IoT Device Access

User Guide

Issue	1.0
Date	2024-12-31

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions

HUAWEI and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Contents

1 Overview

IoT Device Access (IoTDA) lets you connect and manage an enormous number of devices. It can be used together with other Huawei Cloud services to quickly construct IoT applications, simplify device management, and reduce manual operations, thereby improving management efficiency. Using the IoTDA console, you can create, develop, and debug products, register, manage, and authenticate devices, and upgrade software and firmware. You can also create rules for device linkage and data forwarding. In addition, you can monitor the device status based on reports generated from product and device data.

Function	Description
Product	A product is a collection of devices with the same capabilities or features. On the IoTDA console, you can quickly develop product models and codecs, and use functions such as online debugging and topic customization in end-to-end (E2E) IoT development. This helps you improve integration development efficiency and shorten the construction period of IoT solutions.
Product model	A product model defines the properties of a device, such as the color, size, collected data, identifiable commands, and reported events. You can create a product model on the IoTDA console.
Device	A device is a physical entity that belongs to a product. Each device has a unique ID. It can be a device directly connected to the platform, or a gateway that connects child devices to the platform.
Device authentication	The platform authenticates devices that attempt to access it. Currently, the platform supports two authentication modes: secret authentication and X.509 authentication. After the verification is successful and the device is connected to the platform, the devices can communicate with the platform.
Group and tag	A group is a collection of devices. You can create groups for all the devices in a resource space based on different rules, such as regions and types, and you can operate the devices by group. You can define tags and bind tags to devices.

Function	Description
Software and firmware upgrades	You can upgrade software and firmware of devices that support LwM2M and MQTT in over the air (OTA) mode.
Device shadow	A device shadow is a JSON file that stores the device status, latest device properties reported, and device configurations to deliver. Each device has only one shadow. A device can retrieve and set its shadow to synchronize the status, either from the shadow to the device or from the device to the shadow.
Gateway and child device	Devices can be directly or indirectly connected to the IoT platform. Indirectly connected devices access the platform through gateways.
Rule	You can set rules for devices connected to the platform. If the conditions set in a rule are met, the platform triggers the corresponding action. Device linkage and data forwarding rules are available.
Monitoring and O&M	IoTDA provides monitoring and O&M functions such as statistics reports, online debugging, message tracing, current alarms, and run logs. You can also monitor the device running status, device communications, and user operations, and quickly trace and locate faults, ensuring device reliability and security.
Resource space	Resource space is a space allocated for your applications. Resources (such as products and devices) created on the platform must belong to a resource space. You can use the resource space for domain-based management.
IoTDA instance	To meet the requirements of enterprise customers with different IoT device scales, IoTDA provides three editions: basic (shared instance), standard (standard instance), and enterprise (dedicated instance). You can purchase the most appropriate instance type and instance specifications based on your service scenario, device scale, and data collection frequency.
Data reporting	After being registered with the platform and powered on, a device can collect and report data based on the service logic. Data collection and reporting can be triggered by a schedule or by specific events.
Data forwarding	The data forwarding function connects IoTDA with other Huawei Cloud or third-party cloud services to smoothly transfer device data to the message middleware, storage and data analysis services, and applications.
Command delivery	A product model defines commands that can be delivered to the devices. Applications can call platform APIs to deliver commands to the devices to effectively manage these devices.

Browser Requirements

To ensure good display effect and ease of use, use a browser with good compatibility. The table below lists the browser requirements.

Browser Type	Version Requirements	Recommended Resolution
Microsoft Edge	The latest three stable versions are supported and tested.	1366 x 768
Firefox	The latest three stable versions are supported and tested.	
Google Chrome	The latest three stable versions are supported and tested.	

2 IoTDA Instances

2.1 Overview

IoTDA uses instances to implement data and resource isolation. Currently, the platform provides standard edition (standard instance).

- **Standard Edition**

  You can experience all functions of Standard Edition free of charge within the limit on the number of messages and devices per day.

  If you want to increase device and message quotas, modify the configuration on the instance page and select units with desired specifications.

Restrictions on Purchasing Instances

Note the following restrictions to prevent purchasing and creation failures:

- Restrictions on purchasing standard edition instances
  - Each tenant can subscribe to only one free trial instance in the same region.

2.2 Buying an Instance

Procedure

Step 1 In the navigation pane, choose **IoTDA Instances**, click **Buy Instance**, and select **Enterprise**.

Step 2 Enter configuration information about the instance. The system automatically calculates the fee based on **Instance Specifications** and **Required Duration**.

Parameter	Description
Billing Mode	Billing mode of an instance. The value is fixed at **Yearly/Monthly**.

Parameter	Description
Region	Region where IoTDA is deployed. Currently, **CN North-Beijing4**, **CN East-Shanghai1**, and **CN South-Guangzhou** are supported. **NOTE** Select a nearby region to ensure the lowest latency possible.
Network	Select an available VPC and subnet. Create a VPC on the **VPC console**.
Security Group	Select a security group. Create a security group on the **VPC console**.
Public Network Access	Set this parameter based on your requirements. If configured, devices can access the platform via the Internet.
Private Network Access	• If you select **Create Provide Network Access Point**, a VPC endpoint is automatically purchased and an access address is automatically allocated. • If it is not selected, private network ingestion is still required. You can purchase **VPC endpoints** for connection.
Access Ports	You can configure access ports or use the default ports. The following ports are provided: Application access: HTTPS (443) and AMQPS (5671) Device access: CoAP (5683), CoAPS (5684), MQTT (1883), MQTTS (8883), and HTTPS (443)
Instance Version	Select **Enterprise**.
Enterprise Project	This parameter is displayed only for enterprise users who have enabled the enterprise project function. Enterprise projects let you manage cloud resources and users by project. For details, see **Enterprise Center Overview**.
Tags	Tags are key-value pairs, which are used to identify, classify, and search for instances. Instance tags are used to filter and manage instances only. For details, see **Instance Tag Management**.
Instance Name	Set a name for easy management. The value can contain a maximum of 64 characters consisting of letters, numbers, underscores (_), and hyphens (-).
Cryptographic Algorithm	General cryptographic algorithms (such as RSA and SHA-256) and SM series commercial cryptographic algorithms (Chinese cryptographic algorithms such as SM2, SM3, and SM4) are available.
Description	Provide a description for the instance, which can be a brief description based on the instance user and usage.

Parameter	Description
Required Duration	Select a duration as required. You can select **Auto-renew** so the instance will be automatically renewed when it expires.

Step 3 Click **Buy**.

Step 4 Confirm the specifications and click **Pay**.

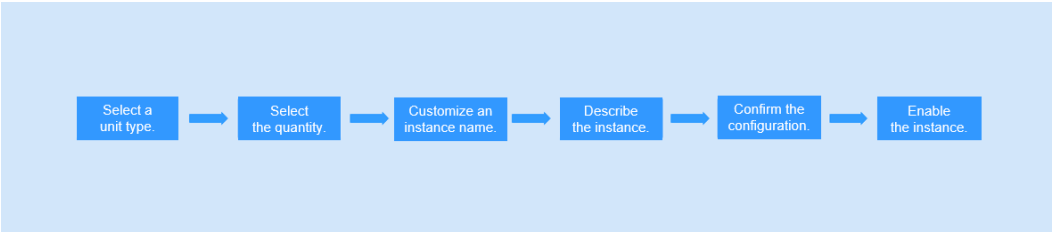
Step 5 On the payment page, select a payment method and click **Pay**.

----End

Buying a Standard Edition Instance

The standard instance provides configurable instance specifications. You can purchase more economical instances based on your service model.

Before the purchase, browse the overall process for a more efficient operation.



Procedure

Step 1 In the navigation pane, choose **IoTDA Instances**, click **Buy Instance**, and select **Standard**.

Step 2 Enter configuration information about the instance. The system automatically calculates the fee based on instance specifications and the pay-per-use mode.

Parameter	Description
Billing Mode	Billing mode of an instance. Currently, only **Pay-per-use** is supported..
Region	Region where IoTDA is available. Currently, AP-Bangkok, AP-Singapore, AF-Johannesburg, LA-Sao Paulo1, and CN-Hong Kong are supported. **NOTE** Select a nearby region to ensure the lowest latency possible.

Parameter	Description
Specifications	SUF unit with free specifications, SU1 unit with small specifications, SU2 unit with medium specifications, SU3 unit with large specifications, and SU4 unit with ultra-large specifications are available. You can obtain the recommended instance specifications based on the required upstream and downstream TPS. **NOTE** • Instance specifications of Standard Edition = Number of units in the instance x Specifications of each unit • A Standard Edition instance can contain multiple units of the same type. The maximum number of units in a single instance is 100, and the maximum TPS of upstream and downstream messages of a single instance is 100,000. For example, even if an instance contains 100 S3 units, the maximum TPS of the instance is 100,000. • You can increase the number of units in the instance online, for example, upgrading a standard instance from three SU1 units to five SU2 units. • An instance cannot contain different types of units, for example, *M* SU1 units and *N* SU2 units.
Enterprise Project	This parameter is displayed only for enterprise users who have enabled the enterprise project function. Enterprise projects let you manage cloud resources and users by project. For details, see **Enterprise Center Overview**.
Tags	Tags are key-value pairs, which are used to identify, classify, and search for instances. Instance tags are used to filter and manage instances only. For details, see **Instance Tag Management**.
Instance Name	Enter a name for easy management. The value can contain a maximum of 64 characters consisting of letters, numbers, underscores (_), and hyphens (-).
Description	Provide a description for the instance, which can be a brief description based on the instance user and usage.

Step 3 Click **Buy**.

Step 4 Confirm the specifications and click **Submit**.

----End

2.3 Instance Management

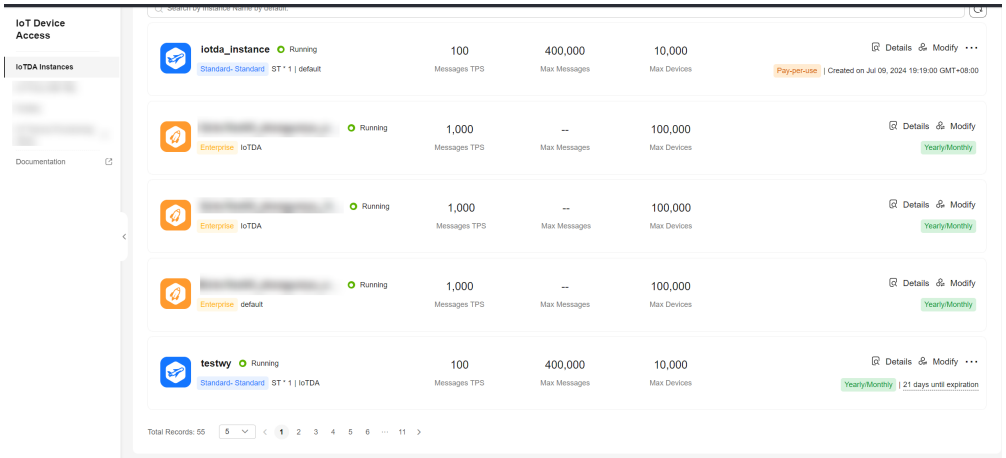
Selecting an Instance

After an instance is created, select the instance before creating products and devices and setting other functions in the instance.

Step 1 Access the **IoTDA** service page and click **Access Console**.

Step 2 In the navigation pane, choose **IoTDA Instances**, and click the target instance card.

Figure 2-1 Instance management - Changing instance



----End

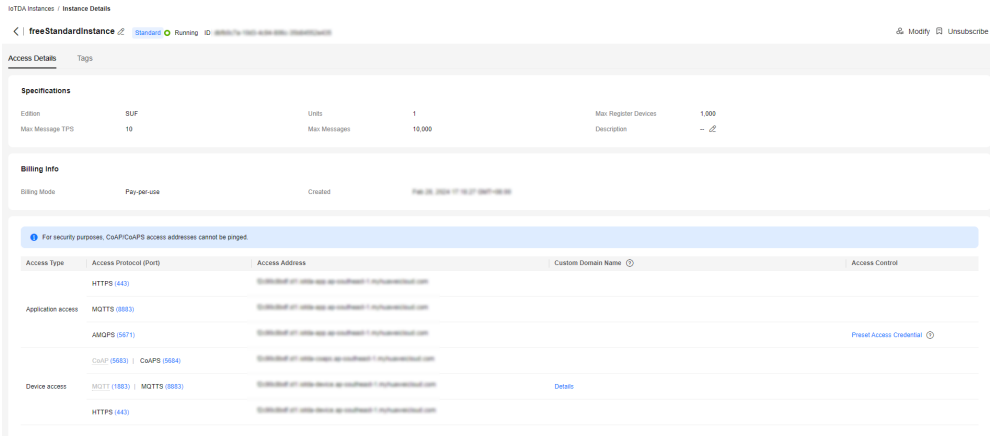
Viewing Instance Details

After purchasing an IoTDA instance, you can view the instance details in the instance details page, including the instance ID, name, and specifications.

Step 1 Access the **IoTDA** service page and click **Access Console**.

Step 2 In the navigation pane, choose **IoTDA Instances** and click **Details** corresponding to an instance.

Figure 2-2 Instance management - Instance details



----End

Modifying Instance Specifications

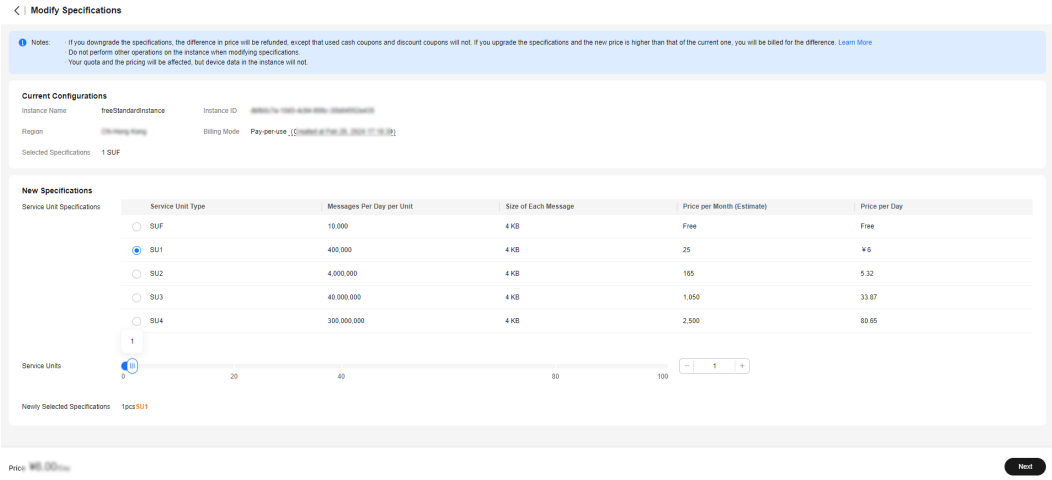
You can upgrade the specifications of an IoTDA instance based on service requirements. Modifying instance specifications does not affect services.

Step 1 Access the **IoTDA** service page and click **Access Console**.

Step 2 In the navigation pane, choose **IoTDA Instances**. Locate the target instance, click **Modify**, and select the new instance specifications.

Step 3 Set a delay for the change to take effect. After you set a maintenance window, the change will be performed in the scheduled time.

Figure 2-3 Instance management - Modifying specifications

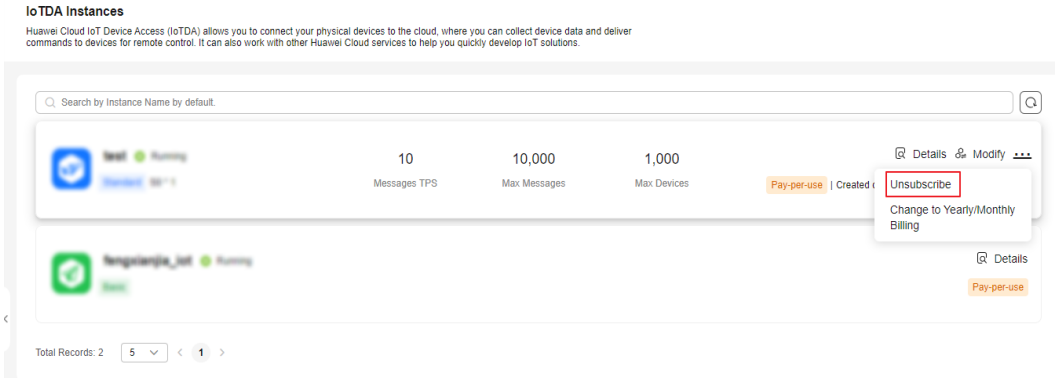


----End

Unsubscribing from an Instance

If an instance is no longer required, you can choose **More** > **Unsubscribe** in the **Operation** column to release your cloud service resources. For details, see **Unsubscriptions**.

Figure 2-4 Instance management - Unsubscribing from an instance



2.4 Tag Management

2.4.1 Overview

Scenarios

You can add tags to cloud resources for quicker search. You can view, modify, and delete these tags in a unified manner, facilitating cloud resource management. You can also use the tags to collect resource cost statistics from the service dimension.

Tag Naming Rules

- Each tag consists of a key-value pair.
- A maximum of 20 tags can be added for an IoTDA instance.
- For each resource, a tag key must be unique and can have only one tag value.
- A tag consists of a tag key and a tag value. Table 1 lists the tag key and value requirements.

Table 2-1 Tag naming rules

Parameter	Rule	Example
Tag key	The value cannot be empty. Must be unique for the same instance. A tag key can contain a maximum of 36 characters. Only letters, digits, hyphens (-), underscores (_), and Unicode characters (\u4E00-\u9FFF) are allowed.	Organization
Tag value	It can contain a maximum of 43 characters and can be left blank. Only letters, digits, periods (.), hyphens (-), underscores (_), and Unicode characters (\u4E00-\u9FFF).are allowed.	Apache

NOTE

If your organization has configured tag policies for IoTDA, add tags to instances based on the policies. If a tag does not comply with the tag policies, instance creation may fail. Contact your administrator to learn more about tag policies.

2.4.2 Adding a Tag

You can add tags for IoTDA instances in either of the following ways:

- **Adding a Tag on the Instance Details Page**
- **Adding a Tag on the Tag Management Service Page**

For details about how to use predefined tags, see **Using Predefined Tags**.

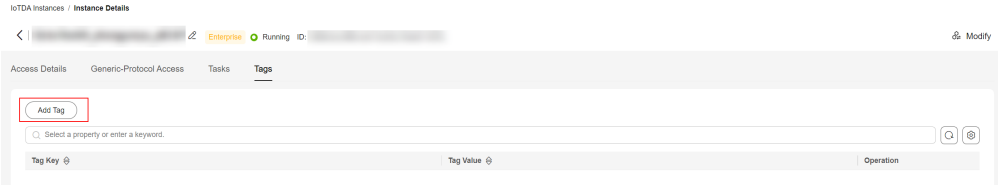
Adding a Tag on the Instance Details Page

Step 1 Access the **IoTDA** service page and click **Access Console**.

Step 2 In the navigation pane, choose **IoTDA Instances** and click **Details** under **Enterprise Edition**.

Step 3 Click the **Tags** tab and then **Add Tag**. In the displayed dialog box, enter the tag key and tag value. For details, see **Tag naming rules**.

Figure 2-5 Instance management - Adding a tag



----End

Adding a Tag on the Tag Management Service Page

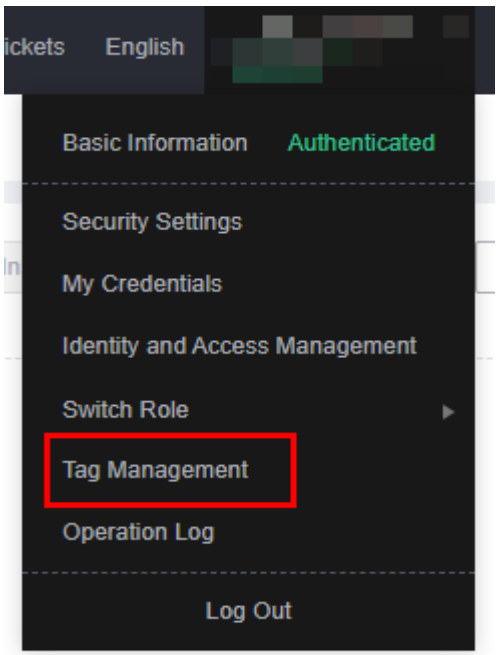
NOTE

This method is suitable for adding tags with the same tag key to multiple resources.

Step 1 Access the **IoTDA** service page and click **Access Console**.

Step 2 In the upper right corner of the page, click the username and select **Tag Management** from the drop-down list.

Figure 2-6 Tag management

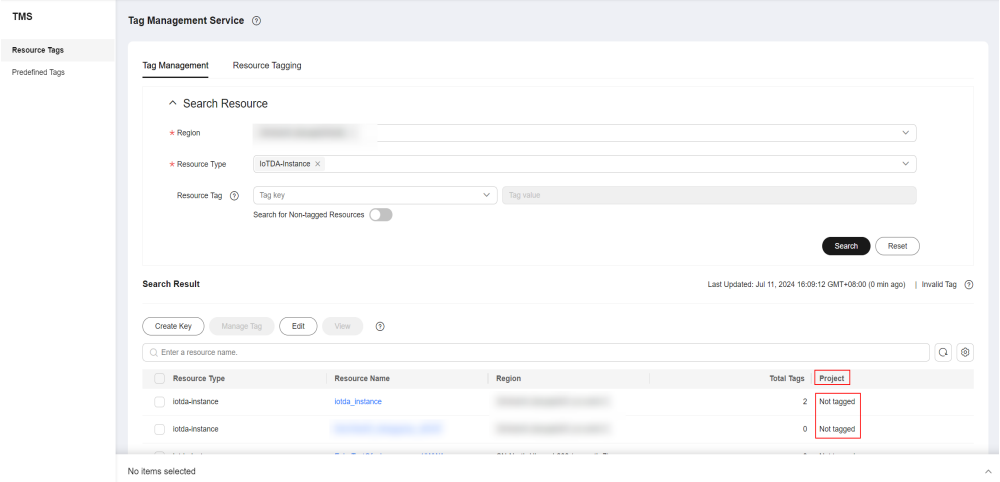


Step 3 On the **Resource Tags** page, select the region where the resource is located, set **Resource Type** to **IoTDA-Instance**, and click **Search**. All IoTDA instance resources in the selected region are displayed.

Step 4 In the **Search Result** area, click **Create Key**. In the displayed dialog box, enter a key (for example **project**) and click **OK**. After the tag is created, the tag key is

added to the resource tag list, as shown in **Figure 2-7**. If the tag is not contained in the list, click  and select the created tag from the drop-down list. By default, the value of the tag key is **Not tagged**. You need to set a value for the tag of each resource to associate the tag with the resource.

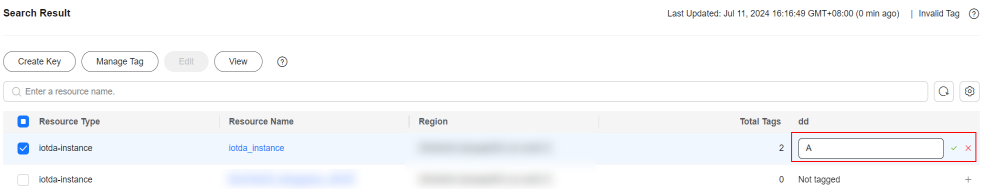
Figure 2-7 Instance tag - Tag management



Step 5 Click **Edit** to make the resource list editable.

Step 6 Select the row where the IoTDA instance resource is located, and enter the tag value (for example, A). After a value is set for a tag key, the number of tags is incremented by 1. Repeat the preceding steps to add tag values for other instances.

Figure 2-8 Instance tag - Entering a tag value



----End

Using Predefined Tags

If you want to add the same tag to multiple resources, you can create a predefined tag on the Tag Management Service (TMS) console and select the tag for the resources. This frees you from having to repeatedly enter tag keys and values. The procedure is as follows:

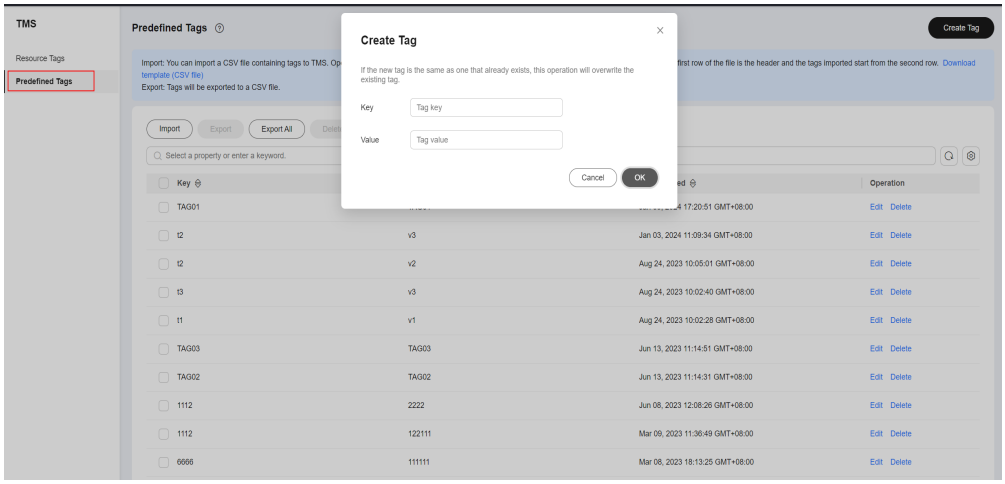
Step 1 Log in to the console.

Step 2 In the upper right corner of the page, click the username and select **Tag Management** from the drop-down list.

Step 3 In the navigation pane, choose **Predefined Tags**. In the right pane, click **Create Tag** enter a key (for example **project**) and a value (for example **A**) in the displayed dialog box.

Step 4 Choose **Service List** > **IoT Device Access** and select the predefined tag by following the procedure for adding a tag.

Figure 2-9 Instance tag - Predefined tags



----End

2.4.3 Deleting a Tag

If you no longer need a tag, delete it in any of the following ways:

- **Deleting a Tag on the Instance Details Page**
- **Deleting a Tag on the TMS Console**
- **Batch Deleting Tags on the TMS Console**

Deleting a Tag on the Instance Details Page

Step 1 Access the **IoTDA** service page and click **Access Console**.

Step 2 In the navigation pane, choose **IoTDA Instances** and click **Details** under **Enterprise Edition**.

Step 3 Click the **Tags** tab. Locate the row containing the tag to be deleted and click **Delete** in the **Operation** column. In the **Delete Tag** dialog box, click **OK**.

Figure 2-10 Instance management - Deleting a tag

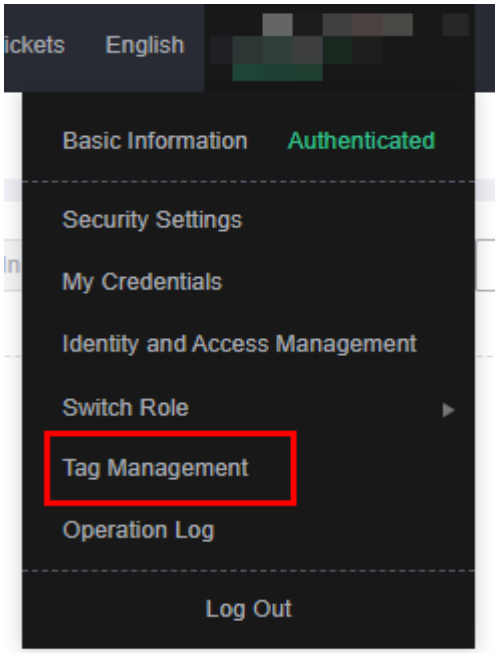
----End

Deleting a Tag on the TMS Console

Step 1 Access the **IoTDA** service page and click **Access Console**.

Step 2 In the upper right corner of the page, click the username and select **Tag Management** from the drop-down list.

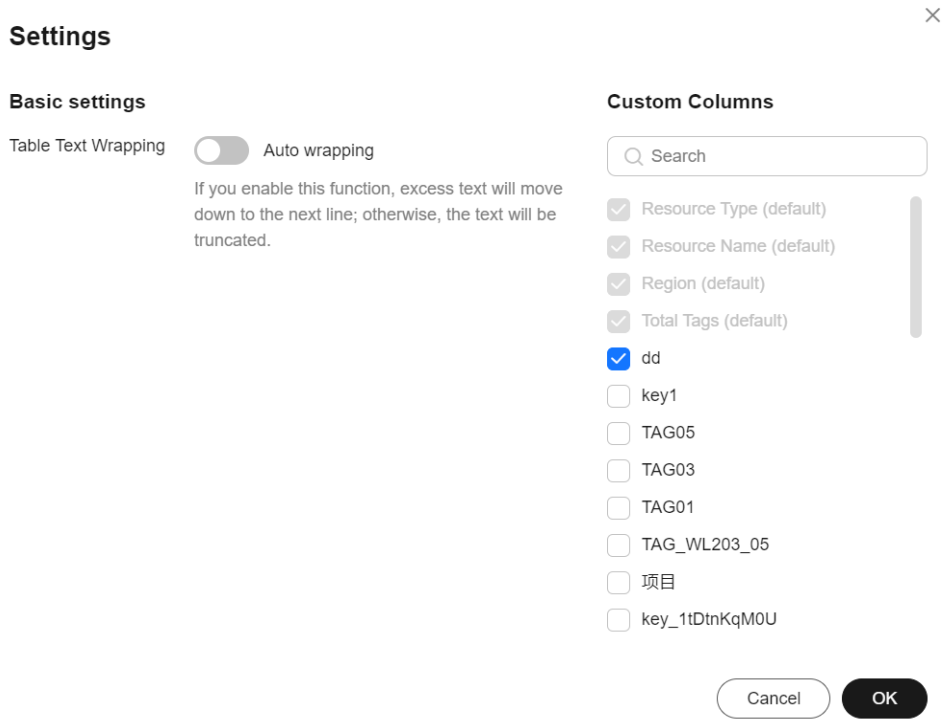
Figure 2-11 Tag management



Step 3 On the **Resource Tags** page, select the region where the resource is located, set **Resource Type** to **IoTDA-Instance**, and click **Search**. All IoTDA instance resources in the selected region are displayed.

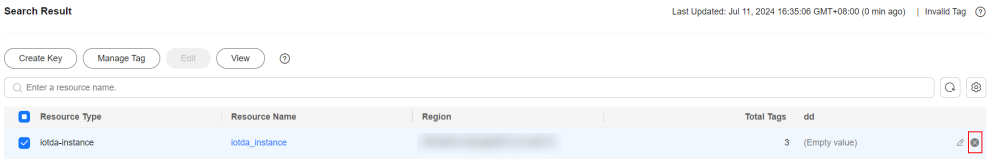
Step 4 In the **Search Result** area, click **Edit** to make the resource tag list editable. Click ⚙ and select the tag key to be deleted from the drop-down list. You are advised not to select more than 10 keys to display.

Figure 2-12 Instance tag - Tag list



Step 5 Locate the row containing the target IoTDA instance resource and click .

Figure 2-13 Instance tag - Deleting a tag



----End

Batch Deleting Tags on the TMS Console

Step 1 Access the **IoTDA** service page and click **Access Console**.

Step 2 In the upper right corner of the page, click the username and select **Tag Management** from the drop-down list.

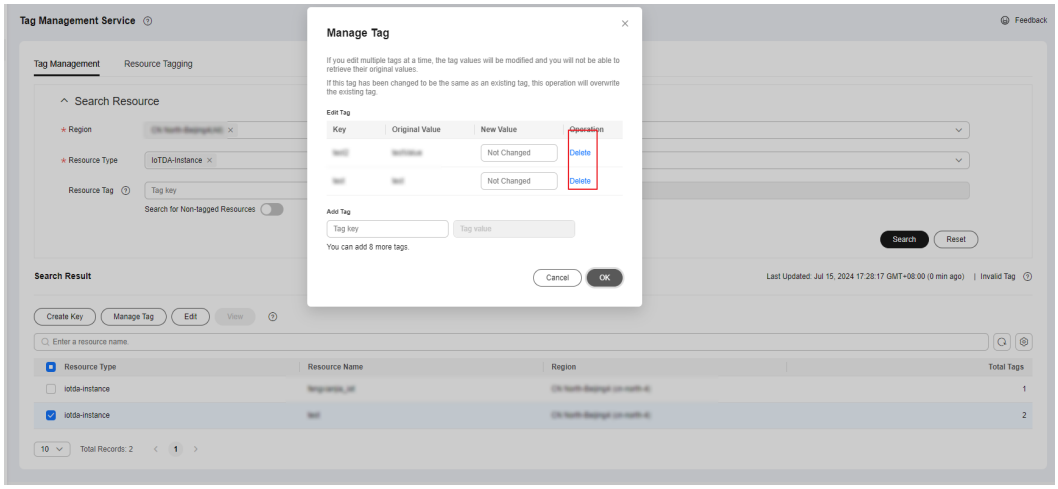
Step 3 On the **Resource Tags** page, select the region where the resource is located, set **Resource Type** to **IoTDA-Instance**, and click **Search**. All IoTDA instance resources in the selected region are displayed.

Step 4 Select the IoTDA instance resource whose tag is to be deleted.

Step 5 Click **Manage Tag** in the upper left corner of the list.

Step 6 In the displayed **Manage Tag** dialog box, click **Delete** in the **Operation** column. Click **OK**.

Figure 2-14 Instance tag - Deleting tags in batches



----End

2.4.4 Searching for Resources by Tag

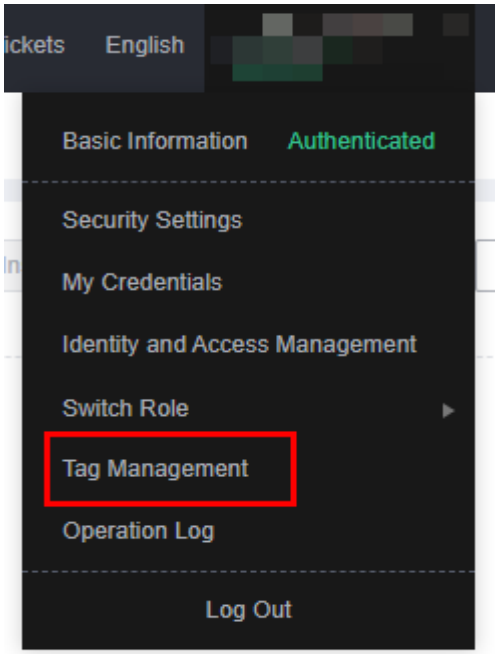
After adding tags to cloud resources, you can use the methods described in this section to search for resources by tag.

Filtering Resources By Tag

Step 1 Access the **IoTDA** service page and click **Access Console**.

Step 2 In the upper right corner of the page, click the username and select **Tag Management** from the drop-down list.

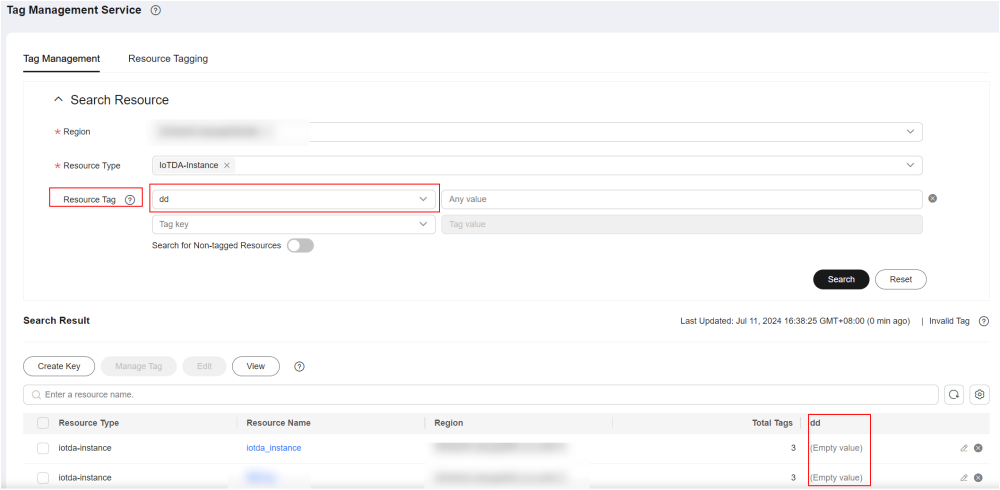
Figure 2-15 Tag management



Step 3 On the **Resource Tags** page, set the search criteria, including **Region**, **Resource Type**, and **Resource Tag**.

Step 4 Click **Search**. All the resources that meet the search criteria will be displayed.

Figure 2-16 Instance tag - Searching for resources by tag



----End

3 Resource Spaces

Resource space is a space allocated for your applications. Resources (such as products and devices) created on the platform must belong to a resource space. You can use the resource space for domain-based management.

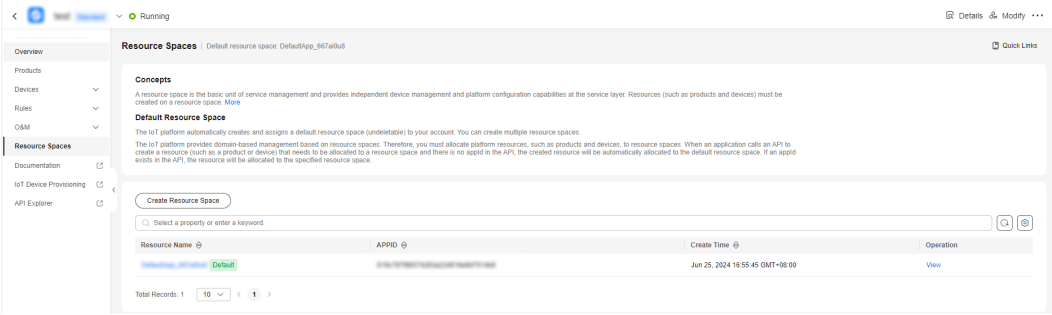
- You can create a maximum of 10 resource spaces. By default, the space automatically created by the platform when the IoTDA service is subscribed to for the first time is the default resource space.
- An **app_id**, which is a unique identifier of a resource space, is allocated by the platform when the resource space is created. **app_id** is also used in API calls.
- After a resource space is created, you can view its **app_id** in the resource space.
- The default resource space cannot be deleted. After a resource space is deleted, all resources in the space, such as devices, products, and subscription data, are deleted from the platform and cannot be restored. Exercise caution when deleting a resource space.

Creating a Resource Space

When you subscribe to IoTDA for the first time, the platform automatically creates the default resource space. Each instance has only one default resource space, which cannot be deleted.

You can create a product or register a device in the default resource space. You can also perform the following steps to create a resource space:

Figure 3-1 Resource space - Resource space list



NOTE

If you subscribed to IoTDA before 00:00 on Apr 27, 2020, see **Which Resource Space Will Be Set As Default on the IoT Platform?**.

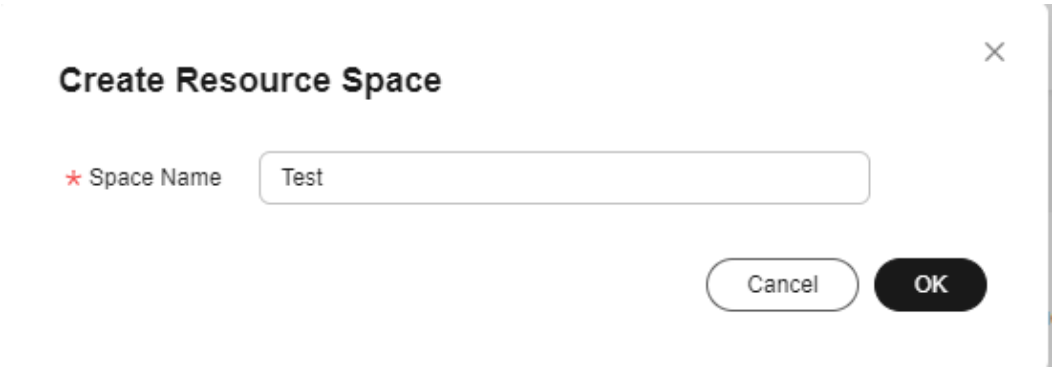
Step 1 Access the **IoTDA** service page and click **Access Console**.

Step 2 In the navigation pane, choose **IoTDA Instances**, and click the target instance card.

Step 3 In the navigation pane, choose **Resource Spaces**. On the displayed page, click **Create Resource Space**. On the displayed dialog box, set **Space Name** and click **OK**.

The resource space name must be unique under the account.

Figure 3-2 Resource space - Creating a resource space

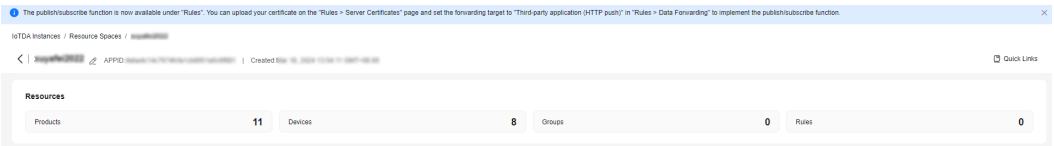


----End

Querying a Resource Space

After a resource space is created, choose **Resource Spaces** and click **View** to check the app ID (**app_id**), creation time, number of products, number of devices, number of groups, and number of created rules under the resource space. To create products, devices, groups, and rules in another resource space, switch to the target resource space.

Figure 3-3 Resource space - Viewing resource spaces

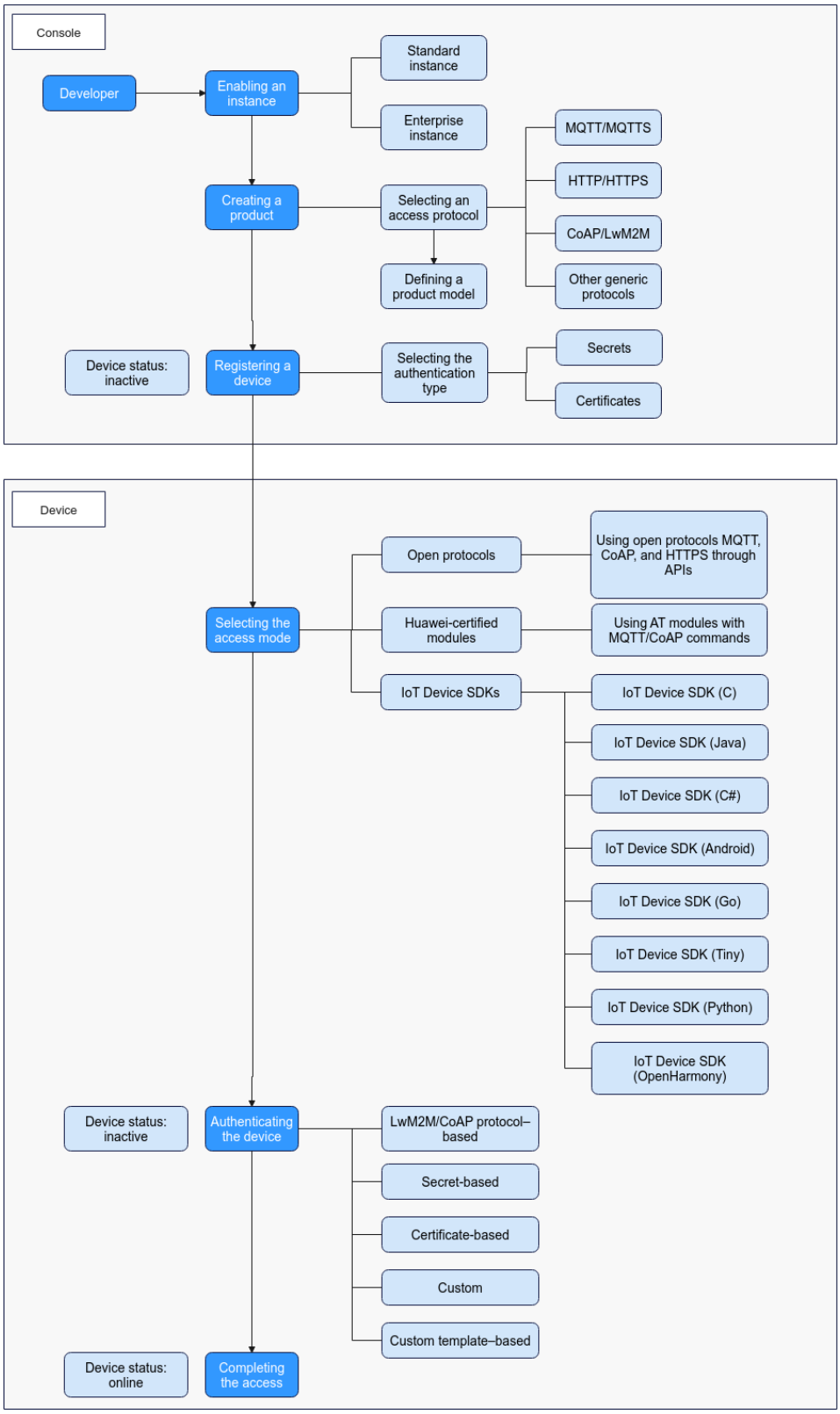


4 Device Access

4.1 Overview

Device Access Process

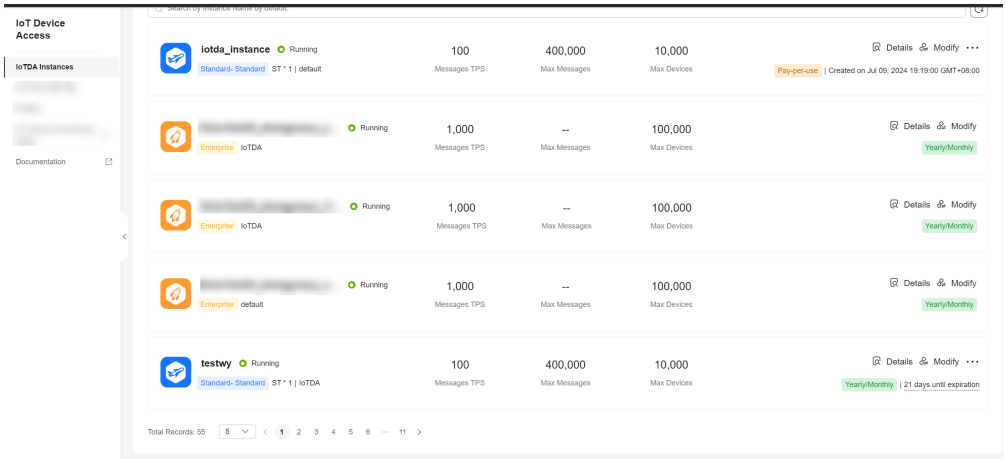
Figure 4-1 Device access process



Platform Connection Information

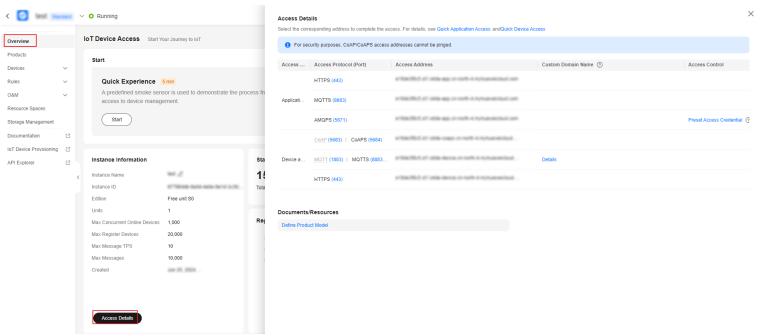
1. Log in to the **IoTDA console**. In the navigation pane, choose **IoTDA Instances**, and click the target instance card.

   Figure 4-2 Instance management - Changing instance



2. In the navigation pane, choose **Overview**. In the **Instance Information** area, click **Access Details**.

   Figure 4-3 Obtaining access information



Certificates

The following certificates are used when devices and applications need to verify IoTDA.

NOTE

- The certificates apply only to Huawei Cloud IoTDA and must be used together with the corresponding domain name.
- CA certificates cannot be used to verify server certificates after their expiration dates. Replace these certificates before expiration dates to ensure that devices can connect to the IoT platform properly.

Table 4-1 Certificates

Certificate Package Name	Region and Edition	Certificate Type	Certificate Format	Description	Download Link
certificate	CN-Hong Kong, AP-Singapore, AP-Bangkok, AF-Johannesburg, LA-Santiago, LA-Sao Paulo1, and ME-Riyadh	Device certificate	pem, jks, and bks	Used by a device to verify the platform identity. The certificate must be used together with the device access domain name.	**Certificate file**

4.2 Device Authentication

4.2.1 Overview

IoTDA authenticates a device when the device attempts to access the platform. The authentication process depends on the access method.

Access Type	Authentication Mode
Device using LwM2M over CoAP	You can call the API for **creating a device** or use the IoTDA console to register a device. Then, when connecting to the platform, a non-security device does not use DTLS/DTLS+, and carries the node ID to get authenticated. A security device uses DTLS/DTLS+, and carries both the secret and node ID to get authenticated.

Access Type	Authentication Mode
Device using MQTT or MQTTS	• Using secrets: You can call the API for **creating a device** or use the IoTDA console to register a device. Then, you can hardcode the device ID and secret returned by the platform into the device, and preset a CA certificate on the device if it uses MQTTS protocols. When connecting to the platform, the device uses the device ID and secret to get authenticated. • Using certificates: You can upload a device CA certificate on the IoTDA console, and register the device, either by calling the API for **creating a device** or using the console. Then, you can hardcode the device ID returned by the platform into the device. When connecting to the platform, the device uses the X.509 certificate to get authenticated. • Using custom authentication: Before connecting a device to the platform, you can use the application to configure custom authentication information on the console, and then configure custom authentication functions by using **FunctionGraph**. When the device connects to IoTDA, the platform obtains parameters such as the device ID and custom authentication function name, and sends an authentication request to FunctionGraph. The user implements the authentication logic to complete access authentication. • Using custom templates: You can use a custom authentication template to orchestrate internal functions provided by the platform and flexibly customize triplet parameters **ClientId**, **Username**, and **Password** for MQTT device authentication.

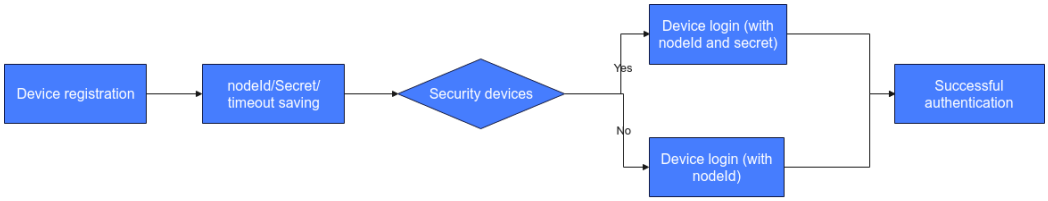
4.2.2 LwM2M/CoAP Authentication

Introduction

LwM2M/CoAP authentication supports both encrypted and non-encrypted access modes. Non-encrypted mode: Devices connect to IoTDA carrying the node ID through port 5683. Encrypted mode: Devices connect to IoTDA carrying node ID and secret through port 5684 by the DTLS/DTLS+ channel.

Authentication for Devices Using LwM2M over CoAP

Figure 4-4 LwM2M/CoAP access authentication process



1. An application calls the API for creating a device to register a device. Alternatively, a user uses the IoTDA console to register a device.
2. The platform allocates a secret to the device and returns **timeout**.

   NOTE

   - The secret can be defined during device registration. If no secret is defined, the platform allocates one.
   - If the device is not connected to the platform within the duration specified by **timeout**, the platform deletes the device registration information.
3. During login, the device sends a connection authentication request carrying the node ID (such as the IMEI) and secret if it is a security device, or carrying the node ID if it is a non-security device.
4. If the authentication is successful, the platform returns a success message, and the device is connected to the platform.

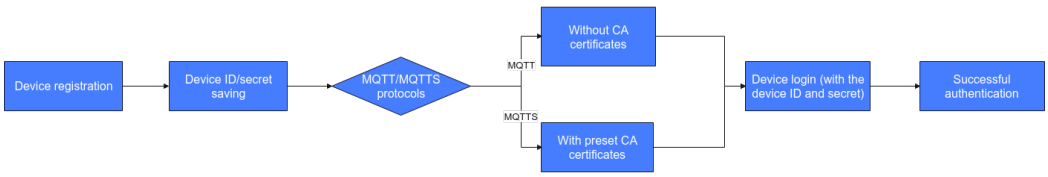
4.2.3 MQTT(S) Secret Authentication

Introduction

MQTT(S) secret authentication requires a device to have its ID and secret for access authentication. For devices connected through MQTTS, a CA certificate must be preconfigured on the devices.

Procedure

Figure 4-5 MQTT(S) secret authentication process



1. An application calls the API for creating a device to register a device. Alternatively, a user uses the IoTDA console to register a device.

   NOTE

   During registration, use the MAC address, serial number, or IMEI of the device as the node ID.

2. The platform allocates a globally unique device ID and secret to the device.

   NOTE

   The secret can be defined during device registration. If no secret is defined, the platform allocates one.

3. The device needs to integrate the preset CA certificate (only for the authentication process of MQTTS access).
4. During login, the device sends a connection request carrying the device ID and secret.
5. If the authentication is successful, the platform returns a success message, and the device is connected to the platform.

4.2.4 MQTT(S) Certificate Authentication

Introduction

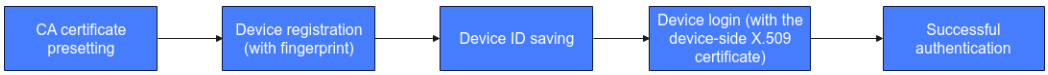
MQTT(S) certificate authentication requires you to upload a device CA certificate on the console first. Then, you can either use the API for **creating a device** or register the device on the console to get the device ID. When the device accesses the IoT platform, it carries the X.509 certificate for authentication, which is a digital certificate used to authenticate the communication entity.

Constraints

- Only MQTT devices can use X.509 certificates for identity authentication.
- You can upload up to 100 device CA certificates.

Procedure

Figure 4-6 MQTT(S) certificate authentication process



1. A user uploads a device CA certificate on the IoTDA console.
2. An application calls the API for creating a device to register a device. Alternatively, a user uses the IoTDA console to register a device.

   NOTE

   During registration, use the MAC address, serial number, or IMEI of the device as the node ID.

3. The platform allocates a globally unique device ID to the device.
4. During login, the device sends a connection request carrying the **X.509 certificate** to the platform.
5. If the authentication is successful, the platform returns a success message, and the device is connected to the platform.

APIs

- **Create a Device**

- Reset a Device Secret
- Obtain the Device CA Certificate List
- Upload a Device CA Certificate
- Delete a Device CA Certificate
- Verify a Device CA Certificate

4.2.5 MQTT(S) Custom Authentication

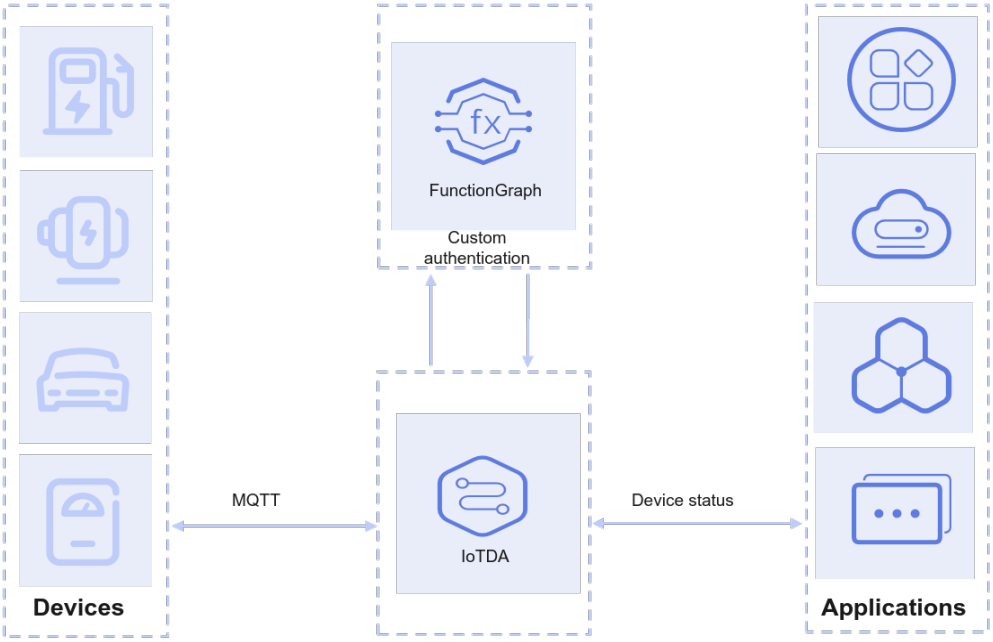
4.2.5.1 Overview

Introduction

You can use FunctionGraph to customize the identity authentication logic for devices connected to the platform.

Before connecting a device to the platform, you can use the application to configure custom authentication on the console, and then configure related functions by using **FunctionGraph**. When the device connects to the platform, the platform obtains parameters such as the device ID and custom authentication function name, and sends an authentication request to FunctionGraph. The user implements the authentication logic to complete access authentication.

Figure 1 Custom authentication architecture



Application Scenarios

- Device migration from third-party cloud platforms to IoTDA: You can configure the custom logic to make it compatible with the original authentication mode. No modification is required on the device side.
- Native access: Custom templates provide flexible authentication.

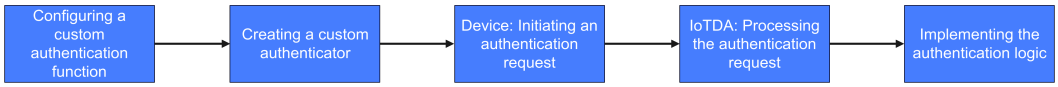
Constraints

- The device must use TLS and support **SNI (Server Name Indication)**. The SNI must carry the domain name allocated by the platform.
- By default, each user can configure up to 10 custom authenticators.
- Max. processing time: 5 seconds. If the function does not return any result within 5 seconds, the authentication fails.
- For max. TPS of authentication requests of a user, see **Specifications**. The max. TPS of custom authentication is 50% of the total authentication TPS (excluding device self-registration).
- If you have enabled the function of caching FunctionGraph authentication results, the modification takes effect only after the cache expires.
- The custom authentication mode is preferentially used for device access if conditions are met, for example, the custom authenticator name carried by the device is matched or a default custom authenticator has been configured.

4.2.5.2 Usage

Process

Figure 4-7 Custom authentication process



Procedure

Step 1 Use **FunctionGraph** to create a custom authentication function.

Figure 4-8 Function list - Creating a function

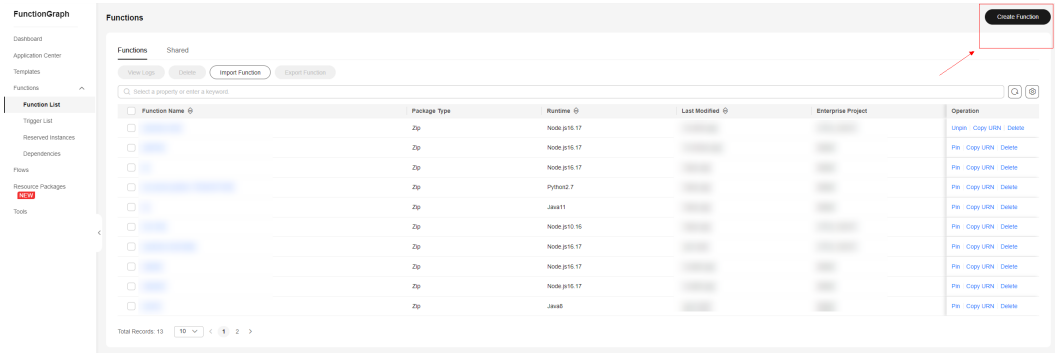
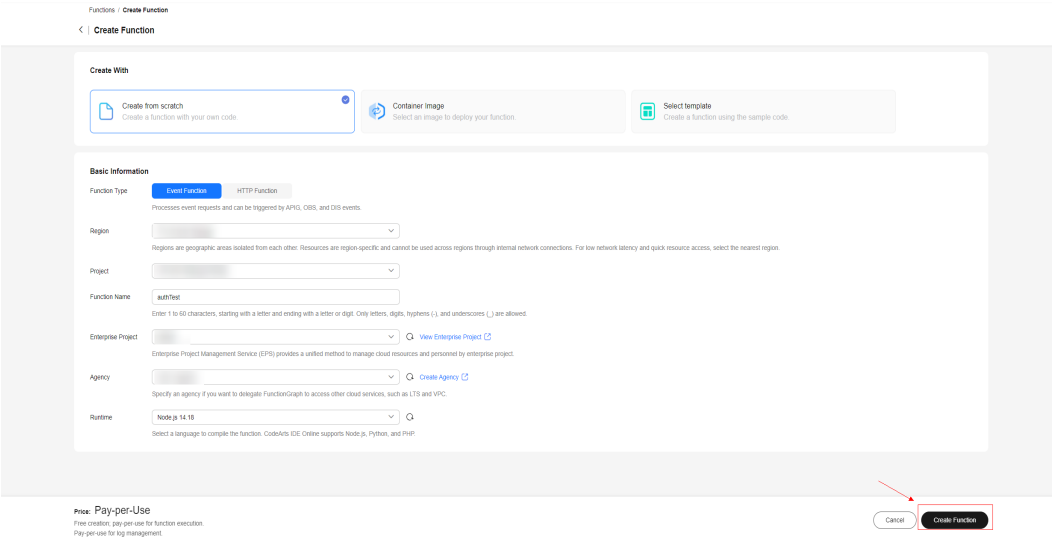


Figure 4-9 Creating a function - Parameters



Step 2 Configure custom authentication on the console for storage, management, and maintenance. You can configure up to 10 custom authenticators and choose one as the default.

Figure 4-10 Custom authentication - Creating an authenticator

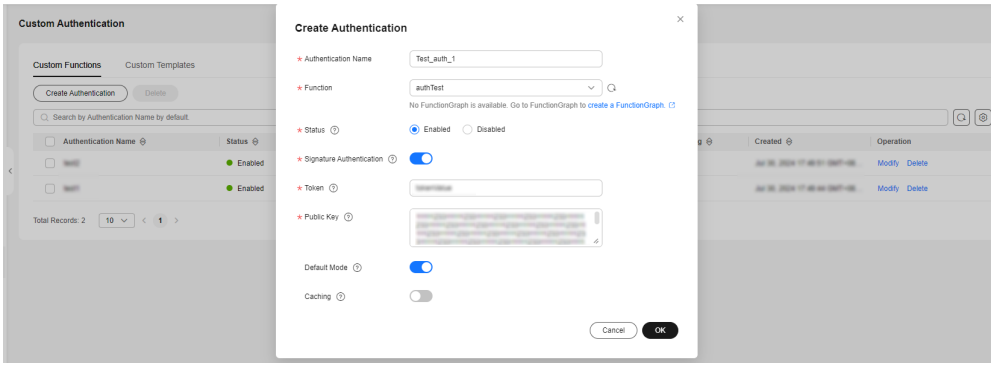


Table 4-2 Custom authentication parameters

Parameter	Mandatory	Description
Authentication Name	Yes	Enter a custom authenticator name.
Function	Yes	Select the corresponding function from the list created with FunctionGraph in **Step 1**.
Status	Yes	To use an authenticator, you must first enable it as it is disabled by default.
Signature Authentication	Yes	After this function is enabled (by default), authentication information that does not meet signature requirements will be rejected to reduce invalid function calls.

Parameter	Mandatory	Description
Token	No	Token for signature authentication. Used to check whether a device's signature information is valid.
Public Key	No	Public key for signature authentication. Used to check whether a device's signature information is valid.
Default Mode	Yes	After this function is enabled (disabled by default), if the username in an authentication request does not contain the **authorizer_name** parameter, this authenticator is used.
Caching	Yes	Whether to cache FunctionGraph authentication results (disabled by default). The cache duration ranges from 300 minutes to 1 day.

Step 3 The device initiates a CONNECT request using MQTT. The request must carry the **username** parameter, which contains optional parameters related to custom authentication.

- Username format requirements: Remove braces ({}) and separate each parameter by a vertical bar (|). Do not add vertical bars (|) in the parameter content.

```
{device-identifier}|authorizer-name={authorizer-name}|authorizer-signature={token-signature}|signing-token={token-value}
Example:
659b70a0bd3f665a471e5ec9_auth|authorizer-name=Test_auth_1|authorizer-signature=***|signing-token=tokenValue
```

Table 4-3 Description of the username parameter

Parameter	Mandatory	Description
device-identifier	Yes	Device identifier. You are advised to set it to the device ID.
authorizer-name	No	Custom authenticator name, which must be the same as the configured authenticator. If this parameter is not carried, the system will use either the default custom authenticator (if configured) or the original secret/certificate authentication mode.
authorizer-signature	No	This parameter is mandatory when the signature verification function is enabled. Obtain the value by encrypting the private key and signing-token. The value must be the same as the authentication name used in **Step 2**.
signing-token	No	This parameter is mandatory when the signature verification function is enabled. The value is used for signature verification and must be the same as the token value used in **Step 2**.

- Run the following command to obtain authorizer-signature:
```
echo -n {signing-token} | openssl dgst -sha256 -sign {private key} | openssl base64
```

Table 4-4 Command parameters

Parameter	Description
echo -n {signing-token}	Run the echo command to output the value of signing-token and use the **-n** parameter to remove the newline character at the end. The value of signing-token must be the same as that of the token in **Step 2**.
openssl dgst -sha256 -sign	Hash the input data with the SHA-256 algorithm.
{private key}	Private key encrypted using the RSA algorithm. You can upload a private key file in .pem or .key format.
openssl base64	Encode the signature result using Base64 for transmission and storage.

Step 4 When receiving an authentication request, IoTDA determines whether to use the custom authentication mode based on the username parameter and related configuration.

1. The system checks whether the username carries the custom authentication name. If yes, the authenticator processing function is matched based on the name. If no, the default custom authenticator is used to match the authentication processing function. If no matching is found, the original key/certificate authentication mode is used.
2. The system checks whether signature verification is enabled. If yes, the system checks whether the signature information carried in the username can be verified. If the verification fails, an authentication failure message is returned.
3. After the processing function is matched, the device authentication information (that is, the input parameter **event** of **Step 5**) is carried and an authentication request is sent to FunctionGraph through the function URN.

Step 5 Develop based on the processing function created with FunctionGraph in **Step 1**. The function return result must meet the following requirements:

```
exports.handler = async (event, context) => {
    console.log("username=" + event.username);
    // Enter the validation logic.


    // Returned JSON format (fixed)
    const authRes = {
        "result_code": 200,
        "result_desc": "successful",
        "refresh_seconds": 300,
        "device": {
            "device_id": "myDeviceId",
            "provision_enable": true,
            "provisioning_resource": {
                "device_name": "myDeviceName",
                "node_id": "myNodeId",
```

```
                "product_id": "myProductId",
                "app_id": "customization0000000000000000000",
                "policy_ids": ["657a4e0c2ea0cb2cd831d12a", "657a4e0c2ea0cb2cd831d12b"]
            }
        }
    }
    return JSON.stringify(authRes);
}
```

Request parameters (**event**, in JSON format) of the function:

```
{
    "username": "myUserName",
    "password": "myPassword",
    "client_id": "myClientId",
    "certificate_info": {
        "common_name": "",
        "fingerprint": "123"
    }
}
```

Table 4-5 Request parameters

Parameter	Type	Mandatory	Description
username	String	Yes	The **username** field in the MQTT CONNECT message, the format of which is the same as that of the **username** field in **Step 3**.
password	String	Yes	**password** parameter in the MQTT CONNECT message.
client_id	String	Yes	**clientId** parameter in the MQTT CONNECT message.
certificate_info	JsonObject	No	Device certificate information in the MQTT CONNECT message.

Table 4-6 certificate_info: certificate information

Parameter	Type	Mandatory	Description
common_name	String	Yes	Common name parsed from the device certificate carried by the device.
fingerprint	String	Yes	Fingerprint information parsed from the device certificate carried by the device.

Table 4-7 Returned parameter information

Parameter	Type	Mandatory	Description
result_code	Integer	Yes	Authentication result code. If 200 is returned, the authentication is successful.
result_desc	String	No	Description of the authentication result.
refresh_seconds	Integer	No	Cache duration of the authentication result, in seconds.
device	JsonObject	No	Device information when the authentication is successful. When self-registration is enabled, the platform creates a device based on the device information provided if the corresponding device ID does not exist.

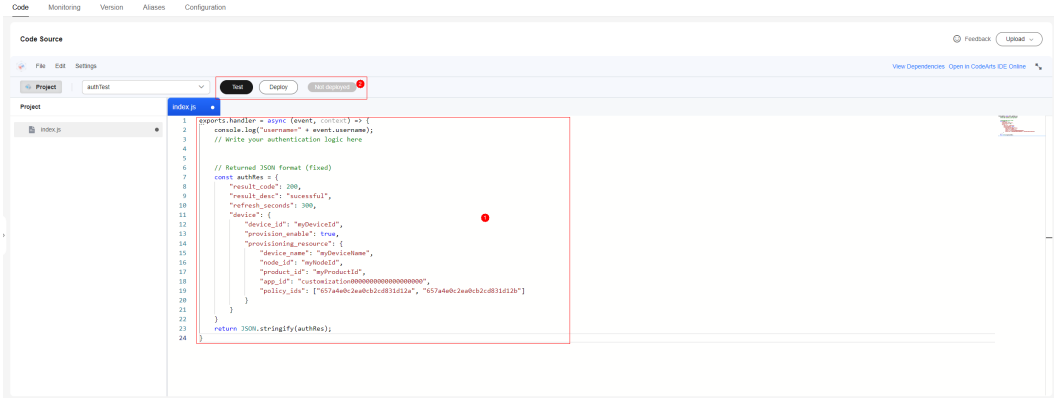
Table 4-8 Device information

Parameter	Type	Mandatory	Description
device_id	String	Yes	A globally unique device ID. This parameter is mandatory in both self-registration and non-self-registration scenarios. If this parameter is carried, the platform sets the device ID to the value of this parameter. Recommended format: *product_id* + _ + *node_id*. The value can contain up to 128 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed. You are advised to use at least 4 characters.
provision_enable	Boolean	No	Whether to enable self-registration. Default value: **false**.
provisioning_resource	JsonObject	Mandatory in the self-registration scenario	Self-registration parameters.

Table 4-9 provisioning_resource self-registration parameters

Parameter	Type	Mandatory	Description
device_name	String	No	Device name, which uniquely identifies a device in a resource space. The value can contain up to 256 characters. Only letters, digits, and special characters (_?'#().,&%@!-) are allowed. You are advised to use at least 4 characters. Min. characters: **1** Max. characters: **256**
node_id	String	Yes	Device identifier. This parameter is set to the IMEI, MAC address, or serial number. It contains 1 to 64 characters (recommended length: 4), including letters, digits, hyphens (-), and underscores (_). (Note: Information cannot be modified once it is hardcoded to NB-IoT modules. Therefore, the node ID of an NB-IoT must be globally unique.)
product_id	String	Yes	Unique ID of the product associated with the device. The value is allocated by IoTDA after the product is created. The value can contain up to 256 characters. Only letters, digits, and special characters (_?'#().,&%@!-) are allowed. You are advised to use at least 4 characters. Min. characters: **1** Max. characters: **256**
app_id	String	Yes	Resource space ID, which specifies the resource space to which the created device belongs. The value is a string of no more than 36 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed.
policy_ids	List<String>	No	Topic policy ID.

Figure 4-11 Compiling a function - Deployment



Step 6 After receiving the result, FunctionGraph checks whether the self-registration is required. If yes, FunctionGraph triggers automatic device registration. By default, all self-registered devices are authenticated using secrets, which are randomly generated. After receiving the authentication result, IoTDA proceeds with the subsequent process.

----End

4.2.6 MQTT(S) Custom Template Authentication

4.2.6.1 Overview

Introduction

In addition to **the default authentication mode,** you can also use the **internal functions** provided by the platform to flexibly orchestrate authentication modes for devices connecting to the platform.

Application Scenarios

- Device migration from third-party IoT platforms to IoTDA: You can configure a custom template to be compatible with the original authentication mode. No modification is required on the device side.
- Native access: Custom templates can support more devices.

Constraints

1. The device must use TLS and support **SNI (Server Name Indication)**. The SNI must carry the domain name allocated by the platform.
2. Max. templates: five for a user. Only one template can be enabled at a time.
3. Max. functions nested: five layers.
4. Max. content length: 4,000 characters. Chinese character not allowed.
5. When the device uses secret authentication, the template password function must contain the original secret parameter (**iotda::device:secret**).
6. The format of the template authentication parameter **username** cannot be the same as that of the custom function authentication parameter **username**. Otherwise, the custom function authentication is used. For example:

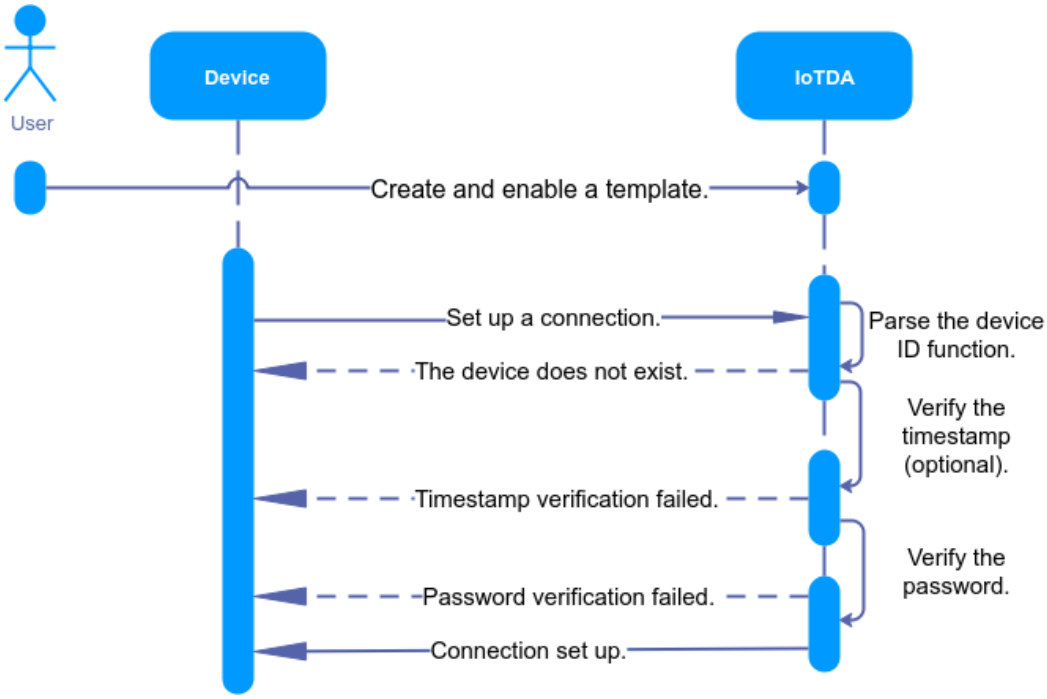
```
{deviceId}|authorizer-name={authorizer-name}|xxx
```

7. As custom authentication templates have higher priority, once you activate a custom authentication template, the platform uses the template instead of the default mode.

4.2.6.2 Usage

Process

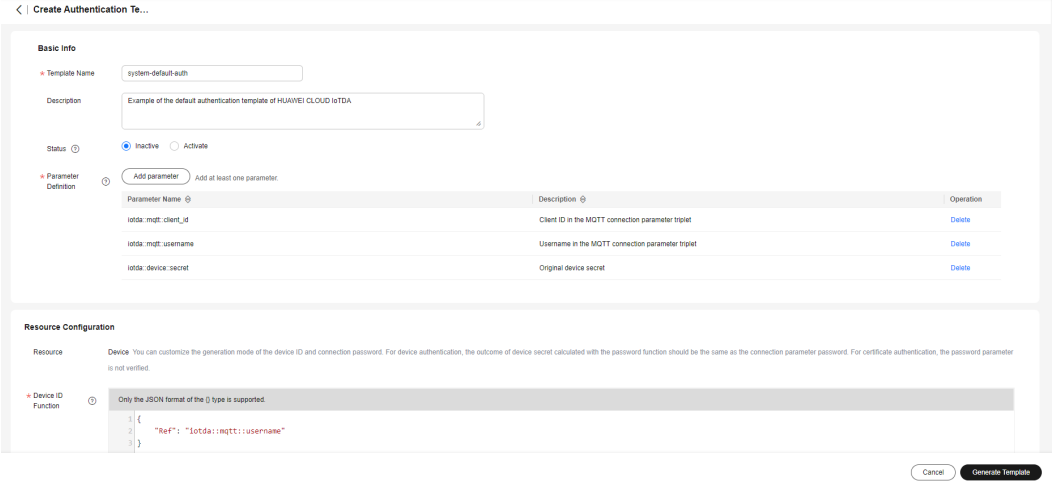
Figure 4-12 Process of authentication based on custom templates



Procedure

Step 1 Create an authentication template. Specifically, log in to the IoTDA console, in the navigation pane, choose **Devices** > **Custom Authentication**, click **Custom Template**, and click **Create Template**. The authentication template used in this example is the same as that used in the **default authentication**.

Figure 4-13 Custom authentication - Creating a template



The overall content of the template is as follows:

```
{
  "template_name": "system-default-auth",
  "description": "Example of the default authentication template of Huawei Cloud IoTDA",
  "status": "ACTIVE",
  "template_body": {
    "parameters": {
      "iotda::mqtt::client_id": {
        "type": "String"
      },
      "iotda::mqtt::username": {
        "type": "String"
      },
      "iotda::device::secret": {
        "type": "String"
      }
    },
    "resources": {
      "device_id": {
        "Ref": "iotda::mqtt::username"
      },
      "timestamp": {
        "type": "FORMAT",
        "pattern": "yyyyMMddHH",
        "value": {
          "Fn::SubStringAfter": [
            "${iotda::mqtt::client_id}",
            "_0_1_"
          ]
        }
      },
      "password": {
        "Fn::HmacSHA256": [
          "${iotda::device::secret}",
          {
            "Fn::SubStringAfter": [
              "${iotda::mqtt::client_id}",
              "_0_1_"
            ]
          }
        ]
      }
    }
  }
}
```

Table 4-10 Authentication template parameters

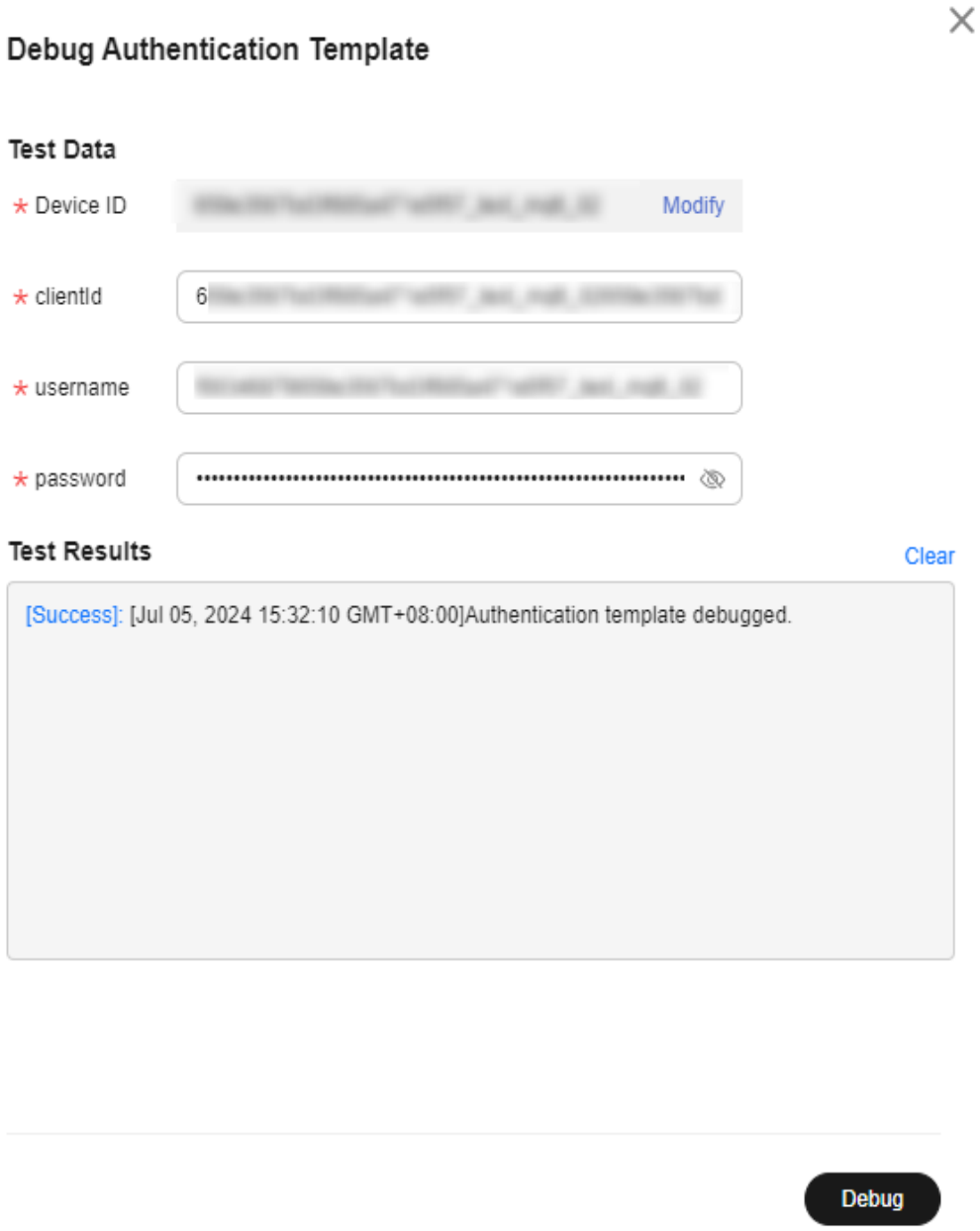
Parameter	Name	Mandatory	Description
template_name	Template name	Yes	Template name. The name must be unique for a single user. Max. length: 128 characters. Use only letters, digits, underscores (_), and hyphens (-).
description	Description	No	Template description. Max. length: 2,048 characters. Use only letters, digits, and special characters (_?'#().,&%@!-).
status	Status	No	Template status. By default, a template is not enabled. A user can only have one enabled template at a time.
parameters	Parameter	Yes	MQTT connection parameters predefined by the platform. When a device uses password authentication, the template must contain the original secret parameter (**iotda::device:secret**). The platform predefines the following parameters: **iotda::mqtt::client_id**: **Client Id** in the MQTT connection parameter triplet **iotda::mqtt::username**: **User Name** in the MQTT connection parameter triplet **iotda::certificate::country**: device certificate (country/region, C) **iotda::certificate::organization**: device certificate (organization, O) **iotda::certificate::organizational_unit**: device certificate (organization unit, OU) **iotda::certificate::distinguished_name_qualifier**: device certificate (distinguishable name qualifier, dnQualifier) **iotda::certificate::state_name**: device_certificate (province/city, ST) **iotda::certificate::common_name**: device certificate (common name, CN) **iotda::certificate::serial_number**: device certificate (serial number, serialNumber) **iotda::device::secret**: original secret of the device
device_id	Device ID function	Yes	Function for obtaining the device ID, in JSON format. The platform parses this function to obtain the corresponding device information.

Parameter	Name	Mandatory	Description
timestamp	Timestamp verification	No	Whether to verify the timestamp in the device connection information. Recommended: Enable this function if the device connection parameters (**clientId** and **username**) contain the timestamp. Verification process: The platform compares the timestamp carried by the device with the platform system time. If the timestamp plus 1 hour is less than the platform system time, the verification fails.
type	Timestamp type	No	**UNIX**: Unix timestamp. Long integer, in seconds. **FORMAT**: formatted timestamp, for example, **2024-03-28 11:47:39** or **2024/03/28 03:49:13**.
pattern	Timestamp format	No	Time format template. Mandatory when the timestamp type is **FORMAT**. **y**: year **M**: month **d**: day **H**: hour **m**: minute **s**: second **S**: millisecond Example: **yyyy-MM-dd HH:mm:ss** and **yyyy/MM/dd HH:mm:ss**
value	Timestamp function	No	Function for obtaining the timestamp when the device establishes a connection. Mandatory when timestamp verification is enabled.

Parameter	Name	Mandatory	Description
password	MQTT password function	No	Password function. Mandatory when the device authentication type is secret authentication. The template parameters must contain the original device secret parameter (**iotda::device:secret**). For details about the device authentication type, see **Registering an Individual Device**. Verification process: The platform uses parameters such as the original secret of the device in the function to calculate. If the result is the same as the **password** carried in the connection establishment request, the authentication is successful. Otherwise, the authentication fails.

Step 2 Select a device debugging template. Specifically, click **Debug**, select a device for debugging, enter MQTT connection parameters, and click **Debug** to view the result. Note: If **clientId** in the standard format is used, the platform verifies whether the value of **username** is the same as the prefix of **clientId**.

Figure 4-14 Custom template - Debugging



After the device debugging is successful, click **Enable** to enable the template. Once the template is enabled, it will be used for authentication of all devices, and the enabled template cannot be modified. You are advised to create a backup template for debugging, and switch to the backup template only when the debugging succeeds.

Step 3 Use the MQTT.fx tool to simulate device connection setup. Set **Broker Address** to **the platform access address**, choose **Overview** > **Access Information**, and set port to **8883**.

Figure 4-15 Device connection establishment

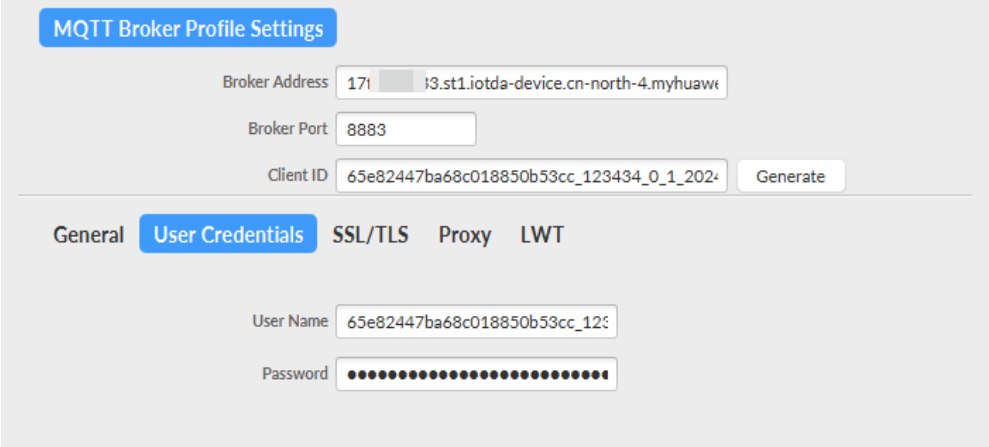
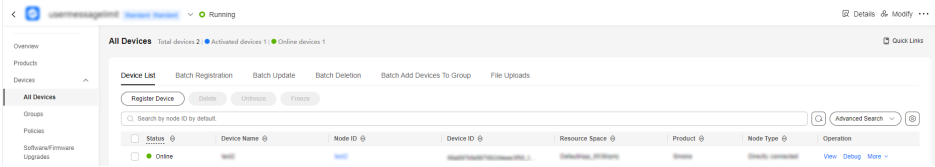


Figure 4-16 Device list - Device online status



----End

4.2.6.3 Examples

Example 1

When a certificate is used to authenticate a device, the values of **UserName** and **ClientId** are not limited. The device ID is obtained from the common name of the device certificate.

Table 4-11 Authentication parameters

Parameter	Description
Client ID	Any value
User Name	Any value
Password	Empty value

Authentication template:

```
{
    "template_name": "template1",
    "description": "template1",
    "template_body": {
        "parameters": {
            "iotda::certificate::common_name": {
                "type": "String"
            }
```

```
        },
        "resources": {
            "device_id": {
                "Ref": "iotda::certificate::common_name"
            }
        }
    }
}
```

Example 2

Device ID format: *${ProductId}_${NodeId}*

Table 4-12 Authentication parameters

Parameter	Description
Client ID	Fixed format: *${ClientId}*\|**securemode=2,signmethod=hmacsha256\|timestamp=***${timestamp}*| ● *${ClientId}* (fixed format): *${ProductId}.${NodeId}* – *${NodeId}*: device node ID – *${ProductId}*: product ID ● *${timestamp}*: Unix timestamp, in milliseconds
User Name	Fixed format: *${NodeId}***&***${ProductId}*
Password	Result value after encrypting the combination of device parameter and parameter value, with the device password as the key and HMAC-SHA256 algorithm as the tool. Encryption string format: **clientId***${clientId}***deviceName***${nodeId}***productKey***${productId}***timestamp***${timestamp}* ● *${ClientId}* (fixed format): ${ProductId}.${NodeId}. ● *${NodeId}*: device node ID ● *${ProductId}*: product ID ● *${timestamp}*: timestamp

Authentication template:

```
{
    "template_name": "template2",
    "description": "template2",
    "template_body": {
        "parameters": {
            "iotda::mqtt::client_id": {
                "type": "String"
            },
            "iotda::mqtt::username": {
                "type": "String"
            },
            "iotda::device::secret": {
                "type": "String"
            }
        },
        "resources": {
```

```
        "device_id": {
            "Fn::Join": [{
                "Fn::SplitSelect": [
                    "${iotda::mqtt::username}",
                    "&",
                    1
                ]
            }, "_", {
                "Fn::SplitSelect": [
                    "${iotda::mqtt::username}",
                    "&",
                    0
                ]
            }]
        },
        "timestamp": {
            "type": "UNIX",
            "value": {
                "Fn::MathDiv": [{
                    "Fn::ParseLong": {
                        "Fn::SplitSelect": [{
                            "Fn::SplitSelect": ["${iotda::mqtt::client_id}", "|", 2]
                        }, "=", 1]
                    }
                }, 1000]
            }
        },
        "password": {
            "Fn::HmacSHA256": [{
                    "Fn::Sub": [
                        "clientId${clientId}deviceName${deviceName}productKey${productKey}timestamp$
{timestamp}",
                        {
                            "clientId": {
                                "Fn::SplitSelect": [
                                    "${iotda::mqtt::client_id}",
                                    "|",
                                    0
                                ]
                            },
                            "deviceName": {
                                "Fn::SplitSelect": [
                                    "${iotda::mqtt::username}",
                                    "&",
                                    0
                                ]
                            },
                            "productKey": {
                                "Fn::SplitSelect": [
                                    "${iotda::mqtt::username}",
                                    "&",
                                    1
                                ]
                            },
                            "timestamp": {
                                "Fn::SplitSelect": [{
                                    "Fn::SplitSelect": ["${iotda::mqtt::client_id}", "|", 2]
                                }, "=", 1]
                            }
                        }
                    ]
                },
                "${iotda::device::secret}"
            ]
        }
    }
  }
}
```

Example 3

Device ID format: *${productId}${nodeId}*

Table 4-13 Parameter

Parameter	Description
Client ID	Fixed format: *${productId}${nodeId}* ● *${productId}*: product ID ● *${nodeId}*: node ID
User Name	Fixed format: *${productId}${nodeId}*;**12010126;***${connid};${expiry}* ● *${productId}*: product ID ● *${nodeId}*: node ID ● *${connid}*: random string ● *${expiry}*: Unix timestamp, in seconds
Password	Fixed format: *${token}*;**hmacsha256** ● *${token}*: result value after encrypting the **User Name** field, with the HMAC-SHA256 algorithm as the tool and the Base64-decoded device password as the key.

Authentication template:

```
{
    "template_name": "template3",
    "description": "template3",
    "template_body": {
        "parameters": {
            "iotda::mqtt::client_id": {
                "type": "String"
            },
            "iotda::mqtt::username": {
                "type": "String"
            },
            "iotda::device::secret": {
                "type": "String"
            }
        },
        "resources": {
            "device_id": {
                "Ref": "iotda::mqtt::client_id"
            },
            "timestamp": {
                "type": "UNIX",
                "value": {
                    "Fn::ParseLong": {
                        "Fn::SplitSelect": ["${iotda::mqtt::username}", ";", 3]
                    }
                }
            },
            "password": {
                "Fn::Sub": [
                    "${token};hmacsha256",
```

```
                {
                    "token": {
                        "Fn::HmacSHA256": [
                            "${iotda::mqtt::username}",
                            {
                                "Fn::Base64Decode": "${iotda::device::secret}"
                            }
                        ]
                    }
                }
            ]
        }
    }
  }
}
```

4.2.6.4 Internal Functions

Description

Huawei Cloud IoTDA provides multiple internal functions to use in templates. This section introduces these functions, including the input parameter type, parameter length, and return value type.

NOTE

- The entire function must be in valid JSON format.
- In a function, the variable placeholders (${}) or the **Ref** function can be used to reference the value defined by the input parameter.
- The parameters used by the function must be declared in the template.
- A function with a single input parameter is followed by a parameter, for example, **"Fn::Base64Decode": "***${iotda::mqtt::username}***"**.
- A function with multiple input parameters is followed by an array, for example, **"Fn::HmacSHA256": ["***${iotda::mqtt::username}***", "***${iotda::device::secret}***"]**.
- Functions can be nested. That is, the parameter of a function can be another function. Note that the return value of a nested function must match its parameter type in the outer function, for example, **{"Fn::HmacSHA256": ["***${iotda::mqtt::username}***", {"Fn::Base64Encode": "***${iotda::device::secret}***"}]}**.

Fn::ArraySelect

The internal function **Fn::ArraySelect** returns a string element whose index is **index** in a string array.

JSON

```
{"Fn::ArraySelect": [index, [StringArray]]}
```

Table 4-14 Parameters

Parameter	Type	Description
index	int	Index of an array element. The value is an integer and starts from 0.
StringArray	String[]	String array element.
Return values	String	Element whose index is **index**.

Example:

```
{
    "Fn::ArraySelect": [1, ["123", "456", "789"]]
}
return: "456"
```

Fn::Base64Decode

The internal function **Fn::Base64Decode** decodes a string into a byte array using Base64.

JSON

```
{ "Fn::Base64Decode" : "content" }
```

Table 4-15 Parameters

Parameter	Type	Description
content	String	String to be decoded.
Return Values	byte[]	Base64-decoded byte array.

Example:

```
{
    "Fn::Base64Decode": "123456"]
}
return: d76df8e7 // The value is converted into a hexadecimal string for display.
```

Fn::Base64Encode

The internal function **Fn::Base64Encode** encodes a string using Base64.

JSON

```
{"Fn::Base64Encode": "content"}
```

Table 4-16 Parameters

Parameter	Type	Description
content	String	String to be encoded.
Return Values	String	Base64-encoded string.

Example:

```
{
    "Fn::Base64Encode": "testvalue"
}
return: "dGVzdHZhbHVl"
```


Fn::GetBytes

The internal function **Fn::GetBytes** returns a byte array encoded from a string using UTF-8.

JSON

```
{"Fn::GetBytes": "content"}
```

Table 4-17 Parameters

Parameter	Type	Description
content	String	String to be encoded.
Return Values	byte[]	Byte array converted from a string encoded using UTF-8.

Example:

```
{
    "Fn::GetBytes": "testvalue"
}
return: "7465737476616c7565" // The value is converted into a hexadecimal string for display.
```

Fn::HmacSHA256

The internal function **Fn::HmacSHA256** encrypts a string using the HMACSHA256 algorithm based on a given secret.

JSON

```
{"Fn::HmacSHA256": ["content", "secret"]}
```

Table 4-18 Parameters

Parameter	Type	Description
content	String	String to be encrypted.
secret	String or byte[]	Secret key, which can be a string or byte array.
Return Values	String	Value encrypted using the HMACSHA256 algorithm.

Example:

```
{
    "Fn::HmacSHA256": ["testvalue", "123456"]
}
return: "0f9fb47bd47449b6ffac1be951a5c18a7eff694940b1a075b973ff9054a08be3"
```

Fn::Join

The internal function **Fn::Join** can concatenate up to 10 strings into one string.

JSON

```
{"Fn::Join": ["element", "element"...]}
```

Table 4-19 Parameters

Parameter	Type	Description
element	String	String to be concatenated.
Return Values	String	String obtained by concatenating substrings.

Example:

```
{
    "Fn::Join": ["123", "456", "789"]
}
return: "123456789"
```

Fn::MathAdd

The internal function **Fn::MathAdd** performs mathematical addition on two integers.

JSON

```
{"Fn::MathAdd": [X, Y]}
```

Table 4-20 Parameters

Parameter	Type	Description
X	long	Augend.
Y	long	Addend.
Return Values	long	Sum of X and Y.

Example:

```
{
    "Fn::MathAdd": [1, 1]
}
return: 2
```

Fn::MathDiv

The internal function **Fn::MathDiv** performs a mathematical division on two integers.

JSON

```
{"Fn::MathDiv": [X, Y]}
```


Table 4-21 Parameters

Parameter	Type	Description
X	long	Dividend.
Y	long	Divisor.
Return Values	long	Value of X divided by Y.

Example:

```
{
    "Fn::MathDiv": [10, 2]
}
return: 5

{
    "Fn::MathDiv": [10, 3]
}
return: 3
```

Fn::MathMod

The internal function **Fn::MathMod** performs the mathematical modulo on two integers.

JSON

```
{"Fn::MathMod": [X, Y]}
```

Table 4-22 Parameters

Parameter	Type	Description
X	long	Dividend.
Y	long	Divisor.
Return Values	long	Residue of X modulo Y.

Example:

```
{
    "Fn::MathMod": [10, 3]
}
return: 1
```

Fn::MathMultiply

The internal function **Fn::MathMultiply** performs mathematical multiplication on two integers.

JSON

```
{"Fn::MathMultiply": [X, Y]}
```


Table 4-23 Parameters

Parameter	Type	Description
X	long	Multiplicand.
Y	long	Multiplier.
Return Values	long	Value of X multiplied by Y.

Example:

```
{
    "Fn::MathMultiply": [3, 3]
}
return: 9
```

Fn::MathSub

The internal function **Fn::MathSub** performs mathematical subtraction on two integers.

JSON

```
{"Fn::MathSub": [X, Y]}
```

Table 4-24 Parameters

Parameter	Type	Description
X	long	Minuend.
Y	long	Subtrahend.
Return Values	long	Value of X minus Y.

Example:

```
{
    "Fn::MathSub": [9, 3]
}
return: 6
```

Fn::ParseLong

The internal function **Fn::ParseLong** can convert a numeric string into an integer.

JSON

```
{"Fn::ParseLong": "String"}
```


Table 4-25 Parameters

Parameter	Type	Description
String	String	String to be converted.
Return Values	long	Value obtained after a string is converted into an integer.

Example:

```
{
    "Fn::ParseLong": "123"
}
return: 123
```

Fn::Split

The internal function **Fn::Split** splits a string into a string array based on the specified separator.

JSON

```
{ "Fn::Split" : ["String", "Separator"] }
```

Table 4-26 Parameters

Parameter	Type	Description
String	String	String to be split.
Separator	String	Separator.
Return Values	String[]	String array obtained after **String** is split by **Separator**.

Example:

```
{
    "Fn::Split": ["a|b|c", "|"]
}
return: ["a", "b", "c"]
```

Fn::SplitSelect

The internal function **Fn::SplitSelect** splits a string into a string array based on the specified separator, and then returns the elements of the specified index in the array.

JSON

```
{ "Fn::SplitSelect" : ["String", "Separator", index] }
```


Table 4-27 Parameters

Parameter	Type	Description
String	String	String to be split.
Separator	String	Separator.
index	int	Index value of the target element in the array, starting from 0.
Return Values	String	Substring of the specified index after a string is split by the specified separator.

Example:

```
{
    "Fn::SplitSelect": ["a|b|c", "|", 1]
}
return: "b"
```

Fn::Sub

The internal function **Fn::Sub** replaces variables in an input string with specified values. You can use this function in a template to construct a dynamic string.

JSON

```
{ "Fn::Sub" : [ "String", { "Var1Name": Var1Value, "Var2Name": Var2Value } ] }
```

Table 4-28 Parameters

Parameter	Type	Description
String	String	A string that contains variables. Variables are defined using placeholders (${}).
VarName	String	Variable name, which must be defined in the **String** parameter.
VarValue	String	Variable value. Function nesting is supported.
Return Values	String	Value of string after replacement in the original **String** parameter

Example:

```
{
    "Fn::Sub": ["${token};hmacsha256", {
        "token": {
            "Fn::HmacSHA256": ["${iotda::mqtt::username}", {
                "Fn::Base64Decode": "${iotda::mqtt::client_id}"
            }]
        }
```

```
    }]
}
If:
${iotda::mqtt::username}="test_device_username"
${iotda::device::client_id}="OozqTPlCWTTJjEH/5s+T6w=="
return: "0773c4fd6c92902a1b2f4a45fdcdec416b6fc2bc6585200b496e460e2ef31c3d"
```

Fn::SubStringAfter

The internal function **Fn::SubStringAfter** truncates the substring after the specified separator in a string.

JSON

```
{ "Fn::SubStringAfter" : ["content", "separator"] }
```

Table 4-29 Parameters

Parameter	Type	Description
content	String	String to be truncated.
separator	String	Separator.
Return Values	String	Substring after the specified separator that separates the string.

Example:

```
{
    "Fn::SubStringAfter": ["content:123456", ":"]
]
return: "123456"
```

Fn::SubStringBefore

The internal function **Fn::SubStringBefore** truncates the substring before the specified separator in a string.

JSON

```
{ "Fn::SubStringBefore" : ["content", "separator"] }
```

Table 4-30 Parameters

Parameter	Type	Description
content	String	String to be truncated.
separator	String	Separator.
Return Values	String	Substring before the specified separator that separates the string.

Example:

```
{
    "Fn::SubStringBefore": ["content:123456", ":"]
]
return: "content"
```

Ref

The internal function **Ref** returns the value of the specified referenced parameter. The referenced parameter must be declared in the template.

JSON

```
{ "Ref" : "paramName" }
```

Table 4-31 Parameters

Parameter	Type	Description
paramName	String	Name of the referenced parameter.
Return Values	String	Value of the referenced parameter.

Example:

```
{
    "Ref": "iotda::mqtt::username"
}
When iotda::mqtt::username="device_123"
return: "device_123"
```

4.3 Open Protocol Access

4.3.1 LwM2M/CoAP Access

Overview

LwM2M, proposed by the Open Mobile Alliance (OMA), is a lightweight, standard, and universal IoT device management protocol that can be used to quickly deploy IoT services in client/server mode. LwM2M establishes a set of standards for IoT device management and application. It provides lightweight, compact, and secure communication interfaces and efficient data models for M2M device management and service support. IoTDA supports encrypted and non-encrypted access. Port 5684 and CoAP over DTLS are used for encrypted service data exchange and access. Port 5683 and CoAP are used for non-encrypted access. You are advised to use the encrypted access mode for security purposes.

NOTE

For details about LwM2M syntax and APIs, see **specifications**.

IoTDA supports the plain text, opaque, Core Link, TLV, and JSON encoding formats specified in the protocol. In the multi-field operation (for example, writing multiple resources), the TLV format is used by default.

Constraints

Table 4-32 Constraints

Description	Limit
Supported LwM2M version	1.1
Supported DTLS version	DTLS 1.2
Supported cryptographic algorithm suite	TLS_PSK_WITH_AES_128_CCM_8 and TLS_PSK_WITH_AES_128_CBC_SHA256
Body length	1 KB
API specifications	**Specifications**

API Calling

For details about the platform endpoint, see **Platform Connection Information**.

NOTE

Use the endpoint corresponding to CoAP (5683) or CoAPS (5684) and port 5683 (non-encrypted) or 5684 (encrypted) for device access.

4.3.2 HTTPS Access

Overview

Hypertext Transfer Protocol Secure (HTTPS) is a secure communication protocol based on HTTP and encrypted using SSL. IoTDA supports communication through HTTPS.

Constraints

Description	Limit
Supported HTTP version	HTTP 1.0 HTTP 1.1
Supported HTTPS	The platform supports only the HTTPS protocol. For details about how to download a certificate, see **Certificates**.
Supported TLS version	TLS 1.2
Body length	1 MB
API specifications	**Specifications**

Description	Limit
Number of child devices of which properties can be reported by a gateway at a time	50

API Calling

For details about the platform endpoint, see **Platform Connection Information**.

NOTE

Use the endpoint of IoTDA and the HTTPS port number 443.

Communication Between HTTPS Devices and the Platform

When a device connects to the platform through HTTPS, HTTPS APIs are used for their communication. These APIs can be used for device authentication as well as message and property reporting.

Message Type	Description
Device authentication	Devices obtain access tokens.
Device reporting properties	Devices report property data to IoTDA in the format defined in the product model.
Device reporting messages	Devices report custom data to IoTDA, which then forwards reported messages to an application or other Huawei Cloud services for storage and processing.
Gateway reporting device properties in batches	A gateway reports property data of multiple child devices to the platform.

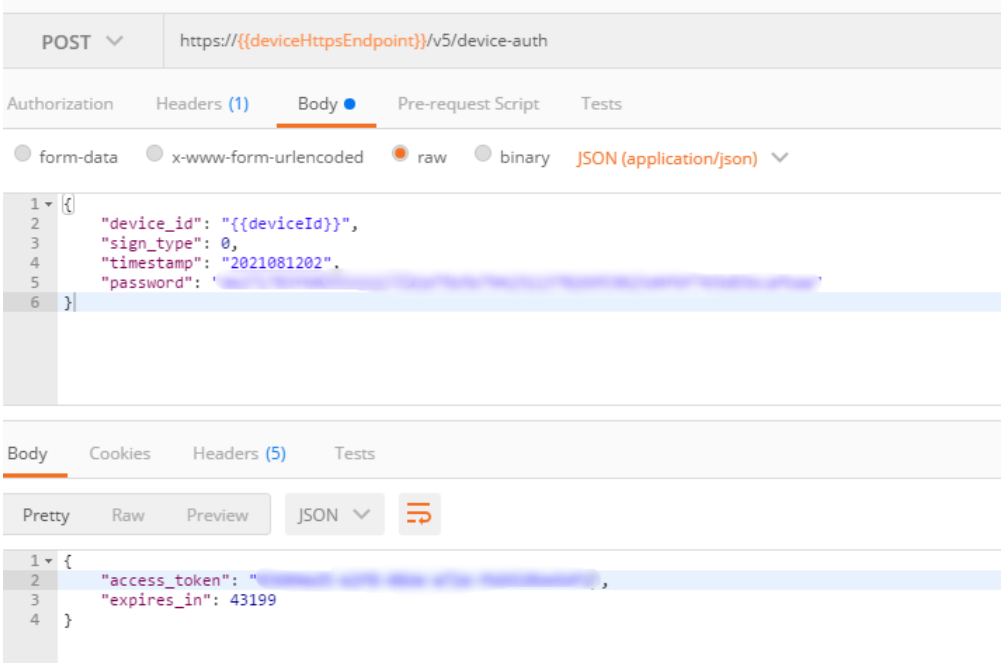
Service Flow

Step 1 Create a product on the IoTDA console or by calling the API for **creating a product**.

Step 2 Register a device on the **IoTDA console** or calling the API for **creating a device**.

Step 3 Call the device authentication API to obtain the access token of the device.

Figure 4-17 Obtaining the access token



Step 4 The obtained access token can be used by devices to report messages and properties. The access token is in the message header. The following uses property reporting as an example.

Figure 4-18 Reporting properties

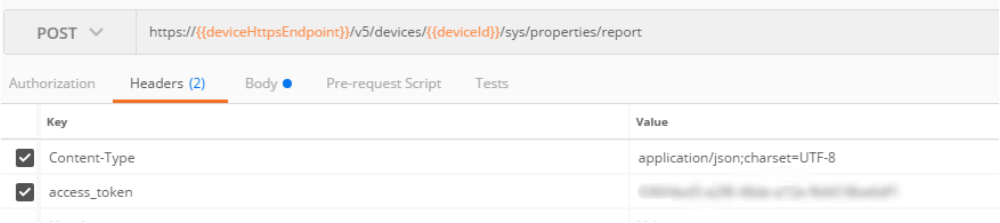
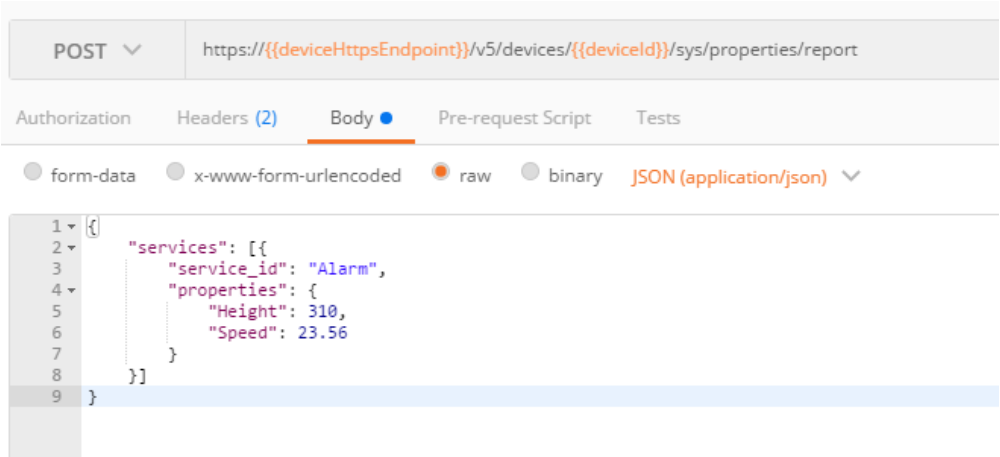


Figure 4-19 Reporting properties



----End

HTTP APIs

The following table describes the platform APIs.

API Category	API	API	Description
Device authentication API	**Device Authentication API**	/v5/device-auth	This API is used to authenticate a device. Connections can be established between devices and IoTDA after successful authentication. After the authentication is successful, IoTDA returns an access token. An access token is required when APIs for property reporting and message reporting are called. If an access token expires, you need to authenticate the device again to obtain an access token. If you obtain a new access token before the old one expires, the old access token will be valid for 30 seconds before expiration.
Device message reporting API	**Device Message Reporting**	/v5/devices/{device_id}/sys/messages/up	This API is used by a device to report custom data to IoTDA, which then forwards reported messages to an application or other Huawei Cloud services for storage and processing.
Device property reporting APIs	**Device Property Reporting**	/v5/devices/{device_id}/sys/properties/report	This API is used by a device to report property data in the format defined in the product model to IoTDA.
	Gateway Reporting Child Device Property	/v5/devices/{device_id}/sys/gateway/sub-devices/properties/report	This API is used to report device data in batches to IoTDA. A gateway can use this API to report the property data of a maximum of 50 child devices at the same time.

Device Authentication API

This API is used to authenticate a device. After successful authentication, connections can be established between devices and IoTDA, and the platform

returns an access token. The access token is required when APIs for property reporting and message reporting are called. If an access token expires, you need to authenticate the device again to obtain an access token. If you obtain a new access token before the old one expires, the old access token will be valid for 30 seconds before expiration.

Request Method	POST
URI	/v5/device-auth
Transport Protocol	HTTPS

Parameter	Mandatory	Type	Location	Description
device_id	Yes	String	Body	Device ID, which uniquely identifies a device. The value of this parameter is specified during device registration or allocated by IoTDA. If the value is allocated by the platform, the value is in the format of *[product_id]_[node_id]*. The value is a string of no more than 128 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed. Value length: 1 to 128 characters
sign_type	Yes	Integer	Body	Password verification mode. **0**: When the timestamp is verified using the HMAC-SHA256 algorithm, IoTDA does not check whether the message timestamp is consistent with the IoTDA time but only checks whether the password is correct. **1**: When the timestamp is verified using the HMAC-SHA256 algorithm, IoTDA checks whether the message timestamp is consistent with the IoTDA time and then checks whether the password is correct. Value range: 0 to 1

Parameter	Mandatory	Type	Location	Description
timestamp	Yes	String	Body	The timestamp is the UTC time when the device was connected to IoTDA, in the format of YYYYMMDDHH. For example, if the UTC time is 2018/7/24 17:56:20, the timestamp is **2018072417**. Value length: a fixed length of 10 characters
password	Yes	String	Body	The value of this parameter is the value of the device secret signed by using the HMAC-SHA256 algorithm with the timestamp as the key. For details, see **the secret generation tool**. The device secret is returned by IoTDA upon successful device registration. Value length: a fixed length of 64 characters

Parameter	Type	Description
access_token	String	Device token, which is used for device authentication. Value length: 32 to 256 characters
expires_in	Integer	Remaining validity period of the authentication information, in seconds.

Request example:

```
POST https://{endpoint}/v5/device-auth
Content-Type: application/json

{
  "device_id" : "********",
  "sign_type" : 0,
  "timestamp" : "2019120219",
  "password" : "********"
}
```

Response example:

Status Code: 200 OK

```
Content-Type: application/json

{
```

```
  "access_token" : "********",
  "expires_in" : 86399
}
```

HTTP Status Code	Description	Error Code	Error Message	Error Description
200	OK	-	-	-
400	Bad Request	IOTDA.000006	Invalid input data.	Invalid request parameters.
401	Unauthorized	IOTDA.000002	Authentication failed.	Authentication failed.
403	Forbidden	IOTDA.021101	Request reached the maximum rate limit.	The request frequency has reached the upper limit.
		IOTDA.021102	The request rate has reached the upper limit of the tenant, limit %s.	The request frequency has reached the upper limit of the tenant.

Device Message Reporting

This API is used by a device to report custom data to IoTDA, which then forwards reported messages to an application or other Huawei Cloud services for storage and processing.

Request Method	POST
URI	/v5/devices/{device_id}/sys/messages/up
Transport Protocol	HTTPS

Parameter	Mandatory	Type	Location	Description
access_token	Yes	String	Header	Access token returned after the device authentication API is called. Value length: 1 to 256 characters

Parameter	Mandatory	Type	Location	Description
device_id	Yes	String	Path	Device ID, which uniquely identifies a device. The value of this parameter is specified during device registration or allocated by IoTDA. If the value is allocated by the platform, the value is in the format of *[product_id]_[node_id]*. The value is a string of no more than 128 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed. Value length: 1 to 128 characters

NOTE

This API allows a device to use a request body to report custom data to IoTDA, which then forwards the body content to applications or other Huawei Cloud services for storage and processing. IoTDA has no specific format requirements on the body content. This API can carry data whose size is smaller than 1 MB.

Request example:

```
POST https://{endpoint}/v5/devices/{device_id}/sys/messages/up
Content-Type: application/json
access_token: ********
{
  "name" : "name",
  "id" : "id",
  "content" : "messageUp"
}
```

Response example:

Status Code: 200 ok

HTTP Status Code	Description	Error Code	Error Message	Error Description
200	OK	-	-	-
400	Bad Request	IOTDA.000006	Invalid input data.	Invalid request parameters.
401	Unauthorized	IOTDA.000002	Authentication failed.	Authentication failed.
403	Forbidden	IOTDA.000004	Invalid access token.	Invalid token.

HTTP Status Code	Description	Error Code	Error Message	Error Description
		IOTDA.021101	Request reached the maximum rate limit.	The request frequency has reached the upper limit.
		IOTDA.021102	The request rate has reached the upper limit of the tenant, limit %s.	The request frequency has reached the upper limit of the tenant.

Device Property Reporting

This API is used by a device to report property data in the format defined in the product model to IoTDA.

Request Method	POST
URI	/v5/devices/{device_id}/sys/properties/report
Transport Protocol	HTTPS

Parameter	Mandatory	Type	Location	Description
access_token	Yes	String	Header	Access token returned after the device authentication API is called. Value length: 1 to 256 characters
device_id	Yes	String	Path	Device ID, which uniquely identifies a device. The value of this parameter is specified during device registration or allocated by IoTDA. If the value is allocated by the platform, the value is in the format of *[product_id]_[node_id]*. The value is a string of no more than 128 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed. Value length: 1 to 128 characters

Parameter	Mandatory	Type	Location	Description
services	Yes	List<Table 4-33>	Body	Device service data list.

Table 4-33 ServiceProperty

Parameter	Mandatory	Type	Description
service_id	Yes	String	Service ID of the device.
properties	Yes	Object	Service properties, which are defined in the product model of the device.
event_time	No	String	UTC time when the device collects data. The value is in the format of yyyy-MM-dd'T'HH:mm:ss.SSS'Z'. If this parameter is not carried in the reported data or is in an incorrect format, the time when IoTDA receives the data is used.

Example request:

```
POST https://{endpoint}/v5/devices/{device_id}/sys/properties/report
Content-Type: application/json
access_token: ********

{
  "services" : [ {
    "service_id" : "serviceId",
    "properties" : {
      "Height" : 124,
      "Speed" : 23.24
    },
    "event_time" : "2021-08-13T10:10:10.555Z"
  } ]
}
```

Response example:

If the status code is 200, reporting is successful.

HTTP Status Code	Description	Error Code	Error Message	Error Description
200	OK	-	-	-
400	Bad Request	IOTDA.000006	Invalid input data.	Invalid request parameters.

HTTP Status Code	Description	Error Code	Error Message	Error Description
		IOTDA.021104	Subdevices in the request does not exist or does not belong to the gateway.	Some child devices in the request do not exist or do not belong to the gateway.
403	Forbidden	IOTDA.000004	Invalid access token.	Invalid token.
		IOTDA.021101	Request reached the maximum rate limit.	The request frequency has reached the upper limit.
		IOTDA.021102	The request rate has reached the upper limit of the tenant, limit %s.	The request frequency has reached the upper limit of the tenant.
		IOTDA.021105	The content reported in a single request cannot exceed 1 MB.	The content reported in a single request cannot exceed 1 MB.

Gateway Reporting Child Device Property

This API is used to report device data in batches to IoTDA. A gateway can use this API to report the property data of a maximum of 50 child devices at the same time.

Request Method	POST
URI	/v5/devices/{device_id}/sys/gateway/sub-devices/properties/report
Transport Protocol	HTTPS

Parameter	Mandatory	Type	Location	Description
access_token	Yes	String	Header	Access token returned after the device authentication API is called. Value length: 1 to 256 characters

Parameter	Mandatory	Type	Location	Description
device_id	Yes	String	Path	Device ID, which uniquely identifies a device. The value of this parameter is specified during device registration or allocated by IoTDA. If the value is allocated by the platform, the value is in the format of *[product_id]_[node_id]*. The value is a string of no more than 128 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed. Value length: 1 to 128 characters
devices	Yes	List<**Table 4-34**>	Body	Device data list. Value length: 50 characters at most

Table 4-34 DeviceProperty

Parameter	Mandatory	Type	Description
device_id	Yes	String	ID of the child device, which is unique and is allocated by IoTDA during device registration. The value is a string of no more than 128 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed.
services	Yes	List<**Table 4-35**>	Device service data list.

Table 4-35 ServiceProperty

Parameter	Mandatory	Type	Description
service_id	Yes	String	Service ID of the device.
properties	Yes	Object	Service properties, which are defined in the product model of the device.

Parameter	Mandatory	Type	Description
event_time	No	String	UTC time when the device collects data. The value is in the format of yyyy-MM-dd'T'HH:mm:ss.SSS'Z'. If this parameter is not carried in the reported data or is in an incorrect format, the time when IoTDA receives the data is used.

An example request is as follows:

```
POST https://{endpoint}/v5/devices/{device_id}/sys/gateway/sub-devices/properties/report
Content-Type: application/json
access_token: ********

{
  "devices" : [ {
    "device_id" : "deviceId_0001",
    "services" : [ {
      "service_id" : "serviceId",
      "properties" : {
        "Height" : 124,
        "Speed" : 23.24
      },
      "event_time" : "2021-08-13T10:10:10.555Z"
    } ]
  }, {
    "device_id" : "deviceId_0002",
    "services" : [ {
      "service_id" : "serviceId",
      "properties" : {
        "Height" : 124,
        "Speed" : 23.24
      },
      "event_time" : "2021-08-13T10:10:10.555Z"
    } ]
  } ]
}
```

Response example:

If the status code is 200, reporting is successful.

HTTP Status Code	Description	Error Code	Error Message	Error Description
200	OK	-	-	-
400	Bad Request	IOTDA.000006	Invalid input data.	Invalid request parameters.
		IOTDA.021104	Subdevices in the request does not exist or does not belong to the gateway.	Some child devices in the request do not exist or do not belong to the gateway.

HTTP Status Code	Description	Error Code	Error Message	Error Description
401	Unauthorized	IOTDA.000002	Authentication failed.	Authentication failed.
403	Forbidden	IOTDA.000004	Invalid access token.	Invalid token.
		IOTDA.021101	Request reached the maximum rate limit.	The request frequency has reached the upper limit.
		IOTDA.021102	The request rate has reached the upper limit of the tenant, limit %s.	The request frequency has reached the upper limit of the tenant.
		IOTDA.021103	The number of child devices in the request has reached the upper limit (%s).	The number of child devices in the request reaches the upper limit.
		IOTDA.021105	The content reported in a single request cannot exceed 1 MB.	The content reported in a single request cannot exceed 1 MB.

4.3.3 MQTT(S) Access

Overview

An MQTT message consists of fixed header, variable header, and payload.

For details on how to define the fixed header and variable header, see **MQTT standard specifications**. The payload can be defined by applications in UTF-8 format, that is, by the devices and IoT platform.

NOTE

For details about MQTT syntax and APIs, see **MQTT standard specifications**.

Common MQTT message types include CONNECT, SUBSCRIBE, and PUBLISH.

- CONNECT: A client requests a connection to a server. For details about main parameters in the payload of a CONNECT message, see **Device Connection Authentication**.
- SUBSCRIBE: A client subscribes to a topic. The main parameter **Topic name** in the payload of a SUBSCRIBE message indicates the topic whose subscriber is a device. For details, see **Topics**.
- PUBLISH: The platform publishes a message.

- The main parameter **Topic name** in the variable header of a PUBLISH message indicates the topic whose publisher is a device. For details, see **Topics**.
- The payload contains the data reported or commands delivered. It is a JSON object.

Topics

If you connect devices to the platform using MQTT, you can use topics to send and receive messages.

- Topics starting with **$oc** are preset system topics in IoTDA. You can subscribe to and publish messages through these topics. For details about the topic list and functions, see **Topics**.
- You can create topics that do not start with **$oc** to send and receive custom messages.

Constraints

Description	Limit
Number of concurrent connections to a directly connected MQTT device	1
Connection setup requests of an account per second on the device side	• Basic edition: 100 • Standard edition: See **Specifications**.
Number of upstream requests for an instance per second on the device side (when average message payload is 512 bytes)	• Basic edition: 500 • Standard edition: See **Specifications**.
Number of upstream messages for an MQTT connection	50 per second
Bandwidth of an MQTT connection (upstream messages)	1 MB (default)
Length of a publish message sent over an MQTT connection (Oversized messages will be rejected.)	1 MB
Standard MQTT protocol	MQTT v5.0, MQTT v3.1.1, and MQTT v3.1
Differences from the standard MQTT protocol	• Not supported: QoS 2 • Not supported: **will** and **retain msg**
Security levels supported by MQTT	TCP channel and TLS protocols (TLS v1, TLS v1.1, TLS v1.2, and TLS v1.3)

Description	Limit
Recommended heartbeat interval for MQTT connections	Range: 30s to 1200s; recommended: 120s
MQTT message publish and subscription	A device can only publish and subscribe to messages of its own topics.
Number of subscriptions for an MQTT connection	100
Length of a custom MQTT topic	128 bytes
Number of custom MQTT topics added to a product	10
Number of CA certificates uploaded for an account on the device side	100

Compatibility

IoTDA supports device access using **MQTT 5.0**, **MQTT 3.1.1**, and **MQTT 3.1**. However, IoTDA is not a simple MQTT broker. It also integrates capabilities such as message communications, device management, rule engine, and data forwarding. The differences between the MQTT function provided by IoTDA and standard MQTT specifications are as follows:

- Devices can communicate with IoTDA using CONNECT, CONNACK, PUBLISH, PUBACK, SUBSCRIBE, SUBACK, UNSUBSCRIBE, UNSUBACK, PINGREQ, PINGRESP, and DISCONNECT packets in MQTT specifications.
- IoTDA supports MQTT QoS 0 and QoS 1, but does not support QoS 2.
- IoTDA supports clean sessions.
- IoTDA does not support the will feature. IoTDA can push device statuses. After a device goes offline, IoTDA pushes its status to your application or other cloud services based on a forwarding rule.
- IoTDA does not support retained messages. IoTDA can cache messages during message reporting and delivery.

