.SessionManagerFactory.init(SessionManager <?> customizedSessionManager)` to replace the original session management component with the customized component.

Configuration management

To customize the configuration management, implement `com.aliyun.iot.as.bridge.core.config.DeviceConfigManager` and `com.aliyun.iot.as.bridge.config.BridgeConfigManager`. Then, before initializing BridgeBootstrapcall, call `com.aliyun.iot.as.bridge.core.config.ConfigFactory.init(BridgeConfigManager bcm, DeviceConfigManager dcm)` to replace the original configuration management component with the customized component. If the parameters are left empty, the general protocol SDK default values will be used.

6.3 Server SDK

6.3.1 Server SDK

You can use the general protocol server SDK to quickly build a bridge service that connects your existing devices or services to Alibaba Cloud IoT Platform.

Prerequisites

Refer to *Overview* for concepts, functions and Maven dependencies of the general protocol server SDK.

Configuration Management

The general protocol server SDK uses file-based configuration management by default. Add the **socketServer** parameter in *application.conf*, and set the socket server related parameters listed in the following table. For customized configuration, refer to *Custom Components > Configuration Management* .

Parameter	Description	Required
address	The connection listening address. Supports network names like eth1, and IPv4 addresses with 10.30 prefix.	No

Parameter	Description	Required
backlog	The number of backlogs for TCP connection.	No
ports:	Connection listening port. The default port is 9123. You can specify multiple ports.	No
listenType	The type of socket server. Can be `udp` or `tcp`. The default value is `tcp` . Case insensitive.	No
broadcastEnabled	Whether UDP broadcasts are supported. Used when **`listenType`** is `udp`. The default value is true.	No
unsecured	Whether unencrypted TCP connection is supported. Used when listenType is tcp.	No
keyPassword	The certificate store password. Used when listenType is tcp.	No **Note:** Effective when keyPassword, keyStoreFile, and keyStoreType are all configured. Otherwise, keyPassword does not need to be configured.
keyStoreFile	The file address of the certificate store. Used when listenType is tcp.	No
keyStoreType	The type of certificate store. Used when listenType is tcp.	No

Interfaces

The following two articles assume that you have a basic understanding of Netty-based development. Refer to *Netty Documentation* for more details on Netty-based development.

- *Interfaces for TCP*
- *Interfaces for UDP*

Custom Components

Besides file-based configuration, you can also set your own customized configurations.

If you want to customize configurations, implement com.aliyun.iot.as.bridge.server.config.BridgeServerConfigManager first and call com.aliyun.iot.as.bridge.server.config.ServerConfigFactory.init(BridgeServerConfigManager bcm) to replace default configuration management components with customized ones, and then initialize these components. Then, connect the net bridge products to the Internet.

6.3.2 Interfaces for TCP

You can build an access service which uses TCP transmission protocol and bridge it to Alibaba Cloud IoT Platform using the interfaces of the general protocol SDK for TCP.

Bootstrap

com.aliyun.iot.as.bridge.server.BridgeServerBootstrap is the bootstrap class for booting socket server and bridge service. After a new BridgeServerBootstrap is created, components based on configuration files will be initialized.

Example:

```
BridgeServerBootstrap bootstrap = new BridgeServerBootstrap(new
TcpDecoderFactory() {
    @Override
    public ByteToMessageDecoder newInstance() {
        // Return decoder
    }
}, new TcpEncoderFactory() {
    @Override
    public MessageToByteEncoder<? > newInstance() {
// Return encoder
    }
}, new TcpBasedProtocolAdaptorHandlerFactory() {
@ Override
    public CustomizedTcpBasedProtocolHandler newInstance() {
// Return protocol adapter
    }
}, new DefaultDownlinkChannelHandler());
try {
    bootstrap.start();
} catch (BootException | ConfigException e) {
// Process boot exception
}
```

Instantiation of TCP type BridgeServerBootstrap

- com.aliyun.iot.as.bridge.server.channel.factory.TcpDecoderFactory: Create a new decoder instance using factory method to decode upload data. Thread is secure. Can be null.
- com.aliyun.iot.as.bridge.server.channel.factory.TcpEncoderFactory: Create a new encoder instance using factory method to encode downstream data to adapt to TCP protocol. Thread is secure. Can be null.

- com.aliyun.iot.as.bridge.server.channel.factory.TcpBasedProtocolAdaptorHandlerFactory: Create a new protocol adapter instance using factory method to adapt decoded data so they can be uploaded to the cloud. Thread is secure. Cannot be null.
- com.aliyun.iot.as.bridge.core.handler.DownlinkChannelHandler: Distributor for downstream data. Supports unicast and broadcast. Unicast forwards data directly to the device by default . Broadcast settings must be customized by developers. Can be null. Null indicates that downstream data is not allowed.

Start socket server

After the creation of BridgeServerBootstrap, call com.aliyun.iot.as.bridge.server.BridgeServ erBootstrap.start() to start the socket server.

Protocol decoding

The component for protocol decoding derives from io.netty.handler.codec.ByteToMessageDecoder. Refer to *ByteToMessageDecoder Documentat ion* for details.

Example:

```
public class SampleDecoder extends ByteToMessageDecoder {
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<
Object> out) throws Exception {
// The decoding protocol
    }
}
```

Protocol encoding

The component for protocol encoding derives from io.netty.handler.codec.MessageToByteEncoder<I>. Refer to *MessageToByteEncoder Documentat ion* for details.

Example:

```
public class SampleEncoder extends MessageToByteEncoder<String>{
    @Override
    protected void encode(ChannelHandlerContext ctx, String msg,
ByteBuf out) throws Exception {
// Protocol encoding
    }
}
```

Protocol adapter

To reduce cost and improve the efficiency of development, the general protocol server SDK provides protocol adapters with extensible and customizable basic capability class com.aliyun.iot.as.bridge.server.channel.CustomizedTcpBasedProtocolHandler. It encapsulates details to access Alibaba Cloud IoT Platform, so you can focus on protocol related business. The protocol adapter derives from this class.

Device Online

Only online devices can establish a connection with or send connection requests to IoT Platform. There are two steps for devices to get online: local session initialization and device authentication.

1. Session Initialization

   Refer to *Core SDK develop > Device Online > Session Initialization*
2. Device Authentication

   After local session initialization, call doOnline(ChannelHandlerContext ctx, Session newSession, String originalIdentity, String... credentials) to complete local device authentication and Alibaba Cloud IoT Platform online authentication. The device can communicate with IoT Platform if authentication succeeds, and will be disconnect from IoT Platform if authentication fails.

   Example:

   ```
   Session session = Session.newInstance(device, channel);
   boolean success = doOnline(session, originalIdentity);
   if (success) {
       // Successfully got online, and will accept communication
   requests.
   } else {
       // Failed to get online, will reject communication requests and
   disconnect (if connected).
   }
   ```

Device Offline

When the device is disconnected or detects that it needs to be disconnected, the device offline action should be initiated. Using server SDK, devices will automatically get offline when they are disconnected, so you can focus on other tasks. Call doOffline(Session session) to bring devices offline.

Report Data

The protocol adapter needs to use override channelRead(ChannelHandlerContext ctx, Object msg). It is the entrance for all devices to report data. Object msg is the data output from the decoder.