Supported MQTT 5.0 Features

NOTE

Only enterprise edition instances support MQTT 5.0-related features.

IoTDA supports the following new MQTT 5.0 features:

- Topic aliases. Message communication topics are reduced to an integer to reduce MQTT packets and save network bandwidth resources.
- Response topics and correlation data. The two parameters can be carried during message reporting and delivery to implement cloud HTTP-like requests and responses.
- User property list. Each property consists of a key and a value and is used to transmit property data in the non-payload area.

- Content-Type. Message reporting packets can carry Content-Type to identify the packet type.
- Return codes can be carried in CONNACK and PUBACK packets, helping devices quickly locate request statuses and issues.

TLS Support for MQTT

TLS is recommended for secure transmission between devices and the platform. Currently, TLS v1.1, v1.2, v1.3, and GMTLS are supported. TLS v1.3 is recommended. TLS v1.1 will not be supported in the future. GMTLS is supported only by the enterprise edition using Chinese cryptographic algorithms.

When TLS connections are used for the basic edition, standard edition, and enterprise edition that support general cryptographic algorithms, the IoT platform supports the following cipher suites:

- TLS_AES_256_GCM_SHA384
- TLS_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA

When the enterprise edition that supports Chinese cryptographic algorithms uses TLS connections, the IoT platform supports the following cipher suites:

- ECC_SM4_GCM_SM3
- ECC_SM4_CBC_SM3
- ECDHE_SM4_GCM_SM3
- ECDHE_SM4_CBC_SM3
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

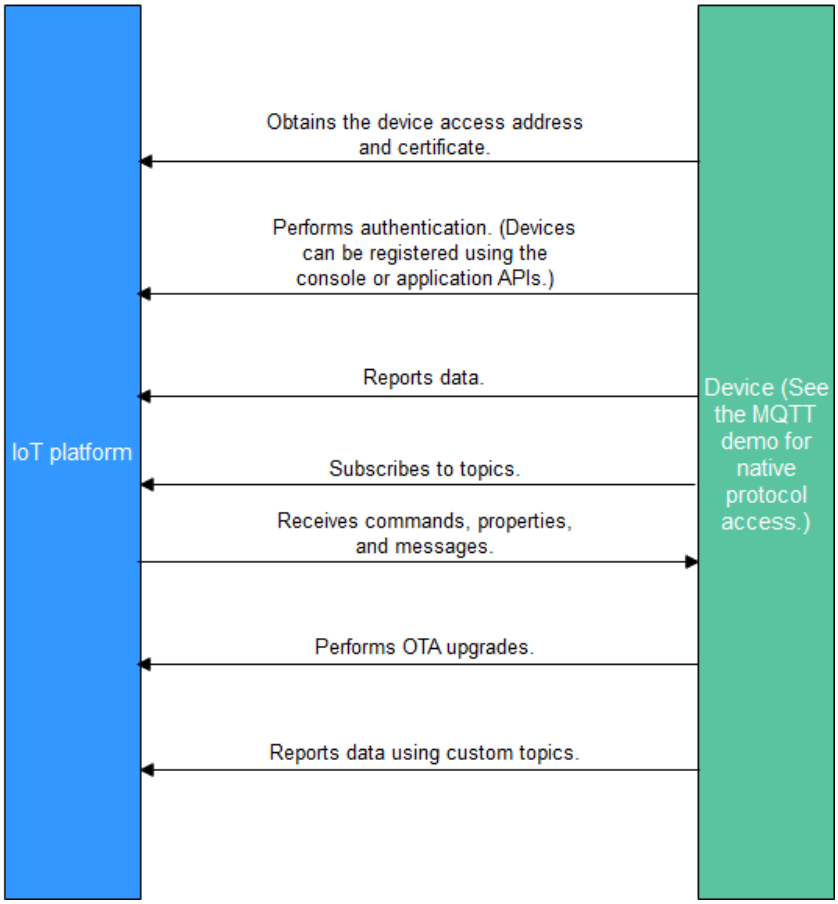
NOTE

CBC cipher suites may pose security risks.

Service Flow

MQTT devices communicate with the platform without data encryption. For security purposes, MQTTS access is recommended.

You are advised to use the **IoT Device SDK** to connect devices to the platform over MQTTS.



1. Create a product on the IoTDA console or by calling the API **Creating a Product**.
2. Register a device on the **IoTDA console** or calling the API **Creating a Device**.
3. The registered device can report messages and properties, receive commands, properties, and messages, perform OTA upgrades, and report data using custom topics. For details about preset topics of the platform, see **Topic Definition**.

NOTE

You can use MQTT.fx to debug access using the native MQTT protocol. For details, see **Developing an MQTT-based Smart Street Light Online**.

4.4 Custom Device Domain Name

Overview

A custom fully qualified domain name (FQDN) for a device to connect to IoTDA. With a custom domain name, you can manage your own server certificates, including the root certificate authority (CA), signature algorithms, and certificate lifecycles.

Scenarios

- Managing the root CAs, signature algorithms, and certificate lifecycles of server certificates.
- Disclosing the domain names to customers for branding.
- Inheriting the original domain names and server certificates during migration.

Constraints

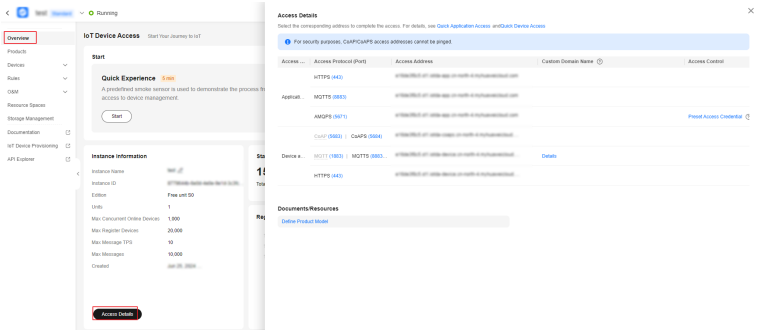
- Only the standard and enterprise editions support this function.
- Only port 8883 connected using MQTT takes effect.
- The device must use TLS and support **SNI (Server Name Indication)**. The SNI must carry the required custom domain name.
- Only one custom domain name can be configured for an IoTDA instance.

Procedure

Step 1 Configure a custom domain name.

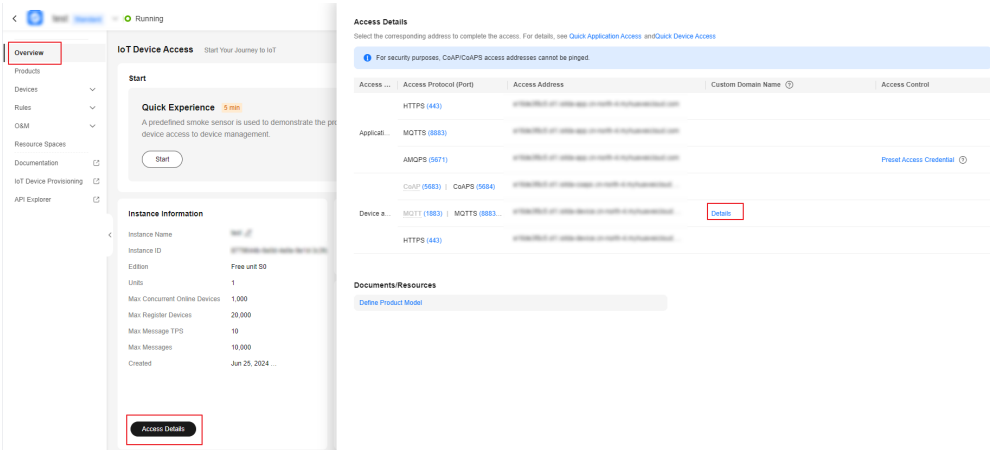
1. In the navigation pane, choose **Overview**. In the **Instance Information** area, click **Access Details**.

Figure 4-20 Obtaining access information



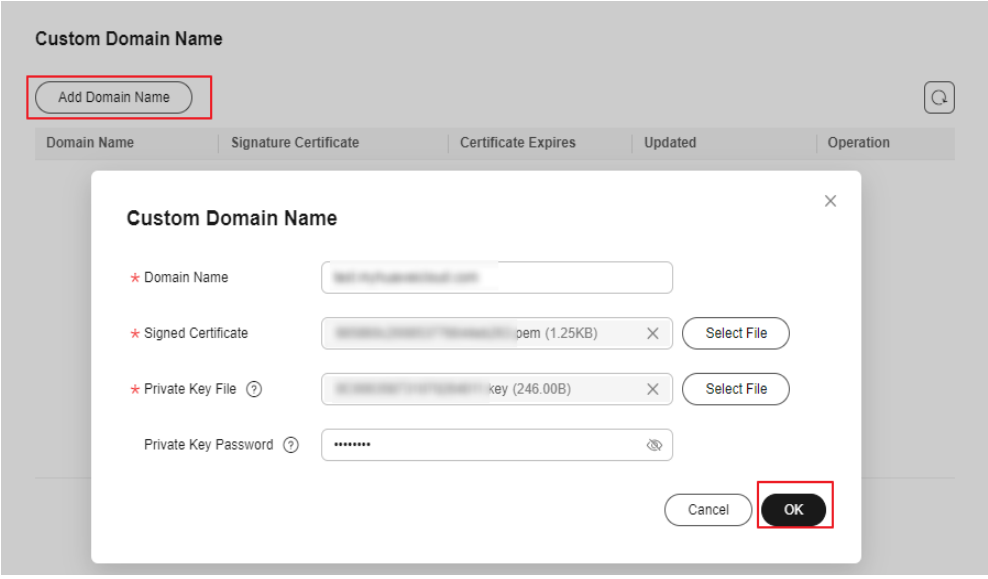
2. On the displayed page, click **Details** in the **Custom Domain Name** column.

Figure 4-21 Access information - Custom domain name details



3. On the displayed page, click **Add Domain Name**, configure parameters as prompted, and click **OK**.

Figure 4-22 Custom domain name - Configuring a custom domain name



Step 2 Create a DNS record. Contact the vendor to add domain name resolution to connect the custom domain name to the IoTDA access point. Obtain the access point by referring to **Platform Interconnection**.

----End

5 Message Communications

5.1 Data Reporting

5.1.1 Overview

Introduction

A device connected to IoTDA can send data to IoTDA in multiple ways.

Table 5-1 Data reporting

Type	Sub-type	Description	Application	Protocol	Product Model	Size
Message reporting	**Device Reporting Messages**	Devices directly report data to the cloud. The platform does not parse or store the reported data, and transparently transfers it from devices to applications.	Used for high-frequency data transmission or in scenarios where user-defined data formats are required. For example, a large amount of sensor data needs to be sent to applications in a short period of time.	MQTT and HTTP	Not required	• Max. reported message in a request: 1 MB • Max. available
	Custom Topic Communications	Devices report data with custom topics. The platform transparently transfers the reported data. Applications can subscribe to custom topics to differentiate services.	Used when devices need to report messages for various service types, or transfer data to specific topics in scenarios such as data migration.	MQTT		

Type	Sub-type	Description	Application	Protocol	Product Model	Size
						lebandwidth:10Mbit/sforstandardeditionusersand

Type	Sub-type	Description	Application	Protocol	Product Model	Size
						50Mbit/s for enterprise editions users

Type	Sub-type	Description	Application	Protocol	Product Model	Size
Property reporting	**Device Reporting Properties**	The platform does not transparently transfer the reported data from devices to applications. Instead, the platform verifies and filters data based on the defined product model. If the reported data does not comply with the product model definition, the platform discards the data.	Used when the platform needs to parse and store device data with unified models that specify the data format and value range. Alternatively, used when the platform needs to store the latest image data. For example, the switch data of street lamps needs to be sent to the application side.	MQTT, HTTP, and LwM2M over CoAP	Required	• Max. reported property in a request: 64 KB • Max. available
	Gateway Reporting Device Properties in Batches	Gateways report the properties of multiple child devices at a time. The platform does not transparently transfer the reported data from devices to applications. The reported data is distributed to the corresponding child devices by the platform.	Used when a gateway is associated with multiple child devices, and you do not have strict requirements on the data reporting time. The data of these child devices can be packaged and then reported together by the gateway.	MQTT		

Type	Sub-type	Description	Application	Protocol	Product Model	Size
						lebandwidth:10Mbit/sforstandardeditionusersand

Type	Sub-type	Description	Application	Protocol	Product Model	Size
						50Mbit/s for enterprise editions users

NOTE

- It is not suitable to report data in JSON format for devices with limited resources or with limits on bandwidth usage. In this case, devices can transparently transfer the original binary data to the platform, but a codec is required to convert binary data to JSON format. For details about how to develop codecs, see **Developing a Codec**.
- You can forward the reported data to other Huawei Cloud services for storage and processing with **data forwarding rules**. Then, the data is further processed through the consoles or APIs of the other services.

Figure 5-1 Conceptual diagram of message reporting

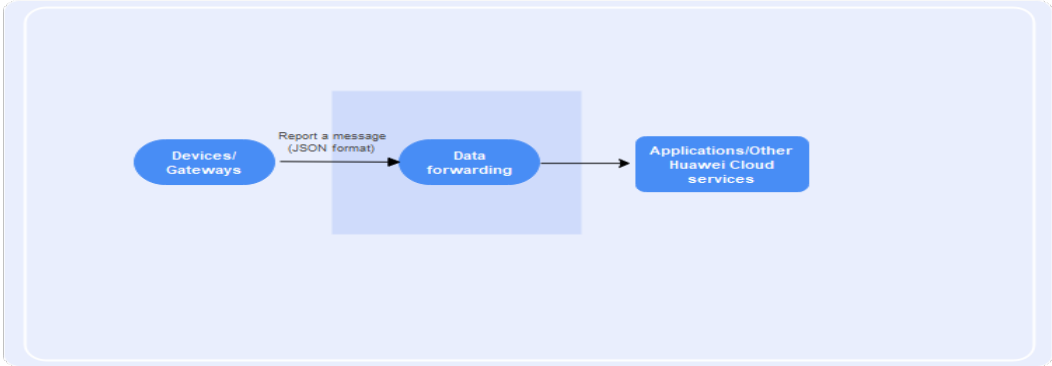


Figure 5-2 Conceptual diagram of property reporting

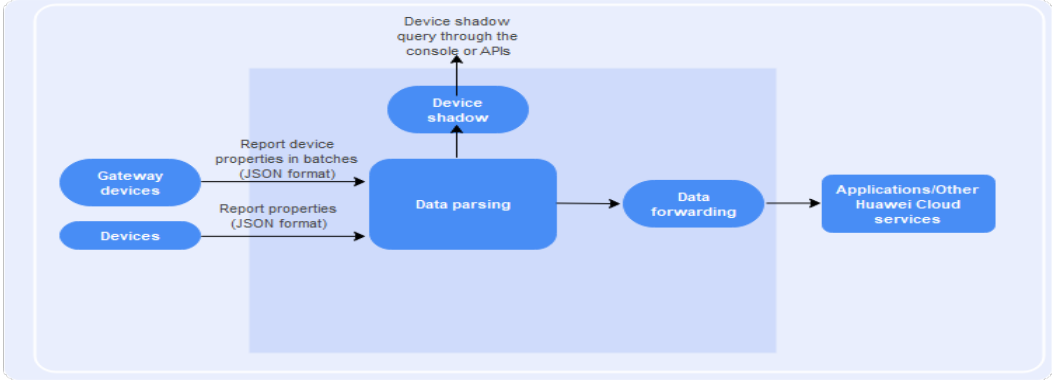
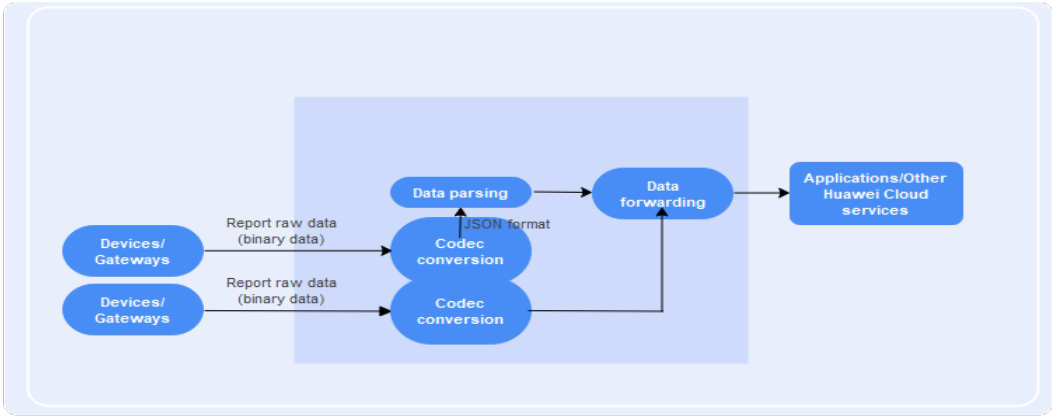


Figure 5-3 Conceptual diagram of raw binary data reporting



Application APIs

- Modify Device Properties
- Query Device Messages
- Query a Device
- Query a Device Shadow

MQTT Device APIs

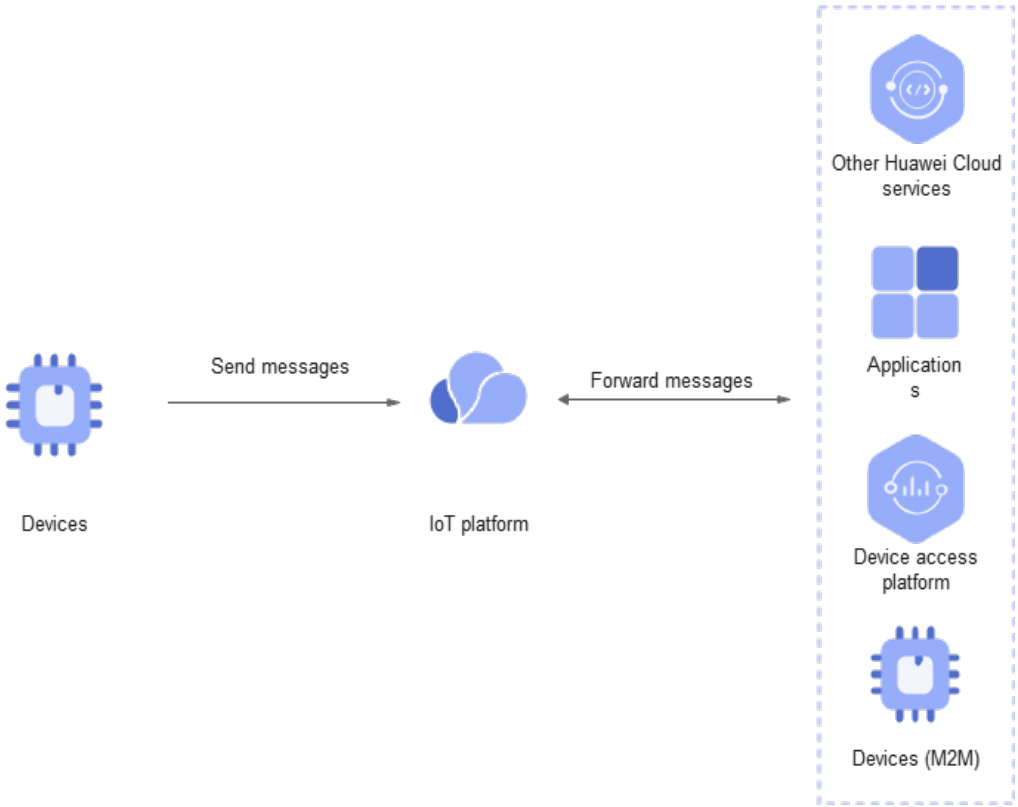
- **Device Reporting a Message**
- **Device Reporting Properties**
- **Gateway Reporting Device Properties in Batches**

5.1.2 Device Reporting Messages

Overview

Message reporting is a method by which a device directly sends data to the cloud and forwards the data to applications or other Huawei Cloud services through data forwarding. The platform does not parse or store the messages reported by devices. In this case, a product model is not required.

Figure 5-4 Process of device message reporting



Scenarios

IoTDA does not parse or store the data reported by devices. Instead, it forwards the data to other Huawei Cloud services for storage and processing based on **data forwarding rules**.

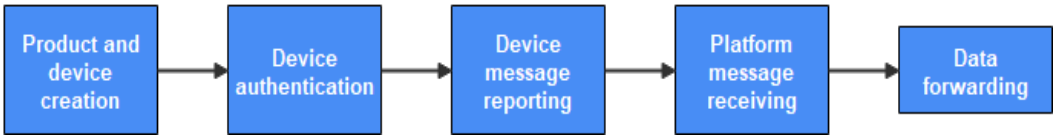
Constraints

- Max. size of a single message: 1 MB.
- Max. bandwidth of a single MQTT connection: 1 Mbit/s.

- Max. upstream messages for a single MQTT connection per second: 50 (one request is considered as one message).

Process

Figure 5-5 Process of device message reporting



Step 1 Product and device creation: For details, see **Creating a Product** and **Registering an Individual Device**.

Step 2 **Device authentication**: The platform checks whether the device has the access permission.

Step 3 Device message reporting: Devices report messages through protocols such as MQTT and HTTPS.

Use different APIs for different protocols.

MQTT: Use the message reporting APIs for **MQTT devices**.

- Example topic of MQTT message reporting:
```
$oc/devices/{device_id}/sys/messages/up
```
- Example format of MQTT message reporting:
```
{
    "content": {"hello":"123"}
}
```

HTTPS: Use the message reporting APIs for **HTTP devices**. To obtain **access_token** for HTTP devices, see **Authenticating a Device**. The following is an example of HTTPS device message reporting.

```
POST https://{endpoint}/v5/devices/{device_id}/sys/messages/up
Content-Type: application/json
access_token: ********
{
    "name": "name",
    "id": "id",
    "content": "messageUp"
}
```

NOTE

For details about devices using different protocols, see **MQTT Device Reporting a Message** and **HTTP Device Reporting a Message**.

Step 4 Data forwarding: With the **data forwarding** function, data can be forwarded to applications or other Huawei Cloud services for further processing.

----End

Message Reporting Using Java SDK

This section describes how to use Java SDKs for the development of message reporting. JDK 1.8 or later is used.

Configure the SDK on the device side:

Step 1 **Download an SDK**.

Step 2 Configure the Maven dependency of the SDK on devices.

```
<dependency>
    <groupId>com.huaweicloud</groupId>
    <artifactId>iot-device-sdk-java</artifactId>
    <version>1.1.4</version>
</dependency>
```

Step 3 Configure the SDK and device connection parameters on devices.

```
// Load the CA certificate of the IoT platform. For details about how to obtain the certificate, visit https://support.huaweicloud.com/intl/en-us/devg-iothub/iot_02_1004.html.
URL resource = BroadcastMessageSample.class.getClassLoader().getResource("ca.jks");
File file = new File(resource.getPath());

// The format is ssl://Access address:Port number.
// To obtain the access address, log in to the IoTDA console. In the navigation pane, choose Overview and click Access Details in the Instance Information area. Select the access address corresponding to port 8883.
String serverUrl = "ssl://localhost:8883";
// Device ID created on the IoT platform
String deviceId = "deviceId";
// Secret corresponding to the device ID
String deviceSecret = "secret";
// Create a device.
IoTDevice device = new IoTDevice(serverUrl, deviceId, deviceSecret, file);
if (device.init() != 0) {
    return;
}
```

Step 4 Report a device message.

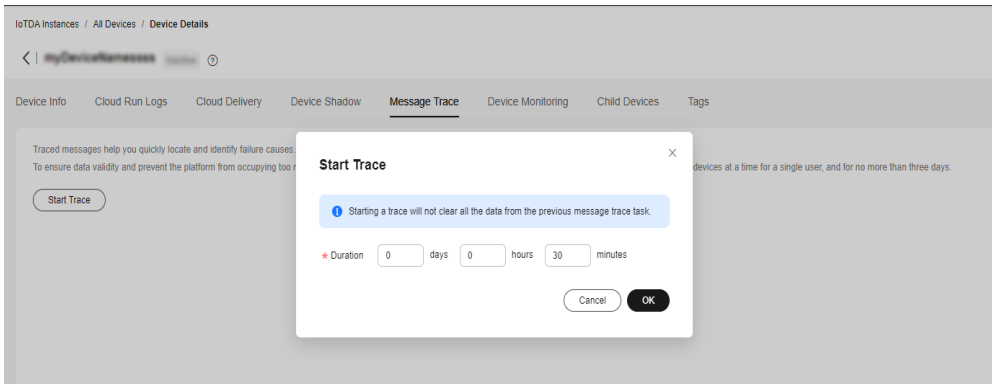
```
device.getClient().reportDeviceMessage(new DeviceMessage("hello"), new ActionListener() {
    @Override
    public void onSuccess(Object context) {
        log.info("reportDeviceMessage success: ");
    }
    @Override
    public void onFailure(Object context, Throwable var2) {
        log.error("reportDeviceMessage fail: "+var2);
    }
});
```

----End

Verify the setting:

Step 1 On the IoTDA console, choose **Devices** > **All Devices**, select a device to access its details page, and click **Start Trace** on the **Message Trace** tab page.

Figure 5-6 Message tracing - Starting message tracing

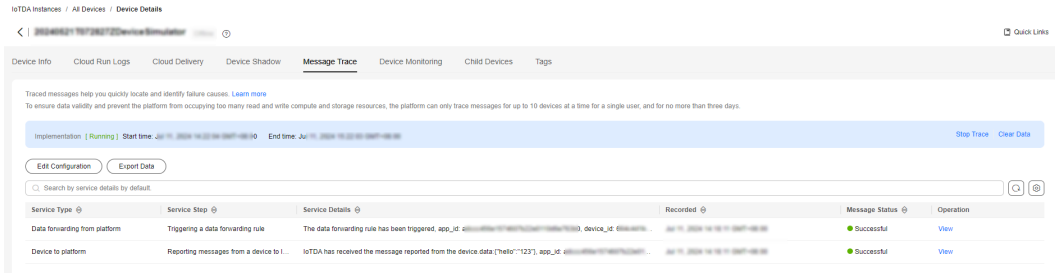


Step 2 Run the SDK code on the device. The following is an example of the log format when the device reports a message.

Figure 5-7 Java SDK message reporting result log

Step 3 Check the result on the **Message Trace** tab page. The platform has received messages from the device and the data forwarding rule has been triggered.

Figure 5-8 Message tracing - Message reporting triggering a forwarding rule



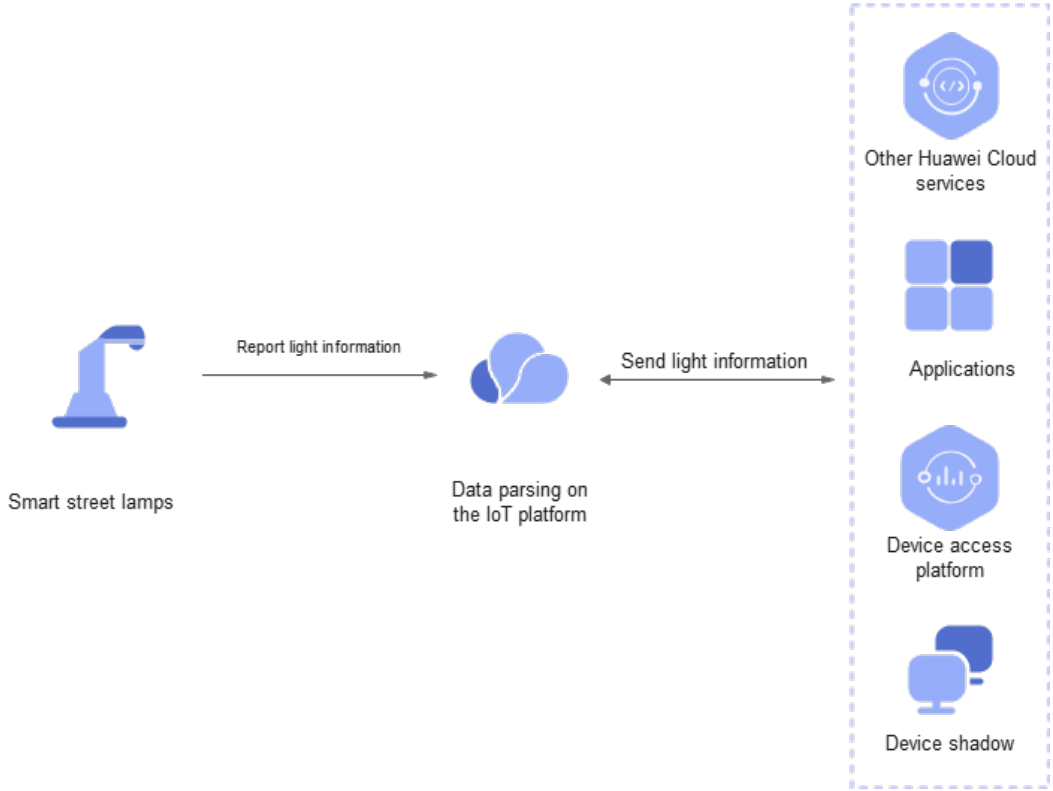
----End

5.1.3 Device Reporting Properties

Overview

Property reporting is a method by which IoTDA parses, caches, and forwards data to applications or other Huawei Cloud services through data forwarding. Product models need to be established on the platform. The platform records the latest reported property value and stores the data that complies with the product model definition. The device can obtain the latest device properties from the platform through **device shadow**.

Figure 5-9 Device Reporting Properties



Scenarios

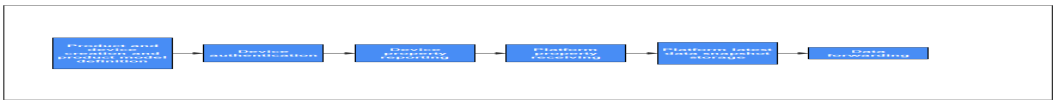
- Data of devices and applications needs to be converted, managed, and cached by the platform.
- Data needs to be forwarded to other Huawei Cloud services for storage and processing based on **data forwarding rules**.

Constraints

- Max. size of a single message: 64 KB.
- A product model is required. The reported data must match the properties defined in the product model.
- Max. child devices of which properties can be reported by a gateway at a time: 100.

Process

Figure 5-10 Process of device property reporting



Step 1 Product and device creation and product model definition: For details, see **Creating a Product**, **Registering an Individual Device**, and **Product Model Definition**.

Step 2 **Device authentication**: The platform checks whether the device has the access permission.

Step 3 Device property reporting: Devices report property data using protocols such as MQTT, HTTP, and LwM2M.

Use different APIs for different protocols.

- **MQTT**: Use the property reporting APIs for **MQTT devices**. The following is an example of reporting MQTT device property.

```
Topic: $oc/devices/{device_id}/sys/properties/report
Data format example:
{
    "services": [
        {
            "service_id": "Temperature",
            "properties": {
                "value": 57,
                "value2": 60
            }
        }
    ]
}
```

- **HTTPS**: Use the property reporting APIs for **HTTP devices**. To obtain **access_token** for HTTP devices, see **Authenticating a Device**. The following is an example of reporting HTTPS device property.

```
POST https://{endpoint}/v5/devices/{device_id}/sys/properties/report
Content-Type: application/json
access_token: ********
{
    "services": [
        {
            "service_id": "serviceId",
            "properties": {
                "Height": 124,
                "Speed": 23.24
            }
        }
    ]
}
```

- **LwM2M/CoAP**: Use the property reporting APIs for **devices using LwM2M over CoAP**. The following is an example of property reporting for devices using LwM2M over CoAP.

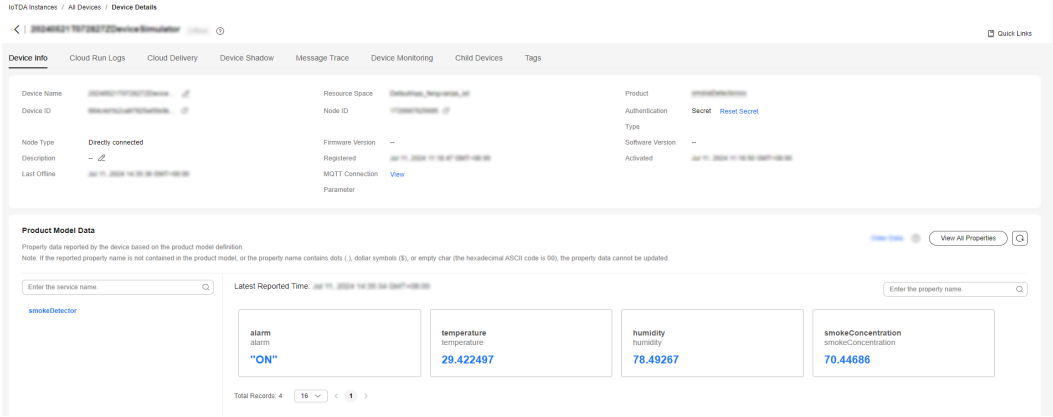
```
// Assume that the data content (value) reported by the device is c4 0d 5a 6e 96 0b c3 0e 2b 30 37.
 NON-2.05   MID=48590, Token=********, OptionSet={"Observe":22, "Content-Format":"application/
octet-stream"}, c4 0d 5a 6e 96 0b c3 0e 2b 30 37
```

NOTE

- The reported device properties must match the properties defined in the product model.
- For details about devices using different protocols, see **MQTT Device Reporting Properties**, **HTTP Device Reporting Properties**, and **LwM2M/CoAP Device Reporting Properties**.

Step 4 The platform stores the latest data snapshot. If the reported data complies with the product model definition, log in to the IoTDA console, choose **Devices** > **All Devices**, and select a device to access its details page. The latest data snapshot is displayed on the **Device Info** tab page. The following figure is as an example.

Figure 5-11 Property reporting - Viewing data



Step 5 Data forwarding: With the **data forwarding** function, data can be forwarded to applications or other Huawei Cloud services for storage and processing.

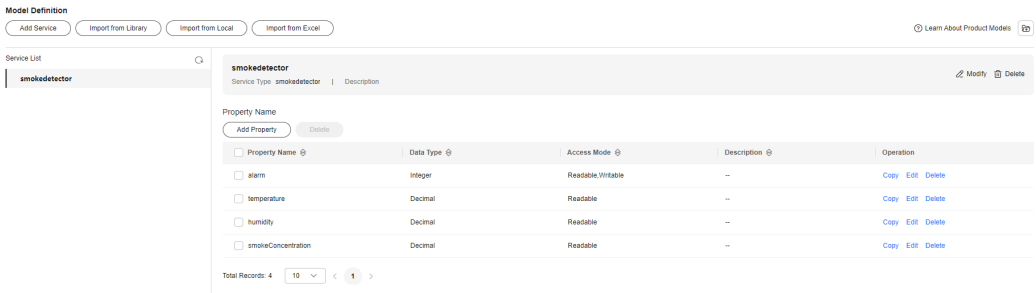
----End

Property Reporting Using Java SDK

This section describes how to use Java SDKs for the development of property reporting. JDK 1.8 or later is used.

The reported properties must match the properties defined in the **product model** corresponding to the device. The following figure provides the information of the example product model used in the SDK code.

Figure 5-12 Model definition - smokeDetector



Configure the SDK on the device side:

Step 1 **Download an SDK**.

Step 2 Configure the Maven dependency of the SDK on devices.

```
<dependency>
    <groupId>com.huaweicloud</groupId>
    <artifactId>iot-device-sdk-java</artifactId>
    <version>1.1.4</version>
</dependency>
```

Step 3 Configure the SDK and device connection parameters on devices.

```
// Load the CA certificate of the IoT platform. For details about how to obtain the certificate, visit https://support.huaweicloud.com/intl/en-us/devg-iothub/iot_02_1004.html.
URL resource = BroadcastMessageSample.class.getClassLoader().getResource("ca.jks");
File file = new File(resource.getPath());
```

```
// The format is ssl://Domain name:Port number.
// To obtain the domain name, log in to the Huawei Cloud IoTDA console. In the navigation pane, choose
Overview and click Access Details in the Instance Information area. Select the access domain name
corresponding to port 8883.
String serverUrl = "ssl://localhost:8883";
// Device ID created on the IoT platform
String deviceId = "deviceId";
// Secret corresponding to the device ID
String deviceSecret = "secret";
// Create a device.
IoTDevice device = new IoTDevice(serverUrl, deviceId, deviceSecret, file);
if (device.init() != 0) {
    return;
}
```

Step 4 Report device properties.

```
Map<String ,Object> json = new HashMap<>();
Random rand = new Random();

// Set properties based on the product model.
json.put("alarm", alarm);
json.put("temperature", rand.nextFloat()*100.0f);
json.put("humidity", rand.nextFloat()*100.0f);
json.put("smokeConcentration", rand.nextFloat() * 100.0f);

ServiceProperty serviceProperty = new ServiceProperty();
serviceProperty.setProperties(json);
serviceProperty.setServiceId("smokeDetector");// The service ID must be consistent with that defined in the
product model.

device.getClient().reportProperties(Arrays.asList(serviceProperty), new ActionListener() {
    @Override
    public void onSuccess(Object context) {
        log.info("reportProperties success" );
    }

    @Override
    public void onFailure(Object context, Throwable var2) {
        log.error("reportProperties failed" + var2.toString());
    }
});
```

----End

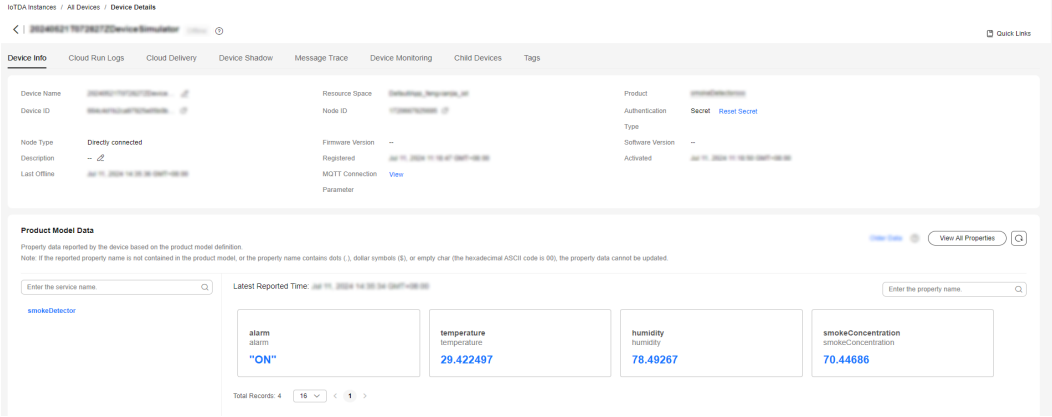
Verify the setting:

Step 1 Run the SDK code on the device. The following figure provides an example of the property reporting log.

Figure 5-13 Java SDK property reporting result log

Step 2 Log in to the IoTDA console, choose **Devices** > **All Devices**, and click a device to access its details page. The latest reported data is displayed on the **Device Info** tab page.

Figure 5-14 Property reporting - Viewing data



----End

5.2 Data Delivery

5.2.1 Overview

After a device is connected to IoTDA, the platform can send data to the device in the following ways.

Type	Description	Application	Device Shadow	Synchronous or Asynchronous	Cached by the Platform	Supported Protocol (Device)	Product Model
Message delivery	The platform directly delivers messages to devices and does not rely on product models. It is a one-way notification to devices. If a device is offline, the platform can cache messages (up to 24 hours) and send the data to the device after it goes online.	This API is used by an application to deliver a message in custom format to a device when the application cannot deliver data in the format defined in the product model. For example, sending data to devices for which no product model is defined.	Not supported	Asynchronous	Supported	MQTT	Not required
Property delivery	This mode is used to set or query device properties. After receiving the properties, the device needs to return the property execution result to the platform in a timely manner. If the device does not return a response, the platform considers that the property delivery times out.	It is used by the platform to proactively obtain or modify the device property value. For example, an app obtains the geographical location of a device at intervals.	Supported	Synchronous	Not supported	MQTT and LwM2M over CoAP	Required

Type	Description	Application	Device Shadow	Synchronous or Asynchronous	Cached by the Platform	Supported Protocol (Device)	Product Model
Command delivery	The platform delivers a device control command to a device and the device needs to respond. The response can carry parameters indicating operation success or failure. ● When the platform delivers a synchronous command, the device needs to return the command execution result within 20 seconds. Otherwise, the command delivery is considered failed. ● Asynchronous command delivery can cache messages. If a device is offline, data is sent after the device goes online. The maximum cache duration is 48 hours.	It is used for the command that needs to be confirmed immediately. For example, turning on the fan and controlling the street lamp switch.	Not supported	Synchronous	Not supported	MQTT	Required
				Asynchronous	Supported	LwM2M/CoAP	

NOTE

It is not suitable to deliver data in JSON format for devices with low configuration and limited resources or with limits on bandwidth usage. In this case, use **codecs** to convert the JSON format data into binary data on applications.

APIs for Applications

- **Deliver a Message to a Device**
- **Query Device Messages**

- **Query Device Properties**
- **Modify Device Properties**
- **Deliver a Command to a Device**
- **Deliver an Asynchronous Command**
- **Query a Command with a Specific ID**

APIs for MQTT Devices

- **Platform Delivering a Command**
- **Platform Delivering a Message**

APIs for Devices Using LwM2M over CoAP

- Platform **Delivering a Command**

5.2.2 Message Delivery

Overview

Message delivery does not rely on product models. The platform provides one-way notifications for devices and caches messages. It delivers messages from the cloud to devices in asynchronous mode (without waiting for responses from devices). If a device is offline, data is sent after the device is online. The maximum cache duration is 24 hours. By default, the platform stores a maximum of 20 messages for each device. If the number of messages exceeds 20, subsequent messages will replace the earliest messages. In addition, messages can be delivered in the format of **custom topics**.

Table 5-2 Message delivery topic type

Message Delivery Topic Type	Description
System topic	The platform predefines topics for communications with devices. For details of the topic list and functions, see **Topics**.
Custom topic	You can customize topics for device-platform communications. **Types of custom topics:** • Topics defined in the product are prefixed with **$oc/devices/{device_id}/user/**. During message reporting or delivery, the platform checks whether the topic is defined in the product. Undefined topics will be rejected by the platform. For details about how to use this type of topics, see **Using a Custom Topic for Communication**. • Topics that do not start with **$oc**, for example, **/aircondition/data/up**. This type of topics enables upstream and downstream message communications based on MQTT rules. The platform does not verify the topic permission.

Scenarios

- The data format needs to be customized and does not rely on the product model.

Constraints

- Max. size of a single message: 256 KB.
- Up to 20 messages can be cached for a single device.
- Max. length of a custom MQTT topic: 128 bytes.
- Max. cache duration (configurable): 24 hours.

Quality of Service

- IoTDA supports MQTT QoS 0 and QoS 1, but does not support QoS 2.
- If the QoS of a topic is 0, the message is delivered only once without waiting for the device to return an ACK message. If the QoS of a topic is 1, the message delivery is successful only after the device returns an ACK message.
- Devices subscribe to the system topic whose QoS is 0 by default. If the downstream system topic whose QoS is 1 is required, devices need to be configured to subscribe to the topic.
- If a device needs to subscribe to a custom topic that does not start with **$oc** and the QoS is 1, **submit a service ticket**.
- If the QoS of the subscribed topic is 1 and the platform does not receive an ACK message from the device, the platform resends the message every 2 seconds for three times by default.

  If the device still does not return an acknowledgment response and the message is still cached, the platform resends the message when the device goes online again or subscribes to a topic. By default, the platform resends the message every 10 seconds for five times.

  In addition, the mechanism of resending every 2 seconds is triggered. Therefore, the device may receive duplicate messages. It is recommended that devices have deduplication mechanisms.

APIs

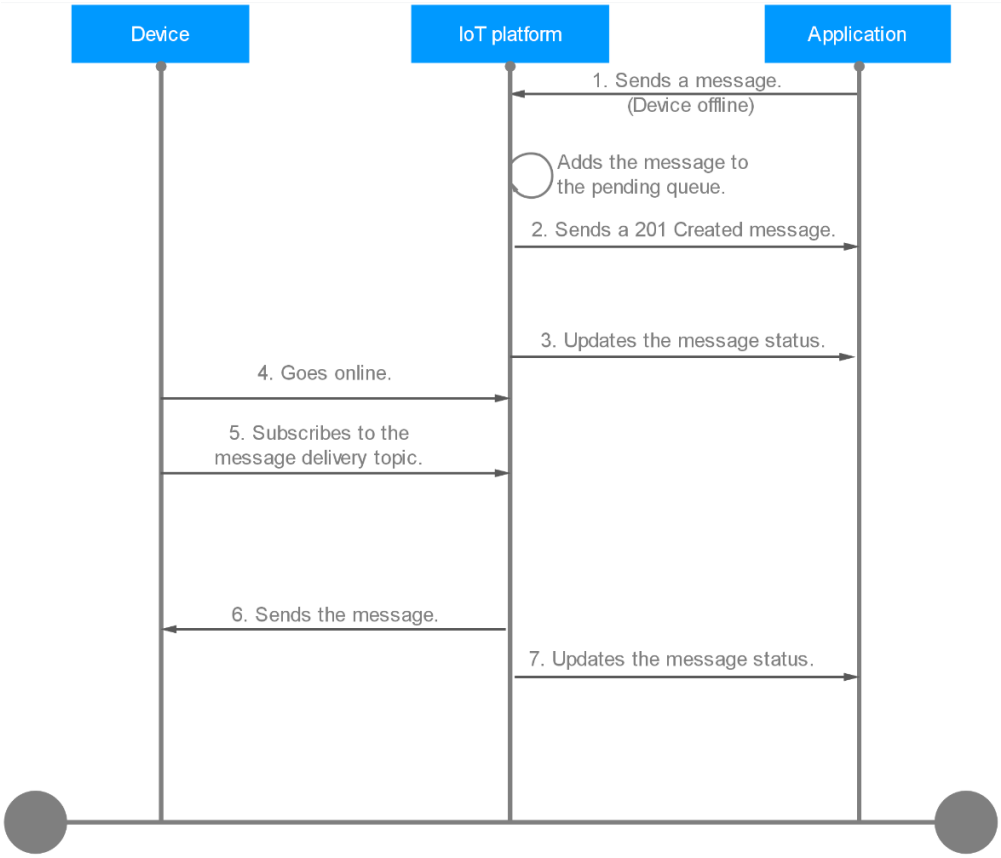
- **Deliver a Message to a Device**
- **Platform Delivering a Message**.

Delayed Message Delivery

Message delivery is a mode in which the platform directly delivers messages to devices. When a device is offline, the platform caches messages to be delivered until the device goes online.

The following describes how to use a system topic to cache and deliver messages to devices.

Figure 5-15 Process of delayed message delivery



1. An application or the third-party platform calls the API for **delivering a message to a device** to send a message to IoTDA. Example message:

```
POST https://{Endpoint}/v5/iot/{project_id}/devices/{device_id}/messages
Content-Type: application/json
X-Auth-Token: ********

{
  "message_id": "99b32da9-cd17-4cdf-a286-f6e849cbc364",
  "name": "messageName",
  "message": "HelloWorld"
}
```

2. The platform sends a 201 Created message carrying the message status **PENDING** to the application.
3. The platform pushes the message result to the application through the API for **pushing a device message status change notification**. If the device is offline, the message status is **PENDING**.

```
Topic: $oc/devices/{device_id}/sys/messages/down
Data format:
{
    "resource": "device.message.status",
    "event": "update",
    "notify_data": {
      "message_id": "string",
      "name": "string",
      "device_id": "string",
      "status": "PENDING",
      "timestamp": "string"
    }
 }
```

4. The device goes online.
5. The device subscribes to the non-system topic to receive messages. (Implicit subscription mode: Devices do not need to subscribe to downstream system topics.)
6. The platform sends the message to the device according to the protocol specifications. Example message:

```
Topic: $oc/devices/{device_id}/sys/messages/down
Data format:
{
    "object_device_id": "{object_device_id}",
    "name": "name",
    "id": "id",
    "content": "hello"
}
```

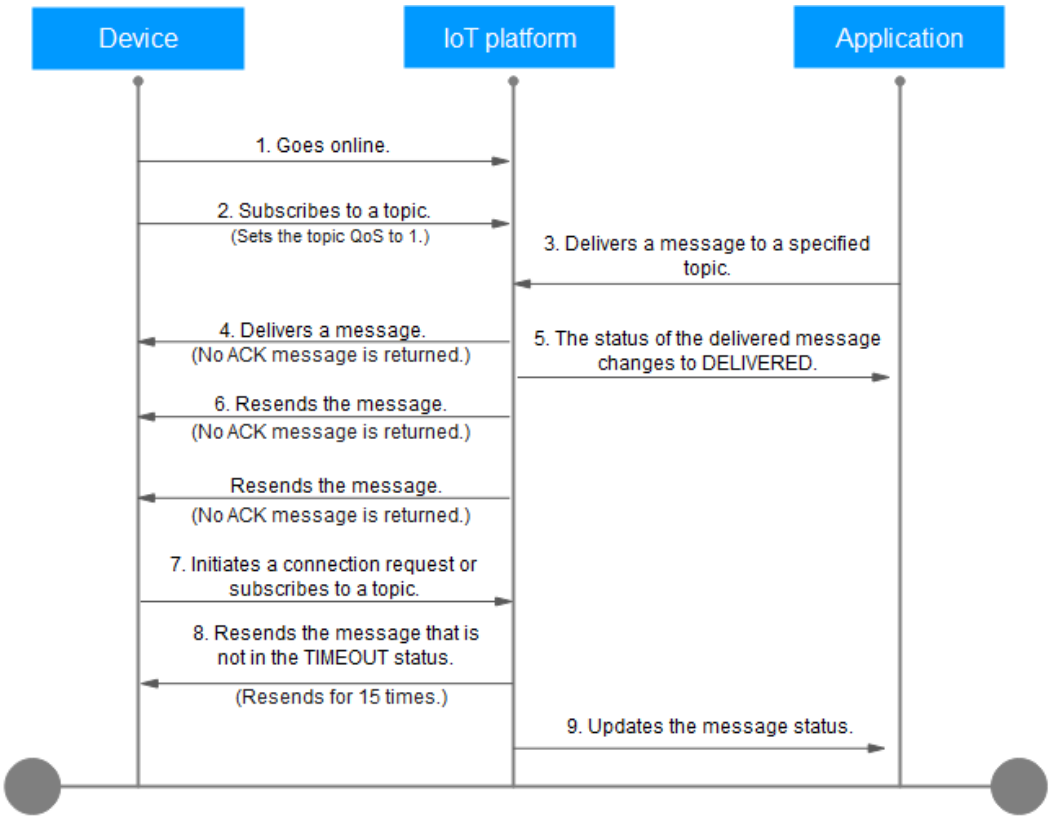
7. The platform pushes the final result of the message to the application. The message status is **DELIVERED**. For details about the used APIs, see **Push a Device Message Status Change Notification**.

```
Topic: $oc/devices/{device_id}/sys/messages/down
Data format:
{
    "resource": "device.message.status",
    "event": "update",
    "notify_data": {
      "message_id": "string",
      "name": "string",
      "device_id": "string",
      "status": "DELIVERED",
      "timestamp": "string"
     }
 }
```

Introduction for QoS 1

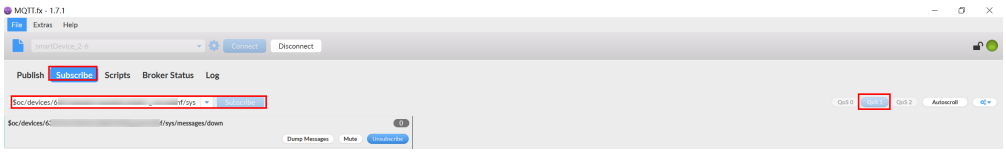
The following uses an MQTT device as an example to describe how to use a system topic whose QoS is 1 to deliver messages to devices.

Figure 5-16 Process of using QoS 1 for message delivery



1. The device goes online.
2. Subscribe to a topic for the device and set QoS to 1.

Figure 5-17 Setting QoS to 1 for the subscribed topic



3. An application or the third-party platform calls the API for **delivering a message to a device** to send a message to IoTDA. Example message:

```
POST https://{Endpoint}/v5/iot/{project_id}/devices/{device_id}/messages
Content-Type: application/json
X-Auth-Token: ********

{
  "message_id": "99b32da9-cd17-4cdf-a286-f6e849cbc364",
  "name": "messageName",
  "message": "HelloWorld"
}
```

4. The platform sends the message to the device according to the protocol specifications. An MQTT device needs to subscribe to the non-system topic to receive messages. (Implicit subscription mode: Devices do not need to subscribe to downstream system topics.) Example message:

```
Topic: $oc/devices/{device_id}/sys/messages/down
Data format:
{
    "object_device_id": "{object_device_id}",
```



```
    "name": "name",
    "id": "id",
    "content": "hello"
}
```

5. After delivering a message to the device, the platform returns a 201 Created message to the application. The message status is **DELIVERED**. Message delivery is an asynchronous operation. The platform can return the response without waiting for an ACK message from the device.
6. If the IoT platform does not receive an ACK response from the device, it resends the message every 2 seconds for three times by default.
7. The device goes online again or subscribes to a topic.
8. If the device does not return an ACK response for the previous message and the message does not time out, the platform resends the message every 10 seconds for five times by default. This mechanism of resending every 2 seconds is triggered.
9. The platform pushes the final result of the message to the application. The message status is **DELIVERED** or **TIMEOUT**. For details about the used APIs, see **Push a Device Message Status Change Notification**.

```
Topic: $oc/devices/{device_id}/sys/messages/down
Data format:
{
    "resource": "device.message.status",
    "event": "update",
    "notify_data": {
      "message_id": "string",
      "name": "string",
      "device_id": "string",
      "status": "DELIVERED",
      "timestamp": "string"
     }
 }
```

Message Delivery Status

The following figure shows the MQTT device message execution status and status change mechanism.

Figure 5-18 Device message status

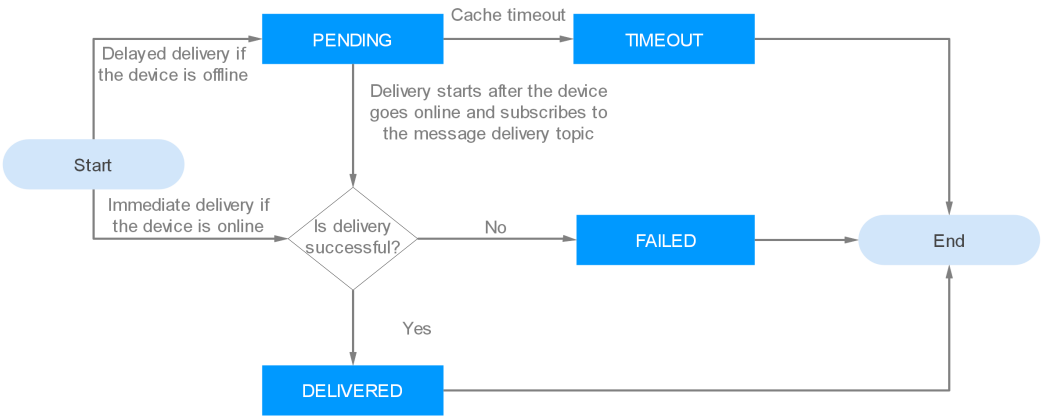


Table 5-3 Status

Status	Description
PENDING	If an MQTT device is offline, the platform caches the message. In this case, the task status is **PENDING**.
TIMEOUT	If the platform does not deliver the message in the pending status after one day, the task status changes to **TIMEOUT**.
DELIVERED	After the platform sends the message to the device, the task status changes to **DELIVERED**.
FAILED	If the platform fails to send a message to the device, the task status changes to **FAILED**.

Example of Platform Message Delivery

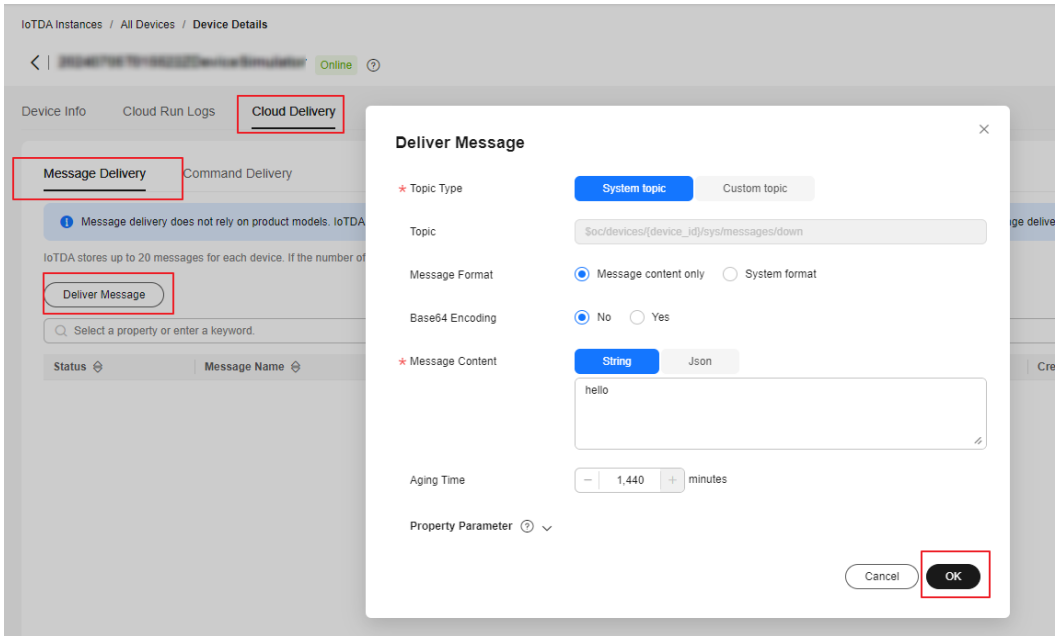
To deliver messages from the cloud, create a delivery task on the console. The following uses an MQTT device as an example to describe how to cache and deliver messages on the IoTDA console.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **All Devices**. On the device list, click a device to access its details page.

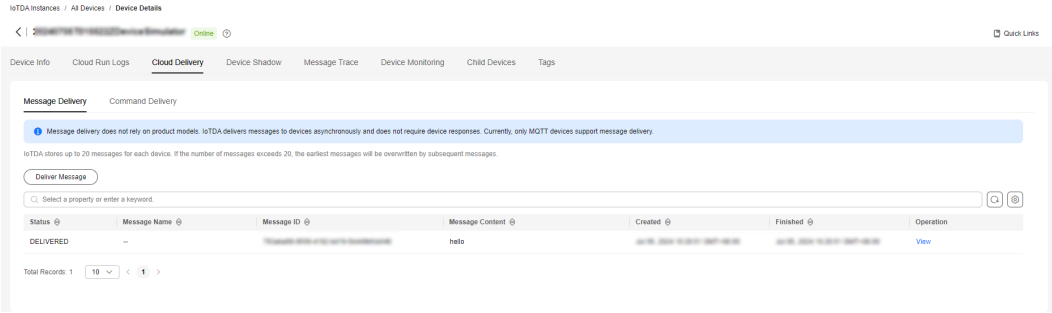
Step 3 Click the **Cloud Delivery** tab. On the **Message Delivery** tab page, click **Deliver Message**. In the displayed dialog box, configure the content and the parameters for the command to deliver.

Figure 5-19 Message delivery - MQTT



Step 4 The delivery status is **DELIVERED** on the platform.

Figure 5-20 Querying results



----End

Configure the Java SDK on the application side:

Step 1 Configure the Maven dependency. In this example, the development environment is JDK 1.8 or later. **Download an SDK**.

```
<dependency>
    <groupId>com.huaweicloud.sdk</groupId>
    <artifactId>huaweicloud-sdk-core</artifactId>
    <version>[3.0.40-rc, 3.2.0)</version>
</dependency>
<dependency>
    <groupId>com.huaweicloud.sdk</groupId>
    <artifactId>huaweicloud-sdk-iotda</artifactId>
    <version>[3.0.40-rc, 3.2.0)</version>
</dependency>
```

Step 2 The following is an example of a message sent by the application to a single device:

```
public class MessageDistributionSolution {
    // REGION_ID: If CN East-Shanghai1 is used, enter cn-east-3. If CN North-Beijing4 is used, enter cn-north-4. If CN South-Guangzhou is used, enter cn-south-4.
    private static final String REGION_ID = "<YOUR REGION ID>";
    // ENDPOINT: On the console, choose Overview and click Access Addresses to view the HTTPS application access address.
    private static final String ENDPOINT = "<YOUR ENDPOINT>";
    // For the standard or enterprise edition, create a region object.
    public static final Region REGION_CN_NORTH_4 = new Region(REGION_ID, ENDPOINT);
    public static void main(String[] args) {
        String ak = "<YOUR AK>";
        String sk = "<YOUR SK>";
        String projectId = "<YOUR PROJECTID>";
        // Create a credential.
        ICredential auth = new BasicCredentials().withDerivedPredicate(AbstractCredentials.DEFAULT_DERIVED_PREDICATE)
            .withAk(ak)
            .withSk(sk)
            .withProjectId(projectId);
        // Create and initialize an IoTDAClient instance.
        IoTDAClient client = IoTDAClient.newBuilder().withCredential(auth)
            // For the basic edition, select the region object in IoTDARegion.
            //.withRegion(IoTDARegion.CN_NORTH_4)
            // For the standard or enterprise edition, create a region object.
            .withRegion(REGION_CN_NORTH_4).build();
        // Instantiate a request object.
        CreateMessageRequest request = new CreateMessageRequest();
        request.withDeviceId("<YOUR DEVICE_ID>");
        DeviceMessageRequest body = new DeviceMessageRequest();
        body.withMessage("<YOUR DEVICE MESSAGE>");
        request.withBody(body);
```

```
        try {
            CreateMessageResponse response = client.createMessage(request);
            System.out.println(response.toString());
        } catch (ConnectionException e) {
            e.printStackTrace();
        } catch (RequestTimeoutException e) {
            e.printStackTrace();
        } catch (ServiceResponseException e) {
            e.printStackTrace();
            System.out.println(e.getHttpStatusCode());
            System.out.println(e.getRequestId());
            System.out.println(e.getErrorCode());
            System.out.println(e.getErrorMsg());
        }
    }
}
```

Table 5-4 Parameters

Parameter	Description
ak	Access key ID (AK) of your Huawei Cloud account. You can create and check your AK/SK on the **My Credentials** > **Access Keys** page of the Huawei Cloud console. For details, see **Access Keys**.
sk	Secret access key (SK) of your Huawei Cloud account.
projectId	Project ID. For details on how to obtain a project ID, see **Obtaining a Project ID**.
IoTDARegion.CN_NORTH_4	Region where the platform to be accessed is located. The available regions of the platform have been defined in the SDK code **IoTDARegion.java**. On the console, you can view the region name of the current service and the mapping between regions and endpoints. For details, see **Platform Connection Information**.
REGION_ID	If CN East-Shanghai1 is used, enter **cn-east-3**. If CN North-Beijing4 is used, enter **cn-north-4**. If CN South-Guangzhou is used, enter **cn-south-4**.
ENDPOINT	On the console, choose **Overview** and click **Access Addresses** to view the HTTPS application access address.
DEVICE_ID	Unique ID of the device that a message is delivered to. The value of this parameter is allocated by the platform during device registration. The value is a string of no more than 128 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed.

----End

In the example, JDK 1.8 or a later version is used. **Download an SDK**. Configure the Java SDK on the device:

Step 1 Configure the Maven dependency of the SDK on devices.

```
<dependency>
    <groupId>com.huaweicloud</groupId>
```

```
    <artifactId>iot-device-sdk-java</artifactId>
    <version>1.1.4</version>
</dependency>
```

Step 2 Configure the SDK and device connection parameters on devices.

```
// Load the CA certificate of the IoT platform. For details about how to obtain the certificate, visit https://support.huaweicloud.com/intl/en-us/devg-iothub/iot_02_1004.html.
URL resource = BroadcastMessageSample.class.getClassLoader().getResource("ca.jks");
File file = new File(resource.getPath());

// The format is ssl://Domain name:Port number.
// To obtain the domain name, log in to the Huawei Cloud IoTDA console. In the navigation pane, choose Overview and click Access Details in the Instance Information area. Select the access domain name corresponding to port 8883.
String serverUrl = "ssl://localhost:8883";
// Device ID created on the platform
String deviceId = "deviceId";
// Secret corresponding to the device ID
String deviceSecret = "secret";
// Create a device.
IoTDevice device = new IoTDevice(serverUrl, deviceId, deviceSecret, file);
if (device.init() != 0) {
    return;
}
```

Step 3 Define the message delivery callback function.

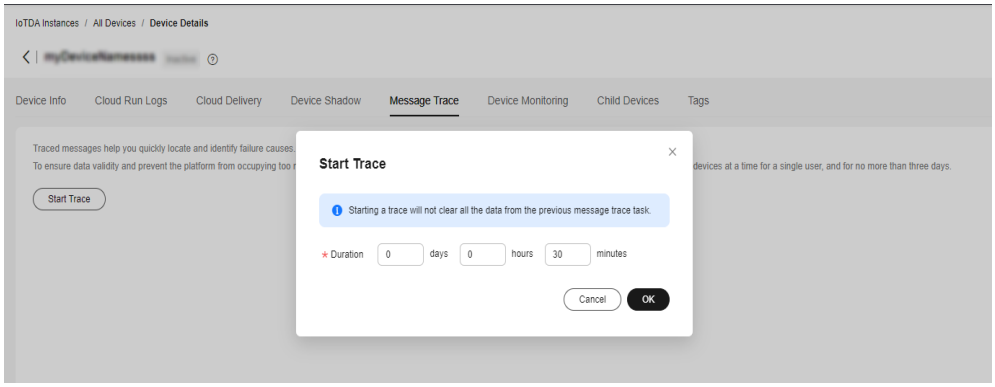
```
client.setDeviceMessageListener(deviceMessage -> {
    log.info("the onDeviceMessage is {}", deviceMessage.toString());
});
```

----End

Verify the setting:

Step 1 On the IoTDA console, click the target instance card. In the navigation pane, choose **Devices** > **All Devices**. On the displayed page, locate the target device, and click **View** in the **Operation** column to access its details page. Click the **Message Trace** tab, and click **Start Trace**.

Figure 5-21 Message tracing - Starting message tracing



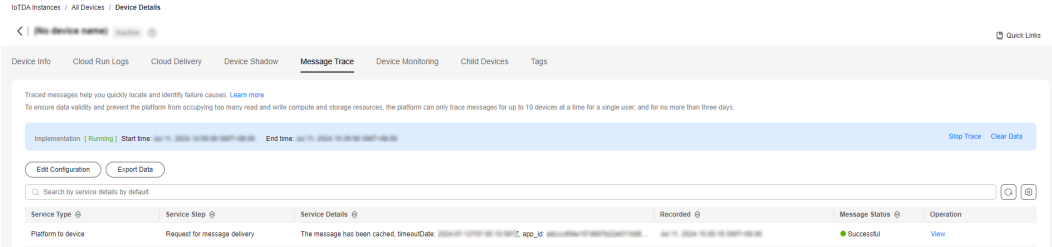
Step 2 Run the SDK code on the application and deliver a message. The following is an example of the response from the platform.

Figure 5-22 Response indicating the delivery success of the application message

```
class CreateMessageResponse {
    messageId: 8a33c102-3ece-471c-ad87-96313daa0e53
    result: class MessageResult {
        status: PENDING
        createdTime: 20230606T083644Z
        finishedTime: null
    }
}
```

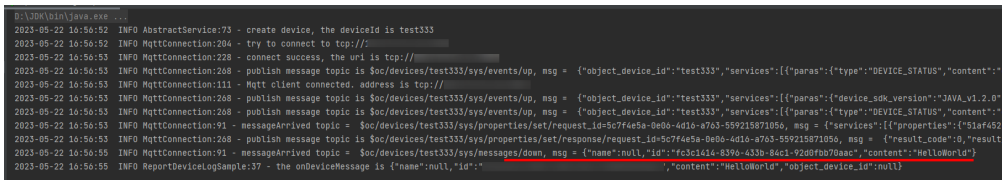
Step 3 The record can be checked on the **Message Trace** tab page.

Figure 5-23 Message tracing - Caching delivered messages



Step 4 Run the SDK code on the device. The following is an example of the log format when the device receives a message.

Figure 5-24 Device receiving messages



----End

5.2.3 Property Delivery

Overview

Property delivery is used for property query or modification. An application or the platform can obtain device property information or modify the properties, and synchronize the modification result to the device. After receiving a message, the device needs to return the property execution result to the platform in a timely manner. If the device does not respond, the platform considers that the property delivery times out.

Scenarios

- The platform proactively obtains or modifies device properties.

- The platform standardizes, parses, and filters the data.

Constraints

- Max. size of a single message: 64 KB.
- **Product models** are required.

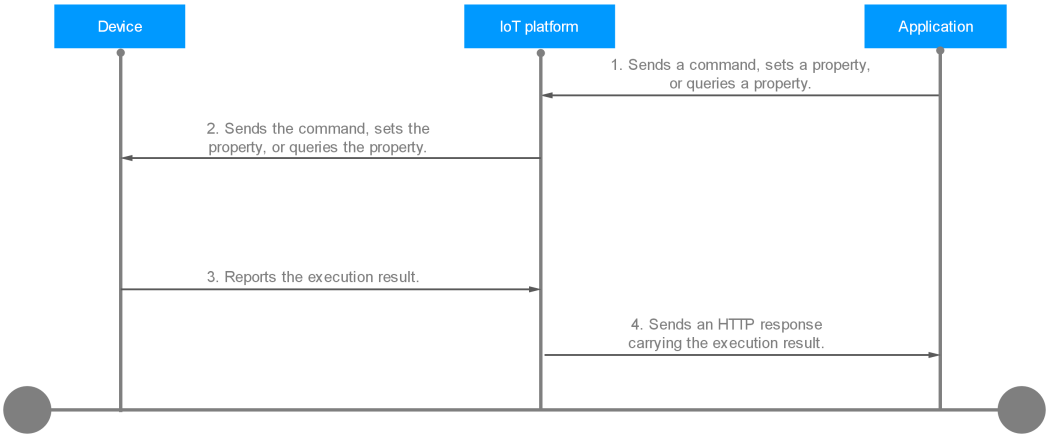
APIs

- **Platform Setting Device Properties**
- **Platform Querying Device Properties**

Property Delivery Usage

Property delivery is used for property modification or query. The following uses property modification as an example.

Figure 5-25 Process of property delivery



1. An application calls the API for **modifying device properties** to deliver a request to the platform. Example message:

```
PUT https://{endpoint}/v5/iot/{project_id}/devices/{device_id}/properties
{
   "services" : [ {
      "service_id" : "Temperature",
      "properties" : {
         "value" : 57
      }
    }, {
      "service_id" : "Battery",
      "properties" : {
         "level" : 80
      }
   } ]
}
```

2. The platform sends the property to the device according to the protocol specifications. The following is an example of **setting properties** through the APIs for MQTT devices.

```
Topic: $oc/devices/{device_id}/sys/properties/set/request_id={request_id}
Data format:
{
   "object_device_id": "{object_device_id} ",
   "services": [
```

```
        {
            "service_id": "Temperature",
            "properties": {
                "value": 57,
                "value2": 60
            }
        },
        {
            "service_id": "Battery",
            "properties": {
                "level": 80,
                "level2": 90
            }
        }
    ]
}
```

3. The device executes the command and returns the execution result. Example message:

```
Topic: $oc/devices/{device_id}/sys/properties/set/response/request_id={request_id}
Data format:
{
    "result_code": 0,
    "result_desc": "success"
}
```

4. The platform synchronously sends a response to the HTTP command. Example message:

```
Status Code: 200 OK
Content-Type: application/json
{
    "response" : {
        "result_code" : 0,
        "result_desc" : "success"
    }
}
```

Using the Java SDK for Property Delivery

This section describes how to use the Java SDK for the development of property configuration. **Download an SDK**. JDK 1.8 or later is used.

Configure the SDK on the application side:

Step 1 Configure the Maven dependency.

```
<dependency>
    <groupId>com.huaweicloud.sdk</groupId>
    <artifactId>huaweicloud-sdk-core</artifactId>
    <version>[3.0.40-rc, 3.2.0)</version>
</dependency>
<dependency>
    <groupId>com.huaweicloud.sdk</groupId>
    <artifactId>huaweicloud-sdk-iotda</artifactId>
    <version>[3.0.40-rc, 3.2.0)</version>
</dependency>
```

Step 2 Set the device properties on the application by referring to the following example.

```
public class AttributeDistributionSolution {
    // REGION_ID: If CN East-Shanghai1 is used, enter cn-east-3. If CN North-Beijing4 is used, enter cn-north-4. If CN South-Guangzhou is used, enter cn-south-4.
    private static final String REGION_ID = "<YOUR REGION ID>";
    // ENDPOINT: On the console, choose Overview and click Access Addresses to view the HTTPS application access address.
    private static final String ENDPOINT = "<YOUR ENDPOINT>";
    // For the standard or enterprise edition, create a region object.
    public static final Region REGION_CN_NORTH_4 = new Region(REGION_ID, ENDPOINT);
    public static void main(String[] args) {
```

```
        String ak = "<YOUR AK>";
        String sk = "<YOUR SK>";
        String projectId = "<YOUR PROJECTID>";
        // Create a credential.
        ICredential auth = new
BasicCredentials().withDerivedPredicate(AbstractCredentials.DEFAULT_DERIVED_PREDICATE)
            .withAk(ak)
            .withSk(sk)
            .withProjectId(projectId);
        // Create and initialize an IoTDAClient instance.
        IoTDAClient client = IoTDAClient.newBuilder().withCredential(auth)
            // For the basic edition, select the region object in IoTDARegion.
            //.withRegion(IoTDARegion.CN_NORTH_4)
            // For the standard or enterprise edition, create a region object.
            .withRegion(REGION_CN_NORTH_4).build();
        // Instantiate a request object.
        UpdatePropertiesRequest request = new UpdatePropertiesRequest();
        request.withDeviceId("<YOUR DEVICE_ID>");
        DevicePropertiesRequest body = new DevicePropertiesRequest();
        body.withServices("[{\"service_id\":\"smokeDetector\",\"properties\":{\"alarm\":\"hello\"," +
                "\"temperature\":10.323,\"humidity\":654.32,\"smokeConcentration\":342.4}}]");
        request.withBody(body);
        try {
            UpdatePropertiesResponse response = client.updateProperties(request);
            System.out.println(response.toString());
        } catch (ConnectionException e) {
            e.printStackTrace();
        } catch (RequestTimeoutException e) {
            e.printStackTrace();
        } catch (ServiceResponseException e) {
            e.printStackTrace();
            System.out.println(e.getHttpStatusCode());
            System.out.println(e.getRequestId());
            System.out.println(e.getErrorCode());
            System.out.println(e.getErrorMsg());
        }
    }
}
```

Table 5-5 Parameters

Parameter	Description
ak	Access key ID (AK) of your Huawei Cloud account. You can create and check your AK/SK on the **My Credentials** > **Access Keys** page of the Huawei Cloud console. For details, see **Access Keys**.
sk	Secret access key (SK) of your Huawei Cloud account.
projectId	Project ID. For details on how to obtain a project ID, see **Obtaining a Project ID**.
IoTDARegion.CN_NORTH_4	Region where the platform to be accessed is located. The available regions of the platform have been defined in the SDK code **IoTDARegion.java**. On the console, you can view the region name of the current service and the mapping between regions and endpoints. For details, see **Platform Connection Information**.
REGION_ID	If CN East-Shanghai1 is used, enter **cn-east-3**. If CN North-Beijing4 is used, enter **cn-north-4**. If CN South-Guangzhou is used, enter **cn-south-4**.

Parameter	Description
ENDPOINT	On the console, choose **Overview** and click **Access Addresses** to view the HTTPS application access address.
DEVICE_ID	Unique ID of the device that a message is delivered to. The value of this parameter is allocated by the platform during device registration. The value is a string of no more than 128 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed.

----End

Configure the SDK on the device side:

Step 1 Configure the Maven dependency of the SDK on devices.

```
<dependency>
    <groupId>com.huaweicloud</groupId>
    <artifactId>iot-device-sdk-java</artifactId>
    <version>1.1.4</version>
</dependency>
```

Step 2 Configure the SDK and device connection parameters on devices.

```
// Load the CA certificate of the IoT platform. For details about how to obtain the certificate, visit https://support.huaweicloud.com/intl/en-us/devg-iothub/iot_02_1004.html.
URL resource = AttributeSample.class.getClassLoader().getResource("ca.jks");
File file = new File(resource.getPath());

// Format: ssl://Domain name:Port number
// To obtain the domain name, log in to the Huawei Cloud IoTDA console. In the navigation pane, choose Overview and click Access Details in the Instance Information area. Select the access domain name corresponding to port 8883.
String serverUrl = "ssl://localhost:8883";
// Device ID created on the platform
String deviceId = "deviceId";
// Secret corresponding to the device ID
String deviceSecret = "secret";
// Create a device.
IoTDevice device = new IoTDevice(serverUrl, deviceId, deviceSecret, file);
if (device.init() != 0) {
    return;
}
```

Step 3 Define the property delivery callback function.

```
device.getClient().setPropertyListener(new PropertyListener() {

    // Process property writing.
    @Override
    public void onPropertiesSet(String requestId, List<ServiceProperty> services) {

        // Traverse services.
        for (ServiceProperty serviceProperty: services){

            log.info("OnPropertiesSet, serviceId =  " + serviceProperty.getServiceId());

            // Traverse properties.
            for (String name :serviceProperty.getProperties().keySet()){
                log.info("property name = "+ name);
                log.info("set property value = "+ serviceProperty.getProperties().get(name));
                if (name.equals("alarm")){
                    // Change the local value.
                    alarm = (Integer) serviceProperty.getProperties().get(name);
                }
```

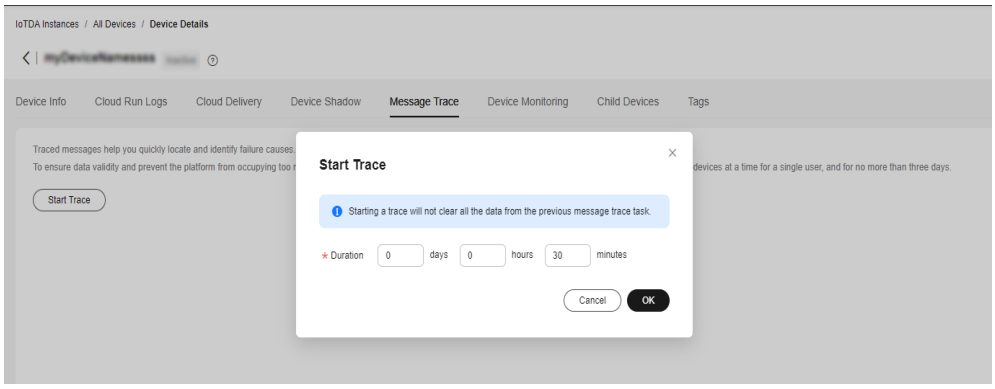
```
            }

        }
        // Set property response.
        device.getClient().respondPropsSet(requestId, IotResult.SUCCESS);
    }

    // Process property reading.
    @Override
    public void onPropertiesGet(String requestId, String serviceId) {
        log.info("OnPropertiesGet " + serviceId);
        Map<String ,Object> json = new HashMap<>();
        Random rand = new Random();
        json.put("alarm", alarm);
        json.put("temperature", rand.nextFloat()*100.0f);
        json.put("humidity", rand.nextFloat()*100.0f);
        json.put("smokeConcentration", rand.nextFloat() * 100.0f);

        ServiceProperty serviceProperty = new ServiceProperty();
        serviceProperty.setProperties(json);
        serviceProperty.setServiceId("smokeDetector");

        // Report the property reading response.
        device.getClient().respondPropsGet(requestId, Arrays.asList(serviceProperty));
    }
});
```

----End

Verify the setting:

Step 1 On the IoTDA console, click the target instance card. In the navigation pane, choose **Devices** > **All Devices**. On the displayed page, locate the target device, and click **View** in the **Operation** column to access its details page. Click the **Message Trace** tab, and click **Start Trace**.

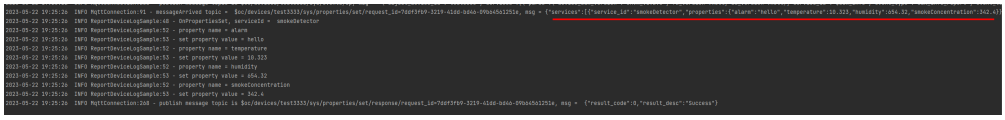
Figure 5-26 Message tracing - Starting message tracing



Step 2 Run the SDK code on the device to bring the device online.

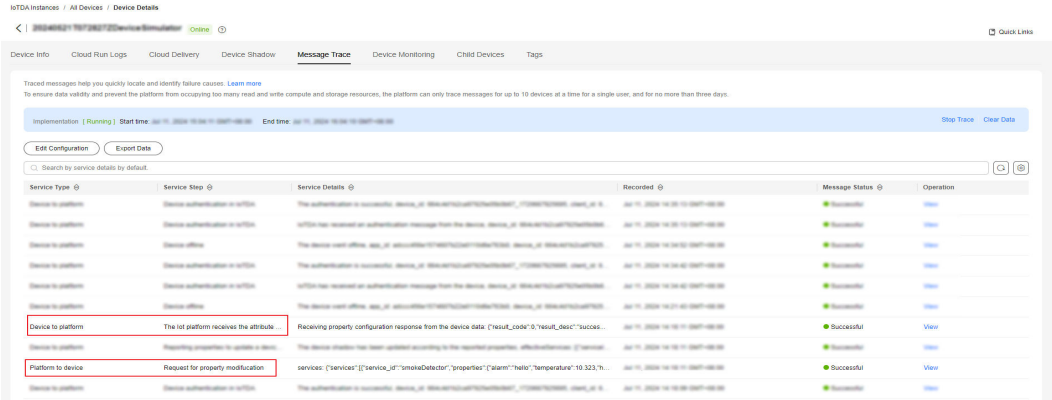
Step 3 Run the SDK code on the application and call the API for modifying device properties to send a request to the device. The following provides the result received by the device.

Figure 5-27 Property modification result on the device



Step 4 Check the result on the **Message Trace** tab page.

Figure 5-28 Message tracing - Delivering properties



----End

5.2.4 Command Delivery

Overview

A product model defines commands that can be delivered to the devices. Applications can call platform APIs to deliver commands to the devices to effectively manage these devices.

IoTDA supports synchronous and asynchronous command delivery.

Table 5-6 Command delivery

Mechanism	Description	Scenario	Devices Using LwM2M over CoAP	Devices Using MQTT
Synchronous command delivery	An application calls the synchronous command delivery API to deliver a command to a specified device for device control. The platform sends the command to the device and returns the command execution result in an HTTP request to the application. If the device does not respond, the platform returns a timeout message to the application.	Applicable to commands that must be executed in real time, for example, turning on a street lamp or closing a gas meter switch. Applications should determine the appropriate time to deliver a command.	Not applicable	Applicable

Mechanism	Description	Scenario	Devices Using LwM2M over CoAP	Devices Using MQTT
Asynchronous command delivery	An application calls the asynchronous command delivery API to deliver a command to a specified device for device control. The platform sends the command to the device and asynchronously pushes the command execution result to the application. Asynchronous command delivery is classified into immediate delivery and delayed delivery. ● In immediate delivery, the platform delivers commands to a device regardless of whether the device is online. If the device is offline or the device does not receive the command, the delivery fails. ● In delayed delivery, the platform caches a command and delivers it to a device when the device goes online or reports data. If a device has multiple pending commands, the platform delivers the commands in sequence.	● Immediate delivery applies to scenarios with high real-time requirements. ● Delayed delivery applies to commands that do not need to be executed immediately, for example, configuring water meter parameters.	Applicable	Not applicable

NOTE

For details, see **Synchronous Command Delivery** and **Asynchronous Command Delivery**.

Scenarios

- Synchronous delivery applies to scenarios that require real-time command delivery. Asynchronous delivery is used for device control.
- Data needs to be forwarded to other Huawei Cloud services for storage and processing based on **data forwarding rules**.

Constraints

- Max. size of a single message: 256 KB.
- **Product models** are required.
- Devices need to respond to the synchronous command within 20 seconds.
- Up to 20 asynchronous commands can be cached at a time.
- Max. cache duration (configurable): 48 hours.

APIs

- APIs for the platform and applications
  - **Deliver a Command to a Device**
  - **Deliver an Asynchronous Command**
  - **Query a Command with a Specific ID**
- APIs for MQTT devices
  - **Platform Delivering a Command**
  - **Platform Delivering a Message**.
- APIs for devices using LwM2M over CoAP
  - **Platform Delivering a Command**

Synchronous Command Delivery

You can deliver synchronous commands to MQTT devices one by one or in batches.

Table 5-7 Synchronous command delivery

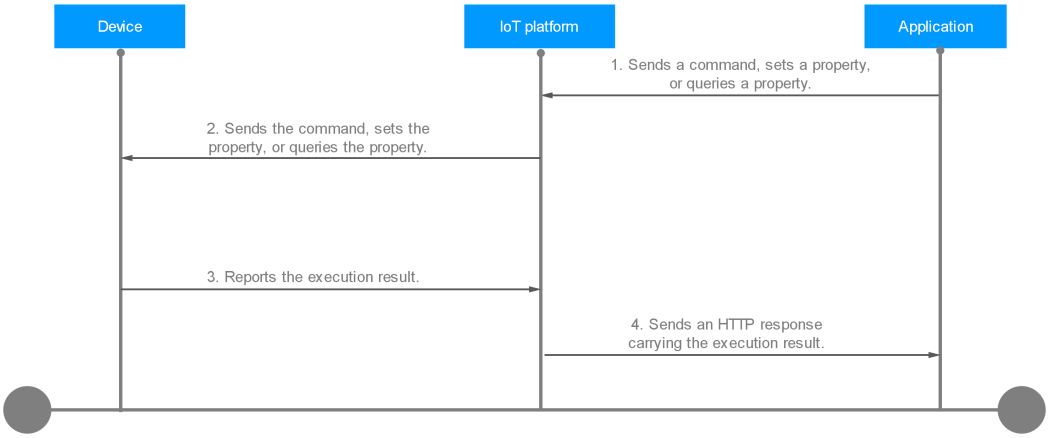
Type	Description	Scenario	Example
Synchronous command delivery to a single MQTT device	IoTDA delivers a control command to a single device.	The function is used to deliver a control command to a single device.	**Synchronous Command Delivery to an Individual MQTT Device**

Type	Description	Scenario	Example
Synchronous command delivery to a batch of MQTT devices	IoTDA delivers control commands to devices in batches. You can create a batch task to perform operations on multiple devices at a time.	The function is used to deliver control commands to devices in batches.	**Synchronous Command Delivery to a Batch of MQTT Devices**

Synchronous Command Delivery to an Individual MQTT Device

For details on how to set and query properties, see instructions of the APIs for **querying device properties** and **modifying device properties**.

Figure 5-29 Command delivery process



1. An application calls the API for **delivering a command to a device** to send a command to the platform. Example message:

```
POST https://{Endpoint}/v5/iot/{project_id}/devices/{device_id}/commands
Content-Type: application/json
X-Auth-Token: ********

{
  "service_id" : "WaterMeter",
  "command_name" : "ON_OFF",
  "paras" : {
    "value" : "ON"
  }
}
```

2. The platform sends the command to the device according to the protocol specifications. Example message:

```
Topic: $oc/devices/{device_id}/sys/commands/request_id={request_id}
Data format:
{
  "object_device_id": "{object_device_id}",
  "command_name": "ON_OFF",
  "service_id": "WaterMeter",
  "paras": {
    "value": "ON"
  }
}
```

3. After executing the command, the device returns the command execution result through the API for **delivering a command**. Example message:

```
Topic: $oc/devices/{device_id}/sys/commands/response/request_id={request_id}
Data format:
{
    "result_code": 0,
    "response_name": "COMMAND_RESPONSE",
    "paras": {
        "result": "success"
    }
}
```

4. The platform synchronously sends a response to the HTTP command. Example message:

```
Status Code: 200 OK
Content-Type: application/json

{
  "command_id" : "b1224afb-e9f0-4916-8220-b6bab568e888",
  "response" : {
    "result_code" : 0,
    "response_name" : "COMMAND_RESPONSE",
    "paras" : {
      "result" : "success"
    }
  }
}
```

Synchronous Command Delivery to a Batch of MQTT Devices

The API for **creating a batch task** can be used to deliver a command to multiple MQTT devices. The following describes how to call the API for **creating a batch task** to deliver commands in batches.

1. An application calls the API for **creating a batch task** to send a command to the platform. Example message:

```
POST https://{Endpoint}/v5/iot/{project_id}/batchtasks
Content-Type: application/json
X-Auth-Token: ********
{
  "app_id": "********",
  "task_name": "task123",
  "task_type": "createCommands",
  "targets": [
    "********",
    "********"
  ],
  "document": {
    "service_id": "water",
    "command_name": "ON_OFF",
    "paras": {
      "value": "ON"
    }
  }
}
```

Table 5-8 Parameters for creating a batch task of command delivery

Parameter	Mandatory	Description
app_id	No	Resource space ID.

Parameter	Mandatory	Description
task_name	Yes	Custom task name.
task_type	Yes	Type of the batch task. For details, see **Create a Batch Task**. Options: • **createCommands**: task for creating synchronous commands in batches • **createAsyncCommands**: task for creating asynchronous commands in batches
targets	No	Device ID array, which is the target for executing the batch task.
document	No	Task execution data file, in JSON format (key-value pairs). For details, see **Deliver a Command to a Device**.

2. The platform returns a **201 Created** message to the application.
3. The device receives the command and sends the command result to the platform through the upstream topic. For details, see **Platform Delivering a Command**.
4. Call the API for **querying the batch task list** to query the execution status of a batch command delivery task.

Asynchronous Command Delivery

Asynchronous command delivery is used for IoTDA or applications to deliver commands to devices using LwM2M over CoAP to access IoTDA. Two modes are available.

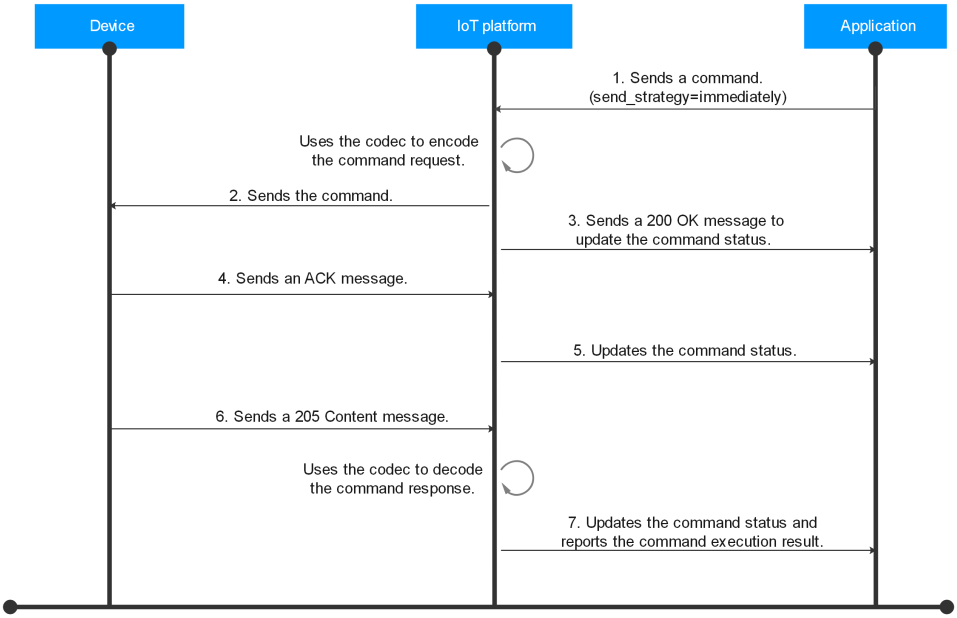
Table 5-9 Asynchronous command delivery

Type	Description	Scenario	Process
Immediate delivery of asynchronous commands	IoTDA delivers commands to a device upon receiving a command regardless of whether the device is online. If the device is offline or the device does not receive the command, the delivery fails.	It is used in scenarios that have high requirements for timeliness.	**Immediate Delivery of Asynchronous Commands**

Type	Description	Scenario	Process
Delayed delivery of asynchronous commands	The platform caches a command and delivers it to a device when the device goes online or reports properties. If a device has multiple cached commands, the platform delivers the commands in sequence.	Delayed delivery applies to commands that do not need to be executed immediately, for example, configuring water meter parameters.	**Delayed Delivery of Asynchronous Commands**

Immediate Delivery of Asynchronous Commands

Figure 5-30 Command delivery for devices using LwM2M over CoAP



An example of the corresponding steps is as follows:

1. An application calls the API for **delivering an asynchronous command** to send a command to the platform. The **send_strategy** parameter in the command request is set to **immediately**. Example message:

```
POST https://{endpoint}/v5/iot/{project_id}/devices/{device_id}/async-commands
Content-Type: application/json
X-Auth-Token: ********

{
  "service_id" : "WaterMeter",
  "command_name" : "ON_OFF",
  "paras" : {
    "value" : "ON"
  },
```

```
  "expire_time": 0,
  "send_strategy": "immediately"
}
```

2. The platform uses the **codec** to encode the command request, and sends the command through the **Execute** operation of the device management and service implementation interface defined in the LwM2M protocol. The message body is in binary format.
3. The platform sends a 200 OK message carrying the command status **SENT** to the application. (If the device is offline or the device does not receive the command, the delivery fails and the command status is **FAILED**.)
4. The device returns an ACK message after receiving the command.
5. If the application has subscribed to command status change notifications, the platform pushes a message to the application by calling the API for pushing a command status change notification. The command status carried in the message is **DELIVERED**. Example message:

```
Method: POST
request:
Body:
{
    "resource": "device.command.status",
    "event": "update",
    "event_time": "20200811T080745Z",
    "notify_data": {
        "header": {
            "app_id": "********",
            "device_id": "********",
            "node_id": "test0001",
            "product_id": "********",
            "gateway_id": "********",
            "tags": []
        },
        "body": {
            "command_id": "********",
            "created_time": "20200811T080738Z",
            "sent_time": "20200811T080738Z",
            "delivered_time": "20200811T080745Z",
            "response_time": "",
            "status": "DELIVERED",
            "result": null
        }
    }
}
```

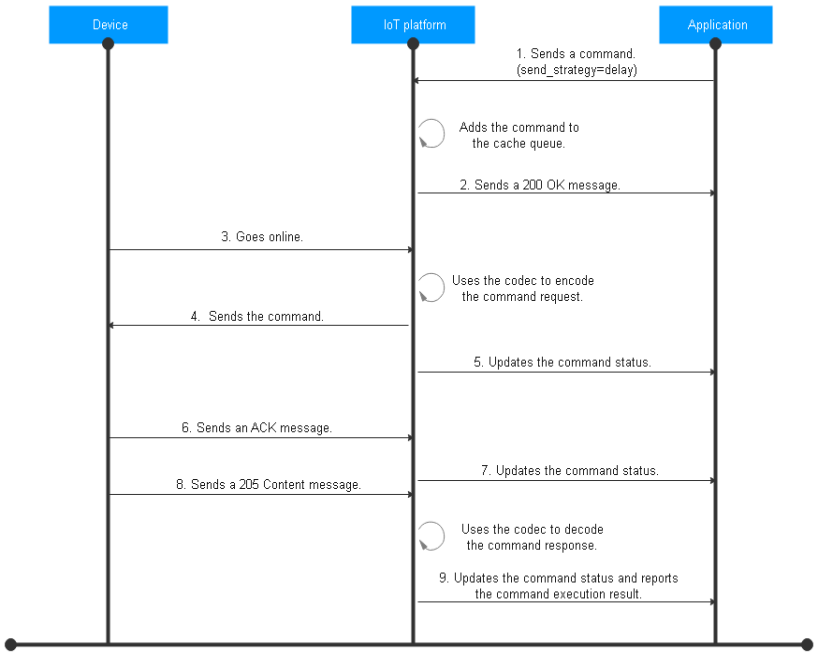
6. After the command is executed, the device returns the command execution result in a 205 Content message.
7. If the application has subscribed to command status change notifications, the platform uses the codec to decode the command response and sends a push message to the application by calling the API for pushing a command status change notification. The command status carried in the message is **SUCCESSFUL**. Example message:

```
Method: POST
request:
Body:
{
    "resource": "device.command.status",
    "event": "update",
    "event_time": "20200811T080745Z",
    "notify_data": {
        "header": {
            "app_id": "********",
            "device_id": "********",
            "node_id": "test0001",
            "product_id": "********",
```

```
            "gateway_id": "********",
            "tags": []
        },
        "body": {
            "command_id": "********",
            "created_time": "20200811T080738Z",
            "sent_time": "20200811T080738Z",
            "delivered_time": "20200811T080745Z",
            "response_time": "20200811T081745Z",
            "status": "SUCCESSFUL",
            "result": {
                "resultCode":"SUCCESSFUL",
                "resultDetail": {
                    "value": "ON"
                }
            }
        }
    }
}
```


Delayed Delivery of Asynchronous Commands

Figure 5-31 Delayed command delivery for devices using LwM2M over CoAP



1. An application calls the API for **delivering an asynchronous command** to send a command to the platform. The **send_strategy** parameter in the command request is set to **delay**.
2. The platform adds the command to the cache queue and reports a 200 OK message. The command status is **PENDING**.
3. The device goes online or reports data to the platform.
4. The platform uses the **codec** to encode the command request and sends the command to the device according to the protocol specifications.

5. If the application has subscribed to command status change notifications, the platform pushes a message to the application by calling the API for pushing a command status change notification. The command status carried in the message is **SENT**.
6. The subsequent flow is the same as **4** to **7** described in the immediate delivery scenario.

LwM2M/CoAP Device Command Execution Status

The figure below illustrates the command execution status and the table below describes the status change mechanism.

Figure 5-32 LwM2M/CoAP command delivery status

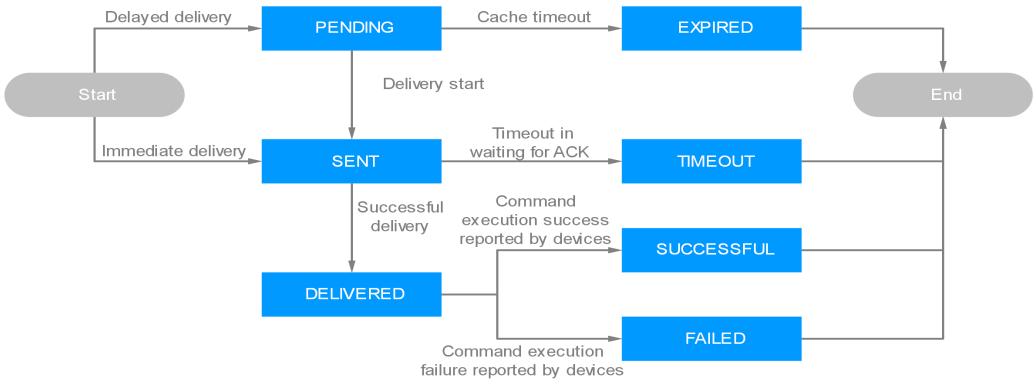


Table 5-10 LwM2M/CoAP command execution status

Status	Description
PENDING	• For a device using LwM2M over CoAP in delayed delivery mode, the platform caches a command if the device has not reported data. The command status is **PENDING**. • This status does not exist for devices using LwM2M over CoAP in immediate delivery mode.
EXPIRED	• For a device using LwM2M over CoAP in delayed delivery mode, if the platform does not deliver a command to the device within a specified time, the command status is **EXPIRED**. The expiration time is subject to the value of **expireTime** carried in the command request. If **expireTime** is not carried, the default value (24 hours) is used. • This status does not exist for devices using LwM2M over CoAP in immediate delivery mode.

Status	Description
SENT	• For a device using LwM2M over CoAP in delayed delivery mode, the platform sends a cached command when receiving data reported by the device. In this case, the command status changes from **PENDING** to **SENT**. • For a device using LwM2M over CoAP in immediate delivery mode, if the device is online when the platform delivers a command, the command status is **SENT**.
TIMEOUT	If the platform does not receive a response within 180 seconds after delivering a command to a device using LwM2M over CoAP, the command status is **TIMEOUT**.
DELIVERED	If the platform receives a response from a device, the command status is **DELIVERED**.
SUCCESSFUL	If the platform receives a result indicating that the command is executed, the command status is **SUCCESSFUL**.
FAILED	• If the platform receives a result indicating that the command execution failed, the command status is **FAILED**. • For a device using LwM2M over CoAP in immediate delivery mode, if the device is offline when the platform delivers a command, the command status is **FAILED**.

Platform Command Delivery Example

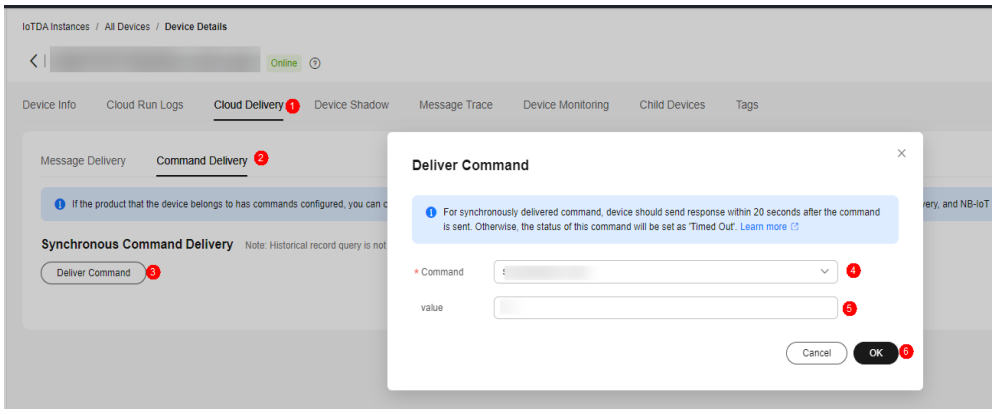
Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **All Devices**. On the device list, click a device to access its details page.

Step 3 The **Cloud Delivery** tab page varies according to the device protocol.

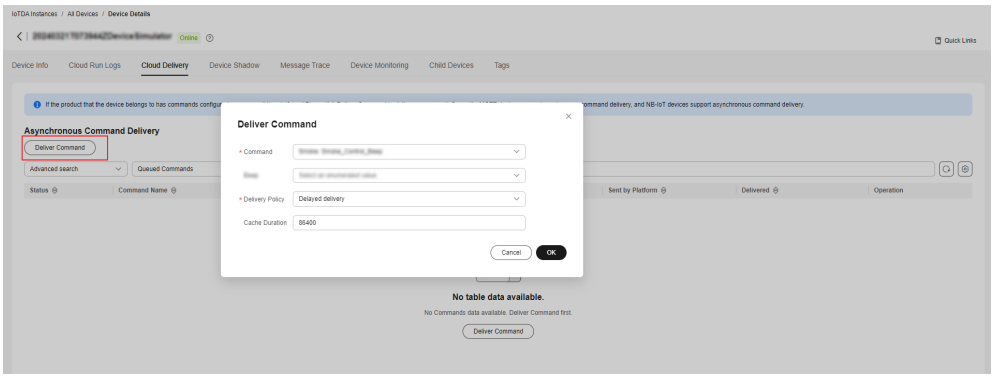
- Devices using MQTT support only synchronous command delivery. Click **Command Delivery** on the right. In the displayed dialog box, select the command to be delivered and set command parameters.

Figure 5-33 Command delivery - Synchronous command delivery



- Devices using LwM2M over CoAP support only asynchronous command delivery. Click **Deliver Command** on the right. In the displayed dialog box, select the command to be delivered and set command parameters. You can choose to send the command immediately or after a delay.

Figure 5-34 Command delivery - Asynchronous command delivery



----End

NOTE

- On the **Message Trace** tab page, you can view the creation time, sending time, delivered time, and the delivery status of a command delivery task. This information helps you learn the **command execution status**.
- In addition, you can call the API for **querying a command with a specific ID** to query the status and content of delivered commands on the platform.

The application uses the Java SDK for the development of synchronous command delivery. The development environment used in the example is JDK 1.8 or later. **Download an SDK**.

Step 1 Configure the Maven dependency.

```
<dependency>
    <groupId>com.huaweicloud.sdk</groupId>
    <artifactId>huaweicloud-sdk-core</artifactId>
    <version>[3.0.40-rc, 3.2.0)</version>
</dependency>
<dependency>
    <groupId>com.huaweicloud.sdk</groupId>
    <artifactId>huaweicloud-sdk-iotda</artifactId>
    <version>[3.0.40-rc, 3.2.0)</version>
</dependency>
```

Step 2 The following is an example of delivering a synchronous command:

```
public class CommandSolution {
    // REGION_ID: If CN East-Shanghai1 is used, enter cn-east-3. If CN North-Beijing4 is used, enter cn-north-4. If CN South-Guangzhou is used, enter cn-south-4.
    private static final String REGION_ID = "<YOUR REGION ID>";
    // ENDPOINT: On the console, choose Overview and click Access Addresses to view the HTTPS application access address.
    private static final String ENDPOINT = "<YOUR ENDPOINT>";
    // For the standard or enterprise edition, create a region object.
    public static final Region REGION_CN_NORTH_4 = new Region(REGION_ID, ENDPOINT);
    public static void main(String[] args) {
        String ak = "<YOUR AK>";
        String sk = "<YOUR SK>";
        String projectId = "<YOUR PROJECTID>";
        // Create a credential.
        ICredential auth = new
```

```
BasicCredentials().withDerivedPredicate(AbstractCredentials.DEFAULT_DERIVED_PREDICATE)
        .withAk(ak)
        .withSk(sk)
        .withProjectId(projectId);
    // Create and initialize an IoTDAClient instance.
    IoTDAClient client = IoTDAClient.newBuilder().withCredential(auth)
        // For the basic edition, select the region object in IoTDARegion.
        //.withRegion(IoTDARegion.CN_NORTH_4)
        // For the standard or enterprise edition, create a region object.
        .withRegion(REGION_CN_NORTH_4).build();
    // Instantiate a request object.
    CreateCommandRequest request = new CreateCommandRequest();
    request.withDeviceId("<YOUR DEVICE_ID>");
    DeviceCommandRequest body = new DeviceCommandRequest();
    body.withParas("{\"value\":\"1\"}");
    request.withBody(body);
    try {
        CreateCommandResponse response = client.createCommand(request);
        System.out.println(response.toString());
    } catch (ConnectionException e) {
        e.printStackTrace();
    } catch (RequestTimeoutException e) {
        e.printStackTrace();
    } catch (ServiceResponseException e) {
        e.printStackTrace();
        System.out.println(e.getHttpStatusCode());
        System.out.println(e.getRequestId());
        System.out.println(e.getErrorCode());
        System.out.println(e.getErrorMsg());
    }
  }
}
```

Table 5-11 Parameters

Parameter	Description
ak	Access key ID (AK) of your Huawei Cloud account. You can create and check your AK/SK on the **My Credentials** > **Access Keys** page of the Huawei Cloud console. For details, see **Access Keys**.
sk	Secret access key (SK) of your Huawei Cloud account.
projectId	Project ID. For details on how to obtain a project ID, see **Obtaining a Project ID**.
IoTDARegion.CN_NORTH_4	Region where the platform to be accessed is located. The available regions of the platform have been defined in the SDK code **IoTDARegion.java**. On the console, you can view the region name of the current service and the mapping between regions and endpoints. For details, see **Platform Connection Information**.
REGION_ID	If CN East-Shanghai1 is used, enter **cn-east-3**. If CN North-Beijing4 is used, enter **cn-north-4**. If CN South-Guangzhou is used, enter **cn-south-4**.
ENDPOINT	On the console, choose **Overview** and click **Access Addresses** to view the HTTPS application access address.

Parameter	Description
DEVICE_ID	Unique ID of the device that a message is delivered to. The value of this parameter is allocated by the platform during device registration. The value is a string of no more than 128 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed.

----End

To configure the device to use the Java SDK to deliver synchronous commands, perform the following steps. In this example, JDK 1.8 or a later version is used.

Step 1 Configure the Maven dependency of the SDK on devices.

```
<dependency>
    <groupId>com.huaweicloud</groupId>
    <artifactId>iot-device-sdk-java</artifactId>
    <version>1.1.4</version>
</dependency>
```

Step 2 Configure the SDK and device connection parameters on devices.

```
// Load the CA certificate of the IoT platform. For details about how to obtain the certificate, visit https://support.huaweicloud.com/intl/en-us/devg-iothub/iot_02_1004.html.
URL resource = BroadcastMessageSample.class.getClassLoader().getResource("ca.jks");
File file = new File(resource.getPath());

// The format is ssl://Domain name:Port number.
// To obtain the domain name, log in to the Huawei Cloud IoTDA console. In the navigation pane, choose Overview and click Access Details in the Instance Information area. Select the access domain name corresponding to port 8883.
String serverUrl = "ssl://localhost:8883";
// Device ID created on the platform
String deviceId = "deviceId";
// Secret corresponding to the device ID
String deviceSecret = "secret";
// Create a device.
IoTDevice device = new IoTDevice(serverUrl, deviceId, deviceSecret, file);
if (device.init() != 0) {
    return;
}
```

Step 3 Set the command delivery callback function and send a response.

```
client.setCommandListener(new CommandListener() {
    @Override
    public void onCommand(String requestId, String serviceId, String commandName, Map<String, Object> paras) {
        log.info("onCommand, serviceId = " +serviceId);
        log.info("onCommand , name = " + commandName);
        log.info("onCommand, paras =  " + paras.toString());

        // Define the processing command.

        // Send a command response.
        device.getClient().respondCommand(requestId, new CommandRsp(0));
    }
});
```

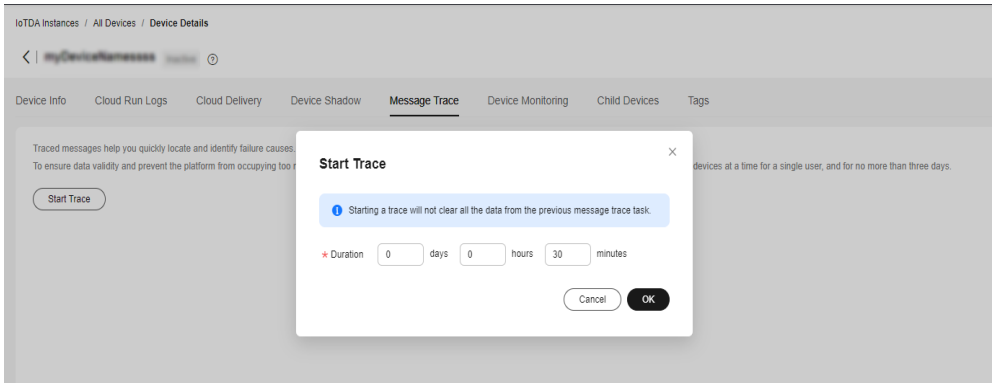
----End

Verify the setting:

Step 1 On the IoTDA console, click the target instance card. In the navigation pane, choose **Devices** > **All Devices**. On the displayed page, locate the target device,

and click **View** in the **Operation** column to access its details page. Click the **Message Trace** tab, and click **Start Trace**.

Figure 5-35 Message tracing - Starting message tracing



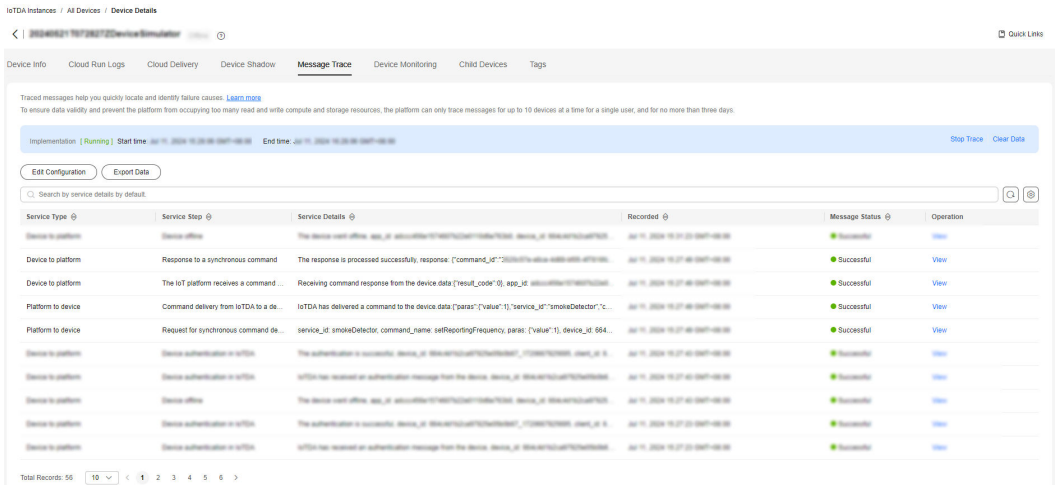
Step 2 Run the SDK code on the device to bring the device online.

Step 3 Run the application code. After receiving a command, the device processes and responds to the command. The following is an example of a command received by the device.

Figure 5-36 Successful command delivery result on the device

Step 4 Check the result on the **Message Trace** tab page.

Figure 5-37 Message tracing - Delivering commands



----End

5.3 Custom Topic Communications

5.3.1 Overview

Introduction

IoTDA uses topics to communicate with devices connected using MQTT. There are custom topics and system topics. System topics are basic communications topics preconfigured on the platform. You can also customize topics on the platform based on service requirements. Note that message through both custom topics and system topics are transparently transmitted on the platform, which means that the platform does not proactively parse data content.

Table 5-12 Topic categories

Category	Description	Scenario
System topic	The platform predefines topics for communications with devices. For details of the topic list and functions, see **Topics**.	Message reporting, property reporting, command delivery, and events
Custom topic	You can customize topics for device-platform communications. **Types of custom topics:** • **Custom Topics Starting with $oc**: Topics defined in the product are prefixed with **$oc/devices/{device_id}/user/**. During message reporting or delivery, the platform checks whether the topic is defined in the product. Undefined topics will be rejected by the platform. • **Custom Topics Not Starting with $oc**: Topics that do not start with **$oc**, for example, **/aircondition/data/up**. They are used for upstream and downstream message communications based on MQTT rules. The platform checks the topic permission using **topic policies**.	Scenarios where services require specific topics, such as **M2M communications**, **broadcast communications**, and device migration.

Scenarios

- Devices publish messages to custom topics. Applications smoothly **forward data** to message middleware, storage, data analysis, and service applications.
- An application calls the API for **delivering a message to a device** to publish messages to a specified custom topic. The device subscribes to this topic to receive messages from the server.
- **M2M communications**, **broadcast communications**, and device migration.

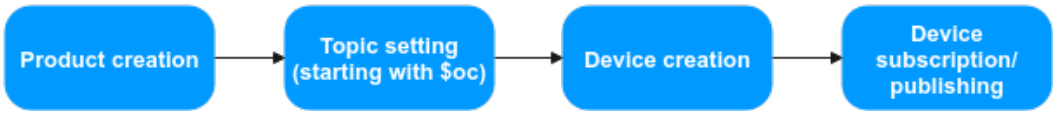
Constraints

- You can define a maximum of 50 custom topics for a product model.
- Custom topics are only available for message communications.
- Max. length of a custom MQTT topic: 128 bytes.

5.3.2 Custom Topics Starting with $oc

Process

Figure 5-38 Communications with custom topics starting with $oc

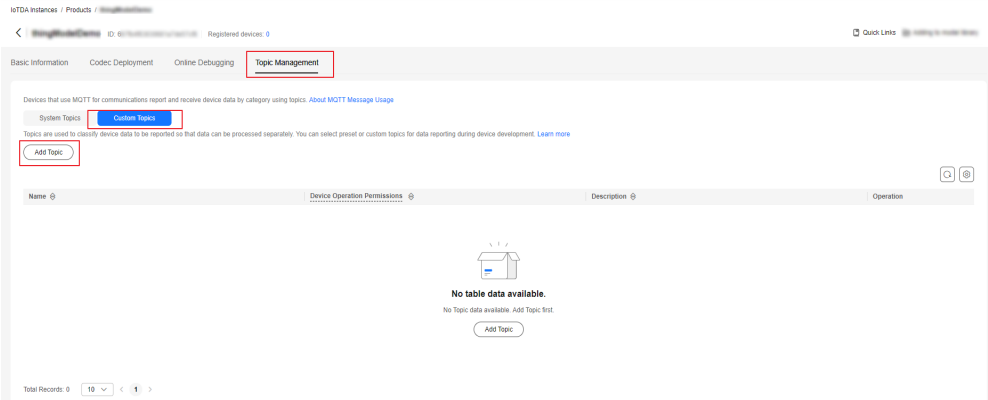


Step 1 Product creation: Access the **IoTDA** service page and click **Access Console**. Click the target instance card. For details, see **Creating a Product**.

Step 2 Topic setting: On the product details page, create a custom topic prefixed with **$oc/devices/{device_id}/user/**.

1. Select an MQTT product. On the product details page, click the **Topic Management** tab, select **Custom Topic**, and click **Add Topic**.

   Figure 5-39 Topic management - Custom topics



2. In the displayed dialog box, select device operation permissions and enter the topic name.

Figure 5-40 Topic management - Adding a custom topic

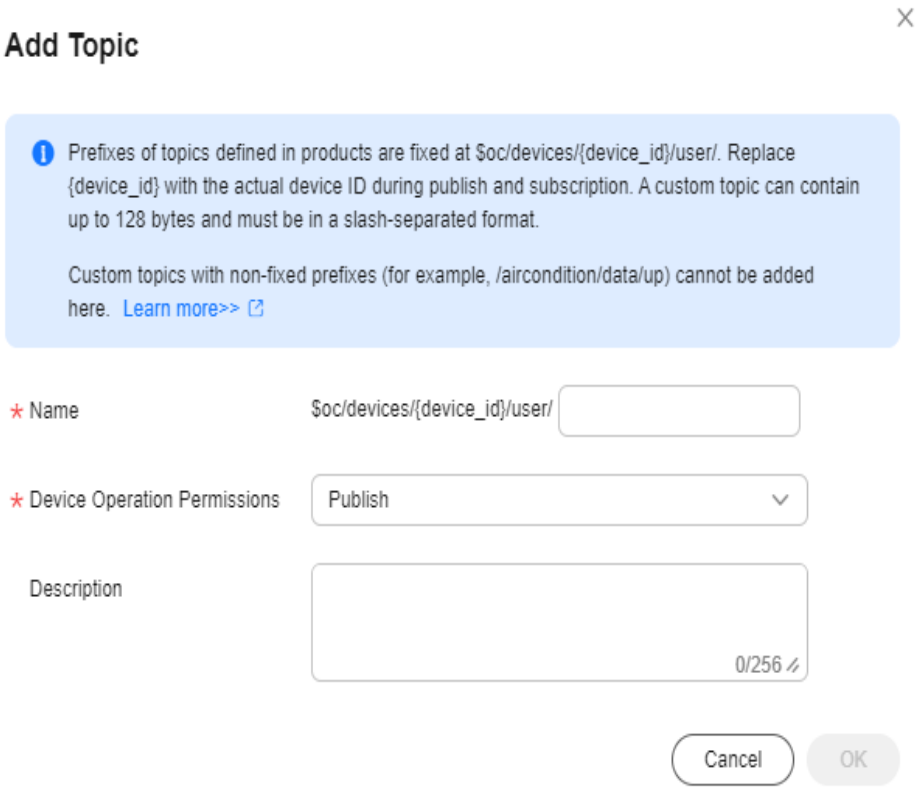


Table 5-13 Parameters

Parameter	Description
Name	The topic prefix is fixed at **$oc/devices/***{device_id}***/user/**. Replace *{device_id}* with the actual device ID during publishing and subscription. A custom topic must be in a slash-separated format. Enter 1 to 64 characters. Use only digits, letters, underscores (_), and slashes (/). The slashes cannot be consecutive. **NOTE** Custom topics do not support custom variables. For example, *{type}* in **$oc/devices/***{device_id}***/user/setting/***{type}* is a variable and is not supported.
Device Operation Permissions	– **Publish**: Devices can report messages using this topic. A topic is carried in a device message during data transfer for better classification. – **Subscribe**: Applications can specify a topic to deliver messages to devices. – **Publish and subscription**: Devices can report and receive messages using this topic.

Parameter	Description
Description	Provide a description of the topic.

3. Click **OK**. After the topic is added, you can modify or delete it in the custom topic list.

Step 3 Device creation: Create a device under the product. The created device inherits the custom topics set for the product. For details, see **Registering an Individual Device**.

Step 4 Device subscription/publishing: For details about how to publish and subscribe to messages through custom topics, see **Using a Custom Topic for Communication**.

----End

Java SDK Usage on the Device Side

Devices can integrate the **device SDKs** provided by Huawei Cloud IoT to quickly connect to Huawei Cloud IoTDA and report messages. The following example uses the Java SDK to connect a device to IoTDA for publishing and subscribing to messages through the custom topic **$oc/devices/ + device.getDeviceId() + /user/wpy**.

1. Configure the Maven dependency of the SDK on the device.

```
<dependency>
    <groupId>com.huaweicloud</groupId>
    <artifactId>iot-device-sdk-java</artifactId>
    <version>1.1.4</version>
</dependency>
```

2. Configure the SDK and device connection parameters on the device.

```
// Load the CA certificate of the IoT platform. For details about how to obtain the certificate, visit
https://support.huaweicloud.com/intl/en-us/devg-iothub/iot_02_1004.html.
URL resource = MessageSample.class.getClassLoader().getResource("ca.jks");
File file = new File(resource.getPath());

// The format is ssl://Domain name:Port number.
// To obtain the domain name, log in to the Huawei Cloud IoTDA console. In the navigation pane,
choose Overview and click Access Details in the Instance Information area. Select the access
domain name corresponding to port 8883.
String serverUrl = "ssl://localhost:8883";
// Device ID created on the IoT platform
String deviceId = "deviceId";
// Secret corresponding to the device ID
String deviceSecret = "secret";
// Initialize the device connection.
IoTDevice device = new IoTDevice(serverUrl, deviceId, deviceSecret, file);
if (device.init() != 0) {
    return;
}
```

3. Report a device message.

```
device.getClient().publishRawMessage(new RawMessage( "$oc/devices/" + device.getDeviceId() + "/
user/wpy", "hello", 1), new ActionListener() {
    @Override
    public void onSuccess(Object context) {
        System.out.println("reportDeviceMessage success: ");
    }
    @Override
    public void onFailure(Object context, Throwable var2) {
        System.out.println("reportDeviceMessage fail: " + var2);
    }
});
```

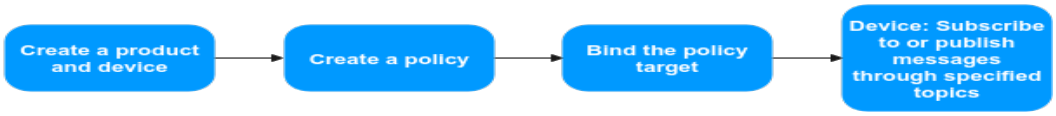
4. Subscribe to the topic.

```
device.getClient().subscribeTopic(new RawMessage("$oc/devices/" + device.getDeviceId() + "/user/wpy", new ActionListener() {
    @Override
    public void onSuccess(Object context) {
        System.out.println("subscribeTopic success: ");
    }
    @Override
    public void onFailure(Object context, Throwable var2) {
        System.out.println("subscribeTopic fail: " + var2);
    }
}, 0);
```

5.3.3 Custom Topics Not Starting with $oc

Process

Figure 5-41 Communications with custom topics not starting with $oc



NOTE

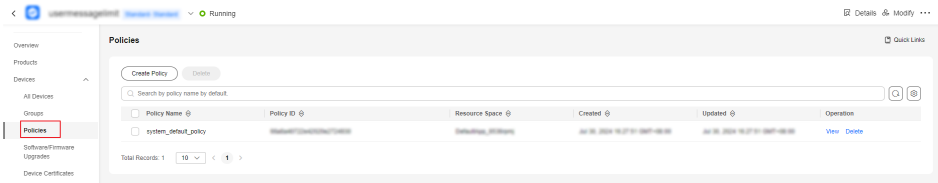
- By default, communications can be performed based on all custom topics not starting with **$oc**. The **system_default_policy** policy is added to newly created resource spaces by default, allowing all associated devices to publish or subscribe to messages through all topics. You can delete the policy if necessary.
- Policies are only used for communications with custom topics that do not start with **$oc**. For custom topics starting with **$oc**, their permissions are determined by product settings.
- Policies are not available in the following regions: CN South-Guangzhou, CN North-Beijing4, and CN East-Shanghai1.

Step 1 **Create a product** and **create a device** on the platform.

Step 2 Create a policy to control the topics subscription/publishing permissions. (optional)

1. Go to the policy page. Access the **IoTDA** service page and click **Access Console**. Click the target instance card. In the navigation pane, choose **Devices** > **Policies**.

Figure 5-42 Device policy - Access page



2. Create a policy. Click **Create Policy**, set policy parameters, and click **Generate**. The following figure shows how to publish and subscribe to messages through topic **/v1/test/hello**.

Table 5-14 Parameters

Parameter	Description
Resource Space	Select a resource space from the drop-down list box or **create one**.
Policy Name	Customize a value, for example, **PolicyTest**. Max: 128 characters. Use only letters, digits, underscores (_), and hyphens (-).
Resource	For MQTT topic publishing and subscription, **topic:** must be used as the parameter prefix. For example, to forbid the subscription to **/test/v1**, set this parameter to **topic:/test/v1**.
Operation	Options: **Publish** and **Subscribe**, meaning the topic publishing and subscription requests of MQTT devices.
Permission	Options: **Allowed** and **Denied**, meaning whether the permission to publish or subscribe to messages through a topic is assigned.

Step 3 Bind the policy target. A policy can be bound to resource spaces, products, or devices. The bound devices are allowed or disallowed to publish or subscribe to messages through a specific topic accordingly. (optional)

Figure 5-43 Device policy - Binding a device

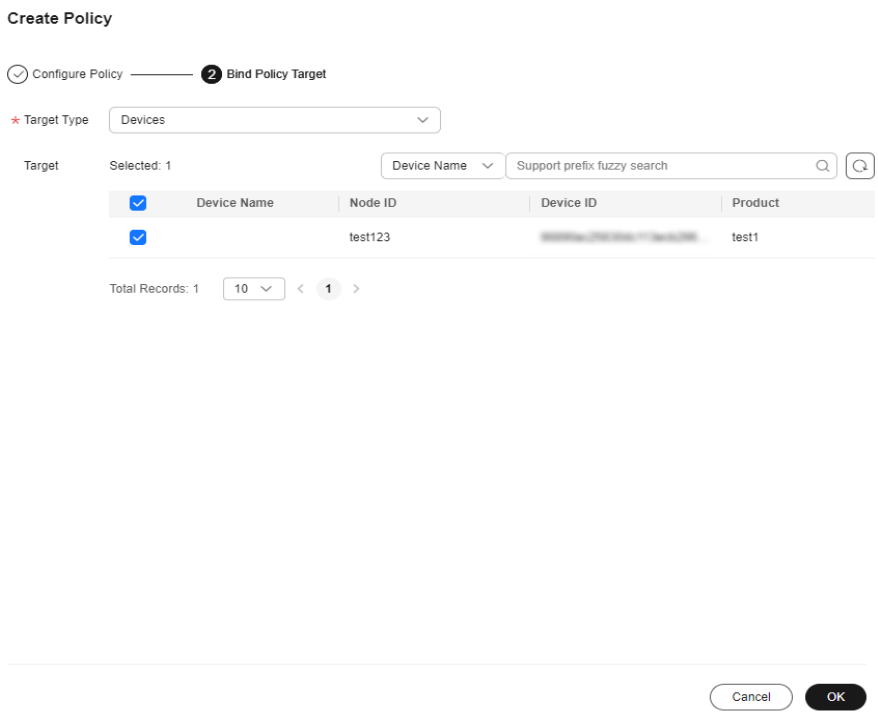


Table 5-15 Parameters

Parameter	Description
Target Type	You can set resource spaces, products, or devices as the target type. The three types can coexist. For example, product A and device C (under product B) can be bound to the same policy. ● Resource space: used for domain-based management of multiple service applications. After a resource space is bound to a policy, all devices in this resource space adopt the policy. You can also select multiple resource spaces for binding. ● Product: Generally, a product has multiple devices. After a product is bound to a policy, all devices of this product adopt the policy. Compared with the resource space, the binding scope is smaller. You can select products in different resource spaces for binding. ● Device: minimum unit for the target bound to a policy. You can select devices from different resource spaces and products for binding.
Target	After you select a policy target type, available targets are displayed in the **Target** area. Select targets as required.

Step 4 Use the device to subscribe to or publish messages through the specified topic. Only custom topics successfully bound in the policy can be used.

----End

Java SDK Usage on the Device Side

Devices can integrate the **device SDKs** provided by Huawei Cloud IoT to quickly connect to Huawei Cloud IoTDA and report messages. The following example uses the Java SDK to connect a device to IoTDA for publishing and subscribing to messages through the custom topic **/test/deviceToCloud**.

1. Configure the Maven dependency of the SDK on the device.

```
<dependency>
    <groupId>com.huaweicloud</groupId>
    <artifactId>iot-device-sdk-java</artifactId>
    <version>1.1.4</version>
</dependency>
```

2. Configure the SDK and device connection parameters on the device.

```
// Load the CA certificate of the IoT platform. For details about how to obtain the certificate, visit
https://support.huaweicloud.com/intl/en-us/devg-iothub/iot_02_1004.html.
URL resource = MessageSample.class.getClassLoader().getResource("ca.jks");
File file = new File(resource.getPath());

// The format is ssl://Domain name:Port number.
// To obtain the domain name, log in to the Huawei Cloud IoTDA console. In the navigation pane,
choose Overview and click Access Details in the Instance Information area. Select the access
domain name corresponding to port 8883.
String serverUrl = "ssl://localhost:8883";
// Device ID created on the IoT platform
String deviceId = "deviceId";
// Secret corresponding to the device ID
String deviceSecret = "secret";
// Initialize the device connection.
```

```
IoTDevice device = new IoTDevice(serverUrl, deviceId, deviceSecret, file);
if (device.init() != 0) {
    return;
}
```

3. Report a device message.

```
device.getClient().publishRawMessage(new RawMessage("/test/deviceToCloud", "hello", 1), new
ActionListener() {
    @Override
    public void onSuccess(Object context) {
        System.out.println("reportDeviceMessage success: ");
    }
    @Override
    public void onFailure(Object context, Throwable var2) {
        System.out.println("reportDeviceMessage fail: " + var2);
    }
});
```

4. Subscribe to the topic.

```
device.getClient().subscribeTopic(new RawMessage("/test/deviceToCloud", new ActionListener() {
    @Override
    public void onSuccess(Object context) {
        System.out.println("subscribeTopic success: ");
    }
    @Override
    public void onFailure(Object context, Throwable var2) {
        System.out.println("subscribeTopic fail: " + var2);
    }
}, 0);
```

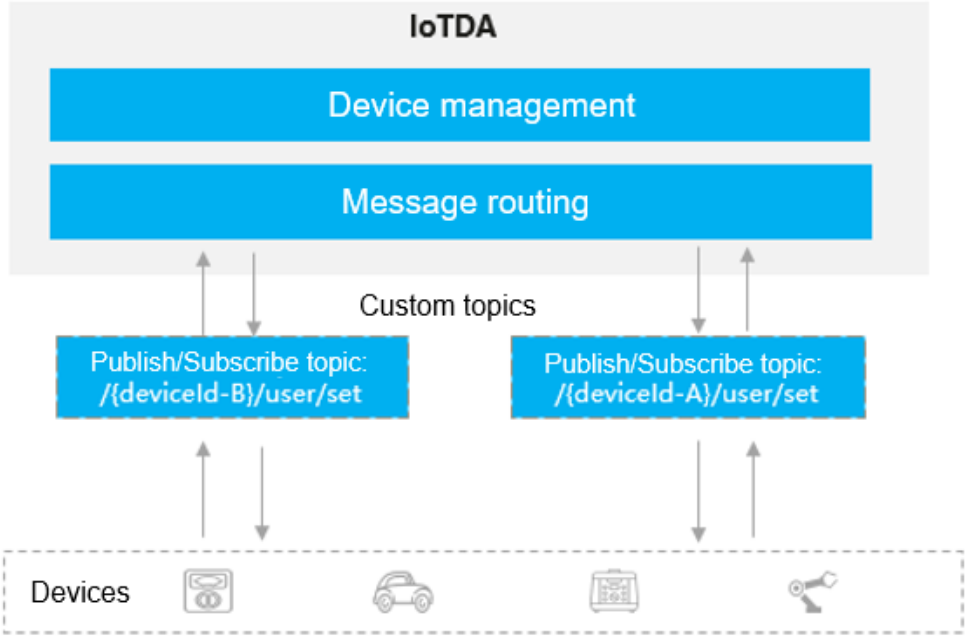
5.4 M2M Communications

5.4.1 Overview

Introduction

IoTDA supports MQTT-based machine-to-machine (M2M) communications. The platform processes the connection and communication requests from devices, so you can focus on service implementation. With M2M communications, devices can communicate with each other flexibly.

Figure 5-44 Service flow



NOTE

- During M2M communications, messages sent through the PUB interface and messages received through the SUB interface are counted as charging messages. No additional fee is generated.

Scenarios

- Instant messaging scenario where a sender and recipient communicate with each other.
- Smart home scenario where messages are exchanged between mobile apps and smart devices.
- Device linkage scenario where devices exchange data and communicate with each other.

Constraints

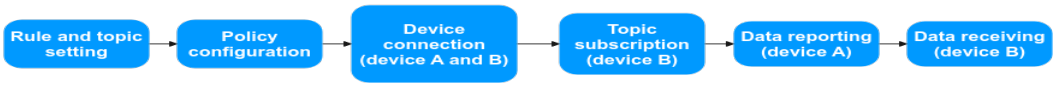
- Not available for the basic edition users.
- Only available for devices connected using MQTT.
- Max. topic length: 128 bytes.
- Max. MQTT message size: 1 MB.
- Max. subscribers of a topic: 1,000 devices.
- An MQTT device can subscribe to up to 100 topics (50 custom topics at most).
- Max. upstream messages for an MQTT device: 50 messages per second.

5.4.2 Usage

Process

This section takes one-to-one communications between devices as an example.

Figure 5-45 M2M communication process



Step 1 Rule and topic setting: Create an M2M data forwarding rule and set a forwarding topic on the console.

Step 2 Policy configuration: On the console, configure policies to allow devices that send and receive data to publish and subscribe to data.

Step 3 Device authentication: Devices A and B initiate connection authentication requests. For details about authentication parameters, see **Device Connection Authentication**.

Step 4 Topic subscription: Device B subscribes to a cloud-based topic that is set during the data forwarding rule creation. If the subscription is successful, the platform returns an **ACK** message.

Step 5 Data reporting: Device A publishes data through the cloud-based topic. If the publishment is successful, the platform returns an **ACK** message.

Step 6 Data receiving: If the data forwarding is successful, device B receives the data from device A.

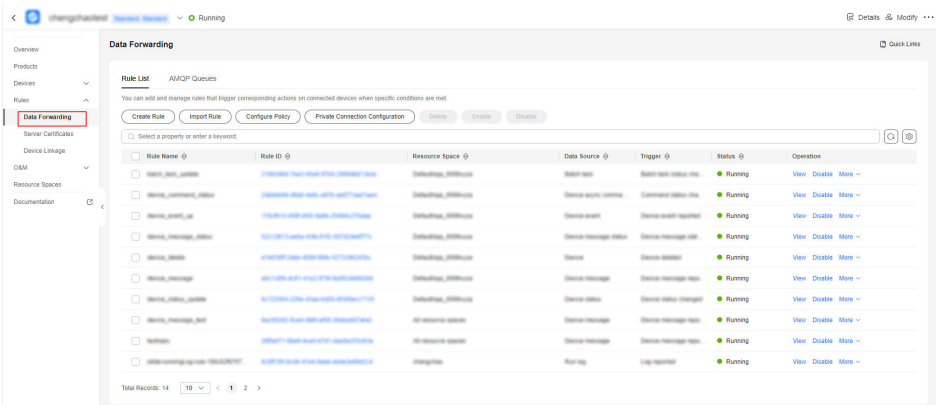
----End

Procedure

The following example describes how to create a data forwarding rule on the platform. You can modify the rule for different scenarios.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card. In the navigation pane, choose **Rules** > **Data Forwarding**.

Figure 5-46 Data forwarding - List



Step 2 Click **Create Rule**, configure the parameters based on the service requirements, and click **Create Rule**. The following figure shows an example.

Figure 5-47 Creating a forwarding rule - M2M

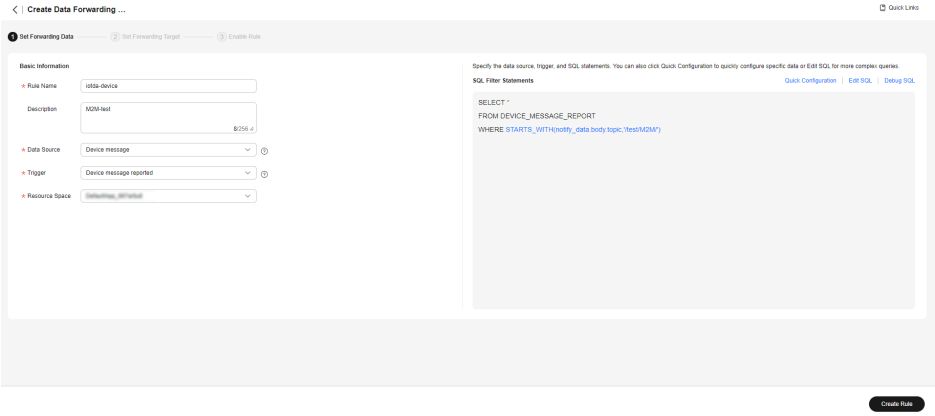


Table 5-16 Parameters for creating a data forwarding rule

Parameter	Description
Rule Name	Customize a value, for example, **test**. The value can contain up to 256 characters. Only letters, digits, and special characters (_?'#().,&%@!-) are allowed.
Description	Description of the rule, which is user-defined.
Data Source	Data source of the forwarding rule. You can select multiple data sources from the drop-down list. In the M2M scenario, select **Device message**.
Trigger	Available trigger events vary with the data sources. In the M2M scenario, select **Device message reported**.
Resource Space	Select an existing resource space from the drop-down list or create a **Resource Space** first.
SQL Filter Statements	You can use SQL statements to filter data. For details, see **SQL Statements**. In the example figure, **notify_data.body.topic IN ('/test/M2M')** in the **WHERE** statement indicates that only the data whose topic is **/test/M2M** will be forwarded.

Step 3 Go to the second stage. You can set the data forwarding target, topic, and cache time. In M2M scenario, select **Device** for **Forwarding Target**, set the parameters based on service requirements, and click **OK**.

Figure 5-48 Creating a forwarding target - to a device

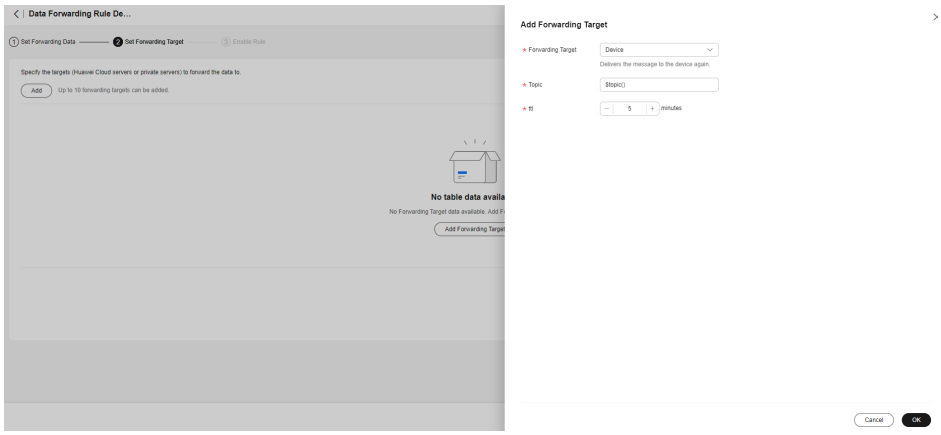
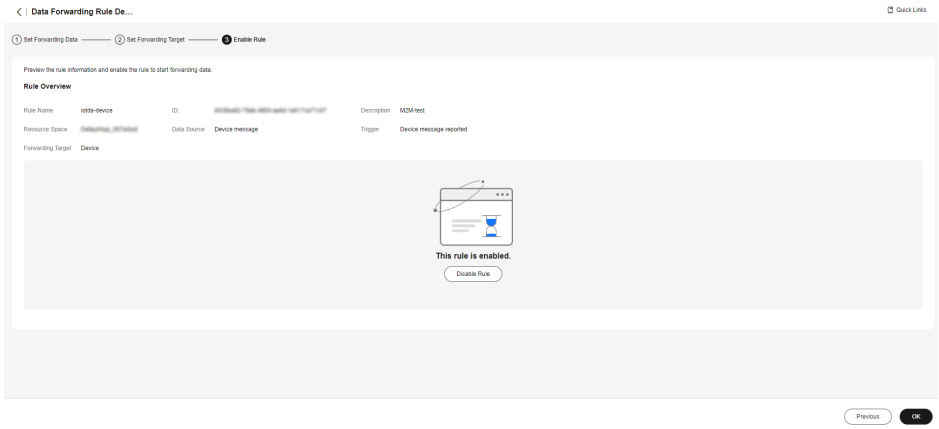


Table 5-17 Parameters for setting a forwarding target

Parameter	Description
Forwarding Target	Select a forwarding target from the drop-down list. In the M2M scenario, select **Device**.
Topic	Max: 128 characters. It can start with dollar signs ($) and slashes (/) but cannot end with them. Use only digits, letters, and the following characters: () ',-.:=@;_!*'%?+\.
ttl	The platform caches messages when a device is offline, and delivers them when the device comes back online. **ttl** is the data cache time whose value ranges from 0 to 1,440 (one day) minutes and must be a multiple of 5. When the value is set to **0**, data is not cached.

Step 4 Go to the third stage and enable the rule.

Figure 5-49 Enabling a rule - Forwarding data to a device



----End

5.4.3 Example

Preparations

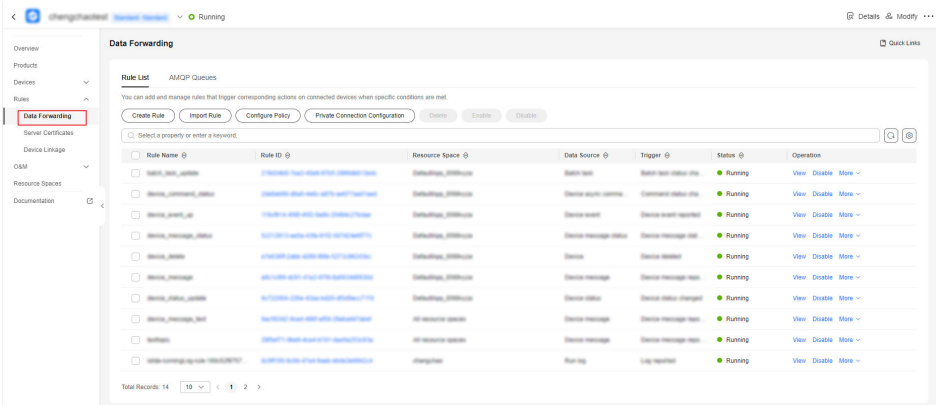
1. Creating a product and device
   - Create a product. Access the **IoTDA** service page and click **Access Console**. Click the target instance card. Choose **Products** in the navigation pane and click **Create Product**. Set the parameters as prompted and click **OK**. For details, see **Creating a Product**.
   - Create a device. On the IoTDA console, choose **Devices** > **All Devices** in the navigation pane, and click **Register Device**. Set the parameters as prompted and click **OK**. For details, see **Registering an Individual Device**.

Configuring a Data Forwarding Rule

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

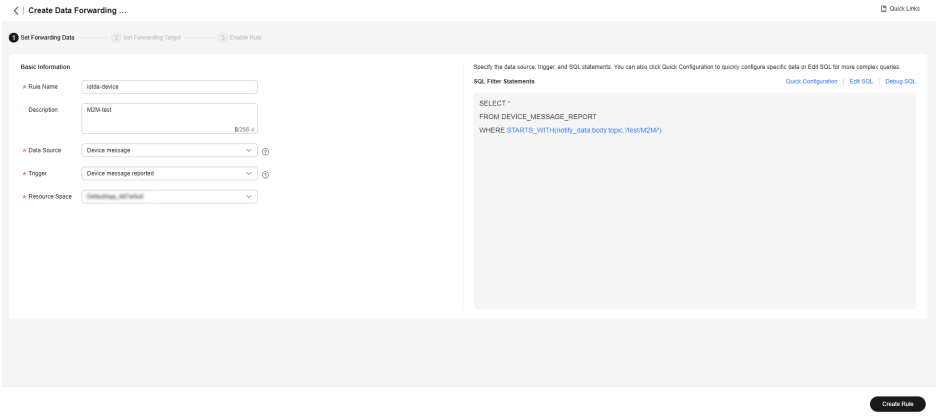
Step 2 In the navigation pane, choose **Rules** > **Data Forwarding**.

Figure 5-50 Data forwarding - List



Step 3 On the **Rule List** tab page, click **Create Rule**. Set the rule parameters and forwarded data, and configure the data filtering statement as follows: **STARTS_WITH(notify_data.body.topic,'/test/M2M/')**.

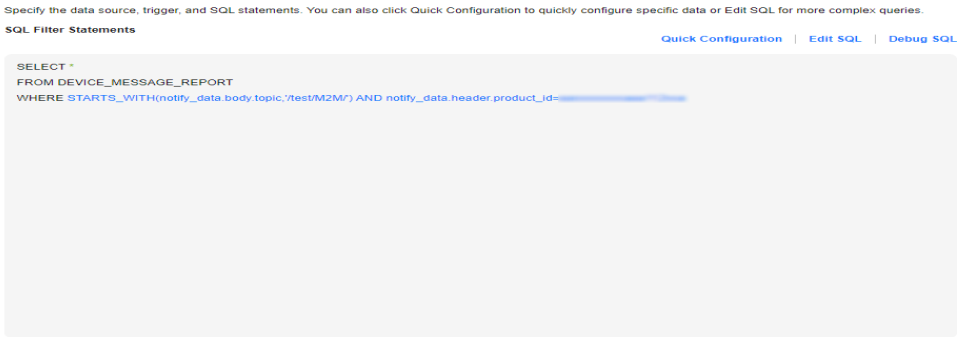
Figure 5-51 Creating a forwarding rule - M2M



NOTE

- Configuration of **Figure 2**: For all devices in the resource space *XXX*, when they report messages with topics containing **/test/M2M/**, the forwarding rule will be triggered and the messages will be forwarded to the specified forwarding target.
- To forward the data reported by a specified device, add **AND notify_data.header.device_id='${*Device ID*}'** to the SQL statements.
- To forward the data reported by a specified **product**, add **AND notify_data.header.product_id='${*Product ID*}'** to the SQL statements.

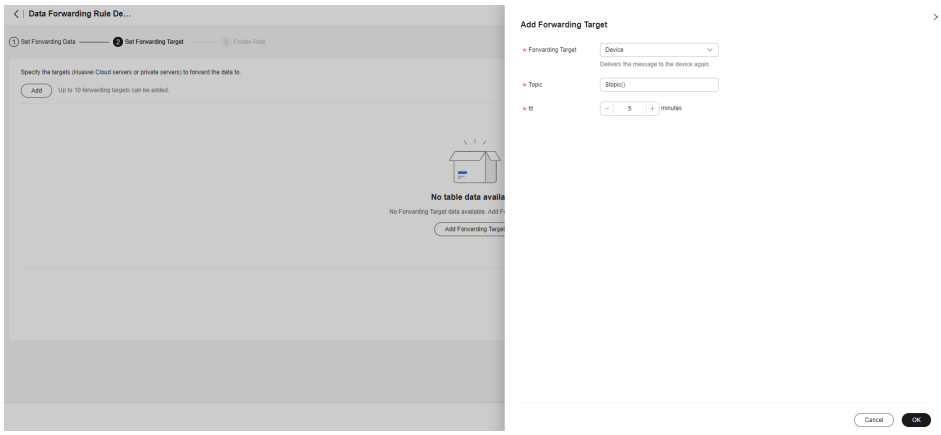
Figure 5-52 Data forwarding - M2M_SQL statement example



- For details about SQL settings, see **SQL Statements**.

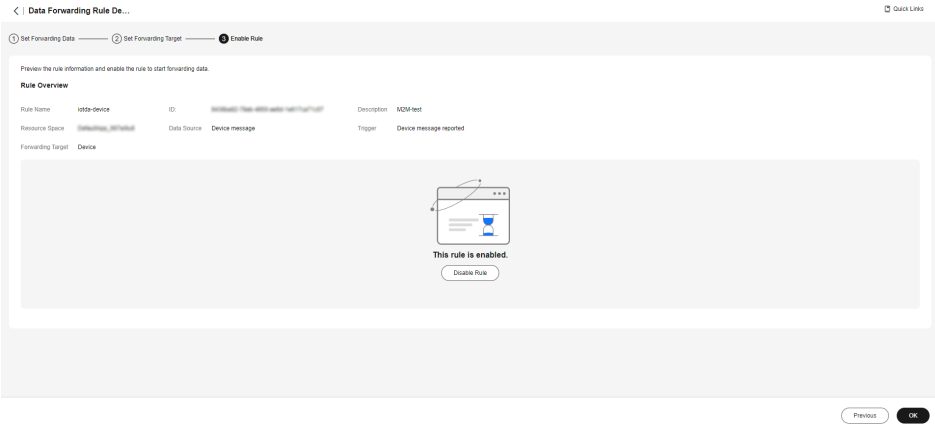
Step 4 Set the forwarding target. Click **Add**. Select **Device** for **Forwarding Target**, set **Topic** to **$topic()** (the topic remains unchanged after forwarding), and set **ttl** to 5 minutes (data is cached for 5 minutes). Click **OK**.

Figure 5-53 Creating a forwarding target - to a device



Step 5 Click **Enable Rule** in the middle of the page.

Figure 5-54 Enabling a rule - Forwarding data to a device

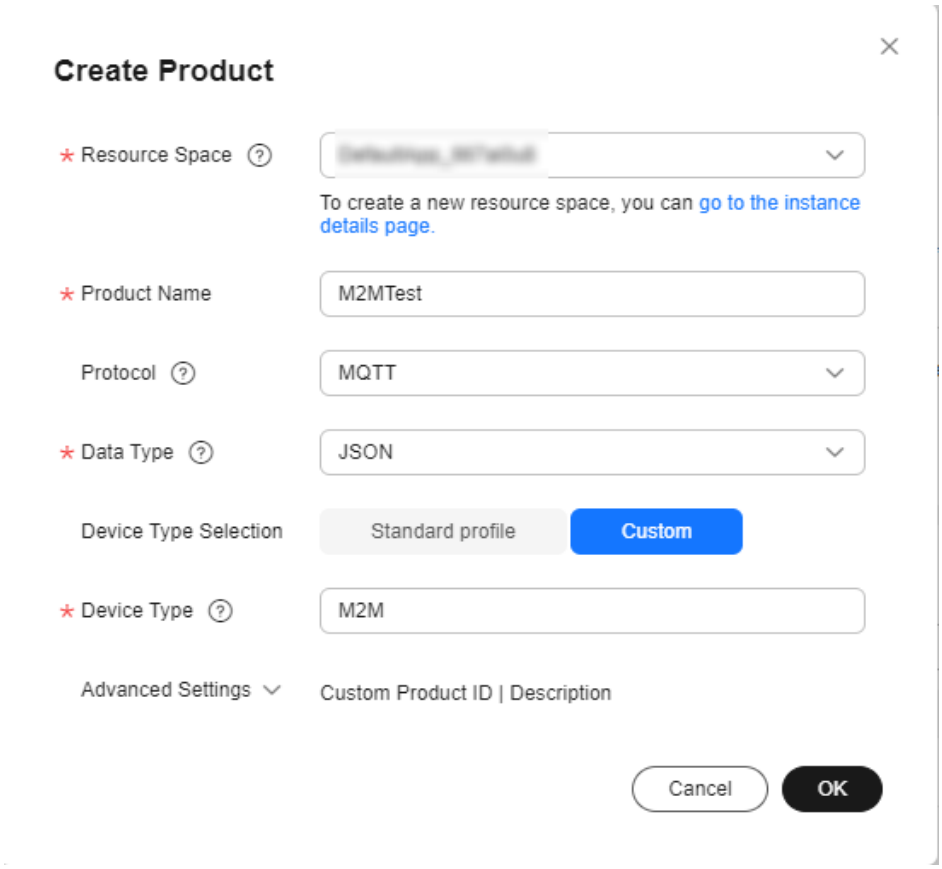


----End

Test

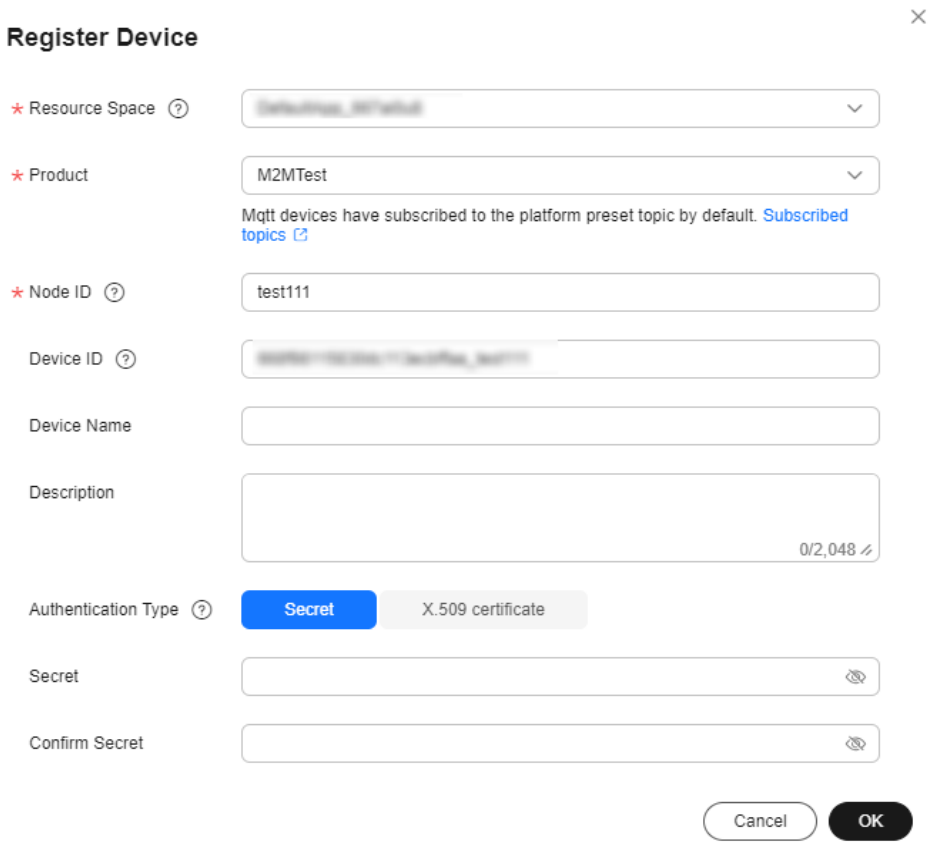
1. To use the M2M communication function, perform the following steps:
   a. Access the **IoTDA** service page and click **Access Console**. Click the target instance card.
   b. **Create a product.** Select **MQTT** for **Protocol**.

   Figure 5-55 Creating a product - M2M



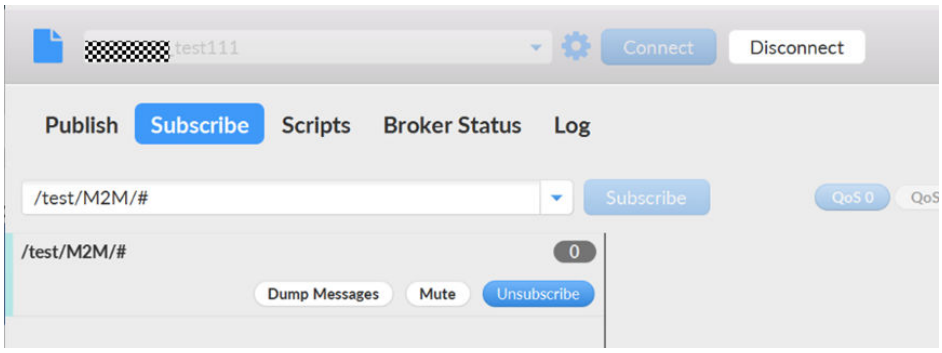
2. Register device A (**test111**) and device B (**test222**) under the product created in step **1**. For details, see **Registering an Individual Device**.

   Figure 5-56 Device - Registering an M2M device



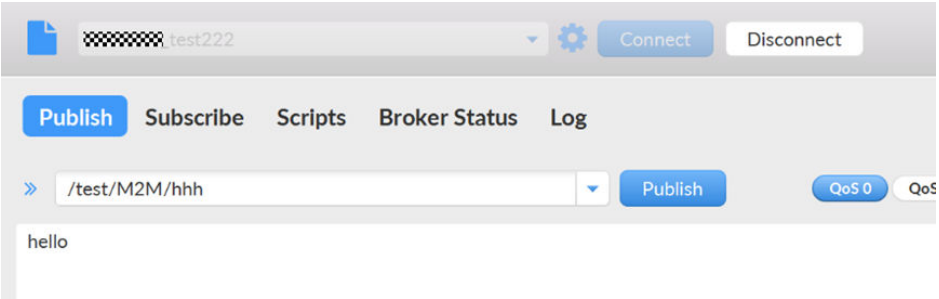
3. This section uses MQTT.fx as an example to describe how to implement M2M communications. You can also test based on your service requirements.
   a. Open two MQTT.fx to simulate devices A (**test111**) and B (**test222**).
   b. On the **Subscribe** page of device B, enter the topic **/test/M2M/#** and click **Subscribe**.

      Figure 5-57 Entering a topic on the **Subscribe** page of device B



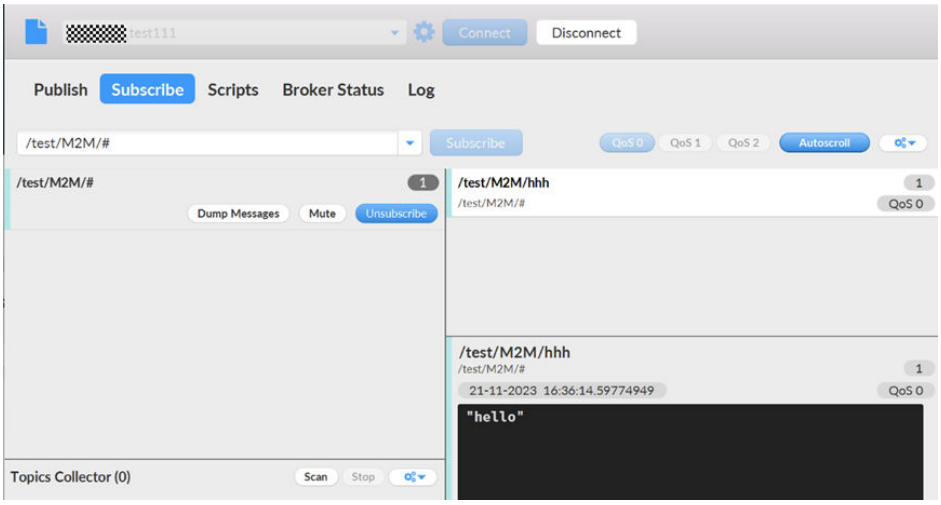
   c. Let device A send a message to device B. On the **Publish** page of device A, enter the topic **/test/M2M/${*Any word*}**. Enter the message to be sent (for example, **hello**) in the text box, and click **Publish**.

Figure 5-58 Entering a message on the **Publish** page of device A



On the **Subscribe** page of device B, you can see the received message, as shown in the following figure.

Figure 5-59 Subscribe page of device B



5.5 Device Topic Policies

5.5.1 Overview

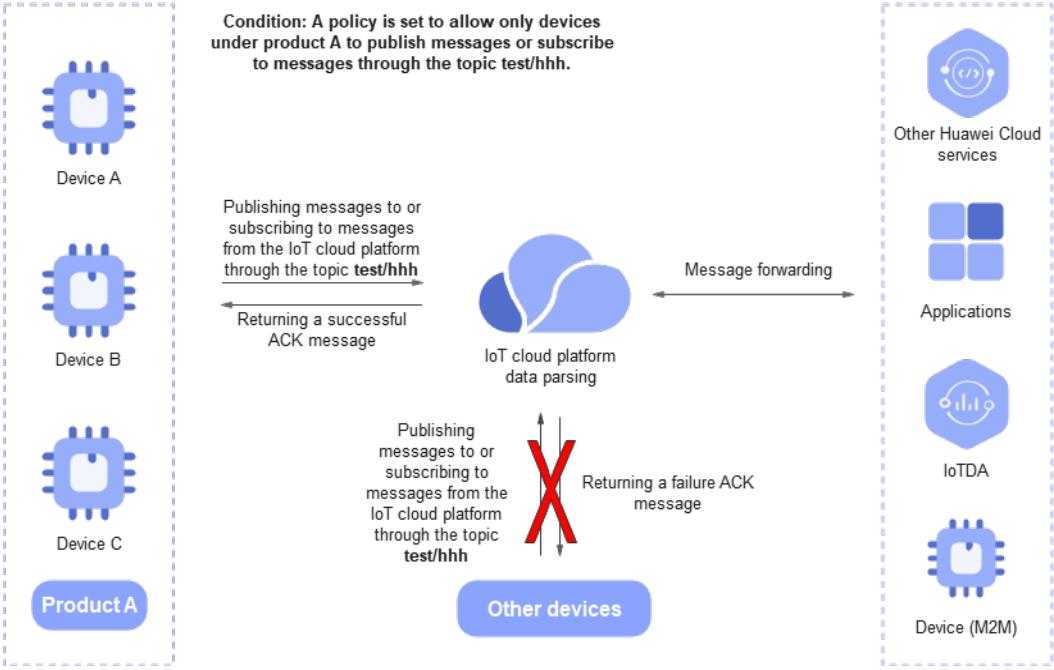
Overview

IoTDA provides device topic policies, with which you can implement flexible role-based access control, and authorize clients to publish or subscribe to messages through topics not starting with **$oc**. You can manage the topic-based data publishing and subscription permissions of devices, products, or groups, improving communications security. Device policies are mainly used for protocols used in data publishing and subscription mechanisms, for example, MQTT and MQTTS on the device side. Currently, this feature is available for users in invitation-only regions of south China and international regions.

NOTICE

The **system_default_policy** policy is added to the newly created resource space by default, which allows devices in this resource space to publish or subscribe to messages through topics (not starting with **$oc**) of all devices under all resource spaces. You can delete the policy if necessary.

Figure 5-60 Conceptual diagram of topic policies



Scenarios

- Group-based communications: For example, devices A, B, and C belong to a group, and only devices A, B, and C are allowed to subscribe to the topic of the group.
- Region-based communications: Regions are divided based on the data publishing and subscription permissions. Only devices of the same region can communicate with each other.

Restrictions

- Max. policies for a tenant: 50.
- Applicable topics: custom topics that do not start with **$oc**.
- Max. policy file size: 10 KB. Max. files configured for a policy: 10.
- Max. policies configured for a device or product: 5.
- Max. topics subscribed by a device (client): 50.
- Max. topic length: 128 bytes.
- Supported QoS: QoS 0 and QoS 1.

5.5.2 Content

IoTDA provides device topic policies, with which you can authorize clients to publish or subscribe to messages through custom topics not starting with **$oc**, enhancing communications security. Device policies are mainly used for protocols used in data publishing and subscription mechanisms, for example, MQTT and MQTTS on the device side. Currently, this feature is available for users in invitation-only regions of south China.

Policy Wildcards

You can use wildcards for policies. An asterisk (*) indicates any combination of characters, and a question mark (?) indicates a character of any kinds. Plus signs (+) and number signs (#) do not have special meanings.

Table 5-18 Policy wildcards

Wild card	MQTT Wildcard	Applicable to Policy	Example MQTT Topic	Example MQTT Topic for Policy
#	Yes	No	test/#	Not applicable. The number sign (#) is regarded as a character without special meaning.
+	Yes	No	test/+/some	Not applicable. The plus sign (+) is regarded as a character without special meaning.
*	No	Yes	Not applicable. The asterisk (*) is regarded as a character without special meaning.	test/* test/*/some
?	No	Yes	Not applicable. The question mark (?) is regarded as a character without special meaning.	test/????/some test/set?????/some

Table 5-19 Example usage of wildcards in policies

Target Topic	Topic Definition in Policy	Description
Example topics: **test/topic1/some** **test/topic2/some** **test/topic3/some**	topic:test/topic?/some	Common points: **test/topic** + *a character* + **/some**. In the policy definition, a question mark (?) indicates a character. Therefore, the policy topic can be defined as **topic:test/topic?/some**.
Example topics: **test/topic1/pub/some** **test/topic2/sub/some** **test/topic3/some**	topic:test/topic*/some	Common points: **test/topic** + *one or more characters* + **/some**. In policy definition, the asterisk (*) indicates multiple or one character. Therefore, the policy topic can be defined as **topic:test/topic*/some**.

Policy Variables

You can use a policy variable as a placeholder for resource or condition key to filter topics when defining policy resource. During MQTT topic verification, the system replaces the variable with the corresponding ID for matching.

Variables are prefixed with a dollar sign ($), followed by a pair of braces ({}), which contain the variable name in the request. The following table lists the supported variables. Assume that the client ID of an MQTT device is **test_clientId**, the product ID is **test_productId**, and the device ID is **test_deviceId**.

Table 5-20 Policy variables

Policy Variable	Description	Example MQTT Topic	Example MQTT Topic for Policy
${devices.deviceId}	Device ID	test/test_deviceId/topic	test/${devices.deviceId}/topic
${devices.clientId}	Client ID	test/test_clientId/topic	test/${devices.clientId}/topic
${devices.productId}	Product ID	test/test_productId/topic	test/${devices.productId}/topic

Table 5-21 Example usage of variables in policies

Scenario	Example Policy Topic Definition	Description
Distinguishing topics by device	test/${devices.deviceId}/topic	Devices can subscribe to or publish messages through topic **test/*${Device ID}*/topic** with their data isolated.
Distinguishing topics by device and time segment	test/${devices.clientId}/topic	Devices can subscribe to or publish messages through topic **test/*${Device client ID}*/topic**. Different from **deviceId**, **clientId** carries a timestamp to distinguish time segments.

Policy Priority

If multiple policies with different effect are bound to the same device, the policy with higher priority (**Denied** over **Allowed**) takes effect.

For example, a device has two policies: policy 1 and policy 2. Policy 1 denies subscription to topic A, and policy 2 allows subscription to topic A. When the device subscribes to topic A, the platform rejects the subscription request from the device.

Table 5-22 Policy priority

Topic	Policy 1	Policy 2	Effective Policy
test/topic	"effect": "ALLOW", "resources": ["topic:test/topic"]	"effect": "DENY", "resources": ["topic:test/topic"]	Denied

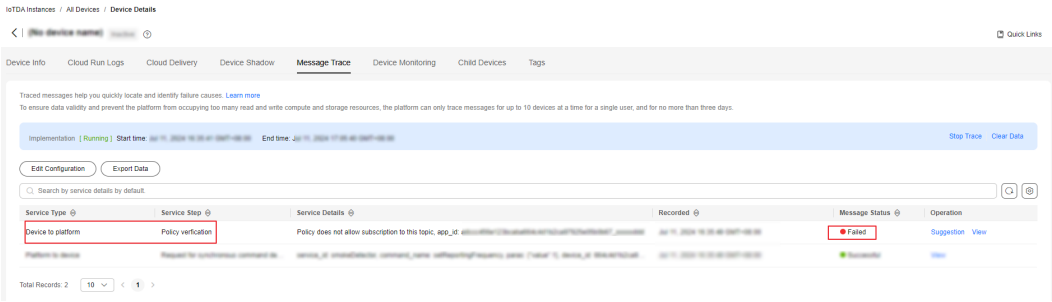
Policy Topic Constraints

1. Max. length: 128 bytes.
2. Unallowed wildcards for topic publishing: number signs (#) and plus signs (+).
3. Consecutive slashes (/) are not allowed, for example, **////test/**.
4. Max. slashes (/) in a topic: 7.

CAUTION

If the topic to be published or subscribed to does not meet the preceding requirements, the subscription or publishing request will be rejected. On the device details page, choose the **Message Trace** tab. The error information is displayed.

Figure 5-61 Message tracing - Verifying a policy



5.5.3 Usage

Process

Figure 5-62 Device policy usage process

Step 1 Policy creation: A user creates a device policy on the console. For details, see **Examples**.

Step 2 Device authentication: An MQTT device initiates a connection authentication request. For details about authentication parameters, see **Device Connection Authentication**.

Step 3 Message subscription or publishing: The device applies to publishing or subscribing to messages through a specific topic on the cloud server.

Step 4 Policy authentication: The cloud server filters topics subscribed to or published by the device based on the policy. If the device is not allowed to subscribe to the topic, the cloud server returns a failure ACK message and the subscription fails. Otherwise, a successful ACK message is returned, indicating that the subscription is successful.

Step 5 Data push: Messages successfully published by the device can be pushed to the application through data transfer.

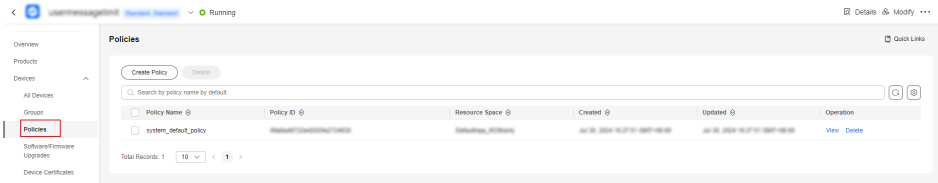
----End

Procedure

The following example describes how to set topic policies and bind policy targets on the IoTDA console for MQTT device.

1. Go to the policy page. Access the **IoTDA** service page and click **Access Console**. Click the target instance card. In the navigation pane, choose **Devices** > **Policies**.

   Figure 5-63 Device policy - Access page

2. Create a policy. Click **Create Policy**, set policy parameters based on service requirements, and click **Generate**. The following figure shows the example parameter values.

   Figure 5-64 Device policy - Creating a policy

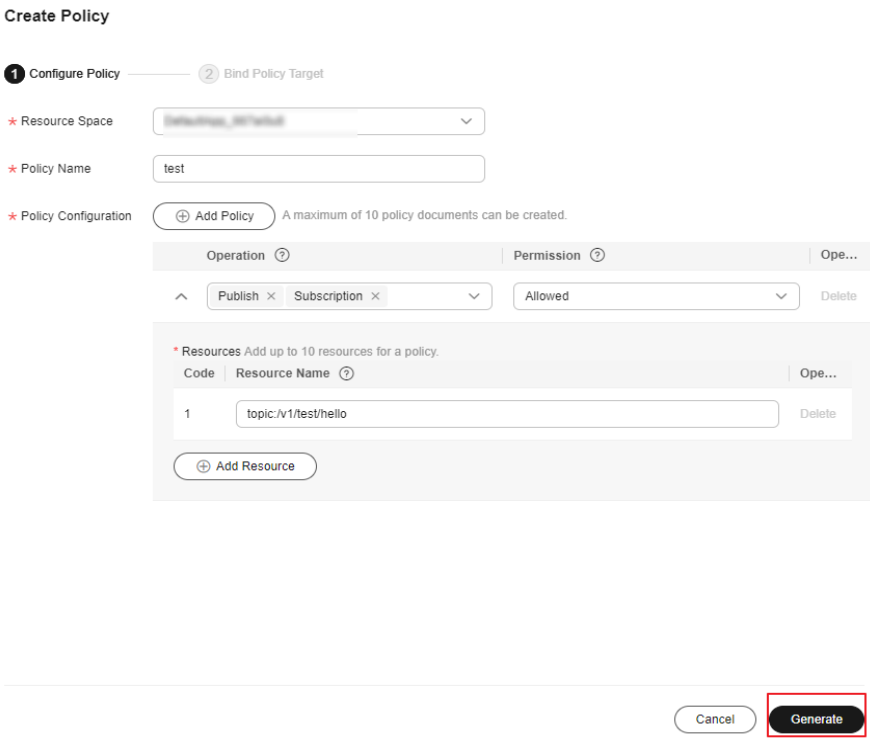
   

Table 5-23 Parameter description

Parameter	Description
Resource Space	Select a resource space from the drop-down list box or **create one**.

Policy Name	Customize a value, for example, **PolicyTest**. The value is a string of no more than 128 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed.
Resource	For MQTT topic publishing and subscription, **topic:** must be used as the parameter prefix. For example, to forbid the subscription to **/test/v1**, set this parameter to **topic:/test/v1**.
Operation	Options: **Publish** and **Subscribe**, meaning the topic publishing and subscription requests of MQTT devices.
Permission	Options: **Allowed** and **Denied**, meaning whether the permission to publish or subscribe to messages of a topic is assigned.

3. Bind the policy target. A policy can be bound to resource spaces, products, or devices. The bound devices are allowed or disallowed to publish or subscribe to messages through a specific topic accordingly.

Figure 5-65 Device policy - Binding a device

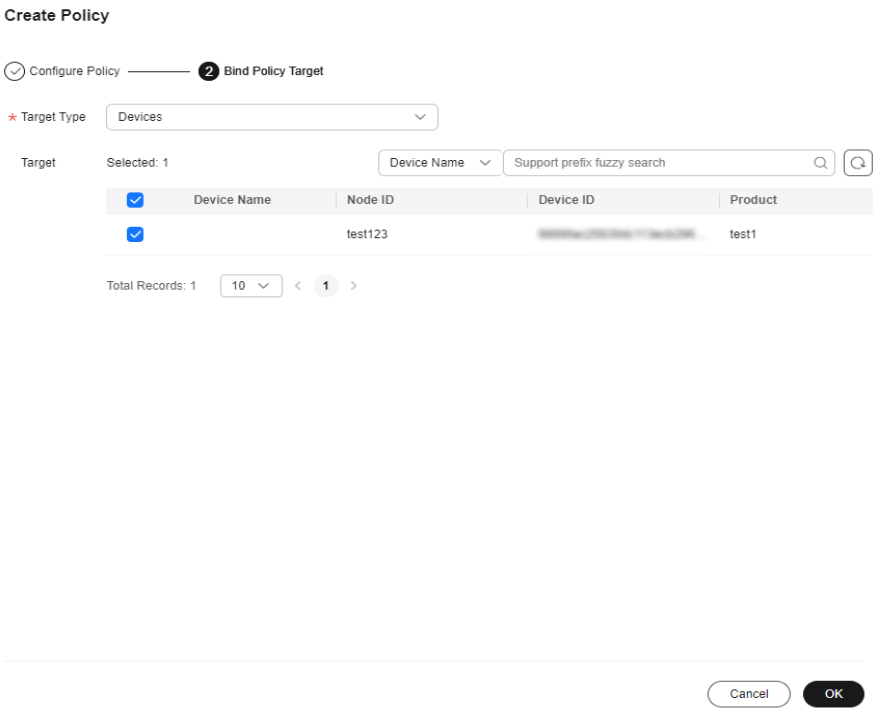


Table 5-24 Parameter description

Parameter	Description

Target Type	You can set resource spaces, products, or devices as the target type. The three types can coexist. For example, product A and device C (under product B) can be bound to the same policy. ● Resource space: used for domain-based management of multiple service applications. After a resource space is bound to a policy, all devices in this resource space adopt the policy. You can also select multiple resource spaces for binding. ● Product: Generally, a product has multiple devices. After a product is bound to a policy, all devices of this product adopt the policy. Compared with the resource space, the binding scope is smaller. You can select products in different resource spaces for binding. ● Device: minimum unit for the target bound to a policy. You can select devices from different resource spaces and products for binding.
Target	After you select a policy target type, available targets are displayed in the **Target** area. Select targets as required.

5.5.4 Examples

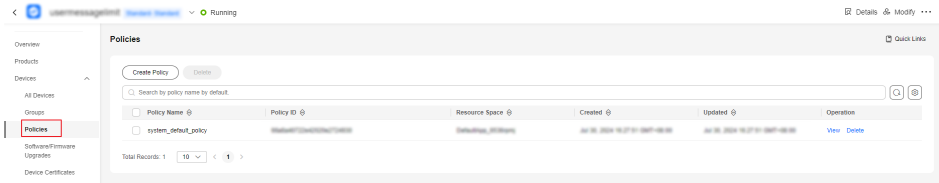
Scenarios

- **Scenario 1: Allowing or Denying the Message Publishing Through a Specific Topic**
- **Scenario 2: Using Policy in E2E (M2M) Communications**

Scenario 1: Allowing or Denying the Message Publishing Through a Specific Topic

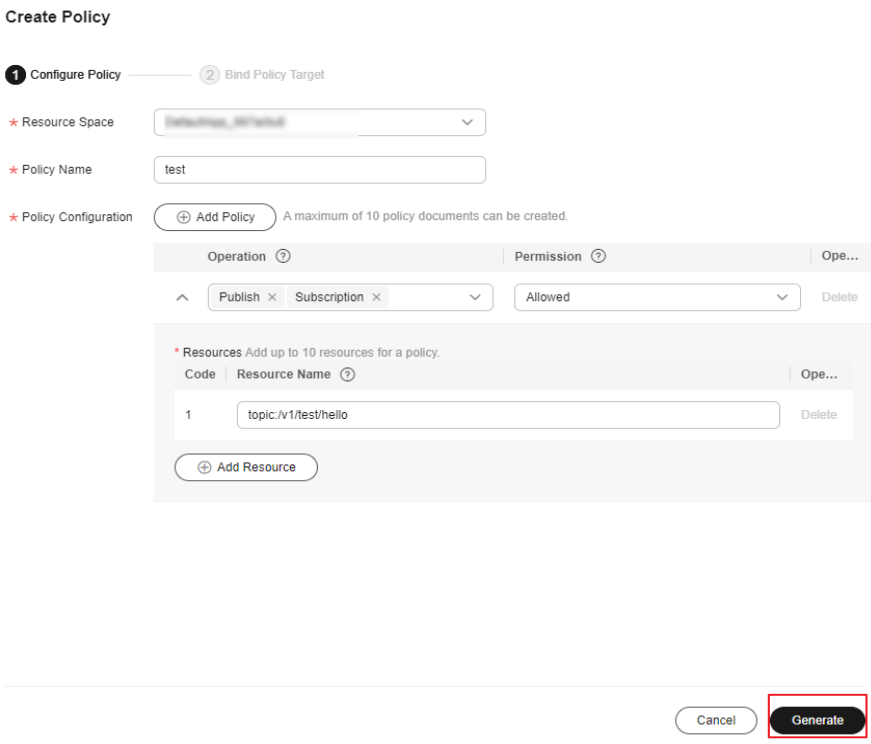
1. Create a product and device.
   - Create a product. Access the **IoTDA** service page and click **Access Console**. Click the target instance card. Choose **Products** in the navigation pane and click **Create Product**. Set the parameters as prompted and click **OK**. For details, see **Creating a Product**.
   - Create a device. On the IoTDA console, choose **Devices** > **All Devices** in the navigation pane, and click **Register Device**. Set the parameters as prompted and click **OK**. For details, see **Registering an Individual Device**.
2. Create a policy.
   - Access the **IoTDA** service page and click **Access Console**. Click the target instance card.
   - Choose **Devices** > **Policies** in the navigation pane.

Figure 5-66 Device policy - Access page



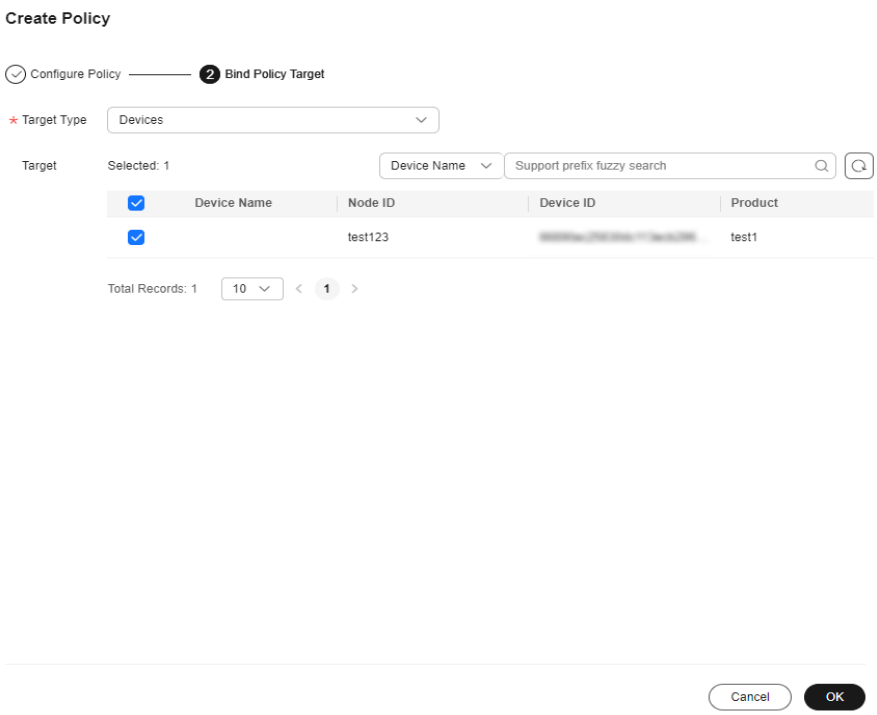
– Click **Create Policy**, set policy parameters, and click **Generate**. The application scope of the policy is the resource space (appId). Resources starting with **topic:** indicate topics in MQTT communications and are used for publishing and subscription. In this example, the topic that can be published and subscribed to is **/v1/test/hello**.

Figure 5-67 Device policy - Creating a policy



– Bind the policy. In this example, set **Target Type** to **Devices** and select the devices to which the policy is to be bound.

Figure 5-68 Device policy - Binding a device



– Verify the policy.

  i. Obtain connection parameters. In the navigation pane, choose **Devices** > **All Devices**, find the devices bound to the policy in the preceding step, go to the device details page, and view the connection parameters.

  Figure 5-69 Device - Device details

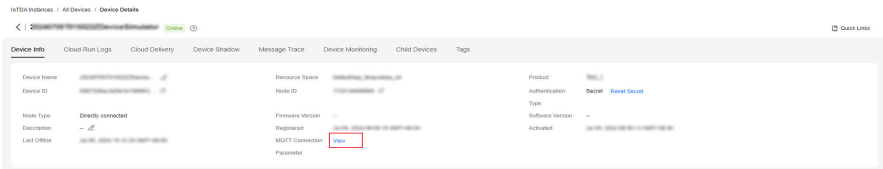
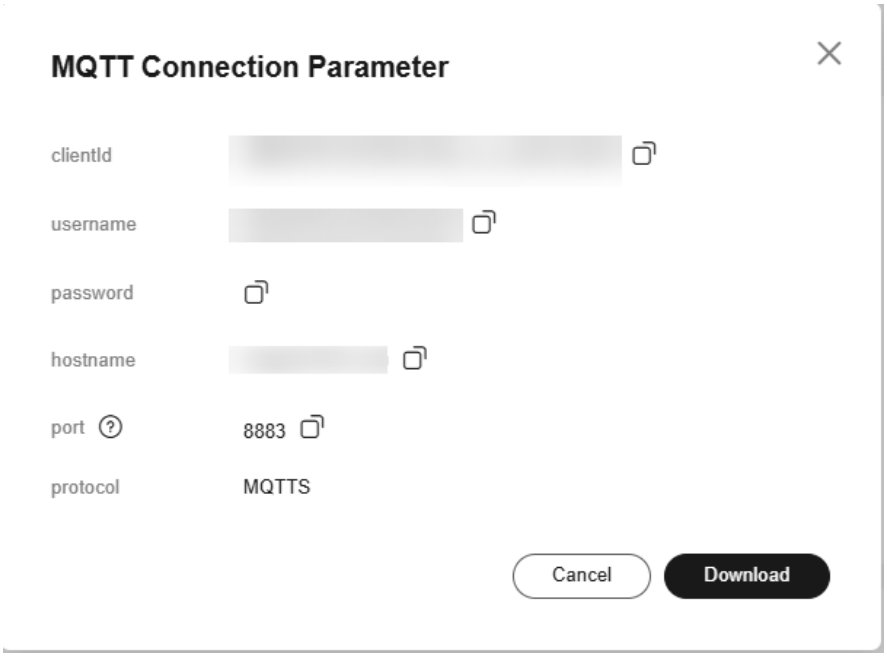


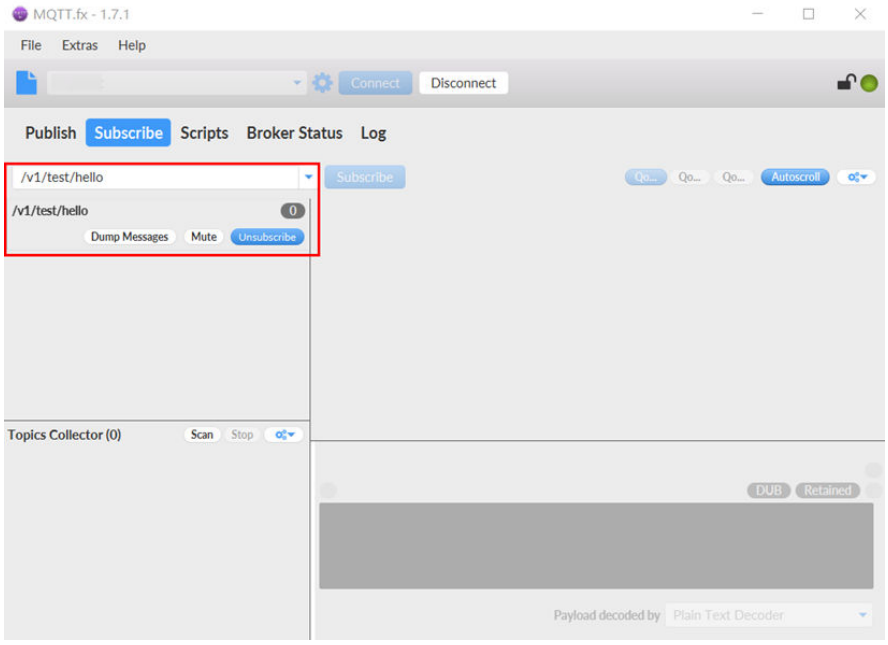
Figure 5-70 Device - Device details - MQTT connection parameters



ii. Use the MQTT.fx tool to connect to the cloud platform. Open the MQTT.fx tool, set authentication parameters for the devices bound to the policy in the preceding step, click **Apply**, and click **Connect** for connection authentication.

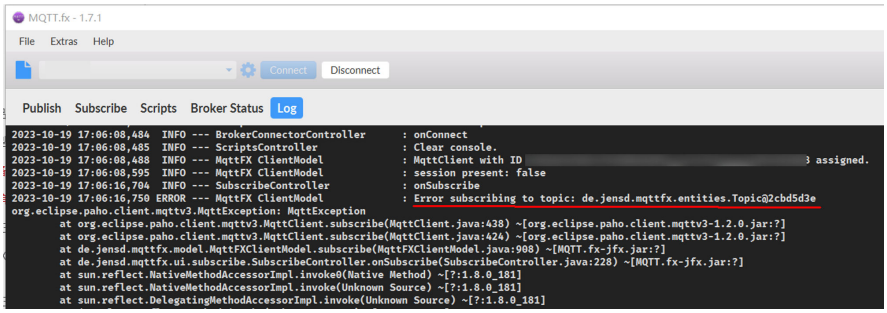
iii. Use the device to subscribe to the allowed topic **/v1/test/hello**. The subscription is successful.

Figure 5-71 Successful subscription



iv. Use the device to subscribe to another topic **/v2/test/hello**. The subscription failed.

Figure 5-72 Failed subscription

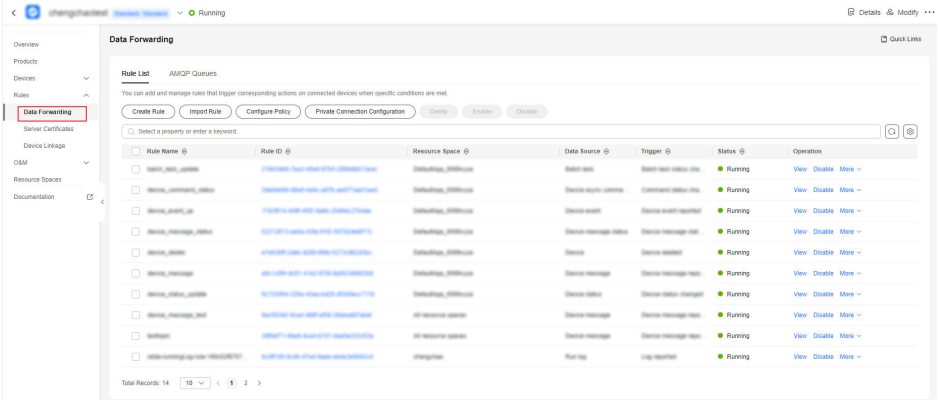


Scenario 2: Using Policy in E2E (M2M) Communications

In this example, you can enable device A under product A and all devices under product B to communicate with each other, and only allow them to subscribe to or publish messages through topics starting with **/test/M2M/**.

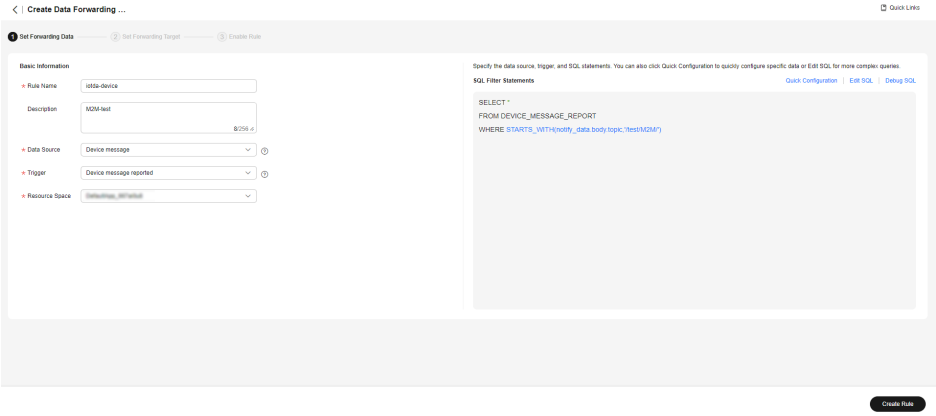
1. Create a product and device.
   - Create a product. Access the **IoTDA** service page and click **Access Console**. Click the target instance card. Choose **Products** in the navigation pane and click **Create Product**. Set the parameters as prompted and click **OK**. For details, see **Creating a Product**.
   - Create a device. On the IoTDA console, choose **Devices** > **All Devices** in the navigation pane, and click **Register Device**. Set the parameters as prompted and click **OK**. For details, see **Registering an Individual Device**.
2. Configure a data forwarding rule.
   - Access the **IoTDA** service page and click **Access Console**. Click the target instance card. In the navigation pane, choose **Rules** > **Data Forwarding**.

     Figure 5-73 Data forwarding - List

   - Click **Create Rule**, set the parameters as required, and click **Create Rule**. Set the SQL filter statement to **STARTS_WITH(notify_data.body.topic,'/test/M2M/')**.

Figure 5-74 Creating a forwarding rule - M2M



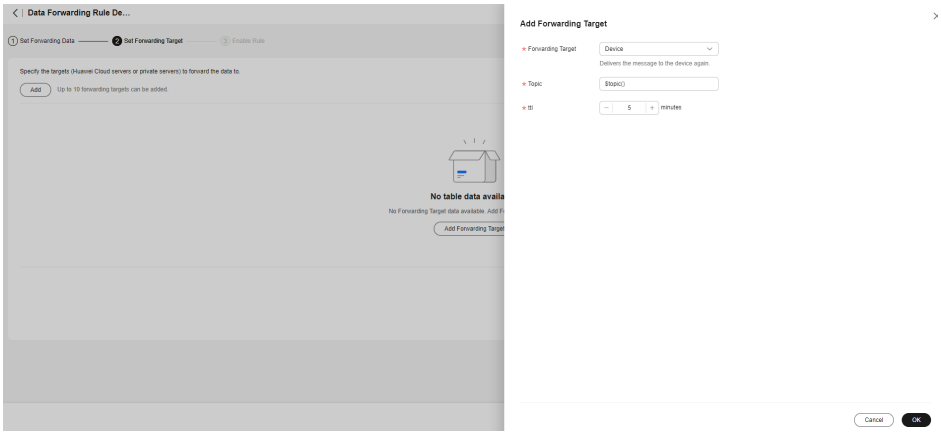
NOTE

For details about how to use SQL filter statements, see **SQL Statements**.

STARTS_WITH(notify_data.body.topic,'/test/M2M/') indicates that data with topics starting with **/test/M2M/** is filtered.

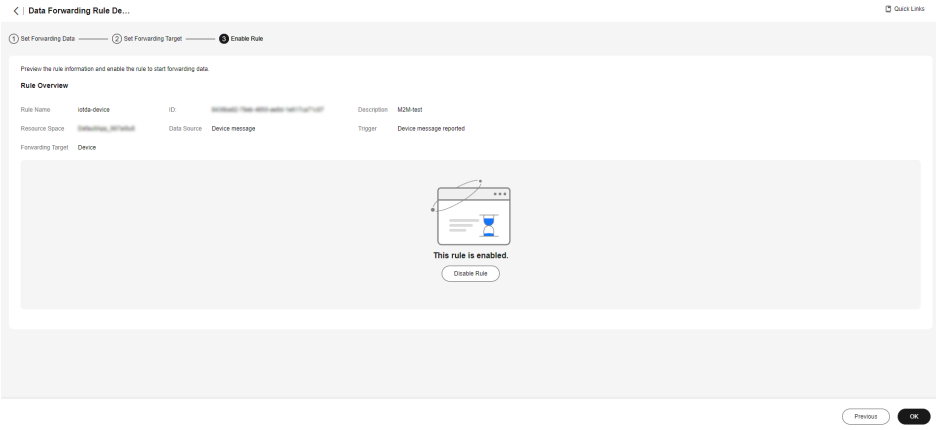
– Set the forwarding target. Set **Forwarding Target** to **Device**, set **Topic** to **$topic()** (indicating that the forwarded topic remains unchanged and the original topic is delivered), and click **OK**.

Figure 5-75 Creating a forwarding target - to a device



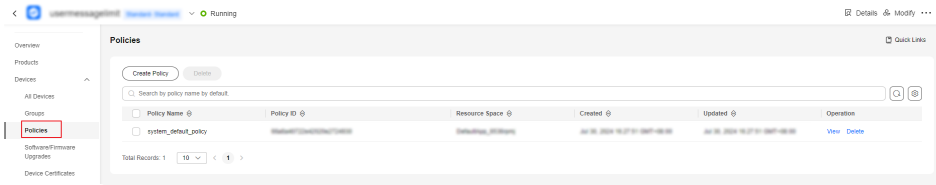
– Click **Enable Rule** in the middle of the page.

Figure 5-76 Enabling a rule - Forwarding data to a device



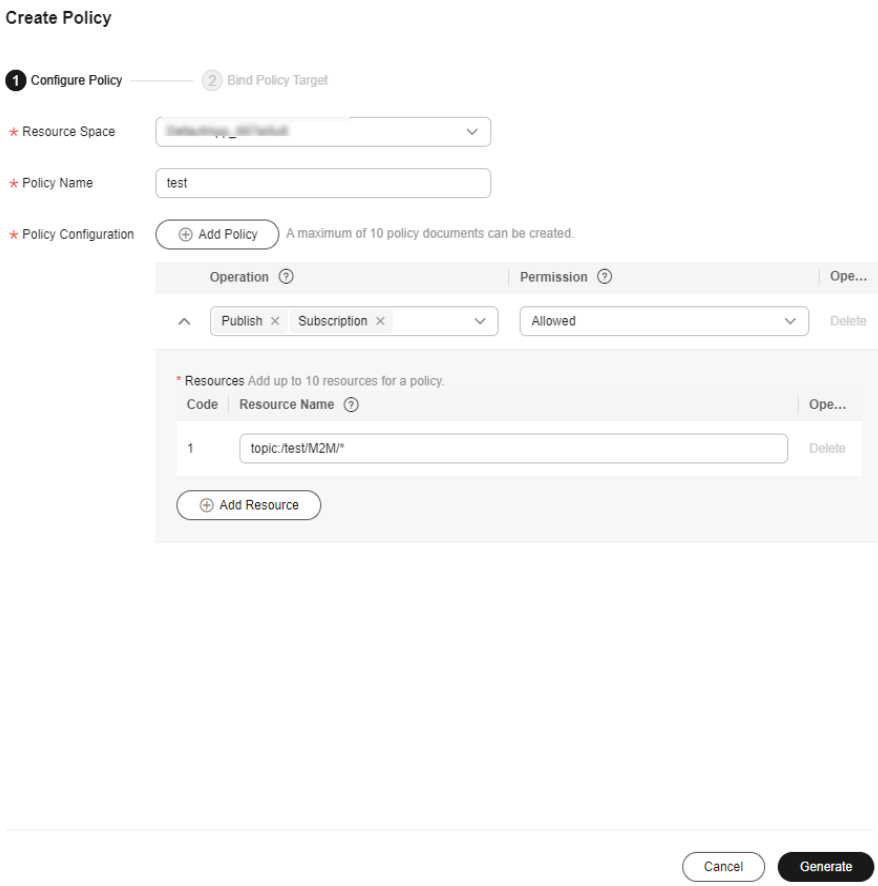
3. Set a policy.
   - Choose **Devices** > **Policies** in the navigation pane.

     Figure 5-77 Device policy - Access page

   - Click **Create Policy**, set policy parameters, and click **Generate**, as shown in the following figure.

Figure 5-78 Device policy - Creating a policy (M2M)



– Bind the policy to the target products and devices. Set **Target Type** to **Products** and select the products to which the policy is to be bound. You can later modify the policy on the policy details page to add the devices to be bound.

Figure 5-79 Device policy - Binding a product

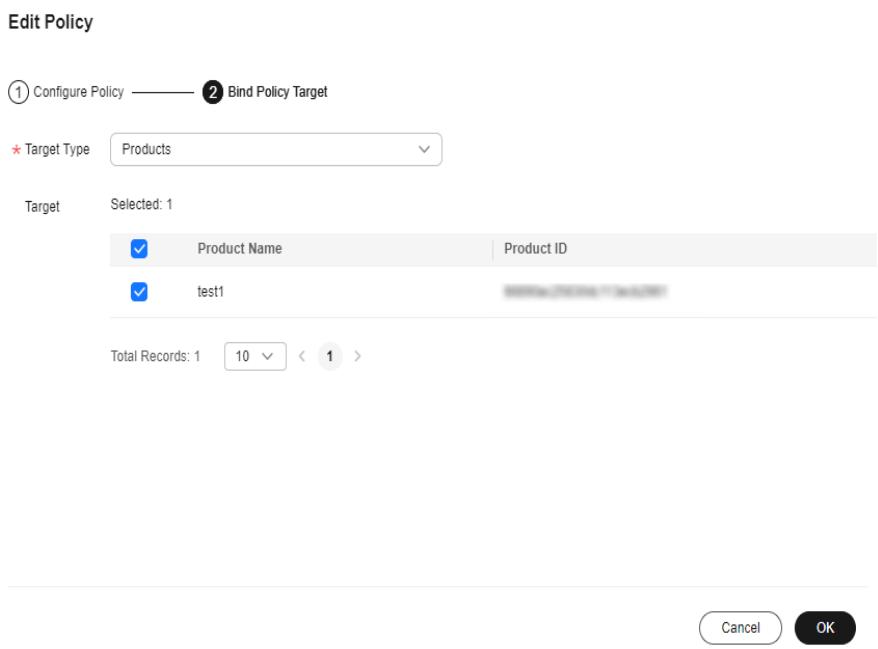
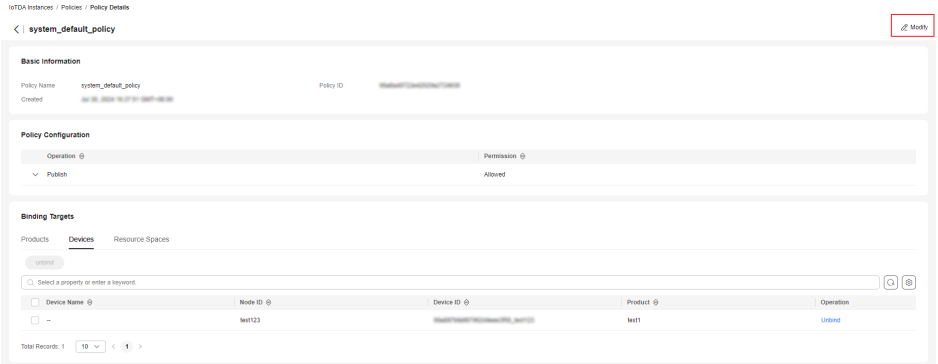
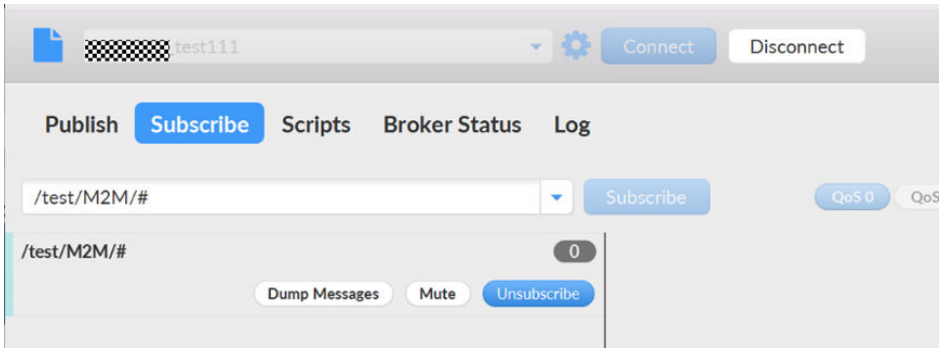


Figure 5-80 Device policy - Encoding a policy



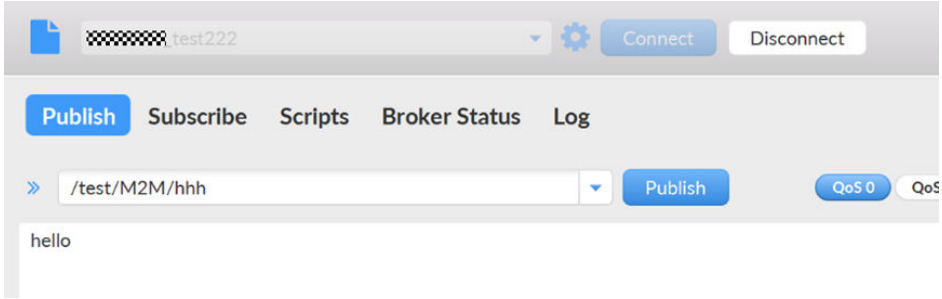
4. Verify the policy.
   a. Open two MQTT.fx tools to simulate device A (**test111**) under product A and device B (**test222**) under product B.
   b. On the **Subscribe** page of device B, enter the topic **/test/M2M/#** and click **Subscribe**.

Figure 5-81 Entering a topic on the Subscribe page of device B



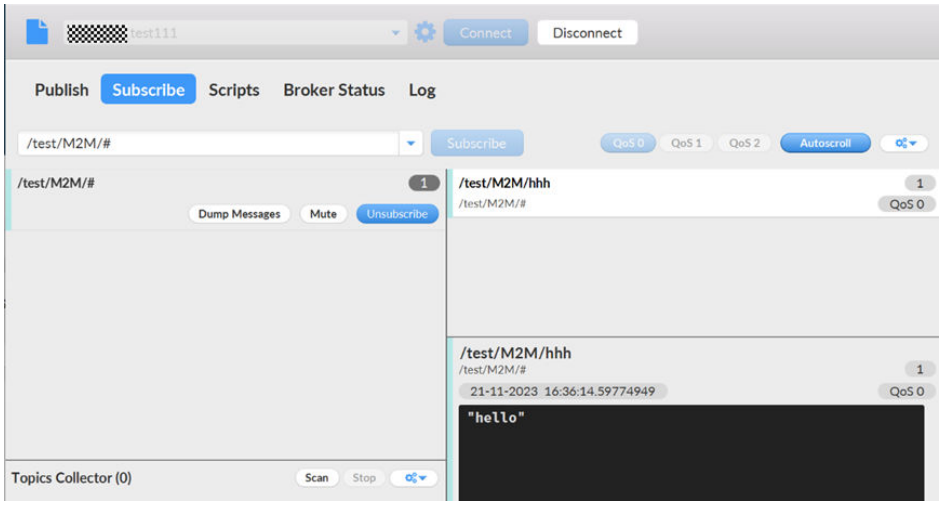
c. Let device A send a message to device B. On the **Publish** page of device A, enter the topic **/test/M2M/${*Any word*}**. Enter the message to be sent (for example, **hello**) in the text box, and click **Publish**.

Figure 5-82 Entering a message on the Publish page of device A



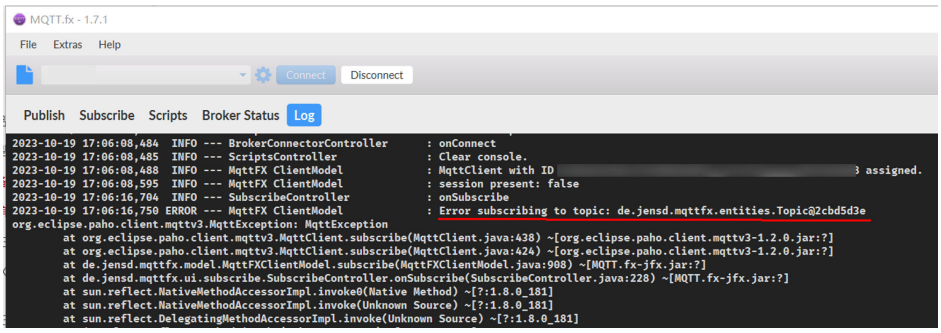
On the **Subscribe** page of device B, you can see the received message, as shown in the following figure.

Figure 5-83 Subscribe page of device B



d. For devices not belonging to product B, they cannot subscribe to or publish messages through the topic **/test/M2M/#**.

Figure 5-84 Failed subscription



5.6 Broadcast Communication

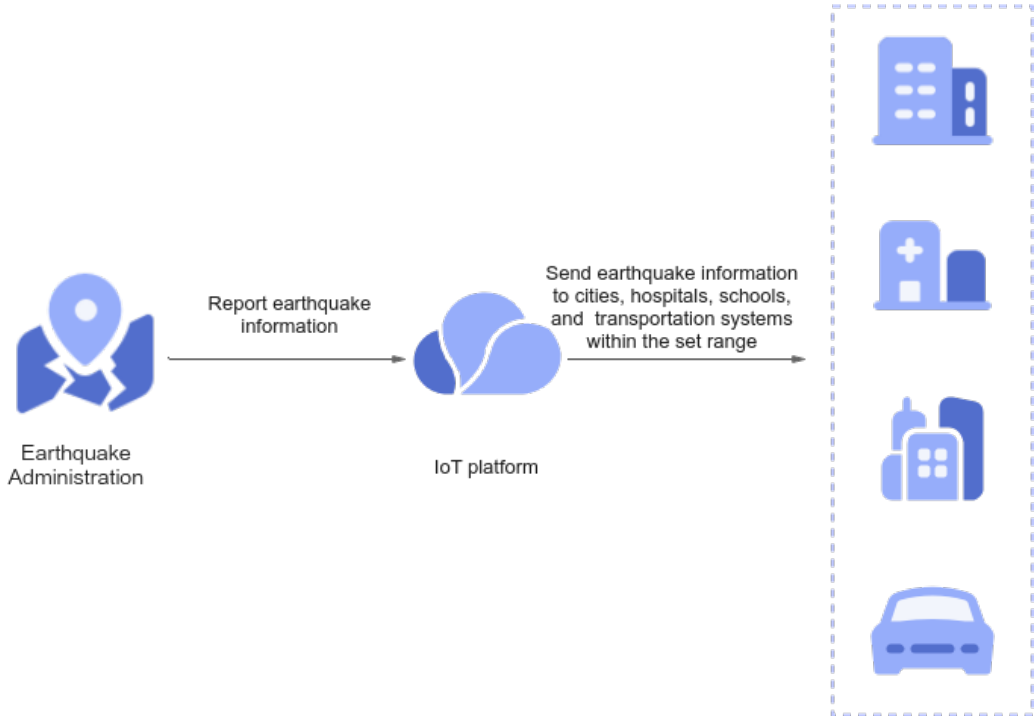
5.6.1 Broadcast Communication Overview

Introduction

Broadcast communication is often used for one-to-many message communication. If multiple devices subscribe to the same broadcast topic, applications can call the broadcast message delivery API to publish messages to these devices when they are online. You can use broadcast to send notifications to devices of specific types.

For example, the Earthquake Administration sends earthquake warning information to all citizens in a specified area.

Figure 5-85 Example broadcast communication scenario



Scenarios

- Broadcast messages are sent to devices in a specified group.
- Broadcast messages are sent to all online devices in a specified area for earthquake warning.

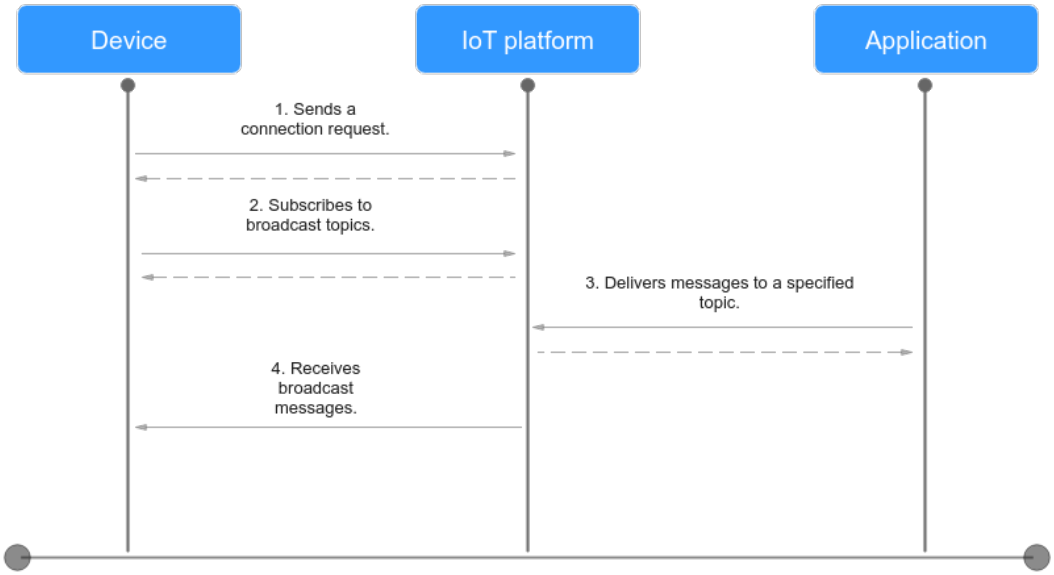
Constraints

- The prefix of a broadcast topic subscribed by devices must be **$oc/broadcast/**.
- The length of a broadcast topic subscribed by devices cannot exceed 128 bytes.
- A maximum of seven slashes (/) can be used in a broadcast topic subscribed by devices.
- A single device can subscribe to a maximum of 50 broadcast topics.
- A topic can be subscribed to by a maximum of 1000 devices.
- The broadcast communication API on the application side can be called only once per minute.
- Currently, this function is available for standard instances in the CN North-Beijing4 region.

5.6.2 Broadcast Communication Usage

Broadcast Communication Usage

Figure 5-86 Broadcast communication sequence diagram



Procedure

Step 1 Initiate the connection authentication for the device. For details, see **Device Connection Authentication**.

Step 2 After the device is authenticated, initiate broadcast topic subscription. The broadcast topic must be prefixed with **$oc/broadcast/**. An example is as follows:

```
$oc/broadcast/test
```

Step 3 The application **broadcasts a message** with the topic name and message content specified.

```
POST https://{Endpoint}/v5/iot/{project_id}/broadcast-messages
Content-Type: application/json
X-Auth-Token: ********

{
  "topic_full_name" : "$oc/broadcast/test",
  "message" : "eyJhIjoxfQ=="
}
```

CAUTION

The topic must be prefixed with **$oc/broadcast/**, and the message content must be encoded using Base64.

Step 4 The device receives the broadcast message. Example message:

```
Topic: $oc/broadcast/test
Data content
{"a":1}
```

----End

5.6.3 Broadcast Communication Example

Java SDK Usage

This section describes how to use the Java SDK for the development of broadcast communication.

Development Environment Requirements

JDK 1.8 or later has been installed.

Configuring the SDK for the Application

1. Configure the Maven dependency.

```
<dependency>
    <groupId>com.huaweicloud.sdk</groupId>
    <artifactId>huaweicloud-sdk-core</artifactId>
    <version>[3.0.40-rc, 3.2.0)</version>
</dependency>
<dependency>
    <groupId>com.huaweicloud.sdk</groupId>
    <artifactId>huaweicloud-sdk-iotda</artifactId>
    <version>[3.0.40-rc, 3.2.0)</version>
</dependency>
```

2. The following is a complete example of a broadcast message. The topic must be prefixed with **$oc/broadcast/**, and the message content must be encoded using Base64.

```
public class BroadcastMessageSolution {
    // REGION_ID: If CN East-Shanghai1 is used, enter cn-east-3. If CN North-Beijing4 is used, enter
cn-north-4. If CN South-Guangzhou is used, enter cn-south-4.
    private static final String REGION_ID = "<YOUR REGION ID>";
    // ENDPOINT: On the console, choose Overview and click Access Addresses to view the HTTPS
```

```
application access address.
    private static final String ENDPOINT = "<YOUR ENDPOINT>";
    // For the standard or enterprise edition, create a region object.
    public static final Region REGION_CN_NORTH_4 = new Region(REGION_ID, ENDPOINT);
    public static void main(String[] args) {
        String ak = "<YOUR AK>";
        String sk = "<YOUR SK>";
        String projectId = "<YOUR PROJECTID>";
        // Create a credential.
        ICredential auth = new
BasicCredentials().withDerivedPredicate(AbstractCredentials.DEFAULT_DERIVED_PREDICATE)
            .withAk(ak)
            .withSk(sk)
            .withProjectId(projectId);
        // Create and initialize an IoTDAClient instance.
        IoTDAClient client = IoTDAClient.newBuilder().withCredential(auth)
            // For the basic edition, select the region object in IoTDARegion.
            //.withRegion(IoTDARegion.CN_NORTH_4)
            // For the standard or enterprise edition, create a region object.
            .withRegion(REGION_CN_NORTH_4).build();
        // Instantiate a request object.
        BroadcastMessageRequest request = new BroadcastMessageRequest();
        DeviceBroadcastRequest body = new DeviceBroadcastRequest();
        body.withMessage(Base64.getEncoder().encodeToString("hello".getBytes()));
        body.withTopicFullName("$oc/broadcast/test");
        request.withBody(body);
        try {
            BroadcastMessageResponse response = client.broadcastMessage(request);
            System.out.println(response.toString());
        } catch (ConnectionException e) {
            e.printStackTrace();
        } catch (RequestTimeoutException e) {
            e.printStackTrace();
        } catch (ServiceResponseException e) {
            e.printStackTrace();
            System.out.println(e.getHttpStatusCode());
            System.out.println(e.getRequestId());
            System.out.println(e.getErrorCode());
            System.out.println(e.getErrorMsg());
        }
    }
}
```

Table 5-25 Parameters

Parameter	Description
ak	Access key ID (AK) of your Huawei Cloud account. You can create and view your AK/SK on the **My Credentials** > **Access Keys** page of the Huawei Cloud console. For details, see **Access Keys**.
sk	Secret access key (SK) of your Huawei Cloud account.
projectId	Project ID. For details on how to obtain a project ID, see **Obtaining a Project ID**.

Parameter	Description
IoTDARegion.CN_NORTH_4	Region where the IoT platform to be accessed is located. The available regions of the IoT platform have been defined in the SDK code **IoTDARegion.java**. On the console, you can view the region name of the current service and the mapping between regions and endpoints. For details, see **Platform Connection Information**.
REGION_ID	If CN East-Shanghai1 is used, enter **cn-east-3**. If CN North-Beijing4 is used, enter **cn-north-4**. If CN South-Guangzhou is used, enter **cn-south-4**.
ENDPOINT	On the console, choose **Overview** and click **Access Addresses** to view the HTTPS application access address.

Configuring the SDK on Devices

1. Configure the Maven dependency of the SDK on devices.

```
<dependency>
    <groupId>com.huaweicloud</groupId>
    <artifactId>iot-device-sdk-java</artifactId>
    <version>1.1.4</version>
</dependency>
```

2. Configure the SDK and device connection parameters on devices.

```
// Load the CA certificate of the IoT platform. For details about how to obtain the certificate, visit
https://support.huaweicloud.com/intl/en-us/devg-iothub/iot_02_1004.html.
URL resource = BroadcastMessageSample.class.getClassLoader().getResource("ca.jks");
File file = new File(resource.getPath());

// The format is ssl://Domain name:Port number.
// To obtain the domain name, log in to the Huawei Cloud IoTDA console. In the navigation pane,
choose Overview and click Access Details in the Instance Information area. Select the access
domain name corresponding to port 8883.
String serverUrl = "ssl://localhost:8883";
// Device ID created on the IoT platform
String deviceId = "deviceId";
// Secret corresponding to the device ID
String deviceSecret = "secret";
// Create a device.
IoTDevice device = new IoTDevice(serverUrl, deviceId, deviceSecret, file);
if (device.init() != 0) {
    return;
}
```

3. Subscribe to a broadcast topic for the device. The broadcast topic must be prefixed with **$oc/broadcast/**.

```
device.getClient().subscribeTopic("$oc/broadcast/test", null, rawMessage -> {
    log.info(" on receive message topic : {} , payload : {}", rawMessage.getTopic(),
        new String(rawMessage.getPayload()));
    rawMessage.getPayload();
}, 0);
```

Testing and Verification

Run the SDK code on the device to bring the device online and subscribe to the broadcast topic for the device. Run the SDK code on the application and call the **broadcastMessage** API to send a broadcast message to the device. Example message:

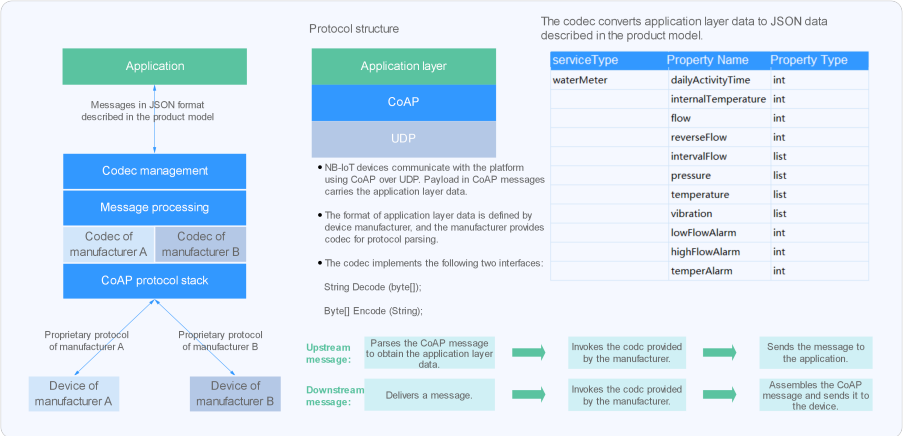
Figure 5-87 Broadcast message example

5.7 Codecs

Definition

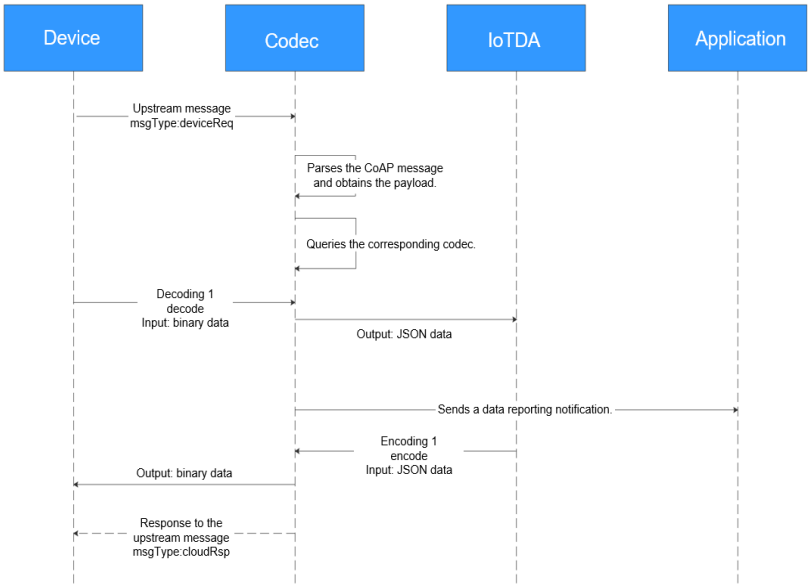
IoTDA uses codecs to convert data between the binary and JSON formats as well as between JSON formats. For MQTT devices, use JavaScript and FunctionGraph to develop codecs. For LwM2M devices, use online (graphical) and offline codec development.

For example, in the NB-IoT scenario where devices use CoAP over UDP to communicate with the platform, the payload of CoAP messages carries data at the application layer, at which the data type is defined by the devices. As NB-IoT devices require low power consumption, data at the application layer is generally in binary format instead of JSON. However, the platform sends data in JSON format to applications. Therefore, codec development is required for the platform to convert data between binary and JSON formats.



Data Reporting

Figure 5-88 Data reporting flowchart

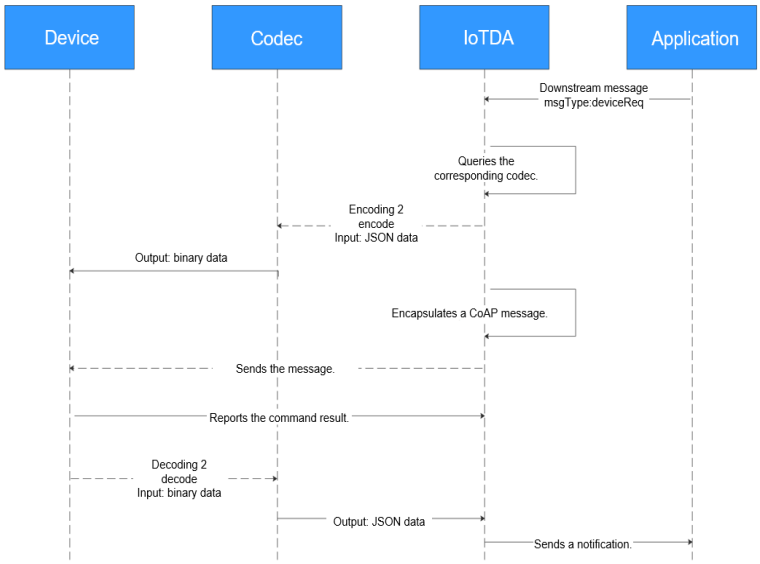


In the data reporting process, the codec is used in the following scenarios:

- Decoding binary data reported by a device into JSON data and sending the decoded data to an application
- Encoding JSON data returned by an application into binary data and sending the encoded data to a device

Command Delivery

Figure 5-89 Command delivery flowchart



In the command delivery process, the codec is used in the following scenarios:

- Encoding JSON data delivered by an application into binary data and sending the encoded data to a device
- Decoding binary data returned by a device into JSON data and reporting the decoded data to an application

Development Methods

The platform provides multiple methods for developing codecs. You can select a method as required. Offline codec development is complex and time-consuming. Graphical codec development and script-based codec development are recommended.

- Graphical development: The codec of a product can be quickly developed in a visualized manner on the IoTDA console. For details, see **Online Development**.
- Script-based development: JavaScript scripts are used to implement encoding and decoding. For details, see **JavaScript Script-based Development**.
- FunctionGraph development: FunctionGraph is used to implement encoding and decoding. For details, see **FunctionGraph Documentation**.

6 Device Management

6.1 Product Creation

The first step of using the IoT platform is to create a product on the IoTDA console. A product is a collection of devices with the same capabilities or features.

Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 Choose **Products** in the navigation pane and click **Create Product** on the left. Set the parameters as prompted and click **OK**.

Set Basic Info	
Resource Space	Select a resource space from the drop-down list box. If a **resource space** does not exist, create it first.
Product Name	Define a product name. The product name must be unique in the same resource space. The value can contain up to 64 characters. Only letters, digits, and special characters (_?'#().,&%@!-) are allowed.

Protocol	• MQTT: MQTT is used by devices to access the platform. The data format can be binary or JSON. If the binary format is used, the codec must be deployed. • LwM2M over CoAP: LwM2M/CoAP is used only by NB-IoT devices with limited resources (including storage and power consumption). The data format is binary. The codec must be deployed to interact with the platform. • HTTPS is a secure communication protocol based on HTTP and encrypted using SSL. IoTDA supports communication through HTTPS. • Modbus: Modbus is used by devices to access the platform. Devices that use the Modbus protocol to connect to IoT edge nodes are called indirectly connected devices. For details about the differences between directly connected devices and indirectly connected devices, see **Gateways and Child Devices**. • HTTP (TLS encryption), ONVIF, OPC UA, OPC DA, other, TCP, and UDP: IoT Edge is used for connection.
Data Type	• **JSON**: JSON is used for the communication protocol between the platform and devices. • **Binary**: You need to develop a **codec** on the IoTDA console to convert binary code data reported by devices into JSON data. The devices can communicate with the platform only after the JSON data delivered by the platform is parsed into binary code.
Industry	Set this parameter based on service requirements.
Device Type	Set this parameter based on service requirements.
Advanced Settings	
Product ID	Set a unique identifier for the product. If this parameter is specified, the platform uses the specified product ID. If this parameter is not specified, the platform allocates a product ID.
Description	Provide a description for the product. Set this parameter based on service requirements.

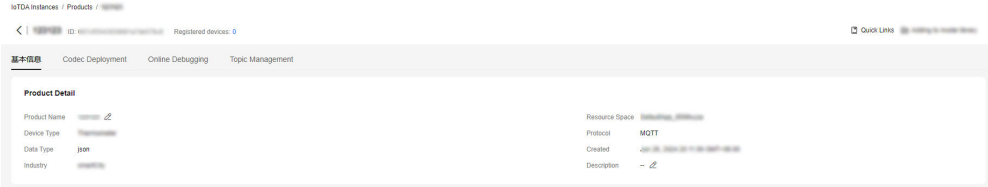
You can click **More** > **Delete** to delete a product that is no longer used. After the product is deleted, its resources such as the product models and codecs will be cleared. Exercise caution when deleting a product.

----End

Follow-Up Procedure

1. In the product list, click the name of a product to access its details. On the product details page displayed, you can view basic product information, such as the product ID, product name, device type, data format, resource space, and protocol type.

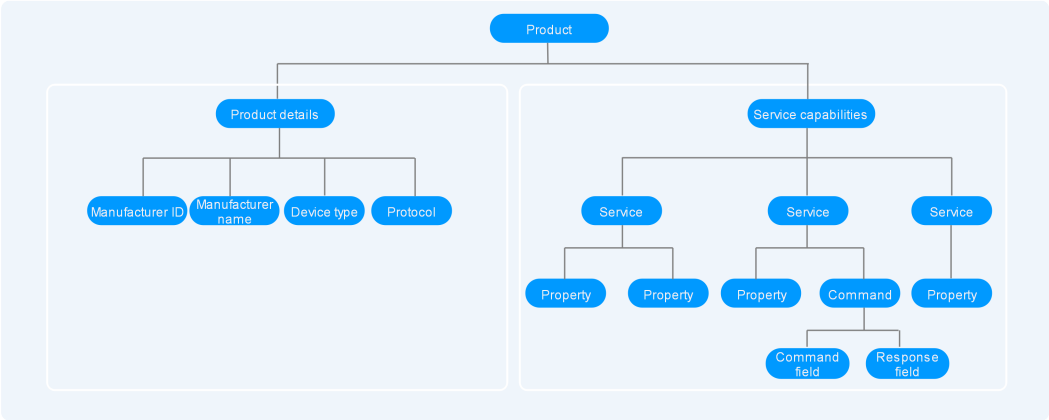
Figure 6-1 Product details



2. On the product details page, you can **develop a product model**, **develop a codec**, **perform online debugging**, and **customize topics**.

Product Models

A product model describes the capabilities and features of a device. You can build an abstract model of a device by defining a product model on the platform so that the platform can know what services, properties, and commands are supported by the device, such as its color or any on/off switches. After defining a product model, you can use it during **device creation**.



A product model consists of product details and service capabilities.

- **Product details**

  Product details describe basic information about a device, including the device and protocol type.

  Example device type: **WaterMeter**. Example protocol type: **CoAP**.

- **Service capabilities**

  The capabilities of a device are divided into several services. Properties, commands, and command parameters are defined for each service.

  For example, a water meter has multiple capabilities. It reports the water flow, alarms, battery life, and connection data, and it receives commands too. The table below describes its capabilities from five services, each of which has its own properties or commands.

Service Type	Description
WaterMeterBasic	Defines parameters reported by the water meter, such as the water flow, temperature, and pressure. If these parameters need to be controlled or modified using commands, these parameters must be defined in the commands.
WaterMeterAlarm	Defines data reported by the water meter in various alarm scenarios. Commands need to be defined if necessary.
Battery	Defines the voltage and current intensity of the water meter.
DeliverySchedule	Defines transmission rules for the water meter. Commands need to be defined if necessary.
Connectivity	Defines connectivity parameters of the water meter.

NOTE

You can define the number of services as required. For example, the **WaterMeterAlarm** service can be further divided into **WaterPressureAlarm** and **WaterFlowAlarm** services or be integrated into the **WaterMeterBasic** service.

Model Development

The platform provides multiple methods for developing product models. You can select a method as required.

- **Custom model (online development)**: Build a product model from scratch. For details, see **Developing a Product Model Online**.
- **Upload local profile (offline development)**: Upload a local product model to the platform. For details, see **Developing a Product Model Offline**.
- **Import from Excel**: Develop a product model by importing an Excel file. For details, see **Import from Excel**.
- **Import from Library**: You can use a preset product model to quickly develop a product. The platform provides standard and manufacturer-specific product models. Standard product models comply with industry standards and are suitable for devices of most manufacturers in the industry. Manufacturer-specific product models are suitable for devices provided by a small number of manufacturers. You can select a product model as required.

6.2 Registering Devices

6.2.1 Registering an Individual Device

A device is a physical entity that belongs to a product. Each device has a unique ID. It can be a device directly connected to the platform, or a gateway that

connects child devices to the platform. You can register a physical device with the platform, and use the device ID and secret allocated by the platform to connect your SDK-integrated device to the platform.

The platform allows an application to call the API **Creating a Device** to register an individual device. Alternatively, you can register an individual device on the IoTDA console. This topic describes the procedure on the IoTDA console.

Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **All Devices**. On the displayed page, click **Register Device**, set parameters based on the table below, and click **OK**.

Figure 6-2 Device - Registering a secret device

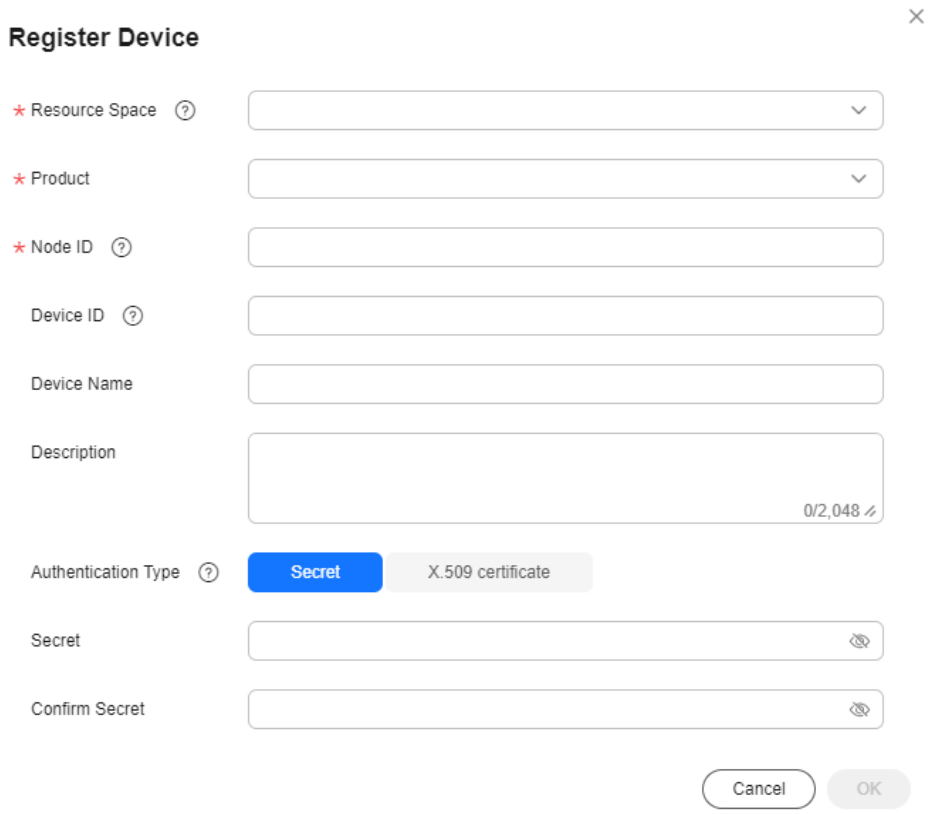


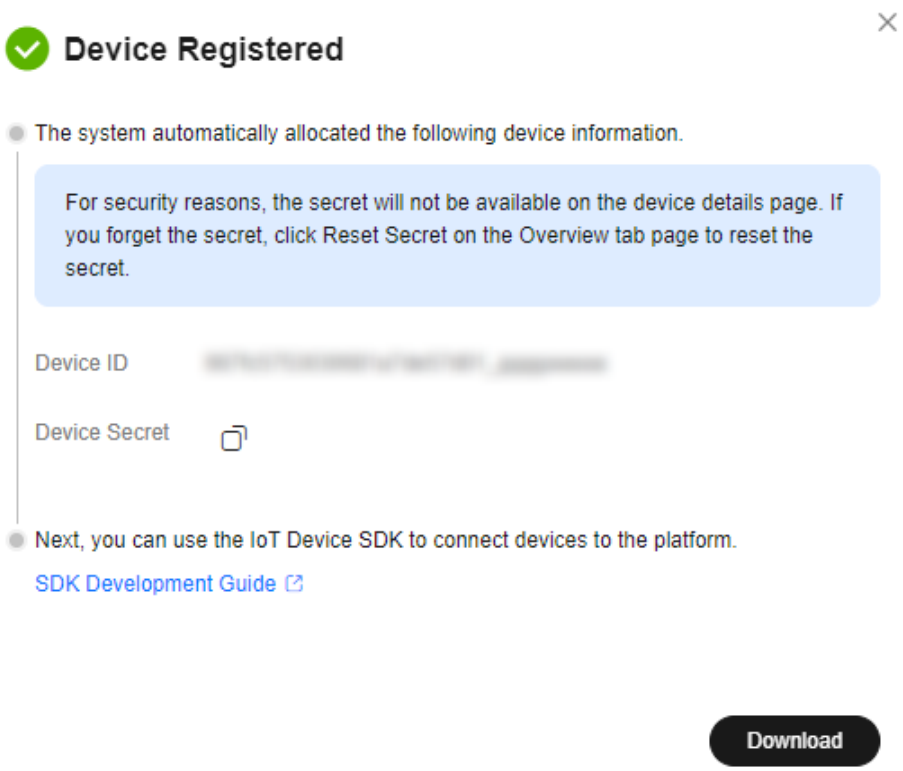
Table 6-1 Registering a device with secret

Parameter	Description
Resource Space	Select the resource space to which a device belongs.

Parameter	Description
Product	Select the product to which the device belongs. You can select a product only after it is defined. If no product is available, create a product by following the instructions provided in **Product Creation**.
Node ID	Set this parameter to the IMEI, MAC address, or serial number of the device. If the device is not a physical one, set this parameter to a custom string that contains letters, digits, hyphens (-), and underscores (_).
Device ID	Enter a unique device ID. If this parameter is carried, the platform will use the parameter value as the device ID. Otherwise, the platform will allocate a device ID, which is in the format of *product_id_node_id*.
Device Name	Customize the name of the device.
Description	Customize device description.
Authentication Type	• **Secret**: The device uses the secret for identity verification. • **X.509 certificate**: The device uses an X.509 certificate for identity verification.
Secret	Customize the secret used for device access. If the secret is left blank, the platform automatically generates one.
Fingerprint	This parameter is displayed when **Authentication Type** is set to **X.509 certificate**. Import the fingerprint corresponding to the **preset device certificate on the device side**. You can run **openssl x509 -fingerprint -sha256 -in deviceCert.pem** in the OpenSSL view to query the fingerprint. [root@k8s-iot-wl2-2-jump cert1223]# openssl x509 -fingerprint -sha256 -in deviceCert.pem SHA256 Fingerprint=F7:91:90:45:BB:88:37:E6:A7:E7:70:4A:90:75:F3:07:DA:27:5B:7C:49:3E:FF:59:7A:6F:4F:08:4D:F8:54:E0 Delete the colons (:) from the obtained fingerprint when filling it.

Save the device ID and secret. They are used for authentication when the device attempts to access the platform.

Figure 6-3 Device registered



NOTE

If the secret is lost, you can **update the secret**. The secret generated during device registration cannot be retrieved.

You can delete a device that is no longer used from the **device list**. Deleted devices cannot be retrieved. Exercise caution when performing this operation.

----End

APIs

- **Querying the Device List**
- **Creating a Device**
- **Querying a Device**
- **Modifying a Device**
- **Deleting a Device**
- **Resetting a Device Secret**

6.2.2 Registering a Batch of Devices

IoTDA allows an application to call the API **Creating a Batch Task** to register a batch of devices. Alternatively, you can perform batch registration on the IoTDA console. This topic describes how to use the IoTDA console to register a batch of devices.

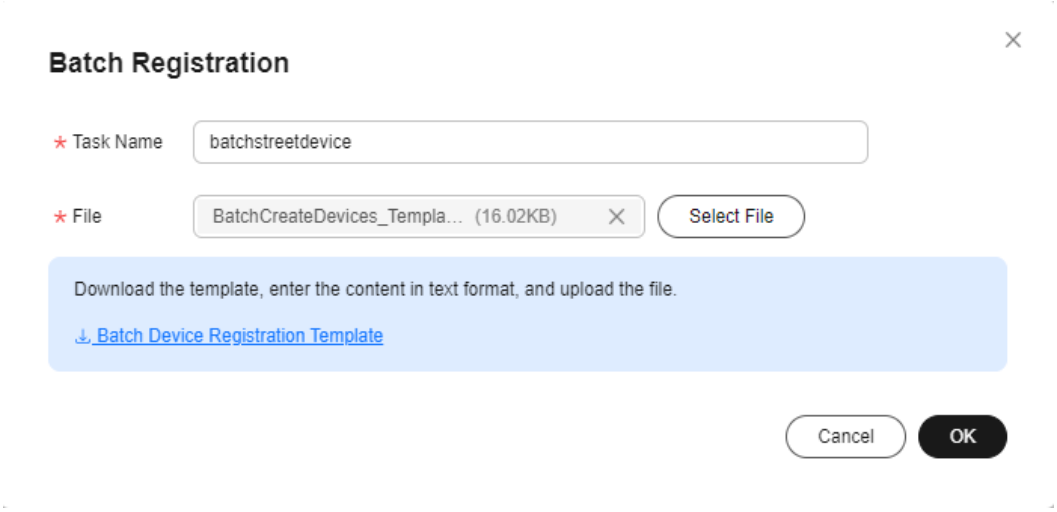
Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**.

Step 2 In the navigation pane, choose **Devices** > **All Devices**, click the **Batch Registration** tab, and then click **Batch Register**.

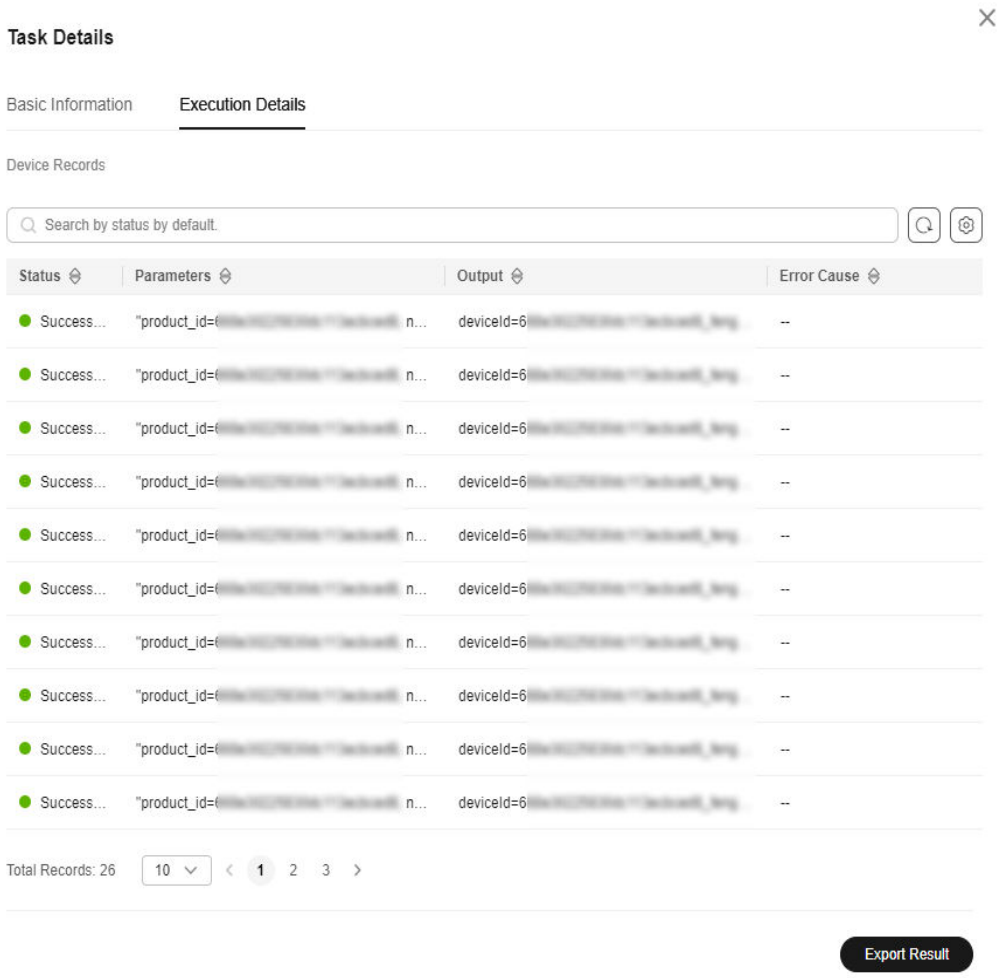
Step 3 In the displayed **Batch Registration** dialog box, enter the task name, download and fill in the **Batch Device Registration Template**, upload the file, and click **OK**.

Figure 6-4 Device - Registering devices in batches



Step 4 If the devices use the native MQTT protocol, click the batch task registration record to open the task execution details, and save the device IDs and secrets generated, which will be used for device access.

Figure 6-5 Batch device registering - Execution details



----End

APIs

- Creating a Device
- Querying the Batch Task List
- Creating a Batch Task
- Querying a Batch Task

6.2.3 Registering a Device Authenticated by an X.509 Certificate

An X.509 certificate is a digital certificate used for communication entity authentication. IoTDA allows devices to use their X.509 certificates for authentication. The use of X.509 certificate authentication protects devices from being spoofed.

Before registering a device authenticated by an X.509 certificate, upload the device CA certificate to the platform and bind the device certificate to the device during device registration. This topic describes how to upload a device CA certificate to

the platform and register a device that uses the X.509 certificate for authentication.

Constraints

- Only MQTT devices can use X.509 certificates for identity authentication.
- You can upload a maximum of 100 device CA certificates.

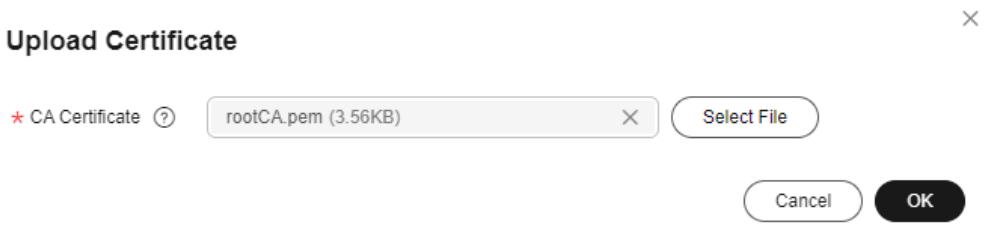
Uploading a Device CA Certificate

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **Device Certificates**. On the **Device CA Certificates** tab page, click **Upload Certificate**.

Step 3 In the displayed dialog box, click **Select File** to add a file, and then click **OK**.

Figure 6-6 Device CA certificate - Uploading a certificate



NOTE

Device CA certificates are provided by device vendors. You can **prepare a commissioning certificate** during commissioning. For security reasons, you are advised to replace the commissioning certificate with a commercial certificate during commercial use. Purchased CA certificates (in formats such as PEM and JKS) can be directly uploaded to the platform.

----End

Making a Device CA Commissioning Certificate

This section uses the Windows operating system as an example to describe how to use OpenSSL to make a commissioning certificate. The generated certificate is in PEM format.

1. Download and install **OpenSSL**.
2. Open the CLI as user **admin**.
3. Run **cd c:\openssl\bin** (replace **c:\openssl\bin** with the actual OpenSSL installation directory) to access the OpenSSL view.
4. Generate a public/private key pair.
```
openssl genrsa -out rootCA.key 2048
```
5. Use the private key in the key pair to generate a CA certificate.
```
openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024 -out rootCA.pem
```
The system prompts you to enter the following information. All the parameters can be customized.

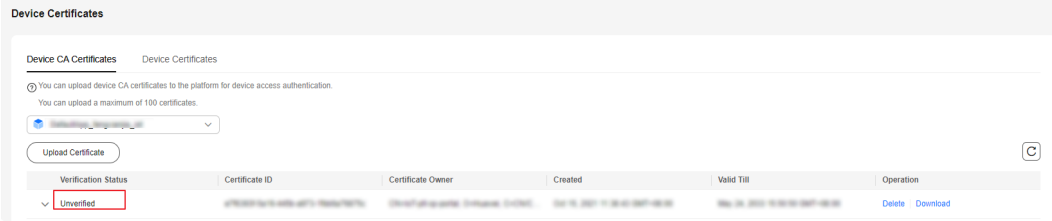
- Country Name (2 letter code) [AU]: country, for example, CN
- State or Province Name (full name) []: state or province, for example, GD
- Locality Name (for example, city) []: city, for example, SZ
- Organization Name (for example, company) []: organization, for example, Huawei
- Organizational Unit Name (for example, section) []: organization unit, for example, IoT
- Common Name (e.g. server FQDN or YOUR name) []: common name, for example, zhangsan
- Email Address []: email address, for example, 1234567@163.com

Obtain the generated CA certificate **rootCA.pem** from the **bin** folder in the OpenSSL installation directory.