There are three steps for data reporting: identify the device that is going to report data, find the corresponding session for this device, and then report data to IoT Platform. Use the following interfaces to report data:

- CompletableFuture doPublishAsync(Session session, String topic, byte[] payload, int qos): send data asynchronously and return immediately. Developers obtain the sending result using future.
- CompletableFuture doPublishAsync(Session session, ProtocolMessage protocolMsg): send data asynchronously and return immediately. Developers obtain the sending result using future.
- boolean doPublish(Session session, ProtocolMessage protocolMsg, int timeout): send data asynchronously and wait for the response.
- boolean doPublish(Session session, String topic, byte[] payload, int qos, int timeout): send data asynchronously and wait for the response.

Example:

```
DeviceIdentity identity = ConfigFactory.getDeviceConfigManager().
getDeviviceIdentity(device);
if (identity == null) {
    // Devices are not mapped with those registered on IoT Platform.
Messages are dropped.
    return;
}
Session session = SessionManagerFactory.getInstance().getSession(
device);
if (session == null) {
    // The device is not online. Please get online or drop messages.
Make sure devices are online before reporting data to IoT Platform.
}
boolean success = doPublish(session, topic, payload, 0, 10);
if(success) {
    // Data is successfully reported to Alibaba Cloud IoT Platform.
} else {
    // Failed to report data to IoT Platform
}
```

Downstream Messaging

Refer to *Core SDK development > Downstream Messaging* for details.

The SDK provides com.aliyun.iot.as.bridge.core.handler.DefaultDownlinkChannelHandler as the downstream data distributor. It supports unicast and broadcast. Unicast forwards data from the cloud directly to the device by default, and broadcast requires developers to customize specific implementations. Customization can be realized by deriving subclass.

Example:

```
import io.netty.channel.Channel;
import Io. netty. Channel. channelfuture;
...

public class SampleDownlinkChannelHandler implements DownlinkCh
annelHandler {
    @Override
    public boolean pushToDevice(Session session, String topic, byte[]
payload) {
        // Obtain communication channel from device's corresponding
session.
        Channel channel = (Channel) session.getChannel().get();
        if (channel ! = null && channel.isWritable()) {
            String body = new String(payload, StandardCharsets.UTF_8);
            // Send downstream data to devices
            ChannelFuture future = channel.pipeline().writeAndFlush(
body);
            future.addListener(ChannelFutureListener.FIRE_EXCEP
TION_ON_FAILURE);
            return true;
        }
        return false;
    }

    @Override
    public boolean broadcast(String topic, byte[] payload) {
        throw new RuntimeException("not implemented");
    }
}
```

6.3.3 Interfaces for UDP

You can build an access service which uses UDP transmission protocol and bridge it to Alibaba Cloud IoT Platform using the interfaces of the general protocol SDK for UDP.

Bootstrap

com.aliyun.iot.as.bridge.server.BridgeServerBootstrap is the bootstrap class for booting socket server and bridge service. After a new BridgeServerBootstrap is created, components based on configuration files will be initialized.

Example:

```
BridgeServerBootstrap bootstrap = new BridgeServerBootstrap(new
UdpDecoderFactory() {
    @Override
 public MessageToMessageDecoder newInstance() {
// Return decoder
    }
}, new UdpEncoderFactory() {
    @Override
public MessageToMessageEncoder<?> newInstance() {
// Return encoder
    }
}, new UdpBasedProtocolAdaptorHandlerFactory() {
    @Override
public CustomizedUdpBasedProtocolHandler newInstance() {
// Return protocol adapter
    }
});
try {
    bootstrap.start();
} catch (BootException | ConfigException e) {
// Process boot exception
}
```

Instantiation of UDP type BridgeServerBootstrap

- com.aliyun.iot.as.bridge.server.channel.factory.UdpDecoderFactory: Create a new decoder instance using the factory method to decode upload data. Thread is secure. Can be null.
- com.aliyun.iot.as.bridge.server.channel.factory.UdpEncoderFactory: Create a new encoder instance using the factory method to encode downstream data to adapt to UDP protocol. Thread is secure. Can be null.
- com.aliyun.iot.as.bridge.server.channel.factory.UdpBasedProtocolAdaptorHandlerFactory: Create a new protocol adapter instance using the factory method to adapt decoded data so they can be uploaded to the cloud. Thread is secure. Cannot be null.

Start socket server

After the creation of BridgeServerBootstrap, call com.aliyun.iot.as.bridge.server.BridgeServerBootstrap.start() to start the socket server.

Protocol decoding

The component for protocol decoding derives from io.netty.handler.codec.MessageToMessageDecoder<I>. Refer to *MessageToMessageDecoder Documentation* for details.

Example:

```
public class SampleDecoder extends  MessageToMessageDecoder<DatagramPa
cket> {
    @Override
protected void decode(ChannelHandlerContext ctx, DatagramPacket in,
List<Object> out) throws Exception {
// The decoding protocol
    }
}
```

Protocol encoding

The component for protocol encoding derives from io.netty.handler.codec.MessageToMessageEncoder<I>. Refer to *MessageToMessageEncoder Documentation* for details.

Example:

```
public class SampleEncoder extends MessageToMessageEncoder<T>{
    @Override
    protected void encode(ChannelHandlerContext ctx, T msg, ByteBuf
out) throws Exception {
// Protocol encoding
    }
}
```

Protocol adapter

To reduce cost and improve the efficiency of development, the general protocol server SDK provides protocol adapters with extensible and customizable basic capability class com.aliyun.iot.as.bridge.server.channel.CustomizedUdpBasedProtocolHandler. It encapsulates details to access Alibaba Cloud IoT Platform, so you can focus on other business. The protocol adapter derives from this class.

Device Online

Only online devices can establish a connection with or send connection requests to IoT Platform. There are two steps for devices to get online: local session initialization and device authentication.

1. Session Initialization

   Refer to *Core SDK develop > Device Online > Session Initialization* for details.

2. Device Authentication

   After local session initialization, call doOnline(Session newSession, String originalIdentity, String... credentials) or doOnline(String originalIdentity, String... credentials) to complete local device authentication and Alibaba Cloud IoT Platform online authentication. The device can communicate with IoT Platform if authentication succeeds, and will be disconnect from IoT Platform if authentication fails.

   Example:

   ```
   Session session = Session.newInstance(device, channel);
   boolean success = doOnline(session, originalIdentity);
   if (success) {
   ```

```
    // Successfully got online, and will accept communication
requests.
} else {
    // Failed to get online, and will reject communication requests
and disconnect (if connected).
}
```

Device Offline

When the device is disconnected or detects that it needs to be disconnected, the device offline action should be initiated. Using server SDK, devices will automatically get offline when they are disconnected, so you can focus on other tasks. Call doOffline(Session session) to bring devices offline.

Report Data

The protocol adapter needs to use override channelRead(ChannelHandlerContext ctx, Object msg). It is the entrance for all devices to report data. Object msg is the data output from the decoder.

There are three steps for data reporting: identify the device that is going to report data, find the corresponding session for this device, and then report data to IoT Platform. Use the following interfaces to report data:

- CompletableFuture doPublishAsync(String originalIdentity, String topic, byte[] payload, int qos): send data asynchronously and return immediately. Developers obtain the sending result using future.
- CompletableFuture doPublishAsync(String originalIdentity, ProtocolMessage protocolMsg): send data asynchronously and return immediately. Developers obtain the sending result using future.
- boolean doPublish(String originalIdentity, ProtocolMessage protocolMsg, int timeout): send data asynchronously and wait for the response.
- boolean doPublish(String originalIdentity, String topic, byte[] payload, int qos, int timeout): send data asynchronously and wait for the response.

Example:

```
DeviceIdentity identity = ConfigFactory.getDeviceConfigManager().
getDeviviceIdentity(device);
if (identity == null) {
    // Devices are not mapped with those registered on IoT Platform.
Messages are dropped.
    return;
}
Session session = SessionManagerFactory.getInstance().getSession(
device);
if (session == null) {
    // The device is not online. Please get online or drop messages.
Make sure devices are online before reporting data to IoT Platform.
}
boolean success = doPublish(session, topic, payload, 0, 10);
if(success) {
    // Data is successfully reported to Alibaba Cloud IoT Platform.
} else {
    // Failed to report data to IoT Platform
}
```

Downstream Messaging

Not supported yet.

7 Alink Protocol

IoT Platform provides device SDKs for you to configure devices. These device SDKs already encapsulate protocols for data exchange between devices and IoT Platform. However, in some cases, the device SDKs provided by IoT Platform cannot meet your requirements because of the complexity of the embedded system. This topic describes how to encapsulate data and establish connections from devices to IoT Platform using Alink protocol. Alink protocol is a data exchange standard for IoT development. Data are in JSON format.

Connection process

As shown in the following figure, devices can be connected to IoT Platform as directly connected devices or sub-devices. The connection process includes these key steps: register the device, establish a connection, and report data.

Directly connected devices can be connected to IoT Platform by using the following methods:

- If *Unique-certificate-per-device authentication* is enabled, install the three key fields (ProductKey, DeviceName, and DeviceSecret) into a device in advance, register the device with IoT Platform, connect the device to IoT Platform, and report data to IoT Platform.
- If dynamic registration based on *Unique-certificate-per-product authentication* is enabled, install the product certificate (ProductKey and ProductSecret) on a device, register the device with IoT Platform, connect the device to IoT Platform, and report data to IoT Platform.

The gateway starts the connection process for sub-devices. Sub-devices can be connected to IoT Platform by using the following methods:

- If *Unique-certificate-per-device authentication* is enabled, install the ProductKey, DeviceName, and DeviceSecret on a sub-device. The sub-device sends these three key fields to the gateway. The gateway adds the topological relationship and sends the data of the sub-device through the gateway connection.
- If dynamic registration is enabled, install ProductKey on a sub-device in advance. The sub-device sends the ProductKey and DeviceName to the gateway. The gateway forwards the ProductKey and DeviceName to IoT Platform. IoT Platform verifies the received DeviceName and sends a DeviceSecret to the sub-device. The sub-device sends the obtained ProductKey, DeviceName, and DeviceSecret to the gateway. The gateway adds the topological relationship and sends data to IoT Platform through the gateway connection.

Device identity registration

The following methods are available for identity registration:

- Unique certificate per device: Obtain the ProductKey, DeviceName, and DeviceSecret of a device on IoT Platform and use them as the unique identifier. Install these three key fields on the firmware of the device. After the device is connected to IoT Platform, the device starts to communicate with IoT Platform.
- Dynamic registration: You can perform dynamic registration based on unique-certificate-per-product authentication for directly connected devices and perform dynamic registration for sub-devices.
  - To dynamically register a directly connected device based on unique-certificate-per-product authentication, follow these steps:
    1. In the IoT Platform console, pre-register the device and obtain the ProductKey and ProductSecret. When you pre-register the device, use device information that can be directly read from the device as the DeviceName, such as the MAC address or SN.
    2. Enable dynamic registration in the console.
    3. Install the product certificate on the device firmware.
    4. The device authenticates to IoT Platform. If the device passes authentication, IoT Platform assigns a DeviceSecret to the device.
    5. The device uses the ProductKey, DeviceName, and DeviceSecret to establish a connection to IoT Platform.
  - To dynamically register a sub-device, follow these steps:
    1. In the IoT Platform console, pre-register the sub-device and obtain the ProductKey. When you pre-register the sub-device, use device information that can be read directly from the sub-device as the DeviceName, such as the MAC address and SN.
    2. Enable dynamic registration in the console.
    3. Install the ProductKey on the firmware of the sub-device or on the gateway.
    4. The gateway authenticates to IoT Platform on behalf of the sub-device.

Dynamically register a sub-device

Upstream

- Topic: /sys/{productKey}/{deviceName}/thing/sub/register
- Reply topic: /sys/{productKey}/{deviceName}/thing/sub/register_reply

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": [
    {
      "deviceName": "deviceName1234",
      "productKey": "1234556554"
    }
  ],
  "method": "thing.sub.register"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": [
    {
      "iotId": "12344",
      "productKey": "1234556554",
      "deviceName": "deviceName1234",
      "deviceSecret": "xxxxxx"
    }
  ]
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	List	Parameters used for dynamic registration.
deviceName	String	Name of the sub-device.
productKey	String	ProductKey of the sub-device.
iotId	String	Unique identifier of the sub-device.
deviceSecret	String	DeviceSecret key.
method	String	Request method.
code	Integer	Result code.

Error messages

Error code	Message	Description
460	request parameter error	The request parameters are incorrect.
6402	topo relation cannot add by self	A device cannot be added to itself as a sub-device.
401	request auth error	Signature verification has failed.

Dynamically register a directly connected device based on unique-certificate-per-product authentication

Directly connected devices send HTTP requests to perform dynamic register. Make sure that you have enabled dynamic registration based on unique certificate per product in the console.

- URL template: https://iot-auth.cn-shanghai.aliyuncs.com/auth/register/device
- HTTP method: POST

Request message

```
POST /auth/register/device HTTP/1.1
Host: iot-auth.cn-shanghai.aliyuncs.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 123
productKey=1234556554&deviceName=deviceName1234&random=567345&sign=
adfv123hdfdh&signMethod=HmacMD5
```

Response message

```
{
  "code": 200,
  "data": {
    "productKey": "1234556554",
    "deviceName": "deviceName1234",
    "deviceSecret": "adsfweafdsf"
  },
  "message": "success"
}
```

Parameter description

Parameters	Type	Description
productKey	String	Unique identifier of the product .
deviceName	String	Device name.
random	String	Random number.

Parameters	Type	Description
sign	String	Signature.
signMethod	String	Signing method. The supported methods are hmacmd5, hmacsha1, and hmacsha256.
code	Integer	Result code.
deviceSecret	String	DeviceSecret key.