Uploading a Verification Certificate

If the uploaded certificate is a commissioning certificate, the certificate status is **Unverified**. In this case, upload a verification certificate to verify that you have the CA certificate.

Figure 6-7 Device CA certificate - Unverified certificate



The verification certificate is created based on the private key of the device CA certificate. Perform the following operations to create a verification certificate:

Step 1 Generate a key pair for the verification certificate.

```
openssl genrsa -out verificationCert.key 2048
```

Step 2 Create a certificate signing request (CSR) for the verification certificate.

```
openssl req -new -key verificationCert.key -out verificationCert.csr
```

The system prompts you to enter the following information. Set **Common Name** to the verification code and set other parameters as required.

- Country Name (2 letter code) [AU]: country, for example, CN
- State or Province Name (full name) []: state or province, for example, GD
- Locality Name (for example, city) []: city, for example, SZ
- Organization Name (for example, company) []: organization, for example, Huawei
- Organizational Unit Name (for example, section) []: organization unit, for example, IoT
- Common Name (e.g. server FQDN or YOUR name) []: verification code for verifying the certificate. For details on how to obtain the verification code, see **Step 5**.

- Email Address []: email address, for example, 1234567@163.com
- Password[]: password, for example, 1234321
- Optional Company Name[]: company name, for example, Huawei

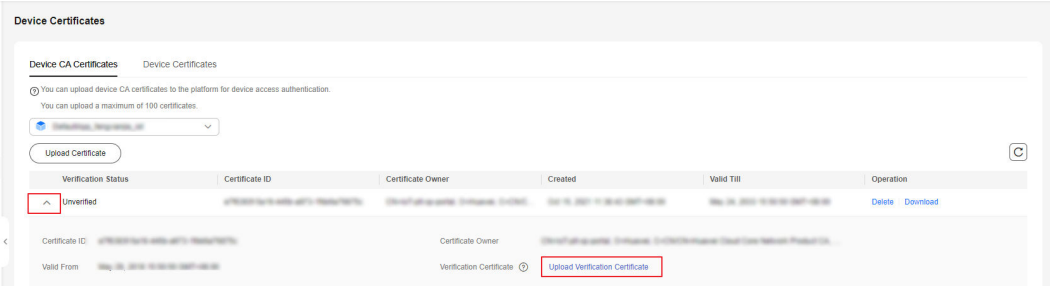
Step 3 Use the CSR to create a verification certificate.

```
openssl x509 -req -in verificationCert.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out verificationCert.pem -days 500 -sha256
```

Obtain the generated verification certificate **verificationCert.pem** from the **bin** folder of the OpenSSL installation directory.

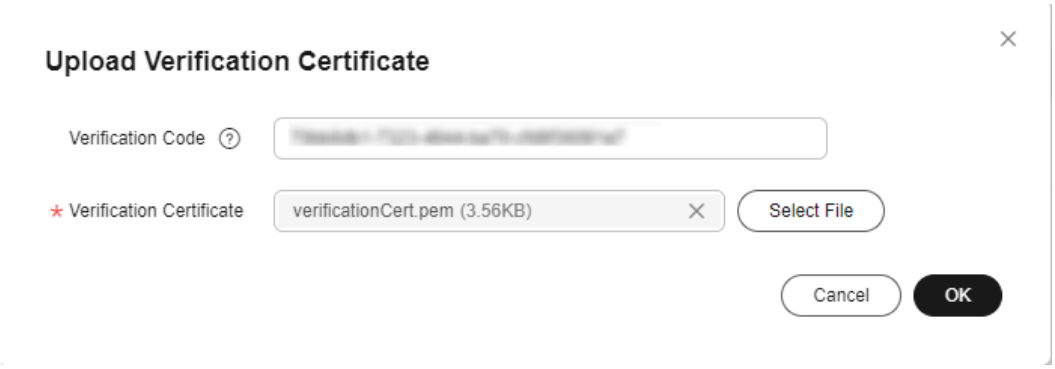
Step 4 Select the corresponding certificate, click ˅, and click **Upload Verification Certificate**.

Figure 6-8 Device CA certificate - Verifying a certificate



Step 5 The verification code is displayed in the dialog box. Click **Select File**, upload the verification certificate, and click **OK**. After the certificate is uploaded, the certificate status changes to **Verified**, indicating that you have the CA certificate.

Figure 6-9 Device CA certificate - Uploading a verified certificate



----End

Presetting an X.509 Certificate

Before registering an X.509 device, preset the X.509 certificate issued by the CA on the device.

NOTE

The X.509 certificate is issued by the CA. If no commercial certificate issued by the CA is available, you can **create an X.509 commissioning certificate**. Purchased certificates or certificates (in formats such as PEM and JKS) issued by authoritative organizations can be directly uploaded to the platform.

Creating an X.509 Commissioning Certificate

1. Run **cmd** as user **admin** to open the CLI and run **cd c:\openssl\bin** (replace **c:\openssl\bin** with the actual OpenSSL installation directory) to access the OpenSSL view.
2. Generate a public/private key pair.
```
openssl genrsa -out deviceCert.key 2048
```
3. Create a CSR for the device certificate.
```
openssl req -new -key deviceCert.key -out deviceCert.csr
```
The system prompts you to enter the following information. All the parameters can be customized.
   - Country Name (2 letter code) [AU]: country, for example, CN
   - State or Province Name (full name) []: state or province, for example, GD
   - Locality Name (for example, city) []: city, for example, SZ
   - Organization Name (for example, company) []: organization, for example, Huawei
   - Organizational Unit Name (for example, section) []: organization unit, for example, IoT
   - Common Name (e.g. server FQDN or YOUR name) []: common name, for example, zhangsan
   - Email Address []: email address, for example, 1234567@163.com
   - Password[]: password, for example, 1234321
   - Optional Company Name[]: company name, for example, Huawei
4. Create a device certificate using CSR.
```
openssl x509 -req -in deviceCert.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out deviceCert.pem -days 500 -sha256
```
Obtain the generated device certificate **deviceCert.pem** from the **bin** folder in the OpenSSL installation directory.

Registering a Device Authenticated by an X.509 Certificate

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **All Devices**, click **Register Device**, set parameters based on the table below, and click **OK**.

Figure 6-10 Device - Registering an X.509 device

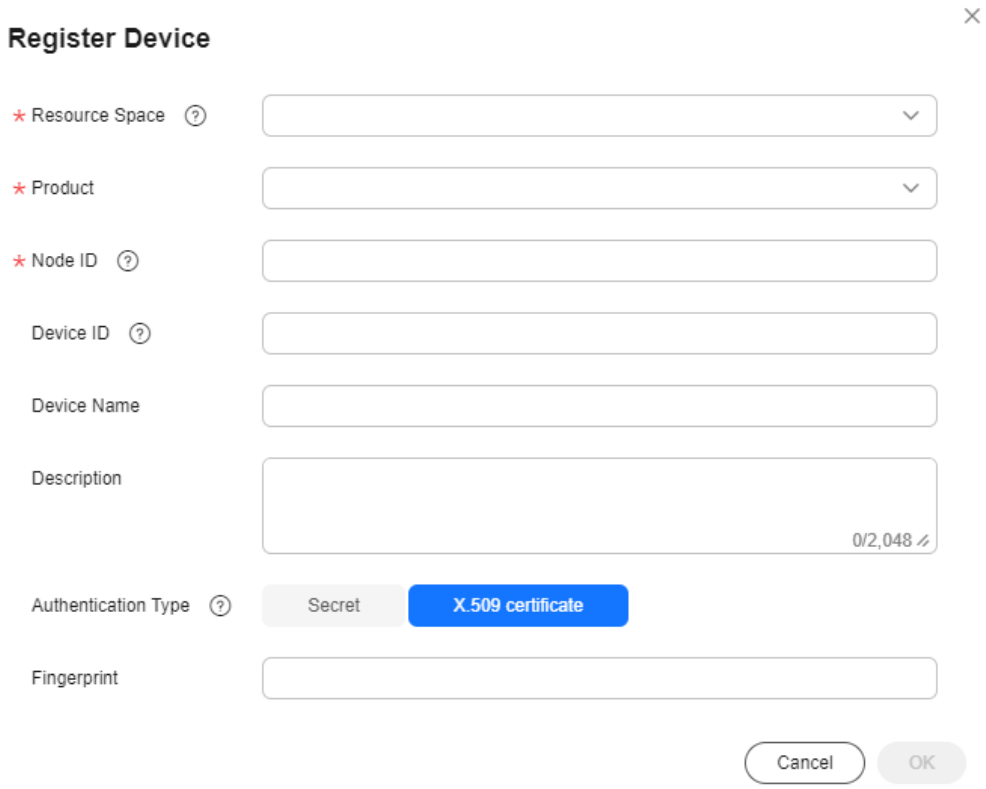


Table 6-2 Registering a device using X.509 certificate

Parameter	Description
Resource Space	Select the resource space to which a device belongs.
Product	Select the product to which the device belongs. You can select a product only after it is defined. If no product is available, create a product by following the instructions provided in **Product Creation**.
Node ID	Set this parameter to the IMEI, MAC address, or serial number of the device. If the device is not a physical one, set this parameter to a custom string that contains letters, digits, hyphens (-), and underscores (_).
Device ID	Enter a unique device ID. If this parameter is carried, the platform will use the parameter value as the device ID. Otherwise, the platform will allocate a device ID, which is in the format of *product_id_node_id*.
Device Name	Customize the device name.
Description	Customize device description.

Parameter	Description
Authentication Type	**X.509 certificate**: The device uses an X.509 certificate for identity verification.

Fingerprint	This parameter is displayed when **Authentication Type** is set to **X.509 certificate**. Import the fingerprint corresponding to the **preset device certificate on the device side**. You can run **openssl x509 -fingerprint -sha256 -in deviceCert.pem** in the OpenSSL view to query the fingerprint. **Note: Delete the colon (:) from the obtained fingerprint when filling it.**
	`[root@k8s-iot-wl2-2-jump_cert1223]# openssl x509 -fingerprint -sha256 -in deviceCert.pem`
	`SHA256 Fingerprint=F7:91:90:45:BB:88:37:E6:A7:E7:70:4A:90:75:F3:87:DA:27:5B:7C:49:3E:FF:59:7A:6F:4F:08:4D:F8:54:E8`

----End

APIs

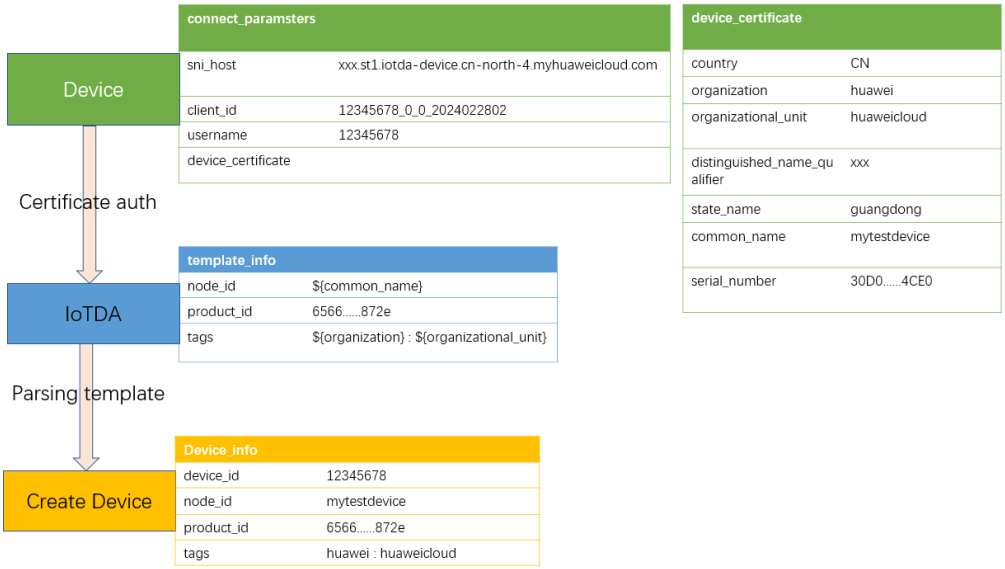
- **Obtaining the Device CA Certificate List**
- **Uploading a Device CA Certificate**
- **Deleting a Device CA Certificate**
- **Verifying a Device CA Certificate**

6.2.4 Device Self-Registration

Overview

For security, devices can connect to IoTDA only after their basic information (such as the device ID and authentication information) is registered on the platform. You can register a device on the platform manually or use self-registration templates, with which the device information is automatically registered when the device connects to the platform for the first time. This section describes how to use certificates and server name indication (SNI) to implement device self-registration.

Figure 6-11 Service flow



Scenarios

- Common scenarios: With self-registration, devices are registered automatically with device certificates, free of device provisioning.
- IoV: With self-registration, head units can go online immediately upon starting, simplifying application development.
- Large enterprise customers: With self-registration, the customers who have purchased multiple IoTDA instances do not need to register and provision devices under different instances separately in advance.

Constraints

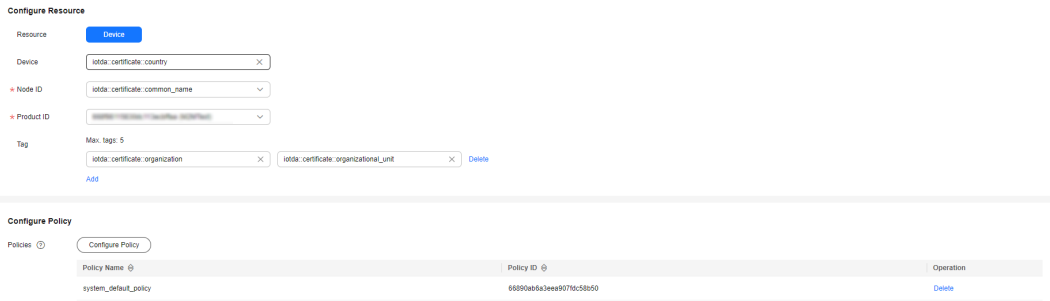
- A maximum of 10 self-registration templates can be created for an account.
- To use the device self-registration function, the device must use TLS and enable the **SNI** extension. The SNI must carry the domain name allocated by the platform. You can obtain the domain name by choosing **Overview** and clicking **Access Details**.
- Currently, this function supports only bidirectional MQTTS certificate authentication.

Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 Create a self-registration template. In the navigation pane, choose **Devices** > **Self-Registration Template**, and click **Create Template**. You can bind policies to devices in the template in advance. For details about how to use device policies, see **Device Topic Policies**. Set the node ID and product ID (mandatory). Set the device ID to the value of **Username** in the MQTT connection parameters. The product must be created on the platform in advance.

Figure 6-12 Self-registration template - Creating a template



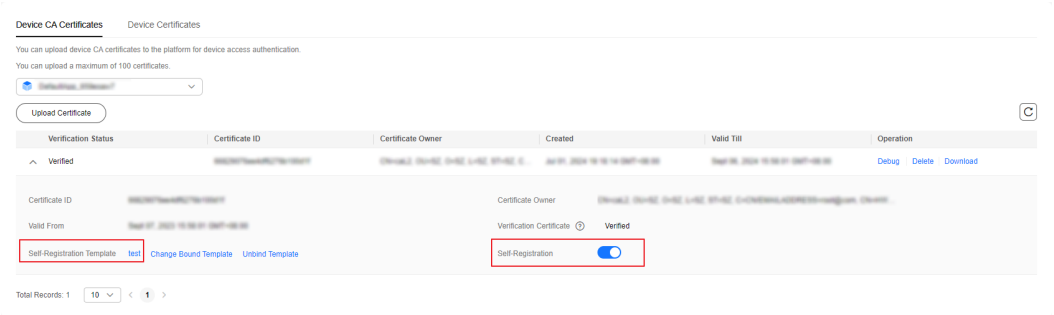
NOTE

The platform predefines the parameters that can be declared and referenced in the template, as shown below. The certificate must contain the parameters referenced in the template.

- **iotda::certificate::country**: country
- **iotda::certificate::organization**: organization
- **iotda::certificate::organizational_unit**: department
- **iotda::certificate::distinguished_name_qualifier**: distinguished name
- **iotda::certificate::state_name**: province/state
- **iotda::certificate::common_name**: common name
- **iotda::certificate::serial_number**: serial number

Step 3 Create a device certificate by referring to **Registering a Device Authenticated by an X.509 Certificate**. Upload the CA certificate to the platform, verify the certificate, bind the self-registration template created in **Step 2**, and enable the self-registration function.

Figure 6-13 Device CA certificate - Binding a template

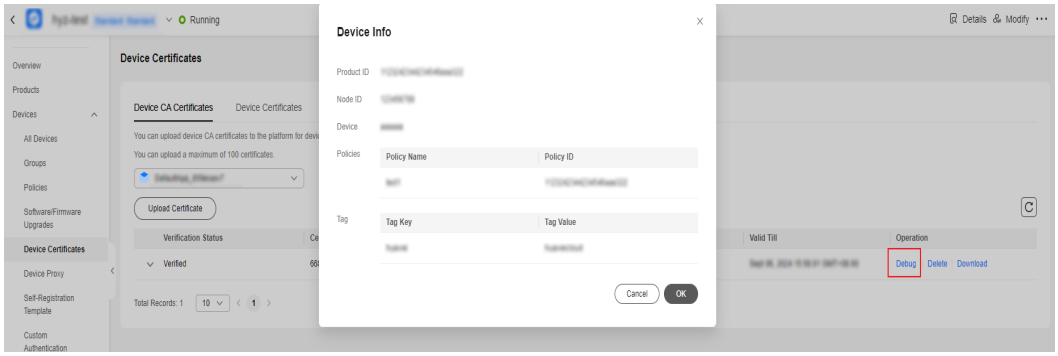


NOTICE

The device to register and its CA certificate must be in the same resource space. Ensure that the CA certificate and the product corresponding to the product ID in the template are in the same resource space.

Step 4 In the navigation pane, choose **Devices** > **Device Certificates**. On the **Device CA Certificates** tab page, click **Debug** to upload the device certificate created in **Step 3**, and check whether the pre-parsed device information meets the expectation.

Figure 6-14 Device CA certificate - Debugging a certificate



----End

Verification

1. Use the MQTT.fx tool to simulate the connection of a device to the platform for the first time and the automatic registration. Set the client ID by referring to **Connection Parameters**. Set **User Name** to the ID of the device registered in the platform. **Password** is not required. Obtain the CA certificate of the platform by referring to **Certificates**. After the connection is successful, check the registered device information on the platform.

   Figure 6-15 Connection parameters

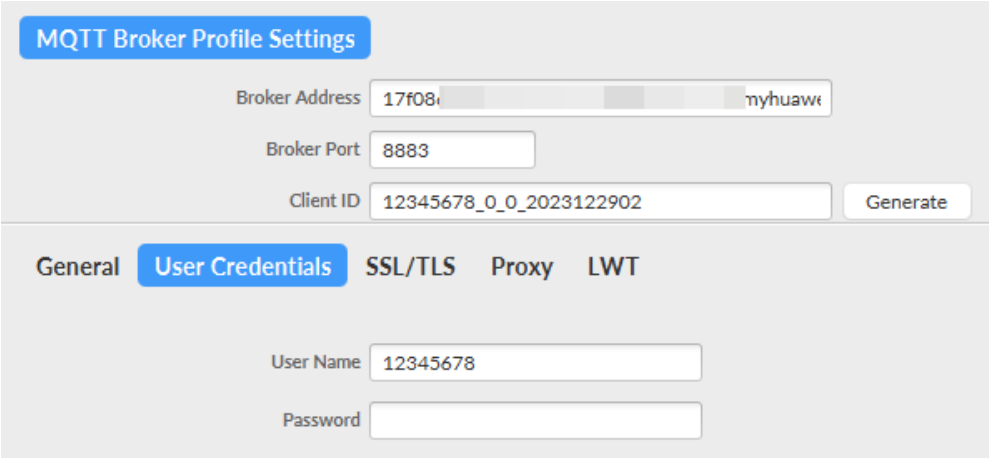
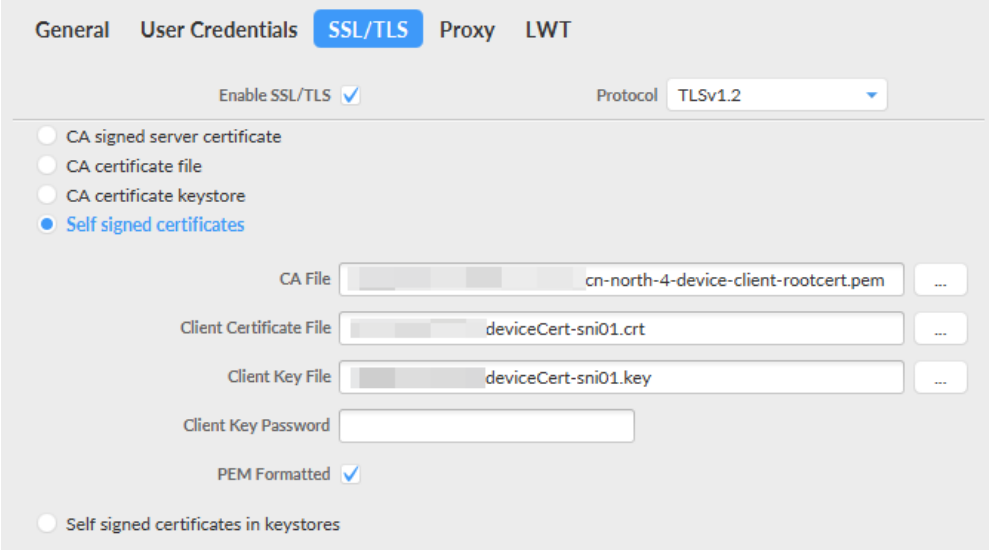
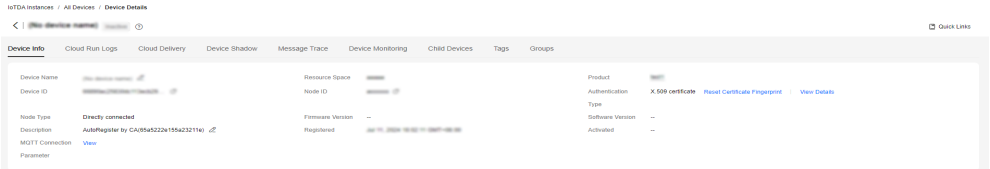


Figure 6-16 Certificate information



2. After the connection is successful, you can find the self-registered device in the device list on the console.

Figure 6-17 Device - Self-registered device details



6.3 Device Management

After a device is registered, you can manage the device, view device information, and freeze the device on the IoTDA console.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **All Devices**. By default, all devices in the current instance are displayed in the device list.

Figure 6-18 Device - Device list

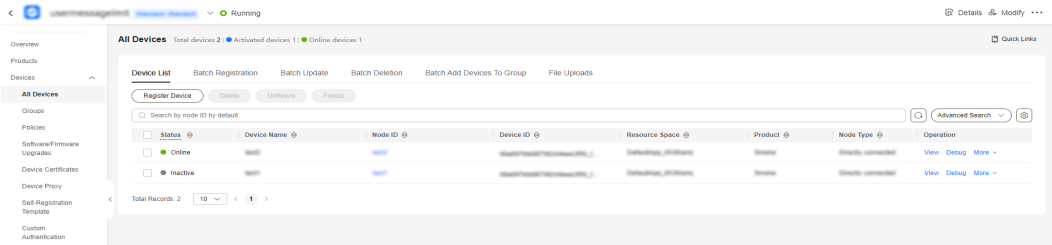


Table 6-3 Device list functions

Function	Description
Search for a device	Search for a specific device based on the status, device name, node ID, device ID, resource space, product, and node type.
View device information	View the **device status**, device name, and node ID in the device list. Click **View** in the row of a device to access the device details.
Delete a device	Click **Delete** in the row of a device to delete the device. **NOTE** After a device is deleted, the related device data is deleted. Exercise caution when performing this operation. To delete a large number of devices, you can call the API for **creating a batch task** or delete devices in batches on the IoTDA console. For details, see **Deleting a Batch of Devices**.
Freeze a device	Click **Freeze** in the row of a device to freeze the device. **NOTE** A frozen device cannot go online. Only devices that are directly connected to the platform can be frozen. To freeze a large number of devices, you can call the API for **creating a batch task**.
Unfreeze a device	Click **Unfreeze** in the row of a device to unfreeze the device. To unfreeze a large number of devices, you can call the API for **creating a batch task**.
Debug a device	Click **Debug** in the row of a device to debug the device.

----End

Device Status

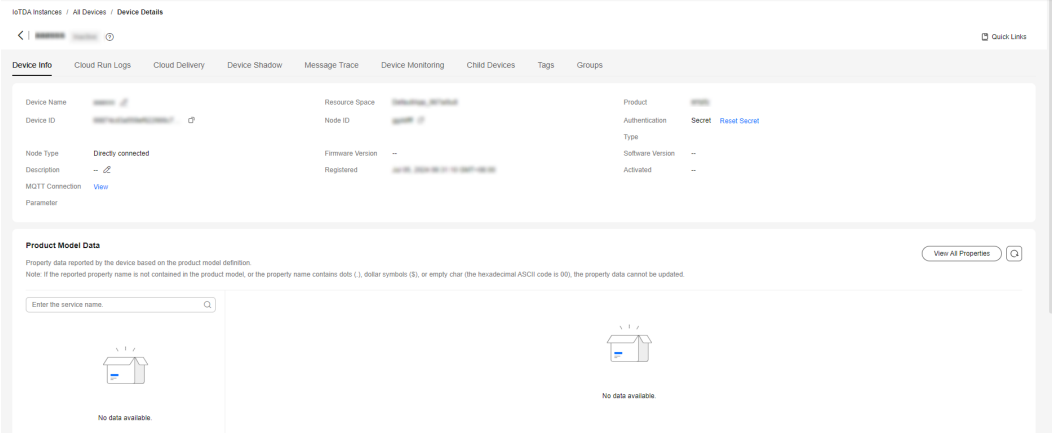
You can view the device status (online, offline, inactive, abnormal, or frozen) on the IoTDA console. You can also learn the device status by means of **subscription**. The table below describes the device statuses.

Type	Status	Short-Connection Device (Such as NB-IoT Devices)	Persistent Connection Device (MQTT Device)
Connection status	Online	If a device has reported data within 25 hours, the device status is **Online**. If no data has been reported within the past 25 hours, the device status is **Abnormal**.	The device is connected to the platform.
	Offline	If a device reports no data for 49 hours after connecting to the platform, the platform sets the device status to **Offline**.	After the device is disconnected from the platform for 1 minute (the data is automatically updated every minute), the device status is set to **Offline**. If you manually refresh the status on the page, the device status is displayed as **Offline**.
	Abnormal	If a device reports no data for 25 hours after connecting to the platform, the platform sets the device status to **Abnormal**.	This status does not apply to persistent connection devices.
	Inactive	The device is registered with but does not connect to the platform. The device activation procedure is described in **Initializing a Device**.	The device is registered with but does not connect to the platform. The device activation procedure is described in **Initializing a Device**.
Management status	Frozen	After a device is frozen, it cannot be connected to the IoT platform. Currently, only devices directly connected to the IoT platform can be frozen.	

Viewing Device Details

In the device list, click **View** in the row of a device to access its details.

Figure 6-19 Device - Device details



Tab	Description
Device Info	• Viewing device information: You can view basic device information, including the node ID, device ID, node type, software version, and firmware version. You can also call the API for **modifying a device**. – Node ID is a unique physical identifier for the device, such as its IMEI or MAC address. This parameter is used by the platform to authenticate the device during device access. – Device ID uniquely identifies a device. It is allocated by the platform during device registration and used for device access authentication and message transmission. • Resetting a secret: The secret is used for authentication when MQTT devices, NB-IoT devices, or SDK-integrated devices access the platform. After the secret is reset, the new secret must be updated on the device, and the device must carry the new secret for authentication during platform connection. • Viewing the latest reported data: View the latest data reported by the device to the platform.
Cloud Run Logs	IoTDA records connections with devices and applications. You can view the information on the console. For details, see **Run Logs (New Version)**.
Cloud Delivery	You can create a command or message (MQTT device only) delivery task for an individual device on the IoTDA console. For details, see **Data Delivery**.
Device Shadow	The platform provides the device shadow to cache the device status. When the device is online, delivered commands can be directly obtained. When the device is offline, it can proactively obtain the delivered commands after going online. For details, see **Device Shadow**.
Message Trace	The platform supports quick fault locating and cause analysis through message trace. For details, see **Message Trace**.

Tab	Description
Device Monitoring	• Device run logs: If you enable device log collection, local logs can be uploaded to Log Tank Service (LTS). (Note that this function is available only for MQTT devices.) • Anomaly detection: IoTDA provides device anomaly detection functions. For details, see **Anomaly Detection**.
Child Devices	Devices can be directly or indirectly connected to the IoT platform. Indirectly connected devices access the platform through gateways. For details, see **Gateways and Child Devices**.
Tags	You can define tags and bind tags to devices. For details, see **Tags**.
Groups	You can add devices to different groups for batch operations. For details, see **Groups and Tags**.

Deleting a Batch of Devices

To delete devices in batches on the IoTDA console, perform the following steps:

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **All Devices**, click the **Batch Deletion** tab, and click **Batch Deletion**.

Step 3 In the displayed dialog box, download **Batch Device Deletion Template**, enter the IDs of the devices to be deleted in the template, specify **Task Name**, upload the file, and click **OK**. Alternatively, you can specify a product to delete devices in batches.

Figure 6-20 Device - Deleting devices in batches (by file)

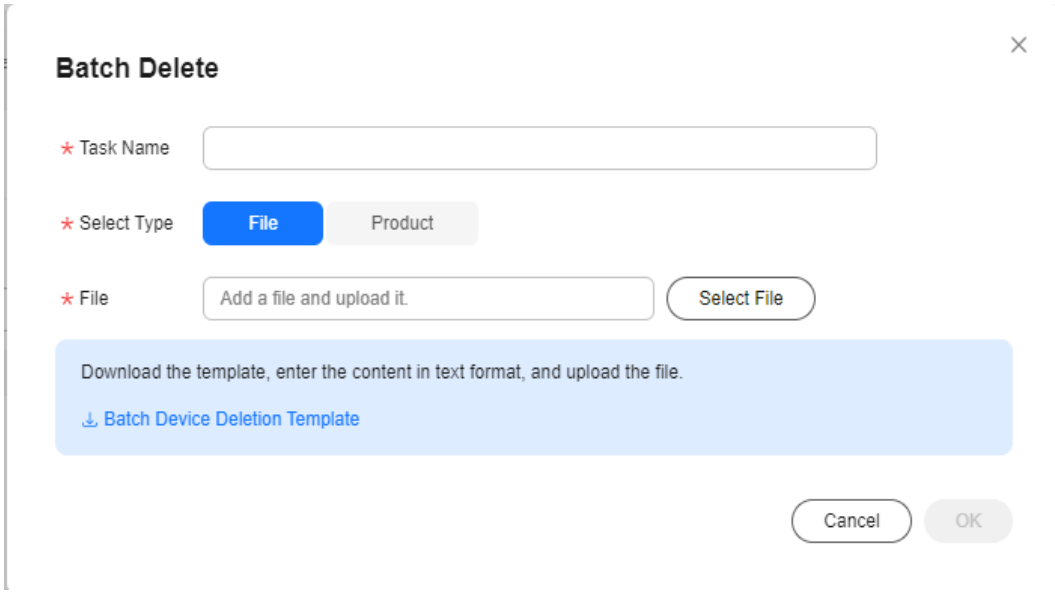
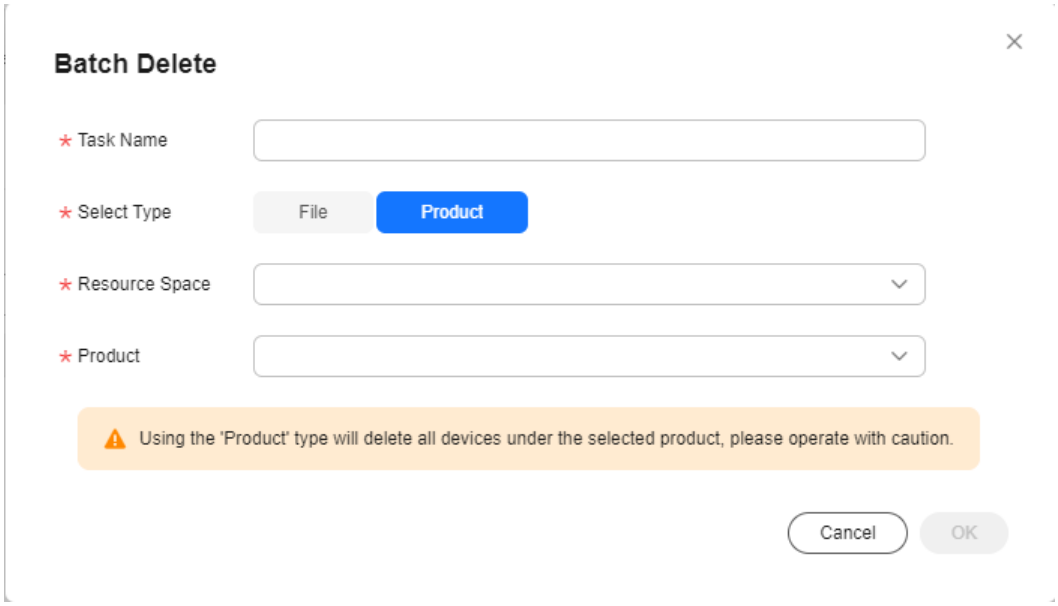


Figure 6-21 Device - Deleting devices in batches (by product)



The task execution status and result are displayed. If the success rate is not 100%, click the task name to open the task details page and view the failure cause.

----End

6.4 Groups and Tags

Group Introduction

A device group is a collection of devices. You can create groups for all the devices in a resource space based on rules (such as regions and types), and operate these devices by group. For example, you can perform a firmware upgrade on a group of water meters in the resource space. Devices in a group can be added, deleted, modified, and queried. A device can be bound to and unbound from multiple groups.

Table 6-4 Classification

Group Type	Description
Static group	You need to manually add devices to or remove devices from a group. Group nesting is supported. **Restrictions:** ● A maximum of 1,000 groups (including nested child groups) can be created for a single instance of an account. ● A maximum of 20,000 devices can be added to a group. ● A device can be bound to a maximum of 10 groups. ● A maximum of five group layers are supported. ● A child group can belong to only one parent group. ● If a group has child groups, the group cannot be deleted directly. You need to delete the child groups before deleting the parent group.
Dynamic group	Devices are automatically added to or removed from the group based on the dynamic query rules of SQL-like statements. You cannot manually manage devices in a dynamic group. **Restrictions:** ● A maximum of 10 dynamic groups can be created for a single instance of an account. ● When a dynamic group is created for the first time, a maximum of 100,000 devices can be matched. (There is no limit on the number of devices that can be added to the dynamic group later.) ● Dynamic groups are parent groups by default. Dynamic groups cannot be nested. ● After a dynamic group is created, its rules cannot be modified. ● Devices in a dynamic group cannot be manually managed. ● This API is supported only by standard and enterprise editions. ● The maximum TPS for an account to create a dynamic group is 1 (one request per second).

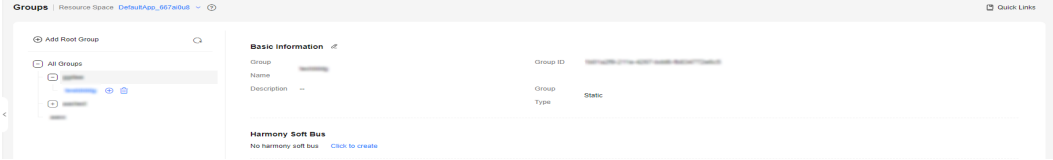
Managing Groups

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **Groups**.

Step 3 You can add, modify, or delete a group.

Figure 6-22 Device - Group



----End

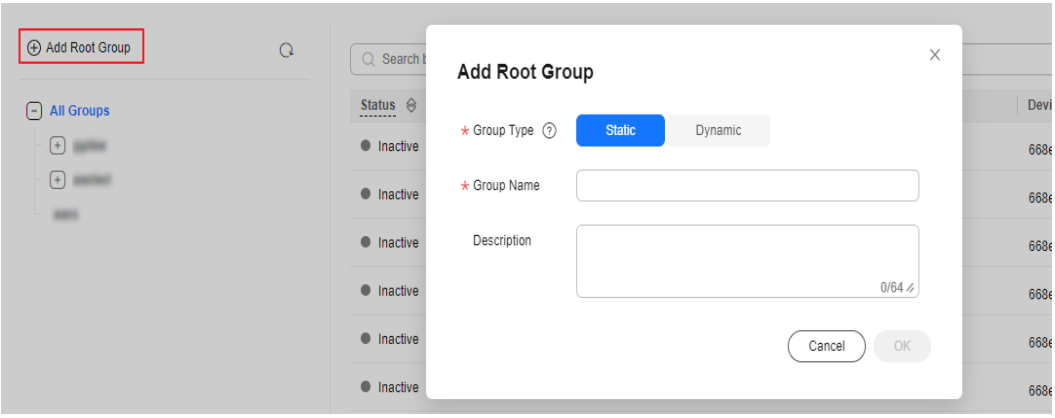
Static Group

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **Groups**.

Step 3 Click **Add Root Group** to add a group. Set group type to **Static Group**, set parameters as prompted, and click **OK**.

Figure 6-23 Group - Creating a static group



Step 4 Access the static group details page. Bind or unbind devices in the group. For details, see **Table 6-5**.

Figure 6-24 Static group - Binding a device

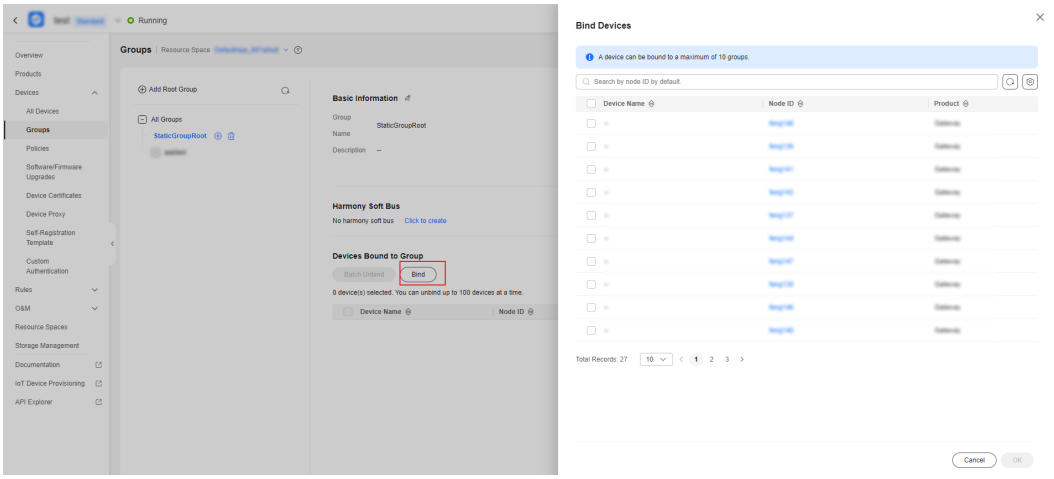
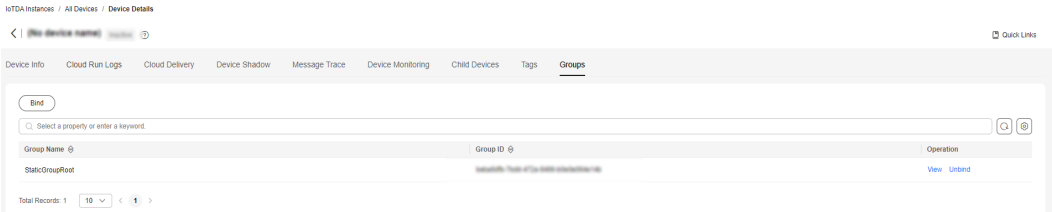


Table 6-5 Description

Operation	Description
Binding	Click **Bind** to bind a device to a group.
Batch unbinding	Select multiple devices (up to 100 devices at a time) and click **Batch Unbind** to unbind the selected devices from the current group.
Unbinding	Locate the target device and click **Unbind** to unbind the device from the group.

Step 5 In the navigation pane, choose **Devices** > **All Devices**. On the displayed page, locate the target device, click **View** in the **Operation** column. Click the **Groups** tab page to check and manage the associated groups. For details, see **Table 6-5**.

Figure 6-25 Device - Group management



----End

Dynamic group

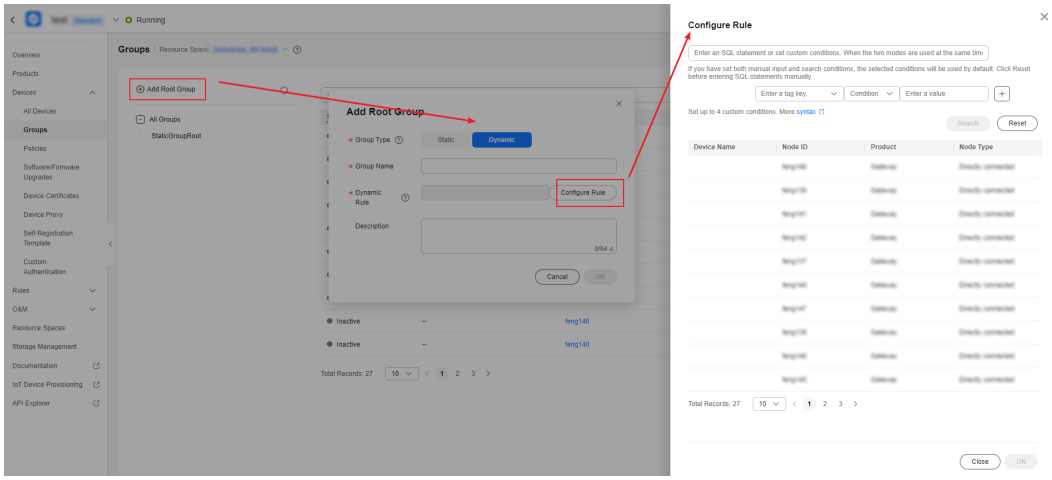
Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **Groups**.

Step 3 Click **Add Root Group** to add a group. Set **Group Type** to **Dynamic**.

Step 4 Set parameters as prompted, enter SQL-like statements, and click **Configure Rule** to check the matched devices. Click **OK** to complete the dynamic group creation.

Figure 6-26 Group - Creating a dynamic group



NOTE

- For details about the dynamic rule syntax, see **Advanced Search**.
- The difference between dynamic group rules and advanced search is that dynamic group rules do not support app_id and group_id filtering.
- You can click **Try** to enter a dynamic rule. After you enter a dynamic rule, click **OK**. The rule is automatically written back.

----End

Dynamic Group Example

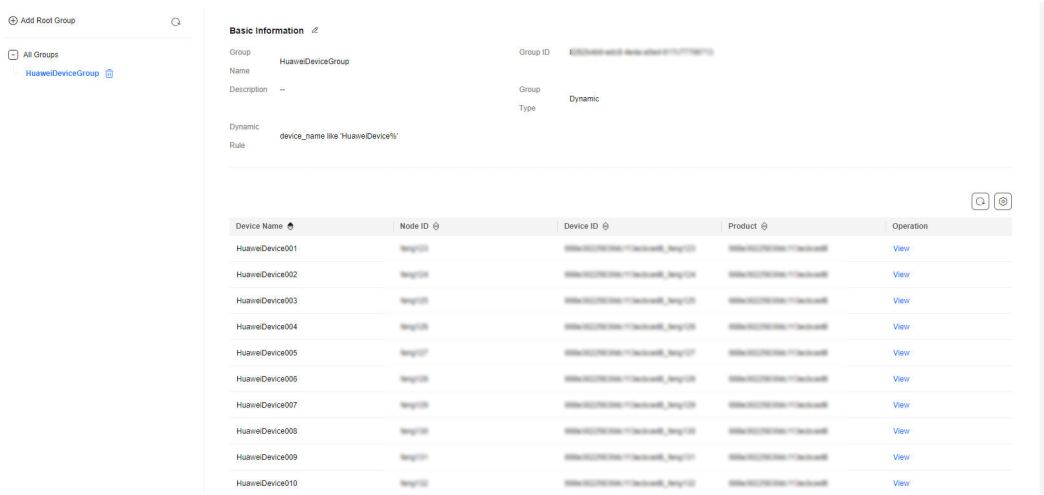
Create a dynamic group based on the device name fuzzy match rule (other conditions can be selected based on the site requirements) and select the dynamic group to execute the OTA upgrade task.

Devices in a dynamic group are dynamically adjusted based on device names, and the status of the OTA upgrade task associated with the dynamic group also changes dynamically.

For details, see **Upgrading the Firmware for a Batch of Devices** and **Dynamic group**.

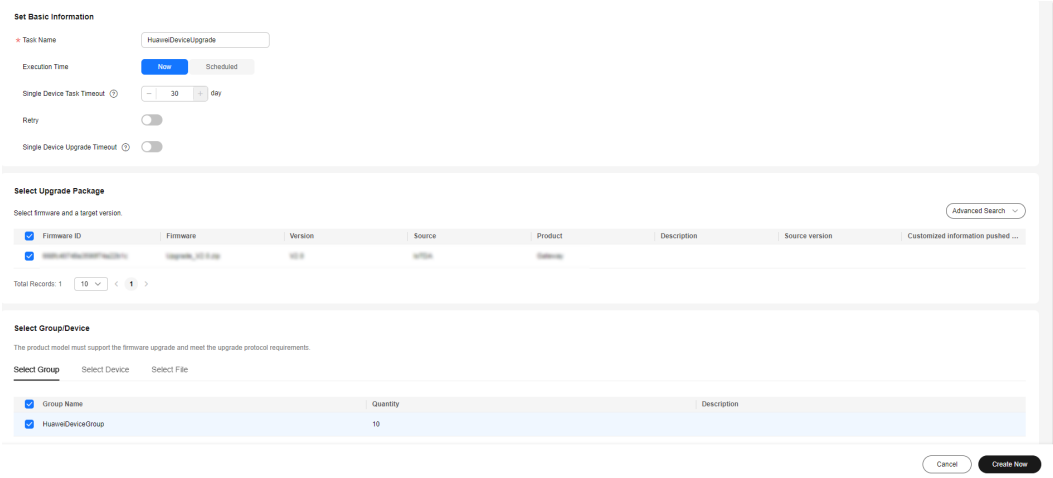
Step 1 Create a dynamic group named **HuaweiDeviceGroup** and set the group rule to **device_name like'HuaweiDevice %'**.

Figure 6-27 Dynamic group - Details



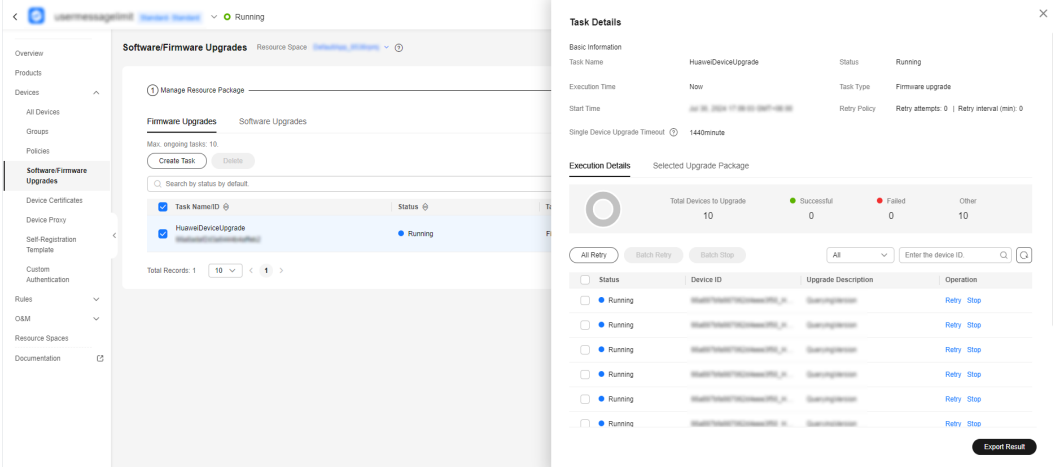
Step 2 Create a device firmware upgrade task and select the dynamic group **HuaweiDeviceGroup**.

Figure 6-28 Creating a firmware upgrade task - Dynamic group



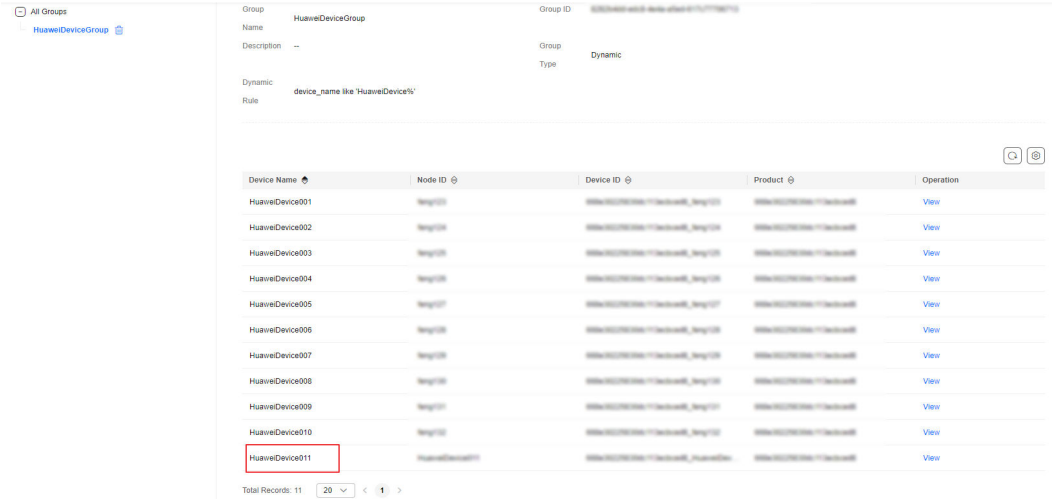
Step 3 After the dynamic group is created, you can view that the devices in the dynamic group are added to the upgrade task.

Figure 6-29 Firmware upgrade task - Details (dynamic group)



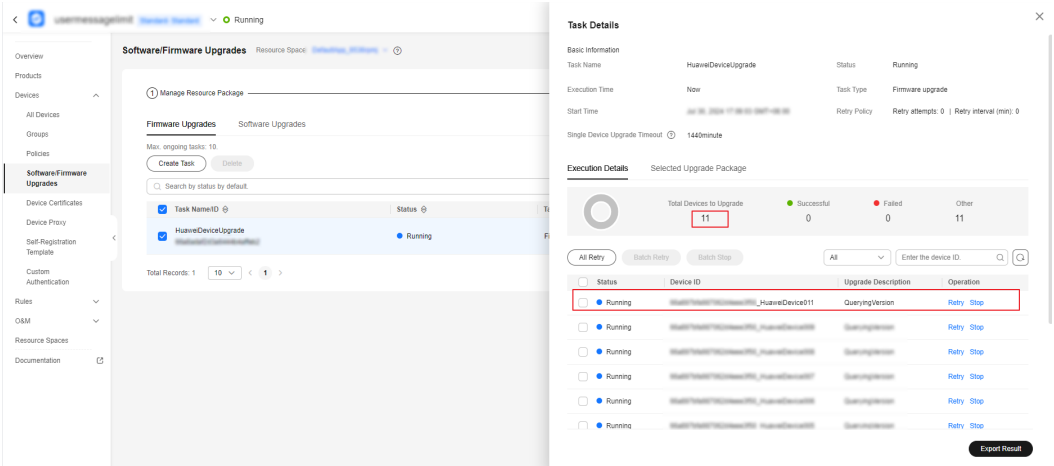
Step 4 Register a device by referring to **Registering a Single Device**. The device name is **HuaweiDevice011**. After the registration is successful, you can view that the device has been automatically added to the dynamic group **HuaweiDeviceGroup**.

Figure 6-30 Dynamic group - Adding a device



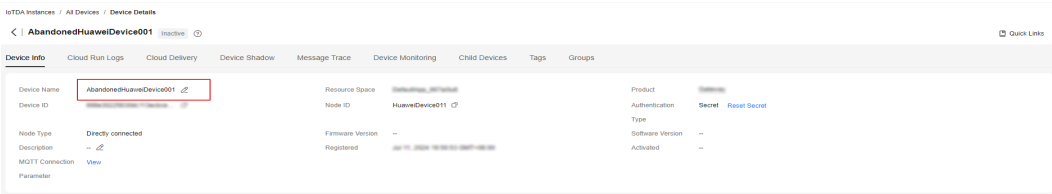
Step 5 View the sub-task details of the software and firmware upgrade task. You can see that the device has been automatically added to the upgrade task.

Figure 6-31 Firmware upgrade task - Adding a device to a dynamic group



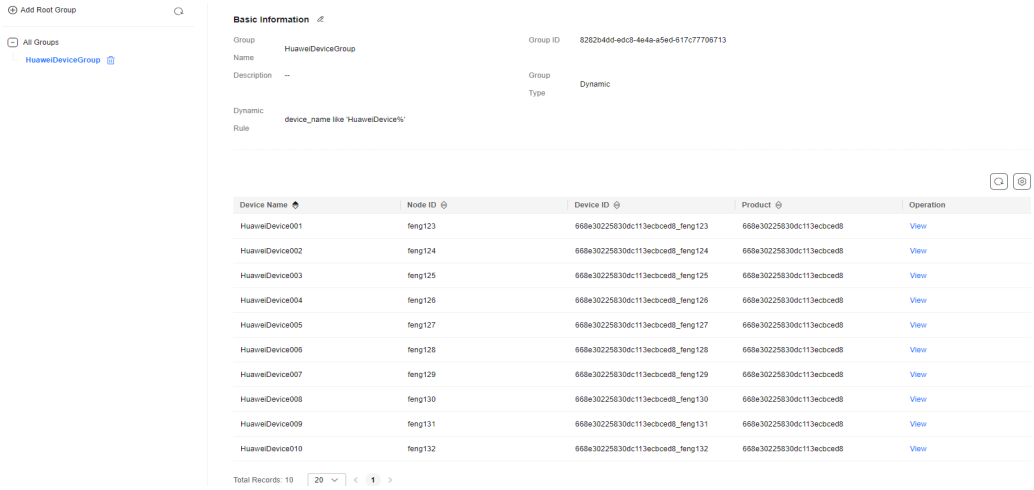
Step 6 On the **HuaweiDevice001** details page, change the device name to **AbandonedHuaweiDevice001**.

Figure 6-32 Device - Changing device name



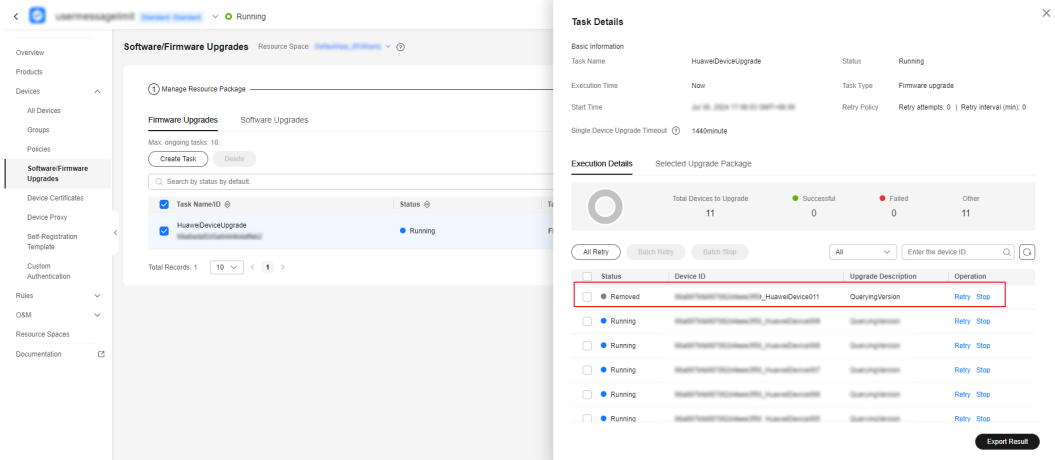
Step 7 After the device name is changed successfully, the device is automatically removed from the **HuaweiDeviceGroup** dynamic group.

Figure 6-33 Dynamic group - Removing a device



Step 8 Check the sub-task details of the software and firmware upgrade task. The upgrade status of the device is **Removed**.

Figure 6-34 Firmware upgrade task - Removing a device from a dynamic group



----End

Tags

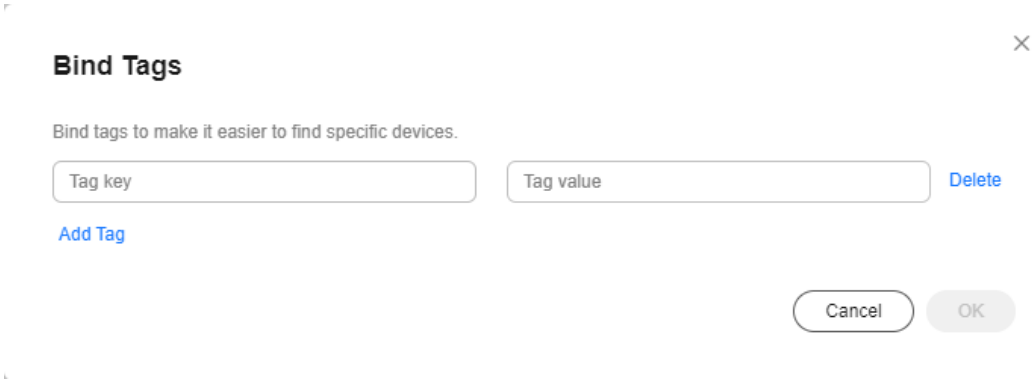
Tags are used to classify devices. You can bind tags to devices on the device details page to manage devices.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **All Devices**. On the displayed page, locate the target device, and click **View** in the **Operation** column to access its details page.

Step 3 On the **Tags** tab page, click **Bind Tags** to bind one or more tags to the device.

Figure 6-35 Device - Binding a tag



----End

Group-related APIs

Query the Device Group List

Create a Device Group

Query a Device Group

Modify a Device Group

Delete a Device Group

Manage Devices in a Device Group

Query Devices in a Group

Tag-related APIs

Binding Tags

Unbinding Tags

6.5 Advanced Search

Overview

To quickly find the desired device, you can use advanced search to set flexible search criteria using SQL-like statements to search. For example, you can search for devices by prefix fuzzy match or by tag. This section guides you on using advanced search and SQL-like syntax.

Constraints

- This API is supported only by Standard and Enterprise editions.
- The maximum TPS for an account to call this API is 1 (one request per second).

Scenarios

Device search: On the **All Devices** > **Device List** page, use SQL-like statements to search for specified devices for subsequent management operations.

Dynamic device grouping: Based on the rules of SQL-like statement, devices that meet the filter criteria are automatically added to the group for management.

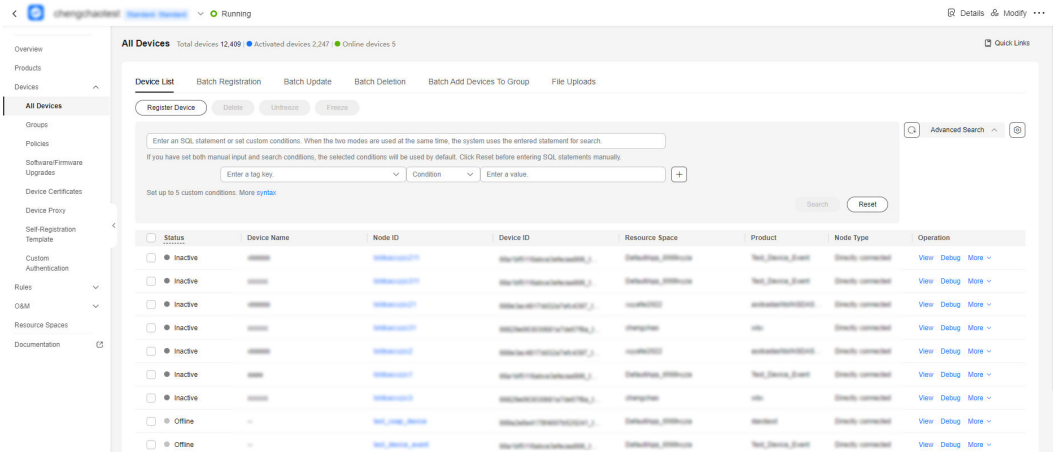
Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **All Devices**. By default, all devices under your account are displayed in the device list.

Step 3 Click **Advanced Search**, enter an SQL-like statement, and click **Search** to display the target devices.

Figure 6-36 Device - Advanced search



----End

SQL-like Syntax Description

When using SQL-like statements on the console, omit the select, from, order by, and limit clauses. **You only need to enter a where clause to edit user-defined conditions**. The maximum length of a statement is 400 characters. The content in the clause is case sensitive, but keywords in SQL statements are case insensitive. On the console, data is sorted based on the marker field desc by default.

A where clause:

```
[condition1] AND [condition2]
```

Example:

```
product_id = 'testProductId'
```

Up to five conditions are supported. Conditions cannot be nested. For details about the parameters that support query, see **Table 6-6** and **Table 6-7**.

AND and **OR** are supported. For details about the priority, see the standard SQL syntax. By default, the priority of **AND** is higher than that of **OR**.

Table 6-6 Description of query condition parameters

Parameter	Data Type	Description	Value Range
app_id	string	Resource space ID.	The value can contain up to 36 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed.
device_id	string	Device ID.	The value can contain up to 128 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed.
gateway_id	string	Gateway ID.	The value can contain up to 128 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed.

Parameter	Data Type	Description	Value Range
product_id	string	ID of the product associated with the device.	The value can contain up to 36 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed.
device_name	string	Device name.	The value can contain up to 256 characters. Only letters, digits, and special characters (_?'#().,&%@!-) are allowed.
node_id	string	Node ID.	The value can contain up to 64 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed.
status	string	Device status.	The value can be **ONLINE**, **OFFLINE**, **ABNORMAL**, **INACTIVE**, or **FROZEN**.
node_type	string	Device node type.	The value can be **GATEWAY** (a directly connected device or gateway) and **ENDPOINT** (an indirectly connected device).
tag_key	string	Tag key.	The value can contain up to 64 characters. Only letters, digits, underscores (_), periods (.), and hyphens (-) are allowed.
tag_value	string	Tag value.	The value can contain up to 128 characters. Only letters, digits, underscores (_), periods (.), and hyphens (-) are allowed.
sw_version	string	Software version.	The value can contain up to 64 characters. Only letters, digits, underscores (_), hyphens (-), and periods (.) are allowed.
fw_version	string	Firmware version.	The value can contain up to 64 characters. Only letters, digits, underscores (_), hyphens (-), and periods (.) are allowed.
group_id	string	Group ID.	The value can contain up to 36 characters, including hexadecimal strings and hyphens (-).
create_time	string	Device registration time.	Format: yyyy-MM-dd'T'HH:mm:ss.SSS'Z', for example, **2015-06-06T12:10:10.000Z**
marker	string	Result record ID.	The value is a string of 24 hexadecimal characters, for example, **ffffffffffffffffffffffff**.

Table 6-7 Supported operators

Operator	Supported By
=	All parameters
!=	All parameters
>	**create_time** and **marker**
<	**create_time** and **marker**
like	**device_name**, **node_id**, **tag_key**, and **tag_value**
in	Parameters except tag_key and tag_value.
not in	Parameters except tag_key and tag_value.

SQL Restrictions

- **like**: Only prefix match is supported. Suffix match or wildcard match is not supported. At least four characters must be contained for prefix match. Special characters cannot be contained. Only letters, digits, underscores (_), and hyphens (-) are allowed. The prefix must end with %.
- Other SQL statements, such as nested SQL statements, union, join, and alias, are not supported.
- The SQL statement can contain up to 400 characters. Up to five request conditions are supported.
- The condition value cannot be null or an empty string.

APIs

Query Device List Flexibly

6.6 Device Shadow

Overview

IoTDA supports the creation of device shadows. A device shadow is a JSON file that stores the device status, latest device properties reported, and device configurations to deliver. Each device has only one shadow. A device can retrieve and set its shadow to synchronize properties, either from the shadow to the device or from the device to the shadow.

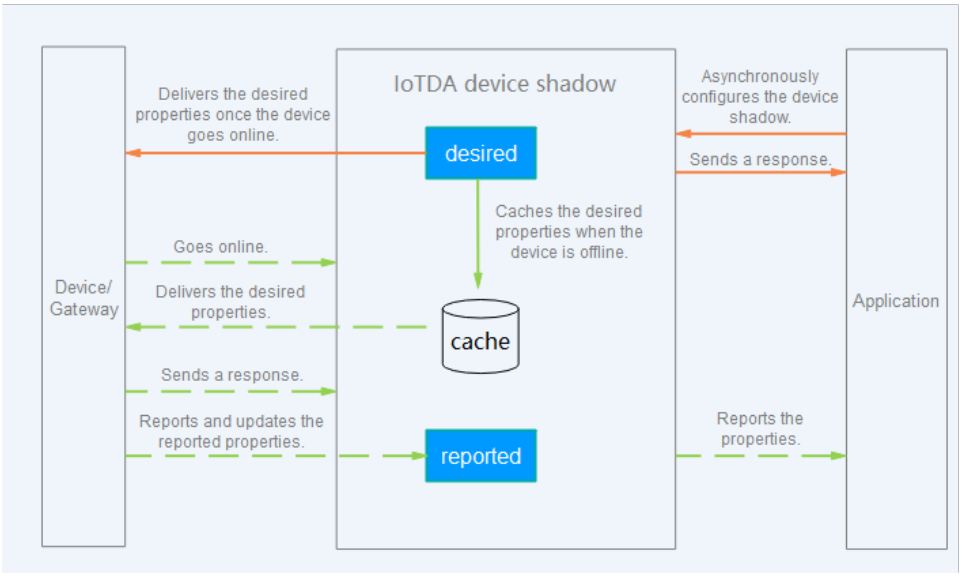
The device shadow contains two sections: desired and reported.

- The desired section stores the desired configurations of device properties. You can modify the desired properties in the device shadow when needed. If the device is online, the desired properties are synchronized to the device immediately. If the device is offline, the desired properties are synchronized to the device when the device goes online or reports data.

- The reported section stores the properties most recently reported by the device. When the device reports data, the platform changes the properties in the reported section to those reported by the device.

  NOTE

  - You can configure the device shadow by calling the application API or using the IoTDA console. (Specifically, access a device details page, click the **Device Shadow** tab, and click **Configure Property**.) The device shadow is mainly used to configure device properties. Its configuration depends on the **product model**.
  - The device shadow configuration is an asynchronous command. The platform directly returns a configuration response. Then, the platform determines whether to deliver the configuration immediately or cache the configuration based on the device status.
  - When the device goes online, the device shadow delivers the desired properties to the device. After the device reports its properties, the device shadow checks whether the reported properties match the delivered ones. If they match, the shadow data is configured on the device and the cache is cleared. If they do not match, the shadow data fails to be configured on the device. When the device goes online or reports properties next time, the platform delivers the desired properties to the device again until the configuration delivery is successful.
  - Restriction: Keys in the device shadow JSON file cannot contain periods (.), dollar signs ($), and the null character (hexadecimal ASCII code 00). Otherwise, the device shadow file cannot be refreshed.
  - When desired properties are delivered to a device, the device needs to return a response to indicate that the request has been received. If the device does not respond, the platform considers that the device does not support device shadow configuration and **sets device properties**. The IoT platform has a 5-minute protection period to prevent excessive traffic from affecting the device. During this period, the platform does not deliver the difference between reported and desired properties even if they are different. If the device responds to property configuration in the delivery process properly, the platform delivers the difference to the device each time the properties are reported.



Application Scenarios

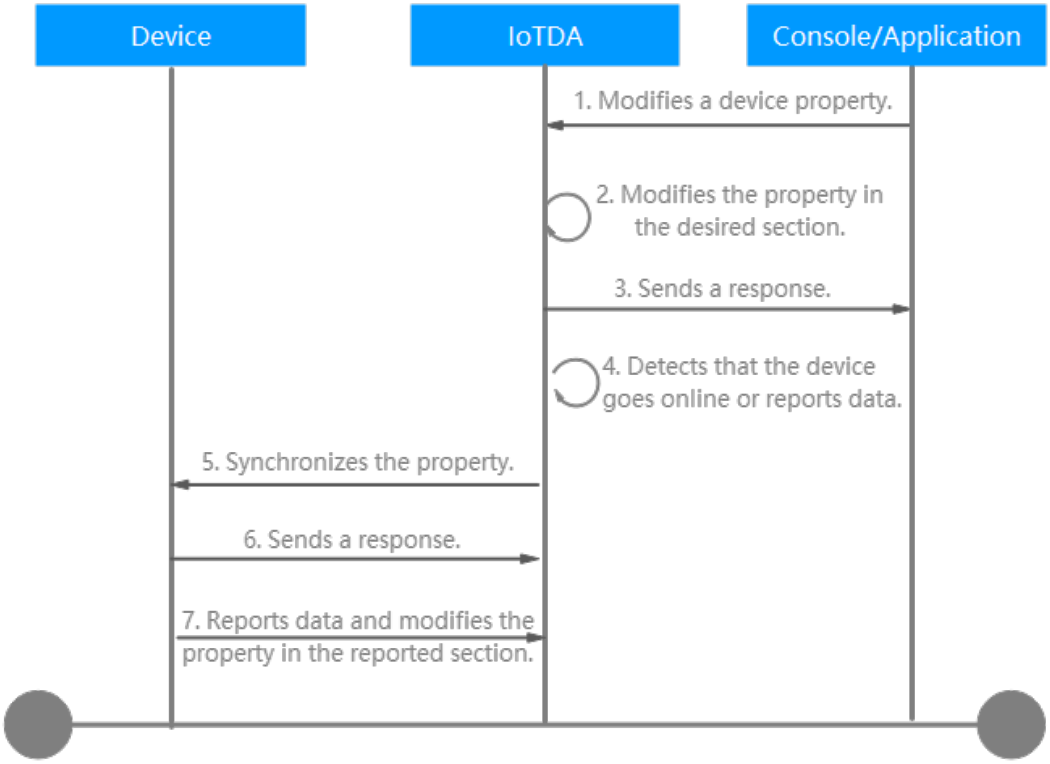
The device shadow is applicable to devices with limited resources and low power consumption or devices in the dormant state for a long time.

- Querying the latest data reported by the device and the latest online status of the device:
  - You may not be able to query the console for the mostly recent data because the device is offline or the network is unstable. With the device shadow, the platform can obtain the data from the shadow.
  - There may be too many applications simultaneously querying the device. IoT devices typically have limited processing capabilities, so too many queries can adversely affect their performance. With the device shadow, the device can synchronize its status to the shadow just once. The applications can obtain the device status from the device shadow, without reaching the real device.
- Modifying device properties: You can modify device properties on the device details page. Because the device may be offline for a long time, the modified device properties cannot be delivered to the device in time. The platform stores these properties in the device shadow. When the device goes online, the platform synchronizes the properties from the shadow to the device.

Service Flow

Modifying a Device Property

After a property in the desired section is modified, it is synchronized to the device immediately if the device is online, or cached and delivered until the device goes online or reports data.



1. A user modifies a device property on the console or application. Example message:

```
PUT https://{Endpoint}/v5/iot/{project_id}/devices/{device_id}/shadow
Content-Type: application/json
X-Auth-Token: ********
```

```
Instance-Id: ********

{
  "shadow" : [ {
    "desired" : {
      "temperature" : "60"
    },
    "service_id" : "WaterMeter",
    "version" : 1
  } ]
}
```

2. The platform modifies the property in the desired section.
3. The platform sends a response.
4. The platform detects that the device goes online or reports data.
5. The platform synchronizes the property to the device. Example message:

```
Topic: $oc/devices/{device_id}/sys/properties/set/request_id={request_id}
Data format:
{
    "object_device_id": "{object_device_id} ",
    "services": [
        {
            "service_id": "Temperature",
            "properties": {
                "value": 57,
                "value2": 60
            }
        },
        {
            "service_id": "Battery",
            "properties": {
                "level": 80,
                "level2": 90
            }
        }
    ]
}
```

6. The device sends a response. When desired properties are delivered to a device, the device needs to return a response to indicate that the request has been received. Example message:

```
Topic: $oc/devices/{device_id}/sys/properties/set/response/request_id={request_id}
Data format:
{
    "result_code": 0,
    "result_desc": "success"
}
```

7. When a device reports properties, the platform stores the latest property values reported by the device.

- When the device reports properties, the platform changes the properties in the reported section to those reported by the device. Example message:

```
Topic: $oc/devices/{device_id}/sys/properties/report
Data format:
{
    "services": [
        {
            "service_id": "Temperature",
            "properties": {
                "value": 57,
                "value2": 60
            },
            "event_time": "20151212T121212Z"
        },
        {
```

```
            "service_id": "Battery",
            "properties": {
                "level": 80,
                "level2": 90
            },
            "event_time": "20151212T121212Z"
        }
    ]
}
```

- The device proactively deletes the reported section of the device shadow.
  - The device proactively deletes a single property from **services** in the reported section.

    When a device reports a **null** property, the platform deletes the property from the reported section of the device shadow. An example message is as follows:

    ```
    Topic: $oc/devices/{device_id}/sys/properties/report
    {
        "services": [
            {
                "service_id": "Temperature",
                "properties": {
                    "value": null,
                    "value2": 60
                },
                "event_time": "20151212T121212Z"
            }
        ]
    }
    ```

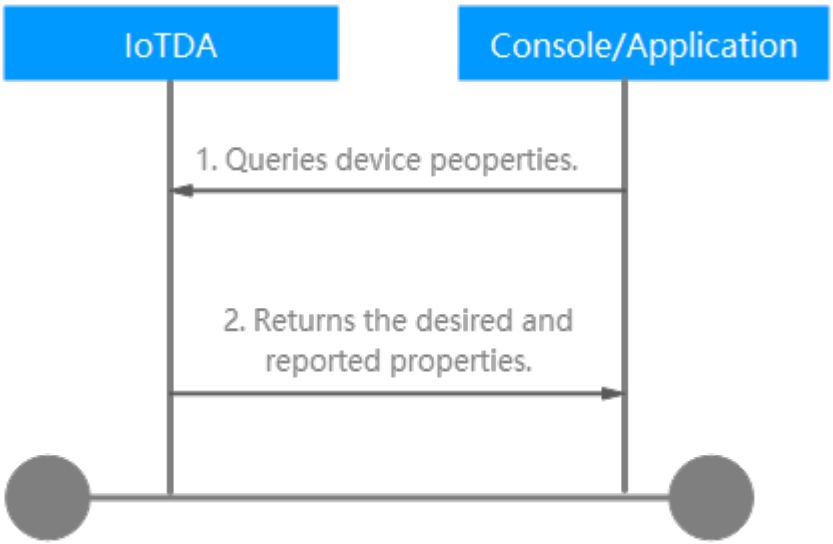
  - The device proactively deletes all properties from **services** in the reported section.

    When a device reports properties that are set to **{}**, the platform deletes all property from **services** in reported section of the device shadow. An example message is as follows:

    ```
    Topic: $oc/devices/{device_id}/sys/properties/report
    {
        "services": [
            {
                "service_id": "Temperature",
                "properties": {},
                "event_time": "20151212T121212Z"
            }
        ]
    }
    ```

Querying Device Properties

The device shadow saves the most recent device properties. Once the device properties change, the device synchronizes the changes to the device shadow. Using the device shadow, a user can obtain the device status quickly regardless of whether the device is online.



1. A user queries device properties on the console or application. Example message:

```
GET https://{Endpoint}/v5/iot/{project_id}/devices/{device_id}/shadow
Content-Type: application/json
X-Auth-Token: ********
Instance-Id: ********
```

2. The platform returns the desired and reported properties. Example message:

   Status Code: 200 OK

```
Content-Type: application/json

{
  "device_id" : "********",
  "shadow" : [ {
    "desired" : {
      "properties" : {
        "temperature" : "60"
      },
      "event_time" : "20151212T121212Z"
    },
    "service_id" : "WaterMeter",
    "reported" : {
      "properties" : {
        "temperature" : "60"
      },
      "event_time" : "20151212T121212Z"
    },
    "version" : 1
  } ]
}
```

Query, Modification, and Deletion

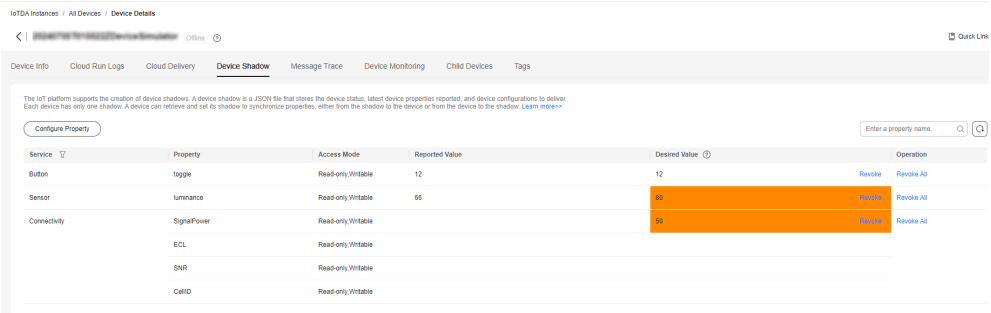
Querying a device shadow

Method 1: Use an application to call the API for **querying a device shadow**.

Method 2: Log in to the **console** and click the target instance card. In the navigation pane, choose **Devices** > **All Devices**. On the displayed page, locate the target device, and click **View** in the **Operation** column to access its details page. Click the **Device Shadow** tab, and check the device properties, including the reported and desired values.

- If the reported value is inconsistent with the desired value, the desired value is highlighted. This may occur when the device is offline and the value is still in the device shadow waiting to be synchronized to the device.
- If the reported value matches the desired value, the desired value is not highlighted. The latest property reported by the device matches the desired property.

Figure 6-37 Device shadow - Viewing

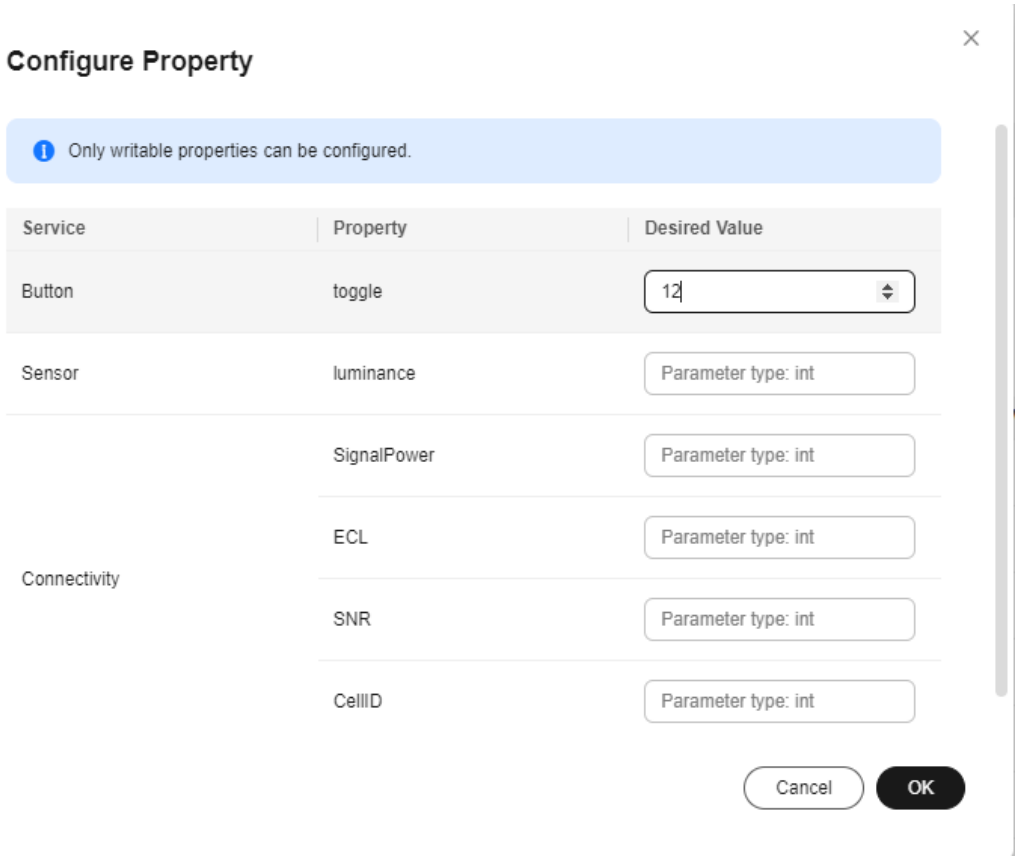


Modifying a device shadow

Method 1: Use an application to call the API for **configuring desired properties in a device shadow**.

Method 2: Log in to the **console** and click the target instance card. In the navigation pane, choose **Devices** > **All Devices**. On the displayed page, locate the target device, and click **View** in the **Operation** column to access its details page. Click the **Device Shadow** tab, and click **Configure Property**. In the displayed dialog box, enter the desired value and click **OK**.

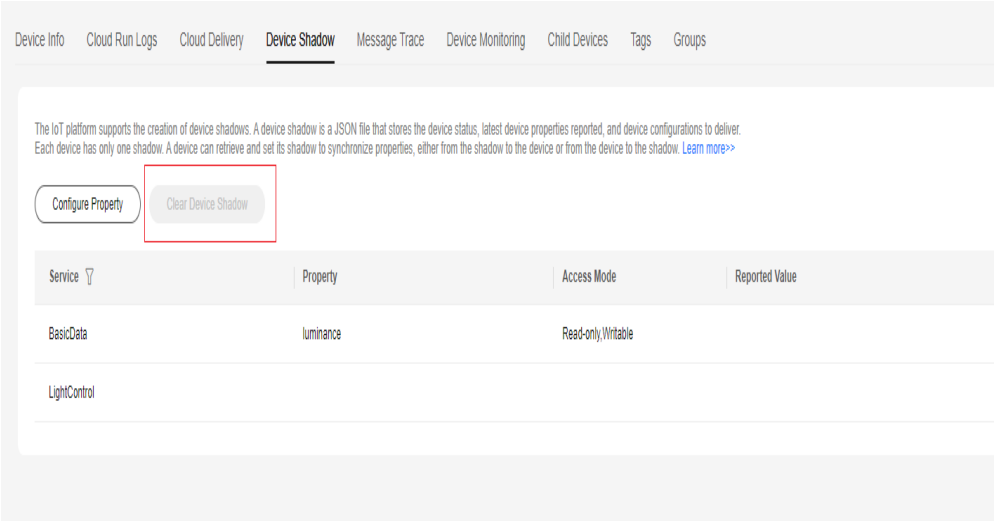
Figure 6-38 Device shadow - Configuring property



Deleting a device shadow

After a device shadow is deleted, the platform clears all data (including the reported and desired values) in the device shadow.

Figure 6-39 Deleting a device shadow



APIs

Querying a Device Shadow

Configuring Desired Properties in a Device Shadow

6.7 OTA Upgrade

6.7.1 Software/Firmware Package Upload

Overview

Software includes system software and application software. The system software provides the basic device functions, such as the compilation tool and system file management. The application software provides functions such as data collection, analysis, and processing, depending on the features the device provides. Software upgrade, also called software over the air (SOTA), allows you to upgrade the software of LwM2M or MQTT devices in OTA mode.

- For an LwM2M product model, the software upgrade complies with **PCP**. You must comply with PCP during **device adaptation development** of software upgrades.
- For an MQTT product model, the software upgrade protocol is not verified.

Firmware is like a device driver for the hardware. It is responsible for the underlying work of a system, for example, the basic input/output system (BIOS) on a computer mainboard. Firmware upgrade, also called firmware over the air (FOTA), allows you to upgrade the firmware of LwM2M or MQTT devices in OTA mode.

Procedure

You need to add an upgrade package on the IoTDA console to upgrade device software and firmware. You can either use an OBS file as an upgrade package or upload a local upgrade package.

NOTE

- The size of the OBS file cannot exceed 1 GB. You will be billed for storing and downloading the OBS file.

  OBS billing items include the storage space, request, data transfer, data restoration, and data processing. Billing modes include pay-per-use and yearly/monthly. For details, see **Billing**.

  For example, if a user in CN North-Beijing4 needs to upgrade 10,000 devices per month and the size of the upgrade package is 100 MB, you will be charged CNY512.15 in total (including CNY0.139 for storage, CNY512 for data transfer, and 0.01 for requests) on a pay-per-use basis.

  If the yearly/monthly billing mode is used, you will be charged CNY506.01 in total (including CNY1.00 for storage, CNY505.00 for 1 TB Internet outbound traffic package for a month, and CNY0.01 for requests).
- You are not billed for uploading local upgrade packages. The maximum package size is 20 MB.
- The upgrade package format can only be .bin, .dav, .tar, .gz, .zip, .gzip, .apk, .tar.gz, .tar.xz, .pack, .exe, .bat, or .img.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **Software/Firmware Upgrades**.

Step 3 To upload a firmware upgrade package, click the **Firmware List** tab and click **Upload**. To upload a software upgrade package, click the **Software List** tab and click **Upload**.

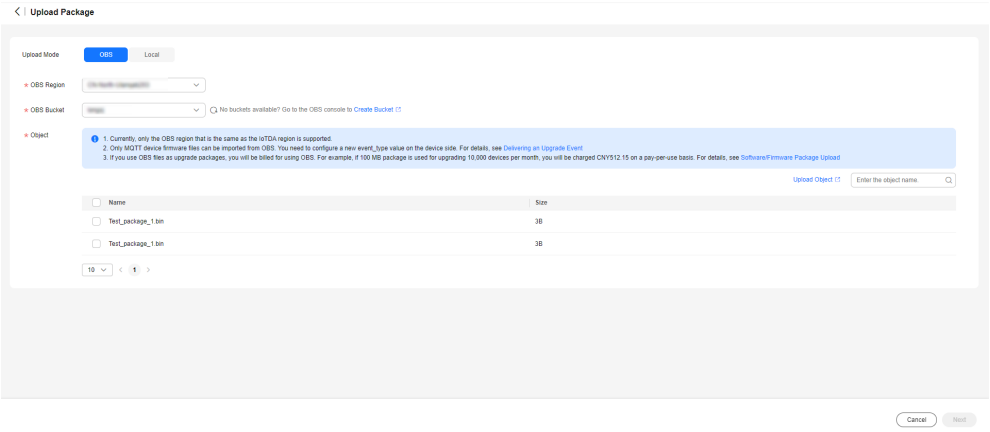
Step 4 On the package upload page, select **OBS** or **Local** for **Upload Mode**.

- **OBS**: Before using this mode, click **Agree** in the displayed authorization dialog box. After authorization, select the bucket where the upgrade file is located. If no bucket is available, click **Create Bucket** to go to the OBS page and create a bucket. If the upgrade file is not uploaded to the corresponding bucket, click **Upload Object** to go to the OBS bucket page and upload the upgrade file. Select the target OBS object and click **Next**.

NOTICE

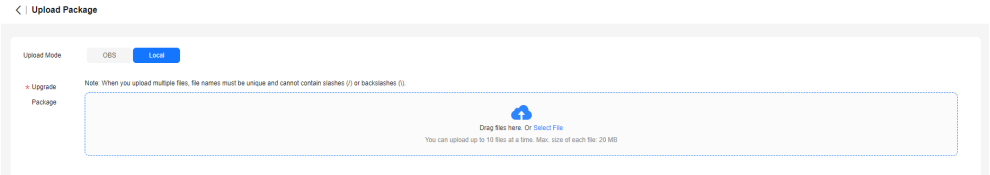
If you have not authorized IoTDA to access Key Management Service (KMS), click **Authorize Access** in **Encryption Service Authorization**. Otherwise, upgrade file downloading will be affected if you set or modify **Default Encryption** for the bucket on the OBS console.

Figure 6-40 Uploading the upgrade package - OBS file



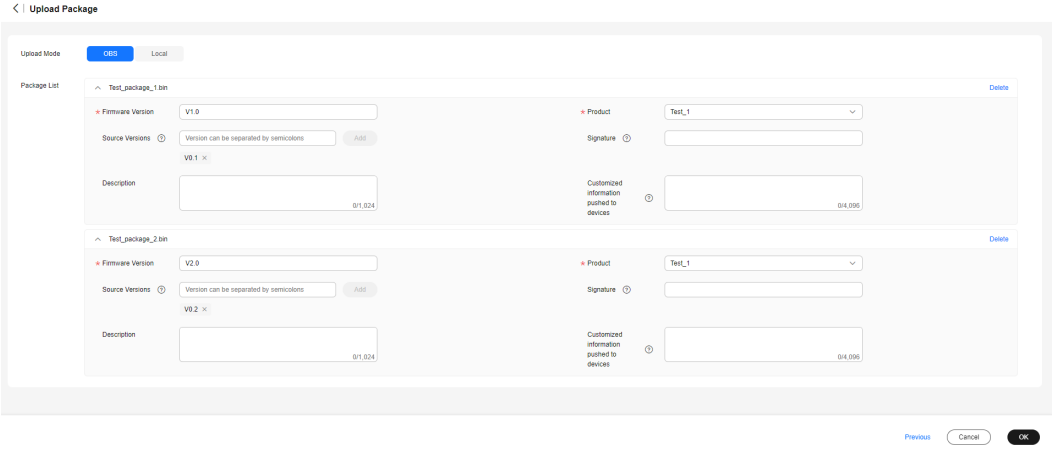
- **Local**: Drag and drop a file or click **Select File** to upload a software/firmware upgrade package.

Figure 6-41 Uploading the upgrade package - local file



Step 5 Configure parameters in the upgrade package list based on the following information, and click **OK**.

Figure 6-42 Uploading the upgrade package - OBS file parameters



Fill in the parameters as follows:

Parameter	Description
Firmware/Software Version	Version of the firmware/software package. (A device reports the version number after the upgrade. The platform checks whether the version number reported by the device is the same as the value of this parameter. If they are the same, the upgrade is successful.)
Product	Select the **product model** of the corresponding device.
Source Versions	Source version of the device that can be upgraded. Enter the version manually. To add multiple versions, press **Enter** after inputting one version, and then input the next. **NOTE** Currently, the platform does not support automatic differential upgrade packages. You can prepare differential packages on your local PC and upload them to the platform. Then, specify different source versions for these differential packages. You can select multiple differential packages when creating an upgrade task. During a software/firmware upgrade, the platform delivers differential packages based on source versions reported by devices.

Parameter	Description
Software Package Segment Size	Size of each segment of the software package downloaded by the device, in bytes. The value ranges from 32 to 500. The default value is **500**. This function is supported only by NB-IoT device software upgrade tasks.
Description	Description of the firmware/software package.
Customized information pushed to devices	The platform delivers the custom information when delivering an **upgrade notification** to devices.

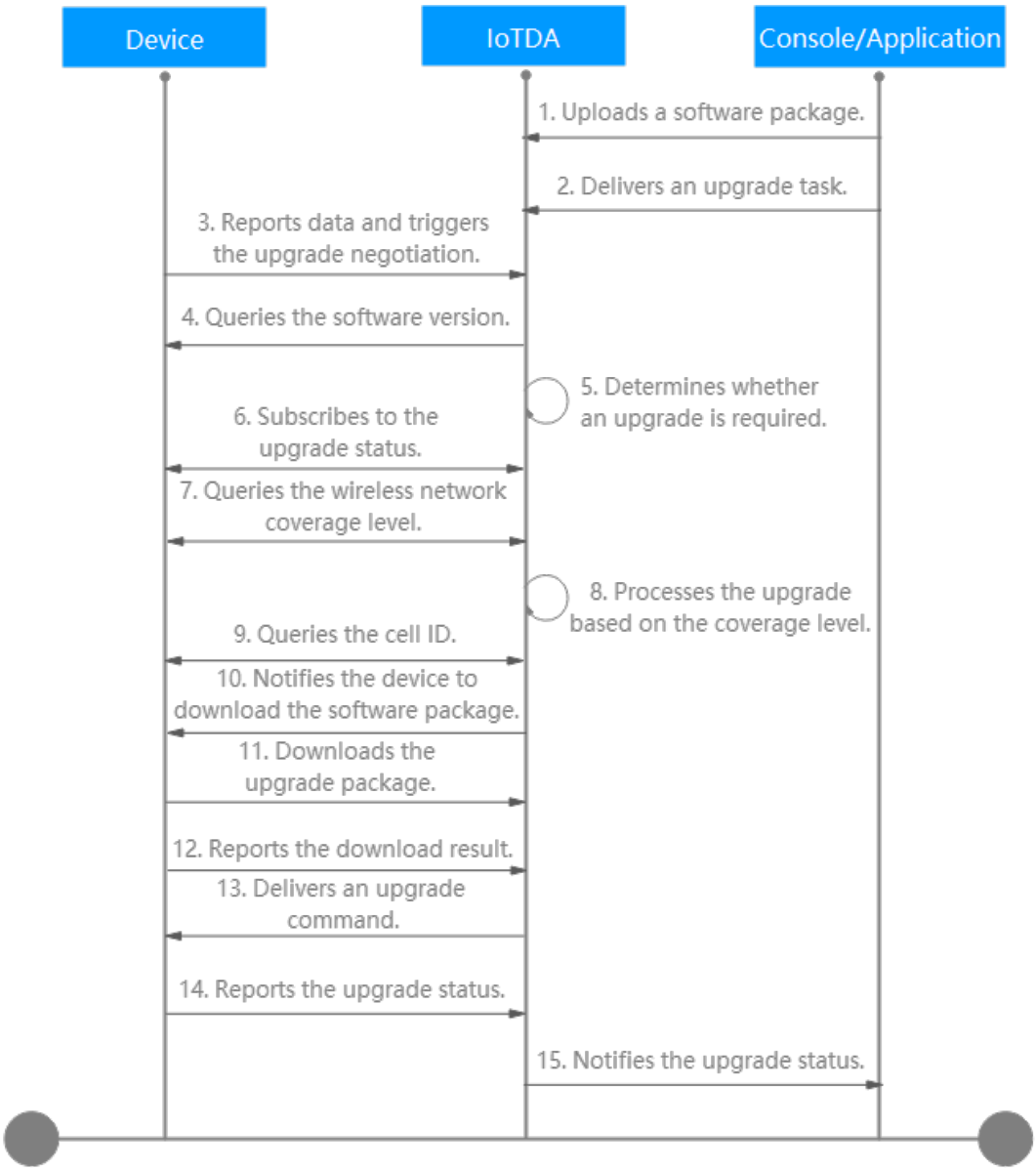
NOTE

- The function of uploading signed software/firmware packages has been brought offline. Uploaded signed software/firmware packages can still be used for upgrade. To ensure proper use of software/firmware upgrades, directly upload the upgrade files to be delivered to devices.
- Only MQTT devices can use OBS files as software/firmware upgrade files. You need to configure a new **event_type** value on the device side.
- If no device source version is specified for an upgrade package, all selected devices will be upgraded.

----End

6.7.2 OTA Upgrade for NB-IoT Devices

Software Upgrade for Devices Using LwM2M over CoAP



The software upgrade process for a device using LwM2M over CoAP is as follows:

1–2: A user uploads a software package on the IoTDA console and creates a software upgrade task on the console or an application.

3. A device reports data to the platform. The platform detects that the device is online and triggers the upgrade negotiation process. (The timeout interval is 24 hours.)

4–5: The platform delivers a command to the device to query its software version and determines whether an upgrade is required based on the target version. (In step 4, the timeout interval for the device to report the software version is 3 minutes.)

- If the returned software version is the same as the target version, no upgrade is required.
- If the returned software version is different from the target version, the platform continues the upgrade.

6. The platform subscribes to the software upgrade status from the device.

7–8: The platform queries the radio coverage of the cell where the device resides, and obtains the cell ID, reference signal received power (RSRP), and signal to interference plus noise ratio (SINR). (The timeout interval for the reporting of the radio coverage level and cell ID is about 3 minutes.)

- If the query is successful, the platform calculates the number of concurrent upgrade tasks based on the RSRP and SINR ranges described in the figure below, and continues with **step 10**.
  - RSRP and SINR in range 0: 50 devices in the cell can be upgraded simultaneously.
  - RSRP in range 0 and SINR in range 1: 10 devices in the cell can be upgraded simultaneously.
  - RSRP in range 1 and SINR in range 2: Only one device in the cell can be upgraded at a time.
  - RSRP and SINR can be queried but are not within any of the three ranges: Only one device in the cell can be upgraded at a time.

Radio Coverage Range	RSRP Range (dBm)	SINR Range (dB)
0	–105 ≤ RSRP	7 ≤ SINR
1	–115 ≤ RSRP < –105	–3 ≤ SINR < 7
2	–125 ≤ RSRP < –115	–8 ≤ SINR < –3

NOTE

If only a small number of devices can be upgraded simultaneously, you can contact the local carrier to see if coverage can be improved.

- If the query fails, the process continues with **step 9**.

9. The platform delivers a command to query the cell ID of the device.

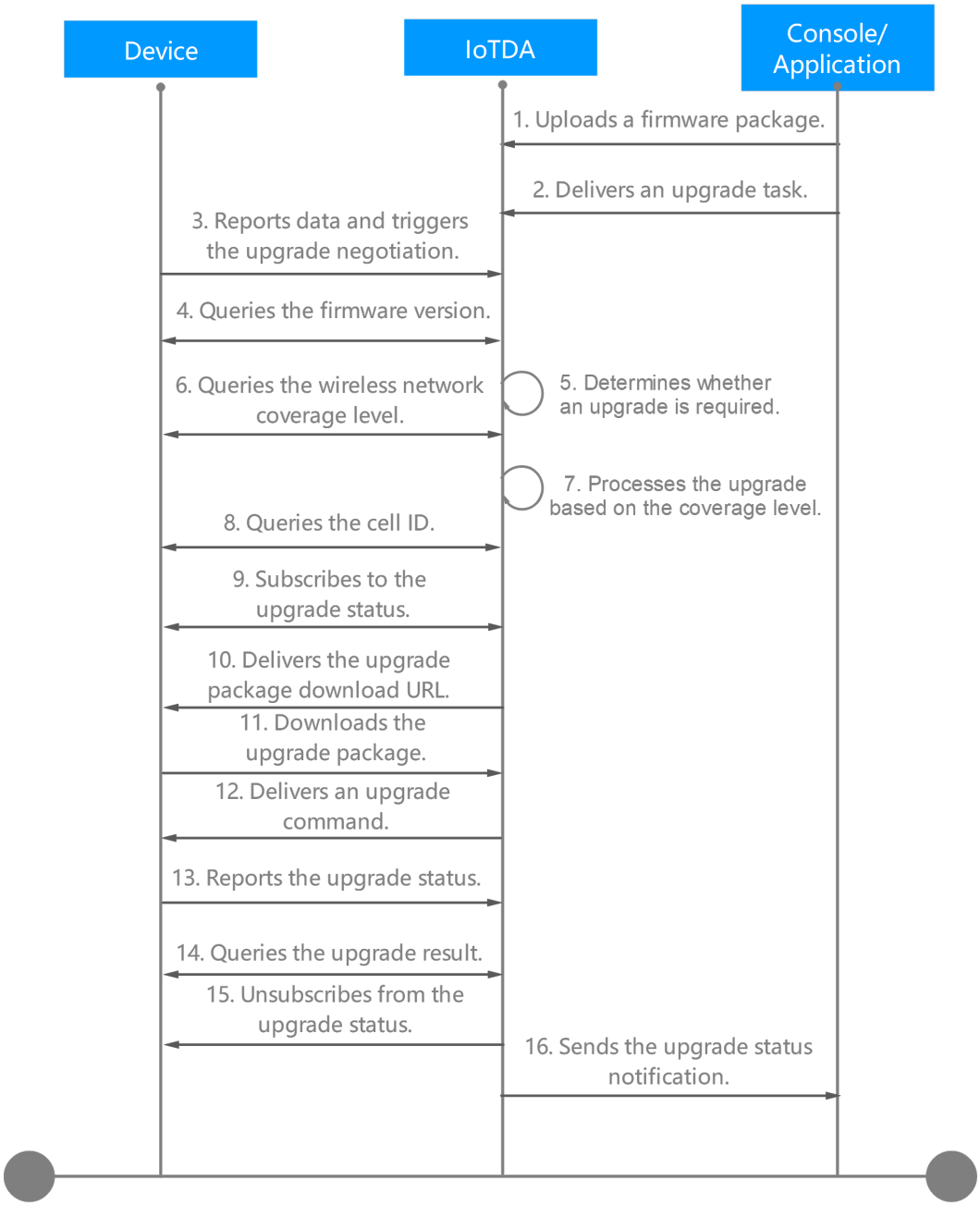
- If the query is successful, 10 devices in the cell can be upgraded simultaneously.
- If the query fails, the upgrade fails.

10–12: The platform notifies the device of a new software package version. The device starts to download the software package. Software packages can be downloaded in segments, and resumable download is supported. The **versionCheckCode** field carried in a software package segment determines the software package to which the segment belongs. After the download is complete, the device notifies the platform. (The timeout interval for step 11 is 60 minutes.)

13–14: The platform delivers an upgrade command to the device, and the device performs the upgrade. After the upgrade is complete, the device notifies the platform. (The timeout interval for the device to report the upgrade result and status is 30 minutes.)

15. The platform notifies the IoTDA console or application of the upgrade result.

Firmware Upgrade for Devices Using LwM2M



The firmware upgrade process for a device using LwM2M is as follows:

1–2: A user uploads a firmware package on the IoTDA console and creates a firmware upgrade task on the console or an application.

3. A device reports data to the platform. The platform detects that the device is online and triggers the upgrade negotiation process. (The timeout interval is 24 hours.)

4–5: The platform delivers a command to query the device firmware version and determines whether an upgrade is required based on the target version. (In step 4, the timeout interval for the device to report the firmware version is 3 minutes.)

- If the returned firmware version is the same as the target version, no upgrade is required.
- If the returned firmware version is different from the target version, the platform continues the upgrade.

6–7: The platform queries the radio coverage of the cell where the device resides, and obtains the cell ID, RSRP, and SINR. (The timeout interval for the reporting of the radio coverage level and cell ID is about 3 minutes.)

- If the query is successful, the platform calculates the number of concurrent upgrade tasks based on the RSRP and SINR ranges described in the figure below, and continues with **step 9**.
  - RSRP and SINR in range 0: 50 devices in the cell can be upgraded simultaneously.
  - RSRP in range 0 and SINR in range 1: 10 devices in the cell can be upgraded simultaneously.
  - RSRP in range 1 and SINR in range 2: Only one device in the cell can be upgraded at a time.
  - RSRP and SINR can be queried but are not within any of the three ranges: Only one device in the cell can be upgraded at a time.

Radio Coverage Range	RSRP Range (dBm)	SINR Range (dB)
0	–105 ≤ RSRP	7 ≤ SINR
1	–115 ≤ RSRP < –105	–3 ≤ SINR < 7
2	–125 ≤ RSRP < –115	–8 ≤ SINR < –3

NOTE

If only a small number of devices can be upgraded simultaneously, you can contact the local carrier to see if coverage can be improved.

- If the query fails, the process continues with **step 8**.

8. The platform delivers a command to query the cell ID of the device.

- If the query is successful, 10 devices in the cell can be upgraded simultaneously.
- If the query fails, the upgrade fails.

9. The platform subscribes to the firmware upgrade status from the device.

10–11: The platform delivers the package download URL. The device downloads the firmware package from the URL. After the download is complete, the device notifies the platform. (Firmware packages can be downloaded in segments, and resumable download is supported.) (The timeout interval for step 11 is 60 minutes.)

12–13: The platform delivers an upgrade command to the device, and the device performs the upgrade. After the upgrade is complete, the device notifies the

platform. (The timeout interval for the device to report the upgrade result and status is 30 minutes.)

14–16: The platform delivers a command to query the firmware upgrade result. After obtaining the result, the platform unsubscribes from the upgrade status and notifies the IoTDA console or application of the upgrade result.

NOTE

The platform supports resumable download.

Firmware Upgrade Failure Causes

The following table lists the failure causes reported by the platform.

Error Message	Description	Solution
Device Abnormal is not online	The device is offline or abnormal.	Check the device.
Task Conflict	A task conflict occurs.	Check whether a software upgrade, firmware upgrade, log collection, or device restart task is in progress.
Waiting for the device online timeout	The device does not go online within the specified time.	Check the device.
Wait for the device to report upgrade result timeout	The device does not report the upgrade result within the specified time.	Check the device.
Waiting for report device firmware version timeout	The device does not report the firmware version within the specified time.	Check the device.
Waiting for report cellId timeout	The device does not report the cell ID within the specified time.	Check the device.
Updating timeout and query device version for check timeout	The device does not report the upgrade result or device version within the specified time.	Check the device.
Waiting for device downloaded package timeout	The device does not finish downloading the firmware package within the specified time.	Check the device.

Error Message	Description	Solution
Waiting for device start to update timeout	The device does not start the update within the specified time.	Check the device.
Waiting for device start download package timeout	The device does not start to download the firmware package within the specified time.	Check the device.

The following table lists the failure causes reported by devices.

Error Message	Description	Solution
Not enough storage for the new firmware package	The storage space is insufficient for the firmware package.	Check the storage space of the device.
Out of memory during downloading process	The memory was insufficient during the download.	Check the device memory.
Connection lost during downloading process	The connection was interrupted during the download.	Check the device connection status.
Integrity check failure for new downloaded package	The integrity check on the firmware package fails.	Check whether the firmware package downloaded is complete.
Unsupported package type	The firmware package type is not supported.	Check whether the device status and firmware package provided by the manufacturer are correct.
Invalid URI	The URI is invalid.	Check whether the download address of the firmware package is correct.
Firmware update failed	The firmware fails to update.	Check the device.

FAQs

The following lists the frequently asked questions about software and firmware upgrades. For more questions, see **OTA Upgrades**.

- **Can the Target Version Be Earlier Than the Source Version?**

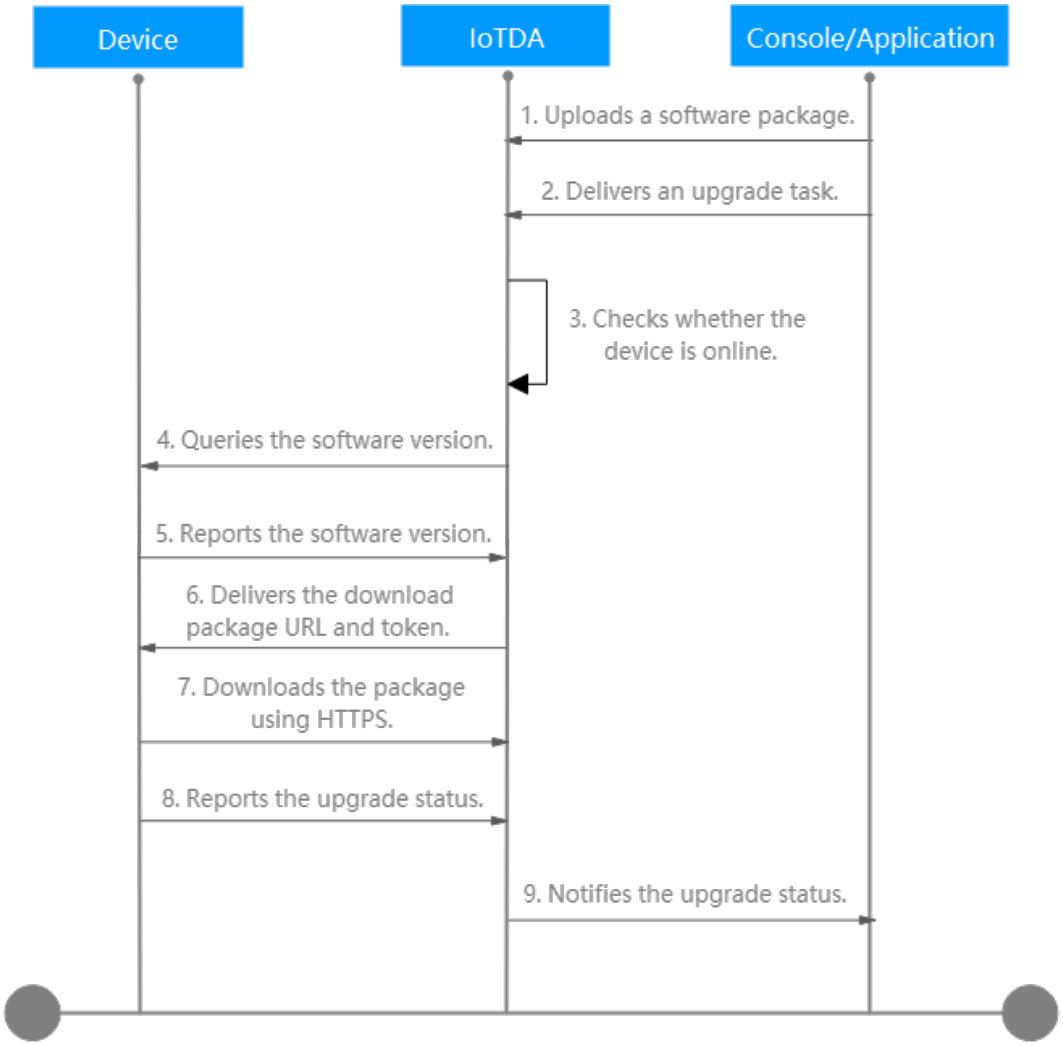
- How Do I Obtain Software or Firmware Packages and Their Version Numbers?
- Are Services Interrupted During a Software or Firmware Upgrade?
- What Are Common Software or Firmware Upgrade Errors?

APIs

- Create a Batch Task
- Query the Batch Task List
- Query a Batch Task

6.7.3 OTA Upgrade for MQTT Devices

Software Upgrade for Devices Using MQTT



The software upgrade process for a device using MQTT is as follows:

1–2: A user uploads a software package on the IoTDA console and creates a software upgrade task on the console or an application.

3. The platform checks whether the device is online and triggers the upgrade negotiation process immediately when the device is online. If the device is offline,

the platform waits for the device to go online and **subscribes to the upgrade topic**. After detecting that the device goes online, the platform triggers the upgrade negotiation process. (The timeout interval for the device to go online is within 25 hours.)

4–5: The platform delivers a command to the device to query its software version and determines whether an upgrade is required based on the target version. (The timeout interval for step 5 is 3 minutes.)

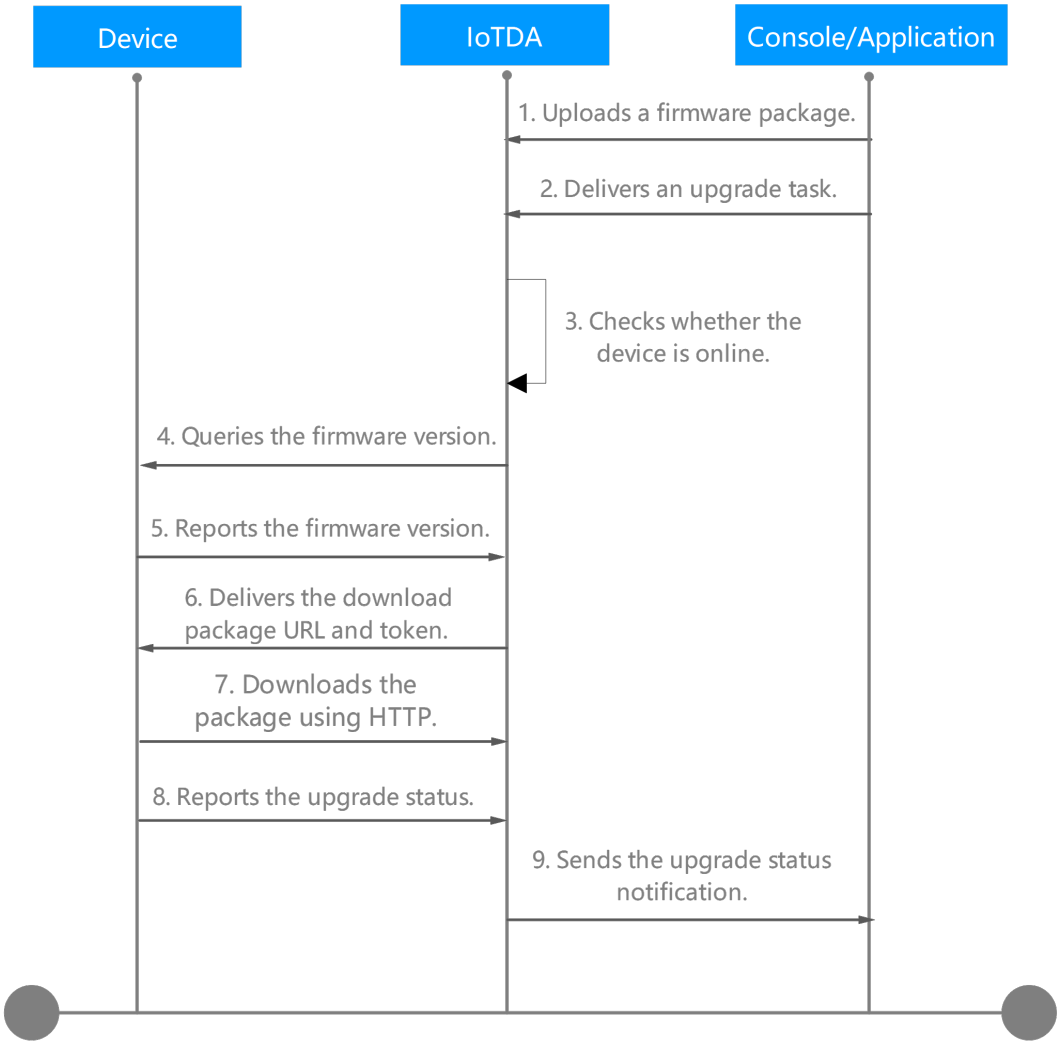
- If the returned software version is the same as the target version, no upgrade is required. The upgrade task is marked successful.
- If the returned software version is different from the target version and this version supports upgrades, the platform continues the upgrade.

6–7: The platform delivers the **package download URL**, token, and package information. The user downloads the software package using HTTPS based on the package download URL and token. The token is valid for 24 hours. (The timeout interval for package download and upgrade status reporting is 24 hours.)

8. The device upgrades the firmware. After the upgrade is complete, the device returns the upgrade result to the platform. (If the version number returned after the device upgrade is the same as the configured version number, the upgrade is successful.)

9. The platform notifies the IoTDA console or application of the upgrade result.

Firmware Upgrade for Devices Using MQTT



The firmware upgrade process for a device using MQTT is as follows:

1–2: A user uploads a firmware package on the IoTDA console and creates a firmware upgrade task on the console or an application.

3. The platform checks whether the device is online and triggers the upgrade negotiation process immediately when the device is online. If the device is offline, the platform waits for the device to go online and **subscribes to the upgrade topic**. After detecting that the device goes online, the platform triggers the upgrade negotiation process. (The timeout interval for the device to go online is within 25 hours.)

4–5: The platform delivers a command to query the device firmware version and determines whether an upgrade is required based on the target version. (The timeout interval for step 5 is 3 minutes.)

- If the returned firmware version is the same as the target version, no upgrade is required. The upgrade task is marked successful.
- If the returned firmware version is different from the target version and this version supports upgrades, the platform continues the upgrade.

6–7: The platform delivers the **package download URL**, token, and package information. The user downloads the software package using HTTPS based on the package download URL and token. The token is valid for 24 hours. (The timeout interval for package download and upgrade status reporting is 24 hours.)

8. The device upgrades the firmware. After the upgrade is complete, the device returns the upgrade result to the platform. (If the version number returned after the device upgrade is the same as the configured version number, the upgrade is successful.)

9. The platform notifies the IoTDA console or application of the upgrade result.

NOTE

The platform supports resumable download.

Firmware Upgrade Failure Causes

The following table lists the failure causes reported by the platform.

Error Message	Description	Solution
Device Abnormal is not online	The device is offline or abnormal.	Check the device.
Task Conflict	A task conflict occurs.	Check whether a software upgrade, firmware upgrade, log collection, or device restart task is in progress.
Waiting for the device online timeout	The device does not go online within the specified time.	Check the device.
Wait for the device to report upgrade result timeout	The device does not report the upgrade result within the specified time.	Check the device.
Waiting for report device firmware version timeout	The device does not report the firmware version within the specified time.	Check the device.
Waiting for report cellId timeout	The device does not report the cell ID within the specified time.	Check the device.
Updating timeout and query device version for check timeout	The device does not report the upgrade result or device version within the specified time.	Check the device.

Error Message	Description	Solution
Waiting for device downloaded package timeout	The device does not finish downloading the firmware package within the specified time.	Check the device.
Waiting for device start to update timeout	The device does not start the update within the specified time.	Check the device.
Waiting for device start download package timeout	The device does not start to download the firmware package within the specified time.	Check the device.

The following table lists the failure causes reported by devices.

Error Message	Description	Solution
Not enough storage for the new firmware package	The storage space is insufficient for the firmware package.	Check the storage space of the device.
Out of memory during downloading process	The memory was insufficient during the download.	Check the device memory.
Connection lost during downloading process	The connection was interrupted during the download.	Check the device connection status.
Integrity check failure for new downloaded package	The integrity check on the firmware package fails.	Check whether the firmware package downloaded is complete.
Unsupported package type	The firmware package type is not supported.	Check whether the device status and firmware package provided by the manufacturer are correct.
Invalid URI	The URI is invalid.	Check whether the download address of the firmware package is correct.
Firmware update failed	The firmware fails to update.	Check the device.

FAQs

The following lists the frequently asked questions about software and firmware upgrades. For more questions, see **OTA Upgrades**.

- **Can the Target Version Be Earlier Than the Source Version?**
- **How Do I Obtain Software or Firmware Packages and Their Version Numbers?**
- **Are Services Interrupted During a Software or Firmware Upgrade?**
- **What Are Common Software or Firmware Upgrade Errors?**

APIs

- **Create a Batch Task**
- **Query the Batch Task List**
- **Query a Batch Task**

6.7.4 OTA Upgrade for a Batch of Devices

Uploading a Software/Firmware Package

You need to upload a software/firmware upgrade package before creating a batch software/firmware upgrade task. The platform supports the following upload modes:

1. Use the application to call the API for **creating an OTA upgrade package**.
2. On the console, choose **Software/Firmware Upgrades**, and upload a software/firmware upgrade package. For details, see **Software/Firmware Package Upload**.

   NOTE

   - The OTA upgrade package uploaded using the API can be used only for upgrading MQTT devices.
   - If the upgrade package is an OBS object, the delivered upgrade package link is the OBS link no matter whether the CDN domain name acceleration is configured for the OBS bucket.

Upgrading the Software for a Batch of Devices

There are two ways to upgrade the software for a batch of devices:

1. Use the application to call the API for **creating a batch task** to create an upgrade task for a batch of devices.
2. Create a software upgrade task on the IoTDA console.

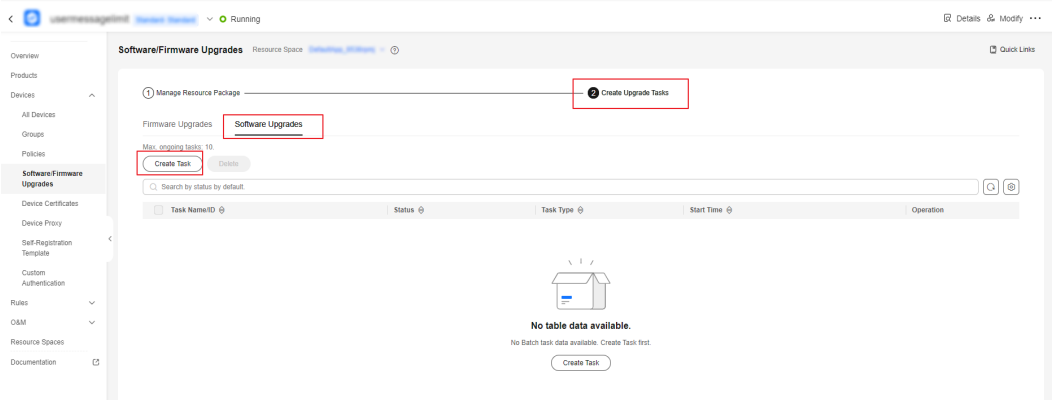
The following describes how to create a software upgrade task for a batch of devices on the console.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **Software/Firmware Upgrades**, and click **Create Upgrade Task**.

Step 3 On the **Software Upgrades** tab page, click **Create Task**.

Figure 6-43 Software/Firmware upgrade - Creating a software upgrade task



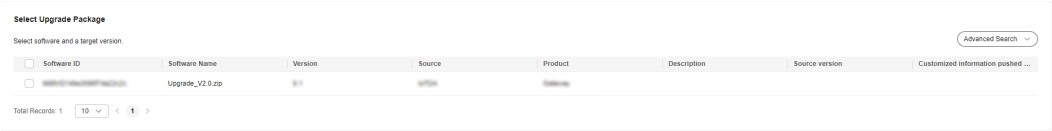
Step 4 In the **Set Basic Information** pane, set the task name, execution time, and retry policy.

If **Retry** is enabled, you can set the number of retry attempts and retry interval. You are advised to set **Retry Attempts** to **2** and **Retry Interval (min)** to **5**. If an upgrade fails, the upgrade will be retried 5 minutes later.

Figure 6-44 Creating a software upgrade task - Basic information

Step 5 Select a software package.

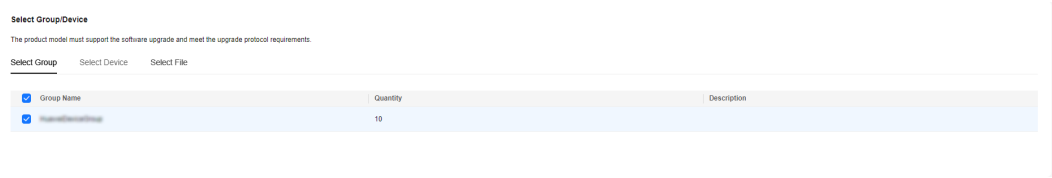
Figure 6-45 Creating a software upgrade task - Selecting an upgrade package



Step 6 Select the device or device group to upgrade and click **Create Now**.

For details on how to create a group and add devices to the group, see **Groups and Tags**.

Figure 6-46 Creating a software upgrade task - Selecting a device group



Step 7 View the result on the task list. Click **View** to check the result for each device on the execution details page.

NOTE

An upgrade task that is being executed cannot be deleted. To delete an upgrade task, manually stop the task first.

----End

Upgrading the Firmware for a Batch of Devices

There are two ways to upgrade the firmware for a batch of devices:

1. Use the application to call the API for **creating a batch task** to create an upgrade task for a batch of devices.
2. Create a firmware upgrade task on the IoTDA console.

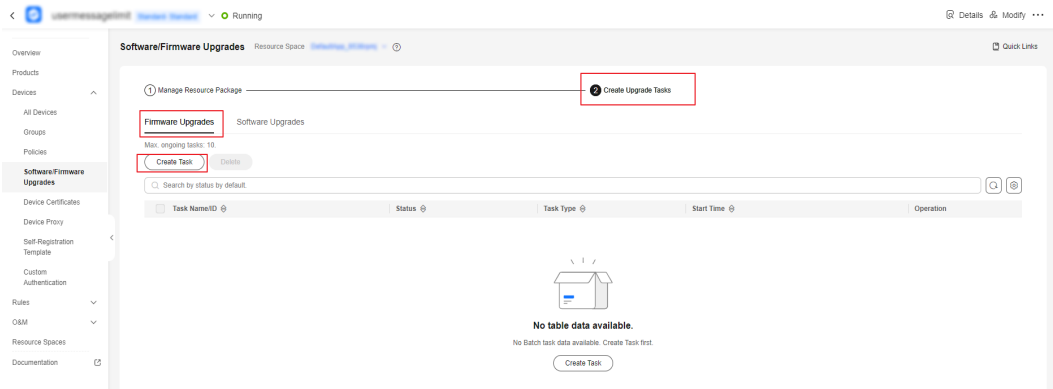
The following describes how to create a firmware upgrade task for a batch of devices on the console.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **Software/Firmware Upgrades**, and click **Create Upgrade Task**.

Step 3 On the **Firmware Upgrades** tab page, click **Create Task**.

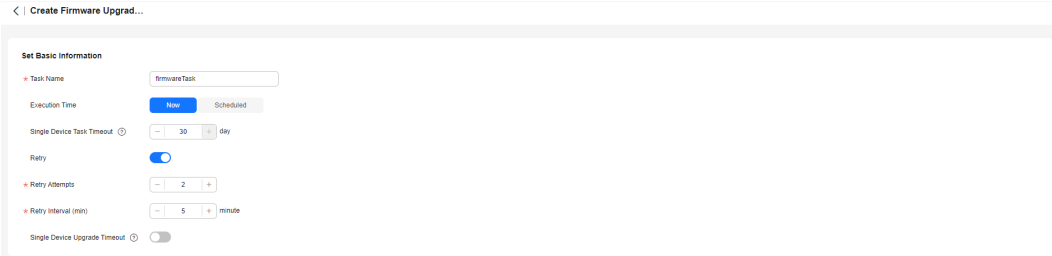
Figure 6-47 Software/Firmware upgrade - Creating a firmware upgrade task



Step 4 In the **Set Basic Information** pane, set the task name, execution time, and retry policy.

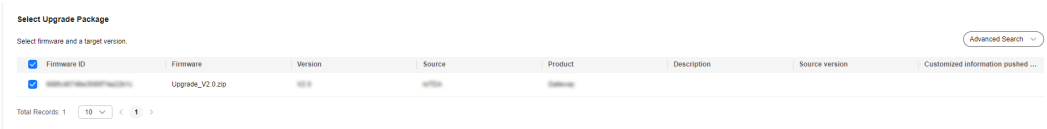
If **Retry** is enabled, you can set the number of retry attempts and retry interval. You are advised to set **Retry Attempts** to **2** and **Retry Interval (min)** to **5**. That is, if the upgrade fails, the upgrade will be retried in 5 minutes. (The maximum number of retry attempts is 5 and the maximum retry interval is 1,440 minutes.)

Figure 6-48 Creating a firmware upgrade task - Basic information



Step 5 Select a firmware package.

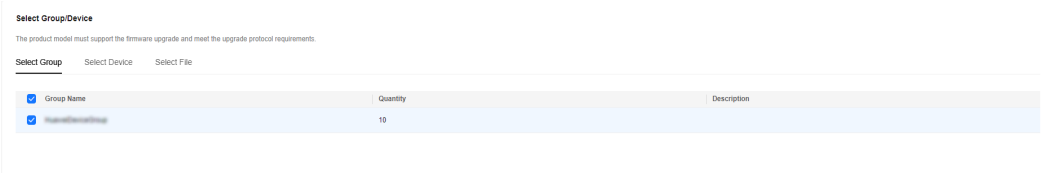
Figure 6-49 Creating a firmware upgrade task - Selecting an upgrade package



Step 6 Select the device group to upgrade and click **Create Now**.

For details on how to create a group and add devices to the group, see **Groups and Tags**.

Figure 6-50 Creating a firmware upgrade task - Selecting a device group



Step 7 View the result on the task list. Click **View** to check the result for each device on the **Execution Details** page.

NOTE

An upgrade task that is being executed cannot be deleted. To delete an upgrade task, manually stop the task first.

----End

Troubleshooting Software/Firmware Upgrade Failure

The following table lists the failure causes reported by the platform.

Error Message	Description	Solution
Device Abnormal is not online	The device is offline or abnormal.	Check the device.
Task Conflict	A task conflict occurs.	Check whether a software upgrade, firmware upgrade, log collection, or device restart task is in progress.

Error Message	Description	Solution
Waiting for the device online timeout	The device does not go online within the specified time.	Check the device.
Wait for the device to report upgrade result timeout	The device does not report the upgrade result within the specified time.	Check the device.
Waiting for report device firmware version timeout	The device does not report the firmware version within the specified time.	Check the device.
Waiting for report cellId timeout	The device does not report the cell ID within the specified time.	Check the device.
Updating timeout and query device version for check timeout	The device does not report the upgrade result or device version within the specified time.	Check the device.
Waiting for device downloaded package timeout	The device does not finish downloading the firmware package within the specified time.	Check the device.
Waiting for device start to update timeout	The device does not start the update within the specified time.	Check the device.
Waiting for device start download package timeout	The device does not start to download the firmware package within the specified time.	Check the device.

The following table lists the failure causes reported by devices.

Error Message	Description	Solution
Not enough storage for the new firmware package	The storage space is insufficient for the firmware package.	Check the storage space of the device.

Error Message	Description	Solution
Out of memory during downloading process	The memory was insufficient during the download.	Check the device memory.
Connection lost during downloading process	The connection was interrupted during the download.	Check the device connection status.
Integrity check failure for new downloaded package	The integrity check on the firmware package fails.	Check whether the firmware package downloaded is complete.
Unsupported package type	The firmware package type is not supported.	Check whether the device status and firmware package provided by the manufacturer are correct.
Invalid URI	The URI is invalid.	Check whether the download address of the firmware package is correct.
Firmware update failed	The firmware fails to update.	Check the device.

FAQs

The following lists the frequently asked questions about software and firmware upgrades. For more questions, see **OTA Upgrades**.

- **Can the Target Version Be Earlier Than the Source Version?**
- **How Do I Obtain Software or Firmware Packages and Their Version Numbers?**
- **Are Services Interrupted During a Software or Firmware Upgrade?**
- **What Are Common Software or Firmware Upgrade Errors?**

APIs

- **Create a Batch Task**
- **Query the Batch Task List**
- **Query a Batch Task**

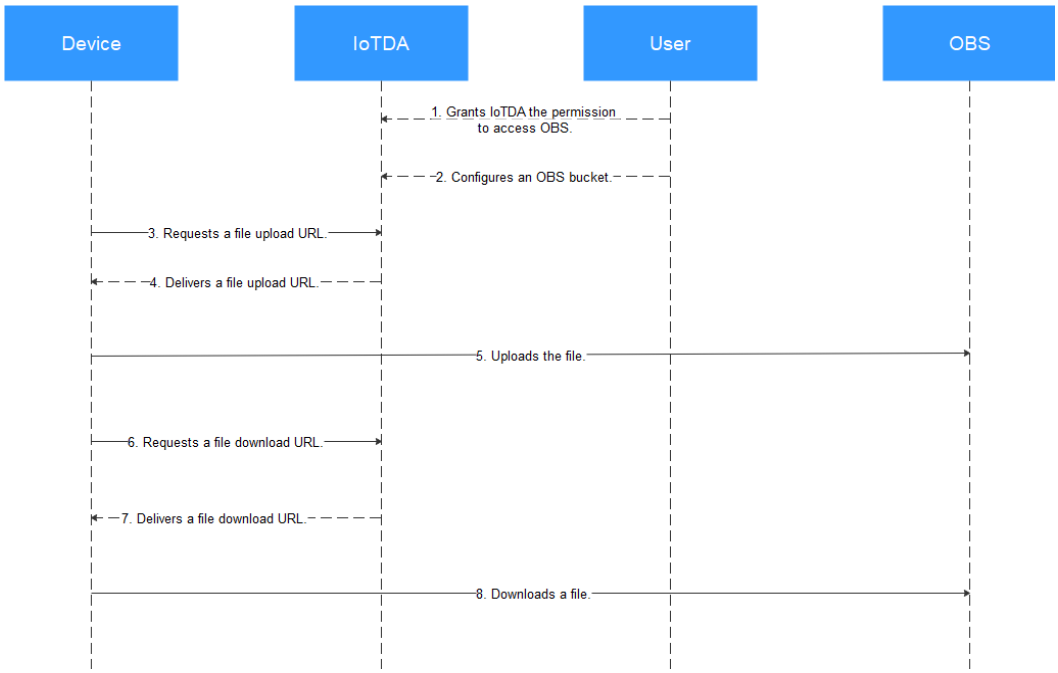
6.8 File Upload

Overview

Devices can upload run logs, configuration files, and other files to the platform for log analysis, fault locating, and device data backup. When a device uploads files to Object Storage Service (OBS) using HTTPS, you can manage the uploaded files on OBS.

Service Flow

Figure 6-51 File upload process



1. A user grants IoTDA the permission to access OBS.

2. The user configures an OBS bucket.

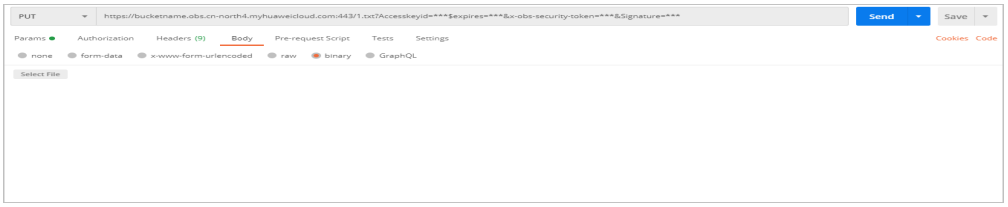
3–4. A device requests a file upload URL, and the platform delivers a URL. For details on the URL format, see **Device Requesting a URL for File Upload**.

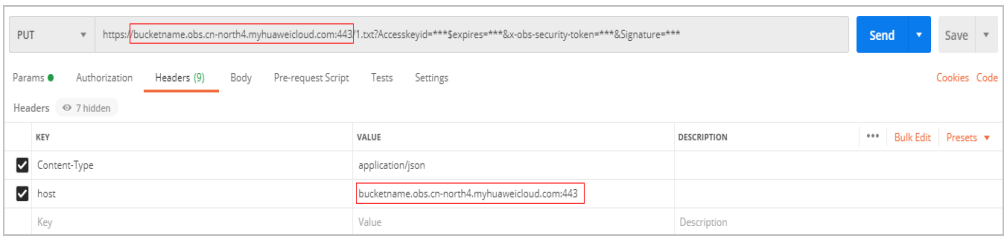
5. The device calls the OBS API and uses the URL delivered by the platform to upload a device file. The validity period of the URL is subject to the value of **expire** (in seconds) delivered by the platform. The default validity period is 1 hour.

- Method 1: Directly use the URL. Postman is used as an example.

  Use the PUT method to call the URL, set the body to binary, and select the file to upload. The file name must be the same as the reported file name.

  The header of the API does not need to contain **Content-Type** or **Host**. If carried, **Content-Type** must be set to **text/plain** and **Host** must be set to the domain name of the URL. Otherwise, the 403 status code **SignatureDoesNotMatch** is returned.





- Method 2: Integrate the OBS SDK to call the API.

  Follow the instructions provided in **Accessing OBS Using a Temporary URL** to use a PUT request to upload an object's SDK to upload the file.

6–7. When the device requests to download a file stored in OBS, the platform delivers a file download URL. For details on the URL format, see **Platform Delivering a Temporary URL for File Upload**.

8. The device calls the OBS API and uses the URL delivered by the platform to download the file.

- Method 1: Use the GET method to call the URL. The header of the API does not need to contain **Content-Type** or **Host**. If carried, **Content-Type** must be set to **text/plain** and **Host** must be set to the domain name of the URL. Otherwise, the 403 status code **SignatureDoesNotMatch** is returned.
- Method 2: Integrate the OBS SDK to call the API and use the GET request to download the object's SDK to download the file.

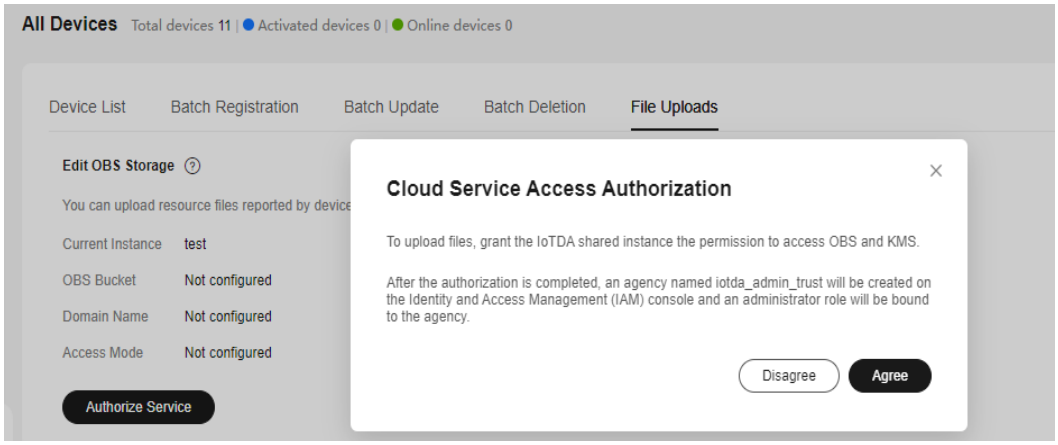
Configuring File Upload

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **All Devices**. On the displayed page, click **File Uploads**.

Step 3 Click **Authorize Service**. In the dialog box displayed, click **Agree**.

Figure 6-52 File uploading - Authorization



Note: If you have only granted IoTDA the permissions to access OBS, choose **Devices** > **All Devices** in the navigation pane, click the **File Uploads** tab, and click **Authorize KMS** to grant IoTDA the permissions to access Key Management Service (KMS).

Step 4 (Optional) Create a bucket on the OBS console if no bucket is available.

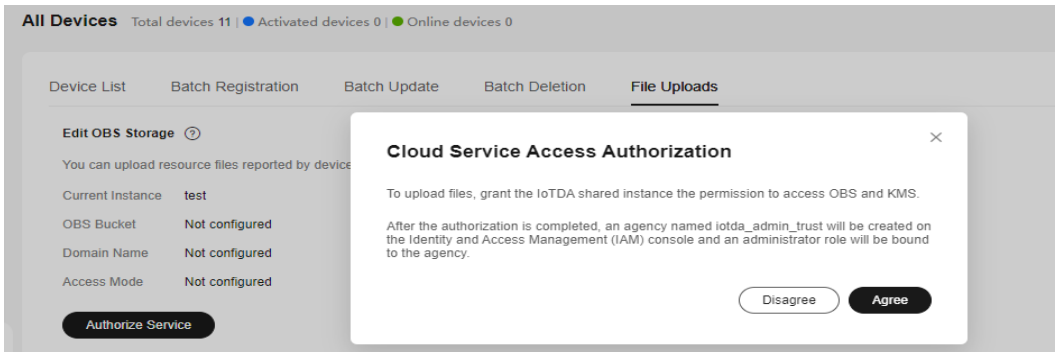
1. Log in to the OBS console.
2. Click **Create Bucket** in the upper right corner to **create a bucket**.

NOTE

If you use OBS to manage files, you will be charged by OBS. IoTDA does not charge you for file storage. For details about OBS billing, see **Billing**.

Step 5 Click **Edit OBS Storage** and select a bucket. All device files in the instance will be uploaded to this bucket. You can click **Edit** to select another bucket.

Figure 6-53 File uploading - Storage configuration

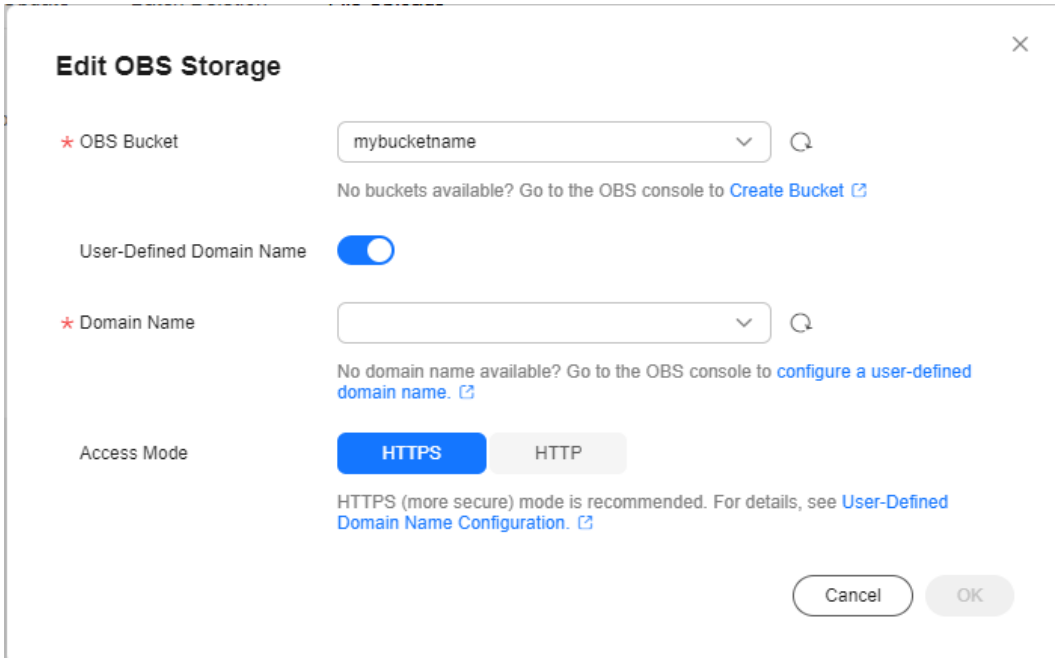


NOTE

When you call the OBS API used for uploading device files, only one file can be uploaded at a time, and the file size cannot exceed 5 GB.

Step 6 If you want to use a custom domain name, enable **User-Defined Domain Name**, select the required domain name configured for the OBS bucket, select **HTTPS** or **HTTP** for **Access Mode**, and click **OK**.

Figure 6-54 File uploading - Configuring a custom domain name



NOTE

The custom domain name is the temporary URL domain name delivered by the platform to the device for OBS file uploading or downloading.

----End

6.9 Gateways and Child Devices

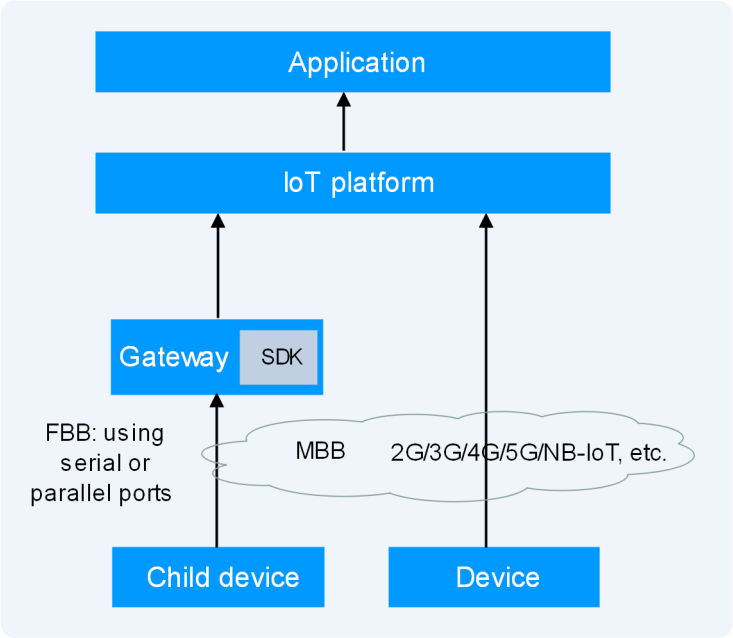
Overview

IoT devices can connect to IoTDA in two modes.

- Directly connected devices: Devices directly connect to the platform using specified protocols.
- Indirectly connected devices: Devices that do not support the TCP/IP protocol stack cannot directly communicate with the platform and need to use gateways as media for data forwarding. Devices directly connected to the platform through MQTT can be used as gateways.

The following figure shows the relationship between directly connected devices and indirectly connected devices.

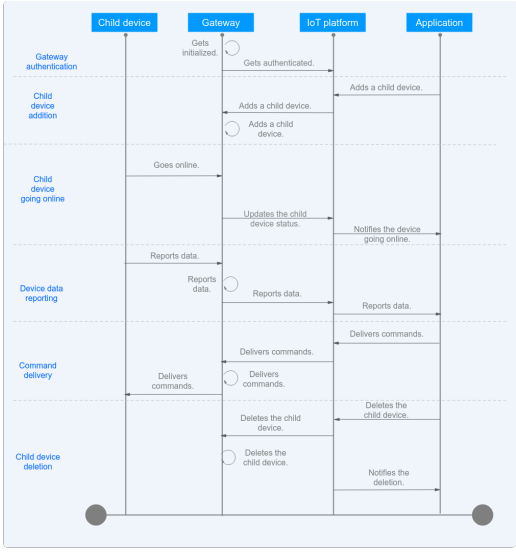
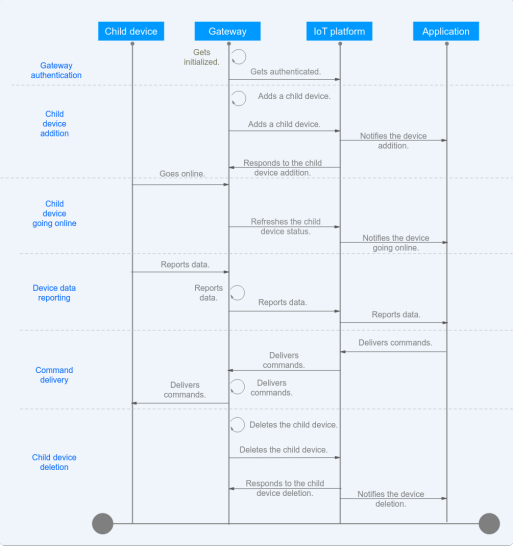
Figure 6-55 Gateways and child devices



Service Flow

You can use the APIs provided by IoT device SDKs to connect gateways and child devices to the platform. API names of SDKs vary depending on the language. For details, see **IoT Device SDK (Java)**, **IoT Device SDK (C)**, **IoT Device SDK (C#)**, **IoT Device SDK (Android)**, and **IoT Device SDK Tiny (C)**.

Table 6-8 Service Flow

Child Device Management Process at the Application Side	Child Device Management Process at the Gateway Side
Figure 6-56 Child device management process at the application side  	**Figure 6-57** Child device management process at the gateway side 
1. A user uploads the product model of a gateway to the platform and registers the gateway.	
2. The gateway calls the authentication API to go online.	
3. The user uploads the product model of a child device to the platform.	
4. After the gateway authentication is successful, an application calls the API for **creating a device**. (The device information entered in the API request must be consistent with that defined in the product model). After the child device is added, the user can view it on the console. For details, see **Viewing a Child Device**. The user can also add child devices on the console. For details, see **Adding a Child Device on the Platform**.	4. After the gateway authentication is successful, the gateway calls the API described in **Platform Notifying a Gateway of New Child Device Connection**. (The device information entered in the API request must be consistent with that defined in the product model). After the processing is complete, the platform sends the processing result to the gateway through the API described in **Platform Responding to a Request for Adding Child Devices**.

Child Device Management Process at the Application Side	Child Device Management Process at the Gateway Side
5. The status of the newly added child device is still displayed as **Inactive** on the console. This is because the gateway has not reported the latest status of the child device to the platform. Call the API described in **Gateway Updating Child Device Status** after the child device is added or before the child device reports data. **NOTE** The status of a child device indicates whether the child device is connected to the gateway, and the gateway reports the status to the platform for status updates. If the gateway cannot report the status of a child device, the child device status is not updated on the platform. For example, after a child device connects to the platform through a gateway, the child device status is displayed as online. If the gateway is disconnected from the platform, the gateway can no longer report the child device status and the platform will consider the child device online.	
6. The gateway calls the API described in **Gateway Reporting Device Properties in Batches** to report the data of the child device. The parameters in the API request are the information about the gateway and the child device.	
7. The gateway subscribes to a topic for command delivery, and receives and processes commands delivered by the application or platform.	
8. The application calls the API for **deleting a device** to command the gateway to delete the child device. The gateway deletes the device upon receiving the command.	8. The gateway calls the API described in **Gateway Requesting for Deleting Child Devices**. After receiving the request, the platform processes the data and sends the result to the device through the API described in **Platform Responding to a Request for Deleting Child Devices**.

Connecting a Gateway to the Platform

Connect a gateway to the platform by integrating the gateway with the SDK. For details, see **Indirectly Connecting to the Platform**.

Adding a Child Device on the Platform

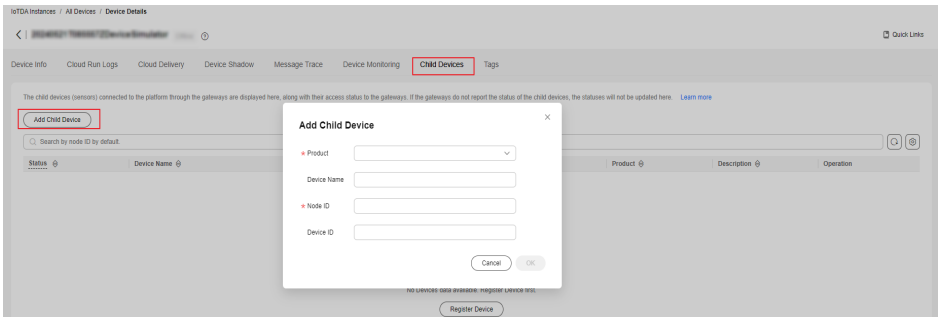
- **Method 1**

  After the gateway is connected to the platform, call the API **Creating a Device** to connect the child device to the platform.

- **Method 2**

  Access the **IoTDA** service page and click **Access Console**. Click the target instance card. In the navigation pane, choose **Devices** > **All Devices**. On the device list, click a gateway to access its details page. On the **Child Devices** tab page, click **Add Child Device**.

Figure 6-58 Device - Adding a child device



Viewing a Child Device

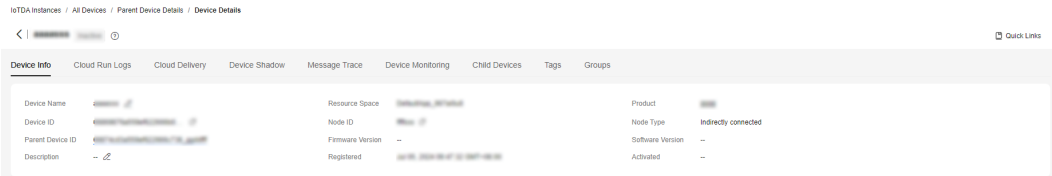
Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **All Devices**. In the device list, click **View** in the row of a gateway to access its details.

Step 3 On the **Child Devices** tab page, view the status, device ID, and node ID of the child devices connected to the platform through the gateway.

Step 4 Click **View** in the row of a child device to view its **details**.

Figure 6-59 Device - Child device details



----End

6.10 Authentication Credentials

Introduction

When connecting to IoTDA, a device must carry a credential for authentication. Currently, two types of authentication credentials are available.

- Secret: the device secret you preset on IoTDA during device registration for future authentication. After successful authentication, the device is activated and communicates with the platform. There are two types of secrets:
  - Master secret: primary secret used for device access authentication.
  - Sub secret: secondary secret used when the master secret fails to pass the authentication. Unavailable for devices accessed using CoAP.
- X.509 certificate: a digital certificate used for communication entity authentication. IoTDA allows devices to use their own X.509 certificates. For details, see **Connecting a Device That Uses the X.509 Certificate Based on MQTT.fx**. In this mode, the platform verifies the device certificate fingerprints

that you preset on the platform for authentication during connection establishment. There are two types of fingerprints:

- Master fingerprint: primary fingerprint used for device access certificate authentication.
- Sub fingerprint: secondary fingerprint used when the master fingerprint fails to pass the authentication. Unavailable for devices accessed using CoAP.

Updating

You need to update a device access credential in some scenarios, for example, when an X.509 certificate is about to expire. You can reset device credentials by calling the APIs for **resetting a device fingerprint** or **resetting a device secret**. IoTDA provides master/sub fingerprints and secrets to prevent device authentication failure and service interruption during credential update and resetting. For example, when you add a new certificate fingerprint, the platform sets the fingerprint as a backup (sub fingerprint). In this way, the corresponding device can use both the old certificate (if not updated on the device side timely) and the new certificate to connect to the platform smoothly.

Scenarios

1. In high availability (HA) scenarios, a device has two secrets to connect to the platform.

2. During the credential update and resetting, a device does not disconnect from the platform and causes no service losses.

Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **All Devices**. By default, all devices in the current instance are displayed in the device list.

Step 3 Click **View** in the **Operation** column of the target device. On the displayed page, click **Reset Secret** or **Reset Fingerprint**. In the displayed dialog box, select **Sub secret** or **Sub fingerprint**.

Figure 6-60 Device details - Resetting sub secret

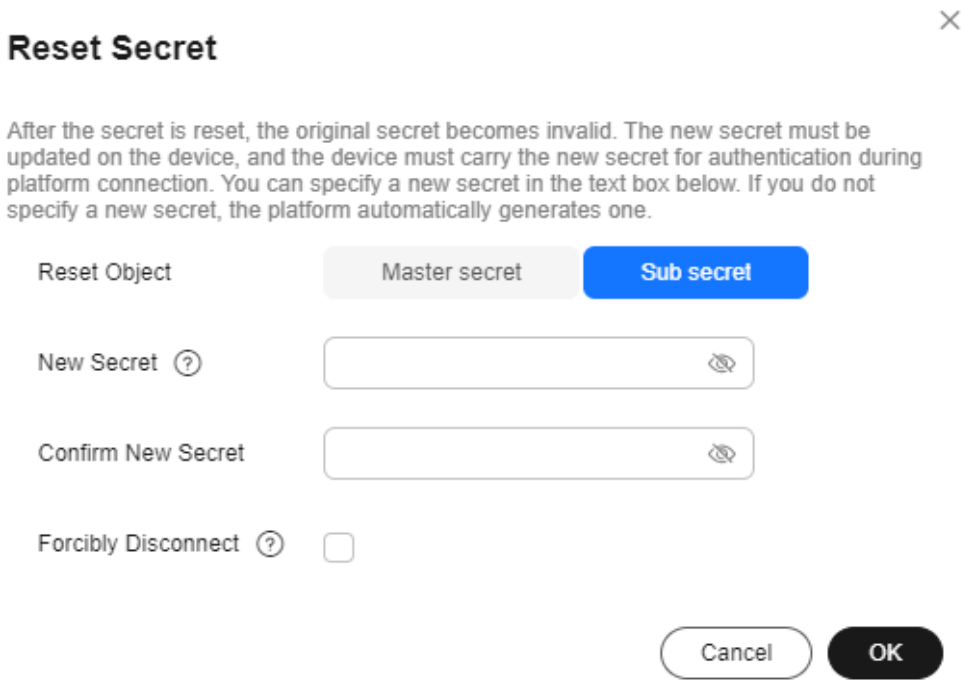
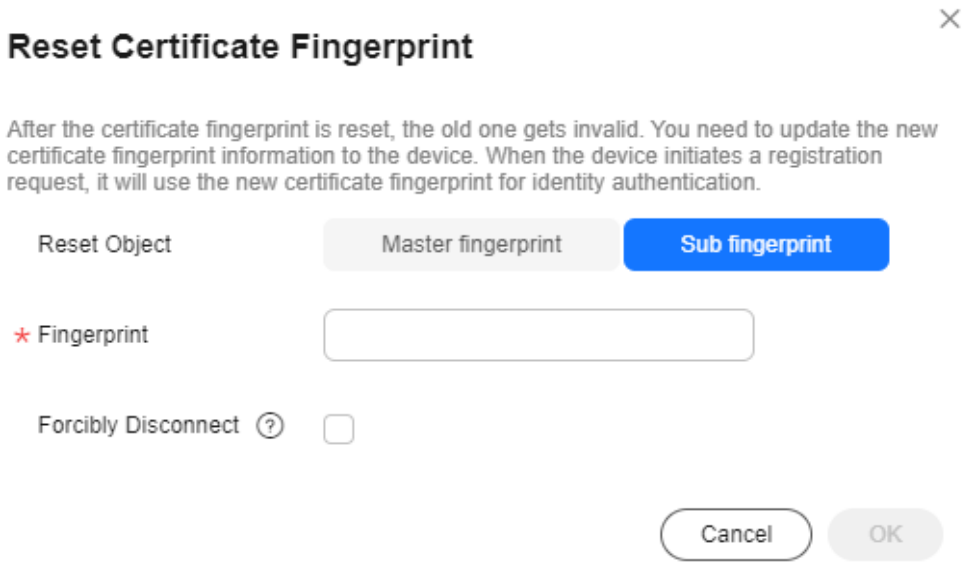


Figure 6-61 Device details - Resetting sub fingerprint



----End

APIs

- **Reset a Device Fingerprint**
- **Reset a Device Secret**

6.11 Device Certificates

Overview

For a device using MQTTS X.509 certificates, when it connects to the platform for the first time, IoTDA utilizes the uploaded and verified device CA certificate to authenticate the device certificate. After the authentication is successful, the platform automatically saves the device certificate and generates expiration warnings. You can check and disable device certificates.

Constraints

1. IoTDA generates alarms for device certificates that are about to expire within 30 days. Update the certificates in a timely manner to prevent access failures.
2. The device certificate quota provided by IoTDA is 1.5 times of the device quantity quota. To ensure smooth storage of new certificates, delete expired certificates in a timely manner. Failure to do so may result in the inability to check certificates on the console. However, this does not affect device access to the platform.
3. A device is associated with a device certificate through the certificate fingerprint. After a device certificate is disabled, all devices associated with it cannot access the platform.

Procedure

Step 1 Upload and verify the device CA certificate. For details about how to create and verify a device CA certificate, see **Registering a Device Authenticated by an X.509 Certificate**.

Step 2 Register a device that uses the X.509 certificate for authentication. Log in to the IoTDA console. In the navigation pane, choose **Devices** > **All Devices**. On the displayed page, click **Register Device**. Set **Authentication Type** to **X.509 certificate** and **Fingerprint** to the SHA-256 fingerprint of the device certificate. If you do not specify the fingerprint, the device certificate fingerprint carried when the device successfully accesses the platform for the first time is recorded by default.

Step 3 Use the device certificate to access the device to the platform.

Figure 6-62 Device connection parameters

Step 4 Log in to the IoTDA console. In the navigation pane, choose **Devices** > **Device Certificates**. Click the **Device Certificates** tab to check the device certificate list and details.

Figure 6-63 Device certificate - Certificate list

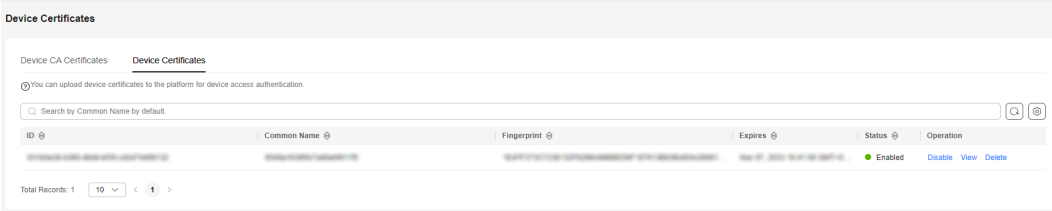
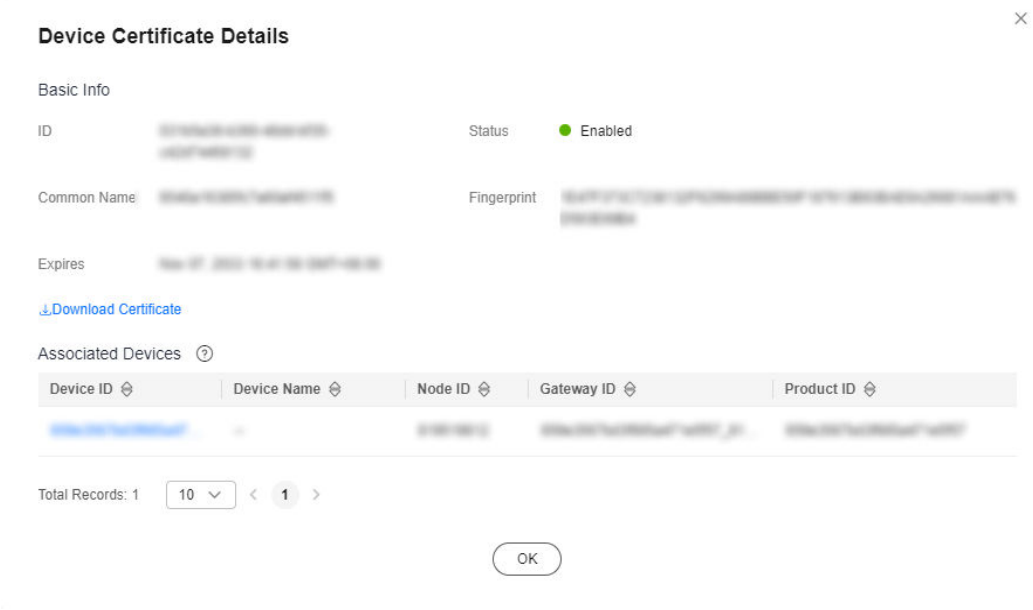
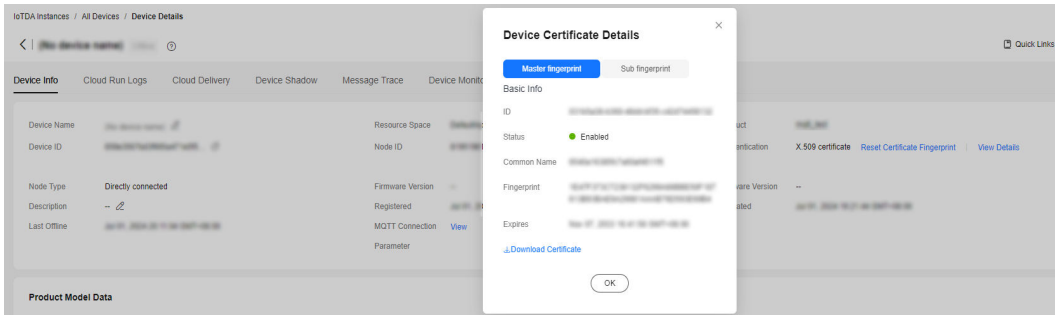


Figure 6-64 Device certificate - Certificate details



Step 5 In the navigation pane, choose **Devices** > **All Devices**. Locate the target device, click **View** in the **Operation** column. On the displayed page, click the button for checking the certificate details.

Figure 6-65 Device - Device details - Certificate details

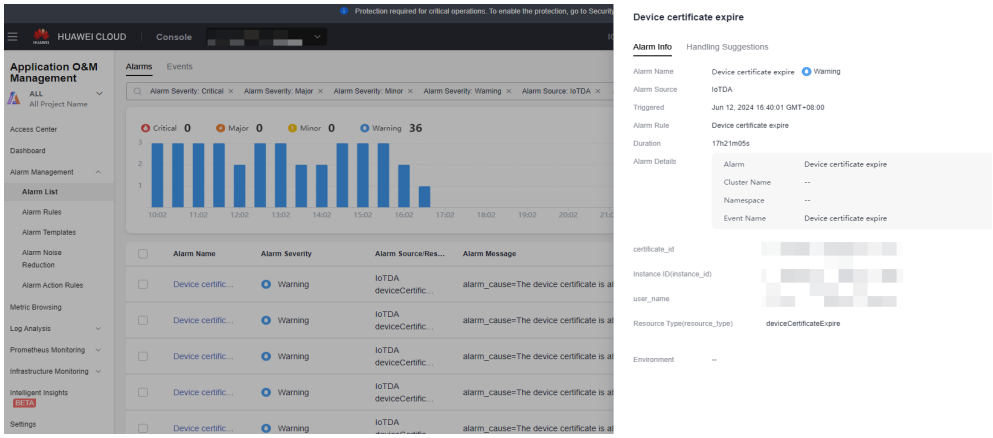


----End

Device Certificate Alarms

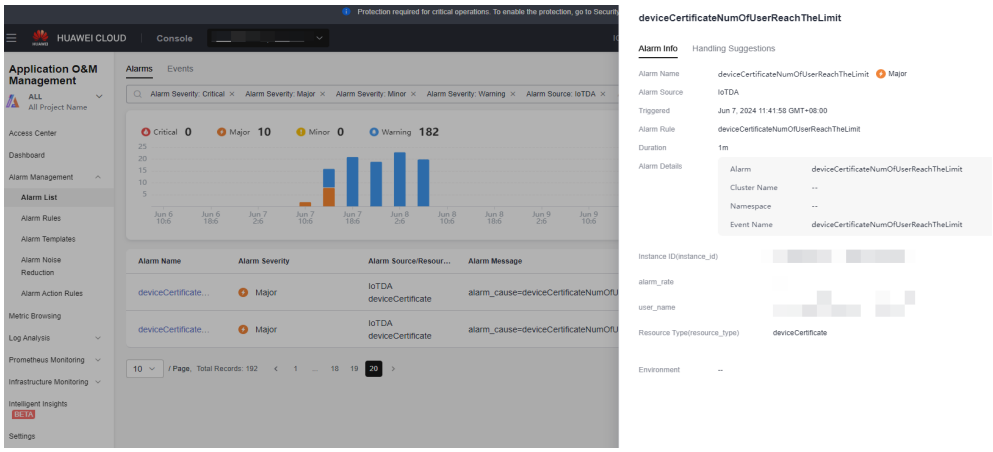
- IoTDA generates an expiration warning for device certificates. You can check the certificates that are about to expire in the recent month in the Application Operations Management (AOM) alarm list.

Figure 6-66 Device certificate expiration alarm -AOM



- IoTDA generates a warning when the number of device certificates exceeds the threshold. Delete expired certificates in a timely manner.

Figure 6-67 Insufficient device certificate quota -AOM



AOM allows you to create an alarm action rule to send alarm notifications through SMSs, emails, and WeCom messages. For details, see **Creating an Alarm Action Rule**.

7 Rules

7.1 Overview

You can set rules for devices connected to the platform. If the conditions set in a rule are met, the platform triggers the corresponding action. Device linkage and data forwarding rules are available.

- **Device linkage**

  When specific conditions are met, the platform triggers collaborative response of multiple devices to implement device linkage and intelligent control. Currently, IoTDA supports cloud rules and device rules. If you want to create a cloud rule and select **Send notifications** for **Action** in a rule, the platform will work with Huawei Cloud **Simple Message Notification (SMN)** to set and deliver topic notifications. If you want to create a device rule, the platform will deliver the rule to devices for unified management and execution.

- **Data forwarding**

  IoT Device Access (IoTDA) can seamlessly forward data to other Huawei Cloud services and third-party applications, providing full-stack services for device data storage, computing, and analysis.

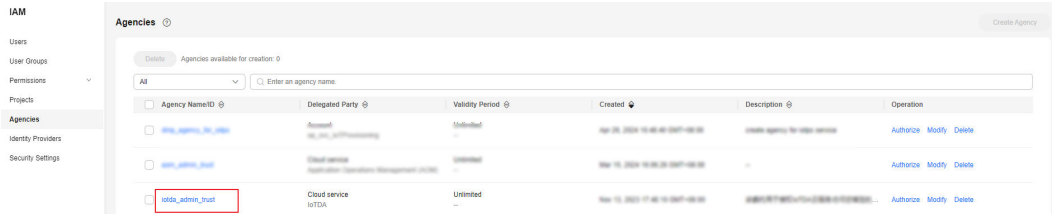
Cloud Service Access Authorization

The platform can connect to Huawei Cloud services. When creating a rule for connecting to Data Ingestion Service (DIS), Distributed Message Service (DMS) for Kafka, Object Storage Service (OBS), ROMA Connect, or SMN for the first time, you must authorize the platform to access the cloud service.

After the authorization, data on the platform can be forwarded to other Huawei Cloud services using data forwarding rules, or the platform can send commands to control devices using device linkage rules.

An agency named **iotda_admin_trust** is created on the **Identity and Access Management (IAM)** console and an administrator role is bound by default.

Figure 7-1 Agency - iotda_admin_trust



7.2 Data Forwarding Process

Overview

The data forwarding function connects IoTDA with other Huawei Cloud or third-party cloud services to smoothly transfer device data to the message middleware, storage and data analysis services, and applications. Currently, IoTDA supports multiple forwarding types.

Table 7-1 Data forwarding types

Type	Forwarding Target	Description	Operation
Third-party services	Third-party application (HTTP push)	Data is transferred to customers' HTTP servers with URL specified in the data forwarding rule.	**HTTP/HTTPS Data Forwarding**
	AMQP message queue	Data is transferred to the AMQP channels specified in the data forwarding rule for client-platform connection and data exchange.	**AMQP Data Forwarding**
	MQTT message queue	Data is transferred to the MQTT topics specified in the data forwarding rule for client-platform connection and data exchange.	**MQTT Data Forwarding**
	M2M communication	IoTDA supports MQTT-based message communication between devices based on topics specified in the data forwarding rule. The platform pushes messages reported by devices to the specified topics. Other devices can receive messages from different devices by subscribing to the specified topics.	**M2M Communications**

Type	Forwarding Target	Description	Operation
Data storage	**GeminiDB Influx**	Data is transferred to GeminiDB Influx, a cloud-native time series database compatible with InfluxDB. GeminiDB Influx reads and writes time series data with high performance and compression ratio in high concurrency scenarios, provides cold and hot tiered storage, elastic scale-out, and monitoring and alarm reporting. It stores the data with compression algorithms, allows you to query data using SQL-like statements, and supports multi-dimensional aggregation computing and visual analysis. Scenarios: It is widely used to monitor resources, services, IoT devices, and industrial production processes, evaluate production quality, and trace faults. With high throughput and concurrency, it can handle a large number of connections in a very short period of time, making it an excellent choice for IoT applications. Learn more about **InfluxDB instance specifications**.	**Forwarding Data to GeminiDB Influx**
	RDS for MySQL	Data is transferred to RDS for MySQL. Compared with self-managed databases, this service is cheaper, out-of-the-box, and easy to operate and maintain. It supports auto scaling and provides functions such as instance management and monitoring, backup and restoration, log management, and parameter management. Standalone and primary/standby deployment modes are available. Scenarios: website, gaming, e-commerce, and financial services, and mobile and enterprise applications Learn more about **RDS for MySQL instance specifications**.	**Forwarding Data to MySQL for Storage**

Type	Forwarding Target	Description	Operation
	Object Storage Service (OBS)	Data is transferred to OBS. OBS provides customers with massive, secure, reliable, and cost-effective data storage capabilities and multiple storage types. OBS can work with **Cloud Stream Service (CS)** to analyze stream data in real time. The analysis results can be used for data visualization in other cloud services or third-party applications. Scenarios: massive big data storage and analysis Learn more about **OBS storage specifications**.	**Forwarding Device Data to OBS for Long-Term Storage**

NOTICE

Maximum data forwarding rate through the public network: 1 Mbit/s. The excess messages will be discarded.

Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Rules** > **Data Forwarding**, and click **Create Rule** in the upper left corner.

Step 3 Configure related parameters and click **Create Rule**.

Table 7-2 Parameters for creating a rule

Parameter	Description
Rule Name	Name of the rule to be created.
Description	Description of the rule.

Parameter	Description
Data Source	• **Device**: Device information, such as device addition, deletion, and update, will be forwarded. When **Data Source** is set to **Device**, quick configuration is not supported. • **Device property**: A property value reported by a device in a resource space will be forwarded. Click **Quick Configuration** on the right and select the product, property, and service data to forward. • **Device message**: A message reported by a device in a resource space will be forwarded. Click **Quick Configuration** on the right and select data of a specified topic to forward. Select the product to which the topic belongs and enter the topic name. You can use a **custom topic** on the product details page or a **preset topic**. • **Device message status**: The status of device messages exchanged between the device and platform will be forwarded. For details on the device message status, see **Message Delivery Status**. When **Data Source** is set to **Device message status**, quick configuration is not supported. • **Device status**: The status change of a directly or an indirectly connected device in a resource space will be forwarded. Click **Quick Configuration** on the right to forward information about devices whose status is **Online**, **Offline**, or **Abnormal** to other services. For details on the status of devices directly connected to the platform, see **Device Management**. • **Batch task**: The batch task status will be forwarded. When **Data Source** is set to **Batch Task**, quick configuration is not supported. • **Product**: Product information, such as product addition, deletion, and update, will be forwarded. When **Data Source** is set to **Product**, quick configuration is not supported. • **Asynchronous command status of the device**: Status changes of asynchronous commands to devices using LwM2M over CoAP will be forwarded. For details on the asynchronous command status of devices, see **Asynchronous Command Delivery**. When **Data Source** is set to **Asynchronous command status of the device**, quick configuration is not supported. • **Run log**: Service run logs of MQTT devices will be forwarded. When **Data Source** is set to **Run log**, quick configuration is not supported.
Trigger	After you select a data source, the platform automatically matches trigger events.
Resource Space	You can select a single resource space or all resource spaces. If **All resource spaces** is selected, quick configuration is not supported.

Step 4 Click the **Set Forwarding Target** tab, and then click **Add** to set a forwarding target.

Data can be forwarded to **Data Ingestion Service (DIS)**, **Distributed Message Service (DMS) for Kafka**, **Object Storage Service (OBS)**, **FunctionGraph**, **Log Tank Service (LTS)**, **GeminiDB Influx**, **RDS for MySQL**, **third-party applications (HTTP push)**, **AMQP message queues**, **MQTT message queues**, and devices.

Table 7-3 Parameters for setting the forwarding target

Forwarding Target	Description
Data Ingestion Service (DIS)	• **Region**: Select the region of the service to which data will be forwarded. If you are not authorized to access the service in this region, perform authorization as required. • **Stream Homing**: You can select either of the following: – **In-house stream**: Select a stream. If no DIS stream is available, **create one** on the DIS console. – **Delegated by others**: You can use DIS streams authorized by other users. Obtain the stream ID from the DIS console.
Distributed Message Service (DMS) for Kafka **NOTE** Data can be forwarded only to Kafka premium instances. You need to enable automatic topic creation.	• **Region**: Select the region of the service to which data will be forwarded. If you are not authorized to access the service in this region, perform authorization as required. • **Connection Address**: Obtain the connection address by following the instructions provided in **Accessing a Kafka Instance with SASL**. Basic and standard instances support only access to Kafka premium instances over the Internet. Enterprise instances support access to Kafka premium instances over a private network. • **Topic**: Customize a topic. • SASL-based authentication: If SASL authentication is enabled, enter the SASL username and password entered in **Buying a Kafka Instance**. • **Kafka Security Protocol**: If SASL authentication is enabled, select the security protocol supported by the Kafka instance you purchased. • **SASL Mechanism**: If SASL authentication is enabled, select the SASL authentication mechanism supported by the Kafka instance you purchased.

Forwarding Target	Description
Object Storage Service (OBS)	• **Region**: Select the region of the service to which data will be forwarded. If you are not authorized to access the service in this region, perform authorization as required. • **OBS Bucket**: Select a bucket as required. If no OBS bucket is available, **create one** on the OBS console. • **Custom Directory**: Separate different directory levels by slashes (/). The directory cannot start or end with a slash (/) or contain two or more consecutive slashes (/).
Third-party application (HTTP push)	You can use HTTP or HTTPS to push messages. For details on how to set parameters, see **HTTP/HTTPS Subscription/Push**.
AMQP message queue	**Message Queue**: Select the queue to which messages are to be pushed. If no queue is available, create one. For details on the restrictions on message queue names, see **AMQP Server Configuration**.
FunctionGraph **NOTE** Currently, only data of instances of Enterprise and Standard editions can be forwarded to FunctionGraph.	• **Function Name**: Select the name (latest version) of the function to be called. Currently, cross-region function calling is not supported. If no function is available, **create one**.

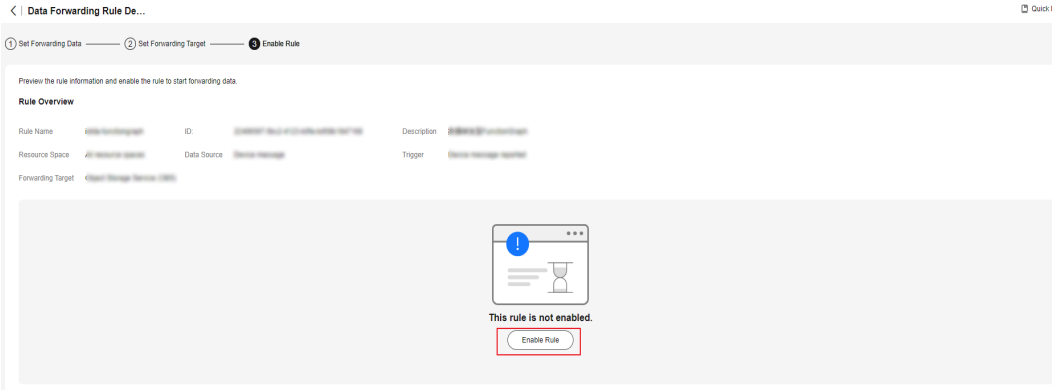
Forwarding Target	Description
GeminiDB Influx **NOTE** Currently, only data of instances of Enterprise and Standard editions can be forwarded to GeminiDB Influx.	• **Database Instance Address**: Enter the address for connecting to the GeminiDB Influx instance. IoTDA enterprise edition instances can connect to GeminiDB Influx using private network IP addresses, while standard edition instances support only public network connection. For details, see **Connection Methods**. • **Database Name**: Enter a database name. If no database exists, go to the GeminiDB Influx console to **create a database**. • **Access Account** and **Access Password**: Access the GeminiDB Influx console to obtain the account and password. For details, see **Resetting the Administrator Password**. • **Table**: Enter the name of the target table (measurement). If the table does not exist, it will be automatically created. • **Field Mappings**: – **Forwarding Field**: Enter the attribute name of the data to be forwarded. Data to be forwarded is in JSON format. Separate multi-level attribute names with periods (.). For details about the format of data to be forwarded, see **Data Transfer APIs**. – **Target Field**: Enter the column name of the database.

Forwarding Target	Description
RDS for MySQL **NOTE** Currently, only data of instances of Enterprise and Standard editions can be forwarded to RDS for MySQL.	• **Database Instance Address**: Enter the address for connecting to the RDS instance. IoTDA enterprise edition instances can connect to RDS using private network IP addresses, while standard edition instances support only public network connection. For details, see **Connection Management**. • **Database Name**: Enter a database name. If no database exists, go to the RDS for MySQL console to **create a database**. • **Access Account** and **Access Password**: Access the RDS console to obtain the account and password. For details, see **Resetting a Password for a Database Account**. • **SSL**: Select whether to connect to the database in SSL mode. You are advised to use the SSL mode. If not, security risks may exist during data transmission. To use the SSL mode, **configure an SSL connection** in the database instance first. • **Table**: Select the name of the table to which data is forwarded. • **Field Mappings**: – **Forwarding Field**: Enter the attribute name of the data to be forwarded. Data to be forwarded is in JSON format. Separate multi-level attribute names with periods (.). For details about the format of data to be forwarded, see **Data Transfer APIs**. – **Target Field**: Enter the column name of the database.
MQTT message queue	Push Topic: Select the topic to which the message is to be pushed.
Device	MQTT is used to implement message communications between devices. For details about the parameters, see **Usage**.

Step 5 Start a rule.

After the rule is configured, click the button for enabling the rule to start data forwarding.

Figure 7-2 Data forwarding - Enabling a rule



Step 6 IoTDA provides connectivity testing of the rule action forwarding target. For details, see **Connectivity Tests**.

----End

7.3 SQL Statements

When creating a data forwarding rule, you must compile SQL statements to parse and process JSON data reported by devices. For details about the JSON data format, see **Data Transfer APIs**. This section describes how to compile SQL statements used in data forwarding rules.

SQL Statements

An SQL statement consists of the SELECT and WHERE clauses. Each clause can contain a maximum of 500 characters. Chinese and other character sets are not supported. Contents in the SELECT and WHERE clauses are case-sensitive. However, keywords such as SELECT, WHERE, and AS are case-insensitive.

The following example uses messages reported by a device as the source data.

```
{
  "resource" : "device.message",
  "event" : "report",
  "event_time" : "20151212T121212Z",
  "notify_data" : {
    "header" : {
      "device_id" : "********",
      "product_id" : "ABC123456789",
      "app_id" : "********",
      "gateway_id" : "********",
      "node_id" : "ABC123456789",
      "tags" : [ {
        "tag_value" : "testTagValue",
        "tag_key" : "testTagName"
      } ]
    },
    "body" : {
      "topic" : "topic",
      "content" : {
          "temperature" : 40,
          "humidity" : 24
      }
    }
  }
}
```

In the source data, **content** in the **body** is the data reported by the device. You can set **temperature** greater than 38 as the trigger condition, and filter out other fields to obtain only **device_id** and **content**. The example SQL statement is as follows:

```
SELECT notify_data.header.device_id AS device_id, notify_data.body.content WHERE
notify_data.body.content.temperature > 38
```

When the temperature reported by the device is higher than 38°C, data forwarding is triggered. The data format after forwarding is as follows:

```
{
   "device_id": "********",
   "notify_data.body.content" : {
         "temperature" : 40,
         "humidity" : 24
    }
}
```

SELECT Clause

The SELECT clause consists of **SELECT** followed by multiple SELECT subexpressions, which can be *, JSON variables, string constants, or integer constants. A JSON variable is followed by an AS keyword and an AS variable, 32 characters in total. If a constant or function is used, you must use AS to specify the name.

- JSON variable

  A JSON variable can contain letters, digits, underscores (_), and hyphens (-). To distinguish a hyphen (-) from the minus sign, use double quotation marks (") to enclose the JSON variable with a hyphen, for example, **"msg-type"**.

  The JSON variable extracts data of the nested structure.

  ```
  {
    "a":"b",
    "c": {
      "d" : "e"
    }
  }
  ```

  c.d can be used to extract character string **e**, which can be nested at multiple layers.

- AS variable

  An AS variable consists of letters and is case sensitive. The variable [a-zA-Z_-]* is supported. If a hyphen (-) is used, enclose it with double quotation marks (").

- Constant integer

  SELECT supports constant integers, which must be followed by an AS clause. For example:

  NOTE

  Value range of the constant integer: –2147483648 to 2147483647

  ```
  SELECT 5 AS number
  ```

- Constant character string

  SELECT supports constant character strings, which must meet the [a-zA-Z_-]* expression. The character strings must be enclosed in single quotation marks (') and followed by an AS clause.

NOTE

A string can contain up to 50 characters.

```
SELECT 'constant_info' AS str
```

WHERE Clause

In the WHERE clause, you can perform Boolean operations using JSON variables, make some non-null judgments, and combine the results using AND or OR.

- **IS NULL and IS NOT NULL**

  Null judgment can be used in the WHERE clause. If the JSON variable cannot extract data or the extracted array is empty, **IS NULL** is true. Otherwise, **IS NOT NULL** is true.

  ```
  WHERE data IS NULL
  ```

- **IN and NOT IN**

  The IN operator can be used in the WHERE clause. If the target value is in the specified value set, IN is true. Otherwise, NOT IN is true. The IN operator supports strings and numbers, the IN set only supports constants. The types of all elements in the IN set must be the consistent and be the same as the type of the target value.

  ```
  WHERE notify_data.header.product_id IN ('productId1','productId2')
  ```

- **Operators > <**

  The greater than (>) and less than (<) operators can be used in the WHERE clause. The operators can be used between two JSON variables, between a JSON variable and a constant, or between a constant and a constant only when the value of a JSON variable is a constant integer. The operators can be used together with AND or OR.

  For example:

  WHERE data.number > 5 Obtains the information of the target whose **data.number** is greater than 5.
  WHERE data.tag < 4 Obtains the information of the target whose **data.tag** is less than 4.
  WHERE data.number > 5 AND data.tag < 4 Obtains the information of the target whose **data.number** is greater than 5 and **data.tag** less than 4.

- **Equals sign (=)**

  The equals sign (=) can be used in the WHERE clause for comparison between JSON variables, between JSON variable integers and integer constants, and between JSON variable strings and string constants. If **IS NULL** for the two JSON variables is true, the comparison result of the equals sign (=) is false. The operators can be used together with AND or OR.

  WHERE data.number = 5 Obtains the information of the target whose **data.number** is **5**.
  WHERE data.tag = 4 Obtains the information of the target whose **data.tag** is 4.
  WHERE data.number = 5 OR data.tag = 4 Obtains the information of the target whose **data.number** is 5 or **data.tag** is 4.

Constraints

Table 7-4 Restrictions on using SQL statements

Object	Restriction
SELECT clause	500 characters

Object	Restriction
WHERE clause	500 characters
AS clause	10 AS clauses
JSON data depth	400 levels

Debugging SQL Statements

The IoT platform provides the online SQL debugging function. To debug SQL statements, perform the following operations:

1. After compiling the SQL statements, click **Debug**.
2. On the **Debug Parameters** tab page, enter the data to debug, and click **Start Debugging**.

Function List

Multiple functions are used in rules. You can use these functions when compiling SQL statements to implement diversified data processing.

Table 7-5 Function list

Function Name	Parameter	Function	Return Value Type	Restriction
GET_TAG	String tagKey	Obtains **tag_value** corresponding to a specified **tag_key**. `GET_TAG('testTagName')`	String	-
CONTAINS_TAG	String tagKey	Checks whether the specified **tag_key** is contained. `CONTAINS_TAG('testTagName')`	Boolean	-

Function Name	Parameter	Function	Return Value Type	Restriction
GET_SERVICE	String serviceId and boolean fuzzy	Obtains the service. If **fuzzy** is set to **false** or left blank, the service with the specified service ID is obtained. If **fuzzy** is set to **true**, the service is queried through fuzzy match. If multiple services with the same service ID exist in a message body, the result is not guaranteed. `GET_SERVICE('Battery',true)`	JSON structure	Used only for property reporting
GET_SERVICES	String serviceId and boolean fuzzy	Obtains services. If **fuzzy** is set to **false** or left blank, services with the specified service ID are obtained. If **fuzzy** is set to **true**, services are queried through fuzzy match. The query results are combined into an array. `GET_SERVICES('Battery',true)`	JSON array	Used only for property reporting
CONTAINS_SERVICES	String serviceId and boolean fuzzy	If **fuzzy** is set to **false** or left empty, the system checks whether the specified service ID exists. If **fuzzy** is set to **true**, fuzzy match is used to determine whether service ID in the property contains the specified parameter. `CONTAINS_SERVICES('Battery',true)`	Boolean	Used only for property reporting

Function Name	Parameter	Function	Return Value Type	Restriction
GET_SERVICE_PROPERTIES	String serviceId	Obtains the **properties** field of the service with a specific service ID. `GET_SERVICE_PROPERTIES('Battery')`	JSON structure	Used only for property reporting
GET_SERVICE_PROPERTY	String serviceId, String propertyKey	Obtain the value of **propertyKey** in **properties** of a service with a specific service ID. Example: `GET_SERVICE_PROPERTY('Battery','batteryLevel')`	String	Used only for property reporting

STARTS_WITH	String input, String prefix	Checks whether the value of input starts with **prefix**.
		`STARTS_WITH('abcd','abc')`
		`STARTS_WITH(notify_data.header.device_id,'abc')`
		`STARTS_WITH(notify_data.header.device_id,notify_data.header.product_id)`
			Boolean	-

ENDS_WITH	String input, String suffix	Checks whether the value of input ends with **suffix**.
		`ENDS_WITH('abcd','bcd')`
		`ENDS_WITH(notify_data.header.device_id,'abc')`
		`ENDS_WITH(notify_data.header.device_id,notify_data.header.node_id)`
			Boolean	-

CONCAT	String input1, String input2	Concatenates character strings and returns the results.
		`CONCAT('ab','cd')`
		`CONCAT(notify_data.header.device_id,'abc')`
		`CONCAT(notify_data.header.product_id,notify_data.header.node_id)`
			String	-

REPLACE	String input, String target, String replacement	Replaces a part of a character string. That is, replace **target** in the input with **replacement**. `REPLACE(notify_data.header.node_id,'nodeId','IMEI')`	-	-

Function Name	Parameter	Function	Return Value Type	Restriction

SUBSTRING	String input, int beginIndex, int endIndex(required=false)	Obtains the substring of the returned string. That is, the **beginIndex** (included) to **endIndex** (excluded) characters of the input value.
		Note: **endIndex** is optional.
		`SUBSTRING(notify_data.header.device_id,3)`
		`SUBSTRING(notify_data.header.device_id,3,12)`
			-	-

LOWER	String input	Converts all values in input to lowercase letters. `LOWER(notify_data.header.app_id)`	-	-
UPPER	String input	Converts all values in input to uppercase letters. `UPPER(notify_data.header.app_id)`	-	-

7.4 Connectivity Tests

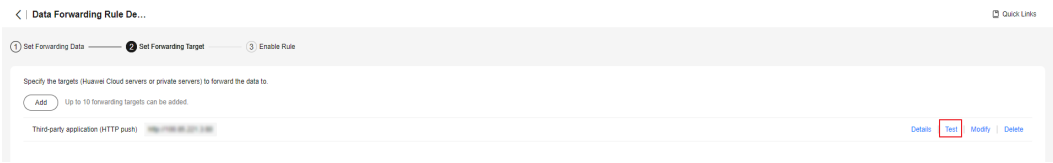
Overview

IoTDA provides connectivity tests on the forwarding targets. In the service debugging phase, you can simulate service data to test the availability of rule actions and the consistency of forwarded data. If a fault occurs in data forwarding during service running, you can perform connectivity tests to reproduce and locate the fault.

Procedure

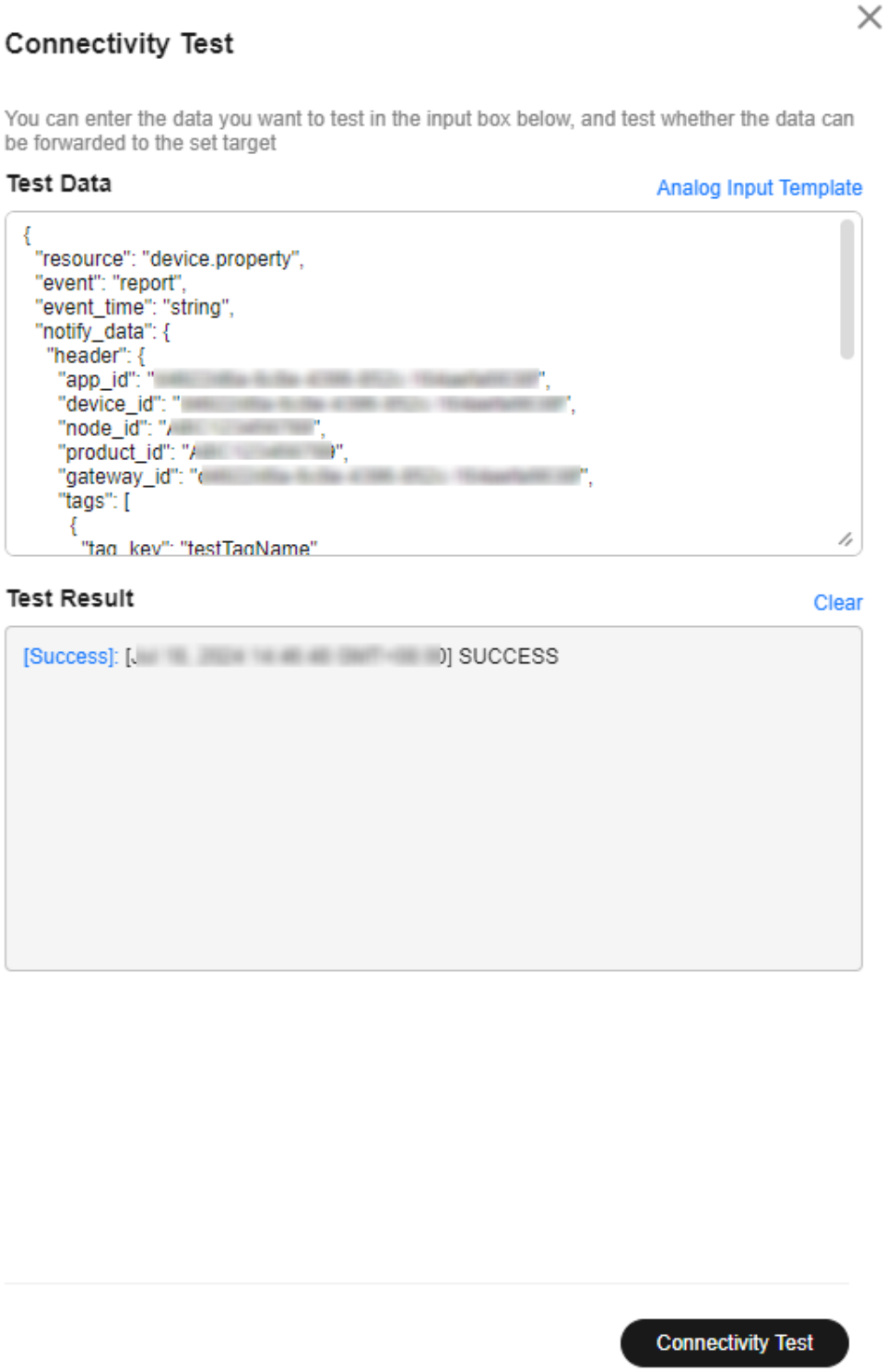
1. After creating a forwarding rule, click **Test** in row of the forwarding target to be debugged.

Figure 7-3 Forwarding target - Test



2. In the **Connectivity Test** dialog box, enter the test data for forwarding in **Test Data**, or click **Analog Input Template** in the upper right corner to use the template data, and then click **Connectivity Test**.

Figure 7-4 Forwarding target - Connectivity test results



7.5 Data Forwarding to Huawei Cloud Services

7.5.1 Forwarding Data to DIS

Scenarios

Forwarding data to Data Ingestion Service (DIS) allows you to collect, process, and distribute real-time streaming data efficiently. DIS interconnects with multiple third-party data collection tools and provides cloud service connectors, agents, and SDKs. You can also dump data to other cloud services for subsequent data processing like data storage and analysis.

Purchasing a DIS Stream (Example: Forwarding Data to DIS and Dumping Data to OBS)

Step 1 Log in to Huawei Cloud and access the **Object Storage Service (OBS)** console.

Step 2 Click **Create Bucket**, configure the parameters as required, and click **Create Now**.

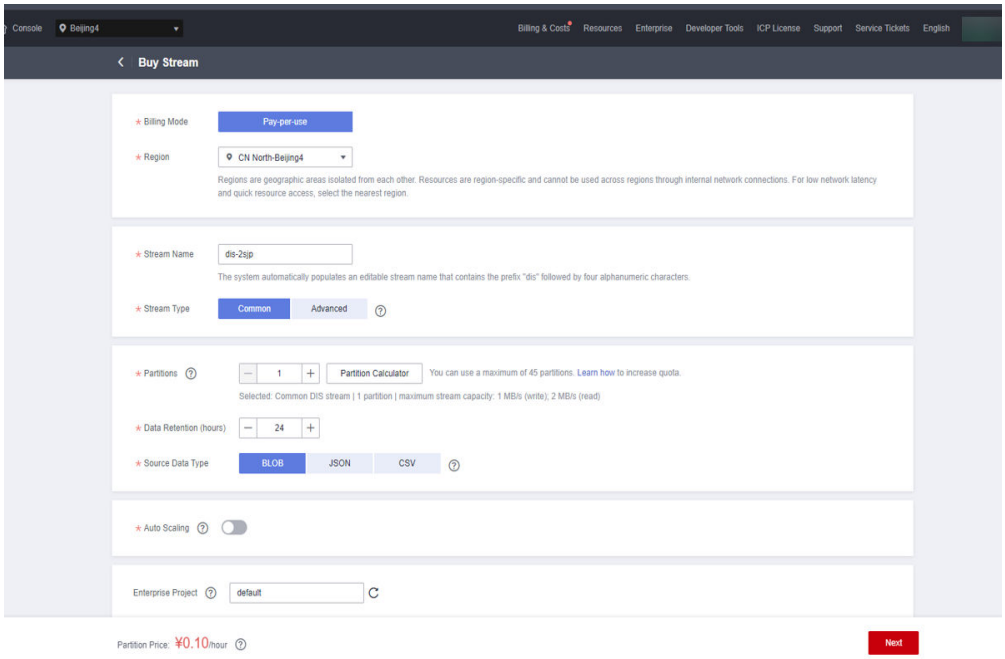
Step 3 In the bucket list, click the created bucket. The **Objects** page is displayed. Click **Create Folder** and enter a folder name as prompted.

Step 4 Log in to Huawei Cloud and access **Data Ingestion Service (DIS)**.

Step 5 Click **Access Console** to go to the DIS console.

Step 6 Click **Buy Stream** in the upper right corner, configure parameters as required, and click **Next**.

Figure 7-5 Buying a DIS stream



Step 7 In the navigation pane, choose **Stream Management** and click a purchased stream. Click the **Dump Tasks** tab. Click **Create Dump Task**. Set **Dump Destination** to **OBS**, **Dump Bucket** to the bucket created in step **2**, and **File Directory** to the folder created in step **3**. Click **Create Now**.

Figure 7-6 Selecting a stream

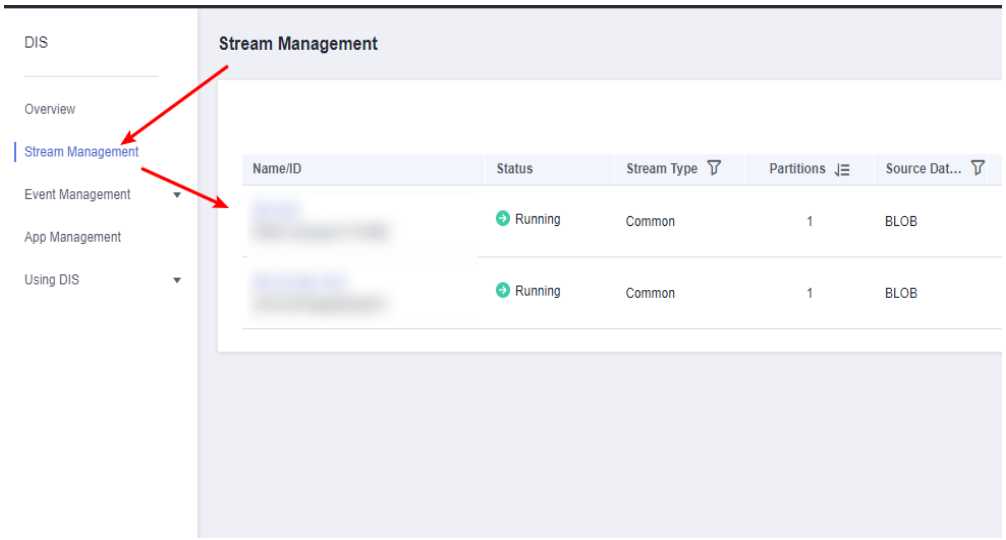


Figure 7-7 Accessing the dump task tab

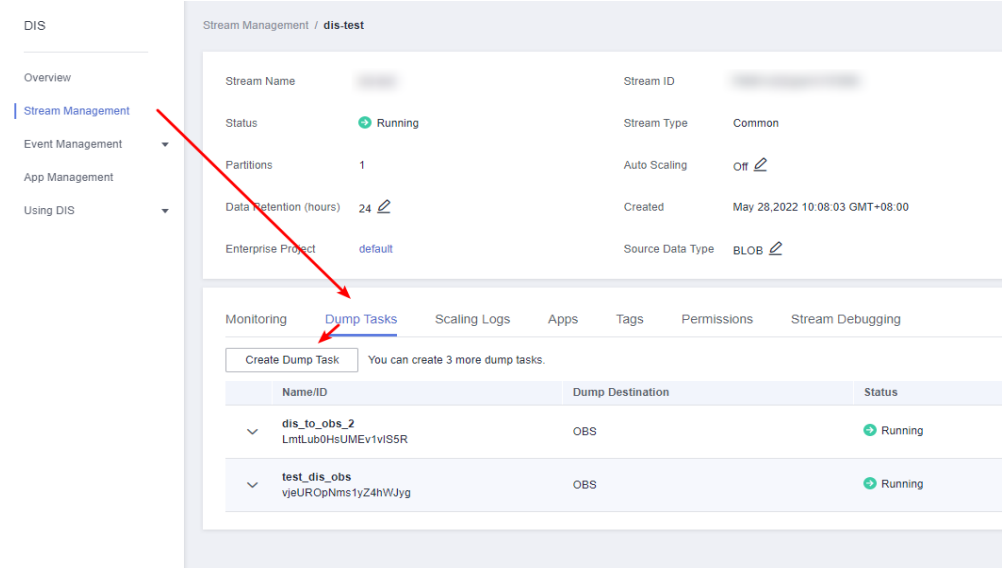
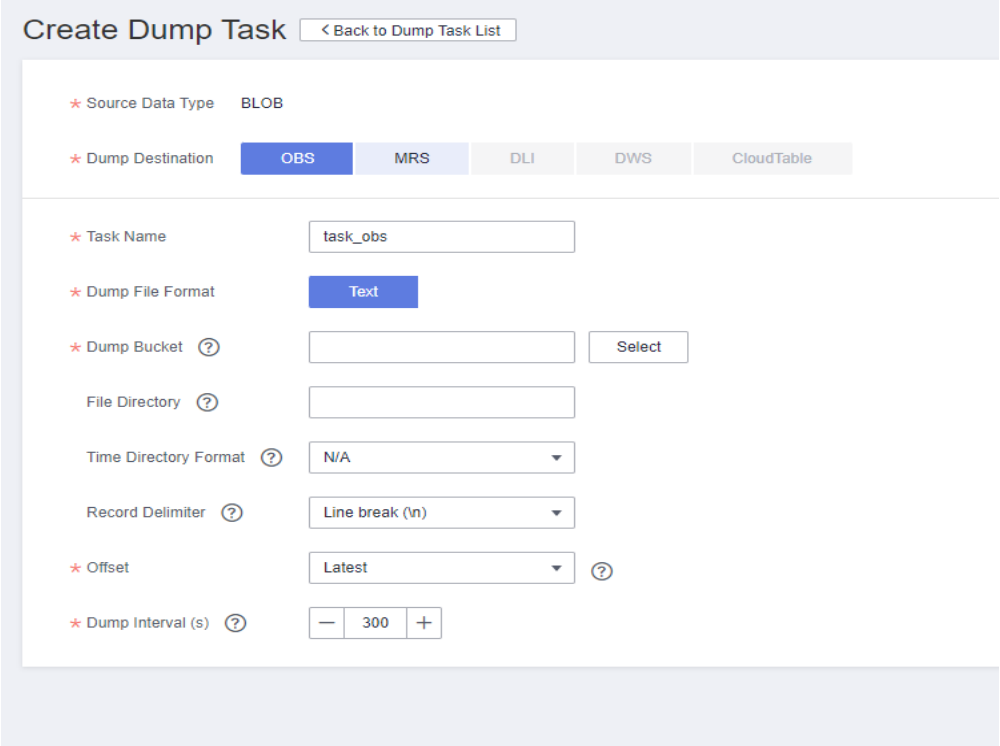


Figure 7-8 Configuring a dump task



----End

Configuring IoTDA

You can configure data forwarding rules in IoTDA to forward data reported by devices to DIS.

Step 1 Go to the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Rules** > **Data Forwarding**, and click **Create Rule** in the upper left corner.

Step 3 Configure the parameters based on the following table. The following parameter values are only examples. You can configure the parameters by referring to **Data Forwarding Overview** and click **Create Rule**.

Table 7-6 Rule parameters

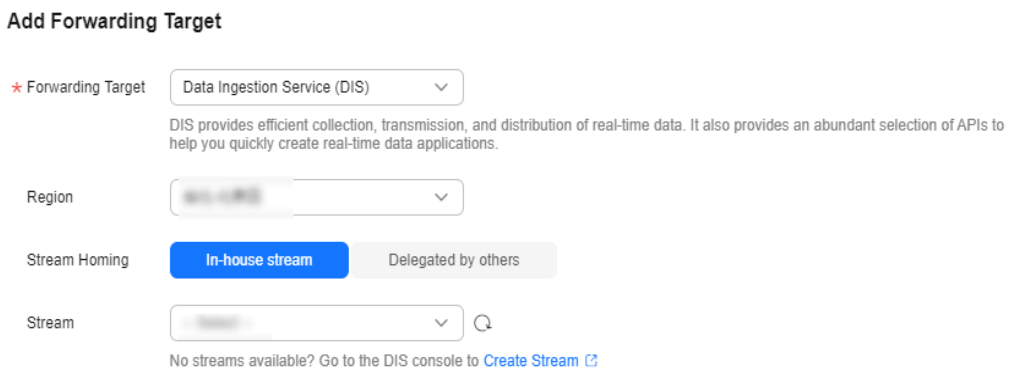
Parameter	Description
Rule Name	Customize a name, for example, **iotda-dis**.
Description	Enter a rule description, for example, **forwarding data to DIS**.
Data Source	Select **Device property**.
Trigger	**Device property reported** is automatically configured.
Resource Space	Select **All resource spaces**.

Step 4 Click the **Set Forwarding Target** tab, click **Add** to set a forwarding target, and click **OK**.

Table 7-7 Forwarding target parameters

Parameter	Description
Forwarding Target	Choose **Data Ingestion Service (DIS)**.
Region	Currently, you can only forward data to DIS in the same region. If IoTDA is not authorized to access the service in this region, configure cloud service access authorization as prompted.
Stream Homing	Select either of the **in-house stream** or the stream **Delegated by others**.
Stream	Select a stream.

Figure 7-9 Creating a forwarding target - to DIS



Step 5 Click **Enable Rule** to activate the configured data forwarding rule.

----End

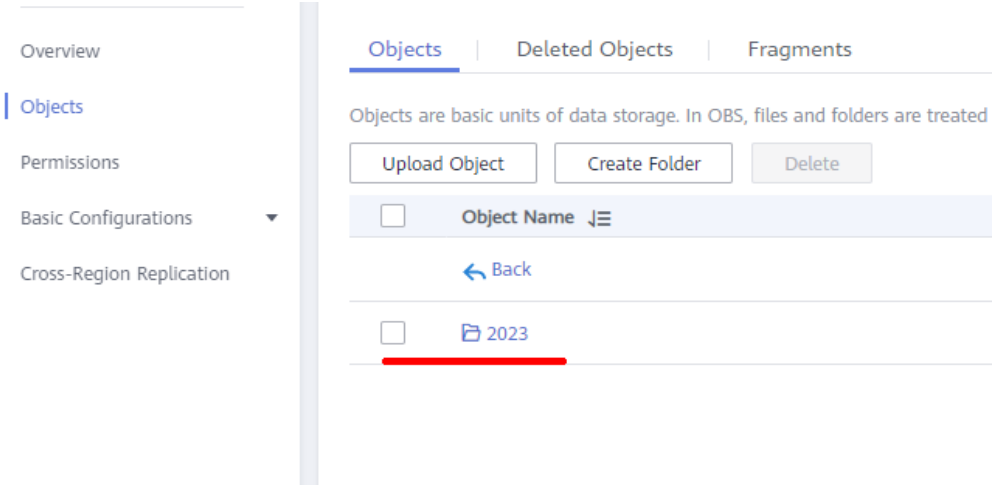
Verifying Configurations

- You can use a registered physical device to access the platform and enable the device to report data.
- You can also use a simulator to simulate a device to report data. For details, see **Developing an MQTT-based Smart Street Light Online**.

Expected result:

Log in to the OBS console, click the bucket created in **2**, and click the folder created in **3** to view the latest data forwarded from DIS to OBS.

Figure 7-10 Viewing OBS data



7.5.2 Forwarding Data to GeminiDB Influx

Scenarios

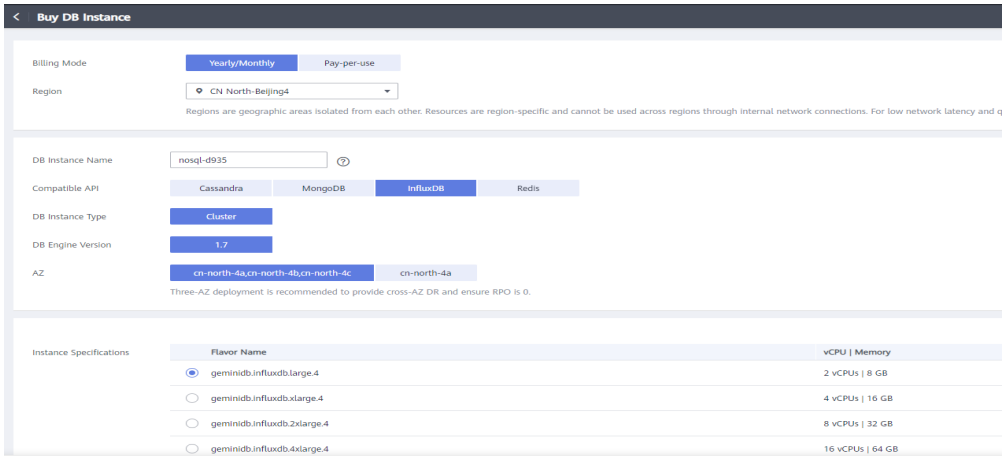
Forward data to GeminiDB Influx and cloud-native time series database with full compatibility with the service. GeminiDB Influx reads and writes time series data with high performance and compression ratio in high concurrency scenarios, provides cold and hot tiered storage, elastic scale-out, and monitoring and alarm reporting. It stores the data with compression algorithms, allows you to query data using SQL-like statements, and supports multi-dimensional aggregation computing and visual analysis. It is widely used to monitor resources, services, IoT devices, and industrial production processes, evaluate production quality, and trace faults. GeminiDB Influx can achieve very high throughput and concurrency, so it can handle a large number of connections in a very short period of time, making it an excellent choice for IoT applications.

Buying GeminiDB Influx Instances

Step 1 Log in to **GeminiDB Influx** and click **Buy Now**.

Step 2 Select either of the **Pay-per-use** or **Yearly/Monthly** as the **Billing Mode**, configure specifications and storage space as required, and set **Compatible API** to **InfluxDB**. For details, see **Buying a Cluster Instance**.

Figure 7-11 Buying an InfluxDB Instance

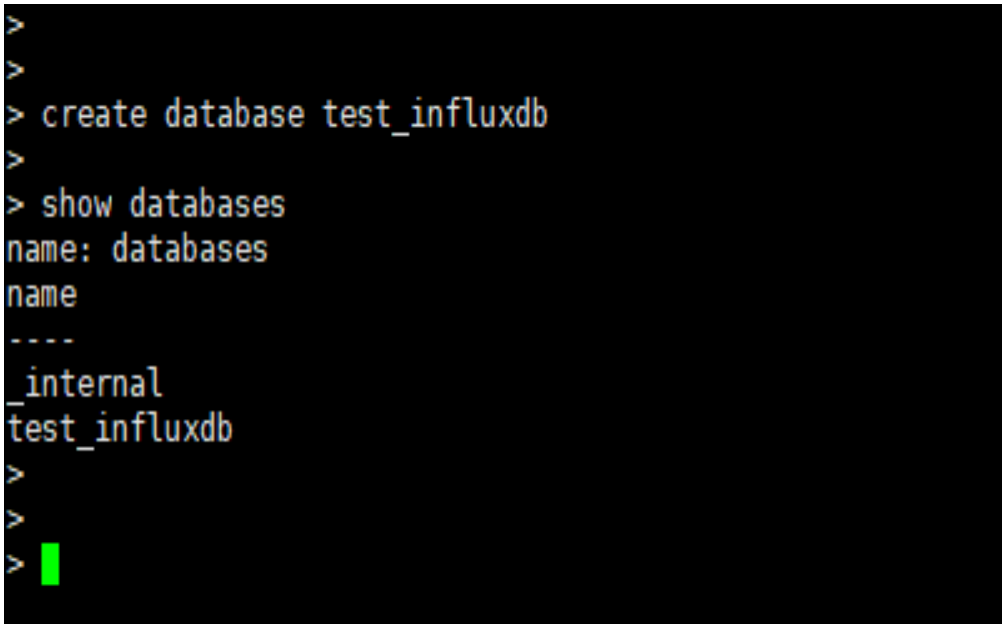


Step 3 Download the InfluxDB client and connect to an instance through the client by referring to **Connecting to an Instance over a Public Network**.

Step 4 After connecting to the instance through the client, run the following command to create a database. **${databaseName}** can be customized.

```
create database ${databaseName}
```

Figure 7-12 Creating a database



----End

Configuring IoTDA

You can configure data forwarding rules in IoTDA to forward data reported by devices to InfluxDB.

Step 1 Go to the **IoTDA** service page and click **Access Console**.

Step 2 In the navigation pane, choose **Rules** > **Data Forwarding**, and click **Create Rule** in the upper left corner.

Step 3 Set the parameters based on the table below. The following parameter values are only examples. You can create a rule by referring to **Data Forwarding Overview** and click **Create Rule**.

Table 7-8 Rule parameters

Parameter	Description
Rule Name	Customize a name, for example, **iotda-InfluxDB**.
Description	Enter a rule description, for example, **forwarding data to InfluxDB**.
Data Source	Select **Device**.
Trigger	**Device added** is automatically configured.
Resource Space	Select **All resource spaces**.

Step 4 Click the **Set Forwarding Target** tab, click **Add** to set a forwarding target, and click **Next**.

Table 7-9 Forwarding target parameters

Parameter	Description
Forwarding Target	Select **GeminiDB Influx**.
Database Instance Address	Enter the connection address of the Influx instance you purchased. IoTDA enterprise edition instances support private network access of Influx instances in the same VPC and subnet.
Database Name	Enter the name of the database created in InfluxDB.
Access Account	Enter the InfluxDB account name.
Access Password	Password of the InfluxDB.
Certificate ID	Truststore certificate, which is used by the client to verify the server certificate. If this parameter is left blank, the default certificate provided by GeminiDB Influx is used. For **GeminiDB Influx instances using private certificates**, upload a custom CA on the **Rules** > **Server Certificates** page and complete binding.

Step 5 Set the target fields and click **OK** to complete configuration.

Table 7-10 Field mapping parameters

Parameter	Description
Save To	Enter the table name, which is user-defined.

Parameter	Description
Field Mappings	Configure the field mappings. You can customize the target field and configure the forwarding field by referring to **Push a Device Addition Notification**.

Figure 7-13 Creating a forwarding target - to InfluxDB

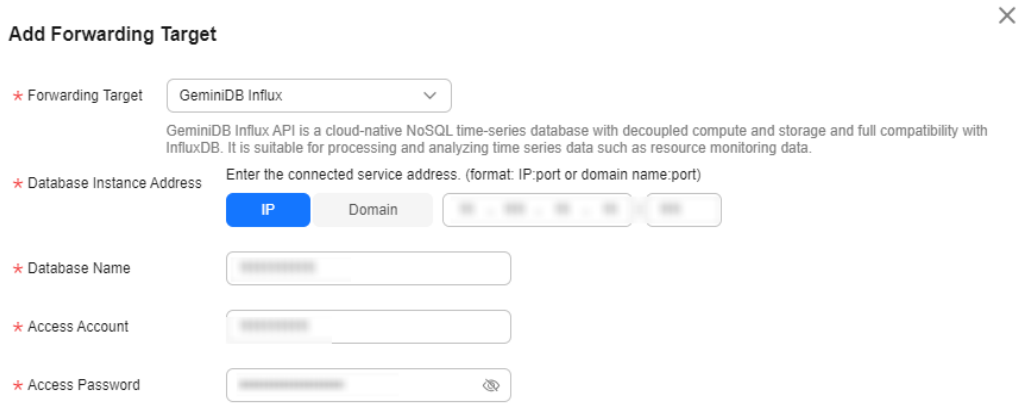
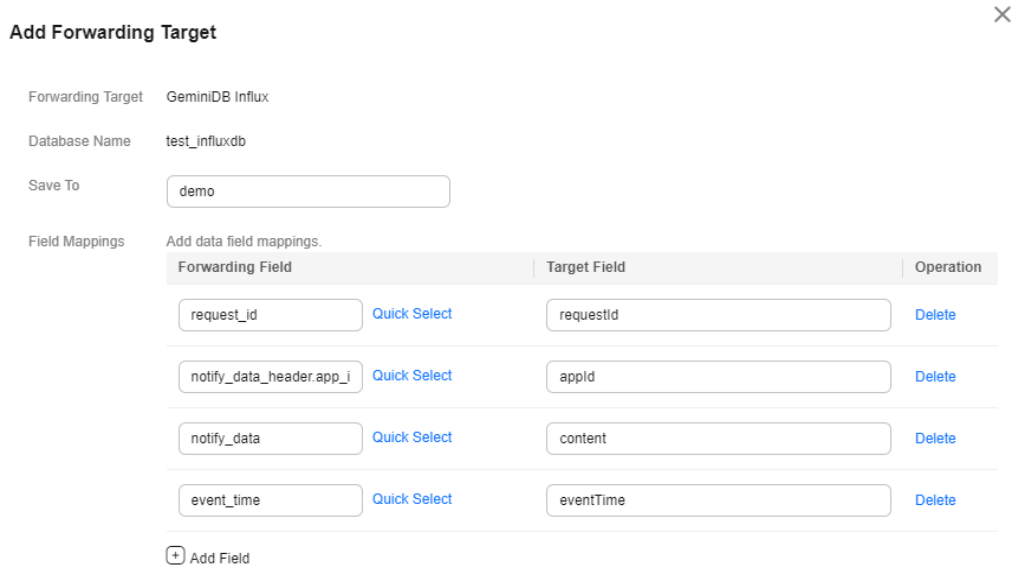


Figure 7-14 Setting InfluxDB field mapping



Step 6 Click **Enable Rule** to activate the configured data forwarding rule.

----End

Verifying Configurations

Log in to the IoTDA console and create a device.

Expected result:

Log in to InfluxDB through the client. Access the database and query data.

```
show databases //Query the database.
use test_influxdb //Switch the database.
select * from demo //Query data.
```

Figure 7-15 Test message

7.5.3 Forwarding Data to DMS for Kafka for Storage

Scenarios

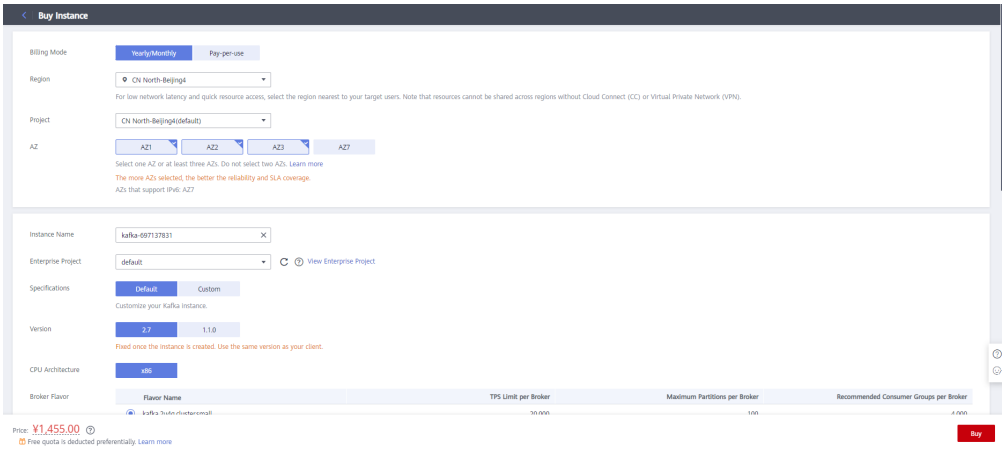
If you want to store data reported by devices, you can either forward the data to application servers or to Distributed Message Service (DMS) for Kafka for storage.

In this example, data reported by all devices is forwarded to DMS for Kafka.

Buying a Kafka Instance

1. Log in to Huawei Cloud and visit **DMS for Kafka**.
2. Click **Access Console** to go to the DMS for Kafka console.
3. Click **Buy Instance** in the upper right corner, select instance specifications and **configure a security group** as required, and click **Buy**.

Figure 7-16 Buying a Kafka Instance



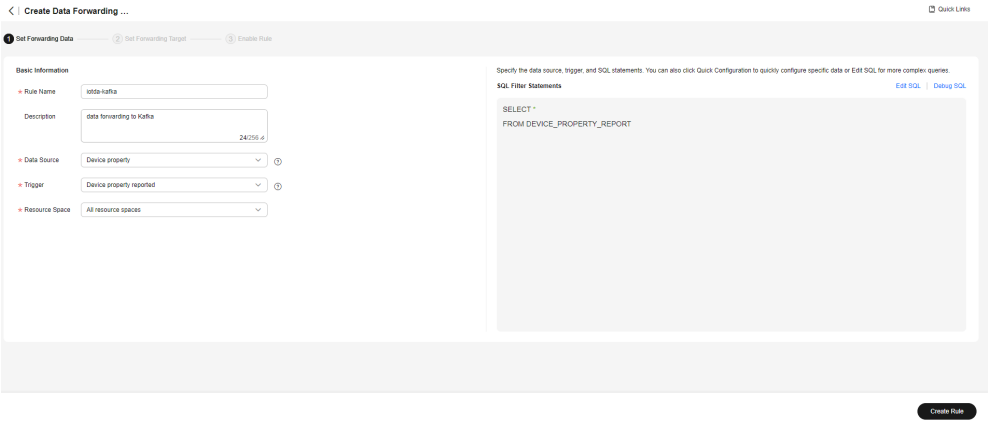
Configuring IoTDA

Using IoTDA, you can create a product model, register a device, and set a data forwarding rule to forward data reported by the device to DMS for Kafka.

1. Access the **IoTDA** service page and click **Access Console**. Click the target instance card.
2. In the navigation pane, choose **Rules > Data Forwarding**, and click **Create Rule** in the upper left corner.
3. Set the parameters based on the table below. The following parameter values are only examples. You can create a rule by referring to **Data Forwarding Overview** and click **Create Rule**.

Parameter	Description
Rule Name	Customize a name, for example, **iotda-kafka**.
Description	Customize a rule description, for example, **forwarding data to DMS for Kafka for storage**.
Data Source	Select **Device property**.
Trigger	**Device property reported** is automatically populated.
Resource Space	Select **All resource spaces**.

Figure 7-17 Rules triggered by property reporting - Forwarding data to Kafka

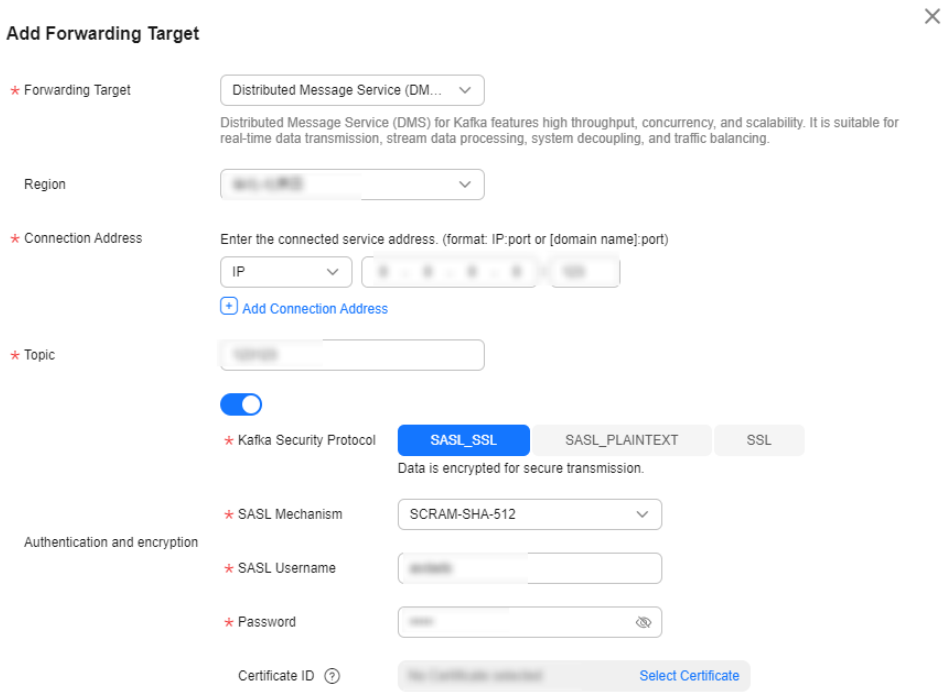


4. Click the **Set Forwarding Target** tab, and then click **Add** to set a forwarding target.

Parameter	Description
Forwarding Target	Select **Distributed Message Service (DMS) for Kafka**.
Region	Select the region where DMS for Kafka is located. If IoTDA is not authorized to access the service in this region, configure cloud service access authorization as prompted.

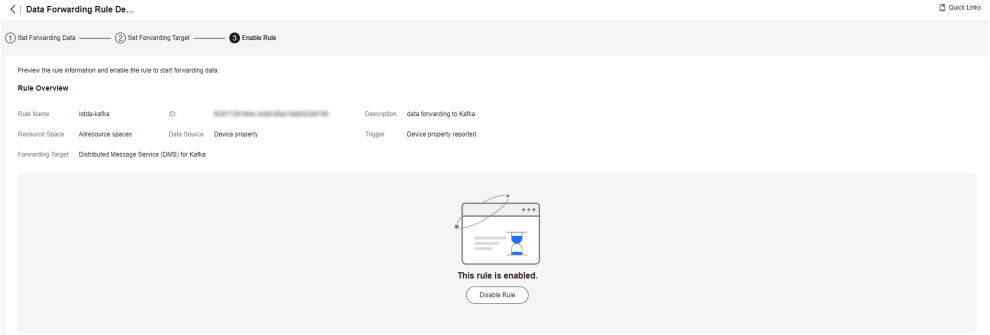
Parameter	Description
Connection Address	Obtain the connection address by following the instructions provided in **Accessing a Kafka Instance with SASL**. Basic and standard instances support only access to Kafka premium instances over the Internet. Enterprise instances support access to Kafka premium instances over a private network.
Topic	Customize a topic. For details, see **Creating a Topic**.
SASL	If SASL authentication is enabled, enter the security protocol, SASL authentication mechanism, and SASL username and password you used when you **buy a Kafka instance**.
Kafka Security Protocol	Select the Kafka security protocol you used when you **buy a Kafka instance**.
SASL Mechanism	Select the SASL authentication mechanism you enabled when you **buy a Kafka instance**.
SASL Username	Enter the SASL username you entered when **buying a Kafka instance**.
Password	Enter the password you entered when **buying a Kafka instance**.

Figure 7-18 Creating a forwarding target - to Kafka with a custom certificate



5. Click **Enable Rule** to activate the configured data forwarding rule.

Figure 7-19 Enabling a rule - Forwarding data to Kafka



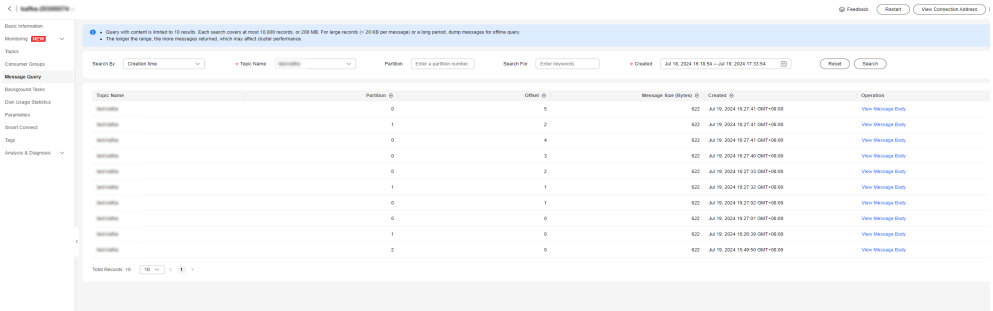
Verifying the Configurations

- You can use a registered physical device to access the platform and enable the device to report data.
- You can also use a simulator to simulate a device to report data. For details, see **Developing an MQTT-based Smart Street Light Online**.

Expected result:

Log in to the DMS for Kafka **management console** and click the Kafka instance name to go to the instance management page. On the **Message Query** page, you can view the data reported by the device.

Figure 7-20 Viewing Kafka messages - Kafka



You can also use the DMS for Kafka API **Querying Messages** to read files.

7.5.4 Forwarding Data to FunctionGraph

Scenarios

FunctionGraph processes the real-time stream data reported by devices to IoTDA. With FunctionGraph, you only need to upload your code and set running conditions to track device properties, message reporting, and status changes as well as analyze, sort out, and measure data flows.

In this example, all properties reported by devices are forwarded to FunctionGraph. The properties are pushed to different paths on your HTTP server based on the resource space ID. You need to deploy an HTTP server. In this example, the data forwarding capability of IoTDA triggers the event function. No additional trigger is required.

Building a Function Project

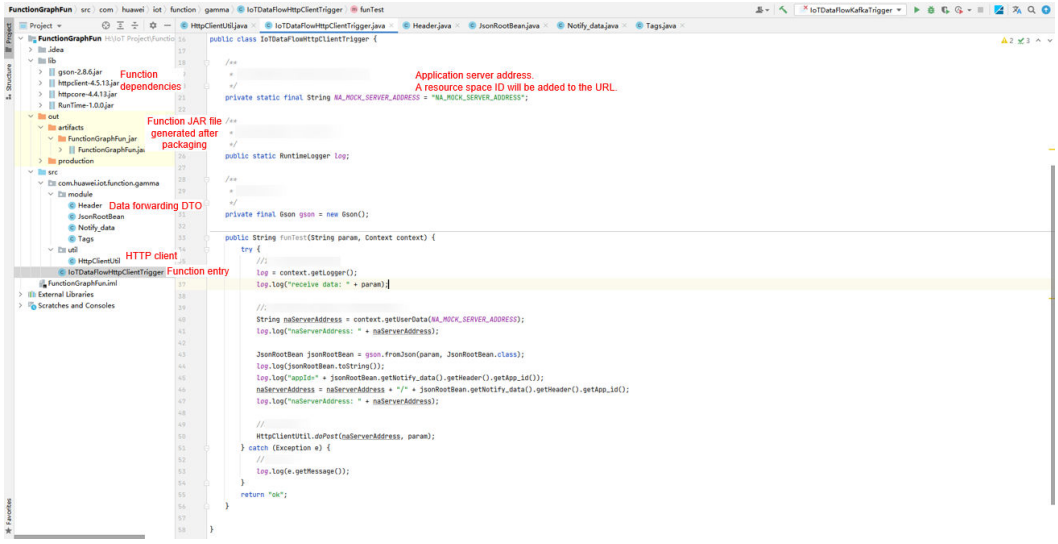
You can download and use the source code (including function dependencies) for converting the format of reported device properties and forwarding the data to a third-party application.

Creating a project

This example uses the Java language to implement device connection, data stream conversion, and data push. For details about function development, see **Developing Functions in Java**.

Download the sample source code, decompress it, and import it to IDEA. For details about the code, see **Sample code**. Transfer your server address through the function environment variable **NA_MOCK_SERVER_ADDRESS**.

Figure 7-21 Sample code



Packaging the project

Package the project into a JAR file using **Build Artifacts** of IDEA. The following figure shows the IDEA configuration and packaging.

Figure 7-22 Artifacts Output configuration

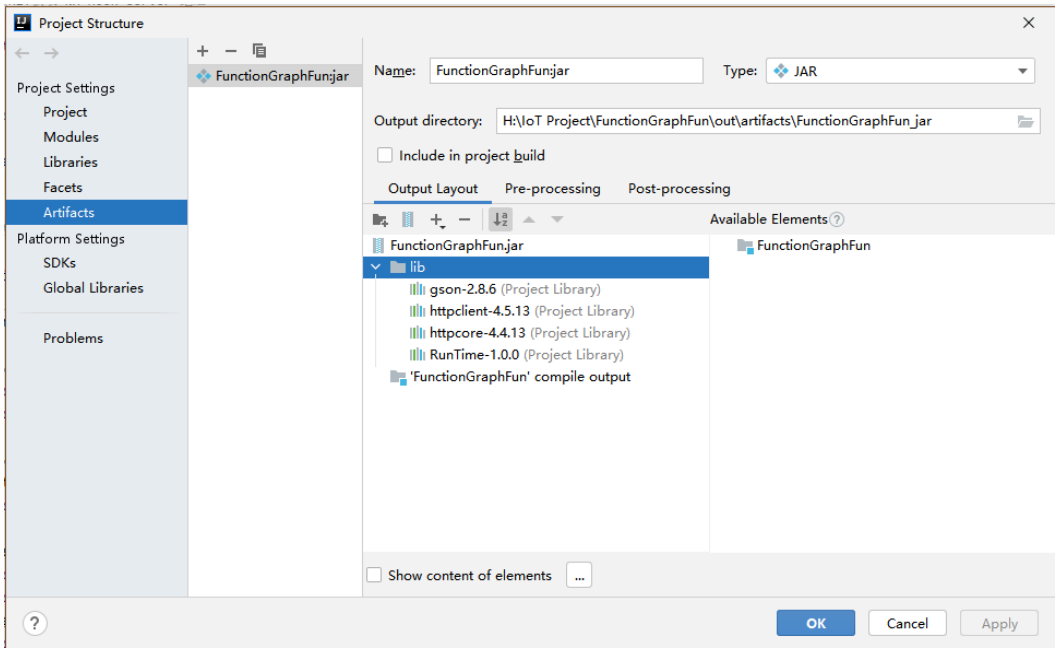
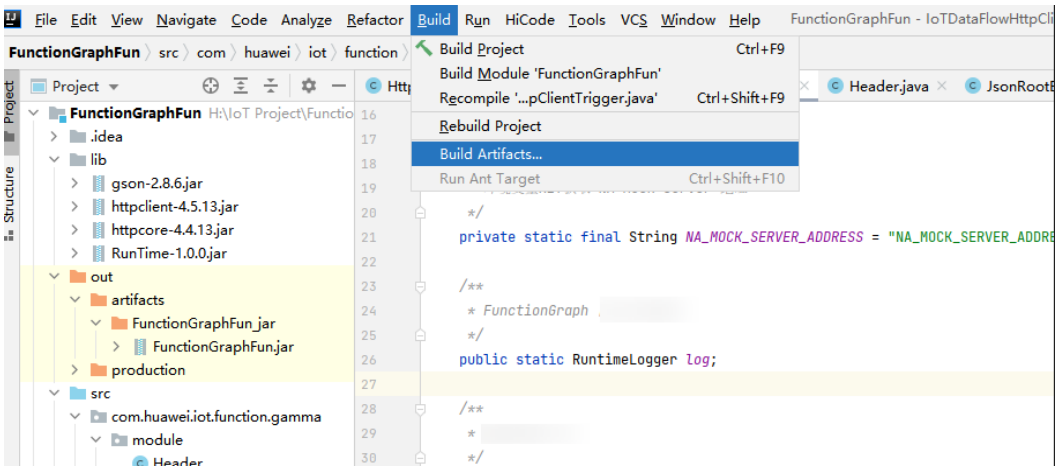


Figure 7-23 Build Artifacts



Uploading the Function to FunctionGraph

Create a function on the FunctionGraph console.

Step 1 Log in to the **FunctionGraph console**, and choose **Functions > Function List** in the navigation pane.

Step 2 Click **Create Function**.

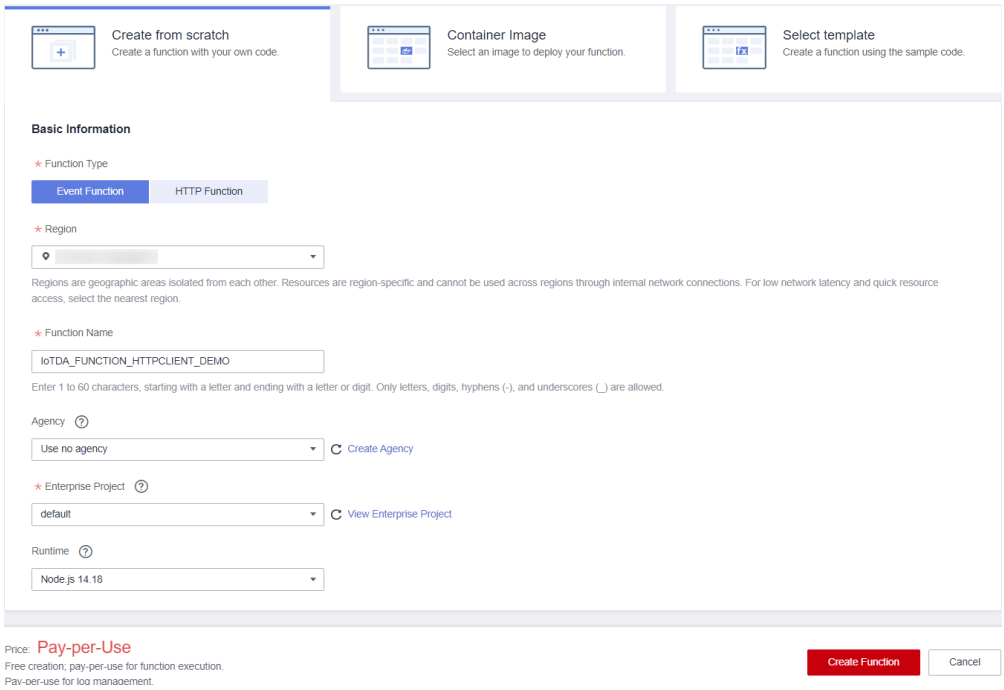
Step 3 Configure the function information, as shown in the following figure.

Click **Create from scratch**.

Enter **IoTDA_FUNCTION_HTTPCLIENT_DEMO** in **Function Name**.

Select **Java 8** for **Runtime**.

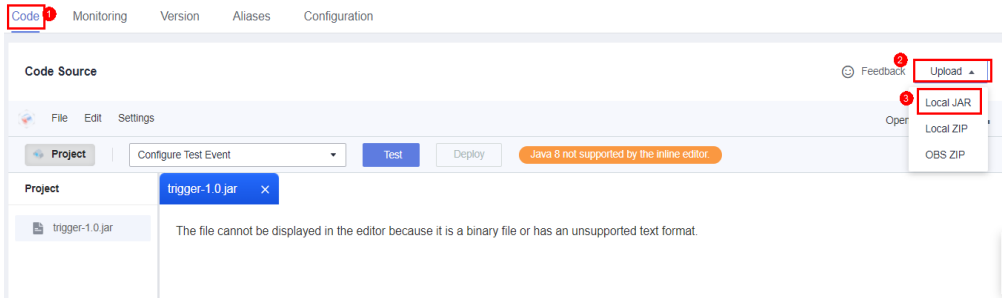
Figure 7-24 Creating a function



Step 4 Click **Create Function**. After the function is created, the function details page is displayed.

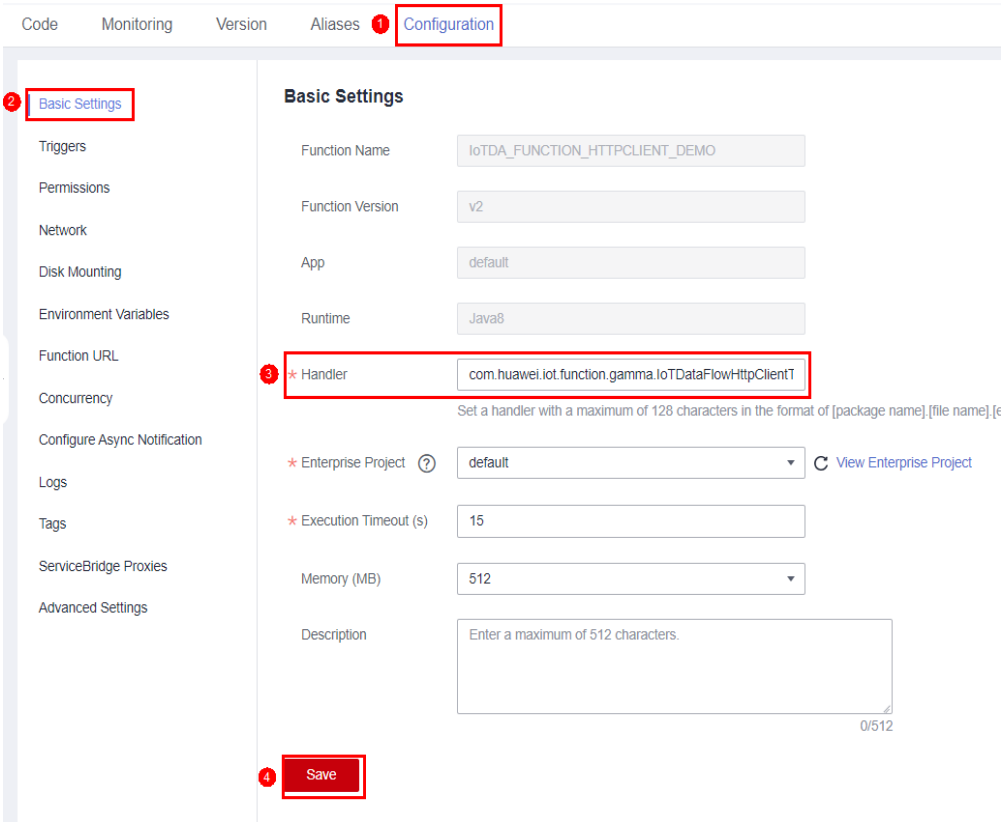
Step 5 Click the **Code** tab and click **Upload** > **Local JAR** to upload the code package **FunctionGraphFun.jar**.

Figure 7-25 Uploading the code



Step 6 Modify the function runtime parameters. Click the **Configuration** tab, choose **Basic Settings**, and set **Handler** to **com.huawei.iot.function.gamma.IoTDataFlowHttpClientTrigger.funTest**. Click **Save** to save the configuration.

Figure 7-26 Setting the handler

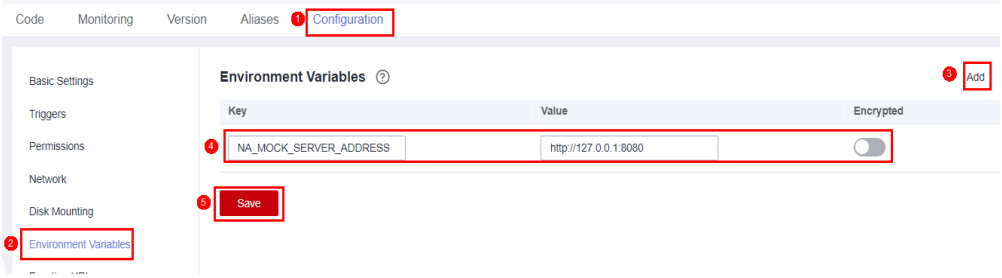


NOTICE

The default function memory size is 512 MB, and the default timeout interval is 15s. This example is only for demonstration. For commercial use, optimize the function parameters based on the site requirements.

Step 7 Modify the environment variables transferred during function calling. Set the environment variable **NA_MOCK_SERVER_ADDRESS** to the target HTTP server address. Note that the server address in the example is not a real one. Replace it with your actual HTTP server address. Click **Save** to save the configuration.

Figure 7-27 Configuring environment variables for function calling



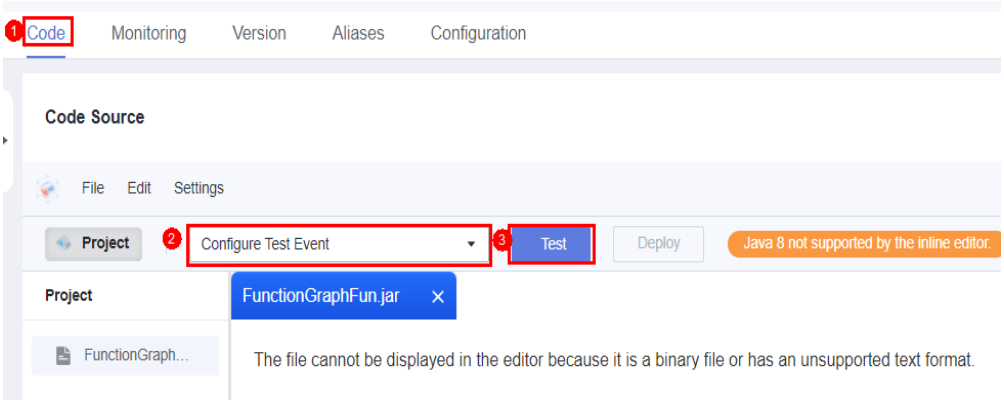
----End

Adding an Event Source

After creating the function, you can add an event source. In this example, an HTTP push test event is configured to simulate device data forwarded by IoTDA. The procedure is as follows:

Step 1 On the **IoTDA_FUNCTION_HTTPCLIENT_DEMO** function page, click the **Code** tab and select **Configure Test Event**.

Figure 7-28 Configuring a test event



Step 2 In the **Configure Test Event** dialog box, enter the test event information.

Select **Create new test event**.

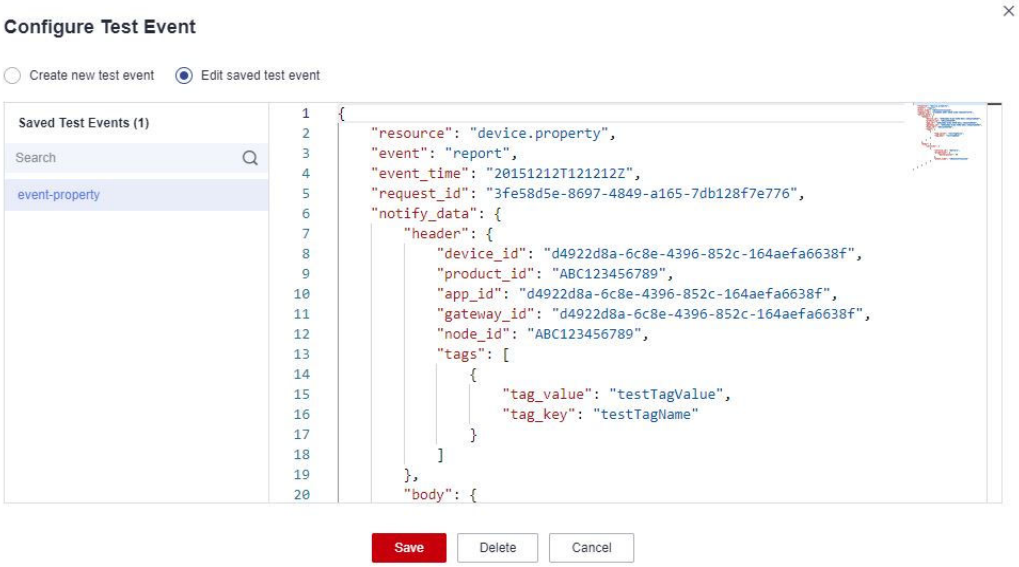
Event Templates: Select **blank-template**.

Event Name: Enter **event-property**.

The following is an example of the test parameters for reporting device properties:

```
{
   "resource":"device.property",
   "event":"report",
   "event_time":"string",
   "notify_data":{
      "header":{
         "app_id":"********",
         "device_id":"********",
         "node_id":"ABC123456789",
         "product_id":"ABC123456789",
         "gateway_id":"********",
         "tags":[{
                "tag_key":"testTagName",
                "tag_value":"testTagValue"
         }]
      },
      "body":{
         "services":[{
                "service_id":"string",
                "properties":{
                },
                "event_time":"string"
         }]
      }
   }
}
```

Figure 7-29 Configuring a test event



Step 3 Click **Save**.

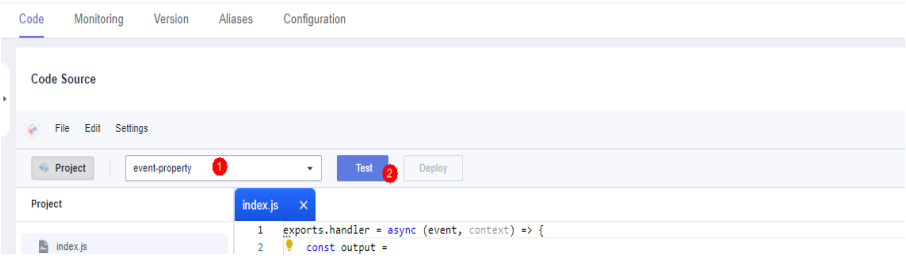
----End

Testing Data

Perform the following steps to process the simulated data:

Step 1 On the function details page, select test event **event-property**, and click **Test** to test the function.