Sign the parameters

All parameters reported to IoT Platform will be signed except **`sign`** and **`signMethod`**. Sort the signing parameters in alphabetical order, and splice the parameters and values without any splicing symbols. Then, sign the parameters by using the algorithm specified by **`signMethod`**.

```
sign = hmac_sha1(productSecret, deviceNamedeviceName1234productKey123455
6554random123)
```

Add topological relationships

After a sub-device has registered with IoT Platform, the gateway reports the topological relationship of *Gateways and sub-devices* to IoT Platform before the sub-device connects to IoT Platform. IoT Platform verifies the identity and the topological relationship during connection. If the verification is successful, IoT Platform establishes a logical connection with the sub-device and associates the logical connection with the physical connection of the gateway. The sub-device uses the same protocols as a directly connected device for data upload and download. Gateway information is not required to be included in the protocols.

After you delete the topological relationship of the sub-device from IoT Platform, the sub-device can no longer connect to IoT Platform through the gateway. IoT Platform will fail the authentication because the topological relationship does not exist.

Add topological relationships of sub-devices

Upstream

- Topic: /sys/{productKey}/{deviceName}/thing/topo/add
- Reply topic: /sys/{productKey}/{deviceName}/thing/topo/add_reply

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": [
    {
      "deviceName": "deviceName1234",
      "productKey": "1234556554",
      "sign": "xxxxxx",
      "signmethod": "hmacSha1",
      "timestamp": "1524448722000",
      "clientId": "xxxxxx"
    }
  ],
  "method": "thing.topo.add"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	List	Request parameters.
deviceName	String	Device name. The value is the name of a sub-device.
productKey	String	ProductKey. The value is the name of a sub-device.
sign	String	Signature.
signmethod	String	Signing method. The supported methods are hmacSha1, hmacSha256, hmacMd5, and Sha256.
timestamp	String	Timestamp.
clientId	String	Identifier of a sub-device. This parameter is optional and may have the same value as ProductKey or DeviceName.

Parameters	Type	Description
code	Integer	Result code. A value of 200 indicates the request is successful.

Signature algorithm

Note:

IoT Platform supports common signature algorithms.

1. All parameters reported to IoT Platform will be signed except **`sign`** and **`signMethod`**. Sort the signing parameters in alphabetical order, and splice the parameters and values without any splicing symbols. Sign the signing parameters by using the algorithm specified by the signing method.
2. For example, sign the parameters in **`params`** in the request as follows:
3. `sign=hmac_md5(deviceSecret, clientId123deviceNametestproductKey123timestamp1524448722000)`

Error messages

Error code	Message	Description
460	request parameter error	The request parameters are incorrect.
6402	topo relation cannot add by self	A device cannot be added to itself as a sub-device.
401	request auth error	Signature verification has failed.

Delete topological relationships of sub-devices

A gateway can publish a message to this topic to request IoT Platform to delete the topological relationship between the gateway and a sub-device.

Upstream

- Topic: /sys/{productKey}/{deviceName}/thing/topo/delete
- Reply topic: /sys/{productKey}/{deviceName}/thing/topo/delete_reply

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": [
    {
      "deviceName": "deviceName1234",
      "productKey": "1234556554"
    }
  ],
  "method": "thing.topo.delete"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	List	Request parameters.
deviceName	String	DeviceName. This is the name of a sub-device.
productKey	String	ProductKey. This is the ProductKey of a sub-device.
method	String	Request method.
code	Integer	Result code. A value of 200 indicates the request is successful.

Error messages

Error code	Message	Description
460	request parameter error	The request parameters are incorrect.
6100	device not found	The device does not exist.

Obtain topological relationships of sub-devices

Upstream

- Topic: /sys/{productKey}/{deviceName}/thing/topo/get
- Reply topic: /sys/{productKey}/{deviceName}/thing/topo/get_reply

A gateway can publish a message to this topic to obtain the topological relationships between the gateway and its connected sub-devices.

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": {},
  "method": "thing.topo.get"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": [
    {
      "deviceName": "deviceName1234",
      "productKey": "1234556554"
    }
  ]
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	Object	Request parameters. This can be left empty.
method	String	Request method.
deviceName	String	Name of the sub-device.
productKey	String	ProductKey of the sub-device.
code	Integer	Result code. A value of 200 indicates the request is successful.

Error messages

Error code	Message	Description
460	request parameter error	The request parameters are incorrect.

Report new sub-devices

Upstream

- Topic: /sys/{productKey}/{deviceName}/thing/list/found
- Reply topic: /sys/{productKey}/{deviceName}/thing/list/found_reply. In some scenarios, the gateway can discover new sub-devices. The gateway reports information of a sub-device to IoT Platform. IoT Platform forwards sub-device information to third-party applications, and the third-party applications choose the sub-devices to connect to the gateway.

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": [
    {
      "deviceName": "deviceName1234",
      "productKey": "1234556554"
    }
  ],
  "method": "thing.list.found"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data":{}
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	Object	Request parameters. This can be left empty.

Parameters	Type	Description
method	String	Request method.
deviceName	String	Name of the sub-device.
productKey	String	ProductKey of the sub-device.
code	Integer	Result code. A value of 200 indicates the request is successful.

Error messages

Error code	Message	Description
460	request parameter error	The request parameters are incorrect.
6250	product not found	The sub-device does not exist.
6280	devicename not meet specs	The name of the sub-device is invalid. The device name must be 4 to 32 characters in length and can contain letters, numbers, hyphens (-), underscores (_), at signs (@), periods (.), and colons (:).

Notify the gateway to add topological relationships of the connected sub-devices

Downstream

- Topic: /sys/{productKey}/{deviceName}/thing/topo/add/notify
- Reply topic: /sys/{productKey}/{deviceName}/thing/topo/add/notify_reply

IoT Platform publishes a message to this topic to notify a gateway to add topological relationships of the connected sub-devices. You can use this topic together with the topic that reports new sub-devices to IoT Platform. IoT Platform can subscribe to a data exchange topic to receive the response from the gateway. The data exchange topic is `/{productKey}/{deviceName}/thing/downlink/reply/message`.

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": [
    {
      "deviceName": "deviceName1234",
      "productKey": "1234556554"
    }
  ],
  "method": "thing.topo.add.notify"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	Object	Request parameters. This can be left empty.
method	String	Request method.
deviceName	String	Name of the sub-device.
productKey	String	ProductKey of the sub-device.
code	Integer	Result code. A value of 200 indicates the request is successful.

Connect devices to IoT Platform

Make sure that a directly connected device has been registered with IoT Platform before connecting to IoT Platform.

Make sure that a sub-device has been registered with IoT Platform before connecting to IoT Platform. In addition, you also need to make sure that the topological relationship with the gateway has been added to the gateway. IoT Platform will verify the identity of the sub-device according to the topological relationship to identify whether the sub-device can use the gateway connection.

Connect sub-devices to IoT Platform

Upstream

- Topic: /ext/session/{productKey}/{deviceName}/combine/login
- Reply topic: /ext/session/{productKey}/{deviceName}/combine/login_reply

Request message

```
{
  "id": "123",
  "params": {
    "productKey": "123",
    "deviceName": "test",
    "clientId": "123",
    "timestamp": "123",
    "signMethod": "hmacmd5",
    "sign": "xxxxxx",
    "cleanSession": "true"
  }
}
```

Response message

```
{
  "id":"123",
  "code":200,
  "message":"success"
  "data":""
}
```

Sign the parameters

1. All parameters reported to IoT Platform will be signed except **`sign`** and **`signMethod`**. Sort the signing parameters in alphabetical order, and splice the parameters and values without any splicing symbols. Then, sign the parameters by using the algorithm specified by **`signMethod`**.
2. `sign= hmac_md5(deviceSecret, cleanSessiontrueclientId123deviceNametestproductKey123timestamp123)`

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
params	List	Input parameters of the request.
deviceName	String	DeviceName of a sub-device.
productKey	String	ProductKey of a sub-device.