Figure 7-30 Configuring a test event



Step 2 After the function is executed, view the function execution status in the log on the right of the function details page.

Figure 7-31 Function execution result

----End

Configuring IoTDA

You can set configure data forwarding rules in IoTDA to forward data reported by devices to FunctionGraph.

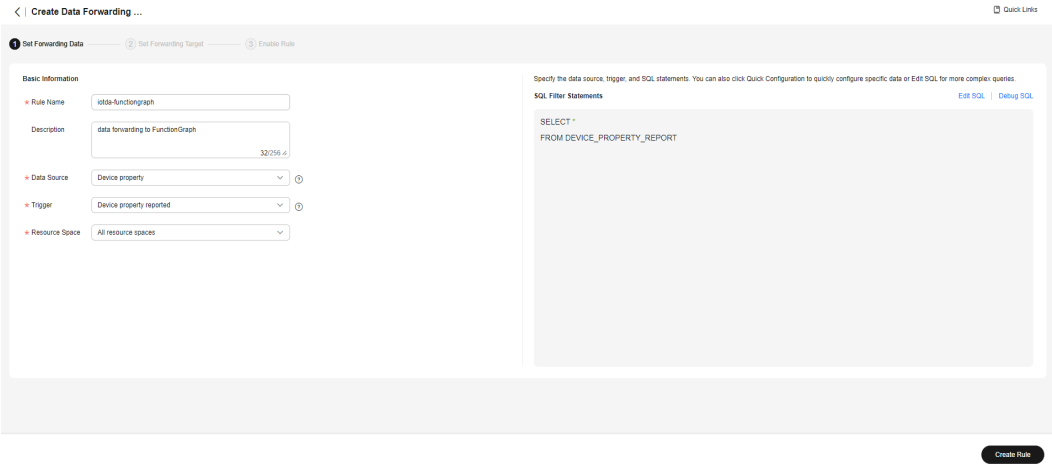
Step 1 Visit the **IoTDA** product page and click **Try Now**.

Step 2 In the navigation pane, choose **Rules>Data Forwarding**, and click **Create Rule** in the upper left corner.

Step 3 Set the parameters based on the table below. The following parameter values are only examples. You can create a rule by referring to **Data Forwarding Overview** and click **Create Rule**.

Parameter	Description
Rule Name	Customize a name, for example, **iotda-functiongraph**.
Description	Enter a rule description, for example, **forwarding data to FunctionGraph**.
Data Source	Select **Device property**.
Trigger	**Device property reported** is automatically populated.
Resource Spaces	Select **All resource spaces**.

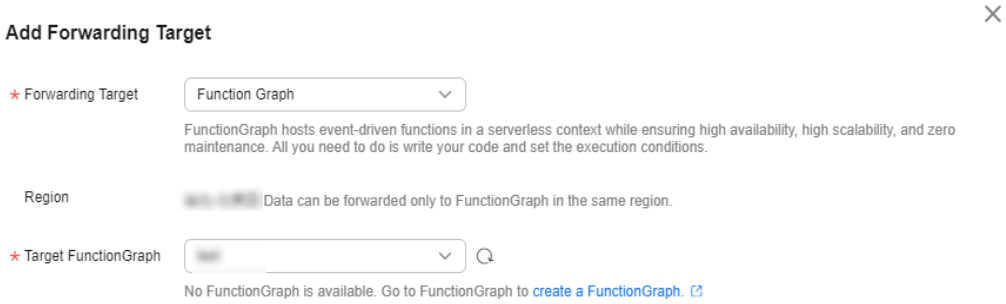
Figure 7-32 Rules triggered by property reporting - Forwarding data to FunctionGraph



Step 4 Click the **Set Forwarding Target** tab, click **Add** to set a forwarding target, and click **OK**.

Parameter	Description
Forwarding Target	Select **FunctionGraph**.
Region	Currently, you can only forward data to FunctionGraph in the same region. If IoTDA is not authorized to access the service in this region, configure cloud service access authorization as prompted.
Target Function	Select the function created in FunctionGraph.

Figure 7-33 Creating a forwarding target - to FunctionGraph



Step 5 Click **Enable Rule** to activate the configured data forwarding rule.

----End

Verifying the Configurations

- You can connect a physical device registered with IoTDA to the platform and report properties defined in the product model.
- You can also use a simulator to simulate device properties reporting. For details, see **Developing an MQTT-based Smart Street Light Online**.

Expected result: The data reported by the device is displayed in the server logs.

Figure 7-34 Expected result

```
@RestController
public class MessageReceiveController {
    private static final Logger log = LoggerFactory.getLogger(MessageReceiveController.class);

    @RequestMapping(value = "/{appId}", method = RequestMethod.POST)
    public void messageReceive(@NotNull @PathVariable("appId") String appId,
        @RequestBody JSONObject requestJson) {
        log.info("receive {}, message is {}", appId, requestJson.toJSONString());
    }
}
```

```
NApplication
onsole   Endpoints
 (javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
2021-04-16 12:39:49.369  INFO 24304 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2021-04-16 12:39:49.398  INFO 24304 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2021-04-16 12:39:49.400  INFO 24304 --- [           main] com.huawei.demo.na.NApplication          : Started NApplication in 2.249 seconds (JVM running for 3.378)
2021-04-16 12:39:51.071  INFO 24304 --- [n(2)-172.23.0.1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring FrameworkServlet 'dispatcherServlet'
2021-04-16 12:39:51.071  INFO 24304 --- [n(2)-172.23.0.1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization started
2021-04-16 12:39:51.131  INFO 24304 --- [n(2)-172.23.0.1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization completed in 60 ms
2021-04-16 12:44:15.071  INFO 24304 --- [nio-8080-exec-1] c.h.d.n.c.MessageReceiveController       : receive f20f[illegible], message is {"resource":"device.property",
 "notify_data":{"header":{"device_id":"d492[illegible]38f","product_id":"[illegible]","app_id":"f20f[illegible]","gateway_id":"d49[illegible]",
 "node_id":"[illegible]","tags":[{"tag_value":"testTagValue","tag_key":"testTagName"}]},"body":{"services":[{"service_id":"temp","properties":"temp","event_time":"20151212T121212Z"}]}},"event":"report",
 "event_time":"20151212T121212Z"}
```

7.5.5 Forwarding Data to MySQL for Storage

Scenarios

IoTDA can forward data reported by devices to RDS for MySQL for storage. You can use device data for service processing without developing additional code to store data.

In this example, data reported by all devices is forwarded to a MySQL database.

Prerequisites

- You have purchased an IoTDA Enterprise or Standard edition instance.
- You have purchased an RDS for MySQL instance.

Creating a MySQL Database

Step 1 Log in to the Huawei Cloud official website, visit **RDS for MySQL**, and purchase an instance. IoTDA enterprise edition instances support **connecting to an RDS for MySQL DB instance through a private network**. IoTDA standard edition instances support **connecting to an RDS for MySQL DB instance through a public network**.

Step 2 When you purchase a MySQL instance, design database tables by referring to **Data Transfer APIs** to enable editing of forwarded data using filter statements. In this example, the default formats in **Push a Device Property Reporting Notification** are used to store the **resource**, **event**, **notify_data**, and **event_time** fields in the forwarded data to the **resource**, **event**, **content**, and **event_time** fields in the database table.

Figure 7-35 Example of creating a database table

```
CREATE TABLE `notify_all` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `resource` varchar(32) DEFAULT NULL,
  `event` varchar(32) DEFAULT NULL,
  `content` json DEFAULT NULL,
  `event_time` char(16) DEFAULT NULL,
  `receive_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=7 DEFAULT CHARSET=utf8mb4;
```

----End

Configuring IoTDA

Using IoTDA, you can create a product model, register a device, and set a data forwarding rule to forward data reported by the device to MySQL.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Products**. Click **Create Product** and select the resource space to which the new product will belong.

NOTE

The product model and device used in this topic are only examples. You can use your own product model and device.

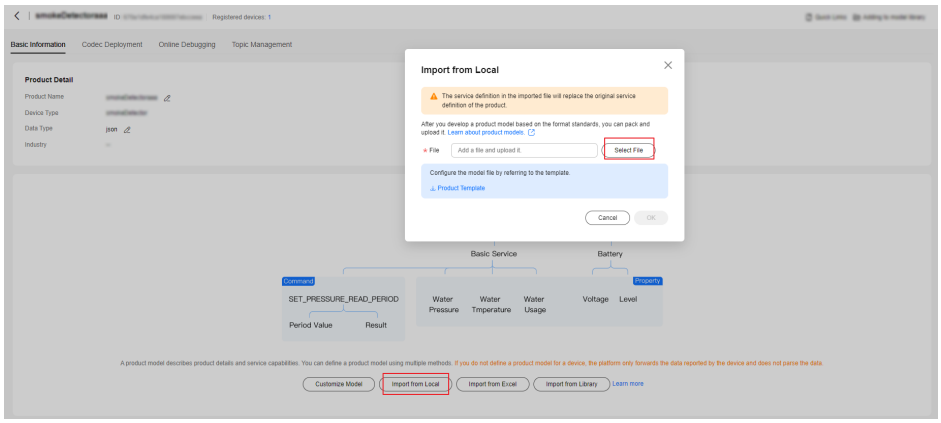
Step 3 Click **Create Product** to create a product using MQTT. Set the parameters and click **OK**.

Basic Information	
Product Name	Enter a value, for example, **MQTT_Device**.
Protocol Type	Select **MQTT**.
Data Type	Select **JSON**.
Industry	Set the parameters as required.
Device Type	

Step 4 Click **here** to download a sample product model.

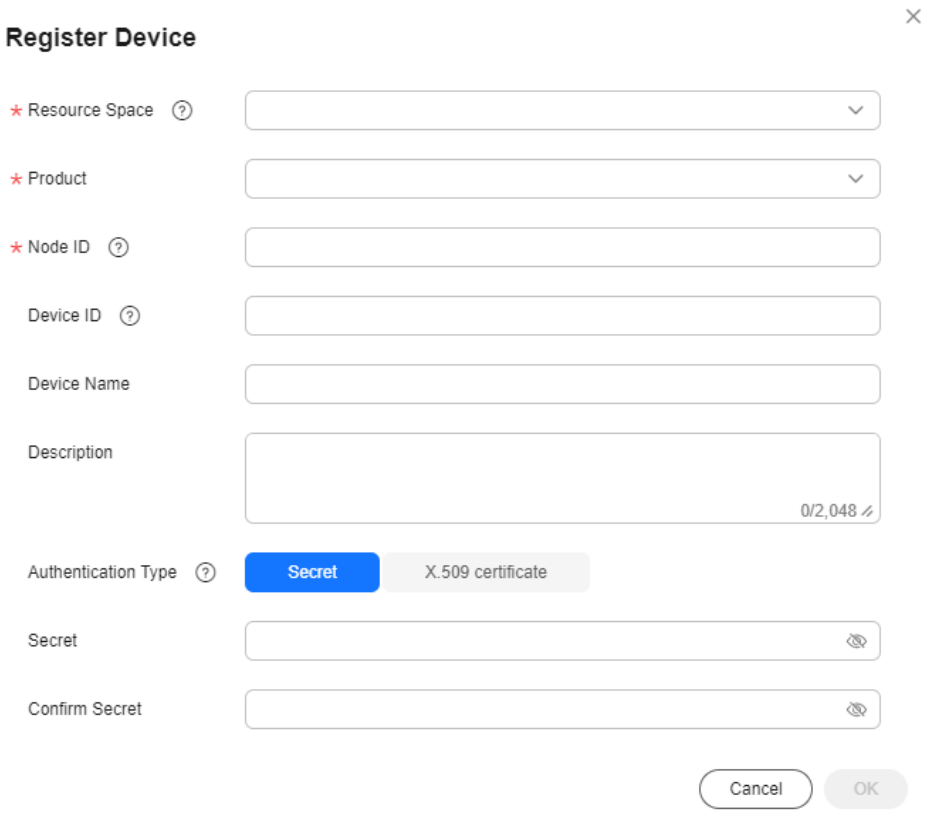
Step 5 On the **Basic Information** tab page, click **Import from Local**. In the displayed dialog box, load the local product model and click **OK**.

Figure 7-36 Product - Uploading a product model



Step 6 In the navigation pane, choose **Devices** > **All Devices**. Click **Register Device**, set device registration parameters, and click **OK**. Save the device ID and secret returned after the registration.

Figure 7-37 Device - Registering a secret device



Parameter	Description
Resource Space	Select the resource space (created in **step 3**) to which the product will belong.

Parameter	Description
Product	Select the product created in **step 3**.
Node ID	Set this parameter to the IMEI, MAC address, or serial number of the device. If the device is not a physical one, set this parameter to a custom character string that contains letters and digits.
Device Name	Customize the product name.
Device ID	Customize the value. You can leave it empty, then the platform will automatically generate a device ID.
Authentication Type	Select **Secret**.
Secret	Customize the secret used for device access. If the secret is left blank, the platform automatically generates one.

Step 7 In the navigation pane, choose **Rules** > **Data Forwarding**, and click **Create Rule** in the upper left corner.

NOTE

You can also add a MySQL database as the forwarding target on the details page of a created rule.

Step 8 Set the parameters based on the table below. The following parameter values are only examples. You can configure parameters of data forwarding rules by referring to **Data Forwarding**. After configuring the parameters, click **Create Rule**.

Parameter	Description
Rule Name	Customize a name, for example, **iotda-mysql**.
Description	Enter a rule description, for example, **forwarding data to MySQL for storage**.
Data Source	Select **Device property**.
Trigger	**Device property reported** is automatically populated.
Resource Spaces	Select a resource space to which the data source to be forwarded belongs or all resource spaces.

Step 9 Click the **Set Forwarding Target** tab, and then click **Add** to set a forwarding target.

Parameter	Description
Forwarding Target	Select **MySQL (RDS)**.
Database Instance Address	Enter the IP address (or port number) for connecting the database instance.

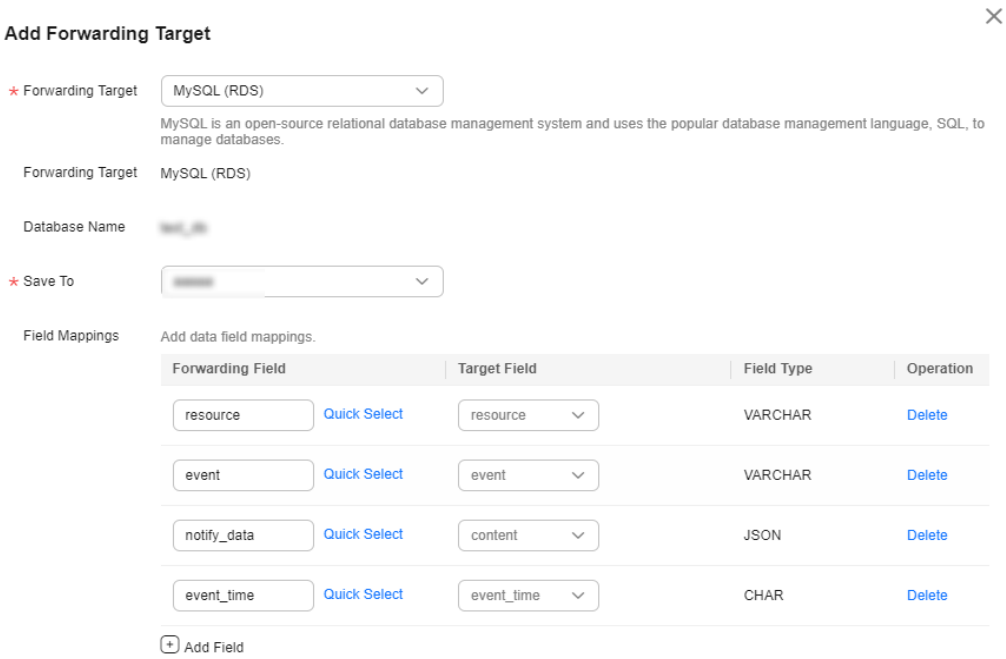
Parameter	Description
Database Name	Enter the name of the destination database in the database instance.
Access Account	Enter the account of the database instance.
Access Password	Enter the password of the database instance.
SSL	Select whether to connect to the database in SSL mode. You are advised to use SSL for connection. If not, security risks may exist during data transmission. To use the SSL mode, **configure an SSL connection** in the database instance first.
Certificate ID	Truststore certificate, which is used by the client to verify the server certificate. If this parameter is left blank, the default certificate provided by RDS for MySQL is used. For **RDS for MySQL instances using custom certificates**, upload the custom CA on the **Rules** > **Server Certificates** page and complete the binding.

Step 10 Click **Next**. IoTDA will connect to the database during the process.

Step 11 Select the target table and configure the mapping between the data to forward and the database table.

- **Forwarding Field**: JSON key of the data to forwarded.
- **Target Field**: field in the database table. After a target field is selected, the field type is automatically matched.

Figure 7-38 Setting MySQL field mapping



Step 12 Click **Enable Rule** to activate the configured data forwarding rule.

----End

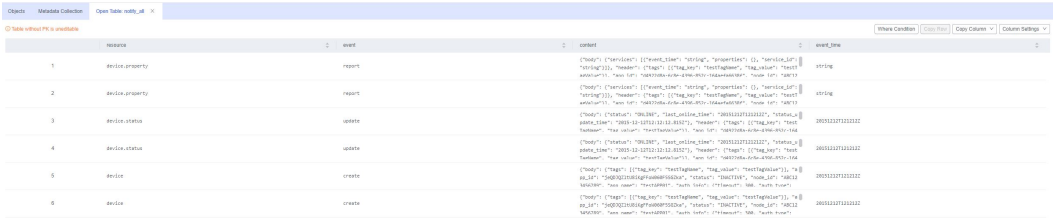
Verifying the Configurations

An event (for example, device property reporting) triggers data forwarding.

Expected result:

Log in to the **RDS for MySQL** console and open the target table. Data forwarded to the target fields is displayed in the table.

Figure 7-39 Querying data reported



7.5.6 Forwarding Device Data to OBS for Long-Term Storage

Scenarios

If you want to store data reported by devices for a long term, you can either forward the data to applications or to Object Storage Service (OBS) for storage.

In this example, data reported by all devices is forwarded to OBS.

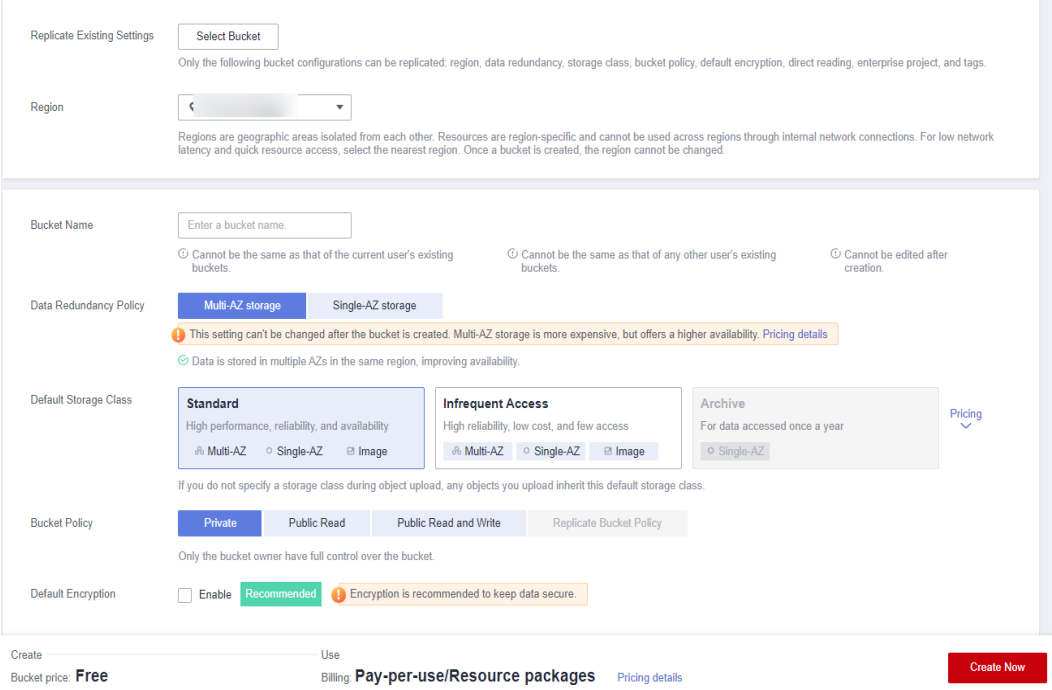
Creating an OBS Bucket

Step 1 Log in to Huawei Cloud and visit **OBS**.

Step 2 Click **Access Console** to go to the OBS console.

Step 3 Click **Create Bucket** in the upper right corner of the page, select bucket specifications as required, and click **Create Now**.

Figure 7-40 Purchasing OBS



----End

Configuring IoTDA

Using IoTDA, you can create a product model, register a device, and set a data forwarding rule to forward data reported by the device to OBS.

Creating a rule

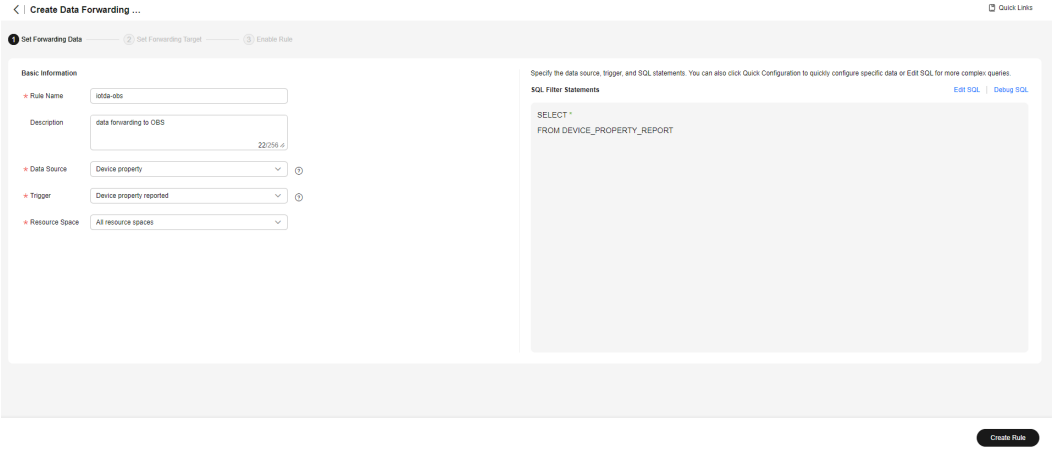
Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Rules** > **Data Forwarding**, and click **Create Rule** in the upper left corner.

Step 3 Set the parameters based on the table below. The following parameter values are only examples. You can create a rule by referring to **Data Forwarding** and click **Create Rule**.

Parameter	Description
Rule Name	Customize a name, for example, **iotda-obs**.
Description	Enter a rule description, for example, **forwarding data to OBS for storage**.
Data Source	Select **Device property**.
Trigger	**Device property reported** is automatically populated.
Resource Spaces	Select **All resource spaces**.

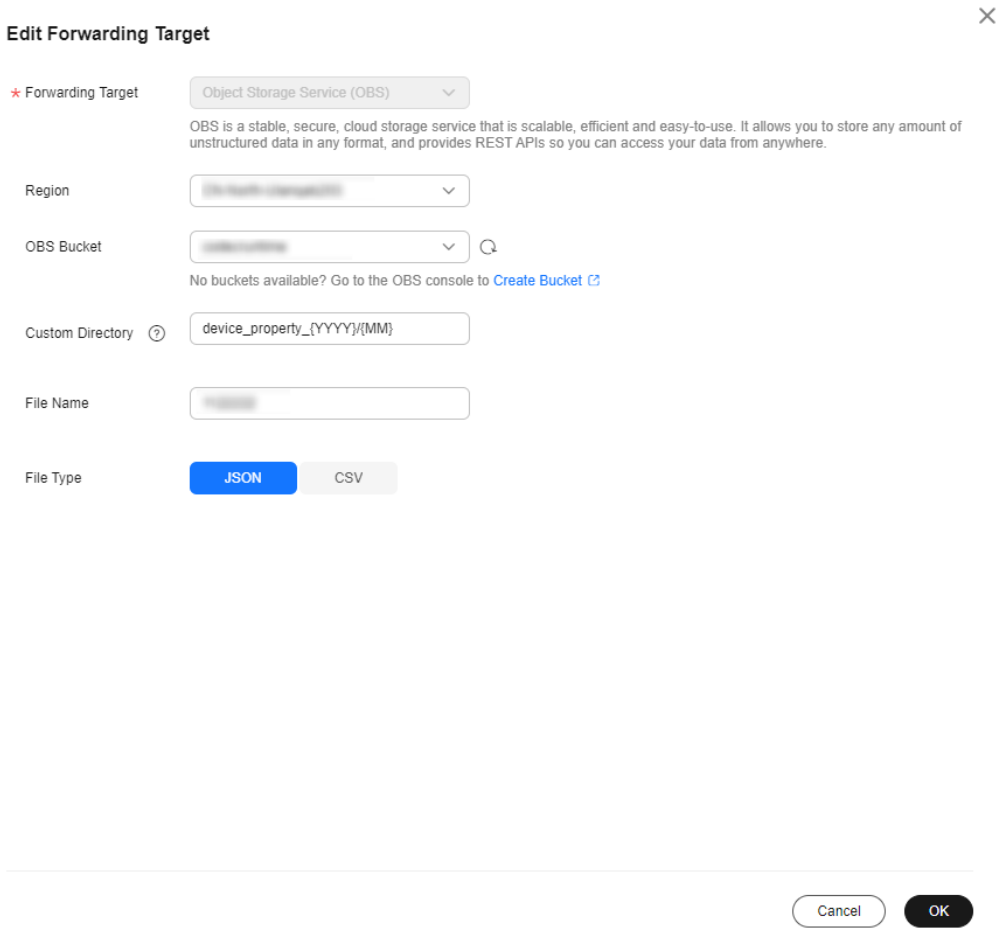
Figure 7-41 Rules triggered by property reporting - Forwarding data to OBS



Step 4 Click the **Set Forwarding Target** tab, and then click **Add** to set a forwarding target.

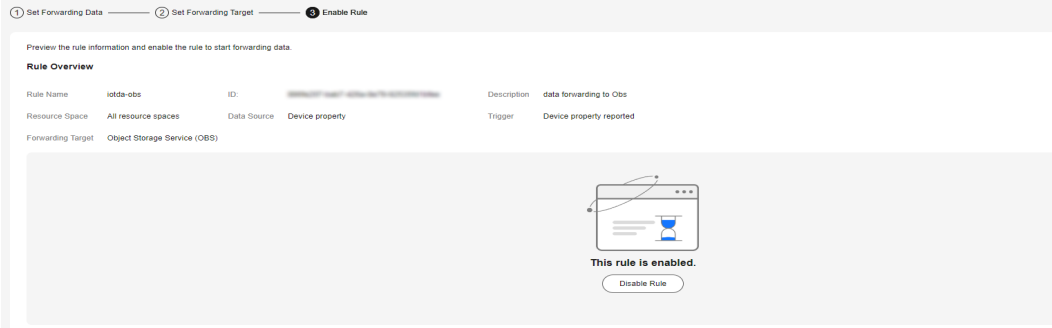
Parameter	Description
Forwarding Target	Select **Object Storage Service (OBS)**.
Region	Select the region where OBS is located. If IoTDA is not authorized to access the service in this region, configure cloud service access authorization as prompted.
OBS Bucket	Select the bucket where data is to be stored. If no OBS bucket is available, **create one** on the OBS console.

Figure 7-42 Creating a forwarding target - to OBS in JSON



Step 5 Click **Enable Rule** to activate the configured data forwarding rule.

Figure 7-43 Enabling a rule - Forwarding data to OBS



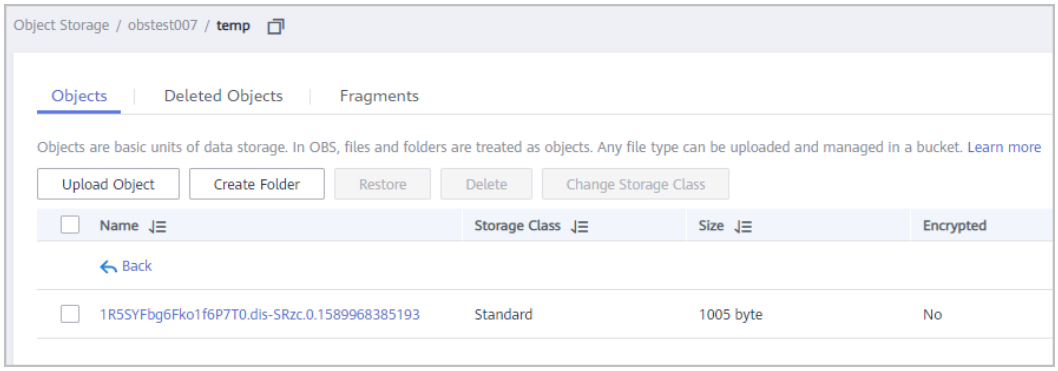
----End

Verifying the Configurations

- You can use a registered physical device to access the platform and enable the device to report data.
- You can also use a simulator to report data by simulating a device. For details, see **Developing an MQTT-based Smart Street Light Online**.

Log in to the OBS console, and click the bucket name. On the **Objects** page, you can view the data reported by the device.

Figure 7-44 Querying reported data in OBS



You can also use the OBS API **Downloading Objects** to read files.

7.6 Data Forwarding to Third-Party Applications

7.6.1 Forwarding Modes

A device can connect to and communicate with the platform. The device reports data to the platform using custom topics or product models. After the subscription/push configuration on the console is complete, the platform forwards messages about device lifecycle changes, reported device properties, reported device messages, device message status changes, device status changes, and batch task status changes to the application.

The platform supports four data forwarding modes: HTTP/HTTPS, AMQP, MQTT, and M2M communications.

- HTTP/HTTPS mode
  - Subscription: You can use an application to call the platform APIs to configure and activate rules, or create a subscription task on the console for obtaining changed device service and management details. Service details involve device lifecycle, device data reporting, device message status, and device status. Management details involve software/firmware upgrade status and result. Related APIs: **Create a Rule Triggering Condition**, **Create a Rule Action**, and **Modify a Rule Trigger Condition**. The URL of the application, also called the callback URL, must be specified during subscription. For details, see **How Do I Obtain the Callback URL When Calling the Subscription API?** .
  - Push: After a subscription is successful, the platform pushes the corresponding change to a specified callback URL based on the type of data subscribed. (For details on the pushed content, see **Data Transfer APIs**.) If an application does not subscribe to a specific type of data notification, the platform does not push the data to the application even if the data has changed. The platform pushes data, in JSON format, using

HTTP or HTTPS. HTTPS requires authentication and is more secure. Therefore, HTTPS is recommended.

For details, see **HTTP/HTTPS Data Forwarding**.

- AMQP mode
  - Subscription: AMQP is short for Advanced Message Queuing Protocol. You can create a subscription task on the IoTDA console, or call platform APIs to configure and activate rules for obtaining changed device service and management details. Service details involve device lifecycle, device data reporting, device message status, and device status. Management details involve software/firmware upgrade status and result. Related APIs: **Create a Rule Triggering Condition**, **Create a Rule Action**, and **Modify a Rule Triggering Condition**. The AMQP message channel must be specified during subscription creation.
  - Push: After a subscription is created, the platform pushes the corresponding change to the specified AMQP message queue based on the type of data subscribed. If an application does not subscribe to a specific type of data notification, the platform does not push the data to the application even if the data has changed. You can use the AMQP client to establish a connection with the platform to receive data.

  For details, see **AMQP Data Forwarding**.
- MQTT mode
  - Subscription: You can call platform APIs to configure and activate rules for obtaining the changed device service and management details. Service details involve device lifecycle, device data reporting, device message reporting, and device status. Management details involve software/firmware upgrade status and result. Related APIs:**Create a Rule Triggering Condition**, **Create a rule action**, and **Modify a Rule Triggering Condition**. The topic for receiving push messages must be specified during subscription creation.
  - Push: After a subscription is created, the platform pushes the corresponding change to the specified topic based on the type of data subscribed. If an application does not subscribe to a specific type of data notification, the platform does not push the data to the application even if the data has changed. You can use the MQTT client to establish a connection with the platform to receive data.

  For details, see **MQTT Data Forwarding**.
- M2M communications
  - Subscription: You can create rules on the console or call the platform APIs to configure and activate rules for obtaining messages reported by devices from the platform. Related APIs: **Create a Rule Trigger Condition**, **Create a Rule Action**, and **Modify the Rule Triggering Condition**. Device subscription supports only message reporting.
  - Push: After the subscription is successful, the platform pushes messages reported by devices to the specified MQTT topic. After devices are connected to the platform, you can subscribe to the topic to receive data for inter-device message communications.

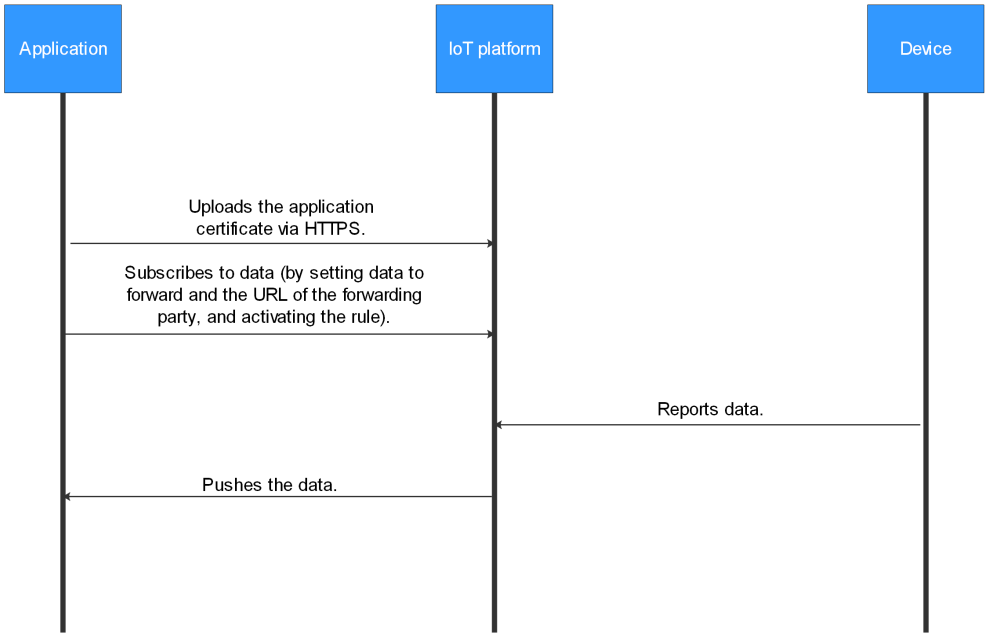
  For details, see **M2M Communications**.

Data Forwarding Mode	Application Scenario	Advantage	Restrictions
HTTP/HTTPS subscription/push	An application functions as the server and passively receives messages from the platform.	-	The traffic control limit is 800 TPS per second. HTTP/HTTPS is not recommended for large-traffic push.
AMQP subscription/push	An application functions as the client and proactively pulls messages from the platform or passively receives messages from the platform by means of listening.	Data can be obtained proactively.	For details, see **Connection Specifications**.
MQTT subscription/push	An application functions as a client and can receive messages from IoT cloud services through subscription.	-	For details, see **Constraints**.
M2M communications	• Smart home scenario where messages are exchanged between mobile apps and smart devices. • Device linkage scenario where devices exchange data and communicate with each other.	Communications among devices are supported.	For details, see **Overview**.

7.6.2 HTTP/HTTPS Data Forwarding

Overview

The figure below shows the subscription and push process.



Before pushing HTTPS messages to an application, the platform must verify the application authenticity. Therefore, the application CA certificate must be loaded to the platform. (You can **create a commissioning certificate** during commissioning and replace it with a commercial certificate during commercial use to avoid security risks.)

Push mechanism: After receiving a push message from the platform, the application returns a 200 OK message. If the application does not respond within 15 seconds or returns a non-200 response code (500, 501, 502, 503, or 504), the message push fails and the message will be discarded. If the platform fails to push the message for 10 times in a row, IoTDA adds the host address of the subscription URL to the blocklist and messages to push will be stacked on the platform for one day or until the stack size of data become 1 GB. To retain only the latest data, see **Data Forwarding Stack Policies**. Then, the platform attempts to push messages to the host address in the blocklist every 3 minutes. If the push fails, the platform keeps the blocklist. If the push succeeds, the platform removes the host address from the blocklist. After the host address is removed from the blocklist, the latest messages are pushed only after all stacked messages are pushed based on the maximum flow control value. The default flow control value is 800 TPS per second. For details about the customized configuration, see **Data Forwarding Flow Control Policies**.

Subscribing to Data

After connecting to IoTDA, an application calls an API to subscribe to data.

- For details on how to configure HTTP or HTTPS subscriptions on the console, see **Configuring HTTP/HTTPS Subscription** and **Loading the CA Certificate**.
- For details on how to subscribe to data through APIs, see **Calling APIs**, **Creating a Rule Trigger Condition**, **Creating a Rule Action**, and **Modifying a Rule Trigger Condition**.

Format of Pushed Data

For details on the format of data pushed by the platform to applications after data subscription is created, see **Data Transfer APIs**.

NOTE

In the HTTP message header, the value of **Content-Type** is **application/json**, and the character set is **UTF-8**.

Loading the CA Certificate

If HTTPS is used, you must load the push certificate by following the instructions provided in this section. Then create a subscription task on the console by following the instructions provided in **Configuring HTTP/HTTPS Subscription**.

- If the application cancels the subscription and then re-subscribes the data again (with the URL unchanged), the CA certificate must be loaded to the platform again.
- If a subscription type (URL) is added, you must load the CA certificate corresponding to the URL to the platform. Even if the CA certificate used by the new URL is the same as that used by the original URL, the CA certificate must be loaded again.

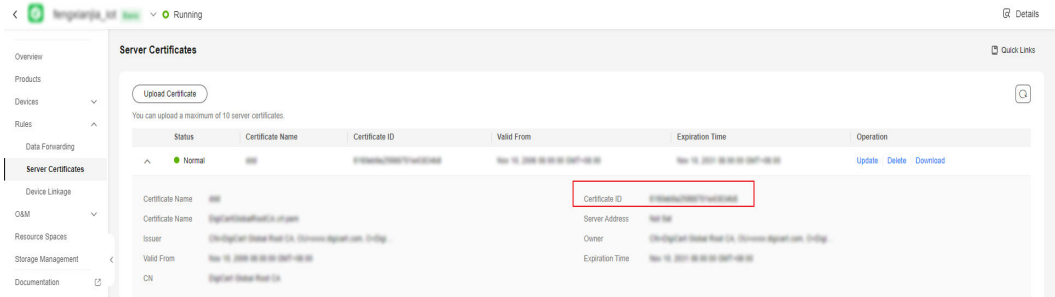
Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Rules** > **Server Certificates**. Click **Upload Certificate**, configure parameters based on the following table, and click **OK**.

Parameter	Description
Certificate Name	Used to distinguish different certificates and can be customized.
CA Certificate	A CA certificate from the application can be applied for and purchased in advance. **NOTE** You can **create a commissioning certificate** during commissioning. For security reasons, you are advised to replace the commissioning certificate with a commercial certificate during commercial use.

Step 3 In the navigation pane, choose **Rules** > **Server Certificate**, locate the target certificate, click ⌄ to obtain the certificate ID, which is used as a parameter in the API **Creating a Rule Action**.

Figure 7-45 Server certificate - Obtaining the certificate ID



----End

Creating a Commissioning Certificate

A commissioning certificate, or a self-signed certificate, is used for authentication when the client accesses the server through HTTPS. When the platform uses HTTPS to push data to an application, the platform authenticates the application. This section uses the Windows operating system as an example to describe how to use OpenSSL to make a commissioning certificate. The generated certificate is in PEM format and the suffix is **.cer**.

The table below lists common certificate storage formats.

Storage Format	Description
DER	Binary code. The suffix is **.der**, **.cer**, or **.crt**.
PEM	Base64 code. The suffix is **.pem**, **.cer**, or **.crt**.
JKS	Java certificate storage format. The suffix is **.jks**.

NOTE

The commissioning certificate is used only for commissioning. During commercial use, you must apply for certificates from a trusted CA. Otherwise, security risks may occur.

Step 1 Download and install **OpenSSL**.

Step 2 Open the CLI as user **admin**.

Step 3 Run **cd c:\openssl\bin** (replace **c:\openssl\bin** with the actual OpenSSL installation directory) to access the OpenSSL view.

Step 4 Generate the private key file **ca_private.key** of the CA root certificate.

```
openssl genrsa -passout pass:123456 -aes256  -out ca_private.key 2048
```

- **aes256**: cryptographic algorithm
- **passout pass**: private key password
- **2048**: key length

Step 5 Use the private key file of the CA root certificate to generate the **ca.csr** file to be used in **6**.

```
openssl req -passin pass:123456 -new -key ca_private.key -out ca.csr -subj "/C=CN/ST=GD/L=SZ/O=Huawei/OU=IoT/CN=CA"
```

Modify the following information based on actual conditions:

- **C**: country, for example, CN
- **ST**: region, for example, GD
- **L**: city, for example, SZ
- **O**: organization, for example, Huawei
- **OU**: organization unit, for example, IoT
- **CN**: common name (the organization name of the CA), for example, CA

Step 6 Create the CA root certificate **ca.cer**.

```
openssl x509 -req -passin pass:123456 -in ca.csr -out ca.cer -signkey ca_private.key -CAcreateserial -days 3650
```

Modify the following information based on actual conditions:

- **passin pass**: The value must be the same as the private key password set in **4**.
- **days**: validity period of the certificate.

Step 7 Generate the private key file for the application.

```
openssl genrsa -passout pass:123456 -aes256 -out server_private.key 2048
```

Step 8 Generate the **.csr** file for the application.

```
openssl req -passin pass:123456 -new -key server_private.key -out server.csr -subj "/C=CN/ST=GD/L=SZ/O=Huawei/OU=IoT/CN=appserver.iot.com"
```

Modify the following information based on actual conditions:

- **C**: country, for example, CN
- **ST**: region, for example, GD
- **L**: city, for example, SZ
- **O**: organization, for example, Huawei
- **OU**: organization unit, for example, IoT
- **CN**: common name. Enter the domain name or IP address of the application.

Step 9 Use the CA private key file **ca_private.key** to sign the file **server.csr** and generate the server certificate file **server.cer**.

```
openssl x509 -req -passin pass:123456 -in server.csr -out server.cer -sha256 -CA ca.cer -CAkey ca_private.key -CAserial ca.srl -CAcreateserial -days 3650
```

Step 10 (Optional) If you need a **.crt** or **.pem** certificate, proceed this step. The following uses the conversion from **server.cer** to **server.crt** as an example. To convert the **ca.cer** certificate, replace **server** in the command with **ca**.

```
openssl x509 -inform PEM -in server.cer -out server.crt
```

Step 11 In the **bin** folder of the OpenSSL installation directory, obtain the CA certificate (**ca.cer**/**ca.crt**/**ca.pem**), application server certificate (**server.cer**/**server.crt**/**server.pem**), and private key file (**server_private.key**). The CA certificate is loaded to the platform, and the application server certificate and private key file are loaded to the application.

----End

Configuring HTTP/HTTPS Subscription

This section describes how to configure HTTP or HTTPS subscription on the console.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Rules** > **Data Forwarding**, and click **Create Rule** in the upper left corner.

Step 3 Set the parameters based on the table below and click **Create Rule**.

Table 7-11 Parameters for creating a rule

Parameter	Description
Rule Name	Name of the rule to be created.
Description	Description of the rule.

Parameter	Description
Data Source	• **Device**: Device information, such as device addition, deletion, and update, will be forwarded. When **Data Source** is set to **Device**, quick configuration is not supported. • **Device property**: A property value reported by a device in a resource space will be forwarded. Click **Quick Configuration** on the right and select the product, property, and service data to forward. • **Device message**: A message reported by a device in a resource space will be forwarded. Click **Quick Configuration** on the right and select data of a specified topic to forward. Select the product to which the topic belongs and enter the topic name. You can use a **custom topic** on the product details page or a **preset topic**. • **Device message status**: The status of device messages exchanged between the device and platform will be forwarded. For details on the device message status, see **Message Delivery Status**. When **Data Source** is set to **Device message status**, quick configuration is not supported. • **Device status**: The status change of a directly or an indirectly connected device in a resource space will be forwarded. Click **Quick Configuration** on the right to forward information about devices whose status is **Online**, **Offline**, or **Abnormal** to other services. For details on the status of devices directly connected to the platform, see **Device Management**. • **Batch task**: The batch task status will be forwarded. When **Data Source** is set to **Batch Task**, quick configuration is not supported. • **Product**: Product information, such as product addition, deletion, and update, will be forwarded. When **Data Source** is set to **Product**, quick configuration is not supported. • **Asynchronous command status of the device**: Status changes of asynchronous commands to devices using LwM2M over CoAP will be forwarded. For details on the asynchronous command status of devices, see **Asynchronous Command Delivery**. When **Data Source** is set to **Asynchronous command status of the device**, quick configuration is not supported. • **Run log**: Service run logs of MQTT devices will be forwarded. When **Data Source** is set to **Run log**, quick configuration is not supported.
Trigger	After you select a data source, the platform automatically matches trigger events.
Resource Space	You can select a single resource space or all resource spaces. If **All resource spaces** is selected, quick configuration is not supported.

Step 4 Under **Set Forwarding Target**, click **Add**. On the displayed page, set the parameters based on the table below and click **OK**.

Parameter	Description
Forwarding Target	Select **Third-party application (HTTP push)**.
Push URL	Enter the URL for IoTDA to push messages to the application. For example, if the URL is **https://www.example.com:8443/example/**, set **Domain/IP and Port** to **www.example.com:8443** in **Loading the CA Certificate**. ● If the push URL uses HTTP, the CA certificate is not required. ● If the push URL uses HTTPS, upload the CA certificate. For details about how to upload a certificate, see **Loading the CA Certificate**.
Token	Used for signature authentication. The value can contain 3 to 32 characters, including letters and digits. When pushing data to the user server, the platform signs the token and assembles the signature information into the header.
Certificate ID	This configuration is valid only for the HTTPS server. It is used as the truststore certificate for the client to verify the compliance of the commercial certificate of the server. This configuration is unavailable for non-compliant certificates such as self-signed certificates and certificates with incomplete certificate chains.
Certificate Domain Name	To enable SNI, configure corresponding certificate and domain name on the server in advance.

Step 5 After the rule is defined, click **Enable Rule** to start forwarding data to the HTTP or HTTPS message queue.

----End

Token-based Platform Authentication for HTTP/HTTPS Push

If you select **Authentication** and enter the token when adding the **Third-party application (HTTP push)** forwarding target, the platform will add the following parameters to the header of the HTTP or HTTPS request:

Parameter	Description
timestamp	Timestamp when the platform pushes data.
nonce	Random number generated by the platform.
signature	Signature consisting of **token**, **timestamp**, and **nonce**.

Signature rules:

1. Sort **token**, **timestamp**, and **nonce** in alphabetical order.
2. Encrypt the sorted string using SHA-256.
3. After receiving the pushed message, you can encrypt **timestamp** and **nonce** in the header and **token** based on rules and compare the obtained value with the signature in the header to determine whether the message is from the platform.

Java example for signature verification:

1. Add the dependency. Use a specific version based on the actual service requirements.
```
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>${commons.version}</version>
</dependency>
```
2. Obtain the signature information from the request header and use the **commons-codec** dependency package for signature.
```
public boolean checkSignature(String nonce, String timestamp, String signature, String token) {
    List<String> list = new ArrayList<>();
    list.add(token);
    if (StringUtil.isNotEmpty(nonce)) {
        list.add(nonce);
    }
    if (StringUtil.isNotEmpty(timestamp)) {
        list.add(timestamp);
    }
    Collections.sort(list);
    StringBuilder signatureBuilder = new StringBuilder();
    for (String s : list) {
        signatureBuilder.append(s);
    }
    String serverSignature = DigestUtils.sha256Hex(signatureBuilder.toString());
    if (StringUtil.isNotEmpty(serverSignature) && serverSignature.equals(signature)) {
        return true;
    }
    return false;
}
```
3. For example, the token is set to **aaaaaa** in a request, and the header contains the following parameters:
```
nonce: 8b9b796d388d49bba43adaa53aaf5bc4
timestamp: 1675654743514
signature: 2ff821fb8a976ede7d06434395ec8c25e4100bff8b3d12d8099ef7e30b58bd4c
```
The string after sorting is **16756547435148b9b796d388d49bba43adaa53aaf5bc4aaaaaa**. The string encrypted using SHA-256 is **2ff821fb8a976ede7d06434395ec8c25e4100bff8b3d12d8099ef7e30b58bd4c**.

CAUTION

After a token is created, you need to configure a new token each time you modify the forwarding target. Otherwise, the token does not take effect.

Platform Certification

As a server, if an application needs to authenticate the platform, the platform CA certificate must be loaded on the application. For details, see **Obtaining Resources**.

FAQ

The following lists the frequently asked questions about the subscription and push service. For more questions, see **Subscription and Push**.

- **How Do I Obtain the Access Addresses and Certificates of the Old and New Domain Names?**
- **How Do I Obtain the Callback URL When Calling the Subscription API?**
- **Can a Domain Name Be Used in a Callback URL?**
- **What Do I Do If Message Push Fails After Subscription?**
- **Why Does the Application Receive Multiple Push Messages After a Device Reports a Piece of Data?**
- **Why Was the Callback URL Invalid During the Subscription API Call?**
- **How Can I Obtain the subscriptionId Needed in Calling the API for Deleting a Subscription?**
- **Can an Application Subscribe to the Platform Data When the Application Only Has an Internal IP Address?**

APIs

Creating a Rule Action

Creating a Rule Trigger Condition

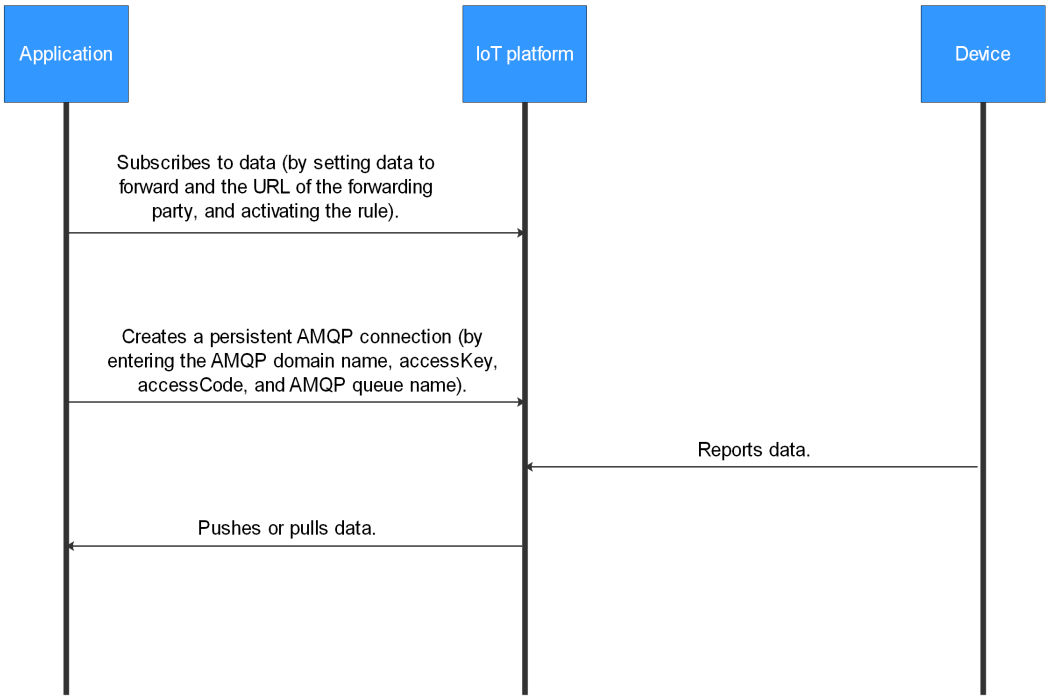
Modifying a Rule Trigger Condition

Data Transfer APIs

7.6.3 AMQP Data Forwarding

7.6.3.1 Overview

The figure below shows the subscription and push process.



Push mechanism: After receiving a message from the platform, the application returns a response. (The automatic response mode is recommended.) If the application does not pull data after the connection is established, data will be stacked on the server. The server stores only the data that is received in the last 24 hours and occupies less than 1 GB disk space. If the application does not pull data in a timely manner, the platform clears expired and excess data on a rolling basis. If the application does not respond in time after receiving the message and the persistent connection is interrupted, the corresponding data will be pushed again in the next connection established.

Subscribing to Data

After connecting to IoTDA, an application calls an API to subscribe to data.

- For details on how to configure subscriptions on the console, see **AMQP Server Configuration**.
- For details on how to subscribe to data through APIs, see **Calling APIs**, **Creating a Rule Trigger Condition**, **Creating a Rule Action**, and **Modifying a Rule Trigger Condition**.

Format of Pushed Data

For details on the format of data pushed by the platform to applications after data subscription is created, see **Data Transfer APIs**.

NOTE

In the HTTP message header, the value of **Content-Type** is **application/json**, and the character set is **UTF-8**.

APIs

Creating a Rule Action

Creating a Rule Trigger Condition

Modifying a Rule Trigger Condition

Data Transfer APIs

Creating an AMQP Queue

Querying the AMQP List

Querying an AMQP Queue

Generating an Access Credential

7.6.3.2 AMQP Server Configuration

This topic describes how to set and manage AMQP server subscription on the IoT platform.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Rules** > **Data Forwarding**, and click **Create Rule** in the upper left corner.

Step 3 Set the parameters based on the table below and click **Create Rule**.

Table 7-12 Parameters for creating a rule

Parameter	Description
Rule Name	Name of the rule to be created.
Description	Description of the rule.

Parameter	Description
Data Source	• **Device**: Device information, such as device addition, deletion, and update, will be forwarded. When **Data Source** is set to **Device**, quick configuration is not supported. • **Device property**: A property value reported by a device in a resource space will be forwarded. Click **Quick Configuration** on the right and select the product, property, and service data to forward. • **Device message**: A message reported by a device in a resource space will be forwarded. Click **Quick Configuration** on the right and select data of a specified topic to forward. Select the product to which the topic belongs and enter the topic name. You can use a **custom topic** on the product details page or a **preset topic**. • **Device message status**: The status of device messages exchanged between the device and platform will be forwarded. For details on the device message status, see **Message Delivery Status**. When **Data Source** is set to **Device message status**, quick configuration is not supported. • **Device status**: The status change of a directly or an indirectly connected device in a resource space will be forwarded. Click **Quick Configuration** on the right to forward information about devices whose status is **Online**, **Offline**, or **Abnormal** to other services. For details on the status of devices directly connected to the platform, see **Device Management**. • **Batch task**: The batch task status will be forwarded. When **Data Source** is set to **Batch Task**, quick configuration is not supported. • **Product**: Product information, such as product addition, deletion, and update, will be forwarded. When **Data Source** is set to **Product**, quick configuration is not supported. • **Asynchronous command status of the device**: Status changes of asynchronous commands to devices using LwM2M over CoAP will be forwarded. For details on the asynchronous command status of devices, see **Asynchronous Command Delivery**. When **Data Source** is set to **Asynchronous command status of the device**, quick configuration is not supported. • **Run log**: Service run logs of MQTT devices will be forwarded. When **Data Source** is set to **Run log**, quick configuration is not supported.
Trigger	After you select a data source, the platform automatically matches trigger events.
Resource Space	You can select a single resource space or all resource spaces. If **All resource spaces** is selected, quick configuration is not supported.

Step 4 Under **Set Forwarding Target**, click **Add**. On the displayed page, set the parameters based on the table below and click **OK**.

Parameter	Description
Forwarding Target	Select **AMQP message queue**.
Message Queue	Click **Select** to select a message queue. • If no message queue is available, create one. The queue name must be unique under a tenant and can contain 8–128 characters, including letters, numbers, underscores (_), hyphens (-), periods (.), and colons (:). • To delete a message queue, click **Delete** on the right of the message queue. **NOTE** A subscribed queue cannot be deleted.

Step 5 After the rule is defined, click **Enable Rule** to start forwarding data to the AMQP message queue.

----End

7.6.3.3 AMQP Queue Alarm Configuration

When you consume messages from a subscribed AMQP queue, the consumer side may go offline and message consumption may slow down due to network communication problems or untimely acknowledgements to received messages. In this case, messages are stacked and cannot be processed in real time.

IoTDA supports AMQP queue alarm configuration. You can set alarm rules to monitor AMQP queue message stacking and consumption speed. After a rule is triggered, alarm information is sent to you immediately so that you can locate and rectify faults in a timely manner. This section describes how to configure alarm rules for AMQP queues.

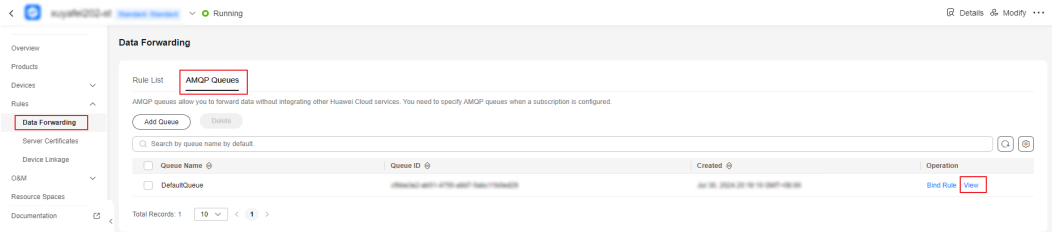
Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Rules** > **Data Forwarding**.

Step 3 Click the **AMQP Queues** tab, locate the queue to configure alarms, and click **View**.

Figure 7-46 Data forwarding - AMQP message queue details



Step 4 Click **Configure Alarm**.

Figure 7-47 AMQP message queue - Alarm configuration

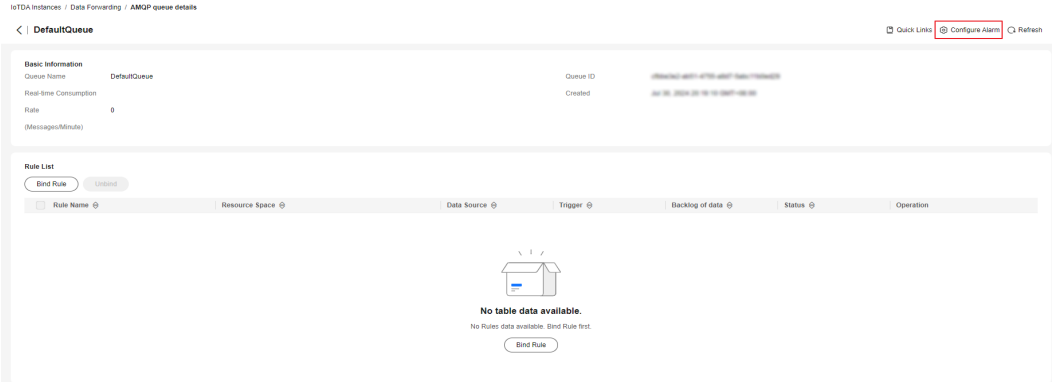
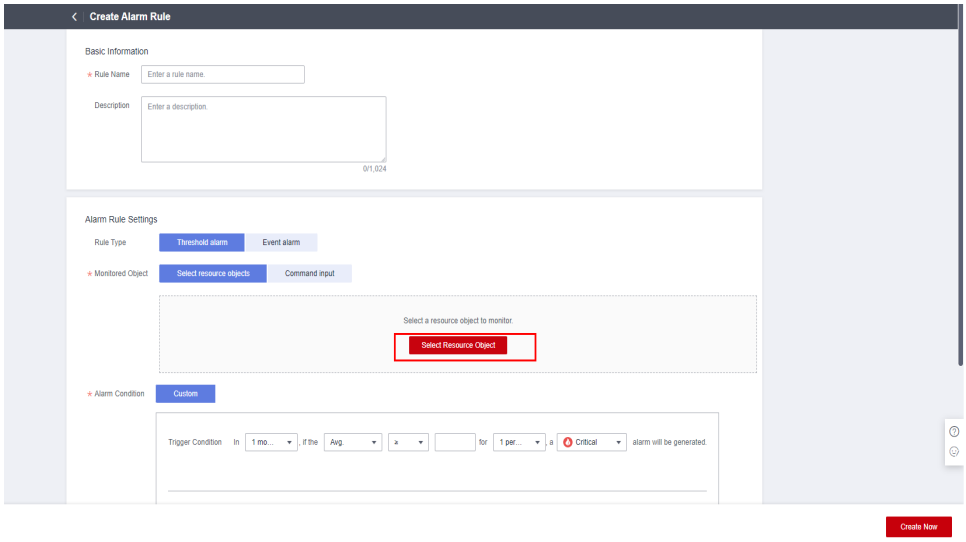


Figure 7-48 Creating an alarm rule



Step 5 Click **Select Resource Object**. Set **Add By** to **Dimension**, and select a proper metric and dimension based on the following tables.

CAUTION

Search for the **iotda_amqp_forwarding_backlog_message_count** and **iotda_amqp_forwarding_consume_rate** under all metrics.

Figure 7-49 Selecting the object to be monitored

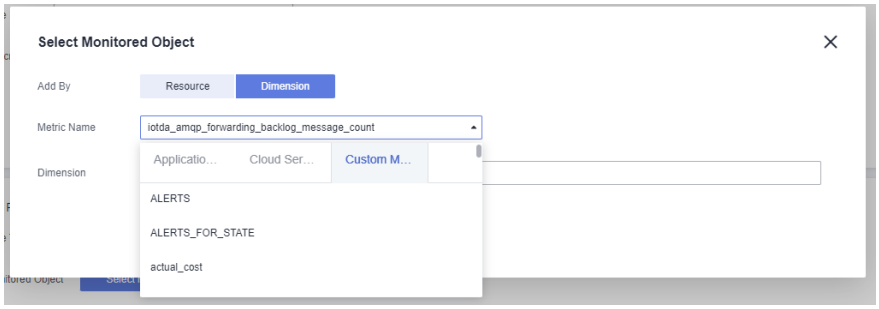


Table 7-13 Metric description

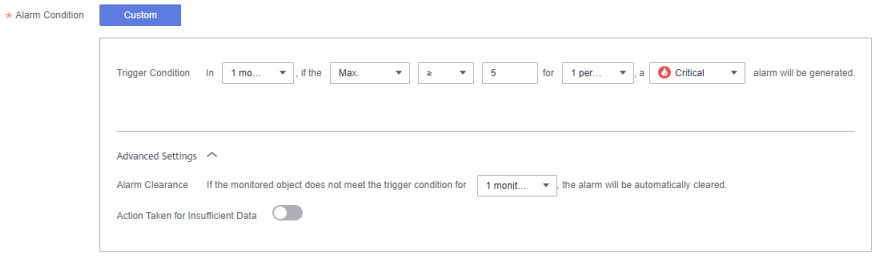
Metric Name	Description
iotda_amqp_forwarding_backlog_message_count	Number of stacked messages in the queue.
iotda_amqp_forwarding_consume_rate	Queue message consumption speed.

Table 7-14 Metric dimension description

Dimension	Description
clusterId	Cluster ID.
namespace	Namespace. The value is fixed to **AOM.IoTDA**.
queueName	AMQP queue name.
userName	Username.

Step 6 Set an alarm condition based on the site requirements.

Figure 7-50 Alarm condition



Step 7 Select an alarm tag. If you want to view the alarm on the **Device Alarms** page in IoTDA, configure the following custom tag.

Table 7-15 Custom tag

Tag Name	Tag Value
resource_provider	IoTDA

Figure 7-51 Current alarms - AMQP alarms

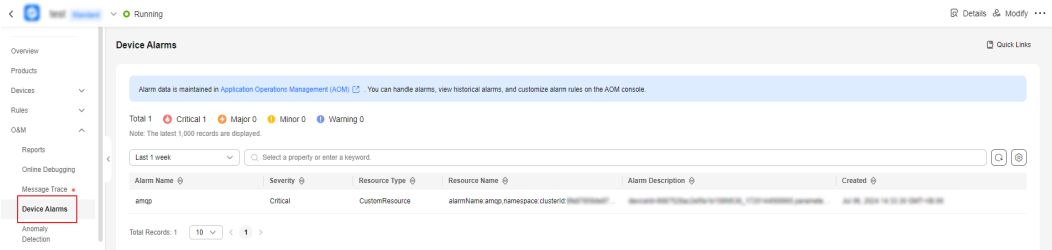
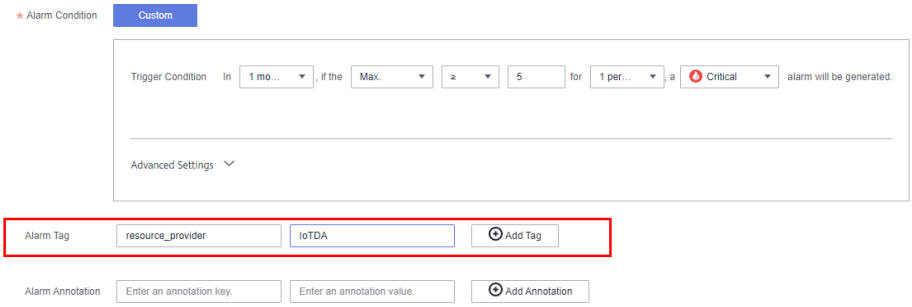
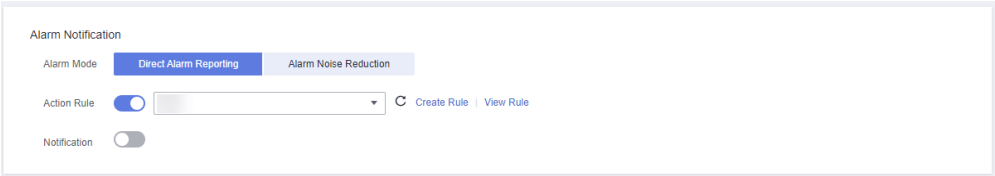


Figure 7-52 Adding a custom tag



Step 8 Specify an action rule for alarm notifications. When an alarm is triggered, the action rule notifies topic subscribers of the alarm through different channels (for example, emails or SMSs) based on the topic. For details, see **Creating an Alarm Action Rule**.

Figure 7-53 Alarm notification



Step 9 Click **Create Now**.

----End

7.6.3.4 AMQP Client Access

After configuring and activating rules by calling the platform APIs **Creating a Rule Triggering Condition**, **Creating a Rule Action**, and **Modifying a Rule Triggering Condition**, connect the AMQP client to IoTDA. Then run the AMQP client on your server to receive subscribed-to messages.

Protocol Version

For details on AMQP, see **AMQP**.

The IoT platform supports only AMQP 1.0.

Connection Establishment and Authentication

1. The AMQP client establishes a TCP connection with the platform and performs TLS handshake verification.

   NOTE

   To ensure security, the AMQP client must use TLS 1.2 for encryption. Non-encrypted TCP transmission is not supported. The difference between the client time and standard time cannot be greater than 5 minutes. Otherwise, the connection will fail.

2. The client requests to set up a connection.
3. The client sends a request to the platform to establish a receiver link (a unidirectional channel for the platform to push data to the client). The receiver link must be set up within 15 seconds after the connection is set up on the client. Otherwise, the platform will close the connection. After the receiver link is set up, the client is connected to the platform.

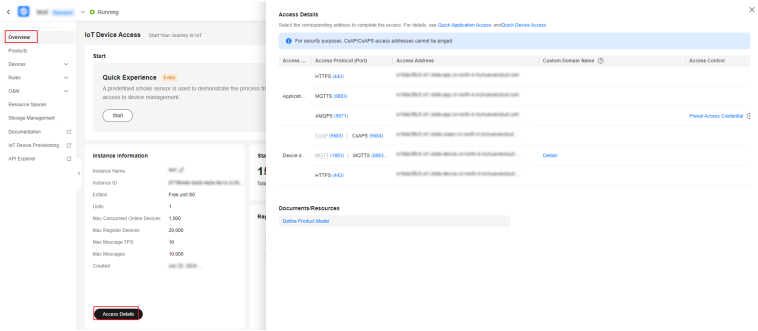
   NOTE

   A maximum of 10 receiver links can be created for a connection, and sender links cannot be created. Therefore, the platform can push messages to the client, but the client cannot send messages to the platform.

Connection Configuration Parameters

The table below describes the connection address and connection authentication parameters for the AMQP client to connect to the platform.

- AMQP access address: amqps://${server.address}:5671
- Connection string: amqps://${server.address}:5671?amqp.vhost=default&amqp.idleTimeout=8000&amqp.saslMechanisms=PLAIN

Parameter	Description
server.address	AMQP server access address. Obtaining method: Log in to the **console**, choose **IoTDA Instances**, and click the target instance card. In the navigation pane, choose **Overview**. Click **Access Details** in the **Instance Information** area, and check the AMQPS access address. **Figure 7-54** Obtaining access information 
amqp.vhost	Currently, AMQP uses the default host. Only the default host is supported.
amqp.saslMechanisms	Connection authentication mode. Currently, PLAIN-SASL is supported.
amqp.idleTimeout	Heartbeat interval, in milliseconds. If the heartbeat interval expires and no frame is transmitted on the connection, the platform closes the connection.

- Port: 5671
- Client identity authentication parameters

  username ="accessKey=${accessKey}|timestamp=${timestamp}|instanceId=${instanceId}"

  password = "${accessCode}"

Parameter	Mandatory or Optional	Description
accessKey	Mandatory	An accessKey can be used to establish a maximum of 32 concurrent connections. When establishing a connection for the first time, preset the parameter by following the instructions provided in **Obtaining the AMQP Access Credential**.

Parameter	Mandatory or Optional	Description
timestamp	Mandatory	Indicates the current time. The value is a 13-digit timestamp, accurate to milliseconds. The server verifies the client timestamp. There is a 5-minute difference between the client timestamp and server timestamp.
instanceId	Optional	Instance ID. This parameter is mandatory when multiple instances of the standard edition are purchased in the same region. For details, see **Viewing Instance Details**.
accessCode	Mandatory	The value can contain a maximum of 256 characters. When establishing a connection for the first time, preset the parameter by following the instructions provided in **Obtaining the AMQP Access Credential**. If the accessCode is lost, you can call the API **Generating an Access Credential** or follow the instructions provided in **Obtaining the AMQP Access Credential** to reset the accessCode.

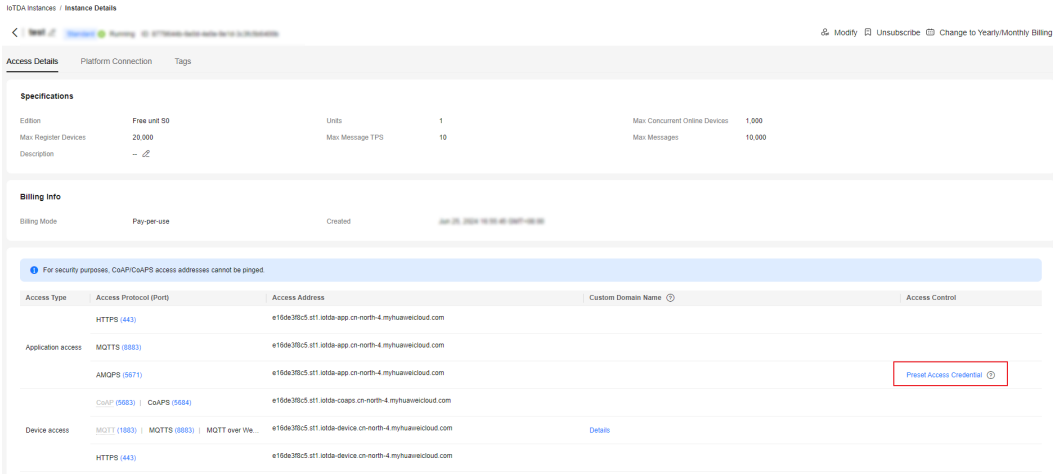
Obtaining the AMQP Access Credential

If an application uses AMQP to access the platform for data transfer, preset an access credential. You can call the API **Generating an Access Credential** or use the console to preset an access credential. The procedure for using the console to generate an access credential is as follows:

Step 1 Choose **IoTDA Instances**, select the edition of your instance, and click **Details** to go to the instance details page.

Step 2 Click **Preset Access Credential** to preset the accessCode and accessKey.

Figure 7-55 Instance management - Preset access credential



NOTE

If you already have an access credential, the accessKey cannot be used after you preset the access credential again.

----End

Connection Specifications

Key	Documentation
Maximum number of queues that can be subscribed for a connection	10
Maximum number of queues for a user	100
Maximum number of connections for a tenant	32
Cache duration of a message (days)	1

Receiving Push Messages

After the receiver link between the client and platform is established, the client can proactively pull data or register a listener to enable the platform to push data. The proactive mode is recommended, because the client can pull data based on its own capability.

7.6.3.5 Java SDK Access Example

This topic describes how to connect an AMQP-compliant JMS client to the IoT platform and receive subscribed messages from the platform.

Development Environment Requirements

JDK 1.8 or later has been installed.

Obtaining the Java SDK

The AMQP SDK is an open-source SDK. If you use Java, you are advised to use the Apache Qpid JMS client. Visit **Qpid JMS** to download the client and view the instructions for use.

Adding a Maven Dependency

```
<!-- amqp 1.0 qpid client -->
 <dependency>
   <groupId>org.apache.qpid</groupId>
   <artifactId>qpid-jms-client</artifactId>
   <version>0.61.0</version>
 </dependency>
```

Sample Code

You can click **here** to obtain the Java SDK access example. For details on the parameters involved in the demo, see **AMQP Client Access**.

CAUTION

All sample code contains the logic of server reconnection upon disconnection.

You can obtain **AmqpClient.java**, **AmqpClientOptions.java**, and **AmqpConstants.java** used in the sample code from **here**.

1. Create **AmqpClient**.

```
// Change the values of the following parameters as required.
AmqpClientOptions options = AmqpClientOptions.builder()
            .host(AmqpConstants.HOST)
            .port(AmqpConstants.PORT)
            .accessKey(AmqpConstants.ACCESS_KEY)
            .accessCode(AmqpConstants.ACCESS_CODE)
            .queuePrefetch(1000) // The SDK allocates the queue with the memory size set in this parameter
to receive messages. If the client memory size is small, set this parameter to a smaller value.
            .build();
      AmqpClient amqpClient = new AmqpClient(options);
      amqpClient.initialize();
```

2. Configure a listener to consume AMQP messages.

```
try {
    MessageConsumer consumer = amqpClient.newConsumer(AmqpConstants.DEFAULT_QUEUE);
    consumer.setMessageListener(message -> {
        try {
            // Process messages. If the processing is time-consuming, you are advised to start a new thread.
Otherwise, the connection may be cut off due to heartbeat timeout.
            processMessage(message.getBody(String.class));
            // If options.isAutoAcknowledge==false is configured, call message.acknowledge();
        } catch (Exception e) {
            log.warn("message.getBody error,exception is ", e);
        }
    });
} catch (Exception e) {
    log.warn("Consumer initialize error,", e);
}
```

3. Pull AMQP messages.

```
 // Create a thread pool to pull messages.
 ExecutorService executorService = new ThreadPoolExecutor(1, 1, 60, TimeUnit.SECONDS,new
LinkedBlockingQueue<>(1));

     try {
          MessageConsumer consumer = amqpClient.newConsumer(AmqpConstants.DEFAULT_QUEUE);
          executorService.execute(() -> {
              while (!isClose.get()) {
                  try {
                      Message message = consumer.receive();
                      // Process messages. If the processing is time-consuming, you are advised to start a new
thread. Otherwise, the connection may be cut off due to heartbeat timeout.
                      processMessage(message.getBody(String.class));
                      // If options.isAutoAcknowledge==false is configured, call message.acknowledge();
                  } catch (JMSException e) {
                      log.warn("receive message error,", e);
                  }
              }
          });
```

```
    } catch (Exception e) {
        log.warn("Consumer initialize error,", e);
    }
```

4. For more demos about AMQP message consumption, see the access demo project that uses the java SDK.

Resources

AmqpClient.java

```
package com.iot.amqp;

import lombok.extern.slf4j.Slf4j;
import org.apache.commons.lang3.StringUtils;
import org.apache.qpid.jms.JmsConnection;
import org.apache.qpid.jms.JmsConnectionExtensions;
import org.apache.qpid.jms.JmsConnectionFactory;
import org.apache.qpid.jms.JmsQueue;
import org.apache.qpid.jms.transports.TransportOptions;
import org.apache.qpid.jms.transports.TransportSupport;

import javax.jms.Connection;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;

@Slf4j
public class AmqpClient {
    private final AmqpClientOptions options;
    private Connection connection;
    private Session session;
    private final Set<MessageConsumer> consumerSet = Collections.synchronizedSet(new HashSet<>());

    public AmqpClient(AmqpClientOptions options) {
        this.options = options;
    }

    public String getId() {
        return options.getClientId();
    }

    public void initialize() throws Exception {
        String connectionUrl = options.generateConnectUrl();
        log.info("connectionUrl={}", connectionUrl);
        JmsConnectionFactory cf = new JmsConnectionFactory(connectionUrl);
        // Trust the server.
        TransportOptions to = new TransportOptions();
        to.setTrustAll(true);
        cf.setSslContext(TransportSupport.createJdkSslContext(to));
        String userName = "accessKey=" + options.getAccessKey();
        cf.setExtension(JmsConnectionExtensions.USERNAME_OVERRIDE.toString(), (connection, uri) -> {
            String newUserName = userName;
            if (connection instanceof JmsConnection) {
                newUserName = ((JmsConnection) connection).getUsername();
            }
            if (StringUtils.isEmpty(options.getInstanceId())) {
            // userName of IoTDA is in the following format: accessKey=${accessKey}|timestamp=${timestamp}.
                return newUserName + "|timestamp=" + System.currentTimeMillis();
            } else {
            //If multiple Standard Editions are purchased in the same region, the format of userName is
accessKey=${accessKey}|timestamp=${timestamp}|instanceId=${instanceId}.
                return newUserName + "|timestamp=" + System.currentTimeMillis() + "|instanceId=" +
options.getInstanceId();
            }
```

```
        });
        // Create a connection.
        connection = cf.createConnection(userName, options.getAccessCode());
        // Create sessions Session.CLIENT_ACKNOWLEDGE and Session.AUTO_ACKNOWLEDGE. For
Session.CLIENT_ACKNOWLEDGE, manually call message.acknowledge() after receiving a message. For
Session.AUTO_ACKNOWLEDGE, the SDK automatically responds with an ACK message (recommended).
        session = connection.createSession(false, options.isAutoAcknowledge() ?
Session.AUTO_ACKNOWLEDGE : Session.CLIENT_ACKNOWLEDGE);
        connection.start();
    }

    public MessageConsumer newConsumer(String queueName) throws Exception {
        if (connection == null || !(connection instanceof JmsConnection) || ((JmsConnection)
connection).isClosed()) {
            throw new Exception("create consumer failed,the connection is disconnected.");
        }
        MessageConsumer consumer;

        consumer = session.createConsumer(new JmsQueue(queueName));
        if (consumer != null) {
            consumerSet.add(consumer);
        }
        return consumer;
    }

    public void close() {
        consumerSet.forEach(consumer -> {
            try {
                consumer.close();
            } catch (JMSException e) {
                log.warn("consumer close error,exception is ", e);
            }
        });

        if (session != null) {
            try {
                session.close();
            } catch (JMSException e) {
                log.warn("session close error,exception is ", e);
            }
        }

        if (connection != null) {
            try {
                connection.close();
            } catch (JMSException e) {
                log.warn("connection close error,exception is", e);
            }
        }
    }
}
```

AmqpClientOptions.java

```
package com.iot.amqp;

import lombok.Builder;
import lombok.Data;
import org.apache.commons.lang3.StringUtils;

import java.text.MessageFormat;
import java.util.HashMap;
import java.util.Map;
import java.util.UUID;
import java.util.stream.Collectors;

@Data
@Builder
public class AmqpClientOptions {
```

```
    private String host;
    @Builder.Default
    private int port = 5671;
    private String accessKey;
    private String accessCode;
    private String clientId;

    /**
     * Specifies the instance ID. This parameter is required when multiple instances of the Standard Edition
are purchased in the same region.
     */
    private String instanceId;

    /**
     * Only true is supported.
     */
    @Builder.Default
    private boolean useSsl = true;

    /**
     * IoTDA supports default only.
     */
    @Builder.Default
    private String vhost = "default";

    /**
     * IoTDA supports PLAIN only.
     */
    @Builder.Default
    private String saslMechanisms = "PLAIN";

    /**
     * true: The SDK automatically responds with an ACK message (default).
     * false: After receiving a message, manually call message.acknowledge().
     */
    @Builder.Default
    private boolean isAutoAcknowledge = true;

    /**
     * Reconnection delay (ms)
     */
    @Builder.Default
    private long reconnectDelay = 3000L;

    /**
     * Maximum reconnection delay (ms). The reconnection delay increases with the number of reconnection
times.
     */
    @Builder.Default
    private long maxReconnectDelay = 30 * 1000L;

    /**
     * Maximum number of reconnection times. The default value is –1, indicating that the number of
reconnection times is not limited.
     */
    @Builder.Default
    private long maxReconnectAttempts = -1;

    /**
     * Idle timeout interval. If the peer end does not send an AMQP frame within the interval, the connection
will be cut off. The default value is 30000, in milliseconds.
     */
    @Builder.Default
    private long idleTimeout = 30 * 1000L;

    /**
     * The values below control how many messages the remote peer can send to the client and be held in a
pre-fetch buffer for each consumer instance.
     */
```

```
    @Builder.Default
    private int queuePrefetch = 1000;

    /**
     * Extended parameters
     */
    private Map<String, String> extendedOptions;

    public String generateConnectUrl() {
        String uri = MessageFormat.format("{0}://{1}:{2}", (useSsl ? "amqps" : "amqp"), host,
String.valueOf(port));
        Map<String, String> uriOptions = new HashMap<>();
        uriOptions.put("amqp.vhost", vhost);
        uriOptions.put("amqp.idleTimeout", String.valueOf(idleTimeout));
        uriOptions.put("amqp.saslMechanisms", saslMechanisms);

        Map<String, String> jmsOptions = new HashMap<>();
        jmsOptions.put("jms.prefetchPolicy.queuePrefetch", String.valueOf(queuePrefetch));
        if (StringUtils.isNotEmpty(clientId)) {
            jmsOptions.put("jms.clientID", clientId);
        } else {
            jmsOptions.put("jms.clientID", UUID.randomUUID().toString());
        }
        jmsOptions.put("failover.reconnectDelay", String.valueOf(reconnectDelay));
        jmsOptions.put("failover.maxReconnectDelay", String.valueOf(maxReconnectDelay));
        if (maxReconnectAttempts > 0) {
            jmsOptions.put("failover.maxReconnectAttempts", String.valueOf(maxReconnectAttempts));
        }
        if (extendedOptions != null) {
            for (Map.Entry<String, String> option : extendedOptions.entrySet()) {
                if (option.getKey().startsWith("amqp.") || option.getKey().startsWith("transport.")) {
                    uriOptions.put(option.getKey(), option.getValue());
                } else {
                    jmsOptions.put(option.getKey(), option.getValue());
                }
            }
        }
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append(uriOptions.entrySet().stream()
                .map(option -> MessageFormat.format("{0}={1}", option.getKey(), option.getValue()))
                .collect(Collectors.joining("&", "failover:(" + uri + "?", ")")));
        stringBuilder.append(jmsOptions.entrySet().stream()
                .map(option -> MessageFormat.format("{0}={1}", option.getKey(), option.getValue()))
                .collect(Collectors.joining("&", "?", "")));
        return stringBuilder.toString();
    }
}
```

AmqpConstants.java

```
package com.iot.amqp;


public interface AmqpConstants {
    /**
     * AMQP access domain name
     * eg: "****.iot-amqps.cn-north-4.myhuaweicloud.com";
     */
    String HOST = "127.0.0.1";

    /**
     * AMQP access port
     */
    int PORT = 5671;

    /**
     * Access key
     * A timestamp does not need to be combined.
     */
```

```
    String ACCESS_KEY = "accessKey";

    /**
     * Access code
     */
    String ACCESS_CODE = "accessCode";

    /**
     * Default queue
     */
    String DEFAULT_QUEUE = "DefaultQueue";
}
```

7.6.3.6 Node.js SDK Access Example

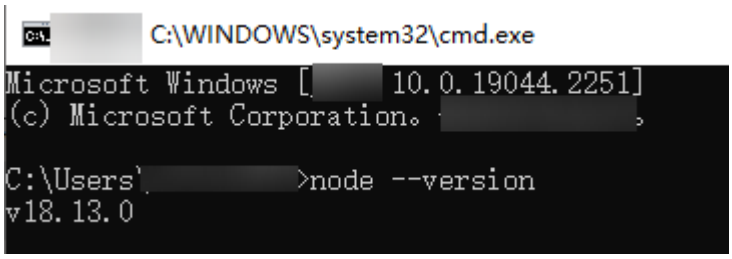
This topic describes how to use a Node.js AMQP SDK to connect to the Huawei Cloud IoT platform and receive subscribed messages from the platform.

Development Environment

Node.js 8.0.0 or later is used. **Download** it from the Node.js official website. After installation, run the following command to check the version:

```
node --version
```

If the version is displayed and is later than 8.0.0, the installation is successful.



Sample Code

1. Create a JavaScript file (for example, **HwIotAmqpClient.js**) on the local computer and save the following sample code to the file. Modify related connection parameters by referring to **AMQP Client Access**.

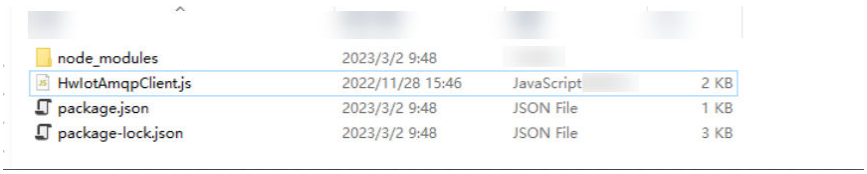
```
const container = require('rhea');
// Obtain the timestamp.
var timestamp = Math.round(new Date());

// Set up a connection.
var connection = container.connect({
    // Access domain name. For details, see AMQP Client Access.
    'host': '${UUCID}.iot-amqps.cn-north-4.myhuaweicloud.com',
    'port': 5671,
    'transport': 'tls',
    'reconnect': true,
    'idle_time_out': 8000,
    // Method to assemble username. For details, see AMQP Client Access.
     'username': 'accessKey=${yourAccessKey}|timestamp=' + timestamp + '|instanceId=${instanceId}',
    // accessCode. For details, see AMQP Client Access.
    'password': '${yourAccessCode}',
    'saslMechannisms': 'PLAIN',
    'rejectUnauthorized': false,
    'hostname': 'default',
});

// Create a Receiver connection. You can use DefaultQueue.
```

```
var receiver = connection.open_receiver('${yourQueue}');

// Callback function for receiving messages pushed from the cloud
container.on('message', function (context) {
    var msg = context.message;
    var content = msg.body;
    console.log(content);
    // Send an ACK message. Note that the callback function should not contain time-consuming logic.
    context.delivery.accept();
 });
```

2. Go to the directory where the **HwIotAmqpClient.js** file is located and run the following command to install the rhea library:

```
npm install rhea
```

After installation, the project files shown in the following figure are displayed.



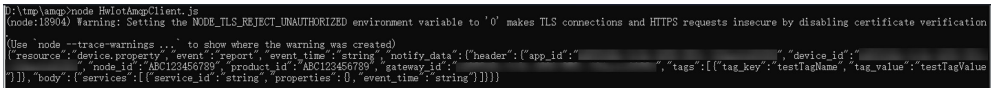
3. Run the following command to start the AMQP client:

```
node HwIotAmqpClient.js
```

4. Running result examples

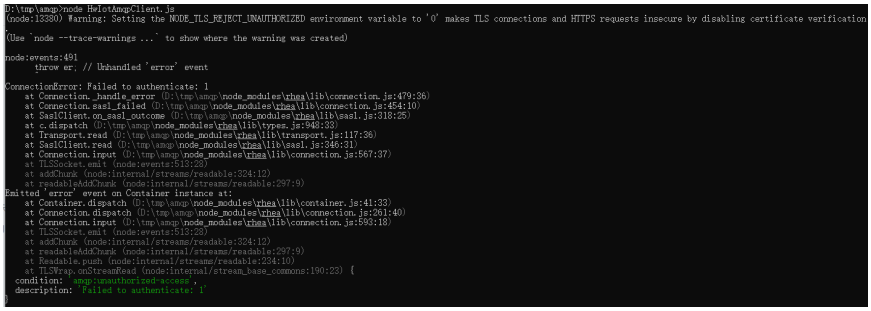
- Successful subscription:

  If the following log information is displayed, the AMQP client successfully subscribes to the IoT platform and obtains data from the platform.



- Failed subscription:

  The following log indicates that the AMQP client authentication fails on the IoT platform. Check whether the access code is correct and whether the difference between the timestamp and the standard time is greater than 5 minutes.



7.6.3.7 C# SDK Access Example

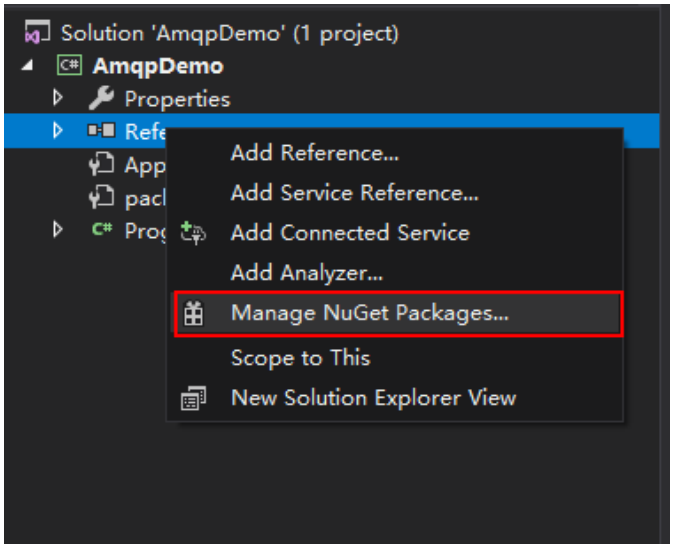
This topic describes how to connect an AMQP.Net Lite client to the IoT platform and receive subscribed messages from the platform.

Requirements for the Development Environment

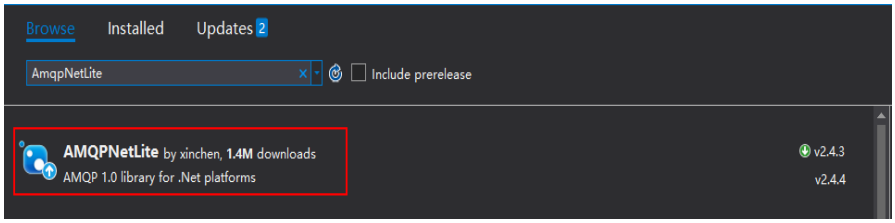
.NET Framework 4.6 or later has been installed.

Obtaining the Java SDK

1. Right-click the project directory and choose **Manage NuGet Packages**.



2. In the NuGet manager, search for **AmqpNetLite** and install the required version.



Sample Code

For details about the parameters in the demo, see **AMQP Client Access**.

```
using Amqp;
using Amqp.Framing;
using Amqp.Sasl;
using System;
using System.Threading;

namespace AmqpDemo
{
    class Program
    {
        /// <summary>
        /// Access domain name. For details, see "AMQP Client Access".
        /// See Connection Configuration Parameters.
        /// </summary>
        static string Host = "${Host}";

        /// <summary>
        /// Port
        /// </summary>
        static int Port = 5671;

        /// <summary>
        /// Access key
        /// </summary>
        static string AccessKey = "${YourAccessKey}";

        /// <summary>
```

```
        /// Access code
        /// </summary>
        static string AccessCode = "${yourAccessCode}";

        /// <summary>
        /// Instance ID. This parameter is required when multiple instances of the Standard Edition are
purchased in the same region.
        /// </summary>
        static string InstanceId = "${instanceId}";

        /// <summary>
        /// Queue name
        /// </summary>
        static string QueueName = "${yourQueue}";

        static Connection connection;

        static Session session;

        static ReceiverLink receiverLink;

        static DateTime lastConnectTime = DateTime.Now;

        static void Main(string[] args)
        {
            try
            {
                connection = CreateConnection();
                // Add a connection exception callback.
                connection.AddClosedCallback(ConnectionClosed);

                // Create a session.
                var session = new Session(connection);

                // Create a receiver link.
                receiverLink = new ReceiverLink(session, "receiverName", QueueName);

                // Receive a message.
                ReceiveMessage(receiverLink);
            }
            catch (Exception e)
            {
                Console.WriteLine(e);
            }

            // Press Enter to exit the program.
            Console.ReadLine();

            ShutDown();
        }

        /// <summary>
        /// Create a connection.
        /// </summary>
        /// <returns>Connection</returns>
        static Connection CreateConnection()
        {
            lastConnectTime = DateTime.Now;
            long timestamp = new DateTimeOffset(DateTime.UtcNow).ToUnixTimeMilliseconds();
            string userName = "accessKey=" + AccessKey + "|timestamp=" + timestamp + "|instanceId=" +
InstanceId;
            Address address = new Address(Host, Port, userName, AccessCode);
            ConnectionFactory factory = new ConnectionFactory();
            factory.SASL.Profile = SaslProfile.External;
            // Trust the server and skip certificate verification.
            factory.SSL.RemoteCertificateValidationCallback = (sender, certificate, chain, sslPolicyError) =>
{ return true; };
            factory.AMQP.IdleTimeout = 8000;
            factory.AMQP.MaxFrameSize = 8 * 1024;
```

```
        factory.AMQP.HostName = "default";
        var connection = factory.CreateAsync(address).Result;
        return connection;
    }

    static void ReceiveMessage(ReceiverLink receiver)
    {
        receiver.Start(20, (link, message) =>
        {
            // Process the message in the thread pool to prevent the thread that pulls the message from
being blocked.
            ThreadPool.QueueUserWorkItem((obj) => ProcessMessage(obj), message);
            // Return an ACK message.
            link.Accept(message);
        });
    }


    static void ProcessMessage(Object obj)
    {
        if (obj is Message message)
        {
            string body = message.Body.ToString();
            Console.WriteLine("receive message, body=" + body);
        }
    }


    static void ConnectionClosed(IAmqpObject amqpObject, Error e)
    {
        // Reconnection upon disconnection
        ThreadPool.QueueUserWorkItem((obj) =>
        {
            ShutDown();
            int times = 0;
            while (times++ < 5)
            {
                try
                {
                    Thread.Sleep(1000);
                    connection = CreateConnection();
            // Add a connection exception callback.
                    connection.AddClosedCallback(ConnectionClosed);

            // Create a session.
                    session = new Session(connection);

                    // Create a receiver link.
                    receiverLink = new ReceiverLink(session, "receiverName", QueueName);

            // Receive a message.
                    ReceiveMessage(receiverLink);
                    break;
                }
                catch (Exception exception)
                {
                    Console.WriteLine("reconnect error, exception =" + exception);
                }
            }
        });
    }

    static void ShutDown()
    {
        if (receiverLink != null)
        {
            try
            {
                receiverLink.Close();
```

```
            }
            catch (Exception e)
            {
                Console.WriteLine("close receiverLink error, exception =" + e);
            }

        }
        if (session != null)
        {
            try
            {
                session.Close();
            }
            catch (Exception e)
            {
                Console.WriteLine("close session error, exception =" + e);
            }

        }

        if (connection != null)
        {
            try
            {
                connection.Close();
            }
            catch (Exception e)
            {
                Console.WriteLine("close connection error, exception =" + e);
            }

        }
    }
  }
}
```

7.6.3.8 Android SDK Access Example

This topic describes how to use AMQP to connect the Android system to the IoT platform and receive subscribed messages from the platform.

Preparations

- Install Android Studio. Go to the **Android Studio website** to download and install a desired version. The following uses Android Studio 4.1.1 running on 64-bit Windows as an example.

  Figure 7-56 Downloading Android Studio

  Android Studio downloads

Platform	Android Studio package	Size	SHA-256 checksum
Windows (64-bit)	android-studio-ide-192.6392135-windows.exe Recommended	756 MB	07b6df807fda59e69f05b85ff6f6bd0c70d09e57fb151197155ef5f115f96e59
	android-studio-ide-192.6392135-windows.zip No .exe installer	770 MB	24f8f9ce467b935c25d89b90cad402d21dd45d4ba9af1ad35baeeb414609e483
Windows (32-bit)	android-studio-ide-192.6392135-windows32.zip No .exe installer	770 MB	7b24742726bbc8b40a55dab1f7cdff923ba384b233c21d35d6e96fa36320d067
Mac (64-bit)	android-studio-ide-192.6392135-mac.dmg	768 MB	c5dd347469be0d995e6b4d74ea72b3a6f2572e72b4eac37a0834b0a0984d9583
Linux (64-bit)	android-studio-ide-192.6392135-linux.tar.gz	772 MB	33ec9f61b20b71ca175cd39083b1379ebba896de78b826ea5df5d440c6adfd2a
Chrome OS	android-studio-ide-192.6392135-cros.deb	653 MB	59023aaabc7d5822fd7b1c5a71589b18e487ca8d7fd4320c3547ee0ad390e4ca

- Install the JDK. You can also use the built-in JDK of the IDE.

a. Go to the **Oracle website** to download a desired version. The following uses JDK 8 for Windows x64 as an example.
b. After the download is complete, run the installation file and install Node.js as prompted.

Development Environment

The development environments used in this example are as follows:

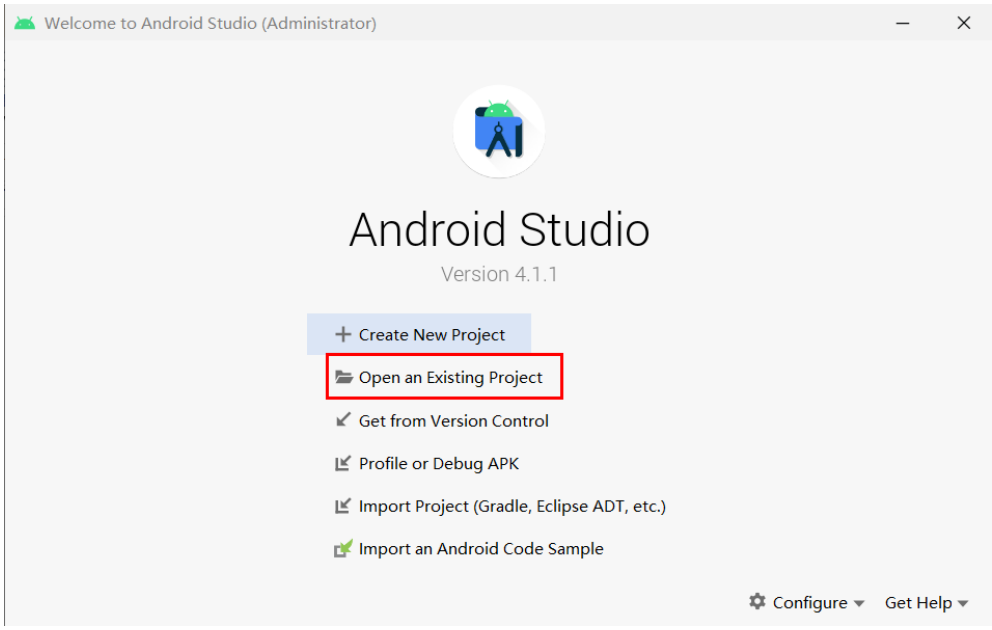
- JDK 1.8 or later
- Android SDK Platform of API level 28 or later
- **Apache Qpid Proton-J** client

Sample Code

Step 1 Download the **AMQP demo**.

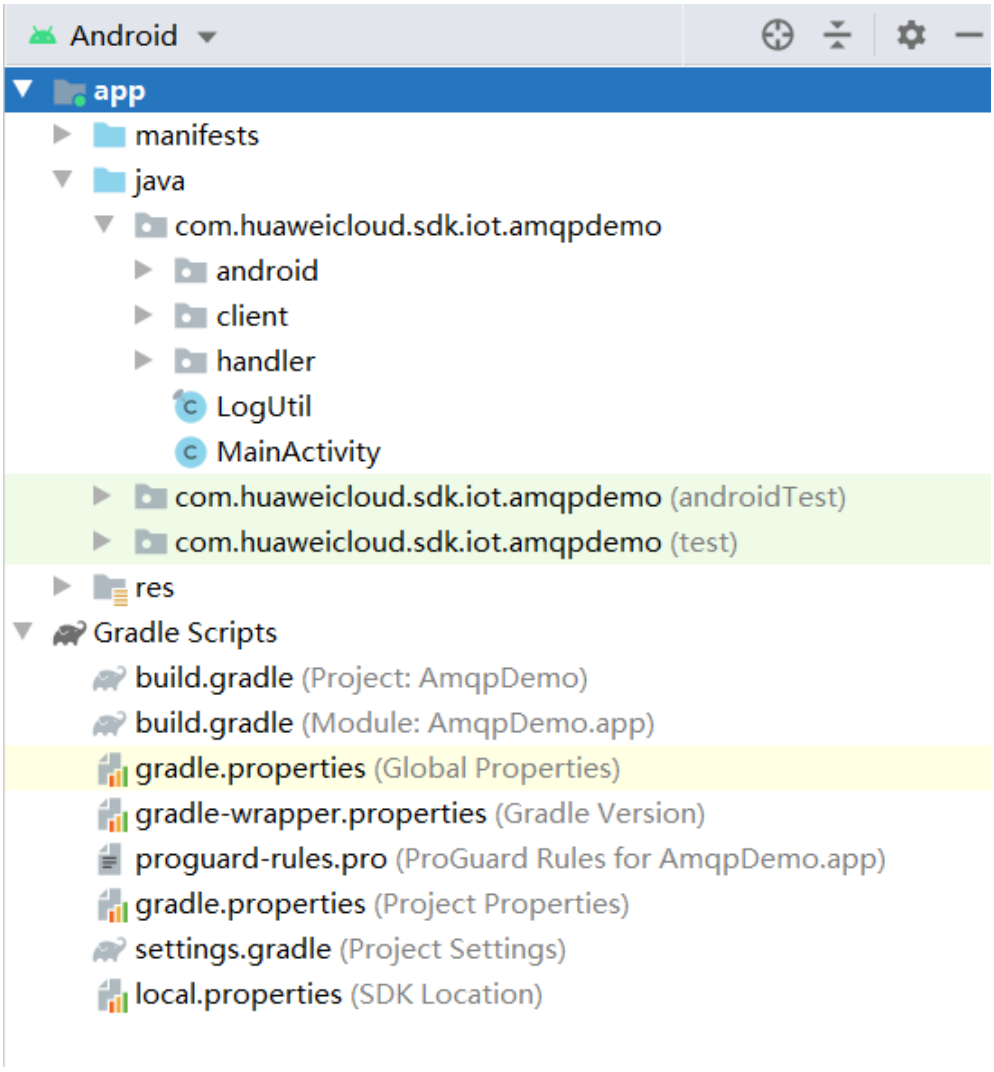
Step 2 Run Android Studio, click **Open**, and select the sample code downloaded in **1**.

Figure 7-57 Incorporate into existing projects



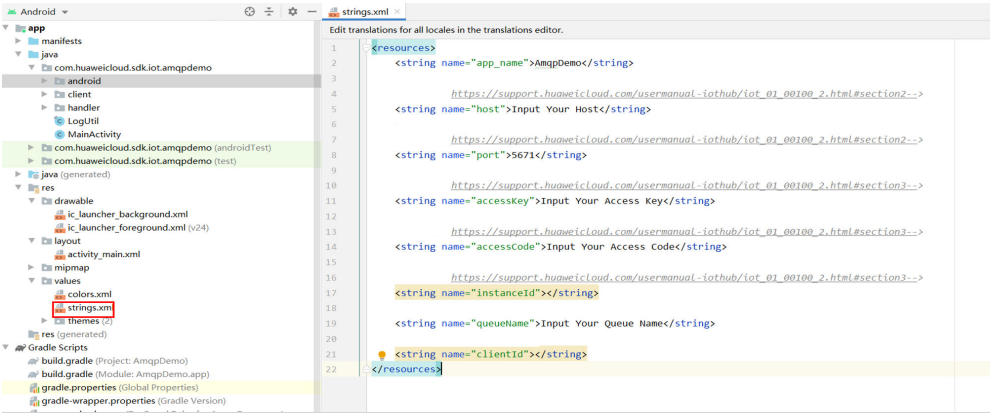
Step 3 Import the sample code.

Figure 7-58 Importing the project structure



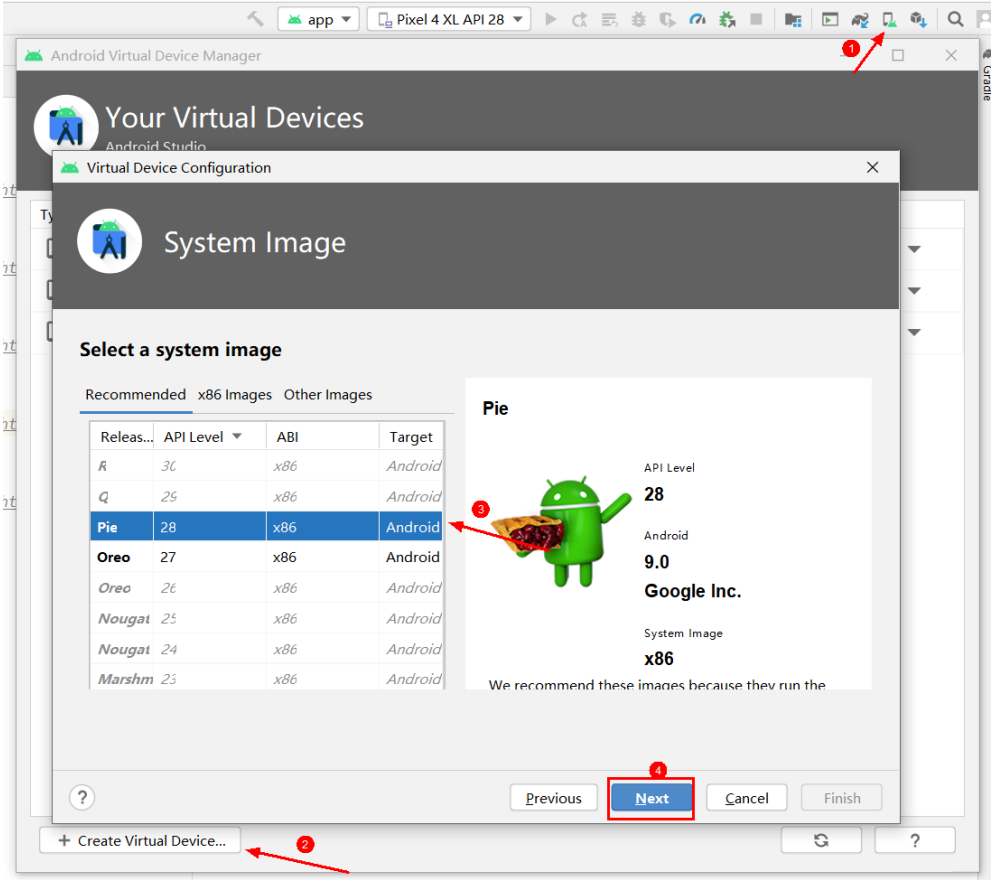
Step 4 (Optional) Set AMQP connection parameters in the **res\values\strings.xml** file in advance. For details, see **AMQP Client Access**.

Figure 7-59 Modifying connection parameters



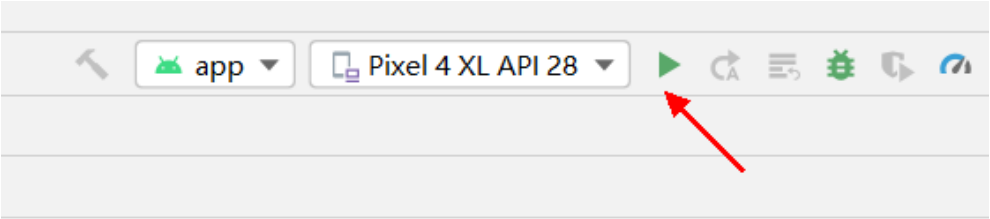
Step 5 In the AVD Manager configuration, select a device model and a virtual device of API level 28 for debugging.

Figure 7-60 Configuring the AVD Manager



Step 6 Start the demo for debugging.

Figure 7-61 Starting the demo

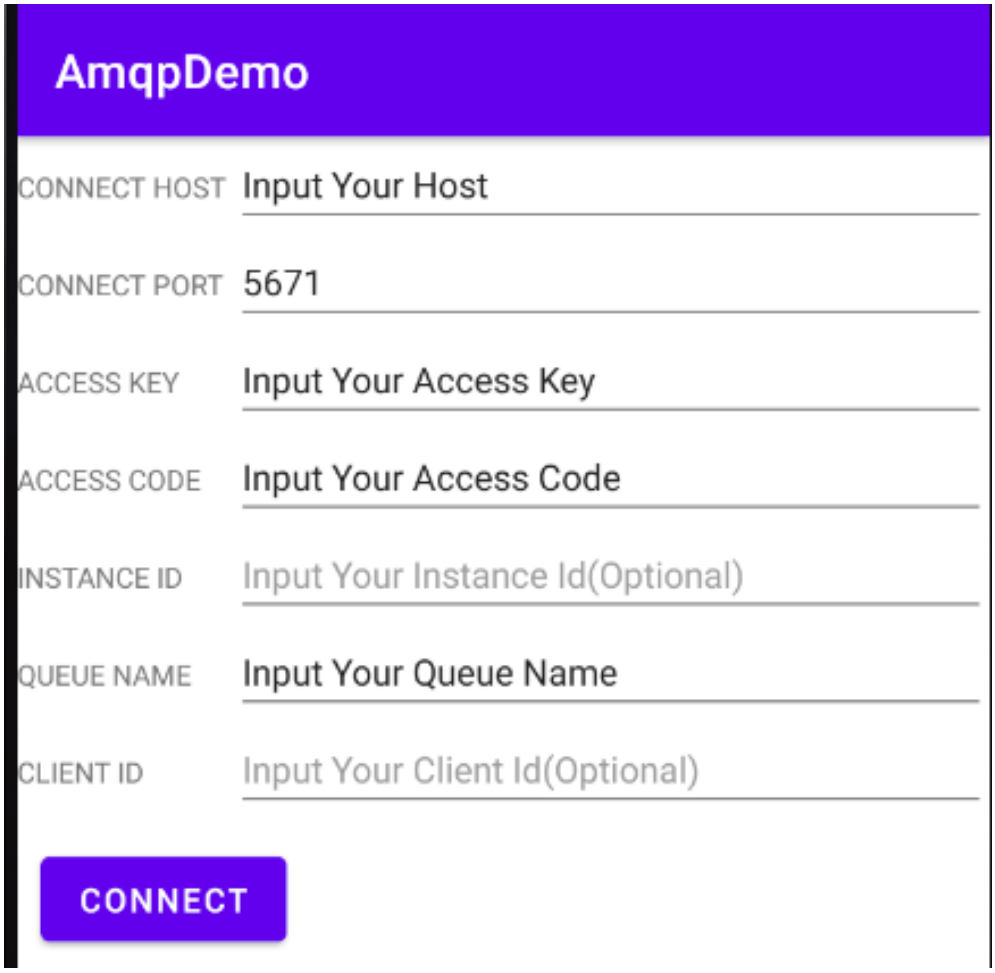


----End

Related Information

AMQP connection configuration page is displayed in the following figure. **INSTANCE ID** is mandatory when multiple standard IoTDA instances are purchased in the same region. For details about the parameters, see **AMQP Client Access**.

Figure 7-62 AMQP connection configuration

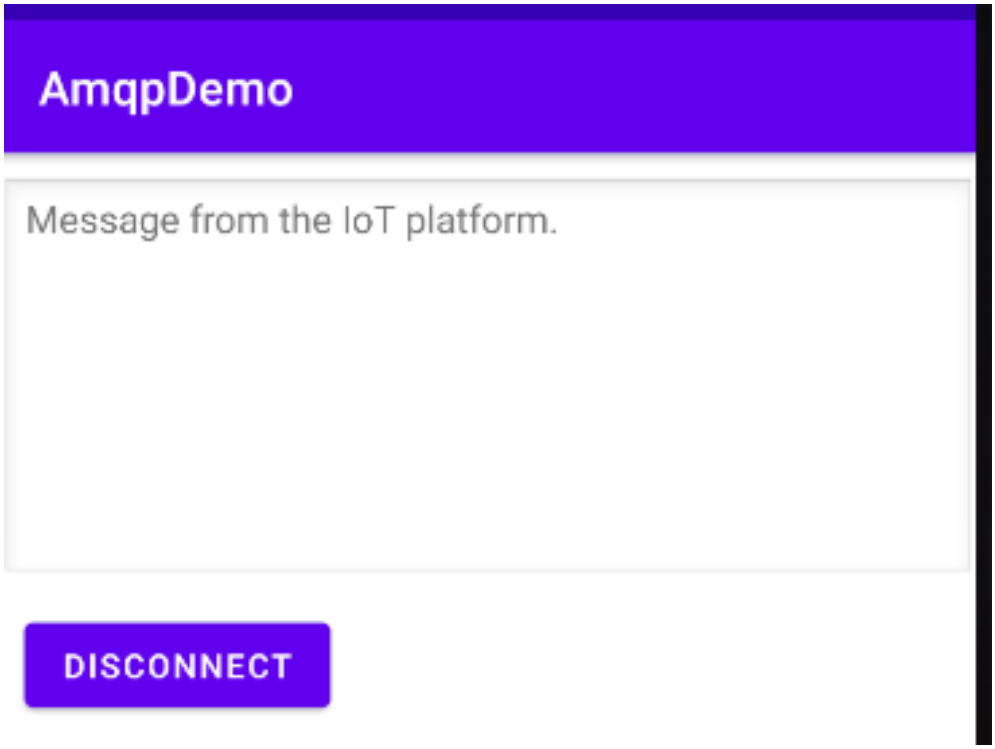


- The following figure shows the page indicating that the connection is successful after the parameter modification.

  Figure 7-63 Connection succeeded

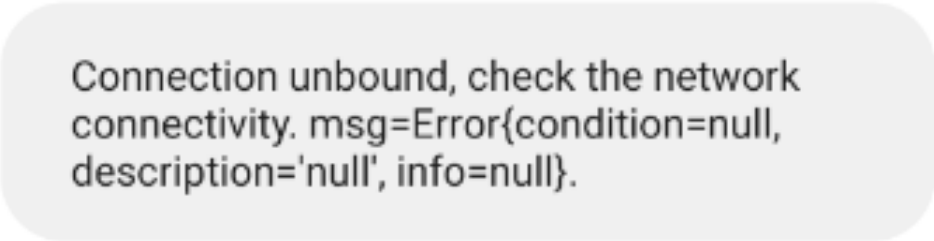
- The following figure shows the page indicating that the transfer data is successfully obtained.

Figure 7-64 Receiving platform messages



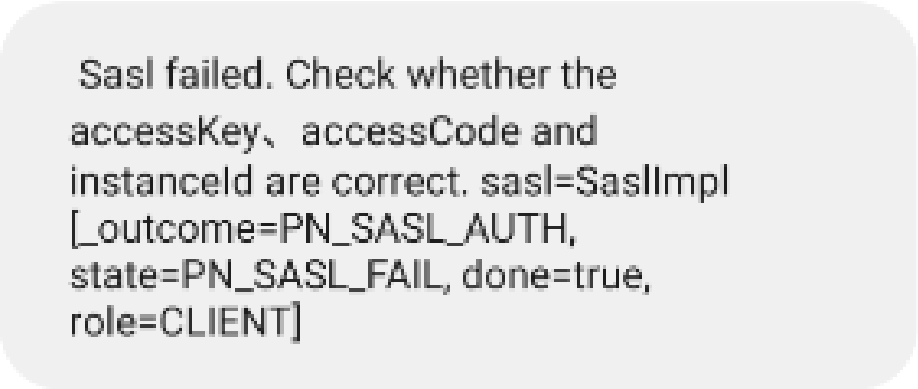
- The following figure shows the page indicating that the network connection failed.

Figure 7-65 Network connection failed



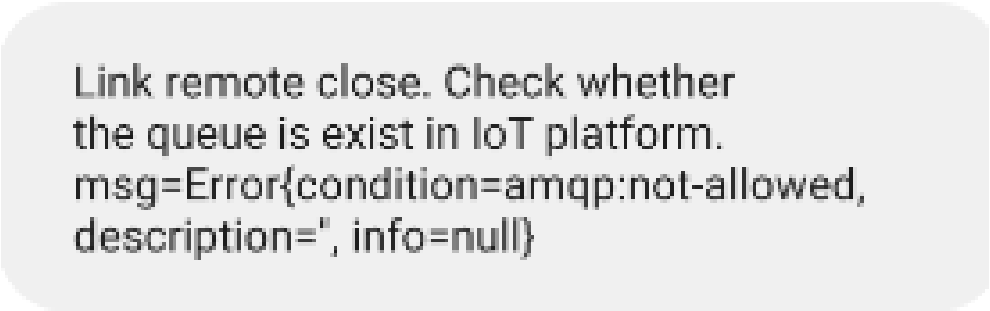
- The following figure shows the page indicating that access information (**accessKey**, **accessCode**, and **instanceId**) is incorrect.

Figure 7-66 Incorrect access information



- The following figure shows the page indicating that the queue does not exist.

Figure 7-67 Queue not existing



7.6.3.9 Python SDK Access Example

This topic describes how to use a Python 3 SDK to connect to the Huawei Cloud IoT platform and receive subscribed messages from the platform based on AMQP.

Development Environment

Python 3.0 or later is required. In this example, Python 3.9 is used.

Downloading the SDK

In this example, AMQP SDK **python-qpid-proton** (version 0.37.0) is used. You can run the following command to install the SDK of the latest version:

```
pip install python-qpid-proton
```

You can also manually install it by referring to **Installing Qpid Proton**.

Sample Code

```
import threading
import time

from proton import SSLDomain
```

```
from proton.handlers import MessagingHandler
from proton.reactor import Container

# Reconnection times
reconnectTimes = 0


def current_time_millis():
    return str(int(round(time.time() * 1000)))


class AmqpClient(MessagingHandler):
    def __init__(self, host, port, accessKey, accessCode, queueName, instanceId):
        super(AmqpClient, self).__init__()
        self.host = host
        self.port = port
        self.accessKey = accessKey
        self.accessCode = accessCode
        self.queueName = queueName
        self.instanceId = instanceId

    def on_start(self, event):
        # Access domain name. For details, see "AMQP Client Access".
        url = "amqps://%s:%s" % (self.host, self.port)

        timestamp = current_time_millis()
        userName = "accessKey=" + self.accessKey + "|timestamp=" + timestamp + "|instanceId=" +
self.instanceId
        passWord = self.accessCode
        # By default, the server certificate is not verified.
        sslDomain = SSLDomain(SSLDomain.MODE_CLIENT)
        sslDomain.set_peer_authentication(SSLDomain.ANONYMOUS_PEER)
        self.conn = event.container.connect(url, user=userName, password=passWord, heartbeat=60,
ssl_domain=sslDomain,
                                            reconnect=False)
        event.container.create_receiver(self.conn, source=self.queueName)

    # Called when the connection is established.
    def on_connection_opened(self, event):
        global reconnectTimes
        reconnectTimes = 0
        print("Connection established, remoteUrl: %s", event.connection.hostname)

    # Called when the connection is cut off.
    def on_connection_closed(self, event):
        print("Connection closed: %s", self)
        ReconnectThread("reconnectThread").start()

    # Called when the remote end is disconnected due to an error.
    def on_connection_error(self, event):
        print("Connection error:%s", self)
        ReconnectThread("reconnectThread").start()

    # Called when an error occurs during AMQP connection establishment. Such errors include
authentication and socket errors.
    def on_transport_error(self, event):
        if event.transport.condition:
            if event.transport.condition.info:
                print("%s: %s: %s" % (event.transport.condition.name, event.transport.condition.description,
                                      event.transport.condition.info))
            else:
                print("%s: %s" % (event.transport.condition.name, event.transport.condition.description))
        else:
            print("Unspecified transport error")
        ReconnectThread("reconnectThread").start()

    # Called when a message is received.
    def on_message(self, event):
        message = event.message
```

```
        content = message.body
        print("receive message: content=%s" % content)


class ReconnectThread(threading.Thread):
    def __init__(self, name):
        threading.Thread.__init__(self)
        self.name = name

    def run(self):
        global reconnectTimes
        reconnectTimes = reconnectTimes + 1
        time.sleep(15 if reconnectTimes > 15 else reconnectTimes)
        Container(AmqpClient(amqpHost, amqpPort, amqpAccessKey, amqpAccessCode, amqpQueueName,
instanceId)).run()


# For details about how to set the following parameters, see Connection Configuration Parameters.
# AMQP access domain name
amqpHost = "127.0.0.1"

# AMQP access port
amqpPort = 5671

# Access key
amqpAccessKey = 'your AccessKey'

# Access code
amqpAccessCode = 'your AccessCode'

# Name of the subscription queue
amqpQueueName = 'DefaultQueue'

# Instance ID. This parameter is mandatory when multiple instances of the standard edition are purchased
in the same region.
instanceId = ''

Container(AmqpClient(amqpHost, amqpPort, amqpAccessKey, amqpAccessCode, amqpQueueName,
instanceId)).run()
```

7.6.3.10 Go SDK Access Example

This topic describes how to use a Go SDK to connect to the Huawei Cloud IoT platform and receive subscribed messages from the platform based on AMQP.

Development Environment Requirements

Go 1.16 or later has been installed.