Parameters	Type	Description
sign	String	Signature of a sub-device. Sub-devices use the same signature rules as the gateway.
signmethod	String	Signing method. The supported methods are hmacSha1, hmacSha256, hmacMd5, and Sha256.
timestamp	String	Timestamp.
clientId	String	Identifier of a device client. This parameter can have the same value as the ProductKey or DeviceName parameter.
cleanSession	String	If the value is true, this indicates to clear offline information for all sub-devices , which is information that has not been received by QoS 1.
code	Integer	Result code. A value of 200 indicates the request is successful.
message	String	Result code.
data	String	Additional information in the response, in JSON format.

Note:

A gateway can accommodate a maximum of 200 concurrent online sub-devices. When the maximum number is reached, the gateway rejects any connection requests.

Error messages

Error code	Message	Description
460	request parameter error	The request parameters are incorrect.
429	rate limit, too many subDeviceOnline msg in one minute	The authentication requests from the device are throttled because the device requests

Error code	Message	Description
		authentication to IoT Platform too frequently.
428	too many subdevices under gateway	Too many sub-devices connect to the gateway at the same time.
6401	topo relation not exist	The topological relationship between the gateway and the sub-device does not exist.
6100	device not found	The sub-device does not exist.
521	device deleted	The sub-device has been deleted.
522	device forbidden	The sub-device has been disabled.
6287	invalid sign	The password or signature of the sub-device is incorrect.

Disconnect sub-devices from IoT Platform

Upstream

- Topic: /ext/session/{productKey}/{deviceName}/combine/logout
- Reply topic: /ext/session/{productKey}/{deviceName}/combine/logout_reply

Request message

```
{
  "id": 123,
  "params": {
    "productKey": "xxxxx",
    "deviceName": "xxxxx"
  }
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "message": "success",
  "data": ""
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
params	List	Input parameters of the request.
deviceName	String	DeviceName of a sub-device.
productKey	String	ProductKey of a sub-device.
code	Integer	Result code. A value of 200 indicates the request is successful.
message	String	Result code.
data	String	Additional information in the response, in JSON format.

Error messages

Error code	Message	Description
460	request parameter error	The request parameters are incorrect.
520	device no session	The sub-device session does not exist.

For information about sub-device connections, see *Connect sub-devices to IoT Platform*. For information about error codes, see *Error codes*.

Device property, event, and service protocols

A device sends data to IoT Platform either in standard mode or in passthrough mode.

1. If passthrough mode is used, the device sends raw data such as a binary data stream to IoT Platform. IoT Platform runs the script you have submitted to convert the raw data to a standard format.
2. If standard mode is used, the device generates data in the standard format and then sends the data to IoT Platform. For information about standard formats, see the requests and responses in this topic.

Report device properties

Note:

Set the parameters according to the output and input parameters in the TSL model.

Upstream (passthrough)

- Topic: /sys/{productKey}/{deviceName}/thing/model/up_raw
- Reply topic: /sys/{productKey}/{deviceName}/thing/model/up_raw_reply

Upstream (non-passthrough)

- Topic: /sys/{productKey}/{deviceName}/thing/event/property/post
- Reply topic: /sys/{productKey}/{deviceName}/thing/model/up_raw_reply

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": {
    "Power": {
      "value": "on",
      "time": 1524448722000
    },
    "WF": {
      "value": 23.6,
      "time": 1524448722000
    }
  },
  "method": "thing.event.property.post"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	Object	Request parameters.
method	String	Request method.

Parameters	Type	Description
Power	String	Property name.
value	String	Property value.
time	Long	UTC timestamp in milliseconds .
code	Integer	Result code.

Error messages

Error code	Message	Description
460	request parameter error	The request parameters are incorrect.
6106	map size must less than 200	A device can report a maximum of 200 properties at any one time.
6313	tsl service not available	When a device reports a property to IoT Platform, IoT Platform examines whether the property format is the same as the predefined format. This error message occurs when this verification service is unavailable. For more information about property verification, see *What is Thing Specification Language (TSL)?*

Note:

IoT Platform compares the format of each reported property with the predefined format in the TSL model to verify the validity of the property. IoT Platform directly drops invalid properties and keeps only the valid properties. If all properties are invalid, IoT Platform drops all properties. However, the response returned to the device will still indicate that the request is successful.

Set device properties

Note:

Set the parameters according to the output and input parameters in the TSL model.

Downstream (passthrough)

- Topic: /sys/{productKey}/{deviceName}/thing/model/down_raw
- Reply topic: /sys/{productKey}/{deviceName}/thing/model/down_raw_reply

Downstream (non-passthrough)

- Topic: /sys/{productKey}/{deviceName}/thing/service/property/set
- Reply topic: /sys/{productKey}/{deviceName}/thing/service/property/set_reply

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": {
    "temperature": "30.5"
  },
  "method": "thing.service.property.set"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	Object	Parameters that are used to set the properties.
method	String	Request method.
temperature	String	Property name.
code	Integer	Result code. For more information, see the common codes on the device.

Get device properties

Note:

- Set the parameters according to the output and input parameters in the TSL model.
- Currently, device properties are retrieved from the device shadow.

Downstream (passthrough)

- Topic: /sys/{productKey}/{deviceName}/thing/model/down_raw
- Reply topic: /sys/{productKey}/{deviceName}/thing/model/down_raw_reply

After a device receives a request to get the device properties, it returns the obtained properties in a reply message to IoT Platform. IoT Platform can subscribe to a data exchange topic to obtain the returned properties. The data exchange topic is `{productKey}/{deviceName}/thing/downlink/reply/message`.

payload: 0x001FFEE23333

Downstream (non-passthrough)

- Topic: /sys/{productKey}/{deviceName}/thing/service/property/get
- Reply topic: /sys/{productKey}/{deviceName}/thing/service/property/get_reply

After a device receives a request to get the device properties, it returns the obtained properties in a reply message. IoT Platform subscribes to a data exchange topic to obtain the returned properties. The data exchange topic is `/{productKey}/{deviceName}/thing/downlink/reply/message`.

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": [
    "power",
    "temp"
  ],
  "method": "thing.service.property.get"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": {
    "power": "on",
    "temp": "23"
  }
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	List	Names of the desired properties.
method	String	Request method.
power	String	Property name.
temp	String	Property name.
code	Integer	Result code.

Report device events

Upstream (passthrough)

- Topic: /sys/{productKey}/{deviceName}/thing/model/up_raw
- Reply topic: /sys/{productKey}/{deviceName}/thing/model/up_raw_reply

Upstream (non-passthrough)

- Topic: /sys/{productKey}/{deviceName}/thing/event/{tsl.event.identifier}/post
- Reply topic: /sys/{productKey}/{deviceName}/thing/event/{tsl.event.identifier}/post_reply

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": {
    "value": {
      "Power": "on",
      "WF": "2"
    },
    "time": 1524448722000
  },
  "method": "thing.event.{tsl.event.identifier}.post"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	List	Parameters of the event to be reported.
method	String	Request method.
value	Object	Parameter values of the event.
Power	String	Parameter name of the event. You can select a parameter name based on the TSL model.
WF	String	Parameter name of the event. You can select a parameter name based on the TSL model.
code	Integer	Result code. For more information, see the common codes on the device.
time	Long	UTC timestamp in milliseconds.

Note:

- `tsl.event.identifier` is the identifier of the event in the TSL model. For information about TSL models, see *What is Thing Specification Language (TSL)?*.
- IoT Platform will compare the format of the received event with the event format that was predefined in the TSL model to verify the validity of the event. IoT Platform directly drops an invalid event and returns a message indicating that the request has failed.

Invoke device services

Downstream (passthrough)

- Topic: /sys/{productKey}/{deviceName}/thing/model/down_raw
- Rely topic: /sys/{productKey}/{deviceName}/thing/model/down_raw_reply

If the invoke method of a service is set to Synchronous in the console, IoT Platform uses RRPC to publish a message synchronously to this topic.

If the invoke method of a service is set to Asynchronous in the console, IoT Platform publishes a message asynchronously to this topic. The device also replies asynchronously. IoT Platform subscribes to the asynchronous reply topic only after the invoke method of the current service has been set to Asynchronous. This reply topic is /sys/{productKey}/{deviceName}/thing/model/down_raw_reply. IoT Platform subscribes to a data exchange topic to obtain the result for an asynchronous call. The data exchange topic is `/{productKey}/{deviceName}/thing/downlink/reply/message`.

Downstream (non-passthrough)

- Topic: /sys/{productKey}/{deviceName}/thing/service/{tsl.service.identifier}
- Reply Topic: /sys/{productKey}/{deviceName}/thing/service/{tsl.service.identifier}_reply

If the invoke method of a service is set to Synchronous in the console, IoT Platform uses RRPC to publish a message synchronously to this topic.

If the invoke method of a service is set to Asynchronous in the console, IoT Platform publishes a message asynchronously to this topic. The device also replies asynchronously. IoT Platform subscribes to the asynchronous reply topic only after the invoke method of the current service has been set to Asynchronous. The reply topic is /sys/{productKey}/{deviceName}/thing/service/{tsl.service.identifier}_reply. IoT Platform subscribes to a data exchange topic to obtain the result for an asynchronous call. The data exchange topic is `/{productKey}/{deviceName}/thing/downlink/reply/message`.

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": {
    "Power": "on",
    "WF": "2"
  },
  "method": "thing.service.{tsl.service.identifier}"
}
```

Response message

```
{
   "id": "123",
   "code": 200,
   "data": {}
 }
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	List	Parameters of the service to be invoked.
method	String	Request method.
value	Object	Parameter name of the service .
Power	String	Parameter name of the service .
WF	String	Parameter name of the service .
code	Integer	Result code. For more information, see the common codes on the device.

Note:

`tsl.service.identifier` is the identifier of the service that has been defined in the TSL model. For more information about TSL models, see *What is Thing Specification Language (TSL)?*.

Disable and delete devices

Disable devices

Downstream

- Topic: /sys/{productKey}/{deviceName}/thing/disable
- Reply topic: /sys/{productKey}/{deviceName}/thing/disable_reply

This topic disables a device connection. IoT Platform publishes messages to this topic asynchrono usly, and the devices subscribe to this topic. Gateways can subscribe to this topic to disable the corresponding sub-devices.

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": {},
  "method": "thing.disable"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	Object	Request parameters. Leave empty.
method	String	Request method.
code	Integer	Result code. For more information, see the common codes on the device.

Enable devices

Downstream

- Topic: /sys/{productKey}/{deviceName}/thing/enable
- Reply topic: /sys/{productKey}/{deviceName}/thing/enable_reply

This topic enables a device connection. IoT Platform publishes messages to this topic asynchrono usly, and the devices subscribe to this topic. Gateways can subscribe to this topic to enable the corresponding sub-devices.

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": {},
  "method": "thing.enable"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	Object	Request parameters. Leave empty.
method	String	Request method.
code	Integer	Result code. For more information, see the common codes on the device.

Delete devices

Downstream

- Topic: /sys/{productKey}/{deviceName}/thing/delete
- Reply topic: /sys/{productKey}/{deviceName}/thing/delete_reply

This topic deletes a device connection. IoT Platform publishes messages to this topic asynchrono usly, and the devices subscribe to this topic. Gateways can subscribe to this topic to delete the corresponding sub-devices.

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": {},
  "method": "thing.delete"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	Object	Request parameters. Leave empty.
method	String	Request method.
code	String	Result code. For more information, see the common codes on the device.

Device tags

Report tags

Upstream

- Topic: /sys/{productKey}/{deviceName}/thing/deviceinfo/update
- Reply topic: /sys/{productKey}/{deviceName}/thing/deviceinfo/update_reply

Device information such as vendor and device model, and static extended information can be saved as device tags.

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": [
    {
      "attrKey": "Temperature",
      "attrValue": "36.8"
    }
  ],
  "method": "thing.deviceinfo.update"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	Object	Request parameters. This parameter can contain a maximum of 200 items.
method	String	Request method.
attrKey	String	Tag name. • Length: Up to 100 bytes. • Valid characters: Lowercase letters a to z , uppercase letters A to Z, numbers 0 to 9, and underscores (_). • The tag name must start with an English letter or underscore (_).
attrValue	String	Tag value.
code	Integer	Result code. A value of 200 indicates the request is successful.

Error code

Error code	Message	Description
460	request parameter error	The request parameters are incorrect.
6100	device not found	The device does not exist.

Delete tags

Upstream

- Topic: /sys/{productKey}/{deviceName}/thing/deviceinfo/delete
- Reply topic: /sys/{productKey}/{deviceName}/thing/deviceinfo/delete_reply

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": [
    {
      "attrKey": "Temperature"
    }
  ],
  "method": "thing.deviceinfo.delete"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	Object	Request parameters.
method	String	Request method.
attrKey	String	Tag name. • Length: Up to 100 bytes.

Parameters	Type	Description
		• Valid characters: Lowercase letters a to z , uppercase letters A to Z, numbers 0 to 9, and underscores (_). • The tag name must start with an English letter or underscore (_).
attrValue	String	Tag value.
code	Integer	Result code. A value of 200 indicates the request is successful.

Error messages

Error code	Message	Description
460	request parameter error	The request parameters are incorrect.
6100	device not found	The device does not exist.

TSL models

A device can publish requests to this topic to obtain the *Device TSL model* from IoT Platform.

- Topic: /sys/{productKey}/{deviceName}/thing/dsltemplate/get
- Reply topic: /sys/{productKey}/{deviceName}/thing/dsltemplate/get_reply

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": {},
  "method": "thing.dsltemplate.get"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": {
    "schema": "https://iot-tsl.oss-cn-shanghai.aliyuncs.com/schema.
json",
    "link": "/sys/1234556554/airCondition/thing/",
    "profile": {
      "productKey": "1234556554",
      "deviceName": "airCondition"
    },
    "properties": [
      {
        "identifier": "fan_array_property",
        "name": "Fan array property",
        "accessMode": "r",
        "required": true,
        "dataType": {
          "type": "array",
          "specs": {
            "size": "128",
            "item": {
              "type": "int"
            }
          }
        }
      }
    ],
    "events": [
      {
        "identifier": "alarm",
        "name": "alarm",
        "desc": "Fan alert",
        "type": "alert",
        "required": true,
        "outputData": [
          {
            "identifier": "errorCode",
             "name": "Error code",
            "dataType": {
              "type": "text",
              "specs": {
                "length": "255"
              }
            }
          }
        ],
        "method": "thing.event.alarm.post"
      }
    ],
    "services": [
      {
        "identifier": "timeReset",
        "name": "timeReset",
        "desc": "Time calibration",
        "inputData": [
          {
            "identifier": "timeZone",
            "name": "Time zone",
            "dataType": {
              "type": "text",
              "specs": {
                "length": "512"
              }
            }
          }
        ],
        "outputData": [
          {
            "identifier": "curTime",
            "name": "Current time",
            "dataType": {
              "type": "date",
              "specs": {}
            }
          }
        ],
        "method": "thing.service.timeReset"
      },
      {
        "identifier": "set",
        "name": "set",
        "required": true,
        "desc": "Set properties",
        "method": "thing.service.property.set",
        "inputData": [
          {
            "identifier": "fan_int_property",
            "name": "Integer property of the fan",
            "accessMode": "rw",
            "required": true,
            "dataType": {
              "type": "int",
              "specs": {
                "min": "0",
                "max": "100",
                "unit": "g/ml",
                "unitName": "Millilitter"
              }
            }
          }
        ],
        "outputData": []
      },
      {
        "identifier": "get",
        "name": "get",
        "required": true,
        "desc": "Get properties",
        "method": "thing.service.property.get",
        "inputData": [
          "array_property",
          "fan_int_property",
          "batch_enum_attr_id",
          "fan_float_property",
           "fan_double_property",
          "fan_text_property",
          "Maid ",
          "batch_boolean_attr_id",
          "fan_struct_property"
        ],
        "outputData": [
          {
            "identifier": "fan_array_property",
            "name": "Fan array property",
            "accessMode": "r",
            "required": true,
            "dataType": {
              "type": "array",
              "specs": {
                "size": "128",
                "item": {
                  "type": "int"
                }
              }
            }
          }
        ]
      }
    ]
  }
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
params	Object	Leave empty.
method	String	Request method.
productKey	String	ProductKey. This is 1234556554 in this example.
deviceName	String	Device name. This is airConditi on in this example.
data	Object	TSL model of the device. For more information, see*What is Thing Specification Language (TSL)?*

Error code

Error code	Message	Description
460	request parameter error	The request parameters are incorrect.
6321	tsl: device not exist in product	The device does not exist.

Update firmware

For information about the firmware update, see *Develop OTA features* and *Firmware update*.

Report the firmware version

Upstream

- Topic: /ota/device/inform/{productKey}/{deviceName}. The device publishes a message to this topic to report the current firmware version to IoT Platform.

Request message