Adding Dependencies

Add the following dependencies to **go.mod**:

```
require (
    pack.ag/amqp v0.12.5 // v0.12.5 is used in this example. Select a version as required.
)
```

Sample Code

```
package main

import (
    "context"
    "crypto/tls"
    "fmt"
```

```
    "pack.ag/amqp"
    "time"
)

type AmqpClient struct {
    Title       string
    Host        string
    AccessKey  string
    AccessCode string
        InstanceId string
    QueueName  string

    address  string
    userName string
    password string

    client   *amqp.Client
    session  *amqp.Session
    receiver *amqp.Receiver
}

type MessageHandler interface {
    Handle(message *amqp.Message)
}

func (ac *AmqpClient) InitConnect() {
    if ac.QueueName == "" {
        ac.QueueName = "DefaultQueue"
    }
    ac.address = "amqps://" + ac.Host + ":5671"
    ac.userName = fmt.Sprintf("accessKey=%s|timestamp=%d|instanceId=%s", ac.AccessKey,
time.Now().UnixNano()/1000000, ac.InstanceId)
    ac.password = ac.AccessCode
}

func (ac *AmqpClient) StartReceiveMessage(ctx context.Context, handler MessageHandler) {
    childCtx, _ := context.WithCancel(ctx)
    err := ac.generateReceiverWithRetry(childCtx)
    if nil != err {
        return
    }
    defer func() {
        _ = ac.receiver.Close(childCtx)
        _ = ac.session.Close(childCtx)
        _ = ac.client.Close()
    }()

    for {
        // Block message receiving. If ctx is a context created based on the background function, message
receiving will not be blocked.
        message, err := ac.receiver.Receive(ctx)
        if nil == err {
            go handler.Handle(message)
            _ = message.Accept()
        } else {
            fmt.Println("amqp receive data error: ", err)

            // If message receiving is manually disabled, exit the program.
            select {
            case <-childCtx.Done():
                return
            default:
            }

            // If message receiving is not manually disabled, retry the connection.
            err := ac.generateReceiverWithRetry(childCtx)
            if nil != err {
                return
            }
```

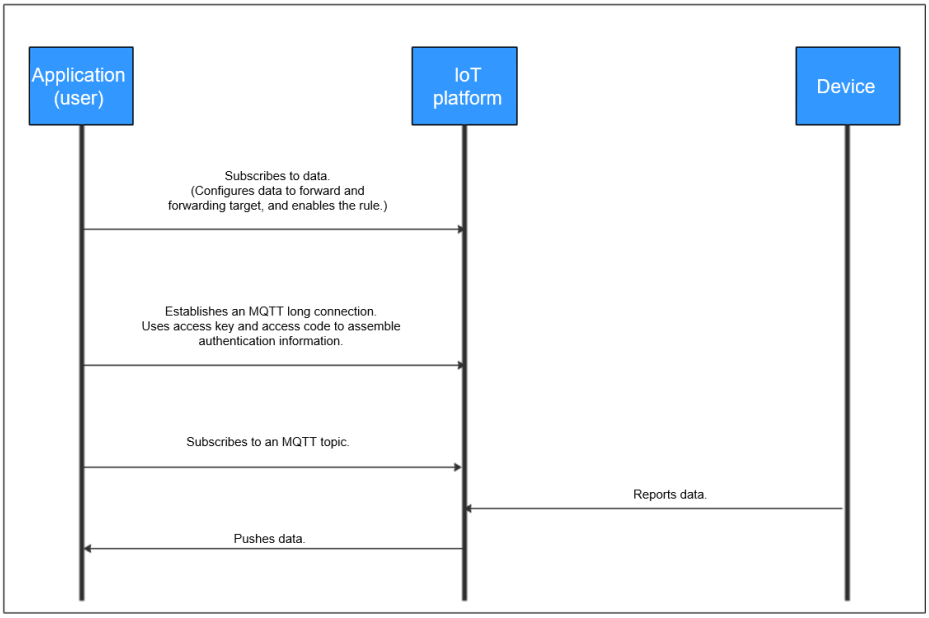
```
        }
    }
}

func (ac *AmqpClient) generateReceiverWithRetry(ctx context.Context) error {
    // Retries with exponential backoff, from 10 ms to 20s.
    duration := 10 * time.Millisecond
    maxDuration := 20000 * time.Millisecond
    times := 1

    // Retries with exponential backoff
    for {
        select {
        case <-ctx.Done():
            return amqp.ErrConnClosed
        default:
        }

        err := ac.generateReceiver()
        if nil != err {
            fmt.Println("amqp ac.generateReceiver error ", err)
            time.Sleep(duration)
            if duration < maxDuration {
                duration *= 2
            }
            fmt.Println("amqp connect retry,times:", times, ",duration:", duration)
            times++
            return nil
        } else {
            fmt.Println("amqp connect init success")
            return nil
        }
    }
}

// The statuses of the connection and session cannot be determined because the packets are unavailable.
Retry the connection to obtain the information.
func (ac *AmqpClient) generateReceiver() error {

    if ac.session != nil {
        receiver, err := ac.session.NewReceiver(
            amqp.LinkSourceAddress(ac.QueueName),
            amqp.LinkCredit(20),
        )
        // If a network disconnection error occurs, the connection is ended and the session fails to be
established. Otherwise, the connection is established.
        if err == nil {
            ac.receiver = receiver
            return nil
        }
    }

    // Delete the previous connection.
    if ac.client != nil {
        _ = ac.client.Close()
    }
        ac.userName = fmt.Sprintf("accessKey=%s|timestamp=%d|instanceId=%s", ac.AccessKey,
time.Now().UnixNano()/1000000, ac.InstanceId)
    fmt.Println("[" + ac.Title + "] Dial... addr=[" + ac.address + "], username=[" + ac.userName + "],
password=[" + ac.password + "]")
    client, err := amqp.Dial(ac.address,
        amqp.ConnSASLPlain(ac.userName, ac.password),
        amqp.ConnProperty("vhost", "default"),
        amqp.ConnServerHostname("default"),
        amqp.ConnTLSConfig(&tls.Config{InsecureSkipVerify: true,
            MaxVersion: tls.VersionTLS12,
        }),
        amqp.ConnConnectTimeout(8*time.Second))
    if err != nil {
```

```
        fmt.Println("Dial", err)
        return err
    }
    ac.client = client
    session, err := client.NewSession()
    if err != nil {
        XDebug("Error: NewSession", err)
        return err
    }
    ac.session = session

    receiver, err := ac.session.NewReceiver(
        amqp.LinkTargetDurability(amqp.DurabilityUnsettledState),
        amqp.LinkSourceAddress(ac.QueueName),
        amqp.LinkCredit(100),
    )
    if err != nil {
        XDebug("Error: NewReceiver", err)
        return err
    }
    ac.receiver = receiver

    return nil
}

func XDebug(s string, err error) {
    fmt.Println(s, err)
}

type CustomerMessageHandler struct {
}

func (c *CustomerMessageHandler) Handle(message *amqp.Message) {
    fmt.Println("AMQP receives messages.", message.Value)
}

func main() {
    // For details about how to set the following parameters, see Connection Configuration Parameters.
    // AMQP access domain name
    amqpHost := "127.0.0.1"

    // Access key
    amqpAccessKey := "your accessKey"

    // Access code
    amqpAccessCode := "your accessCode"

        // Instance ID
    instanceId:= "your intanceId"

    // Name of the subscription queue
    amqpQueueName := "DefaultQueue"

    amqpClient := &AmqpClient{
        Title:      "test",
        Host:       amqpHost,
        AccessKey:  amqpAccessKey,
        AccessCode: amqpAccessCode,
                InstanceId: instanceId,
        QueueName:  amqpQueueName,
    }

    handle := CustomerMessageHandler{}
    amqpClient.InitConnect()
    ctx := context.Background()
    amqpClient.StartReceiveMessage(ctx, &handle)
}
```

7.6.4 MQTT Data Forwarding

7.6.4.1 Overview

The figure below shows the subscription and push process.



Push mechanism: The IoT platform pushes QoS 0 messages to users. If a user does not establish a connection or does not subscribe to the topic after the connection is established, the IoT platform will delete expired data and data that exceeds the capacity limit in rolling mode when the maximum cache duration (24 hours) or maximum cache size (1 GB) is reached or exceeded.

Subscribing to Data

1. You can create a rule and add an MQTT message queue as the forwarding target on the console to subscribe to data. For details, see **MQTT Server Configuration**.
2. Call APIs to subscribe to data. For details, see **Calling APIs**, **Creating a Rule Triggering Condition**, **Creating a Rule Action**, and **Modifying a Rule Triggering Condition**.

Format of Pushed Data

For details on the format of data pushed by the platform to applications after data subscription is created, see **Data Transfer APIs**.

Constraints

Description	Constraint
Supported MQTT version	3.1.1

Description	Constraint
Differences from the standard MQTT protocol	• QoS 0 is supported. • Custom topics are supported. • Shared subscription is supported. • QoS 1 and QoS 2 are not supported. • Will and retained messages are not supported. • Client publishing is not supported.
Security level supported by MQTTS	TCP + TLS (TLS v1.2) Supported cipher suites: • TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 • TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
MQTT connection requests for an account per second	10
MQTT connections for an account	10 per access credential
Push rate of an MQTT connection	1,000 TPS
Message cache duration and size	The maximum duration is one day, and the maximum size is 1 GB. Caching is limited by either of the item. For example, if the cache duration exceeds one day, data will not be cached even if the size does not reach 1 GB.
Recommended heartbeat interval for MQTT connections	Range: 30s to 1200s; recommended: 120s
Message publishing and subscription	• Shared subscription is supported. Clients that subscribe to the same topic consume pushed data in polling mode. Clients can subscribe to only the topics created in the forwarding rule. • Message publishing is not supported.
Subscriptions per subscription request	Maximum number of topics supported by an account
Topics subscribed to by an account (created during rule action creation)	100

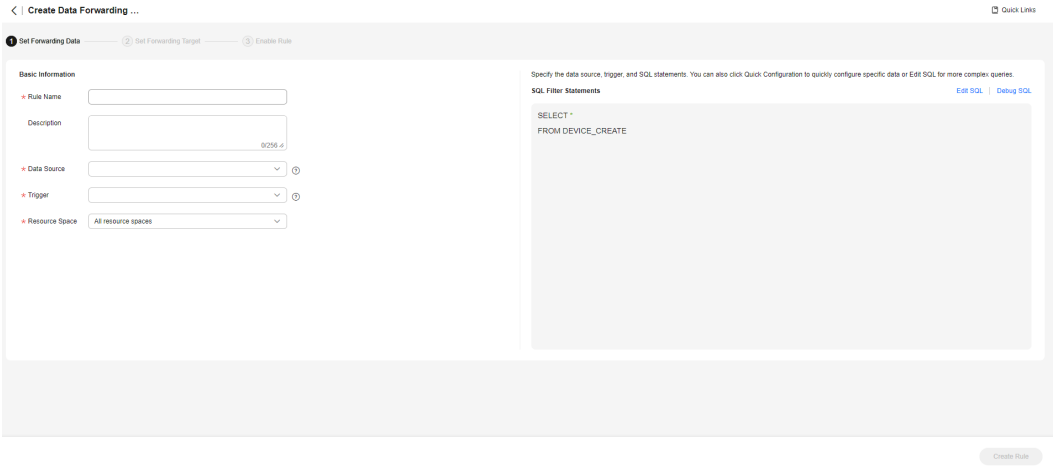
7.6.4.2 MQTT Server Configuration

This topic describes how to set and manage MQTT server subscription on the IoT platform.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Rules** > **Data Forwarding**, and click **Create Rule** on the left.

Figure 7-68 Data forwarding - Creating a rule



Step 3 Set the parameters based on the table below and click **Create Rule**.

Table 7-16 Parameters for creating a rule

Parameter	Description
Rule Name	Name of the rule to be created.
Description	Description of the rule.

Parameter	Description
Data Source	• **Device**: Device information, such as device addition, deletion, and update, will be forwarded. When **Data Source** is set to **Device**, quick configuration is not supported. • **Device property**: A property value reported by a device in a resource space will be forwarded. Click **Quick Configuration** on the right and select the product, property, and service data to forward. • **Device message**: A message reported by a device in a resource space will be forwarded. Click **Quick Configuration** on the right and select data of a specified topic to forward. Select the product to which the topic belongs and enter the topic name. You can use a **custom topic** on the product details page or a **preset topic**. • **Device message status**: The status of device messages exchanged between the device and platform will be forwarded. For details on the device message status, see **Message Delivery Status**. When **Data Source** is set to **Device message status**, quick configuration is not supported. • **Device status**: The status change of a directly or an indirectly connected device in a resource space will be forwarded. Click **Quick Configuration** on the right to forward information about devices whose status is **Online**, **Offline**, or **Abnormal** to other services. For details on the status of devices directly connected to the platform, see **Device Management**. • **Batch task**: The batch task status will be forwarded. When **Data Source** is set to **Batch Task**, quick configuration is not supported. • **Product**: Product information, such as product addition, deletion, and update, will be forwarded. When **Data Source** is set to **Product**, quick configuration is not supported. • **Asynchronous command status of the device**: Status changes of asynchronous commands to devices using LwM2M over CoAP will be forwarded. For details on the asynchronous command status of devices, see **Asynchronous Command Delivery**. When **Data Source** is set to **Asynchronous command status of the device**, quick configuration is not supported. • **Run log**: Service run logs of MQTT devices will be forwarded. When **Data Source** is set to **Run log**, quick configuration is not supported.
Trigger	After you select a data source, the platform automatically matches trigger events.
Resource Space	You can select a single resource space or all resource spaces. If **All resource spaces** is selected, quick configuration is not supported.

Step 4 Under **Set Forwarding Target**, click **Add**. On the displayed page, set the parameters based on the table below and click **OK**.

Parameter	Description
Forwarding Target	Select **MQTT message queue**.
Push Topic	Enter the MQTT topic to which data is forwarded. ● The topic queue name can be customized and must be unique under an account. It can contain up to 128 characters. Use only letters, digits, underscores (_), hyphens (-), and slashes (/). ● The topic that is used for the first time belongs to the resource space selected during rule creation. The topic can be used only in the resource space. If **All resource spaces** is selected during rule creation, the topic can be used in all resource spaces.

Figure 7-69 Creating a forwarding target - to an MQTT push message queue

Step 5 After the rule is defined, click **Enable Rule** to start forwarding data to the MQTT message queue.

----End

7.6.4.3 MQTT Client Access

After configuring and activating rules by calling the platform APIs **Creating a Rule Triggering Condition**, **Creating a Rule Action**, and **Modifying a Rule Triggering Condition**, connect the MQTT client to IoTDA. Then run the MQTT client on your server to receive subscribed-to messages.

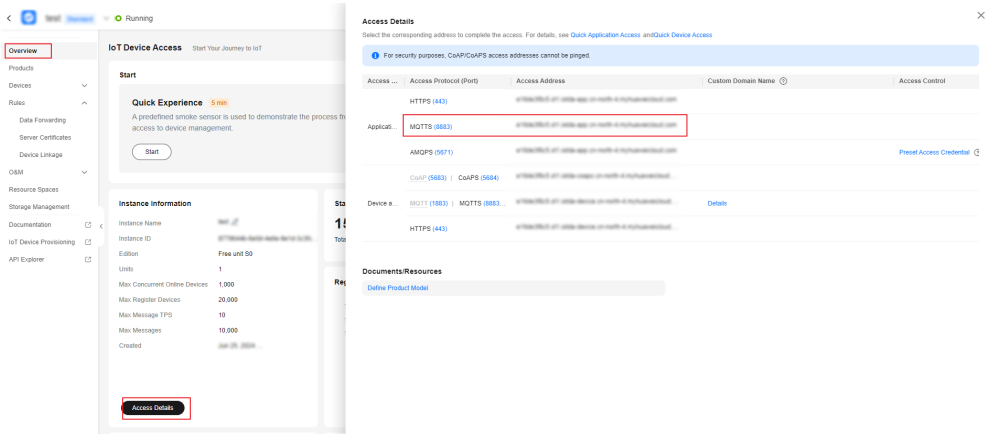
Connection Configuration Parameters

The table below describes the connection address and connection authentication parameters for the MQTT client to connect to the platform.

- MQTT access domain name

  It is automatically generated for each account. Log in to the **IoTDA console** to obtain it on the **Access Details** page.

  Figure 7-70 Access information - MQTT access address on the application side



- Port: 8883
- Client identity authentication parameters

  clientId: The value must be globally unique. You are advised to use **username**.

  username ="accessKey=${accessKey}|timestamp=${timestamp}|instanceId=${instanceId}"

  password ="${accessCode}"

Parameter	Mandatory	Description
${accessKey}	Yes	An accessKey can be used to establish a maximum of 10 concurrent connections. When establishing a connection for the first time, preset the parameter by following the instructions provided in **Obtaining the AMQP Access Credential**.
${timestamp}	Yes	Current time. The value is a 13-digit timestamp, accurate to milliseconds. The server verifies the client timestamp. There is a 5-minute difference between the client timestamp and server timestamp.
instanceId	Optional	Instance ID. This parameter is mandatory when multiple instances of the standard edition are purchased in the same region. For details, see **Viewing Instance Details**.
${accessCode}	Yes	The value can contain a maximum of 256 characters.

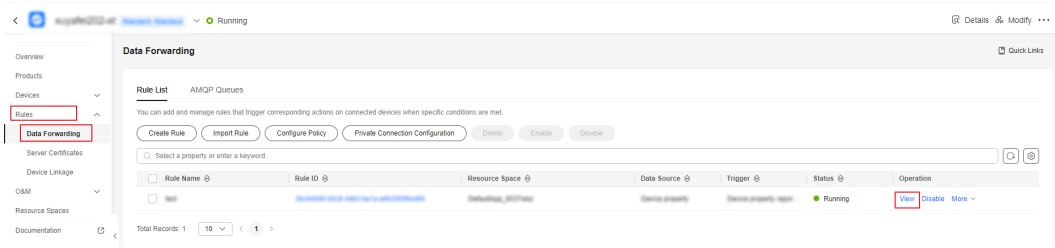
Obtaining the MQTT Access Credential

An access credential is required for an application that uses MQTT to connect to the platform for data forwarding. If you use an access credential for the first time or forget it, preset an access credential. You can call the API for **generating an access credential** or use the console to preset an access credential. The procedure for using the console to generate an access credential is as follows:

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

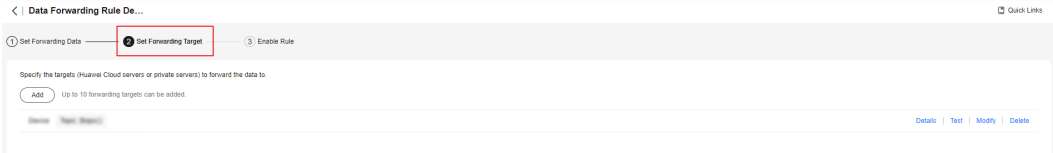
Step 2 Choose **Rules > Data Forwarding**. The **Rule List** page is displayed.

Figure 7-71 Rule details - Viewing rule details



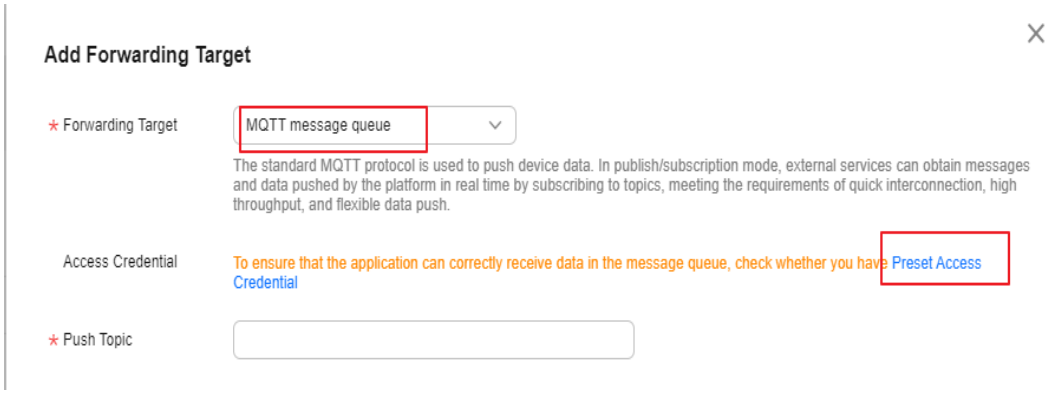
Step 3 Click **View**. (If no rule exists, create one.) On the rule details page that is displayed, click the **Set Forwarding Target** tab.

Figure 7-72 Forwarding target - Setting a target



Step 4 Click **Add**. On the **Add Forwarding Target** page that is displayed, select **MQTT message queue** for **Forwarding Target**, and click **Preset Access Credential** to preset the access code and access key.

Figure 7-73 Creating a forwarding target - to an MQTT push message queue with preset credentials



NOTE

If you already have an access credential, the accessKey cannot be used after you preset the access credential again.

----End

Receiving Push Messages

After a connection is established between the client and the platform, subscribe to the MQTT topic in the data forwarding rule. When a device reports data and the rule is triggered, the platform pushes the data to the MQTT client.

7.6.4.4 Java Demo Usage Guide

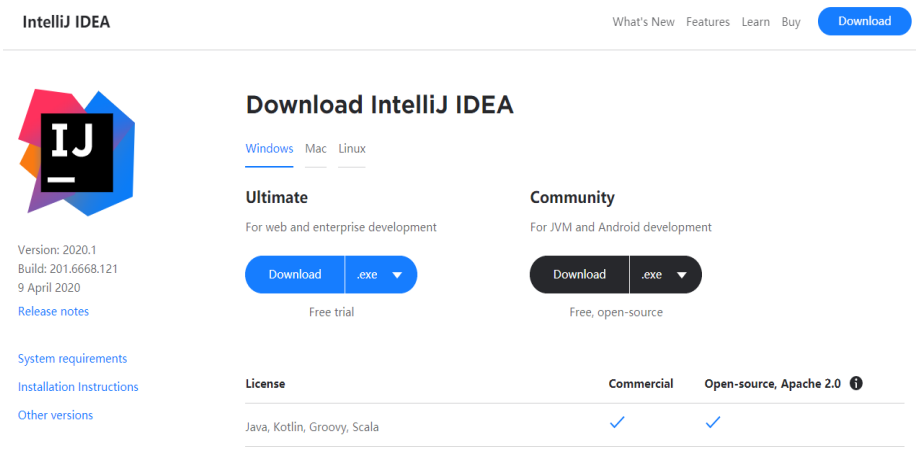
This topic uses Java as an example to describe how to connect an MQTTS client to the platform and receive subscribed messages from the platform.

Prerequisites

You have installed IntelliJ IDEA by following the instructions provided in For details about the installation, see **Installing IntelliJ IDEA**.

Installing IntelliJ IDEA

1. Go to the **IntelliJ IDEA website** to download and install a desired version. The following uses 64-bit IntelliJ IDEA 2019.2.3 Ultimate as an example.

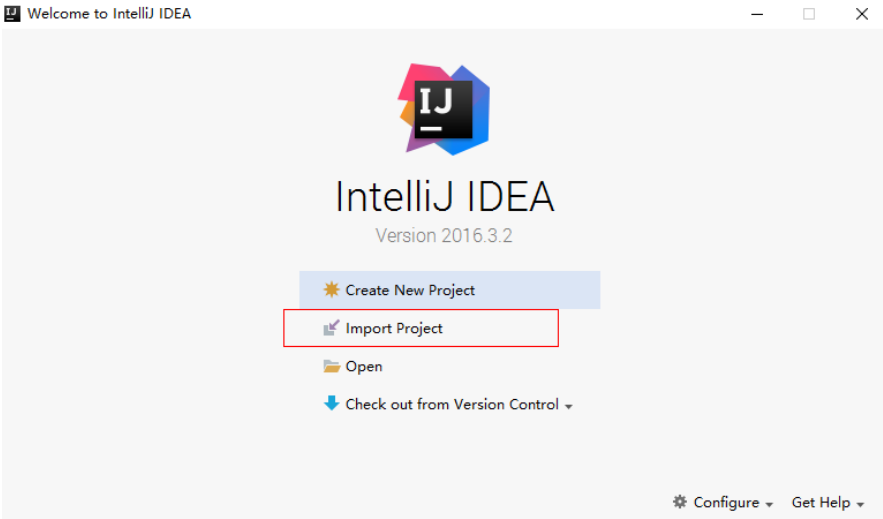


2. After the download is complete, run the installation file and install Node.js as prompted.

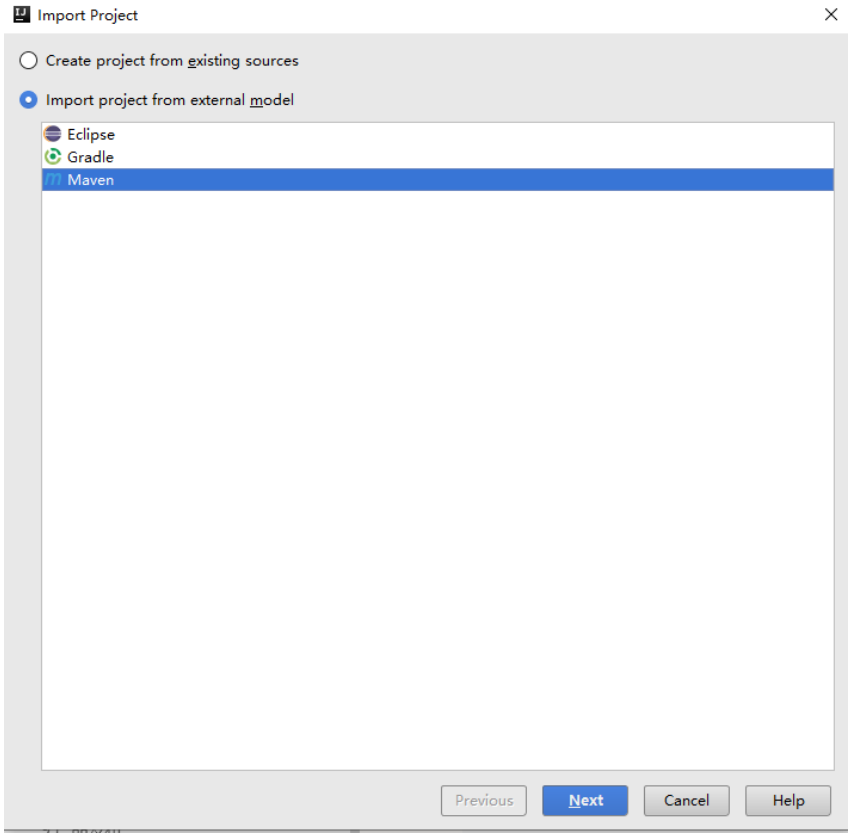
Importing Sample Code

Step 1 Download the **Java demo**.

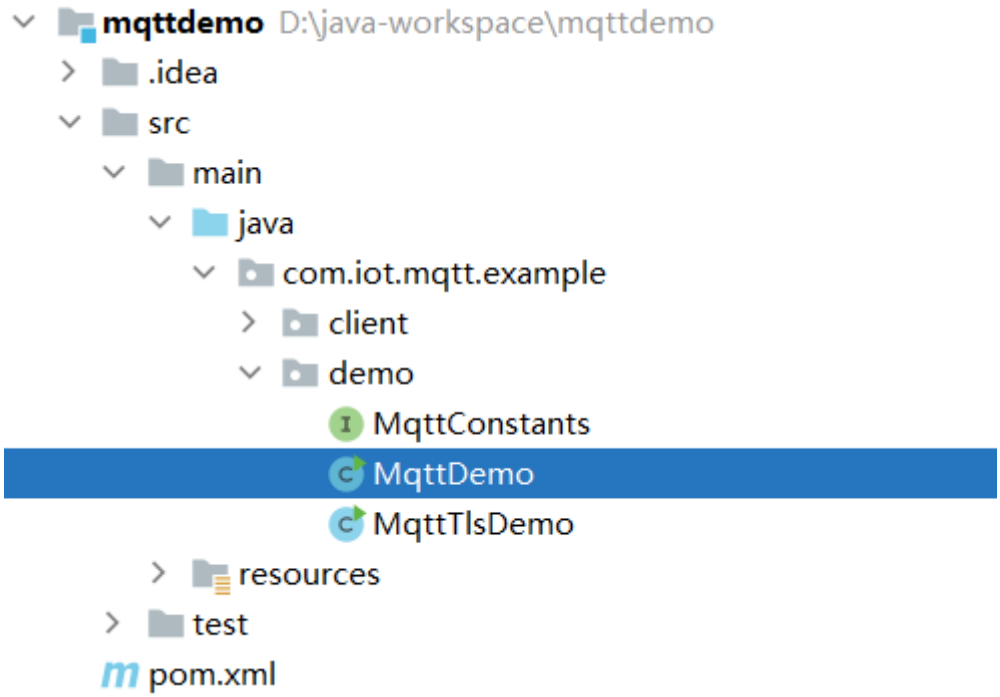
Step 2 Open the IDEA developer tool and click **Import Project**.



Step 3 Select the Java demo downloaded in **1** and click **Next**.



Step 4 Import the sample code.



----End

Establishing a Connection

Step 1 Set the access address and authentication parameters in com.iot.mqtt.example.demo.MqttConstants.

```
// Address for connecting the MQTT client to the platform. Replace it with the MQTT access domain name in "Connection Configuration Parameters".
String HOST = "${HOST}";
// Access credential. Replace it with the access credential obtained in "Obtaining the MQTT Access Credential".
String ACCESS_KEY = "${accessKey}";
String ACCESS_CODE = "${accessCode}";
// Instance ID. This parameter is mandatory when multiple standard instances are purchased in the same region.
String INSTANCE_ID = "${instanceId}";
// Topic for receiving data. Replace it with the topic used for rule action creation.
String SUBSCRIBE_TOPIC = "${subscribeTopic}";
```

NOTE

For details about the parameters in the demo, see **Connection Configuration Parameters**.

Step 2 Run the sample code (com.iot.mqtt.example.demo.MqttDemo) and check whether the subscription is successful based on the log information. This example does not involve the server certificate verification. For details about how to verify the server certificate, see **com.iot.mqtt.example.demo.MqttTlsDemo**.

- Successful subscription

  Figure 7-74 Successful subscription

  ```
  connect success, server url: ssl://[redacted]
  Mqtt client connected. address is ssl://[redacted]
  ibeListenerImpl - Subscribe mqtt topic onSuccess qos: 0
  ```

- Failed subscription

  a. The username or password is incorrect.

  Figure 7-75 Incorrect username or password

  ```
  connect failed, the reason is Bad user name or password (4)
  ```

  b. The topic does not exist.

  Figure 7-76 The topic does not exist

  ```
  connect success, server url: ssl://[redacted]
  Mqtt client connected. address is ssl://[redacted]
  ribeListenerImpl - Subscribe mqtt topic qos: 128
  ribeListenerImpl - Subscribe mqtt topic [redacted] failed.
  ```

----End

Receiving Data

After topic subscription, when a device reports data and a rule is triggered, the MQTT client can receive the forwarded data. The following figure shows the logs generated when the forwarded data is received.

Figure 7-77 Receiving the forwarded data

```
nt - connect success, server url: ssl://
nt - Mqtt client connected. address is ssl://
ubscribeListenerImpl - Subscribe mqtt topic qos: 0
 begin to handler msg. topic =  payload = {"resource":"device.property","event":
```

7.6.4.5 Python Demo

This section uses Python as an example to describe how to connect an MQTTS client to the platform and receive subscribed messages from the platform.

Prerequisites

Knowledge of basic Python syntax and how to configure development environments.

Development Environment

In this example, Python 3.8.8 is used.

Dependency

In this example, **paho-mqtt** (version 2.0.0) is used. You can run the following command to download the dependency:

```
pip install paho-mqtt==2.0.0
```

Sample Code

ClientConf code:

```
from typing import Optional
class ClientConf:
    def __init__(self):
        # MQTT subscription address
        self.__host: Optional[str] = None
        # MQTT subscription port number
        self.__port: Optional[int] = None
        # MQTT access credential access_key
        self.__access_key: Optional[str] = None
        # MQTT access credential access_code
        self.__access_code: Optional[str] = None
        # MQTT subscription topic
        self.__topic: Optional[str] = None
        # Instance ID. This parameter is mandatory when multiple instances of the standard edition are
purchased in the same region.
        self.__instance_id: Optional[str] = None
        # mqtt qos
        self.__qos = 1

    @property
    def host(self):
        return self.__host
    @host.setter
    def host(self, host):
        self.__host = host
    @property
    def port(self):
        return self.__port
    @port.setter
```

```
    def port(self, port):
        self.__port = port
    @property
    def access_key(self):
        return self.__access_key
    @access_key.setter
    def access_key(self, access_key):
        self.__access_key = access_key
    @property
    def access_code(self):
        return self.__access_code
    @access_code.setter
    def access_code(self, access_code):
        self.__access_code = access_code
    @property
    def topic(self):
        return self.__topic
    @topic.setter
    def topic(self, topic):
        self.__topic = topic
    @property
    def instance_id(self):
        return self.__instance_id
    @instance_id.setter
    def instance_id(self, instance_id):
        self.__instance_id = instance_id
    @property
    def qos(self):
        return self.__qos
    @qos.setter
    def qos(self, qos):
        self.__qos = qos
```

MqttClient code:

```
import os
import ssl
import threading
import time
import traceback
import secrets
from client_conf import ClientConf
import paho.mqtt.client as mqtt
class MqttClient:
    def __init__(self, client_conf: ClientConf):
        self.__host = client_conf.host
        self.__port = client_conf.port
        self.__access_key = client_conf.access_key
        self.__access_code = client_conf.access_code
        self.__topic = client_conf.topic
        self.__instance_id = client_conf.instance_id
        self.__qos = client_conf.qos
        self.__paho_client: Optional[mqtt.Client] = None
        self.__connect_result_code = -1
        self.__default_backoff = 1000
        self.__retry_times = 0
        self.__min_backoff = 1 * 1000  # 1s
        self.__max_backoff = 30 * 1000  # 30s

    def connect(self):
        self.__valid_params()
        rc = self.__connect()
        while rc != 0:
            # Backoff reconnection
            low_bound = int(self.__default_backoff * 0.8)
            high_bound = int(self.__default_backoff * 1.0)
            random_backoff = secrets.randbelow(high_bound - low_bound)
            backoff_with_jitter = int(pow(2, self.__retry_times)) * (random_backoff + low_bound)
            wait_time_ms = self.__max_backoff if (self.__min_backoff + backoff_with_jitter) > self.__max_backoff
else (
```

```
                self.__min_backoff + backoff_with_jitter)
            wait_time_s = round(wait_time_ms / 1000, 2)
            print("client will try to reconnect after " + str(wait_time_s) + " s")
            time.sleep(wait_time_s)
            self.__retry_times += 1
            self.close() # Release the previous connection.
            rc = self.__connect()
            # If the value of rc is 0, the connection is set up successfully. If not, the connection fails.
            if rc != 0:
                print("connect with result code: " + str(rc))
                if rc == 134:
                    print("connect failed with bad username or password, "
                          "reconnection will not be performed")
                    pass
        return rc
    def __connect(self):
        try:
            timestamp = self.current_time_millis()
            user_name = "accessKey=" + self.__access_key + "|timestamp=" + timestamp
            if self.__instance_id:
                user_name = user_name + "|instanceId=" + self.__instance_id
            pass_word = self.__access_code
            self.__paho_client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2, "mqttClient")
            # Disable automatic retry and update the timestamp by manual retry.
            self.__paho_client._reconnect_on_failure = False
            # Set the callback function.
            self._set_callback()
            # Topics are stored in userdata. The callback function directly subscribes to topics.
            self.__paho_client.user_data_set(self.__topic)
            self.__paho_client.username_pw_set(user_name, pass_word)
            # Currently, the MQTT broker supports only TLS 1.2.
            context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
            # Not verifying the server certificate.
            context.verify_mode = ssl.CERT_NONE
            context.check_hostname = False
            self.__paho_client.tls_set_context(context)
            rc = self.__paho_client.connect(self.__host, self.__port)
            self.__connect_result_code = rc
            if rc == 0:
                threading.Thread(target=self.__paho_client.loop_forever, args=(1, False),
name="MqttThread").start()
            # Wait for connection establishment.
            time.sleep(1)
        except Exception as e:
            self.__connect_result_code = -1
            print("Mqtt connection error. traceback: " + traceback.format_exc())
        if self.__paho_client.is_connected():
            return 0
        else:
            return self.__connect_result_code
    def __valid_params(self):
        assert self.__access_key is not None
        assert self.__access_code is not None
        assert self.__topic is not None

    @staticmethod
    def current_time_millis():
        return str(int(round(time.time() * 1000)))
    def _set_callback(self):
        # Execute self._on_connect() when the platform responds to the connection request.
        self.__paho_client.on_connect = self._on_connect
        # Execute self._on_disconnect() when disconnecting from the platform.
        self.__paho_client.on_disconnect = self._on_disconnect
        # Execute self._on_subscribe when subscribing to a topic.
        self.__paho_client.on_subscribe = self._on_subscribe
        # Execute self._on_message() when an original message is received.
        self.__paho_client.on_message = self._on_message
    def _on_connect(self, client, userdata, flags, rc: mqtt.ReasonCode, properties):
        if rc == 0:
```

```
            print("Connected to Mqtt Broker! topic " + self.__topic)
            client.subscribe(userdata, 1)
        else:
            # Automatic reconnection is not performed only when the username or password is incorrect.
            # If the disconnect() method is not used here, loop_forever keeps reconnecting.
            if rc == 134:
                self.__paho_client.disconnect()
            print("Failed to connect. return code :" + str(rc.value) + ", reason" + rc.getName())
    def _on_subscribe(self, client, userdata, mid, granted_qos, properties):
        print("Subscribed: " + str(mid) + " " + str(granted_qos) + " topic: " + self.__topic)
    def _on_message(self, client, userdata, message: mqtt.MQTTMessage):
        print("topic " + self.__topic + " Received message: " + message.payload.decode())
    def _on_disconnect(self, client, userdata, flags, rc, properties):
        print("Disconnect to Mqtt Broker. topic: " + self.__topic)
        # Shut down the client after the disconnection and manually reconnect the client to refresh the
timestamp.
        try:
            self.__paho_client.disconnect()
        except Exception as e:
            print("Mqtt connection error. traceback: " + traceback.format_exc())
        self.connect()
    def close(self):
        if self.__paho_client is not None and self.__paho_client.is_connected():
            try:
                self.__paho_client.disconnect()
                print("Mqtt connection close")
            except Exception as e:
                print("paho client disconnect failed. exception: " + str(e))
        else:
            pass
```

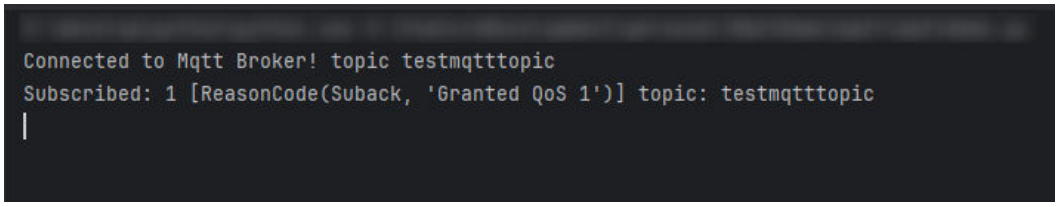
MqttDemo code:

```
from client_conf import ClientConf
from mqtt_client import MqttClient
import os
from typing import Optional
def main():
    client_conf = ClientConf()
    client_conf.host = "your ip host"
    client_conf.port = 8883
    client_conf.topic = "your mqtt topic"
    # MQTT access credential access_key can be injected using environment variables.
    client_conf.access_key = os.environ.get("MQTT_ACCESS_KEY")
    # MQTT access credential access_code can be injected using environment variables.
    client_conf.access_code = os.environ.get("MQTT_ACCESS_CODE")
    client_conf.instance_id = "your instance id"
    mqtt_client = MqttClient(client_conf)
    if mqtt_client.connect() != 0:
        print("init failed")
        return
if __name__ == "__main__":
    main()
```

Success Example

After the access is successful, the following information is displayed on the client.

Figure 7-78 Example of successful MQTT subscription using Python



7.6.4.6 GO Demo

This section uses Go as an example to describe how to connect an MQTTS client to the platform and receive subscribed messages from the platform.

Prerequisites

Knowledge of basic Go syntax and how to configure development environments.

Development Environment

In this example, Go 1.18 is used.

Dependency

In this example, **paho.mqtt.golang** (version 1.4.3) is used. You can run the following command to add the dependency to go.mod.

```
require (
    github.com/eclipse/paho.mqtt.golang v1.4.3
)
```

Sample Code

```
package main

import (
   "crypto/tls"
   "fmt"
   mqtt "github.com/eclipse/paho.mqtt.golang"
   "os"
   "os/signal"
   "time"
)
type MessageHandler func(message string)
type MqttClient struct {
   Host            string
   Port            int
   ClientId        string
   AccessKey       string
   AccessCode      string
   Topic           string
   InstanceId      string
   Qos             int
   Client          mqtt.Client
   messageHandlers []MessageHandler
}
func (mqttClient *MqttClient) Connect() bool {
   return mqttClient.connectWithRetry()
}
func (mqttClient *MqttClient) connectWithRetry() bool {
   // Retries with exponential backoff, from 10 ms to 20s.
   duration := 10 * time.Millisecond
   maxDuration := 20000 * time.Millisecond
   // Retry upon connection establishment failure.
   internal := mqttClient.connectInternal()
   times := 0
   for !internal {
      time.Sleep(duration)
      if duration < maxDuration {
         duration *= 2
      }
      times++
```

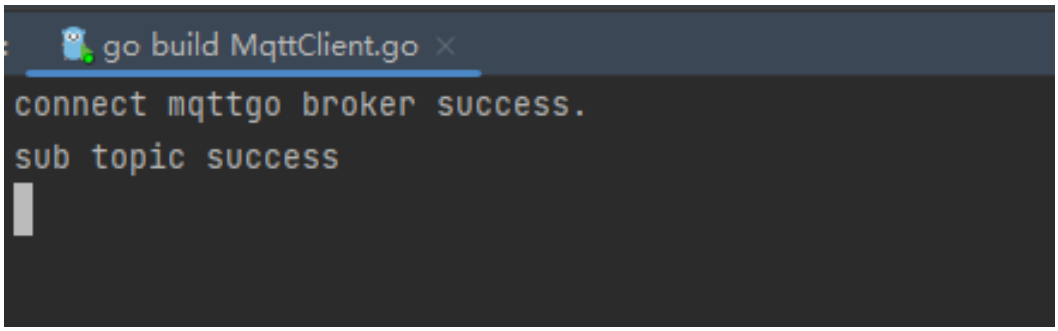
```
        fmt.Println("connect mqttgo broker retry. times: ", times)
        internal = mqttClient.connectInternal()
    }
    return internal
}
func (mqttClient *MqttClient) connectInternal() bool {
    // Close the existing connection before establishing a connection.
    mqttClient.Close()
    options := mqtt.NewClientOptions()
    options.AddBroker(fmt.Sprintf("mqtts://%s:%d", mqttClient.Host, mqttClient.Port))
    options.SetClientID(mqttClient.ClientId)
    userName := fmt.Sprintf("accessKey=%s|timestamp=%d", mqttClient.AccessKey, time.Now().UnixNano()/
1000000)
    if len(mqttClient.InstanceId) != 0 {
        userName = userName + fmt.Sprintf("|instanceId=%s", mqttClient.InstanceId)
    }
    options.SetUsername(userName)
    options.SetPassword(mqttClient.AccessCode)
    options.SetConnectTimeout(10 * time.Second)
    options.SetKeepAlive(120 * time.Second)
    // Disable the SDK internal reconnection and use the custom reconnection to refresh the timestamp.
    options.SetAutoReconnect(false)
    options.SetConnectRetry(false)
    tlsConfig := &tls.Config{
        InsecureSkipVerify: true,
        MaxVersion:         tls.VersionTLS12,
        MinVersion:         tls.VersionTLS12,
    }
    options.SetTLSConfig(tlsConfig)
    options.OnConnectionLost = mqttClient.createConnectionLostHandler()
    client := mqtt.NewClient(options)
    if token := client.Connect(); token.Wait() && token.Error() != nil {
        fmt.Println("device create bootstrap client failed,error = ", token.Error().Error())
        return false
    }
    mqttClient.Client = client
    fmt.Println("connect mqttgo broker success.")
    mqttClient.subscribeTopic()
    return true
}
func (mqttClient *MqttClient) subscribeTopic() {
    subRes := mqttClient.Client.Subscribe(mqttClient.Topic, 0, mqttClient.createMessageHandler())
    if subRes.Wait() && subRes.Error() != nil {
        fmt.Printf("sub topic failed,error is %s\n", subRes.Error())
        panic("subscribe topic failed.")
    } else {
        fmt.Printf("sub topic success\n")
    }
}
func (mqttClient *MqttClient) createMessageHandler() func(client mqtt.Client, message mqtt.Message) {
    messageHandler := func(client mqtt.Client, message mqtt.Message) {
        fmt.Println("receive message from server.")
        go func() {
            for _, handler := range mqttClient.messageHandlers {
                handler(string(message.Payload()))
            }
        }()
    }
    return messageHandler
}
func (mqttClient *MqttClient) createConnectionLostHandler() func(client mqtt.Client, reason error) {
    // Perform custom reconnection after disconnection.
    connectionLostHandler := func(client mqtt.Client, reason error) {
        fmt.Printf("connection lost from server. begin to reconnect broker. reason: %s\n", reason.Error())
        connected := mqttClient.connectWithRetry()
        if connected {
            fmt.Println("reconnect mqttgo broker success.")
        }
    }
```

```
    return connectionLostHandler
}
func (mqttClient *MqttClient) Close() {
    if mqttClient.Client != nil {
        mqttClient.Client.Disconnect(1000)
    }
}
func main() {
    // For details about how to set the following parameters, see the connection configuration description.
    // MQTT access domain name
    mqttHost := "your mqtt host"
    // MQTT access port
    mqttPort := 8883
    // Access credential key value
    mqttAccessKey := os.Getenv("MQTT_ACCESS_KEY")
    // Access credential secret
    mqttAccessCode := os.Getenv("MQTT_ACCESS_CODE")
    // Name of the subscribed topic
    mqttTopic := "your mqtt topic"
    // Instance ID
    instanceId := "your instance Id"
    //mqttgo client id
    clientId := "your mqtt client id"

    mqttClient := MqttClient{
        Host:       mqttHost,
        Port:       mqttPort,
        Topic:      mqttTopic,
        ClientId:   clientId,
        AccessKey:  mqttAccessKey,
        AccessCode: mqttAccessCode,
        InstanceId: instanceId,
    }
    // Customize the handler for processing messages.
    mqttClient.messageHandlers = []MessageHandler{func(message string) {
        fmt.Println(message)
    }}
    connect := mqttClient.Connect()
    if !connect {
        fmt.Println("init mqttgo client failed.")
        return
    }
    // Block method to keep the MQTT client always pulling messages.
    interrupt := make(chan os.Signal, 1)
    signal.Notify(interrupt, os.Interrupt)
    for {
        <-interrupt
        break
    }
}
```

Success Example

After the access is successful, the following information is displayed on the client.

Figure 7-79 Example of successful MQTT client access using Go



7.6.4.7 Node.js Demo

This section uses Node.js as an example to describe how to connect an MQTTS client to the platform and receive subscribed messages from the platform

Prerequisites

Knowledge of basic Node.js syntax and how to configure development environments.

Development Environment

In this example, Node.js 13.14.0 is used. **Download** it from the Node.js official website. After installation, run the following command to check the version:

```
node --version
```

Dependency

In this example, **mqtt** (version 4.0.0) is used. You can run the following command to download the dependency:

```
npm install mqtt@4.0.0
```

Sample Code

```
const mqtt = require('mqtt');
// Name of the subscribed topic
var topic = "your mqtt topic";
// Key value of the access credential, which can be preset using environment variables.
var accessKey = process.env.MQTT_ACCESS_KEY;
// Access credential secret, which can be preset using environment variables.
var accessCode = process.env.MQTT_ACCESS_CODE;
// MQTT access address
var mqttHost = "your mqtt host";
// MQTT access port
var mqttPort = 8883;
// Instance ID
var instanceId = "your instanceId";
// mqtt client id
var clientId = "your clientId";
// MQTT client
var client = null;
connectWithRetry();
async function connectWithRetry() {
    // Retries with exponential backoff, from 1s to 20s.
   var duration = 1000;
```

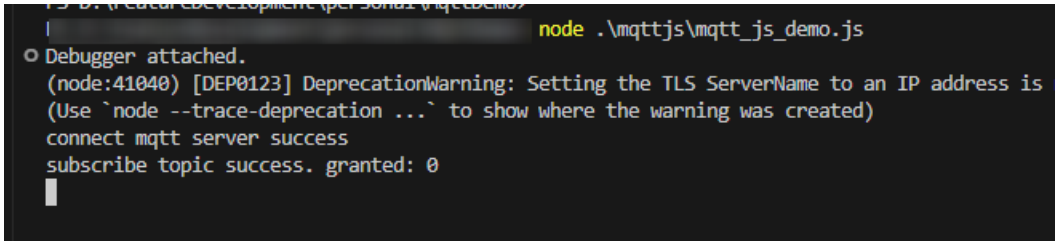
```
    var maxDuration = 20000;
    var success = connect(topic);
    var times = 0;
    while (!success) {
      await sleep(duration)
      if (duration < maxDuration) {
        duration *= 2
      }
      times++
      console.log('connect mqtt broker retry. times: ' + times)
       if (client == null) {
           connect(topic)
           continue
       }
      client.end(true, function() {
           connect(topic)
       });
    }
}
function sleep(ms) {
    return new Promise(resolve => setTimeout(() => resolve(), ms))
}
function connect(topic) {
    try {
        client = mqtt.connect(getClientOptions())
        if (client == null) {
            return false
        }
        client.on('connect', connectCallBack)
        client.subscribe(topic, subscribeCallBack)
        client.on('message', messageCallBack)
        client.on('error', clientErrorCallBack)
        client.on('close', closeCallBack)
        return true
    } catch (error) {
        console.log('connect to mqtt broker failed. err ' + error)
    }
    return false
}
function getClientOptions() {
    var timestamp = Math.round(new Date);
    const username = 'accessKey=' + accessKey + '|timestamp=' + timestamp + '|instanceId=' + instanceId;
    var options = {
        host: mqttHost,
        port: mqttPort,
        connectTimeout: 4000,
        clientId: clientId,
        protocol: 'mqtts',
        keepalive: 120,
        username: username,
        password: accessCode,
        rejectUnauthorized: false,
        secureProtocol: 'TLSv1_2_method'
    };
    return options;
};
function connectCallBack() {
    console.log('connect mqtt server success');
};
function subscribeCallBack(err, granted) {
    if (err != null || granted[0].qos === 128) {
        console.log('subscribe topic failed. granted: ' + granted[0].qos)
        return
    }
    console.log('subscribe topic success. granted: ' + granted[0].qos);
};
function clientErrorCallBack(err) {
    console.log('mqtt client error ' + err);
};
```

```
function messageCallBack(topic, message) {
    console.log('receive message ' + message);
};
function closeCallBack() {
    console.log('Disconnected from mqtt broker')
    client.end(true, function() {
        console.log('close connection');
        connectWithRetry();
     });
}
```

Success Example

After the access is successful, the following information is displayed on the client.

Figure 7-80 Example of successful MQTT client access using Node.js



7.6.4.8 C# Demo

This section uses C# as an example to describe how to connect an MQTTS client to the platform and receive subscribed messages from the platform

Prerequisites

Knowledge of basic C# syntax and how to configure .NET Framework development environments.

Development Environment

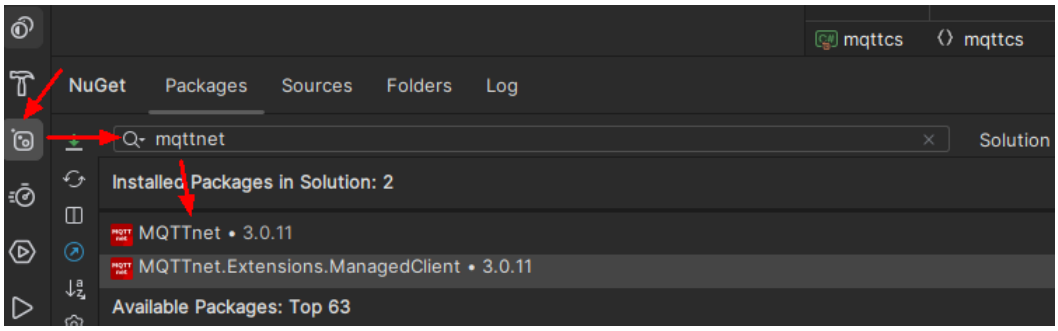
In this example, .NET Framework 4.6.2 and .NET SDK 6.0.421 are used. **Download** them from the .NET official website. After installation, run the following command to check the version:

```
dotnet -v
```

Dependency

In this example, **MQTTnet** and **MQTTnet.Extension.ManagedClient** (version 3.0.11) are used. You can search for **MQTTnet** in the NuGet manager and install the required version.

Figure 7-81 nuget installation dependency



Sample Code

ClientConf.cs code:

```
using MQTTnet.Protocol;

namespace mqttcs
{
    public class ClientConf
    {
        // MQTT subscription address
        public string ServerUri { get; set; }

        // MQTT subscription port number
        public int Port { get; set; }

        // MQTT access credential access_key
        public string AccessKey { get; set; }

        // MQTT access credential access_code
        public string AccessCode { get; set; }

        // MQTT client ID
        public string ClientId { get; set; }

        // Instance ID. This parameter is mandatory when multiple instances of the standard edition are
purchased in the same region.
        public string InstanceId { get; set; }

        // MQTT subscription topic
        public string Topic { get; set; }

        // mqtt qos
        public MqttQualityOfServiceLevel Qos { get; set; }

    }
}
```

MqttListener code:

```
using System;
using MQTTnet.Client.Connecting;
using MQTTnet.Client.Disconnecting;
using MQTTnet.Extensions.ManagedClient;

namespace mqttcs
{
    public interface MqttListener
    {
        // Callback function when the MQTT client is disconnected from the server
        void ConnectionLost(MqttClientDisconnectedEventArgs e);

        // Callback function for successful connection establishment between the MQTT client and server
```

```
        void ConnectComplete(MqttClientConnectResultCode resultCode, String reason);

        // Callback function for consuming messages on the MQTT client
        void OnMessageReceived(String message);

        // Callback function when the MQTT client fails to establish a connection with the server
        void ConnectFail(ManagedProcessFailedEventArgs e);
    }
}
```

MqttConnection.cs code:

```
using System;
using System.Text;
using System.Threading;
using MQTTnet;
using MQTTnet.Client.Connecting;
using MQTTnet.Client.Disconnecting;
using MQTTnet.Client.Options;
using MQTTnet.Client.Receiving;
using MQTTnet.Extensions.ManagedClient;
using MQTTnet.Formatter;

namespace mqttcs
{
    public class MqttConnection
    {
        private static IManagedMqttClient client = null;

        private static ManualResetEvent mre = new ManualResetEvent(false);

        private static readonly ushort DefaultKeepLive = 120;

        private static int _retryTimes = 0;

        private readonly int _retryTimeWait = 1000;

        private readonly ClientConf _clientConf;

        private MqttListener _listener;

        public MqttConnection(ClientConf clientConf, MqttListener listener)
        {
            _clientConf = clientConf;
            _listener = listener;
        }

        public int Connect()
        {
            client?.StopAsync();
        // Backoff retry from 1s to 20s
            var duration = 1000;
            var maxDuration = 20 * 1000;
            var rc = InternalConnect();
            while (rc != 0)
            {
                Thread.Sleep((int)duration);
                if (duration < maxDuration)
                {
                    duration *= 2;
                }
                client?.StopAsync();
                _retryTimes++;
                Console.WriteLine("connect mqtt broker retry. times: " + _retryTimes);
                rc = InternalConnect();
            }

            return rc;
        }
```

```
    private int InternalConnect()
    {
        try
        {
            client = new MqttFactory().CreateManagedMqttClient();
            client.ApplicationMessageReceivedHandler =
                new
MqttApplicationMessageReceivedHandlerDelegate(ApplicationMessageReceiveHandlerMethod);
            client.ConnectedHandler = new MqttClientConnectedHandlerDelegate(OnMqttClientConnected);
            client.DisconnectedHandler = new
MqttClientDisconnectedHandlerDelegate(OnMqttClientDisconnected);
            client.ConnectingFailedHandler = new
ConnectingFailedHandlerDelegate(OnMqttClientConnectingFailed);
            IManagedMqttClientOptions options = GetOptions();
            // Connects to the platform.
            client.StartAsync(options);
            mre.Reset();

            mre.WaitOne();
            if (!client.IsConnected)
            {
                return -1;
            }

            var mqttTopicFilter = new
MqttTopicFilterBuilder().WithTopic(_clientConf.Topic).WithQualityOfServiceLevel(_clientConf.Qos).Build();

            client.SubscribeAsync(mqttTopicFilter).Wait();
            Console.WriteLine("subscribe topic success.");
            return 0;
        }
        catch (Exception e)
        {
            Console.WriteLine("Connect to mqtt server failed. err: " + e);
            return -1;
        }
    }

    private void ApplicationMessageReceiveHandlerMethod(MqttApplicationMessageReceivedEventArgs e)
    {
        string payload = null;
        if (e.ApplicationMessage.Payload != null)
        {
            payload = Encoding.UTF8.GetString(e.ApplicationMessage.Payload);
        }
        try
        {
            _listener?.OnMessageReceived(payload);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Message received error, the message is " + payload);
        }

    }

    private void OnMqttClientConnected(MqttClientConnectedEventArgs e)
    {
        try
        {
            _retryTimes = 0;
            _listener?.ConnectComplete(e.AuthenticateResult.ResultCode, e.AuthenticateResult.ReasonString);
            mre.Set();
        }
        catch (Exception exception)
        {
            Console.WriteLine("handle connect callback failed. e: " + exception.Message);
        }
    }
```

```
        private void OnMqttClientDisconnected(MqttClientDisconnectedEventArgs e)
        {
            try
            {
                _listener?.ConnectionLost(e);
            }
            catch (Exception exception)
            {
                Console.WriteLine("handle disConnect callback failed. e: " + exception.Message);
            }

        }

        private void OnMqttClientConnectingFailed(ManagedProcessFailedEventArgs e)
        {
            try
            {
                if (_listener != null)
                {
                    _listener.ConnectFail(e);
                }
                Thread.Sleep(_retryTimeWait);
                Connect();
            }
            catch (Exception exception)
            {
                Console.WriteLine("handle connect failed callback failed. e: " + exception.Message);
            }
        }

        private IManagedMqttClientOptions GetOptions()
        {
            IManagedMqttClientOptions options = null;
            long timestamp = new DateTimeOffset(DateTime.UtcNow).ToUnixTimeMilliseconds();
            string userName = "accessKey=" + _clientConf.AccessKey + "|timestamp=" + timestamp + "|
instanceId=" + _clientConf.InstanceId;

            options = new ManagedMqttClientOptionsBuilder()
                .WithClientOptions(new MqttClientOptionsBuilder()
                    .WithTcpServer(_clientConf.ServerUri, _clientConf.Port)
                    .WithCredentials(userName, _clientConf.AccessCode)
                    .WithClientId(_clientConf.ClientId)
                    .WithKeepAlivePeriod(TimeSpan.FromSeconds(DefaultKeepLive))
                    .WithTls(new MqttClientOptionsBuilderTlsParameters()
                    {
                        AllowUntrustedCertificates = true,
                        UseTls = true,
                        CertificateValidationHandler = delegate { return true; },
                        IgnoreCertificateChainErrors = false,
                        IgnoreCertificateRevocationErrors = false,
                        SslProtocol = System.Security.Authentication.SslProtocols.Tls12,
                    })
                    .WithProtocolVersion(MqttProtocolVersion.V500)
                    .Build())
                .Build();
            return options;
        }
    }
}
```

MqttClient.cs code:

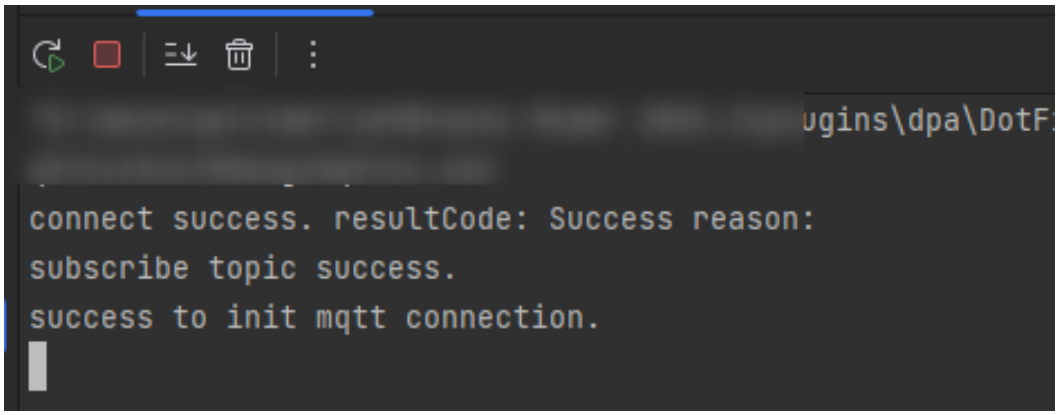
```
using System;
using System.Threading;
using System.Threading.Tasks;
using MQTTnet.Client.Connecting;
using MQTTnet.Client.Disconnecting;
using MQTTnet.Extensions.ManagedClient;
using MQTTnet.Protocol;
```

```
namespace mqttcs
{
    class MqttClient: MqttListener
    {
        private static ManualResetEvent mre = new ManualResetEvent(false);

        public static async Task Main(string[] args)
        {
            ClientConf clientConf = new ClientConf();
            clientConf.ClientId = "your mqtt clientId";
            clientConf.ServerUri = "your mqtt host";
            clientConf.Port = 8883;
            clientConf.AccessKey = Environment.GetEnvironmentVariable("MQTT_ACCESS_KEY");
            clientConf.AccessCode = Environment.GetEnvironmentVariable("MQTT_ACCESS_CODE");
            clientConf.InstanceId = "your instanceId";
            clientConf.Topic = "your mqtt topic";
            clientConf.Qos = MqttQualityOfServiceLevel.AtMostOnce;

            MqttConnection connection = new MqttConnection(clientConf, new MqttClient());
            var connect = connection.Connect();
            if (connect == 0)
            {
                Console.WriteLine("success to init mqtt connection.");
                mre.WaitOne();
            }
        }

        public void ConnectionLost(MqttClientDisconnectedEventArgs e)
        {
            if (e?.Exception != null)
            {
                Console.WriteLine("connect was lost. exception: " + e.Exception.Message);
                return;
            }
            Console.WriteLine("connect was lost");

        }

        public void ConnectComplete(MqttClientConnectResultCode resultCode, String reason)
        {
            Console.WriteLine("connect success. resultCode: " + resultCode + " reason: " + reason);
        }

        public void OnMessageReceived(string message)
        {
            Console.WriteLine("receive msg: " + message);
        }

        public void ConnectFail(ManagedProcessFailedEventArgs e)
        {
            Console.WriteLine("connect mqtt broker failed. e: " + e.Exception.Message);
        }
    }
}
```

Success Example

After the access is successful, the following information is displayed on the client.

Figure 7-82 Example of successful client access using C#



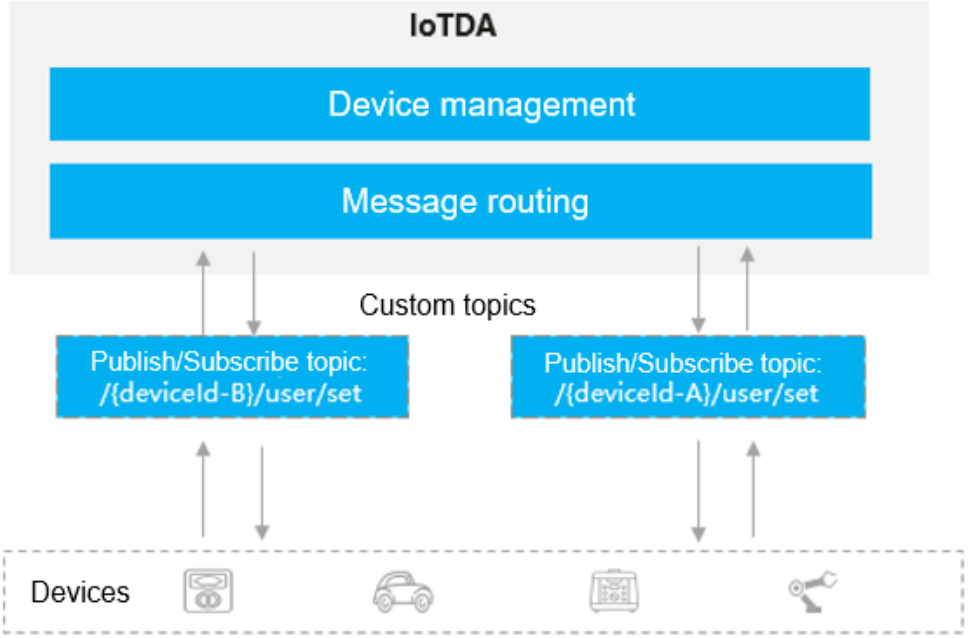
7.6.5 M2M Communications

Overview

Subscription: You can create rules on the console or call the platform APIs to configure and activate rules for obtaining messages reported by devices from the platform. Related APIs: **Create a Rule Trigger Condition**, **Create a Rule Action**, and **Modify the Rule Triggering Condition**. Device subscription supports only message reporting.

Push: After the subscription is successful, the platform pushes messages reported by devices to the specified MQTT topic. After devices are connected to the platform, you can subscribe to the topic to receive data for inter-device message communications. The following figure shows the message communications process between devices.

Figure 7-83 M2M communications



Device Subscription

For details, see **Usage**.

7.7 Data Forwarding Channel Details

Overview

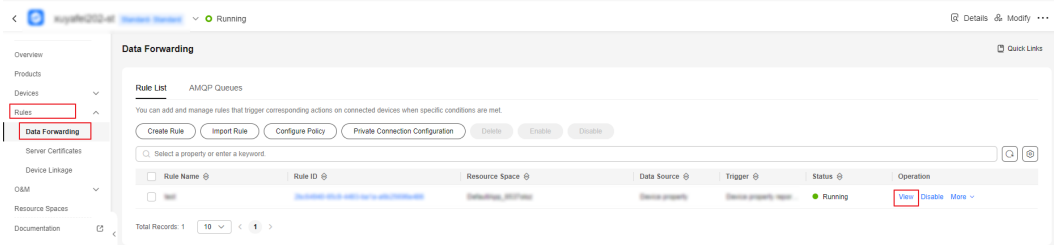
When using the data forwarding function, you can check whether the performance of the forwarding target (such as a third-party application) meets service requirements based on the usage of the data forwarding channel. For example, if the data forwarded to the target (third-party application) cannot be quickly processed, the data will be stacked (cached) on the platform. In this case, you can view the channel details. If you find that the message production rate is always higher than the push rate, and the number of stacked messages keeps increasing, it may indicate that the performance of the target (third-party application) cannot meet the service requirements and scale-out is required. In addition, you can clear stacked data in the data forwarding channel.

Viewing Forwarding Channel Details

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Rules** > **Data Forwarding**, locate the target rule, and click **View** in the **Operation** column.

Figure 7-84 Rule details - Viewing rule details



Step 3 Click **Set Forwarding Target**, find the target data forwarding channel, and click **Details**. View the push details in the displayed dialog box.

Figure 7-85 Forwarding rule details

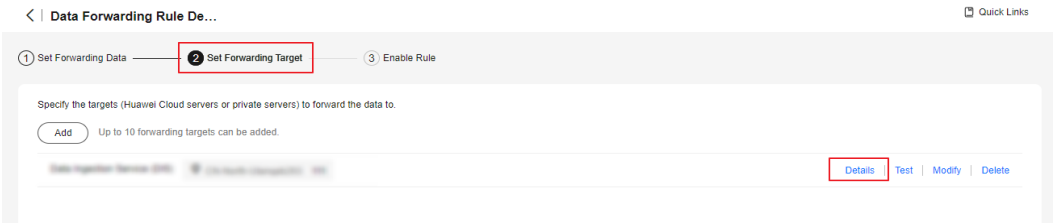


Figure 7-86 Push details - Data forwarding rule

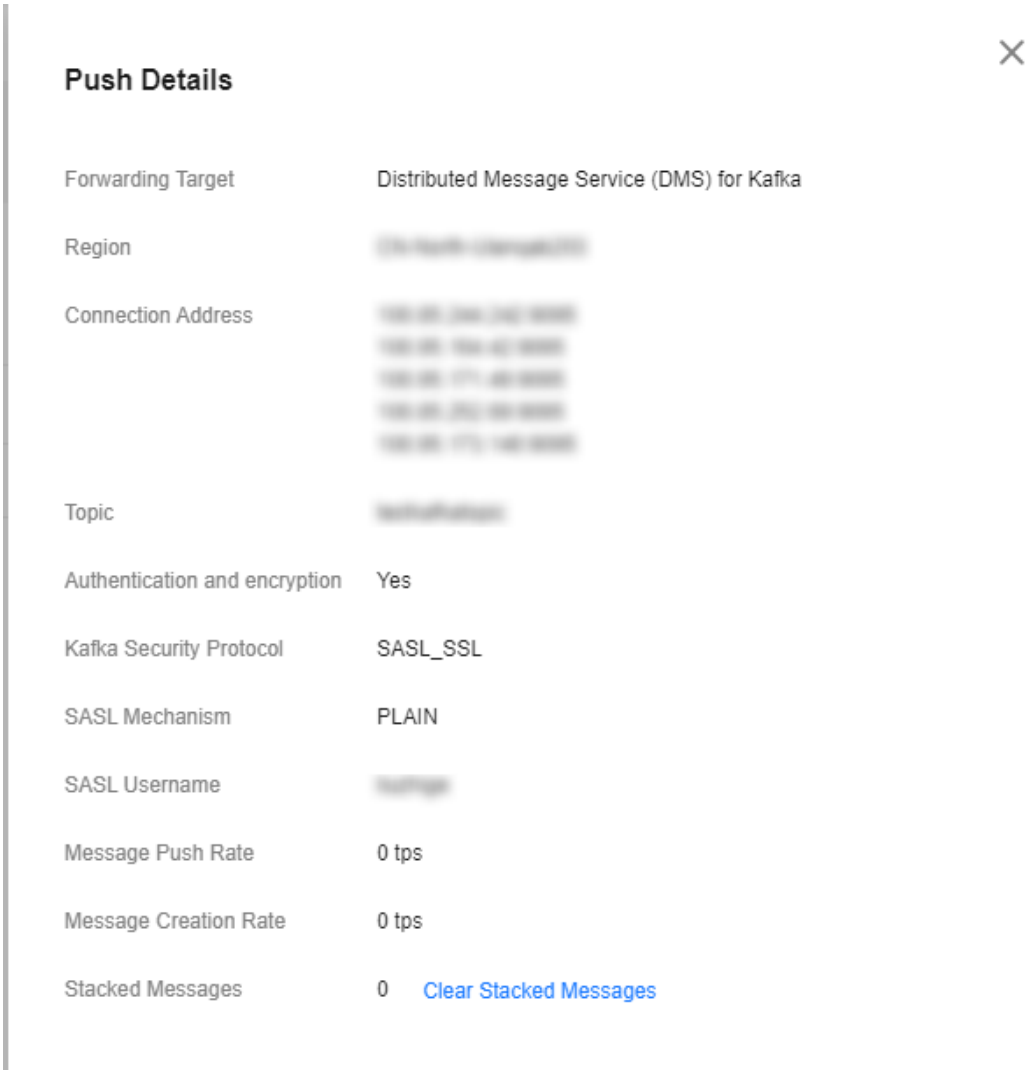


Table 7-17 Parameter description

Parameter	Description
Message Push Rate	Number of messages forwarded by the platform to the target per second.
Message Creation Rate	Number of messages sent by the device to the platform per second.
Stacked Messages	Number of messages stacked on the platform when the production rate is higher than the push rate. For a data forwarding rule, the max. stacking (cache) data size is 1 GB, and the max. stacking (cache) duration is 24 hours by default. To change the values, see **Data Forwarding Stack Policies**.

----End

Clearing Stacked Messages

When the rule engine forwards messages to a third-party application, if the application cannot process the data in real time, the data will be stacked on the platform. You can clear the data stacked in the forwarding channel for timely processing.

For example, a water meter periodically reports user usage to a server. When the server is faulty, the forwarded data piles up. In this case, you can clear the stacked data and let the system process the newly reported data.

NOTICE

In the details page of a forwarding target, if you click **Clear Stacked Messages**, all data that has not been transferred to the forwarding target will be cleared. Exercise caution.

7.8 Data Forwarding Stack Policies

Overview

If the forwarding target (such as a third-party application server) cannot process data forwarded by IoTDA in a timely manner due to insufficient performance, unprocessed data will be stacked (cached) on IoTDA. By default, the maximum stack (cache) size of data to forward for a single forwarding rule is 1 GB, and the maximum stack duration is 24 hours. If the maximum stack size or stack duration is exceeded, the earliest unprocessed data will be discarded to meet the stack size and duration requirements.

To control data stacking on IoTDA, you can create stack policies based on specific service scenarios and performance of the forwarding target (for example, a third-party application server).

If your service has higher requirements on real-time data than integrity and the performance of the forwarding target is insufficient or the service fails to process forwarded data in a timely manner due to an interruption, a large amount of data will be stacked on IoTDA. As a result, the forwarding target always receives delayed data. You can use a stack policy to configure a small stack size and stack duration. In this way, outdated data will be discarded, and real-time data will be received and processed.

Constraints

You can create one stack policy for an IoTDA instance.

NOTICE

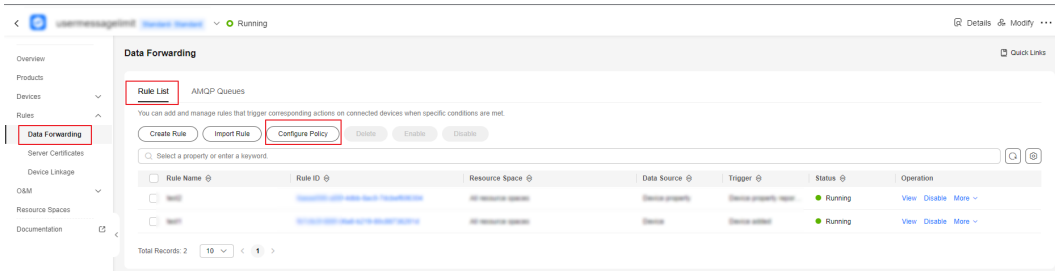
1. After a stack policy is created, it applies to all forwarding rules.and overwrites the default stack size (1 GB) and stack duration (24 hours).
2. If the maximum stack size or stack duration is exceeded, the earliest unprocessed data will be discarded. Use a stack policy carefully and configure a proper stack size and duration.

Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Rules** > **Data Forwarding**, click **Configure Policy**, and click **Stack Policies**.

Figure 7-87 Data forwarding - Policy configuration



Step 3 In the displayed dialog box, set **Policy Name**, **Description**, **Stack Size**, and **Stack Time**, and click **OK**.

Figure 7-88 Data forwarding - Creating a stack policy

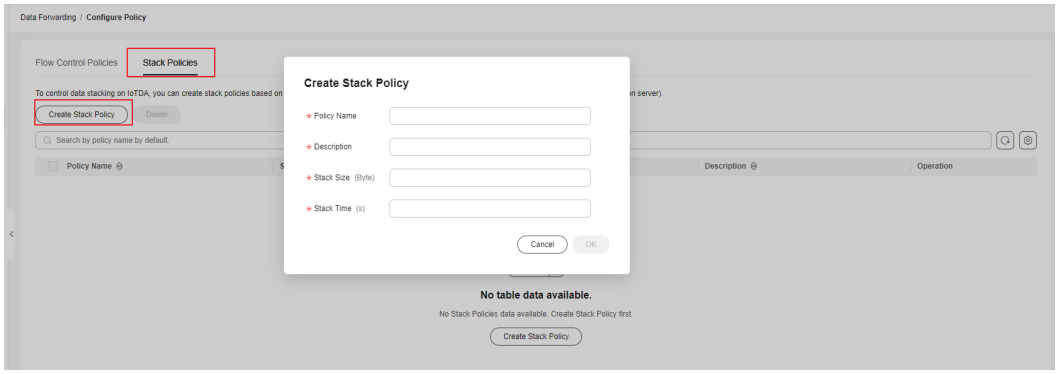


Table 7-18 Parameters

Parameter	Description
Policy Name	The value can contain 4 to 256 characters. Only letters, digits, and special characters (_?'#().,&%@!-) are allowed.

Parameter	Description
Description	Description of the policy. The value can contain 4 to 256 characters. Only letters, digits, and special characters (_?'#().,&%@!-) are allowed.
Stack Size	Maximum stack (cache) size of data to forward in a rule on IoTDA. The unit is byte. The maximum value is **1073741823** bytes (1 GB).
Stack Time	Maximum stack (cache) duration of data to forward in a rule on IoTDA. The unit is second. The maximum value is **86399** seconds (24 hours).

----End

7.9 Data Forwarding Flow Control Policies

Overview

You can create flow control policies in different dimensions on IoTDA based on your service scenarios and performance of the forwarding target (such as a third-party application server) to control data forwarding flows.

Dimensions of Flow Control Policies

Table 7-19 Types of flow control policies

Policy Type	Description
By instance	The policy applies to all data forwarding flows on the instance. Data that exceeds the threshold will be discarded.
By forwarding target	The policy applies to all data forwarding flows of the forwarding target you specify.
By forwarding rule	The policy applies to all data forwarding flows of the forwarding rule you specify. Data that exceeds the threshold will be discarded.
By forwarding action	The policy applies to all data forwarding flows of the forwarding action you specify.

NOTICE

1. After a flow control policy is created, the policy type cannot be modified.
2. Data that exceeds the threshold set in instance- and rule-level flow control policies will be discarded. Use flow control policies carefully.
3. If you create different types of flow control policies, the policy with the threshold reached first takes effect. For example, if the threshold in the flow control policy for forwarding rule A is 50 TPS, the threshold in the policy for action B of forwarding rule A is 100 TPS, and the actual data flow of forwarding rule A is 80 TPS, the flow control policy for forwarding rule A is triggered.

Constraints

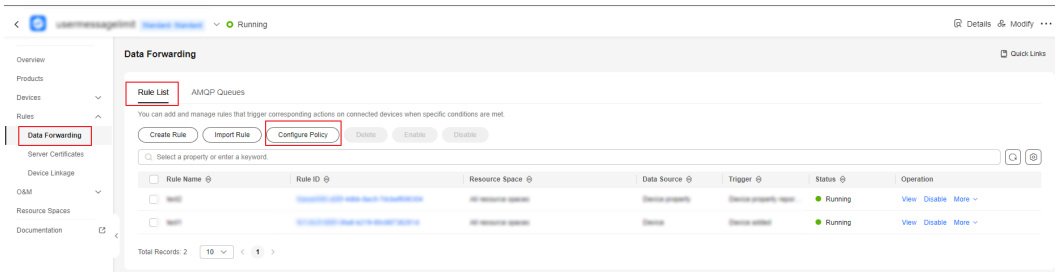
You can create up to four flow control policies for an IoTDA instance.

Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Rules** > **Data Forwarding**, and click **Configure Policy**. The **Flow Control Policies** page is displayed.

Figure 7-89 Data forwarding - Policy configuration



Step 3 In the displayed dialog box, configure parameters and click **OK**.

Figure 7-90 Data forwarding - Creating a flow control policy

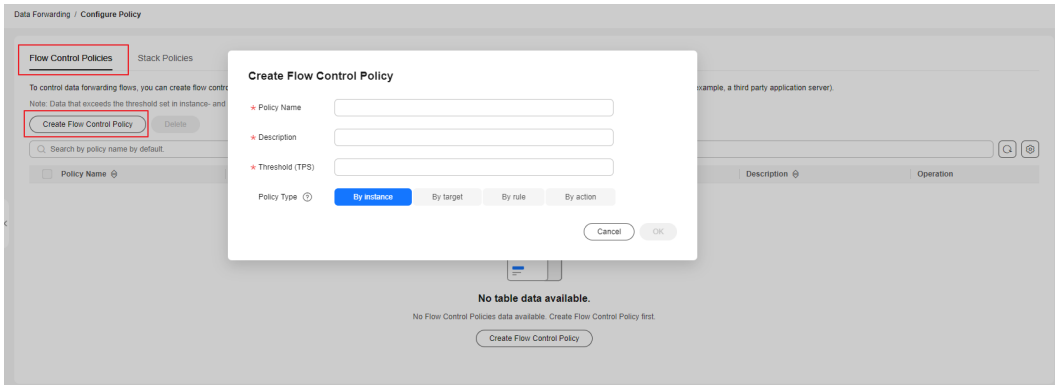


Table 7-20 Parameters

Parameter	Description
Policy Name	The value can contain 4 to 256 characters. Only letters, digits, and special characters (_?'#().,&%@!-) are allowed.
Description	Description of the policy. The value can contain 4 to 256 characters. Only letters, digits, and special characters (_?'#().,&%@!-) are allowed.
Threshold	The value ranges from 1 to 1000.
Policy Type	The options include **By instance**, **By target**, **By rule**, and **By action**.
Forwarding Target	Forwarding targets supported by the current instance. This parameter is available only when **Policy Type** is set to **By target**.
Bound Rule	Data forwarding rules on IoTDA. This parameter is available only when **Policy Type** is set to **By rule**.
Bound Action	Data forwarding actions on IoTDA. This parameter is available only when **Policy Type** is set to **By action**.

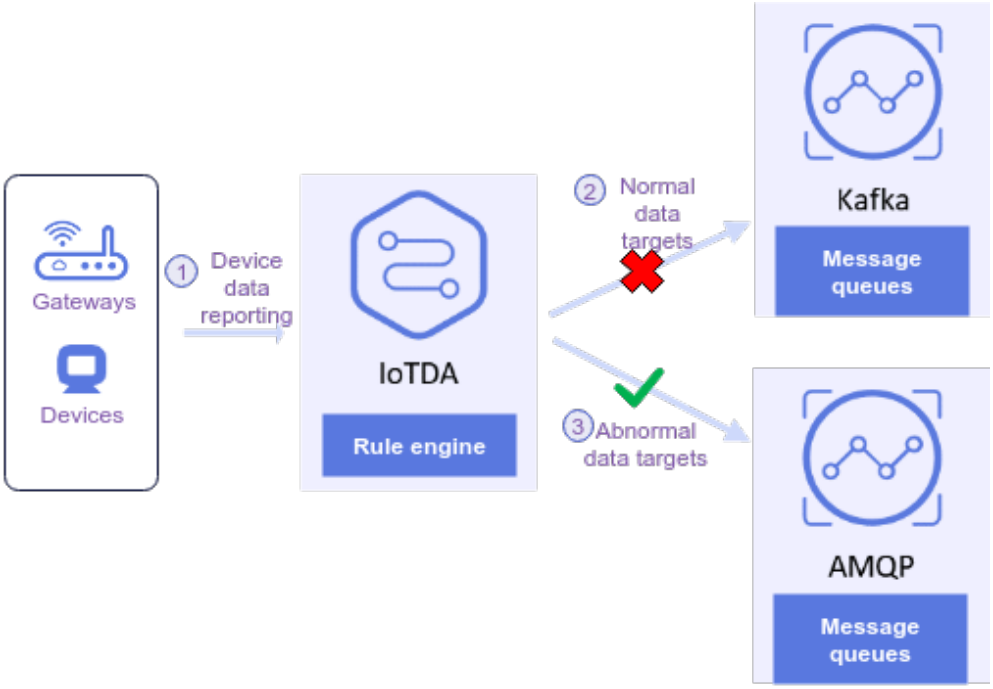
----End

7.10 Abnormal Data Target

Overview

When data is forwarded to other Huawei Cloud services or third-party applications, IoTDA stops message pushing if the target server is unreachable due to insufficient permissions or service unavailability. It checks the channel status every 3 minutes, and if the check result is normal, the channel will be restored. If you require real-time messages, you can configure abnormal data forwarding targets to obtain **abnormal data**. In this way, you can continue service processing and analyze failure causes, reducing the impact of single channel faults on services.

Figure 7-91 Example of abnormal data targets



NOTE

If the normal data target is unreachable within 24 hours, data is directly pushed to the configured abnormal data target. If the normal data target remains unreachable after 24 hours, the platform suspends data push.

Constraints

- If there is only 1 normal data target, 1 abnormal data target can be added.
- Up to five abnormal data targets can be created for each IoTDA instance.
- Supported rule data sources for abnormal data targets: device, device property, device message, device message status, device status, batch task, product, and device asynchronous command status.
- Configurable normal data targets for abnormal data forwarding: Data Ingestion Service (DIS), Distributed Message Service for Kafka, Object Storage Service (OBS), ROMA Connect, third-party application service (HTTP push), Distributed Message Service (DMS) for RocketMQ, FunctionGraph, GeminiDB Influx, Relational Database Service (RDS) for MySQL, MapReduce Service (MRS) Kafka, Blockchain Service (BCS), and Document Database Service (DDS).
- AMQP message queues can be used as abnormal data targets.

Data Format

The following is an example of the abnormal data forwarding format:

```
{
    "request_id": "2131d048-234f-4564-9190-6030234678ad",
    "rule_id": "6519d048-3b7f-442b-9190-6030773879cc",
    "action_id": "f376ab9f-d060-4fbf-a383-3e52af98ae9d",
    "channel": "MYSQL_FORWARDING",
```

```
    "base64_original_payload":
"ewogICAgInJlc291cmNlIjogImRldmljZS5tZXNzYWdlIiwKICAgICJldmVudCI6ICJyZXBvcnQiLAogICAgImV2ZW50
X3RpbWUiOiAiMjAyNDA2MDFUMDAwMDAwWiIsCiAgICAiZXZlbnRfdGltZV9tcyI6ICIyMDI0LTA2LTAxVDAwOjA
wOjAwLjAwMFoiLAogICAgInJlcXVlc3RfaWQiOiAiMjEzMWQwNDgtMjM0Zi00NTY0LTkxOTAtNjAzMDIzNDY3O
GFkIiwKICAgICJub3RpZnlfZGF0YSI6IHsKICAgICAgICAiaGVhZGVyIjogewogICAgICAgICAgICAiYXBwX2lkIjogIjU4
N2MyMzc3ZGRmYzQzMmI4OTMxZGNhYmNhODkyOTUyIiwKICAgICAgICAgICAgImRldmljZV9pZCI6ICIwMDA
wMDAwMDAwMDAwMCIsCiAgICAgICAgICAgICJub2RlX2lkIjogIjIwMjAwNzExMTk5OTkxMDAwIiwKICAgICAgI
CAgICAgInByb2R1Y3RfaWQiOiAiMTIzNDU2Nzg5MDAiLAogICAgICAgICAgICAiZ2F0ZXdheV9pZCI6ICIyMDIwM
DcxMTE5OTk5MTAwMCIKICAgICAgICB9LAogICAgICAgICJib2R5IjogewogICAgICAgICAgICAidG9waWMiOiAidG
VzdFRvcGljIiwKICAgICAgICAgICAgImNvbnRlbnQiOiAiV2VsY29tZSB0byBldWF3ZWkgQ2xvdWQgSW9UIERldml
jZSBBY2Nlc3MoSW9UREEpIgogICAgICAgIH0KICAgIH0KfQo=",
    "error_message": "Push failed. Cause:  Table 'xxx' doesn't exist. The iot platform will resume the push
within 3 minutes after the channel is restored."
}
```


Table 7-21 Abnormal data forwarding

Parameter	Type	Description
request_id	String	ID of the message that fails to be pushed, which is specified by the device or generated by the platform and is used to trace the service process.
rule_id	String	Unique ID of the rule triggering condition, which is allocated by the platform during rule creation.
action_id	String	Unique ID of the data target to which a push exception occurs. The ID is allocated by the platform during forwarding target creation.
channel	String	Type of the data target to which an exception occurs.
base64_original_payload	String	Base64-encoded raw stream data.
error_message	String	Error description.

NOTE

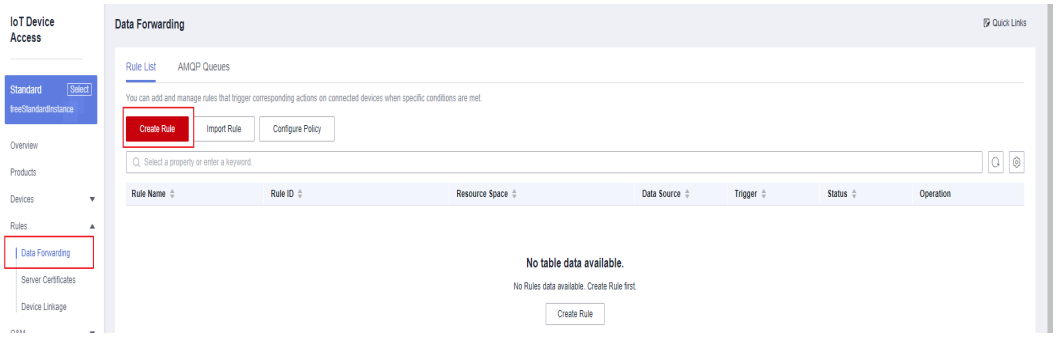
There may be additional fields during abnormal data forwarding. Ensure serialization compatibility when using the data format.

Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

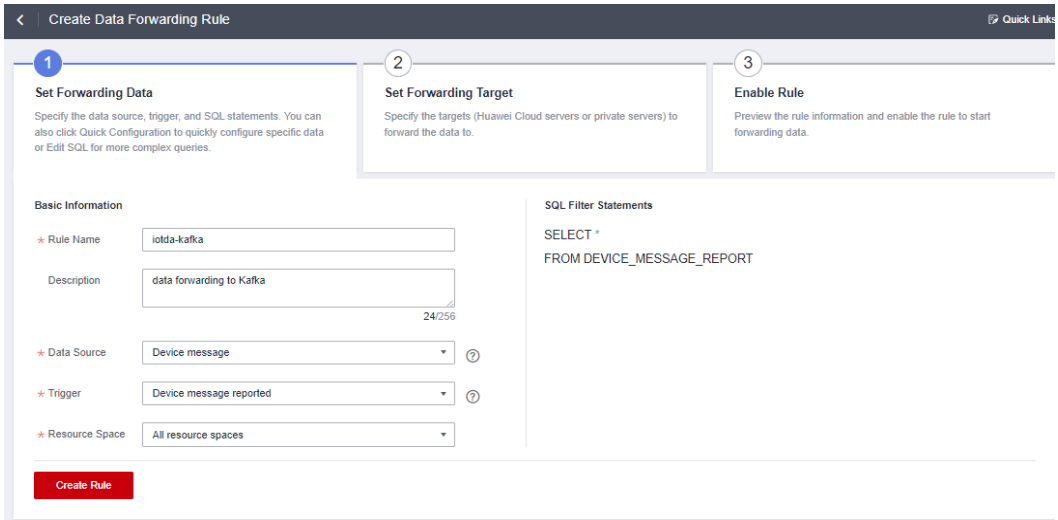
Step 2 In the navigation pane, choose **Rules** > **Data Forwarding**. On the displayed page, click **Create Rule**.

Figure 7-92 Creating a data forwarding rule



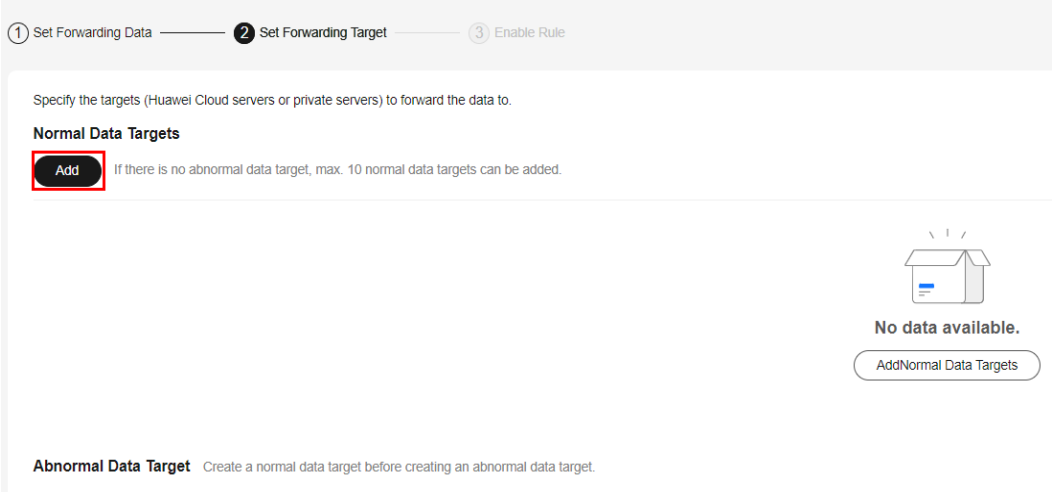
Step 3 On the displayed page, enter related information and click **Create Rule**.

Figure 7-93 Rules triggered by message reporting - forwarding data to Kafka



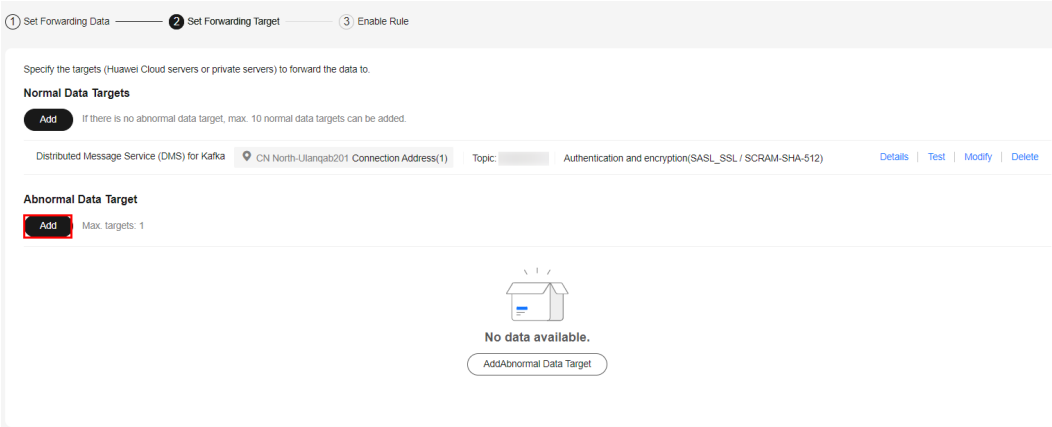
Step 4 Click the **Set Forwarding Target** tab and click **Add** to add a normal data target.

Figure 7-94 Adding a normal data target



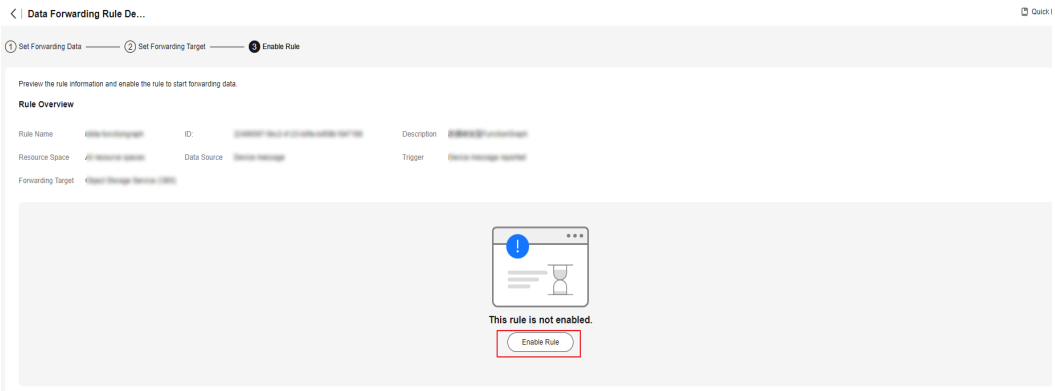
Step 5 Click **Add** to add an abnormal data target.

Figure 7-95 Adding an abnormal data target



Step 6 Click **Enable Rule**.

Figure 7-96 Data forwarding - Enabling a rule



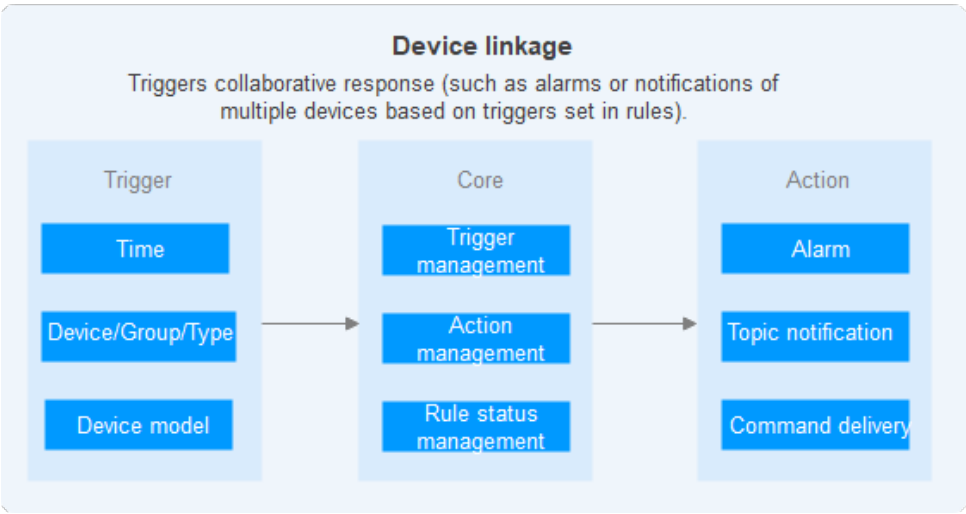
----End

7.11 Device Linkage

Overview

When specific conditions are met, the platform triggers collaborative response of multiple devices to implement device linkage and intelligent control. For example, when the battery level of a water meter drops to 20% or less, a low battery alarm is reported so that the battery can be replaced before it goes dead.

Figure 7-97 Device linkage architecture



To further explore device linkage, see **Triggering Alarms and Sending Email or SMS Notifications**.

7.11.1 Cloud Rules

Overview

If you set a cloud rule, IoTDA determines whether the rule triggering condition is met. If the condition is met, IoTDA performs actions you set, such as alarm reporting, topic notification, and command delivery.

Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Rules** > **Device Linkage**. Click **Create Rule**.

Step 3 Create a device linkage rule based on the table below.

Parameter	Description	Best Practice
Rule Name	Specify the name of a rule to create.	• **Automatic Device Shutdown Upon High Temperature** • **Triggering Alarms and Sending Email or SMS Notifications** • **Automatically Opening the Window upon High Gas Concentration** • **Monitoring Device Status Changes and Sending Notifications**
Activate upon creation	• Selected: The rule is activated upon creation. • Deselected: The rule is not activated after creation.	
Rule Type	• **Cloud**: The rule to create is executed on the platform. • **Device side**: The rule to create is delivered to devices for execution. Target devices must have the SDK with the device-side rule engine. For details, see **Device-side Rules**.	
Effective Period	• **Always effective**: There is no time limit. IoTDA always checks whether conditions are met. • **Specific time**: You can select a time segment during which the platform checks whether the conditions are met.	
Description	Describe the rule.	

Parameter	Description	Best Practice
Set Triggers	You can set whether all conditions or any of the conditions need to meet. ● **Device Property**: Properties reported by devices can be used as a trigger. For example, the device is powered off when the temperature reaches 80°C. – **Select product**: Select a specific product. – Select the device range: ▪ **All Devices**: Set the trigger for all devices under the selected product. ▪ **Specified device**: Set the trigger for a specified device under the selected product. – **Select service**: Select a service type. – **Select property**: Select a property. **NOTE** ▪ If the data type of a property is **int**, **long**, or **decimal**, you can select multiple operators. ▪ If the data type of a property is **string**, **date time**, or **jsonObject**, you can only select the equal sign (=) as the operator. – **Triggering Mechanism**: Select a trigger strategy. **Repetition suppression** is recommended. – **Data Validity Period (s)**: Specify the data validity period. For example, if **Data Validity Period** is set to 30 minutes, a device generates data at 19:00, and the platform receives the data at 20:00, the action is not triggered even if the conditions are met. ● **Timer**: Set the time at which the rule is triggered. It is usually used for periodic triggering conditions, such as turning off street lights at 07:00 every day. **NOTE** If **Timer** is selected, **Send notifications**, **Report alarms**, and **Clear alarms** cannot be selected for **Actions**. – **Triggered every day**: The rule is triggered at a specified time every day. – **Triggered by policy:** ▪ **Select a date and time**: start time for triggering the rule. ▪ **Repeat**: number of times that the rule can be triggered. The value ranges from 1 to 1440.	

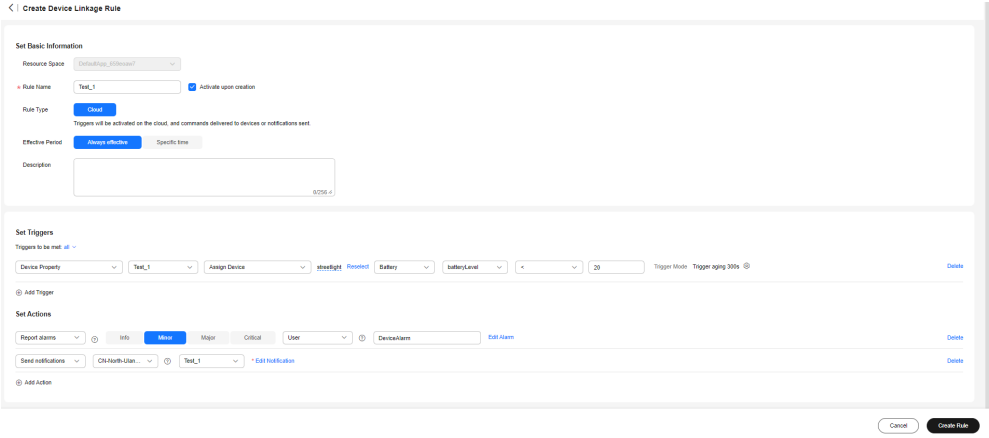
Parameter	Description	Best Practice
	▪ **Interval**: interval for triggering the rule after the start time. The value ranges from 1 to 1440, in units of minutes. ● **Device Status**: The online/offline status of a device can be used as a trigger. For example, if a device has been offline for 5 minutes, an alarm will be reported. – **Select product**: Select a specific product. – Select the device range: ▪ **All Devices**: Set the trigger for all devices under the selected product. ▪ **Specified device**: Set the trigger for a specified device under the selected product. – Select the device status: ▪ **Online**: The device status changes from offline to online. ▪ **Offline**: The device status changes from online to offline. ▪ **Online and Offline**: The device status changes. – **Duration**: duration of the new status after the device status change, in minutes. The value range is 0–60.	

Parameter	Description	Best Practice
Set Actions	Click **Add Action** to set the action to execute after the rule is triggered. ● **Deliver commands**: Select the device, service, and command to be delivered in sequence, and set the command delivery parameters. ● **Send notifications**: Select the region where the SMN service is located. If the platform has not been granted with the permissions to access SMN, perform the authorization as prompted. Click the corresponding link to go to the SMN console and set the topic. – **Message Title**: used as the email subject when an email is sent to an email subscriber. – **Message Type**: Use a temple or customize the settings. – **Message Content**: content of the message to be sent. – **Template**: Use the template defined by SMN. When sending messages, the variables in the template is replaced with corresponding parameter values. IoTDA defines some common template variables. After a rule is triggered, the following template variables will be replaced with specific values. {*ruleName*}: name of the triggered rule {*ruleId*}: ID of the triggered rule {*deviceId*}: ID of the device that triggers the rule {*deviceName*}: name of the device that triggers the rule {*productId*}: ID of the product to which the device that triggers the rule belongs {*productName*}: name of the product to which the device that triggers the rule belongs {*YYYY*}: year (UTC) when the rule is triggered. {*MM*}: month when the rule is triggered (UTC) {*DD*}: date when the rule is triggered (UTC) {*HH*}: hour (UTC) when the rule is triggered. {*mm*}: minute (UTC) when the rule is triggered {*ss*}: second (UTC) when the rule is triggered	

Parameter	Description	Best Practice
	NOTE Example SMN template: Time: {*YYYY*}-{*MM*}-{*DD*} {*HH*}:{*mm*}:{*ss*} Rule name: {*ruleName*} Rule ID: {*ruleId*} Product ID: {*productId*} Product name: {*productName*} Device ID: {*deviceId*} Device name: {*deviceName*} Event: device going online and offline After the device goes online and the rule is triggered, the received message is shown in the following figure.  • **Report alarms**: Define the alarm severity, name, isolation dimension, and content. When the configured condition is met, a device alarm is generated on the **Application Operations Management (AOM)** console. – Alarm severity: Options include **Info**, **Minor**, **Major**, and **Critical**. – Alarm isolation dimension: Options include **User**, **Resource Space**, and **Device**. Reported alarms carry different isolation dimension identifiers. If you select **Device** for the dimension, reported alarms will carry device IDs as isolation dimension identifiers. – Alarm name: name of the reported alarm. – Alarm content: content carried in the reported alarm. • **Clear alarms**: Define the alarm severity, name, isolation dimension, and content. If conditions are met, alarms reported by the device to the platform will be cleared. The parameters are the same as those for reporting alarms. **NOTE** In AOM, the alarm severity, alarm name, and alarm isolation dimension together identify an alarm. When an alarm is cleared, the three attributes must be the same as those specified during alarm reporting.	

Step 4 Click **Create Rule** in the lower right corner. Newly created rules are in the activated state by default. You can disable a rule in the **Status** column of the rule list.

Figure 7-98 Creating a linkage rule - BatteryProperty



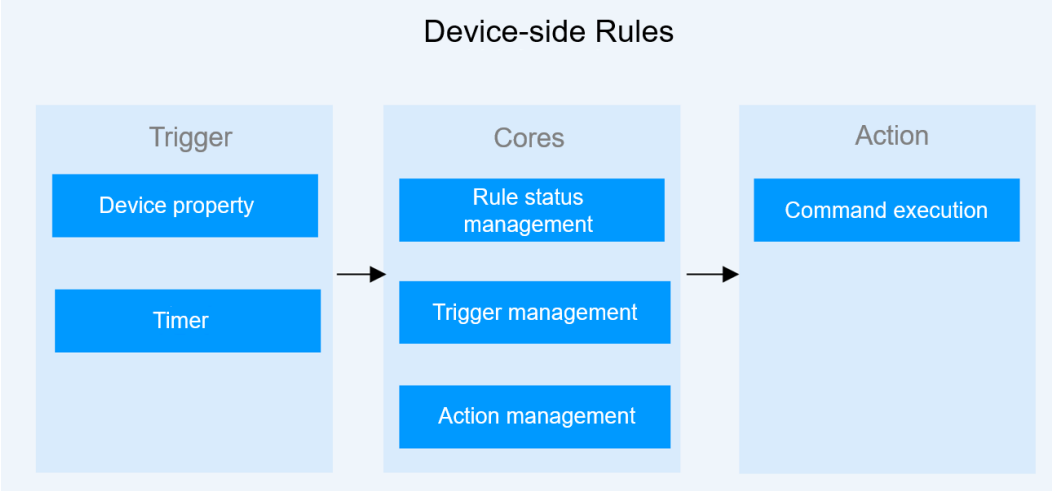
----End

7.11.2 Device-side Rules

Overview

Cloud rules are parsed and executed on the cloud. IoTDA determines whether triggering conditions are met and triggers corresponding device linkage actions. Device-side rules are device linkage rules delivered to devices, where the device-side rule engine parses and executes the rules. Device-side rules can still run on devices when the network is interrupted or devices cannot communicate with the platform. Device-side rules can extend application scenarios and improve device running stability and execution efficiency. For example, when the indoor light intensity is lower than 20, lights can be automatically turned on. This implements intelligent control independent of network devices.

Figure 7-99 Device-side rule architecture



For details, see **Basic Concepts**.

Scenarios

There are a large number of surveillance devices in highway tunnels. The network environment is complex and the network quality is unstable. However, emergency handling has high requirements on real-time network performance. Linkage between emergency devices cannot completely depend on cloud rules. Device-side rules are required to implement emergency plan linkage. Device linkage plans can be formulated in advance based on different situations such as fires and traffic accidents. Monitoring personnel can start device linkage plans with one click based on tunnel conditions. Device-side rules enable simultaneous status changes of different types of devices. This reduces dependency on network quality and improves overall device linkage efficiency. For example, if the temperature of a flue pipe is too high, the controller can be linked to open the drainage valve to reduce the temperature. If the concentration of carbon monoxide (CO) is too high, a COVI device can be linked to control fans for ventilation.

Constraints

- Device-side rules support only command delivery actions.
- Devices must be integrated with IoT Device SDK (C) v1.1.2 or later.
- Devices need to report the SDK version number to IoTDA using the APIs provided by the SDK.

Procedure

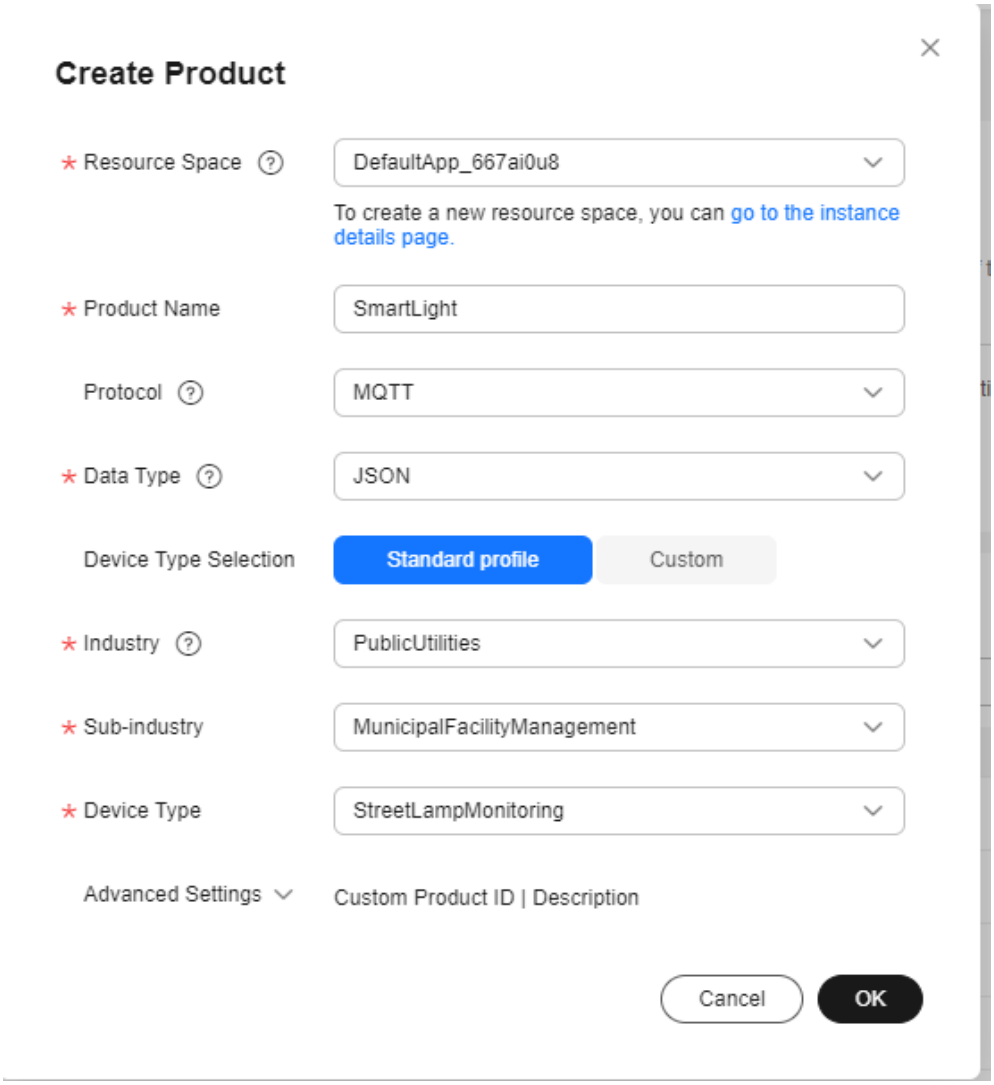
The following uses a smart street light system as an example to describe how to use device-side rules.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 Create a product and model.

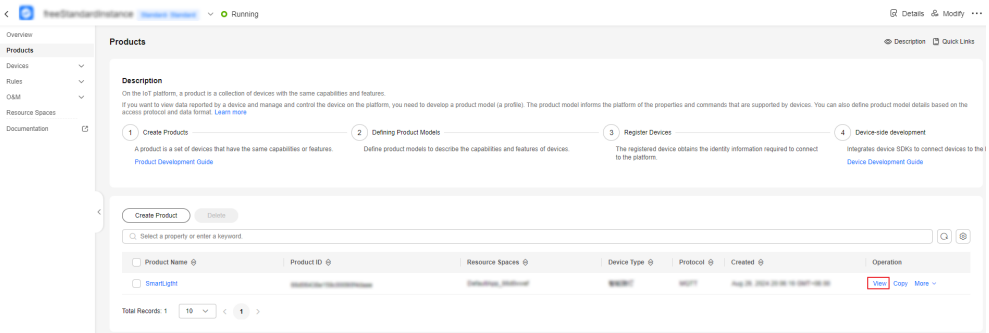
1. Log in to the **IoTDA console** and click the target instance card. Choose **Products** from the navigation pane and click **Create Product**. In the displayed dialog box, select **StreetLampMonitoring** for **Device Type**, enter the product name, and click **OK**.

Figure 7-100 Creating a product - SmartLight



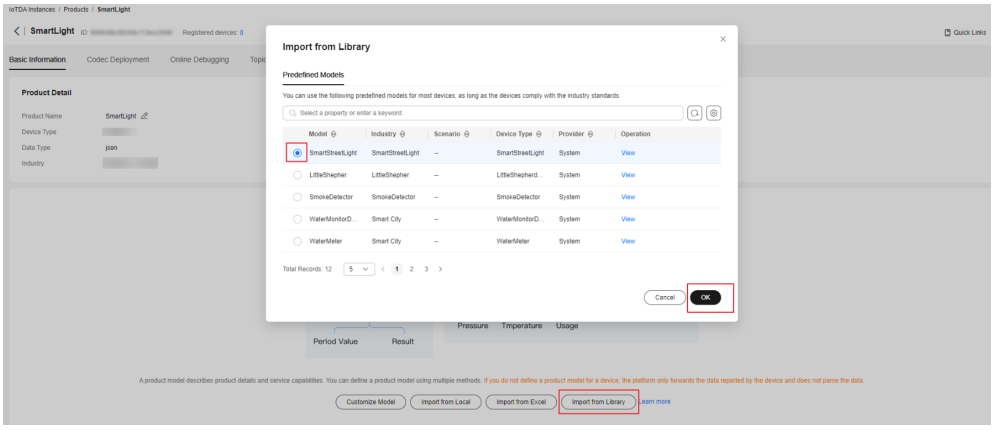
2. Locate the **SmartLight** product and click **View**.

Figure 7-101 Viewing SmartLight details



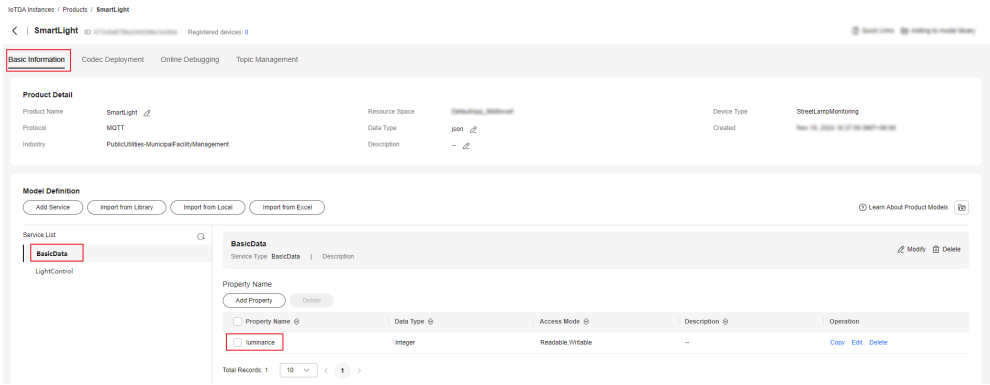
3. On the product details page, click **Import from Library**, select **SmartStreetLight**, and click **OK**.

Figure 7-102 Product - Import model SmartStreetLight



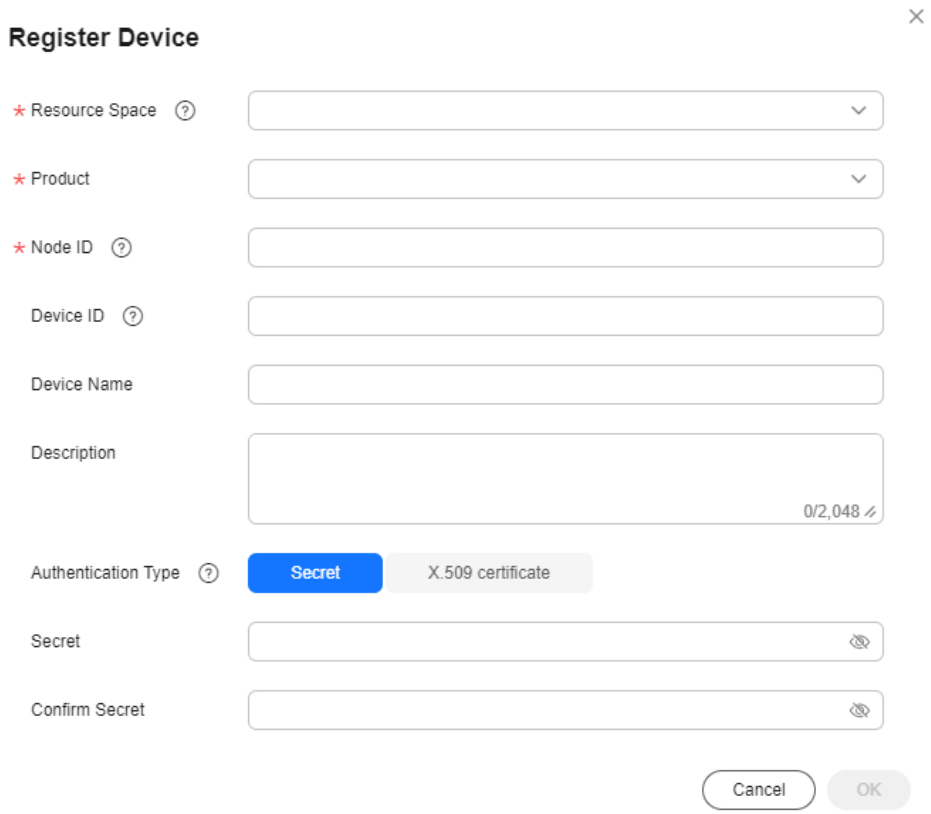
4. On the **Basic Information** page, **BasicData** and **LightControl** services are displayed. The **BasicData** service contains the **luminance** property. The **LightControl** service contains a **switch** command.

Figure 7-103 Model definition - SmartLight



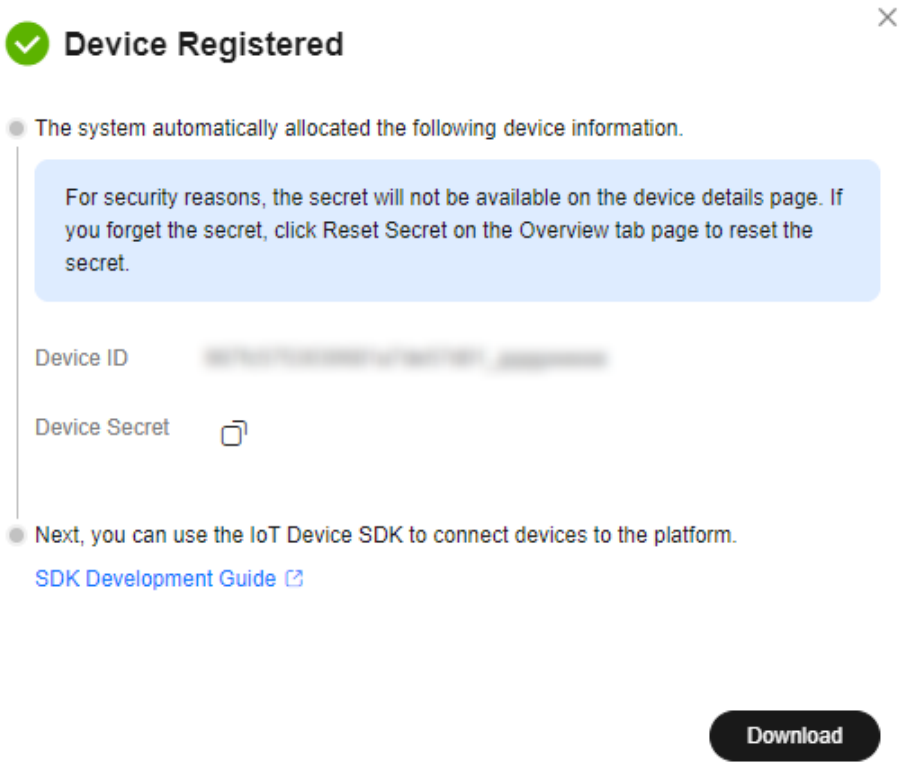
Step 3 In the navigation pane, choose **Devices** > **All Devices** and click **Register Device**. Select the resource space you select in **Step 2** and a product, enter a node ID, and click **OK**.

Figure 7-104 Device - Registering a secret device



Step 4 After the device is created, copy and save the device secret for later use.

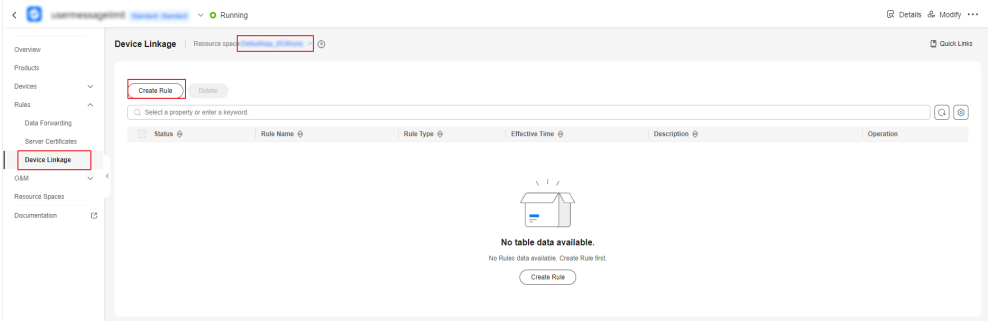
Figure 7-105 Device registered



Step 5 Create a rule.

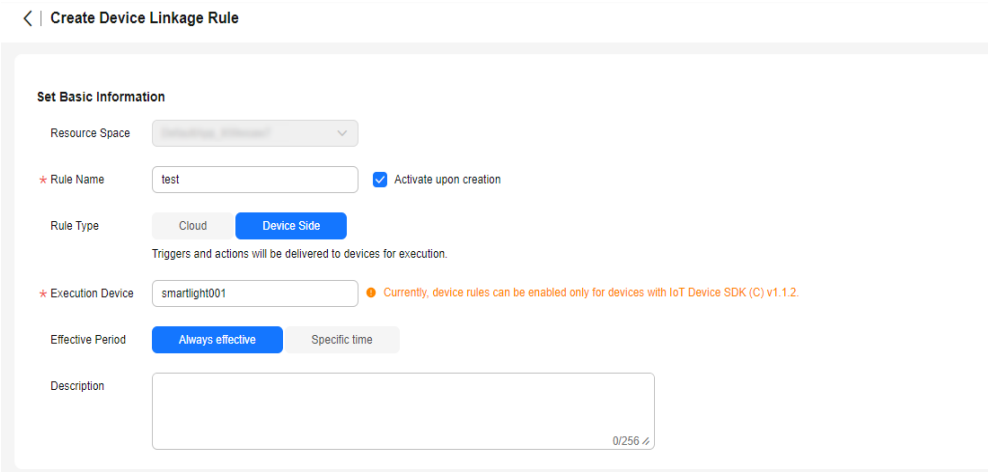
1. In the navigation pane, choose **Rules** > **Device Linkage**. In the upper part of the page, select the resource space of the product to which the device belongs. Click **Create Rule**.

   Figure 7-106 Device linkage - Creating a rule



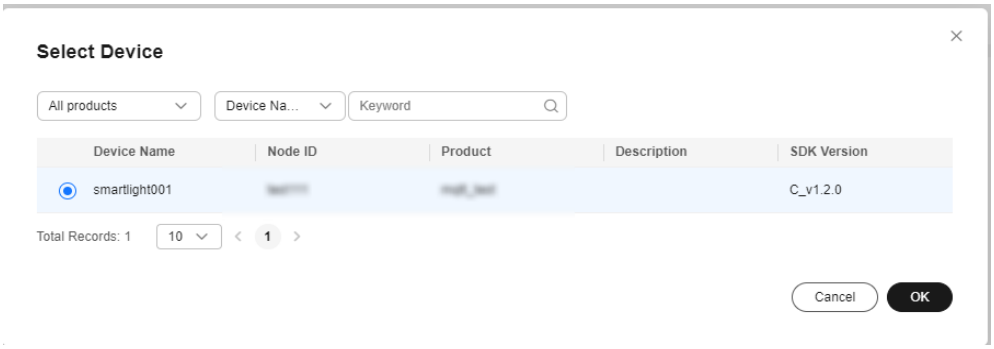
2. On the page for creating a rule, enter a rule name, select **Device Side** for **Rule Type**, and select a device for **Execution Device**. The rule will be delivered to the device you select for parsing and execution.