```
{
  "id": 1,
  "params": {
    "version": "1.0.1"
  }
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Version information of the firmware.

Push firmware information

Upstream

- Topic: /ota/device/upgrade/{productKey}/{deviceName}

IoT Platform publishes messages to this topic to push firmware information. The devices subscribe to this topic to obtain the firmware information.

Request message

```
{
  "code": "1000",
  "data": {
    "size": 432945,
    "version": "2.0.0",
    "url": "https://iotx-ota-pre.oss-cn-shanghai.aliyuncs.com/nopoll_0
.4.4.tar.gz?Expires=1502955804&OSSAccessKeyId=XXXXXXXXXXXXXXXXXXXX
&Signature=XfgJu7P6DWWejstKJgXJEH0qAKU%3D&security-token=CAISuQJ1q6
Ft5B2yfSjIpK6MGsyN1Jx5jo6mVnfBglIPTvlvt5D50Tz2IHtIf3NpAusdsv03nWxT7v4f
lqFyTINVAEvYZJOPKGrGR0DzDbDasumZsJbo4f%2FMQBqEaXPS2MvVfJ%2BzLrf0ceu
sbFbpjzJ6xaCAGxypQ12iN%2B%2Fr6%2F5gdc9FcQSkL0B8ZrFsKxBltdUROFbIKP%
2BpKWSKuGfLC1dysQcO1wEP4K%2BkkMqH8Uic3h%2Boy%2BgJt8H2PpHhd9NhXuV2WMzn2
%2FdtJOiTknxR7ARasaBqhelc4zqA%2FPPlWgAKvkXba7aIoo01fV4jN5JXQfAU8KLO8tR
jofHWmojNzBJAAPpYSSy3Rvr7m5efQrrybY1lLO6iZy%2BVio2VSZDxshI5Z3McK
ARWct06MWV9ABA2TTXXOi40BOxuq%2B3JGoABXC54TOlo7%2F1wTLTsCUqzzeIiXVOK
8CfNOkfTucMGHkeYeCdFkm%2FkADhXAnrnGf5a4FbmKMQph2cKsr8y8UfWLC6Iz
vJsClXTnbJBMeuWIqo5zIynS1pm7gf%2F9N3hVc6%2BEeIk0xfl2tycsUpbL2
FoaGk6BAF8hWSWYUXsv59d5Uk%3D",
    "md5": "93230c3bde425a9d7984a594ac55ea1e",
    "sign": "93230c3bde425a9d7984a594ac55ea1e",
    "signMethod": "Md5"
  },
  "id": 1507707025,
  "message": "success"
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
message	String	Result code.
version	String	Version information of the firmware.
size	Long	Firmware size in bytes.
url	String	OSS address of the firmware.
sign	String	Firmware signature.
signMethod	String	Signing method. Currently, the supported methods are MD5 and sha275.
md5	String	This parameter is reserved. This parameter is used to be compatible with old device information. When the signing method is MD5, IoT Platform will assign values to both the sign and md5 parameters.

Report update progress

Upstream

- Topic: /ota/device/progress/{productKey}/{deviceName}

A device subscribes to this topic to report the firmware update progress.

Request message

```
{
  "id": 1,
  "params": {
    "step": "-1",
    "desc": "Firmware update has failed. No firmware information is
available."
  }
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
step	String	Firmware upgrade progress information. Values: • A value from 1 to 100 indicates the progress percentage. • A value of -1 indicates the firmware update has failed. • A value of -2 indicates that the firmware download has failed. • A value of -3 indicates that firmware verification has failed. • A value of -4 indicates that the firmware installation has failed.
desc	String	Description of the current step . If an exception occurs, this parameter displays an error message.

Request firmware information from IoT Platform

- Topic: /ota/device/request/{productKey}/{deviceName}

Request message

```
{
  "id": 1,
  "params": {
    "version": "1.0.1"
  }
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Version information of the firmware.

Remote configuration

Request configuration information from IoT Platform

Upstream

- Topic: /sys/{productKey}/{deviceName}/thing/config/get
- Reply topic: /sys/{productKey}/{deviceName}/thing/config/get_reply

Request message

```
{
  "id": 123,
  "version": "1.0",
  "params": {
    "configScope": "product",
    "getType": "file"
  },
  "method": "thing.config.get"
}
```

Response message

```
{
  "id": "123",
  "version": "1.0",
  "code": 200,
  "data": {
    "configId": "123dagdah",
    "configSize": 1234565,
    "sign": "123214adfadgadg",
    "signMethod": "Sha256",
    "url": "https://iotx-config.oss-cn-shanghai.aliyuncs.com/nopoll_0
.4.4.tar.gz?Expires=1502955804&OSSAccessKeyId=XXXXXXXXXXXXXXXXXXXX
&Signature=XfgJu7P6DWWejstKJgXJEH0qAKU%3D&security-token=CAISuQJ1q6
Ft5B2yfSjIpK6MGsyN1Jx5jo6mVnfBglIPTvlvt5D50Tz2IHtIf3NpAusdsv03nWxT7v4f
lqFyTINVAEvYZJOPKGrGR0DzDbDasumZsJbo4f%2FMQBqEaXPS2MvVfJ%2BzLrf0ceu
sbFbpjzJ6xaCAGxypQ12iN%2B%2Fr6%2F5gdc9FcQSkL0B8ZrFsKxBltdUROFbIKP%
2BpKWSKuGfLC1dysQcO1wEP4K%2BkkMqH8Uic3h%2Boy%2BgJt8H2PpHhd9NhXuV2WMzn2
%2FdtJOiTknxR7ARasaBqhelc4zqA%2FPPlWgAKvkXba7aIoo01fV4jN5JXQfAU8KLO8tR
jofHWmojNzBJAAPpYSSy3Rvr7m5efQrrybY1lLO6iZy%2BVio2VSZDxshI5Z3McK
ARWct06MWV9ABA2TTXXOi40BOxuq%2B3JGoABXC54TOlo7%2F1wTLTsCUqzzeIiXVOK
8CfNOkfTucMGHkeYeCdFkm%2FkADhXAnrnGf5a4FbmKMQph2cKsr8y8UfWLC6Iz
vJsClXTnbJBMeuWIqo5zIynS1pm7gf%2F9N3hVc6%2BEeIk0xfl2tycsUpbL2
FoaGk6BAF8hWSWYUXsv59d5Uk%3D",
    "getType": "file"
  }
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.
version	String	Protocol version. Currently, the value can only be 1.0.
configScope	String	Configuration scope. Currently , IoT Platform supports only product dimension configurat ion. Set the value to product.
getType	String	Type of the desired configurat ion. Currently, the supported type is file. Set the value to file.
configId	String	Configuration ID.
configSize	Long	Configuration size in bytes.
sign	String	Signature.
signMethod	String	Signing method. The supported signing method is Sha256.
url	String	OSS address of the confgurati on.
code	Integer	Result code. A value of 200 indicates that the request is successful.

Error code

Error code	Message	Description
6713	thing config function is not available	Remote configuration is disabled for the device. Enable remote configuration for the device.
6710	no data	No configuration data is available.

Push configurations

Downstream

- Topic: /sys/{productKey}/{deviceName}/thing/config/push
- Reply topic: /sys/{productKey}/{deviceName}/thing/config/push_reply

A device subscribes to this topic to obtain configurations that have been pushed by IoT Platform. After you have configured configuration push for multiple devices in the IoT Platform console, IoT Platform pushes configurations to the devices asynchronously. IoT Platform subscribes to a data exchange topic to obtain the result that is returned by the device. The data exchange topic is `/{productKey}/{deviceName}/thing/downlink/reply/message`.

Request message

```
{
  "id": "123",
  "version": "1.0",
  "params": {
    "configId": "123dagdah",
    "configSize": 1234565,
    "sign": "123214adfadgadg",
    "signMethod": "Sha256",
    "url": "https://iotx-config.oss-cn-shanghai.aliyuncs.com/nopoll_0
.4.4.tar.gz?Expires=1502955804&OSSAccessKeyId=XXXXXXXXXXXXXXXXXXXX
&Signature=XfgJu7P6DWWejstKJgXJEH0qAKU%3D&security-token=CAISuQJ1q6
Ft5B2yfSjIpK6MGsyN1Jx5jo6mVnfBglIPTvlvt5D50Tz2IHtIf3NpAusdsv03nWxT7v4f
lqFyTINVAEvYZJOPKGrGR0DzDbDasumZsJbo4f%2FMQBqEaXPS2MvVfJ%2BzLrf0ceu
sbFbpjzJ6xaCAGxypQ12iN%2B%2Fr6%2F5gdc9FcQSkL0B8ZrFsKxBltdUROFbIKP%
2BpKWSKuGfLC1dysQcO1wEP4K%2BkkMqH8Uic3h%2Boy%2BgJt8H2PpHhd9NhXuV2WMzn2
%2FdtJOiTknxR7ARasaBqhelc4zqA%2FPPlWgAKvkXba7aIoo01fV4jN5JXQfAU8KLO8tR
jofHWmojNzBJAAPpYSSy3Rvr7m5efQrrybY1lLO6iZy%2BVio2VSZDxshI5Z3McK
ARWct06MWV9ABA2TTXXOi40BOxuq%2B3JGoABXC54TOlo7%2F1wTLTsCUqzzeIiXVOK
8CfNOkfTucMGHkeYeCdFkm%2FkADhXAnrnGf5a4FbmKMQph2cKsr8y8UfWLC6Iz
vJsClXTnbJBMeuWIqo5zIynS1pm7gf%2F9N3hVc6%2BEeIk0xfl2tycsUpbL2
FoaGk6BAF8hWSWYUXsv59d5Uk%3D",
    "getType": "file"
  },
  "method": "thing.config.push"
}
```

Response message

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

Parameter description

Parameters	Type	Description
id	Long	Message ID. Reserved parameter for future use.

Parameters	Type	Description
version	String	Protocol version. Currently, the value can only be 1.0.
configScope	String	Configuration scope. Currently , IoT Platform supports only product dimension configurat ion. Set the value to product.
getType	String	Type of the desired configurat ion. The supported type is files . Set the value to file.
configId	String	Configuration ID.
configSize	Long	Configuration size in bytes.
sign	String	Signature.
signMethod	String	Signing method. The supported signing method is Sha256.
url	String	OSS address of the confgurati on.
code	Integer	Result code. For more information, see the common codes on the device.

Common codes on devices

Common codes on devices indicate the results that are returned to IoT Platform in response to requests from IoT Platform.

Result code	Message	Description
200	success	The request is successful.
400	request error	Internal service error.
460	request parameter error	The request parameters are invalid. The device has failed input parameter verification.
429	too many requests	The system is busy. This code can be used when the device is too busy to process the request.

Result code	Message	Description
100000-110000		Devices use numbers from 100000 to 110000 to indicate device-specific error messages to distinguish them from error message on IoT Platform.