Figure 7-107 Creating a linkage rule - Basic information on the device side



3. Select **smartlight001** and click **OK**.

Figure 7-108 Creating a linkage rule - Selecting a device

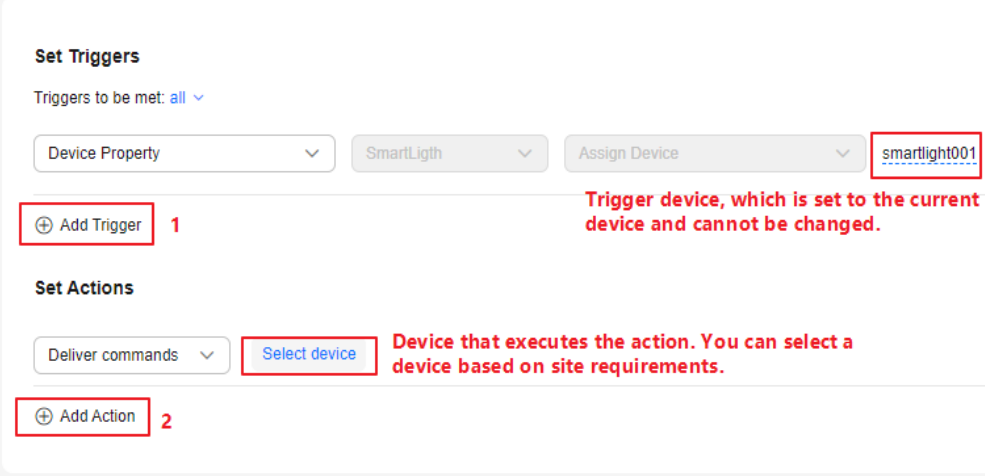


NOTE

Device-side rules can be created only for devices with the IoT Device SDK. Currently, only IoT Device SDK (C) v1.1.2 is supported.

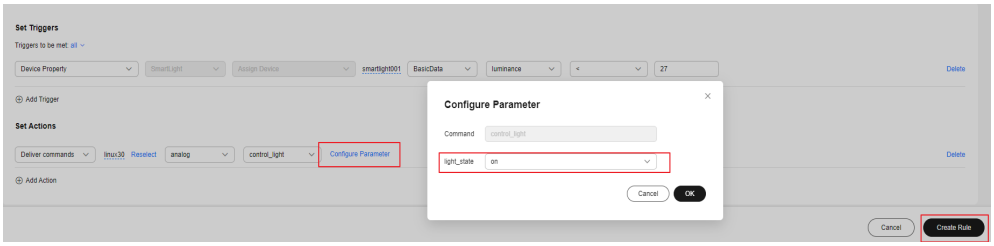
4. Click **Add Trigger**. The current device is used by default, and other devices are not available. Click **Add Action** and select the current device or other devices.

Figure 7-109 Creating a linkage rule - Conditions and actions



5. In **Set Triggers**, set the property trigger to **luminance<= 27**. In **Set Actions**, configure the **control_light** command and configure the parameter to set **light_state** to **on**.

Figure 7-110 Creating a linkage rule - Conditions and actions



Step 6 Create a device linkage rule based on the table below.

Table 7-22 Parameters

Parameter	Description
Rule Name	Specify the name of a rule to create.
Activate upon creation	Selected: The rule is activated upon creation. Deselected: The rule is not activated after creation.
Effective Period	• **Always effective**: There is no time limit. IoTDA always checks whether conditions are met. • **Specific time**: You can select a time segment during which the platform checks whether the conditions are met. **NOTE** Device-side rules are stored in the memory. When a device is powered off, rules stored on the device are cleared. When the device is restarted or powered on, the device updates all historical rules from IoTDA.
Description	Describe the rule.

Parameter	Description
Set Triggers	You can set whether all conditions or any of the conditions need to meet. **NOTE** If all conditions need to meet, **Device Property** and **Timer** cannot be both set as triggers at the same time. You can only set multiple device properties as triggers. Trigger type: Currently, only **Device Property** and **Timer** are supported. ● **Device Property**: The rule will be triggered when a device reports properties. – **Select service**: Select the corresponding service type. – **Select property**: Select a property in the data reported. **NOTE** – If the data type of a property is **int** or **decimal**, you can select multiple operators. – If the data type of a property is **string**, you can only select the equal sign (=) as the operator. ● **Timer**: You can select **Triggered every day** or **Triggered by policy**. – **Triggered every day**: Set the time at which the rule is triggered. It is usually used for periodic triggering conditions, such as turning off street lights at 07:00 every day. – **Triggered by policy** ▪ **Select a date and time**: start time for triggering the rule. ▪ **Repeat**: number of times that the rule can be triggered. The value ranges from 1 to 1440. ▪ **Interval**: interval for triggering the rule after the start time. The value ranges from 1 to 1440, in units of minutes.
Set Actions	Click **Add Action** to set the action to execute after the rule is triggered. **Deliver commands**: Select the device, service, and command to be delivered in sequence, and set the command delivery parameters.

Step 7 Click **Create Rule** in the lower right corner. Newly created rules are in the activated state by default. You can disable a rule in the **Status** column of the rule list.

Step 8 Compile the device-side code. In SDKs that supports device-side rules (only IoT Device SDK C is supported currently), you only need to implement the callback functions for property reporting and command processing. Click **here** to obtain the IoT Device SDK (C) and perform the following operations after the operations in **Preparations** are complete.

1. Open the **src/device_demo/device_demo.c** file and find the **HandleCommandRequest** function.

Figure 7-111 Command processing

```
void HandleCommandRequest(EN_IOTA_COMMAND *command)
{
    if (command == NULL) {
        return;
    }

    PrintfLog(EN_LOG_LEVEL_INFO, "device_demo: HandleCommandRequest(), messageId %d\n",
        command->mqtt_msg_info->messageId);

    PrintfLog(EN_LOG_LEVEL_INFO, "device_demo: HandleCommandRequest(), object_device_id %s\n",
        command->object_device_id);
    PrintfLog(EN_LOG_LEVEL_INFO, "device_demo: HandleCommandRequest(), service_id %s\n", command->service_id);
    PrintfLog(EN_LOG_LEVEL_INFO, "device_demo: HandleCommandRequest(), command_name %s\n", command->command_name);
    PrintfLog(EN_LOG_LEVEL_INFO, "device_demo: HandleCommandRequest(), paras %s\n", command->paras);
    PrintfLog(EN_LOG_LEVEL_INFO, "device_demo: HandleCommandRequest(), request_id %s\n", command->request_id);
                                                    <- Implement the command here.
    Test_CommandResponse(command->request_id); // response command
}
```

The following commands are use for demonstration only.

```
printf("----- execute command----- \n");
printf("service_id: %s\n", command->service_id);
printf("command_name: %s\n", command->command_name);
printf("paras: %s\n", command->paras);
```

2. Open the **src/device_demo/device_demo.c** file and find the **TestPropertiesReport** function.

Figure 7-112 Replacing the code

```
void Test_PropertiesReport()
{
    const int serviceNum = 2; // reported services' totol count
    ST_IOTA_SERVICE_DATA_INFO services[serviceNum];

    // ---------------the data of service1-----------------------------
    char *service1 = "{\"Load\":\"6\",\"ImbA_strVal\":\"7\"}";

    services[0].event_time =
        GetEventTimesStamp(); // if event_time is set to NULL, the time will be the iot-platform's time.
    services[0].service_id = "parameter";
    services[0].properties = service1;

    // ---------------the data of service2-----------------------------   Replace the code here.
    char *service2 = "{\"PhV_phsA\":\"9\",\"PhV_phsB\":\"8\"}";

    services[1].event_time = NULL;
    services[1].service_id = "analog";
    services[1].properties = service2;

    int messageId = IOTA_PropertiesReport(services, serviceNum, 0, NULL);
    if (messageId != 0) {
        PrintfLog(EN_LOG_LEVEL_ERROR, "device_demo: Test_PropertiesReport() failed, messageId %d\n", messageId);
    }

    MemFree(&services[0].event_time);
}
```

Use the following code:

```
const int serviceNum = 1; // reported services' total count
ST_IOTA_SERVICE_DATA_INFO services[serviceNum];

#define MAX_BUFFER_LEN 70
char propsBuffer[MAX_BUFFER_LEN];
// This is an example of obtaining a temperature value. Obtain the actual value from a sensor.
if(sprintf_s(propsBuffer, sizeof(propsBuffer), "{\"luminance\": %d}", 20) == -1){
printf("can't create string of properties\n");
return;
}

services[0].event_time = GetEventTimesStamp(); // if event_time is set to NULL, the time will be the
```

```
iot-platform's time.
services[0].service_id = "BasicData";
services[0].properties = propsBuffer;

int messageId = IOTA_PropertiesReport(services, serviceNum, 0, NULL);
if (messageId != 0) {
printf("report properties failed, messageId %d\n", messageId);
}
free(services[0].event_time);
```

3. Compile and run the SDK. You can see the corresponding command from the output.

```
----- execute command-----
service_id: BasicData
command_name: control_light
paras: {
        "light_state":       "on"
}
```

The preceding log is only an example. You need to implement the specific command processing code in **1**.

Before running commands across devices, ensure that the devices can communicate with each other. You may use different communication protocols, such as Wi-Fi, BLE, and ZigBee, so you need to call **IOTA_SetDeviceRuleSendMsgCallback** to register a custom sending function.

HandleDeviceRuleSendMsg is registered in the demo by default. You need to implement message sending in **HandleDeviceRuleSendMsg**. After receiving the message, the target device needs to parse and execute the command.

Figure 7-113 Parsing and executing commands

```
huaweicloud-iot-device-sdk-c > src > device_demo > device_demo.c > SetMyCallbacks()
1052
1053  int HandleDeviceRuleSendMsg(char *deviceId, char *message)
1054  {
1055      PrintfLog(EN_LOG_LEVEL_INFO, "device_demo: HandleDeviceRuleSendMsg(), deviceId is %s, the message is %s",
1056          deviceId, message);
1057      return 0;
1058  }
1059
1060  // ------------------------------------------------------------------------
1061
1062 > void SetAuthConfig() ···
1082
1083
1084  void SetMyCallbacks()
1085  {
1086      IOTA_SetProtocolCallback(EN_IOTA_CALLBACK_CONNECT_SUCCESS, HandleConnectSuccess);
1087      IOTA_SetProtocolCallback(EN_IOTA_CALLBACK_CONNECT_FAILURE, HandleConnectFailure);
1088
1089      IOTA_SetProtocolCallback(EN_IOTA_CALLBACK_DISCONNECT_SUCCESS, HandleDisConnectSuccess);
1090      IOTA_SetProtocolCallback(EN_IOTA_CALLBACK_DISCONNECT_FAILURE, HandleDisConnectFailure);
1091      IOTA_SetProtocolCallback(EN_IOTA_CALLBACK_CONNECTION_LOST, HandleConnectionLost);
1092
1093      IOTA_SetProtocolCallback(EN_IOTA_CALLBACK_SUBSCRIBE_SUCCESS, HandleSubscribesuccess);
1094      IOTA_SetProtocolCallback(EN_IOTA_CALLBACK_SUBSCRIBE_FAILURE, HandleSubscribeFailure);
1095
1096      IOTA_SetProtocolCallback(EN_IOTA_CALLBACK_PUBLISH_SUCCESS, HandlePublishSuccess);
1097      IOTA_SetProtocolCallback(EN_IOTA_CALLBACK_PUBLISH_FAILURE, HandlePublishFailure);
1098
1099      IOTA_SetMessageCallback(HandleMessageDown);
1100      IOTA_SetUserTopicMsgCallback(HandleUserTopicMessageDown);
1101      IOTA_SetCmdCallback(HandleCommandRequest);
1102      IOTA_SetPropSetCallback(HandlePropertiesSet);
1103      IOTA_SetPropGetCallback(HandlePropertiesGet);
1104      IOTA_SetEventCallback(HandleEventsDown);
1105      IOTA_SetShadowGetCallback(HandleDeviceShadowRsp);
1106      IOTA_SetDeviceRuleSendMsgCallback(HandleDeviceRuleSendMsg);
1107  }
```

----End

8 Monitoring and O&M

8.1 Message Trace

Message trace can be used to quickly locate and analyze faults that occur during device authentication, command delivery, data reporting, and data forwarding. The platform supports message trace for NB-IoT and MQTT devices. You can trace messages for up to 10 devices simultaneously.

Procedure

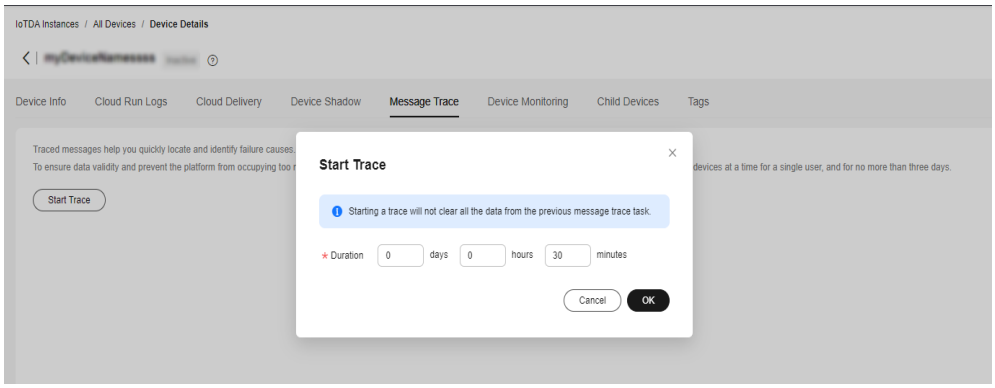
Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **All Devices**.

Step 3 Search for the device to trace and click **View** to access its details page.

Step 4 On the **Message Trace** tab page, click **Start Trace**, and set the message trace duration, which indicates the duration from the time when message trace starts to the time when message trace ends. You can also click **Edit Configuration** to modify the message trace configuration. The message trace duration is subject to the new one.

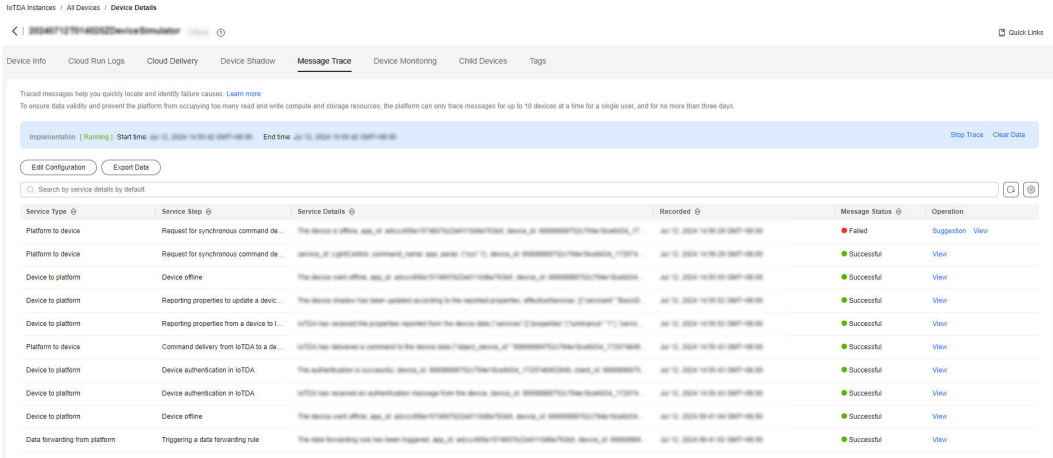
Figure 8-1 Message tracing - Starting message tracing



Step 5 View the services that are being traced. You can also click **Stop Trace** to stop the task.

If a large number of message trace records are displayed, you can filter the records by message status, service type, and recording time. If you need to further analyze the result data, you can export the data.

Figure 8-2 Message tracing - Viewing data



If the message status is **Failed**, you can click **View** to view the result details and locate the fault based on the failure handling suggestions.

----End

NOTE

During data reporting, you can specify **request_id** by adding **?request_id={request_id}** after a topic. For example, use **$oc/devices/{device_id}/sys/properties/report?request_id={request_id}** for property reporting. If you do not specify **request_id**, the platform automatically generates one.

8.2 Reports

IoTDA has various dashboards to intuitively present data.

In the navigation pane of the IoTDA console, choose **Overview**. The multiple reports displayed are from collected data of each instance and are valid for one month. The following table describes each report's name and function.

Table 8-1 Overview page

Report Name	Description	Data Update	Time Frame
Registered Devices	Number of registered devices.	Every hour	Hour, day, and month

Report Name	Description	Data Update	Time Frame
Online Devices	Number of online devices. This number is the highest collected in the time period (by hour or by day).	Every hour	Hour, day, and month
Device Messages	Number of device messages in either direction. Upstream messages are involved in message, property, and event reporting. Downstream messages are involved in message and command delivery, and property setting and query. **NOTE** Supported by the basic and enterprise editions.	Every hour	Hour, day, and month
Upstream Message TPS	Highest throughput of upstream messages per second, that is, the total number of messages sent from all devices to the platform in an instance per second. Only MQTT messages can be reported. The chart data comes from the average value every ten seconds. **NOTE** Supported by the standard edition.	Every minute	10 minutes, 30 minutes, 1 hour, or 1 day
User Messages	Number of user messages. This is mainly messages sent from devices to the cloud and vice versa. Any forward messages in excess of the messages sent from devices to the cloud are also counted.	Every hour	Hour, day, and month

In the navigation pane of the IoTDA console, choose **O&M** > **Reports** to see multiple O&M reports. Click the + icon in the upper right corner of a report to view data by instance or resource space. Each report is valid for one month. The following table describes each report's name and function.

Table 8-2 O&M reports

Report Name	Description	Data Update	Time Frame
Device Connection Status	Number of devices (and percentage of total number of devices) in each status. Statuses: **Online**, **Inactive**, **Offline**, **Abnormal**	Every hour	-

Report Name	Description	Data Update	Time Frame
Device Messages	Messages reported: number of messages reported by devices to the platform. Downstream messages: number of messages delivered by the platform to devices.	Every hour	Hour and day
MQTT Reported Messages TPS	Maximum number of upstream requests sent by MQTT devices to the platform per second in the current instance. The chart data comes from the average value every ten seconds. **NOTE** Supported by the standard edition.	Every minute	10 minutes, 30 minutes, 1 hour, or 1 day
MQTT Concurrent Connection Setup TPS	The most new connection requests from MQTT devices per second. The chart data comes from the average value every ten seconds. **NOTE** Supported by the standard edition.	Every minute	10 minutes, 30 minutes, 1 hour, or 1 day
General Device Trends	Trends in the number of devices. Total and online devices counted separately.	Every hour	Hour and day
Device Online Trends	Trends in the number (and percentage of total number devices) of devices. Online and offline devices counted separately.	Every hour	Hour and day
Devices by Status	Trends in the number of devices in each status. Statuses: **Inactive**, **Abnormal**, **Offline**	Every hour	Hour and day
Number of Online Devices (Accumulated)	Total number of online devices. **NOTE** Supported by the standard edition.	Every hour	Hour and day

Report Name	Description	Data Update	Time Frame
Software Upgrade Statuses	Number of upgrades of device software from the start. Successful and failed upgrades counted separately.	Every hour	-
Firmware Upgrade Statuses	Number of upgrades of device firmware from the start. Successful and failed upgrades counted separately.	Every hour	-
Device Configuration Statuses	Number of updates of device configuration from the start. Successful and failed updates counted separately.	Every hour	-

For more reports, log in to the **AOM console**, and choose **Monitoring** > **Cloud Service Monitoring** > **IoT** > **IoT Device Access (IoTDA)**. Currently, AOM shows you the IoTDA monitoring information by instance or resource space.

Table 8-3 Dashboards

Report Name	Description	Data Update	Time Frame
Device Status	Number of devices in each status. Statuses: **Online**, **Inactive**, **Offline**, **Abnormal**.	Every 10 minutes	1 hour, 6 hours, 12 hours, 1 day, or 7 days
General Device Trends	Trends in the number of devices. Total, online, and offline devices counted separately.	Every 10 minutes	1 hour, 6 hours, 12 hours, 1 day, or 7 days
Data Transfer Trend	Trends in the number of data transfers. AMQP transfers and HTTP message pushes counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days

Report Name	Description	Data Update	Time Frame
Data Report Trend	Trends in the number of reporting records. Reporting of NB-IoT data, MQTT events, MQTT properties, and MQTT messages counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days

Table 8-4 Metrics

Report Name	Description	Data Update	Time Frame
Total Devices	Trends in the total number of devices and the number of devices in each status. Statuses: **Online**, **Offline**, **Abnormal**, **Inactive**	Every 10 minutes	1 hour, 6 hours, 12 hours, 1 day, or 7 days
Reported NB Data Records	Trends in the number of NB-IoT data reporting records. Total, successful, and failed reporting counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
Reported MQTT Events	Trends in the number of MQTT event reporting records. Total, successful, and failed reporting counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
Reported MQTT Properties	Trends in the number of MQTT property reporting records. Total, successful, and failed reporting counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
Reported MQTT Messages	Trends in the number of MQTT message reporting records. Total, successful, and failed reporting counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
AMQP Transfers	Trends in the number of AMQP transfers. Total, successful, and failed transfers counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
FunctionGraph Transfers	Trends in the number of FunctionGraph transfers. Total, successful, and failed transfers counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days

Report Name	Description	Data Update	Time Frame
MRS Kafka Transfers	Trends in the number of MRS Kafka transfers. Total, successful, and failed transfers counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
MQTT Transfers	Trends in the number of MQTT transfers. Total, successful, and failed transfers counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
MySQL Transfers	Trends in the number of MySQL transfers. Total, successful, and failed transfers counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
InfluxDB Transfers	Trends in the number of InfluxDB transfers. Total, successful, and failed transfers counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
HTTP Message Pushes	Trends in the number of HTTP message pushes. Total, successful, and failed pushes counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
OBS Transfers	Trends in the number of OBS transfers. Total, successful, and failed transfers counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
DMS Kafka Transfers	Trends in the number of DMS for Kafka transfers. Total, successful, and failed transfers counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
DIS Transfers	Trends in the number of DIS transfers. Total, successful, and failed transfers counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
ROMA Transfers	Trends in the number of ROMA Connect transfers. Total, successful, and failed transfers counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
LTS Transfers	Trends in the number of LTS transfers. Total, successful, and failed transfers counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days

Report Name	Description	Data Update	Time Frame
BCS Huawei Cloud Blockchain Transfers	Trends in the number of BCS Huawei Cloud Blockchain transfers. Total, successful, and failed transfers counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
BCS-Hyperledger Fabric Enhanced Edition Transfers	Trends in the number of BCS-Hyperledger Fabric Enhanced Edition transfers. Total, successful, and failed transfers counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days
MongoDB Transfers	Trends in the number of MongoDB transfers. Total, successful, and failed transfers counted separately.	Every minute	1 hour, 6 hours, 12 hours, 1 day, or 7 days

To obtain report data using an AOM API, see **Querying Monitoring Data**. The following tables show the details about custom parameters of IoTDA metrics. In Table 5, **Name** corresponds to **metrics[].metric.dimensions[].name** and **Value** to **metrics[].metric.dimensions[].value**. In Table 6, **Namespace** corresponds to **metrics[].metric.namespace** and **MetricName** to **metrics[].metric.metricName**.

Table 8-5 Dimensions

Name	Value
app	Resource space ID
instance	Instance ID
taskType	Task type: software upgrade status (**softwareUpgrade**), firmware upgrade status (**firmwareUpgrade**), and device configuration status (**deviceConfig**)

Table 8-6 Metrics and namespaces

Report Name	Namespace	MetricName
Total Devices	IoTDA.DEVICE_STATUS	Total number of devices: **iotda_device_status_totalCount**
		Number of online devices: **iotda_device_status_onlineCount**

Report Name	Namespace	MetricName
		Number of offline devices: **iotda_device_status_offlineCount**
		Number of abnormal devices: **iotda_device_status_abnormalCount**
		Number of inactive devices: **iotda_device_status_inactiveCount**
Reported NB Data Records	IoTDA.NB_DATA_REPORT	Total number of reported NB-IoT data records: **iotda_south_dataReport_totalCount**
		Number of NB-IoT data reporting successes: **iotda_south_dataReport_successCount**
		Number of NB-IoT data reporting failures: **iotda_south_dataReport_failedCount**
Reported MQTT Events	IoTDA.EVENT_UP	Total number of reported MQTT event records: **iotda_south_eventUp_totalCount**
		Number of MQTT event reporting successes: **iotda_south_eventUp_successCount**
		Number of MQTT event reporting failures: **iotda_south_eventUp_failedCount**
Reported MQTT Properties	IoTDA.PROPERTIES_REPORT	Total number of reported MQTT property records: **iotda_south_propertiesReport_totalCount**
		Number of MQTT property reporting successes: **iotda_south_propertiesReport_successCount**
		Number of MQTT property reporting failures: **iotda_south_propertiesReport_failedCount**

Report Name	Namespace	MetricName
Reported MQTT Messages	IoTDA.MESSAGE_UP	Total number of reported MQTT message records: **iotda_south_messageUp_totalCount**
		Number of MQTT message reporting successes: **iotda_south_messageUp_successCount**
		Number of MQTT message reporting failures: **iotda_south_messageUp_failedCount**
AMQP Transfers	IoTDA.AMQP_FORWARDING	Total number of AMQP transfers: **iotda_amqp_forwarding_totalCount**
		Number of successful AMQP transfers: **iotda_amqp_forwarding_successCount**
		Number of failed AMQP transfers: **iotda_amqp_forwarding_failedCount**
FunctionGraph Transfers	IoTDA.FUNCTIONGRAPH_FORWARDING	Total number of FunctionGraph transfers: **iotda_functionGraph_forwarding_totalCount**
		Number of successful FunctionGraph transfers: **iotda_functionGraph_forwarding_successCount**
		Number of failed FunctionGraph transfers: **iotda_functionGraph_forwarding_failedCount**
MRS Kafka Transfers	IoTDA.MRS_KAFKA_FORWARDING	Total number of MRS Kafka transfers: **iotda_mrsKafka_forwarding_totalCount**
		Number of successful MRS Kafka transfers: **iotda_mrsKafka_forwarding_successCount**

Report Name	Namespace	MetricName
		Number of failed MRS Kafka transfers: **iotda_mrsKafka_forwarding_failedCount**
MQTT Transfers	IoTDA.MQTT_FORWARDING	Total number of MQTT transfers: **iotda_mqtt_forwarding_totalCount**
		Number of successful MQTT transfers: **iotda_mqtt_forwarding_successCount**
		Number of failed MQTT transfers: **iotda_mqtt_forwarding_failedCount**
MySQL Transfers	IoTDA.MYSQL_FORWARDING	Total number of MySQL transfers: **iotda_mysql_forwarding_totalCount**
		Number of successful MySQL transfers: **iotda_mysql_forwarding_successCount**
		Number of failed MySQL transfers: **iotda_mysql_forwarding_failedCount**
InfluxDB Transfers	IoTDA.INFLUXDB_FORWARDING	Total number of InfluxDB transfers: **iotda_influxDB_forwarding_totalCount**
		Number of successful InfluxDB transfers: **iotda_influxDB_forwarding_successCount**
		Number of failed InfluxDB transfers: **iotda_influxDB_forwarding_failedCount**
HTTP Message Pushes	IoTDA.HTTP_FORWARDING	Total number of HTTP message push transfers: **iotda_http_forwarding_totalCount**
		Number of successful HTTP message push transfers: **iotda_http_forwarding_successCount**

Report Name	Namespace	MetricName
		Number of failed HTTP message push transfers: **iotda_http_forwarding_failedCount**
OBS Transfers	IoTDA.OBS_FORWARDING	Total number of OBS transfers: **iotda_obs_forwarding_totalCount**
		Number of successful OBS transfers: **iotda_obs_forwarding_successCount**
		Number of failed OBS transfers: **iotda_obs_forwarding_failedCount**
DMS Kafka Transfers	IoTDA.DMS_KAFKA_FORWARDING	Total number of DMS for Kafka transfers: **iotda_dmsKafka_forwarding_totalCount**
		Number of successful DMS for Kafka transfers: **iotda_dmsKafka_forwarding_successCount**
		Number of failed DMS for Kafka transfers: **iotda_dmsKafka_forwarding_failedCount**
DIS Transfers	IoTDA.DIS_FORWARDING	Total number of DIS transfers: **iotda_dis_forwarding_totalCount**
		Number of successful DIS transfers: **iotda_dis_forwarding_successCount**
		Number of failed DIS transfers: **iotda_dis_forwarding_failedCount**
ROMA Transfers	IoTDA.ROMA_FORWARDING	Total number of ROMA Connect transfers: **iotda_roma_forwarding_totalCount**
		Number of successful ROMA Connect transfers: **iotda_roma_forwarding_successCount**

Report Name	Namespace	MetricName
		Number of failed ROMA Connect transfers: **iotda_roma_forwarding_failedCount**
LTS Transfers	IoTDA.LTS_FORWARDING	Total number of LTS transfers: **iotda_lts_forwarding_totalCount**
		Number of successful LTS transfers: **iotda_lts_forwarding_successCount**
		Number of failed LTS transfers: **iotda_lts_forwarding_failedCount**
BCS Huawei Cloud Blockchain Transfers	IoTDA.BCS_HW_FORWARDING	Total number of BCS Huawei Cloud blockchain transfers: **iotda_bcshw_forwarding_totalCount**
		Number of successful BCS Huawei Cloud blockchain transfers: **iotda_bcshw_forwarding_successCount**
		Number of failed BCS Huawei Cloud blockchain transfers: **iotda_bcshw_forwarding_failedCount**
BCS-Hyperledger Fabric Enhanced Edition Transfers	IoTDA.BCS_FABRIC_FORWARDING	Total number of BCS-Hyperledger Fabric Enhanced Edition transfers: **iotda_bcsfabric_forwarding_totalCount**
		Number of successful BCS-Hyperledger Fabric Enhanced Edition transfers: **iotda_bcsfabric_forwarding_successCount**
		Number of failed BCS-Hyperledger Fabric Enhanced Edition transfers: **iotda_bcsfabric_forwarding_failedCount**
MongoDB Transfers	IoTDA.MONGODB_FORWARDING	Total number of MongoDB transfers: **iotda_mongodb_forwarding_totalCount**

Report Name	Namespace	MetricName
		Number of successful MongoDB transfers: **iotda_mongodb_forwarding_successCount**
		Number of failed MongoDB transfers: **iotda_mongodb_forwarding_failedCount**
Software and Firmware Upgrades/ Remote Configuration	AOM.IoTDA	Number of successes: **count: iotda_batchtask_success_count**
		Number of failures: **iotda_batchtask_failure_count**

8.3 Alarms

The IoT platform generates an alarm when it detects that the alarm triggering condition set in a rule is met or the device message reporting rate exceeds the threshold preset on the platform. Pay close attention to the alarms and handle them in a timely manner to ensure the normal device running.

Alarms are classified into rule alarms, system alarms, and custom metric alarms.

- Rule alarms: If you set the action **Report alarms** when configuring a device linkage **rule** and define the alarm properties and severity, the platform reports an alarm when the trigger condition is met. For example, if a smart water meter does not report data for three consecutive days, the platform generates an alarm to notify maintenance personnel of the water meter fault. Maintenance personnel then locate the faulty water meter based on the alarm information and repair it promptly.
- System alarms: When some resources of a user, for example, the number of devices, reach the upper limit of the user quota, the IoTDA platform reports a system alarm to the AOM. This type of alarm is automatically triggered by the IoTDA platform, but notification rules need to be configured. **Table 8-7** lists the system alarms.

Table 8-7 System alarms

Alarm	Description
MQTT Message Flow Control for a Single Device	When the volume of data sent by an MQTT device per second exceeds the threshold (3 KB/s by default), the platform starts flow control on the MQTT device and generates this alarm.

Alarm	Description
Device Upstream Messages Exceeding the Tenant Flow Control	The sum of the upstream message rate and connection setup rate exceeds the threshold. (PUBLISH indicates upstream message, CONNECT indicates connection setup, and BANDWIDTH indicates bandwidth.) By default, the rate of upstream messages is 500 messages per second in the basic edition, and the rate of link setup is 100 messages per second in the basic edition. For details about the standard and enterprise editions, see **Specifications**. If the rate exceeds the default value, flow control will be performed and an alarm will be generated.
Number of User Devices Reaching the Threshold	This alarm is generated when the number of registered user devices reaches 80% or 100% of the instance threshold (50,000 for the basic edition, and 20 times of the number of online devices for the standard or enterprise edition. For details, see **Specifications**).
Number of Online User Devices Reaching the Threshold	This alarm is generated when the number of online user devices reaches 80% or 100% of the threshold. (The threshold depends on the number of purchased units. For the standard or enterprise edition, see **Specifications**.) When the number of online user devices exceeds the threshold, device access is rejected. The alarm is triggered once an hour.
Number of Child Devices Under a Gateway Reaching the Threshold	This alarm is generated when the number of child devices under a gateway reaches 80% or 100% of the threshold.
Linkage Rule Triggering Concurrency Threshold	This alarm is generated when the number of linkage rules triggered per second exceeds the threshold (10/s for the basic or standard edition and 100/s for the enterprise edition), and flow control is triggered on the excess part. This alarm is triggered only once a day.
Number of API Calls from a Tenant Reaching the Flow Control Threshold	This alarm is generated when the TPS of API calls made by a tenant exceeds the threshold. (Unless otherwise specified, the default limit of an API is 50/s. Maximum number of API calls made by an account per second: 100/s for the basic and standard editions.) Flow control is triggered on the excess part. This alarm is triggered only once a day.
Dafa Forwarding Target Added to the Blacklist	This alarm is generated when the number of data forwarding failures reaches a specified value (10 by default) and the current forwarding target is added to the blacklist.

- Custom metric alarms: You can log in to the **AOM 1.0** or **AOM 2.0** console to configure custom metric alarms. For details, see **Configuration Procedure for AOM 1.0**. Currently, the following metrics are supported.

Table 8-8 Custom alarm metrics

Metric	Name
Total number of devices	iotda_device_status_totalCount
Number of online devices	iotda_device_status_onlineCount
Number of offline devices	iotda_device_status_offlineCount
Number of abnormal devices	iotda_device_status_abnormalCount
Number of inactive devices	iotda_device_status_inactiveCount
Number of activated devices	iotda_device_status_activeCount
Number of online devices (accumulated)	iotda_device_status_dailyOnlineCount
Total number of reported NB-IoT data records	iotda_south_dataReport_totalCount
Number of NB-IoT data reporting failures	iotda_south_dataReport_failedCount
Total number of MQTT event reporting times	iotda_south_eventUp_totalCount
Number of MQTT event reporting successes	iotda_south_eventUp_successCount
Number of MQTT event reporting failures	iotda_south_eventUp_failedCount
Total number of MQTT property reporting times	iotda_south_propertiesReport_totalCount
Number of MQTT property reporting successes	iotda_south_propertiesReport_successCount
Number of MQTT property reporting failures	iotda_south_propertiesReport_failedCount
Total number of MQTT message reporting times	iotda_south_messageUp_totalCount

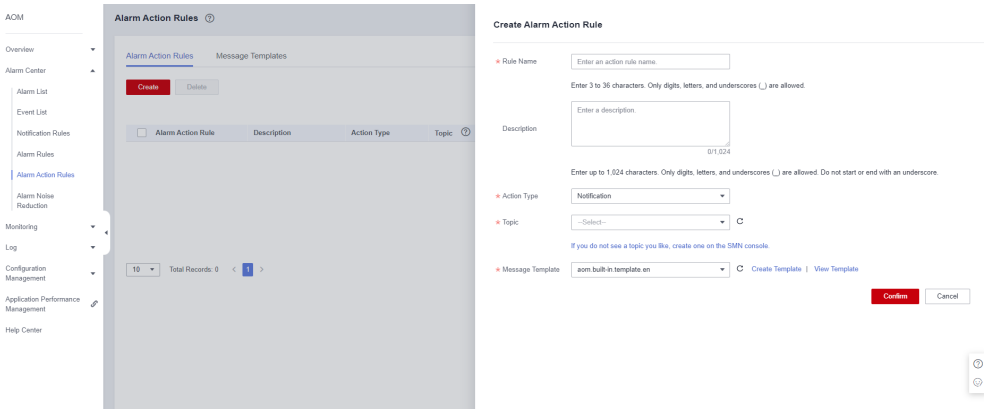
Metric	Name
Number of MQTT message reporting successes	iotda_south_messageUp_successCount
Number of MQTT message reporting failures	iotda_south_messageUp_failedCount
AMQP transfers	iotda_amqp_forwarding_totalCount
Number of AMQP transfer successes	iotda_amqp_forwarding_successCount
Number of AMQP transfer failures	iotda_amqp_forwarding_failedCount
FunctionGraph transfers	iotda_functionGraph_forwarding_totalCount
Number of FunctionGraph transfer successes	iotda_functionGraph_forwarding_successCount
Number of FunctionGraph transfer failures	iotda_functionGraph_forwarding_failedCount
MRS Kafka transfers	iotda_mrsKafka_forwarding_totalCount
Number of MRS Kafka transfer successes	iotda_mrsKafka_forwarding_successCount
Number of MRS Kafka transfer failures	iotda_mrsKafka_forwarding_failedCount
MQTT transfers	iotda_mqtt_forwarding_totalCount
Number of MQTT transfer successes	iotda_mqtt_forwarding_successCount
Number of MQTT transfer failures	iotda_mqtt_forwarding_failedCount
MySQL transfers	iotda_mysql_forwarding_totalCount
Number of MySQL transfer successes	iotda_mysql_forwarding_successCount
Number of MySQL transfer failures	iotda_mysql_forwarding_failedCount
InfluxDB transfers	iotda_influxDB_forwarding_totalCount
Number of InfluxDB transfer successes	iotda_influxDB_forwarding_successCount
Number of InfluxDB transfer failures	iotda_influxDB_forwarding_failedCount
HTTP message pushes	iotda_http_forwarding_totalCount

Metric	Name
Number of HTTP message push transfer successes	iotda_http_forwarding_successCount
Number of HTTP message push transfer failures	iotda_http_forwarding_failedCount
OBS transfers	iotda_obs_forwarding_totalCount
Number of OBS transfer successes	iotda_obs_forwarding_successCount
Number of OBS transfer failures	iotda_obs_forwarding_failedCount
DMS Kafka transfers	iotda_dmsKafka_forwarding_totalCount
Number of DMS Kafka transfer successes	iotda_dmsKafka_forwarding_successCount
Number of DMS Kafka transfer failures	iotda_dmsKafka_forwarding_failedCount
DIS transfers	iotda_dis_forwarding_totalCount
Number of DIS transfer successes	iotda_dis_forwarding_successCount
Number of DIS transfer failures	iotda_dis_forwarding_failedCount
ROMA transfers	iotda_roma_forwarding_totalCount
Number of ROMA Connect transfer successes	iotda_roma_forwarding_successCount
Number of ROMA Connect transfer failures	iotda_roma_forwarding_failedCount
LTS transfers	iotda_lts_forwarding_totalCount
Number of LTS transfer successes	iotda_lts_forwarding_successCount
Number of LTS transfer failures	iotda_lts_forwarding_failedCount

Configuration Procedure for AOM 1.0

Step 1 Log in to the **AOM** console. In the navigation pane, choose **Alarm Center** > **Alarm Action Rules**. Click **Create** and configure parameters.

Figure 8-3 Creating an alarm action rule



Step 2 In the navigation pane, choose **Alarm Center** > **Alarm Rules**. Click **Create Alarm Rule** in the upper right corner.

Step 3 Setting a threshold alarm rule

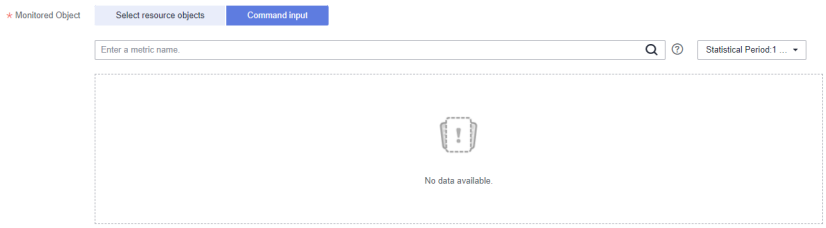
1. Set basic information such as the rule name and description.

   Figure 8-4 Setting basic alarm information

2. Set details about the rule.
   a. Set **Rule Type** to **Threshold alarm**.
   b. Set **Monitored Object** to **Command input** and enter the corresponding command.

   Figure 8-5 Setting objects to be monitored

NOTE

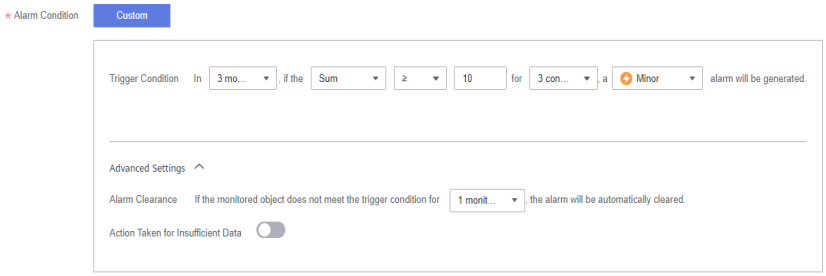
Enter Prometheus commands. For details about Prometheus commands, move the cursor to ⓘ next to the search box and click **Learn more**.

For example, to query the number of DMS Kafka transfer failures in instance A, run the following command:
sum(label_replace(sum_over_time(iotda_dmsKafka_forwarding_failedCount{instance="*ID of instance A***"}[59999ms]),"__name__","iotda_dmsKafka_forwarding_failedCount","","")) by(__name__,instance)**

iotda_dmsKafka_forwarding_failedCount indicates the metric name, which can be obtained from **Table 8-8**.

c. Set **Alarm Condition** to **Custom**. In the **Trigger Condition** area, set trigger condition parameters, such as the statistical period, consecutive period, and threshold condition. For details about the parameters, see **Table 8-9**.

Figure 8-6 Setting alarm conditions



Taking the preceding figure as an example, a minor alarm will be generated when the total number is greater than 10 in three statistical periods.

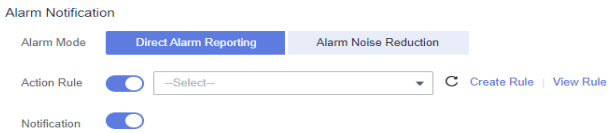
Table 8-9 Alarm condition parameters

Category	Parameter	Description
Trigger Condition	Statistical Period	Interval at which metric data is collected. By default, only one period is measured. A maximum of five periods can be measured.
	Consecutive Periods	When the metric value meets the threshold condition for a specified number of consecutive periods, a threshold alarm will be generated.
	Statistic	Method used to measure metrics. Options: **Avg.**, **Min.**, **Max.**, **Sum**, and **Samples**.

Category	Parameter	Description
	Threshold Condition	Trigger condition of a threshold alarm. A threshold condition consists of two parts: operators (≥, ≤, >, and <) and threshold value. For example, if **Threshold Condition** is set to **> 85** and an actual metric value exceeds 85, a threshold alarm will be generated.
	Alarm Severity	Severity of a threshold alarm. Options: **Critical**, **Major**, **Minor**, and **Warning**.
Advanced Configuration	Alarm Clearance	An alarm will be cleared if the monitored object does not meet the trigger condition within the monitoring period. By default, metrics in only one period are monitored. You can set up to five monitoring periods.
	Action Taken for Insufficient Data	Action to be taken when no metric data is generated or metric data is insufficient within the monitoring period. You can configure this option based on your requirements. By default, metrics in only one period are monitored. You can set up to five monitoring periods. Options: **Alarm**, **Insufficient data**, **Keep previous status**, and **Normal**.

3. Configure alarm notifications.
   a. Set **Alarm Mode** to **Direct Alarm Reporting**.
   b. Select the action rule created in **Step 1**.
   c. Enable **Notification**.

Figure 8-7 Setting alarm notifications



NOTE

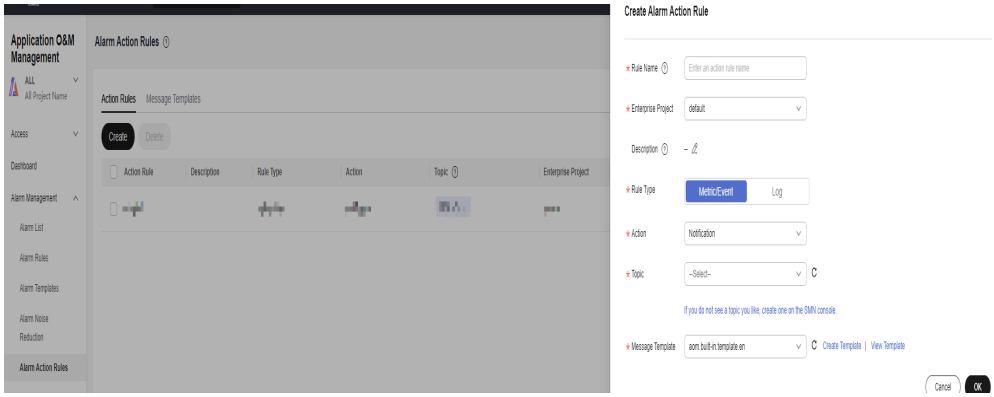
For details about how to use alarm noise reduction, see **Alarm Noise Reduction**.

----End

Configuration Procedure for AOM 2.0

Step 1 Log in to the **AOM** console. In the navigation pane, choose **Alarm Management** > **Alarm Action Rules**. On the displayed page, click **Create** and configure parameters.

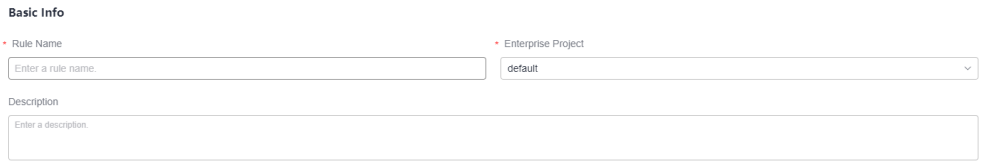
Figure 8-8 Creating an alarm action rule



Step 2 In the navigation pane, choose **Alarm Management** > **Alarm Rules**. On the displayed page, click **Create**.

Step 3 Enter a rule name, select an enterprise project from the drop-down list, and enter the rule description as required.

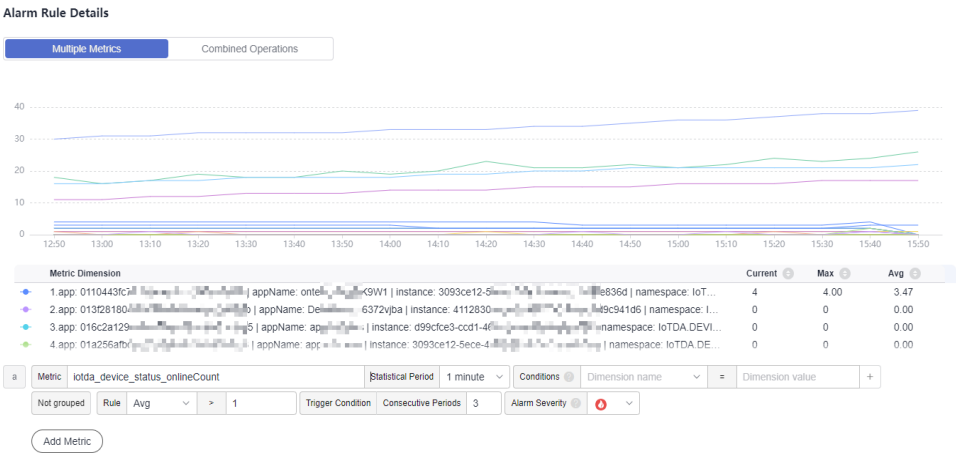
Figure 8-9 Creating an alarm rule



Step 4 Set details about the rule.

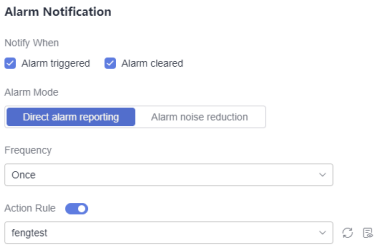
1. **Rule Type**: Select **Metric alarm rule**.
2. **Configuration Mode**: Select **Select from all metrics**.
3. **Prometheus Instance**: Select the target instance.
4. **Alarm Rule Details**: Select **Multiple Metrics**.
5. **Metric**: Enter **iotda** in the **Metric** text box to get related metrics. For details about the metric, see **Table 8-8**.
6. **Conditions**: Specify the dimension name, filter criteria, and dimension value.
7. **Rule**: Enter the metric alarm threshold.
8. **Trigger Condition**: Enter the consecutive periods for triggering the alarm.
9. **Alarm Severity**: Select an alarm severity icon.

Figure 8-10 Setting alarm rules



Step 5 Set alarm notification. Enable the alarm action rule and select a rule from the drop-down list. If no action rule is available, click the check icon on the right to go to the page for creating an alarm action rule.

Figure 8-11 Setting alarm rules



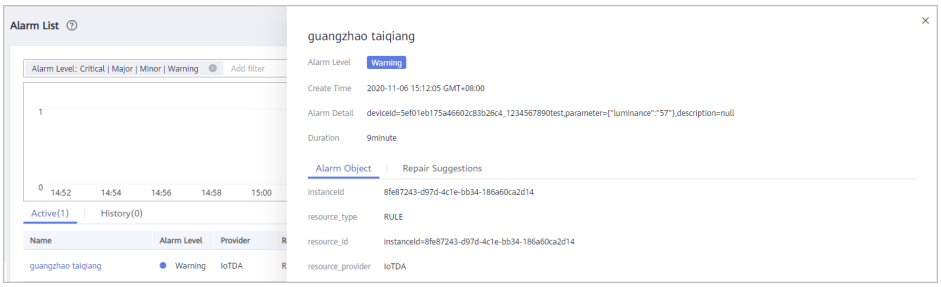
----End

Checking Alarm Information

You can use AOM to view alarms generated in the last 15 days. For details, see **Viewing Alarms**.

1. Access the **IoTDA** service page and click **Access Console**. Click the target instance card.
2. In the navigation pane, choose **O&M** > **Device Alarms**. Click **Application Operations Management (AOM)** to access the AOM console and view alarms generated for IoTDA.
3. Click an alarm to check the alarm details.

Figure 8-12 Viewing alarm details



4. Clear an alarm. After the fault is rectified, click in the **Operation** column of the target alarm.

8.4 Audit Logs

Scenarios

After you enable CTS and the management tracker is created, CTS starts recording operations on cloud resources. After a data tracker is created, the system starts to record user operations on data in OBS buckets. CTS retains operation records of the latest seven days.

This section describes how to query and export operation records of the last seven days on the CTS console.

- **Viewing Real-Time Traces in the Trace List of the New Edition**
- **Viewing Real-Time Traces in the Trace List of the Old Edition**

Constraints

- Traces of a single account can be viewed on the CTS console. Multi-account traces can be viewed only on the **Trace List** page of each account, or in the OBS bucket or the **CTS/system** log stream configured for the management tracker with the organization function enabled.
- You can only query operation records of the last seven days on the CTS console. To store operation records for more than seven days, you must configure an OBS bucket to transfer records to it. Otherwise, you cannot query the operation records generated seven days ago.
- After performing operations on the cloud, you can query management traces on the CTS console 1 minute later and query data traces on the CTS console 5 minutes later.

Viewing Real-Time Traces in the Trace List of the New Edition

Step 1 Log in to the console.

Step 2 Click in the upper left corner and choose **Management & Governance** > **Cloud Trace Service**.

Step 3 Choose **Trace List** in the navigation pane.

Step 4 On the **Trace List** page, use advanced search to query traces. You can combine one or more filters.

- **Trace Name**: Enter a trace name.
- **Trace ID**: Enter a trace ID.
- **Resource Name**: Enter a resource name. If the cloud resource involved in the trace does not have a resource name or the corresponding API operation does not involve the resource name parameter, leave this field empty.
- **Resource ID**: Enter a resource ID. Leave this field empty if the resource has no resource ID or if resource creation failed.
- **Trace Source**: Select a cloud service name from the drop-down list.
- **Resource Type**: Select a resource type from the drop-down list.
- **Operator**: Select one or more operators from the drop-down list.
- **Trace Status**: Select **normal**, **warning**, or **incident**.
  - **normal**: The operation succeeded.
  - **warning**: The operation failed.
  - **incident**: The operation caused a fault that is more serious than the operation failure, for example, causing other faults.
- Time range: Select **Last 1 hour**, **Last 1 day**, or **Last 1 week**, or specify a custom time range.

Step 5 On the **Trace List** page, you can also export and refresh the trace list, and customize the list display settings.

1. Enter any keyword in the search box and click to filter desired traces.
2. Click **Export** to export all traces in the query result as an .xlsx file. The file can contain up to 5,000 records.
3. Click to view the latest information about traces.
4. Click to customize the information to be displayed in the trace list. If **Auto wrapping** is enabled ( ), excess text will move down to the next line; otherwise, the text will be truncated. By default, this function is disabled.

Step 6 For details about key fields in the trace structure, see **Trace Structure** and **Example Traces**.

Step 7 (Optional) On the **Trace List** page of the new edition, click **Go to Old Edition** in the upper right corner to switch to the **Trace List** page of the old edition.

----End

Viewing Real-Time Traces in the Trace List of the Old Edition

Step 1 Log in to the console.

Step 2 Click in the upper left corner and choose **Management & Governance** > **Cloud Trace Service**.

Step 3 Choose **Trace List** in the navigation pane.

Step 4 Each time you log in to the CTS console, the new edition is displayed by default. Click **Go to Old Edition** in the upper right corner to switch to the trace list of the old edition.

Step 5 Specify the filters used for querying traces. The following filters are available:

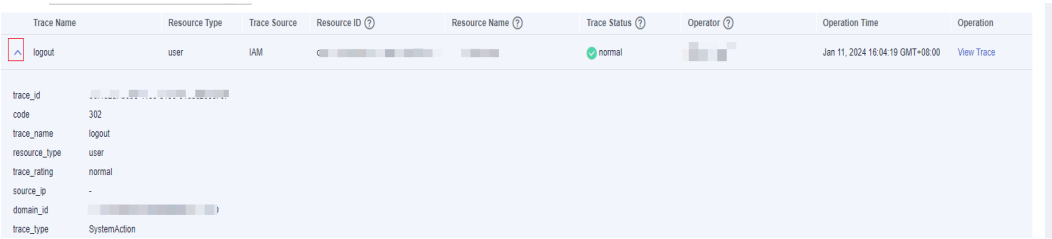
- **Trace Type**, **Trace Source**, **Resource Type**, and **Search By**: Select a filter from the drop-down list.
  - If you select **Resource ID** for **Search By**, specify a resource ID.
  - If you select **Trace name** for **Search By**, specify a trace name.
  - If you select **Resource name** for **Search By**, specify a resource name.
- **Operator**: Select a specific operator (a user other than an account).
- **Trace Status**: Select **All trace statuses**, **Normal**, **Warning**, or **Incident**.
- **Time Range**: You can query traces generated during any time range of the last seven days.
- Click **Export** to export all traces in the query result as a CSV file. The file can contain up to 5,000 records.

Step 6 Click **Query**.

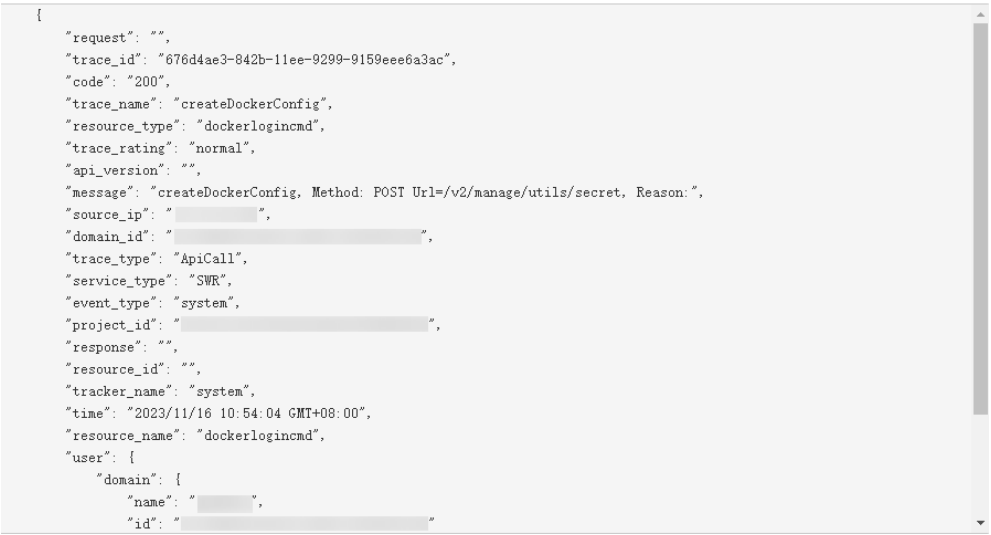
Step 7 On the **Trace List** page, you can also export and refresh the trace list.

- Click **Export** to export all traces in the query result as a CSV file. The file can contain up to 5,000 records.
- Click to view the latest information about traces.

Step 8 Click on the left of a trace to expand its details.



Step 9 Click **View Trace** in the **Operation** column. The trace details are displayed.



Step 10 For details about key fields in the trace structure, see **Trace Structure** and **Example Traces**.

Step 11 (Optional) On the **Trace List** page of the old edition, click **New Edition** in the upper right corner to switch to the **Trace List** page of the new edition.

----End

IoTDA Operations That Can Be Recorded by CTS

Using Cloud Trace Service (CTS), you can view user and platform operations and results. If an exception occurs, you can locate and rectify the fault based on the logs. The table below lists IoTDA operations that are logged.

Table 8-10 IoTDA operations that can be recorded by CTS

Category	Operation	Resource Type	Trace Name
Linkage rule management	Creating a rule	rules	createRules
	Deleting a rule	rules	deleteRules
	Updating a rule	rules	updateRules
	Modifying the rule status	rules	changeRuleStatus
JavaScript script management	Uploading JavaScript plug-in scripts	scripts	createScript
	Deleting JavaScript plug-in scripts	scripts	deleteScript
	Debugging JavaScript plug-in scripts	scripts	runScript
Function plug-in management	Uploading a Function plug-in	functions	createProductFunctions
	Deleting a Function plug-in	functions	deleteProductFunctions
	Downloading a Function plug-in	functions	getProductFunctions
Batch task management	Creating a batch task	batchtasks	createBatchtasks
	Retrying a batch task	batchtasks	retryBatchtasks
	Stopping a batch task	batchtasks	stopBatchtasks

Category	Operation	Resource Type	Trace Name
	Deleting a batch task	batchtasks	deleteBatchtasks
Batch task file management	Uploading a batch task file	batchtask-files	uploadBatchTask-File
	Deleting a batch task file	batchtask-files	deleteBatchTaskFile
Export tasks	Creating an export task	export-tasks	createExportTasks
	Deleting an export task	export-tasks	deleteExportTask
	Downloading an export file	export-tasks	createTaskreport
Application certificate management	Uploading a push CA certificate	Certificate	createCertificate
	Updating a push CA certificate	Certificate	updateCertificate
	Deleting a push CA certificate	Certificate	deleteCertificate
Certificate management	Uploading a device CA certificate	certificate	addCertificate
	Deleting a device CA certificate	certificate	deleteCertificate
	Commissioning the device CA certificate	certificate	debugCertificate
	Verifying a device CA certificate	certificate	verifyCertificate
	Downloading a device CA certificate	certificate	downloadCertifi-cate
Server certificate management	Creating a certificate for the enterprise edition	ServerCertificate	addServerCertifi-cate
	Replacing the certificate of the enterprise edition	ServerCertificate	updateServerCerti-ficate

Category	Operation	Resource Type	Trace Name
	Deleting a certificate of the enterprise edition	ServerCertificate	deleteServerCertificate
Resource space management	Creating a resource space	application	addApplication
	Deleting a resource space	application	deleteApplication
	Modifying a resource space	application	updateApplication
Access code management	Creating an access code	accessCode	createAccessCode
	Verifying an access code	accessCode	verifyAccessCode
Software/ Firmware upgrade package management	Creating an OTA upgrade package	upgradeTask	uploadOtaPackages
	Deleting an OTA upgrade package	upgradeTask	deleteOtaPackages
File storage and management	Configuring an OBS bucket for file upload.	upgradeTask	createBucket
Forwarding rule management	Creating a rule triggering condition	routing-rule	addRule
	Modifying a rule triggering condition	routing-rule	modifyRule
	Deleting a rule triggering condition	routing-rule	deleteRule
	Testing the SQL connectivity	rule-sql	checkSql
Forwarding rule action management	Creating a rule action	rule-action	addAction
	Modifying a rule action	rule-action	modifyAction
	Deleting a rule action	rule-action	deleteAction

Category	Operation	Resource Type	Trace Name
	Testing the connectivity interface	rule-action	sendMessage
Outbound flow control policy management	Creating an outbound flow control policy	create-flow-control-policy	createRoutingFlow-ControlPolicy
	Updating an outbound flow control policy	update-flow-control-policy	updateRoutingFlow ControlPolicy
	Deleting an outbound flow control policy	delete-flow-control-policy	deleteRoutingFlow-ControlPolicy
Outbound push stacking policy management	Creating an outbound push stacking policy	create-routing-backlog-policy	createRoutingBack-logPolicy
	Modifying an outbound push stacking policy	update-routing-backlog-policy	updateRoutingBack logPolicy
	Deleting an outbound push stacking policy	delete-routing-backlog-policy	deleteRoutingBack-logPolicy
Device shadow	Configuring desired data in the device shadow	deviceShadow	updateDeviceShad-ow
Plug-in mapping management	Modifying the mapping	plugin	addMapping
Plug-in message management	Modifying message information	plugin	addMessage
Plug-in management	Deploying an online plug-in	plugin	deployPlugin
	Saving plug-in information	plugin	savePluginMessage
	Updating plug-in information	plugin	modifyPluginMes-sage
	Deploying an offline plug-in	plugin	bundlePackages
Simulator management	Registering and debugging a device simulator	plugin	registerEmulated-Device

Category	Operation	Resource Type	Trace Name
Device debugging messages	Sending upstream code stream	plugin	simulateReport
Tunnel management	Creating a tunnel	tunnels	createTunnel
	Deleting a tunnel	tunnels	deleteTunnel
	Modifying a tunnel	tunnels	updateTunnel
Product management	Creating a product	product	addProduct
	Modifying product information	product	updateProduct
	Deleting a product	product	deleteProduct
Custom topic management	Modifying a custom topic	topic	updateTopic
	Deleting a custom topic	topic	deleteTopic
	Creating a custom topic	topic	addTopic
Exception detection configuration	Configuring the exception detection	productConfig	addProductConfig
AMQP queue management	Creating an AMQP queue	amqp	addQueue
	Deleting an AMQP queue	amqp	deleteQueue
	Terminating the receive-link consumption capability	amqp	hangUpConnection
Cloud interconnection configuration management	Creating cloud interconnection configurations	service-integration	addServiceIntegra-tionConfig
	Deleting cloud interconnection configurations	service-integration	deleteServiceInte-grationConfig
	Modifying cloud interconnection configurations	service-integration	modifyServiceInte-grationConfig

Category	Operation	Resource Type	Trace Name
Group management	Adding a group	device-group	addDeviceGroup
	Modifying a group	device-group	updateDeviceGroup
	Deleting a group	device-group	deleteDeviceGroup
	Managing devices in a group	device-group	manageDevicesInGroup
Device tag management	Binding a tag	tag	bindTagsToResource
	Unbinding a tag	tag	unbindTagsToResource
Device management	Creating a device	device	addDevice
	Modifying device information	device	updateDevice
	Deleting a device	device	deleteDevice
	Resetting a device secret	device	resetDeviceSecret
	Freezing a device	device	freeze-device
	Unfreezing a device	device	unfreeze-device
HarmonyOS soft bus	Creating a HarmonyOS soft bus	harmony-soft-bus	create-harmony-soft-bus
	Deleting a HarmonyOS soft bus	harmony-soft-bus	delete-harmony-soft-bus
	Resetting a HarmonyOS soft bus key	harmony-soft-bus	reset-harmony-soft-bus-key
	Synchronizing a HarmonyOS soft bus	harmony-soft-bus	sync-harmony-soft-bus
Device proxy management	Deleting a device proxy	device-proxy	deleteDeviceProxy
	Creating a device proxy	device-proxy	addDeviceProxy
	Modifying a device proxy	device-proxy	updateDeviceProxy

Category	Operation	Resource Type	Trace Name
Device policy management	Creating a device policy	device-policy	addDevicePolicy
	Deleting a device policy	device-policy	deleteDevicePolicy
	Updating a device policy	device-policy	updateDevicePolicy
	Binding a device policy	device-policy	bindDevicePolicy
	Unbinding a device policy	device-policy	unbindDevicePolicy
Message tracing management	Modifying message tracing configurations	message-trace	updateMessageTraceConfig
	Deleting message tracing configurations	message-trace	deleteMessageTraceConfig
	Deleting message tracing data	message-trace	deleteMessageTraceData
O&M configuration management	Modifying O&M configurations	device-config	updateDeviceConfig
Command management	Delivering a command	command	sendCommand
	Delivering an asynchronous command	asyncCommand	sendAsyncCommand
Remote login	Creating an SSH channel	SshConnect	SshConnect
	Delivering an SSH command	SshComand	SshComand
	Disabling an SSH channel	SshDisconnect	SshDisconnect

8.5 Run Logs (Old Version)

IoTDA can record the connections with devices and applications and report logs to Log Tank Service (LTS). LTS provides real-time query, massive log storage, log structuring, visualization, and analysis capabilities. It provides a free quota of 500

MB per month. When this quota is used up, you will be billed for any excess usage on a pay-per-use basis. For details, see **What Is Log Tank Service?**

Currently, only service run logs of MQTT devices can be recorded. For details, see the following table.

Table 8-11 Service types

Service Type	Service Flow
Device status	**Device going online**
	Device going offline
Device message	**Application requesting message delivery to devices**
	Platform delivering messages to devices
	Device reporting messages
Device command	**Application requesting command delivery to devices**
	Platform delivering commands to devices
	Platform receiving device response to commands
Device property	**Application requesting device property modification**
	Platform delivering property modification to devices
	Device reporting properties
	Gateway reporting device properties in batches
Device event	**Platform notifying a gateway of new child device connection**
	Platform notifying a gateway of child device deletion
	Gateway synchronizing child device information
	Gateway updating child device statuses
	Gateway requesting for adding child devices
	Platform responding to a request for adding child devices
	Gateway requesting for deleting child devices
	Platform responding to a request for deleting child devices
	Gateway updating child device statuses
	Platform responding to a request for updating child device statuses
	Platform delivering a command to obtain version information

Service Type	Service Flow
	Device reporting the software and firmware versions
	Platform delivering an upgrade command
	Device reporting the upgrade status
	Device requesting a URL for file upload
	Platform delivering a temporary URL for file upload
	Device reporting file upload results
	Device requesting a URL for file download
	Platform delivering a temporary URL for file download
	Device reporting file download results
	Device requesting time synchronization
	Platform responding to a request for time synchronization
	Device reporting information
	Platform delivering a log collection notification
	Device reporting log content
	Platform delivering a configuration notification
	Device reporting the configuration response
	Device downloading upgrade package
Batch task	**Subtask execution result of a batch task**
Device self-registration	**Device self-registration result**.
Device authentication	Custom device authentication result

Table 8-12 Format of batch task run logs

Field	Description
appId	Application ID.
deviceId	Device ID.
categoryName	Log type: batch.task.
operation	Action. Set this parameter to the ID of a batch task. For details, see **Create a Batch Task**.

Field	Description

request	Request content in JSON format.
	{
	"task_type": "createDevices", / / Task type
	"package_id": "f2303267a6e8f0053037c2a9", // Software/Firmware upgrade package
	"package_ids": ["65f3ebe2682b9f4bcc38baad"] // Software/Firmware upgrade package
	}
	NOTE
	When the batch task type is **softwareUpgrade** or **firmwareUpgrade**, the **package_id** and **package_ids** parameters are supported.

response	Response content in JSON format.
	Success:
	{
	"output" : "xxxxxxxxxxxx"
	}
	Failure:
	{
	"error" : {
	"error_code" : "IOTDA.XXXXX",
	"error_msg" : "XXXXX."
	}
	}

status	Execution result. Subtask status. Options: **Success**, **Fail**, **Stopped**, or **Removed**.

Table 8-13 Format of non-batch tasks

Field	Description
recordTime	Log collection time. The time format is yyyy-MM-dd'T'HH:mm:ss,SSS'Z'. Example: 2020-06-16T09:24:45,708Z
deviceId	Device ID.
requestId	Request ID.

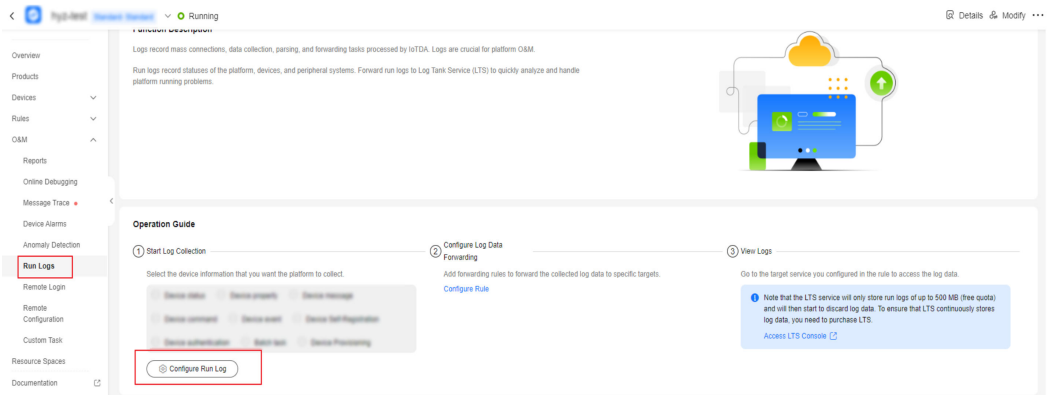
Field	Description
categoryName	Options: device.status device.message device.command device.property device.event device.auth device.provisioning
operation	Operation name. Example: API URL or MQTT message topic.
request	Request parameter of an operation. Example: API request body.
response	Operation result. Example: API response body or error information.
result	Operation status code.

Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

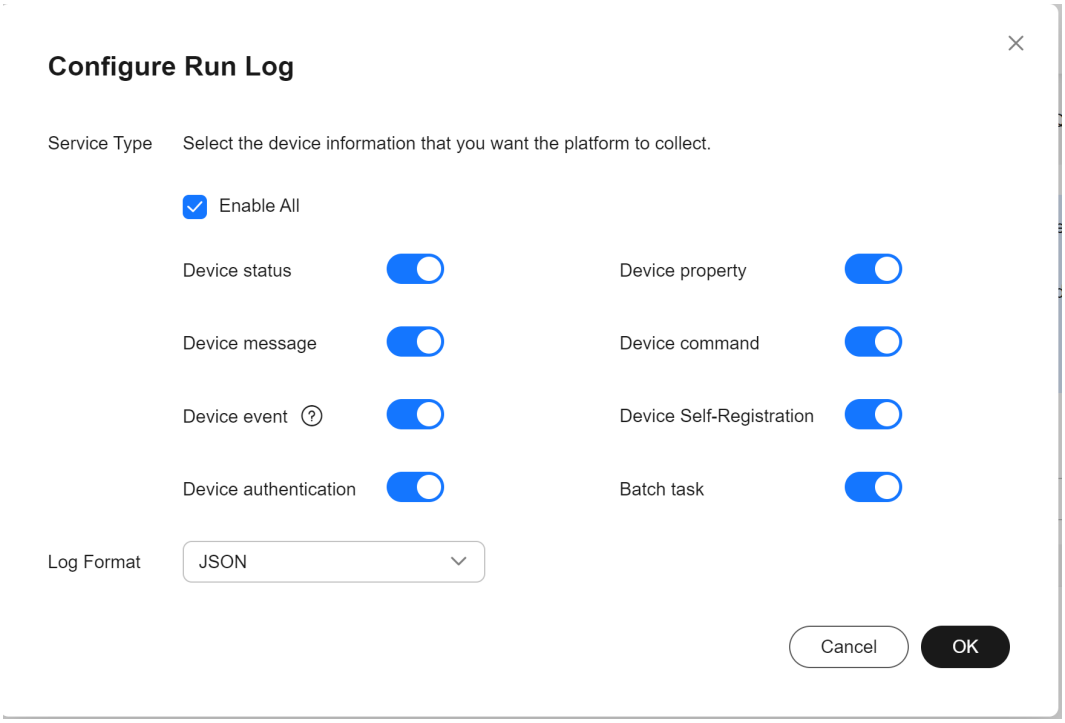
Step 2 In the navigation pane, choose **O&M** > **Run Logs**, and click **Configure Run Log**.

Figure 8-13 Run log - Old version configuration



Step 3 On the displayed dialog box, select the service type for data collection and click **OK**.

Figure 8-14 Run log - Log switch of old version configuration



Step 4 Create a run log forwarding rule to forward the collected log data to other cloud services, so that you can view and process it. You are advised to forward log data to LTS. The following uses LTS as an example.

1. In the navigation pane, choose **Rules** > **Data Forwarding**, and click **Create Rule**.
2. Configure parameters by referring to the following table, and click **Create Rule**.

Table 8-14 Creating a rule

Parameter	Description
Rule Name	Specify the name of a rule to create.
Description	Describe the rule.
Data Source	Select **Run log**.
Trigger	After the data source is selected, the platform automatically matches the trigger event.
Resource Space	You can select a single resource space or all resource spaces.

3. Click the **Set Forwarding Target** tab, and then click **Add** to set a forwarding target.

Table 8-15 Setting the forwarding target

Parameter	Description
Forwarding Target	Select **Log Tank Service (LTS)**.
Region	Currently, log data can be forwarded only to LTS in the same region.
Log Group/log Stream	Select a log group and log stream of LTS. If no log group or log stream is available, create them by referring to **Managing Log Groups** and **Managing Log Streams**.

NOTE

Logs are retained in the log group for 7 days by default. Logs older than the retention period will be automatically deleted. For long-term storage, you can transfer logs to Object Storage Service (OBS) buckets. For details, see **Transferring Logs to OBS**.

LTS provides a free quota of 500 MB per month. By default, it continues to collect logs when the quota is used up. You will be billed for the excess usage on a pay-per-use basis. You can log in to the LTS console and choose **Configuration Center** to disable this function.

4. Click **OK**, and then click **Enable Rule** to forward run logs to LTS.

Step 5 Log in to the **LTS console** and choose **Log Management**.

Step 6 Select the log group and log stream created in **3** to view the logs reported by IoTDA. Search for raw and target logs by referring to **Log Search**. For example, search for logs by device ID and service type.

Figure 8-15 Run logs - Checking the log list

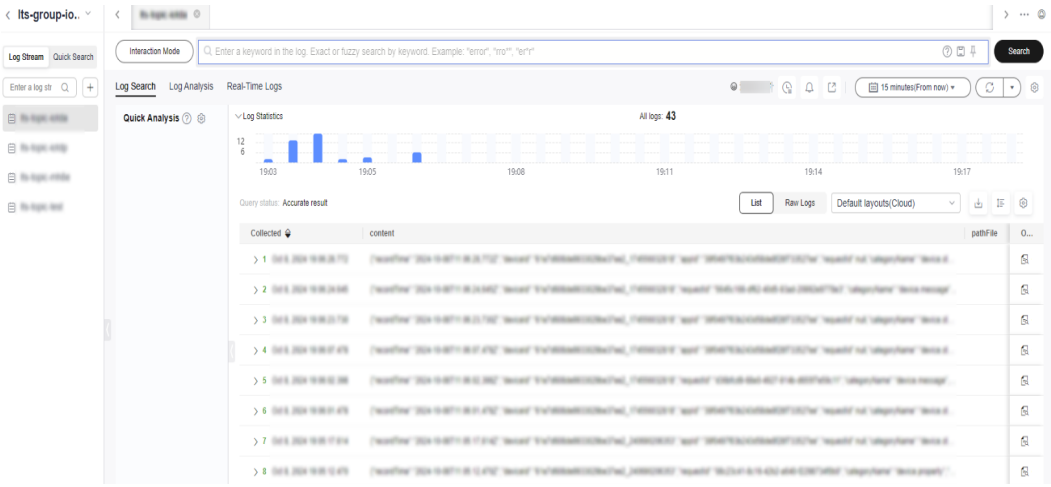
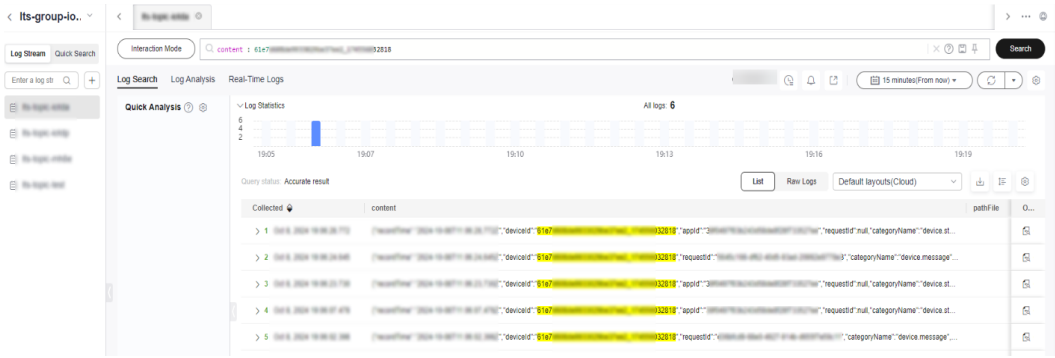
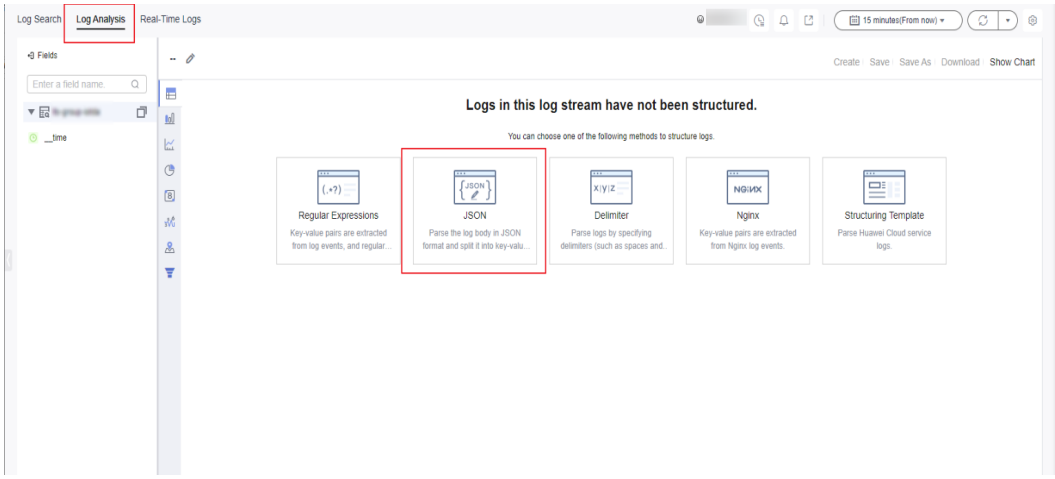


Figure 8-16 Run logs - Searching for run logs



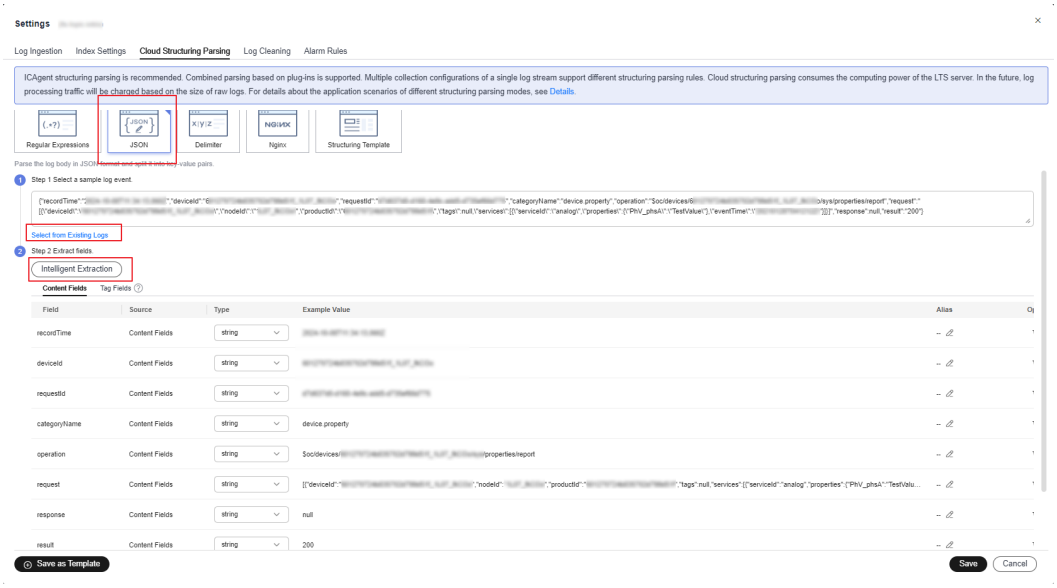
Step 7 On the log stream details page, click the **Log Analysis** tab. On the log structuring page that is displayed, select **JSON**.

Figure 8-17 Run logs - Log analysis



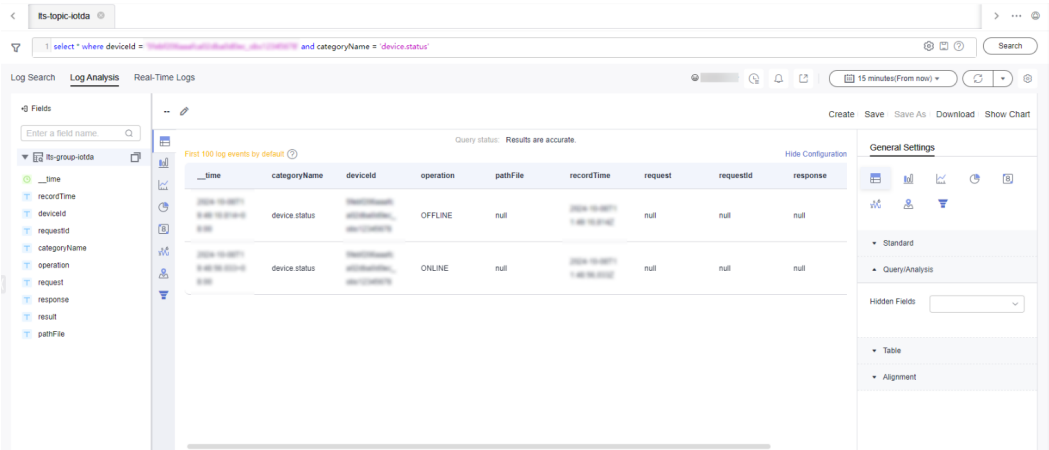
Step 8 Select a sample log that contains time, device ID, request ID, service type, operation, request parameter, result information, and execution status. Click the button of intelligent extraction to modify the field names, for example, recordTime, requestId, deviceId, categoryName, operation, request, response and result. Change the field type to string. Click **Save** to complete the log structuring configuration. For details, see **Structuring Logs**.

Figure 8-18 Run logs - Configuring log structuring



Step 9 After the raw logs are structured, the service will respond. On the **Log Management** page, select the log group and log stream created in **3**, and click the button for log analysis to check the structured target logs. You can query logs using SQL statements. For example, to query logs by device ID and service type, run **select * where deviceId = '*{{deviceId}}*' and categoryName = 'device.status'**.

Figure 8-19 Run logs - Log analysis and search



----End

8.6 Run Logs (New Version)

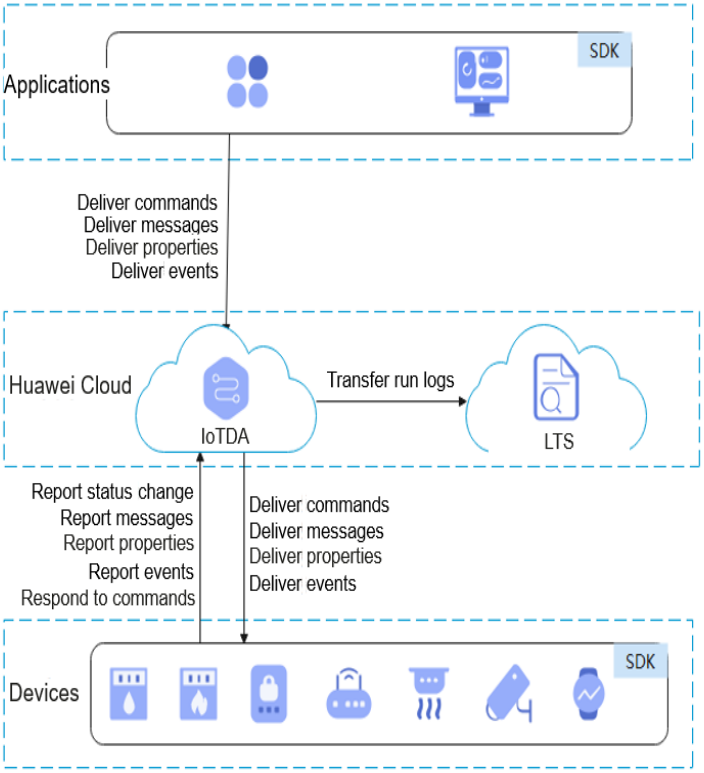
Overview

IoTDA records connections with devices and applications and reports them as logs to Log Tank Service (LTS). LTS provides real-time query, mass storage, structured processing, and visualized chart analysis.

You do not need to manually create log groups, log streams, and forwarding rule action when using the new version of run logs. The IoT platform automatically

completes configurations for you. You can long in to the IoT platform to view the run logs.

Figure 8-20 Process of run logs



Constraints

Currently, only service run logs of MQTT devices can be recorded. For details, see **Table 1**.

Table 8-16 Service type

Service Type	Service Process
Device status	**Device going online**
	Device going offline
Device message	**Application requesting message delivery to devices**
	Platform delivering messages to devices
	Device reporting messages
Device command	**Application requesting command delivery to devices**
	Platform delivering commands to devices

Service Type	Service Process
	Platform receiving device response to commands
Device property	Application requesting device property modification
	Platform delivering property modification to devices
	Device reporting properties
	Gateway reporting device properties in batches
Device event	Platform notifying a gateway of new child device connection
	Platform notifying a gateway of child device deletion
	Gateway synchronizing child device information
	Gateway updating child device status
	Gateway requesting for adding child devices
	Platform responding to a request for adding child devices
	Gateway requesting for deleting child devices
	Platform responding to a request for deleting child devices
	Gateway updating child device status
	Platform responding to a request for updating child device statuses
	Platform delivering a command to obtain version information
	Device reporting the software and firmware versions
	Platform delivering an upgrade command
	Device reporting the upgrade status
	Device requesting a URL for file upload
	Platform delivering a temporary URL for file upload
	Device reporting file upload results
	Device requesting a URL for file download
	Platform delivering a temporary URL for file download
	Device reporting file download results
	Device requesting time synchronization

Service Type	Service Process
	Platform responding to a request for time synchronization
	Device reporting information
	Platform delivering a log collection notification
	Device reporting log content
	Platform delivering a configuration notification
	Device reporting the configuration response
	Device downloading upgrade package
Batch task	**Subtask execution result of a batch task**
Device self-registration	**Device self-registration result**.
Device authentication	Custom device authentication result

Table 8-17 Format of batch task run logs

Field	Description
appId	Application ID.
deviceId	Device ID.
categoryName	Log type: batch.task.
operation	Action. Set this parameter to the ID of a batch task. For details, see **Create a Batch Task**.

request	Request content in JSON format.
	{
	"task_type": "createDevices", / / Task type
	"package_id": "f2303267a6e8f0053037c2a9", // Software/Firmware upgrade package
	"package_ids": ["65f3ebe2682b9f4bcc38baad"] // Software/Firmware upgrade package
	}
	NOTE
	When the batch task type is **softwareUpgrade** or **firmwareUpgrade**, the **package_id** and **package_ids** parameters are supported.

Field	Description

response	Response content in JSON format.
	Success:
	{
	"output" : "xxxxxxxxxxxx"
	}
	Failure:
	{
	"error" : {
	"error_code" : "IOTDA.XXXXX",
	"error_msg" : "XXXXX."
	}
	}

status	Execution result. Subtask status. Options: **Success**, **Fail**, **Stopped**, or **Removed**.

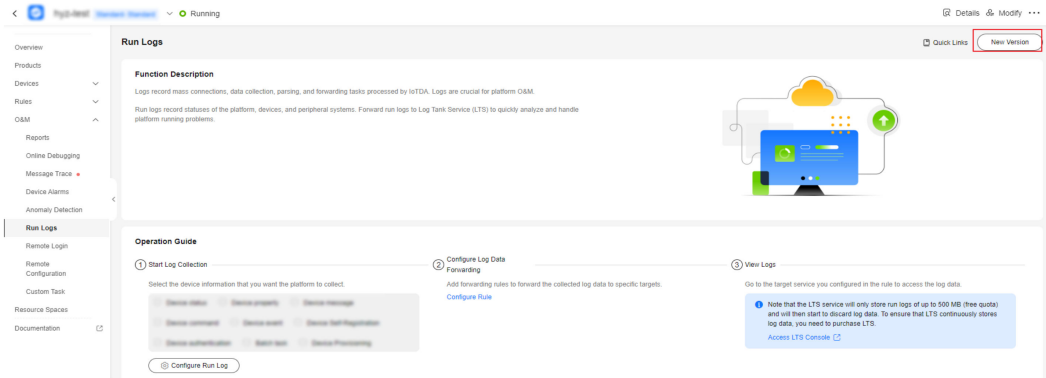
Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **O&M** > **Run Logs**.

Step 3 If you are using run logs of an earlier version, you can click **New Version** in the upper right corner. The new **Run Logs** page is displayed. If you have used the new version, the new version page is auto displayed.

Figure 8-21 Run logs - Going to new version

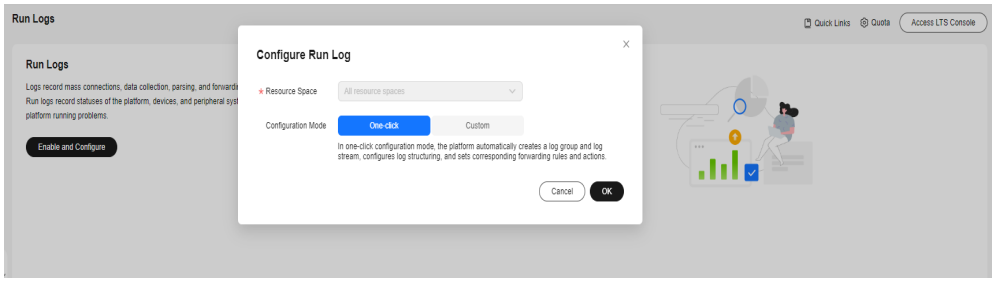


Step 4 If you use the function for the first time, you need to click **Enable and Configure**. Two configuration modes are available.

1. One-click configuration:

   The platform automatically creates a log group and log stream, configures log structuring, and sets corresponding forwarding rules and actions.

Figure 8-22 Run log - One-click configuration

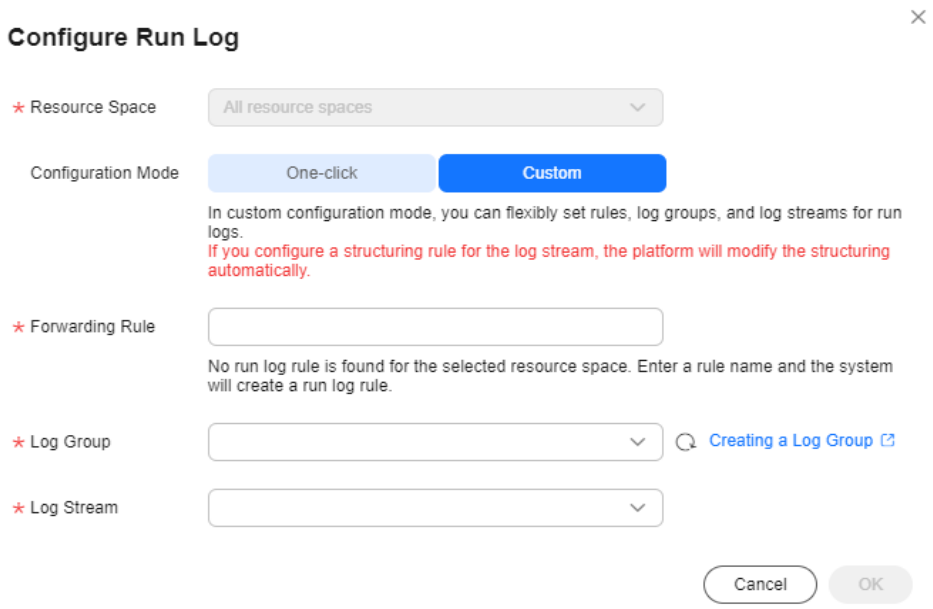


2. Custom configuration

   You can flexibly set rules, log groups, and log streams for run logs.

   If you configure a structuring rule for the log stream, the platform will modify the structuring automatically.

Figure 8-23 Run log - Custom configuration

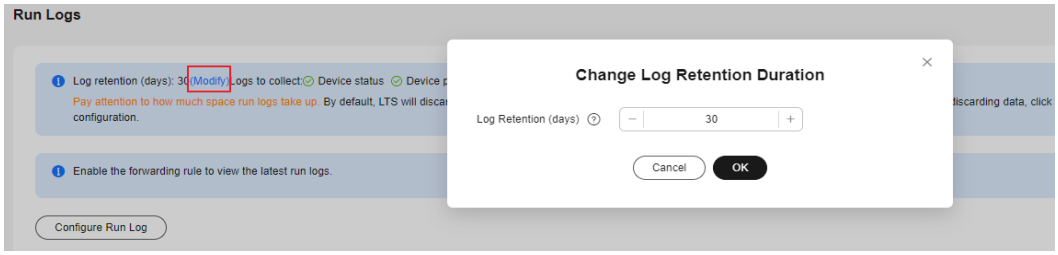


NOTE

Deleting a forwarding rule whose **Data Source** is **Run log** may affect functions. Exercise caution.

Step 5 After configuration, you can view or search for run logs (by time, log type, device ID, action, and request content) to analyze services. Run logs are stored in the LTS for 30 days by default. You can modify the retention duration to 365 days at most on the console.

Figure 8-24 Run log - Modifying the storage time

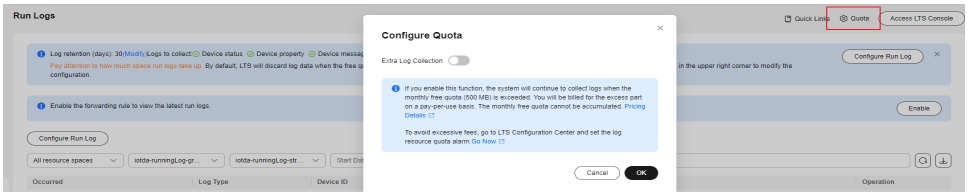


NOTE

Run logs are stored in LTS. LTS provides a free quota of 500 MB per month. You can click **Quota** in the upper right corner to configure the **Extra Log Collection** function. You can also go to **LTS Configuration Center** and set the log resource quota alarm. For details, see **What Is Log Tank Service?**

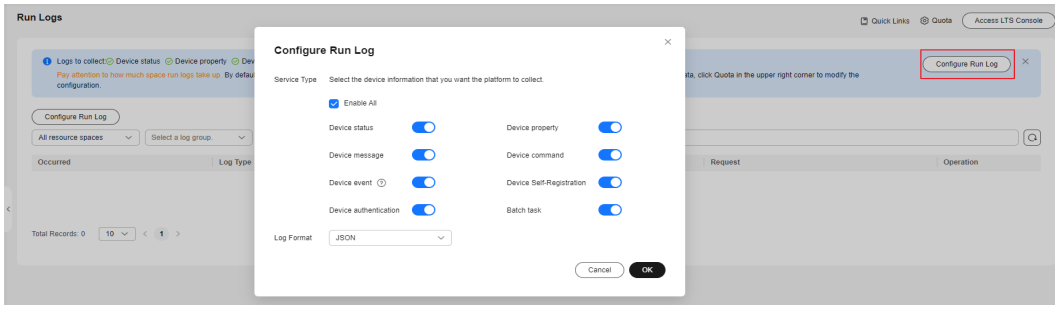
- If you enable **Extra Log Collection**, LTS will continue to collect log data when the monthly free quota (500 MB) is reached. You will be billed for the excess part on a pay-per-use basis. For details, see **Billing**.
- If you disable **Extra Log Collection**, LTS will discard log data when the monthly free quota (500 MB) is reached.

Figure 8-25 Run log - Quota setting



Step 6 One or more service types can be enabled for run logs. You can click **Configure Run Log** and change the log collection type in the displayed dialog box. If you do not select any type, the function of collecting logs is disabled.

Figure 8-26 Run log - Configuring the log switch



----End

Example of Run Logs

This section describes how to use the **Java SDK** to report messages, trigger run logs to be transferred to LTS, and check message reporting logs on IoTDA. JDK 1.8 or later is used.

Prerequisites:

1. The device has been registered on IoTDA.
2. The run logs of the new version has been enabled and configured, and the device message log switch has been enabled.

Configure the SDK on the device side:

Step 1 Configure the Maven dependency of the SDK on devices.

```
<dependency>
    <groupId>com.huaweicloud</groupId>
    <artifactId>iot-device-sdk-java</artifactId>
    <version>1.1.4</version>
</dependency>
```

Step 2 Configure the SDK and device connection parameters on devices. Note: Replace the domain name (domain), device ID (deviceId), and device secret (secret) in the actual code.

```
// Load the CA certificate of the IoT platform. For details about how to obtain the certificate, visit https://support.huaweicloud.com/intl/en-us/devg-iothub/iot_02_1004.html.
URL resource = BroadcastMessageSample.class.getClassLoader().getResource("ca.jks");
File file = new File(resource.getPath());

// The format is ssl://Domain name:Port number.
// To obtain the domain name, log in to the Huawei Cloud IoTDA console. In the navigation pane, choose Overview and click Access Details in the Instance Information area. Select the access domain name corresponding to port 8883.
String serverUrl = "ssl://{domain}:8883";
// Device ID created on the IoT platform
String deviceId = "{deviceId}";
// Secret corresponding to the device ID
String deviceSecret = "{secret}";
// Create a device.
IoTDevice device = new IoTDevice(serverUrl, deviceId, deviceSecret, file);
if (device.init() != 0) {
    return;
}
```

Step 3 Report a message.

```
device.getClient().reportDeviceMessage(new DeviceMessage("hello"), new ActionListener() {
    @Override
    public void onSuccess(Object context) {
        log.info("reportDeviceMessage ok");
    }

    @Override
    public void onFailure(Object context, Throwable var2) {
        log.error("reportDeviceMessagefail: "+ var2);
    }
});
```

----End

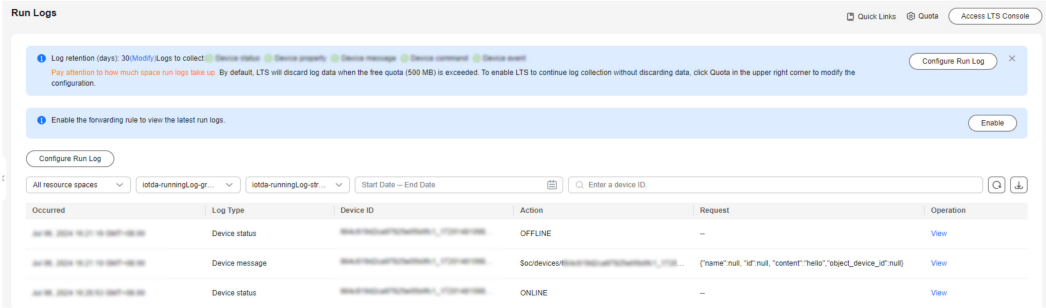
Verify the setting:

Step 1 Run the SDK code on the device. If the following information is displayed on the console, the device goes online and reports messages successfully.

```
2023-04-27 17:05:26  INFO MqttConnection:88 - Mqtt client connected. address :ssl://{domain}:8883
2023-04-27 17:05:26  INFO MqttConnection:214 - publish message topic =  $oc/devices/{deviceId}/sys/messages/up, msg = {"name":null,"id":null,"content":"hello","object_device_id":null}
2023-04-27 17:05:26  INFO MessageSample:43 - reportDeviceMessage ok
```

Step 2 Check run logs on the console. You can check the records of device login and logout and messages reported by devices.

Figure 8-27 Run logs - Log example



----End

8.7 Anomaly Detection

IoTDA provides device anomaly detection functions, including security checks and disconnection analysis.

Security Checks

IoTDA continuously detects device security threats. This section describes security check items and how to view and handle detected security risks.

Common detection items

Item	Description
Connection mode	No encryption protocol is used to establish secure connections between devices and IoTDA. This may cause man-in-the-middle and replay attacks and affect services.
TLS version	Insecure TLS protocol versions (TLS v1.0 and v1.1) have security vulnerabilities, which may cause security risks such as device data leakage.
Cryptographic algorithm suite	Currently, IoTDA checks the following insecure cryptographic algorithm suites: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA, TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA, TLS_PSK_WITH_AES_128_CBC_SHA, TLS_PSK_WITH_AES_256_CBC_SHA Insecure cryptographic algorithm suites have security vulnerabilities, which may cause security risks such as device data leakage.
Device connection	If a device attempts to establish connections with IoTDA multiple times within 1 second, the device may be cracked with brute force. As a result, identity information may be leaked, normal devices may be forced to go offline, and service data may be stolen.

Item	Description
Device authentication	Incorrect device identity authentication information causes device connection failures. This may affect services.

The preceding common check items are enabled by default. You can manually enable other non-common check items as required.

Table 8-18 Non-common detection items

Item	Description
Memory leak check	Checks device memory leaks.
Abnormal port	Checks whether abnormal ports are enabled on the device.
CPU usage	Checks whether the CPU usage of the device is too high.
Disk space	Checks whether the disk space of the device is insufficient.
Battery level	Checks whether the battery level of the device is too low.
Malicious IP address	Checks whether the device communicates with malicious IP addresses.
Local login	Checks whether attackers log in to the device through non-SSH networks.
Brute-force cracking login	Checks whether attackers attempt to log in to the device through brute force cracking.
Device file tampering	Checks whether files in a specified directory of a device are tampered with.

Disconnection Analysis

IoTDA help you analyze device disconnection causes by collecting statistics on the disconnection time range and characteristics of disconnected devices.

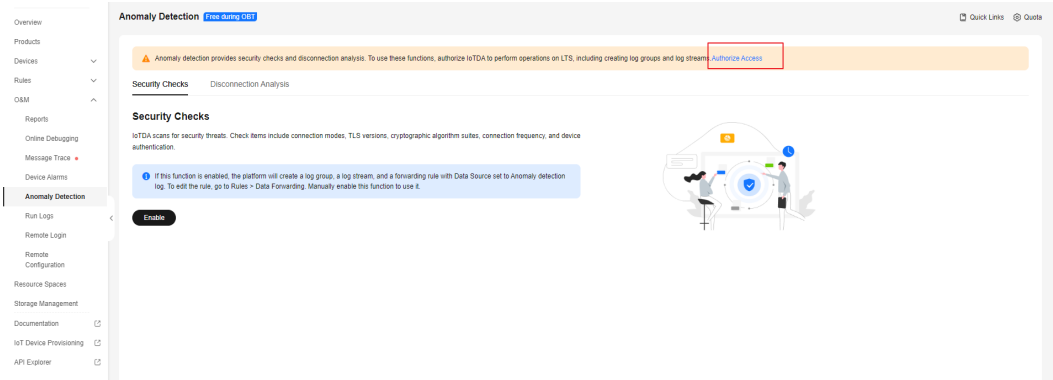
Disconnection Cause	Description
Disconnection requested by device	The device sends an MQTT disconnect packet to IoTDA for disconnection.
Device heartbeat timed out	The device does not comply with the MQTT protocol. It sends MQTT heartbeat packets to IoTDA within 1.5 times the configured heartbeat interval. As a result, IoTDA considers that the device connection is invalid and cuts off the connection according to protocol requirements. (Note: The heartbeat interval is specified when the device establishes a connection with IoTDA.)
Device-platform TCP connection cut off	IoTDA receives a TCP disconnection packet from the device. As a result, the TCP connection between the device and IoTDA is cut off.
Device deleted	The device is deleted from IoTDA, and IoTDA cuts off the connection with the device.
Device frozen	The device is frozen on IoTDA, and IoTDA cuts off the connection with the device.
Connection cut off by IoTDA	IoTDA cuts off the connection with the device during upgrade.
Earlier connection cut off	The device establishes connections with IoTDA repeatedly. IoTDA cuts off the existing connection and retains the new connection.
Device secret reset	When the device secret is reset and the connection is manually cut off, IoTDA cuts off the connection with the device.

Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **O&M** > **Anomaly Detection** and click **Authorize**. To use anomaly detection, authorize IoTDA service to perform operations on LTS.

Figure 8-28 Anomaly detection - Access authorization



Step 3 After authorization, **Security Checks** and **Disconnection Analysis** pages are available. Click **Enable** to enable the two functions. Otherwise, they cannot be used.

Figure 8-29 Anomaly detection - Security checks

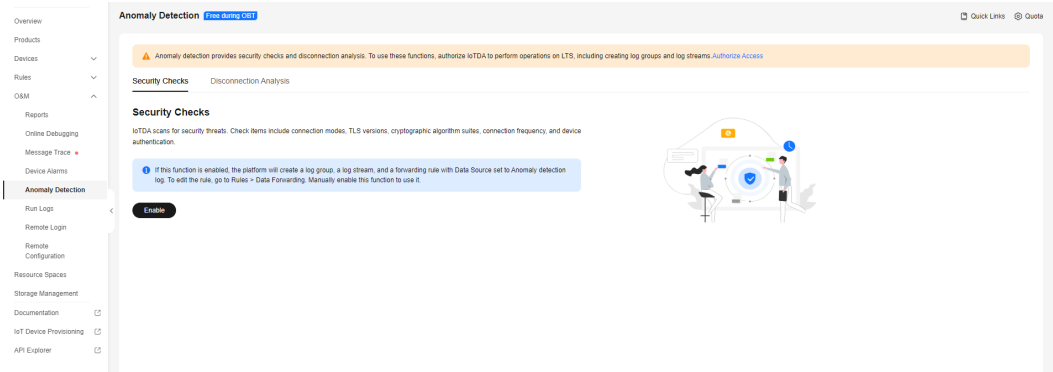
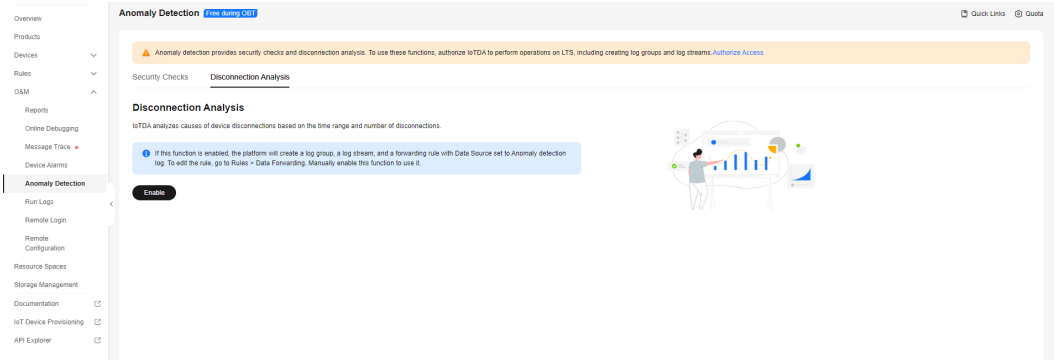


Figure 8-30 Anomaly detection - Offline analysis



NOTE

1. When this function is enabled, IoTDA automatically creates a log group, a log stream, and a data forwarding rule with the data source set to run logs of all resource spaces.

The log group name is *{domainName}*-**device-exception-group**, the log stream name is *{domainName}*-**device-exception-stream**, and the forwarding rule name is *{domainName}*-**device-exception-rule**.

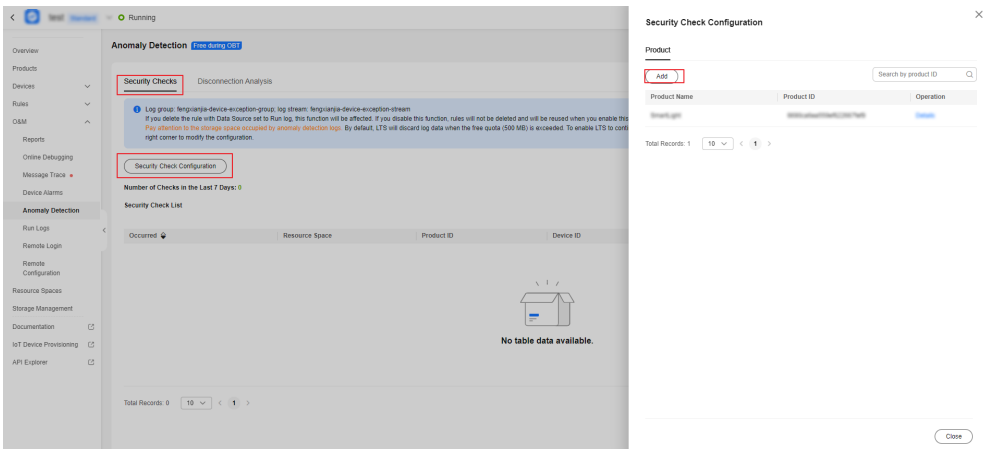
If you delete the rule with the data source set to run logs, this function will be affected. If you disable this function, rules will not be deleted and will be reused when you enable this function again.

2. Pay attention to the storage space occupied by anomaly detection logs. When the free quota (500 MB) is used up, LTS will discard data or continue collecting data after you purchase LTS. You can click **Quota** in the upper right corner to modify the quota.

Step 4 To enable **Security Checks**, perform the following steps. Otherwise, skip them.

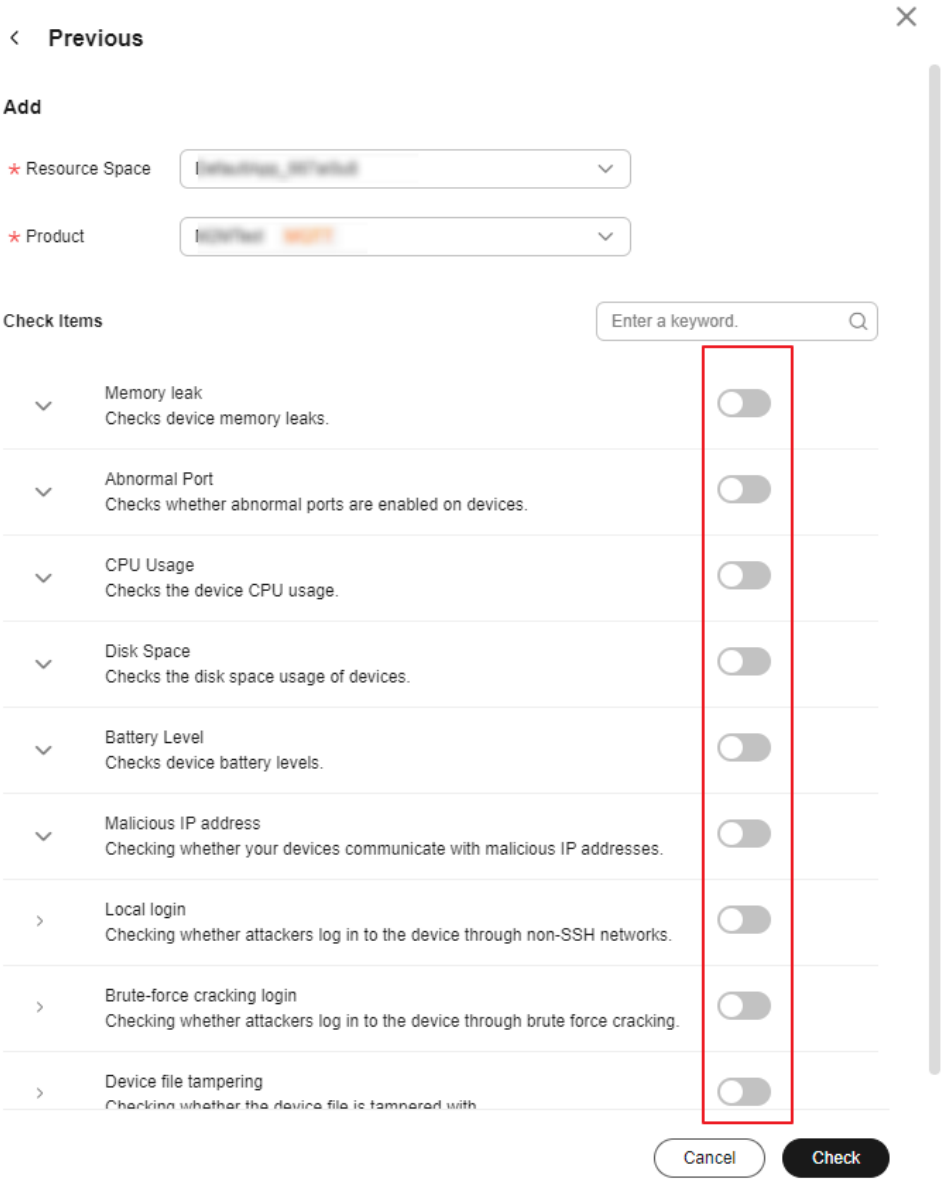
1. On the **Security Checks** tab page, click **Security Check Configuration**. In the displayed dialog box, click **Add**.

Figure 8-31 Anomaly detection - Security check configuration



2. On the configuration page, select the resource space and product name to be configured, and enable the corresponding check items as required.

Figure 8-32 Anomaly detection - Security check item configuration

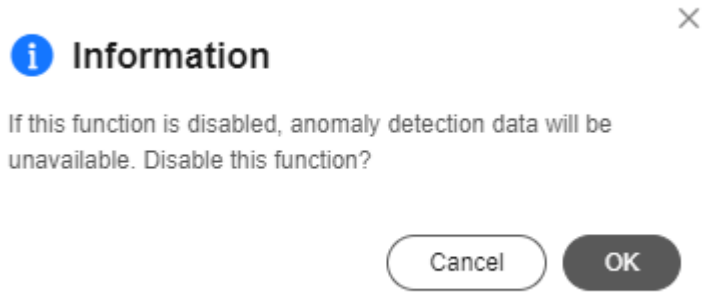


NOTE

Memory/CPU usage/Disk space/Battery level checks: The system compares the values reported by the device with the thresholds configured in the check items to determine whether to generate alarms. Abnormal port/Malicious IP address checks: Enter whitelisted ports or IP addresses for checks. The system compares the parameters reported by the device and the configured whitelist members. You can add IP address segments to the whitelist, for example, 192.168.1.10/24.

Step 5 You can click **Disable** on the corresponding pages to manually disable the security check and disconnection analysis functions, and click **Enable** to use them again.

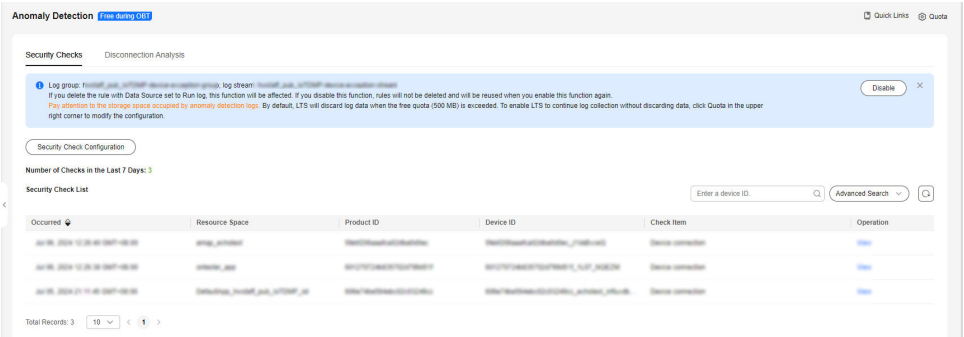
Figure 8-33 Anomaly detection - Disabling the function



NOTE

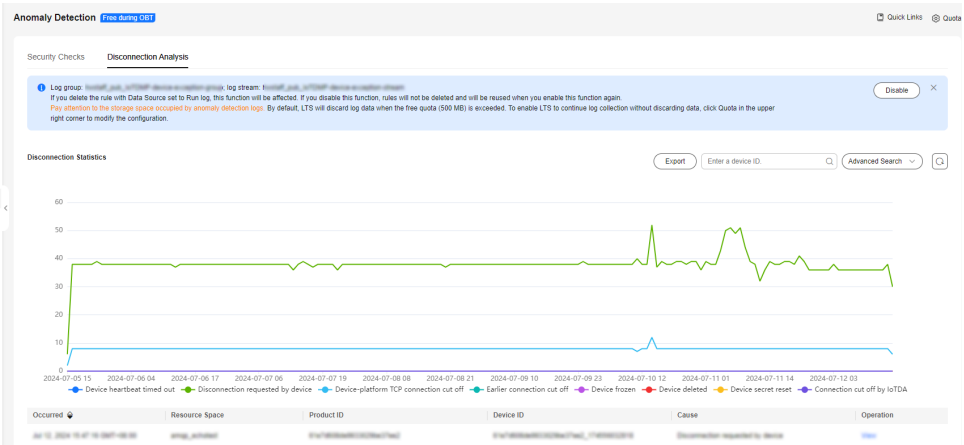
- After you enable security checks, IoTDA starts security checks on devices. Security check data of the last seven days can be stored at most. You can search for anomaly records by device ID, resource space, product, check item, and time range, and click the button to check record details.

  Figure 8-34 Anomaly detection - Security check overview



- After you enable disconnection analysis, IoTDA automatically analyzes the causes of device disconnections. Disconnection analysis data of the last seven days can be stored at most. You can search for disconnection data by device ID, resource space, product, cause, and time range, and click the button to check record details.

  Figure 8-35 Anomaly detection - Offline analysis overview



----End

8.8 Remote Login

IoTDA allows you to remotely log in to devices from the console over the Secure Shell Protocol (SSH). You can enter commands supported by devices to debug functions and locate faults. This facilitates device management and remote O&M. The following describes how to use this function.

Prerequisites

1. The device runs on Linux.
2. An SSH server has been installed on the device.
3. The device has been IoTDA SDK integrated. IoT Device SDK C v1.1.1 or later is supported. For details, see **IoT Device SDK (C)**.
4. The device is online.

Constraints

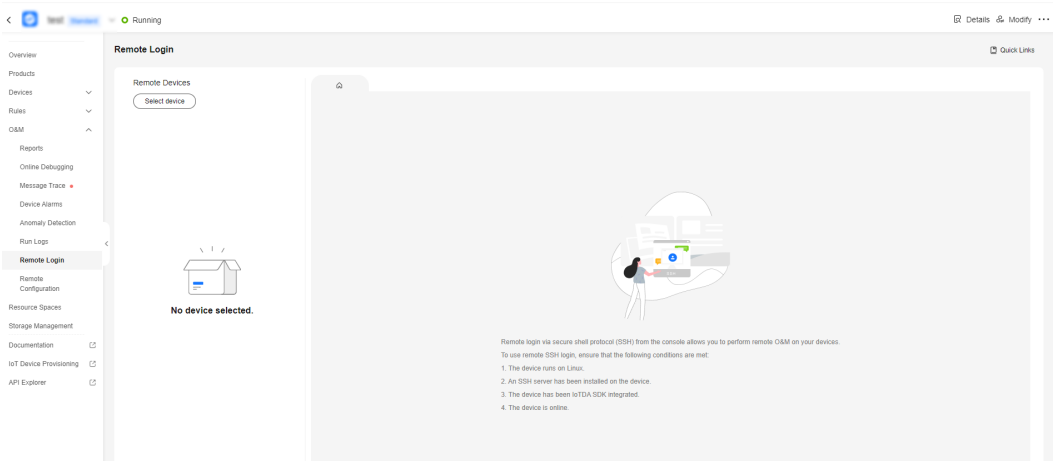
1. Remote connections are based on SSH. IoTDA only establishes SSH channels for devices. You need to develop the management capabilities supported by the console on the device side.
2. Only the standard and enterprise editions support remote login. The domain name access mode must be provided for application access of the enterprise edition.
3. Only one remote connection can be enabled for a device at a time. Up to 100 devices can be remotely logged in at a time for each instance of a tenant.

Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **O&M** > **Remote Login**.

Figure 8-36 Remote login - Remote login page

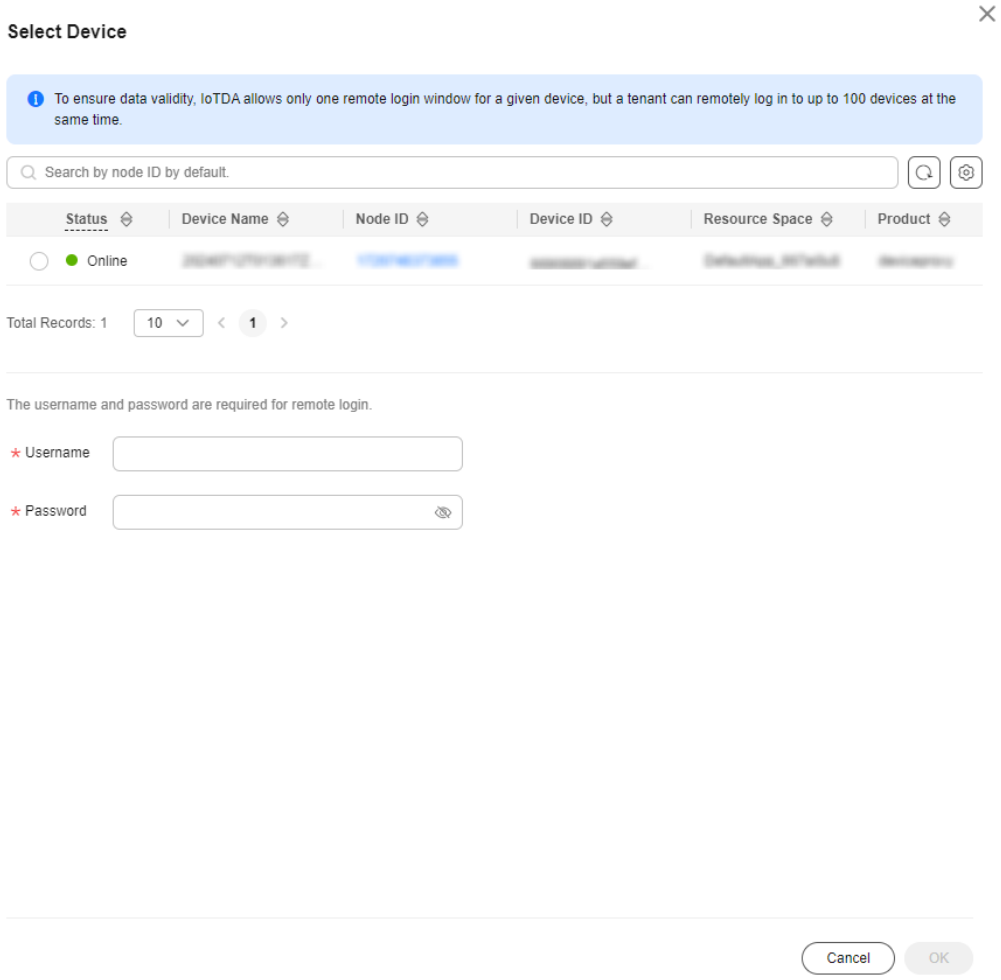


Step 3 Click **Select device** to select the device you want to log in, and enter the username and password for SSH login.

NOTE

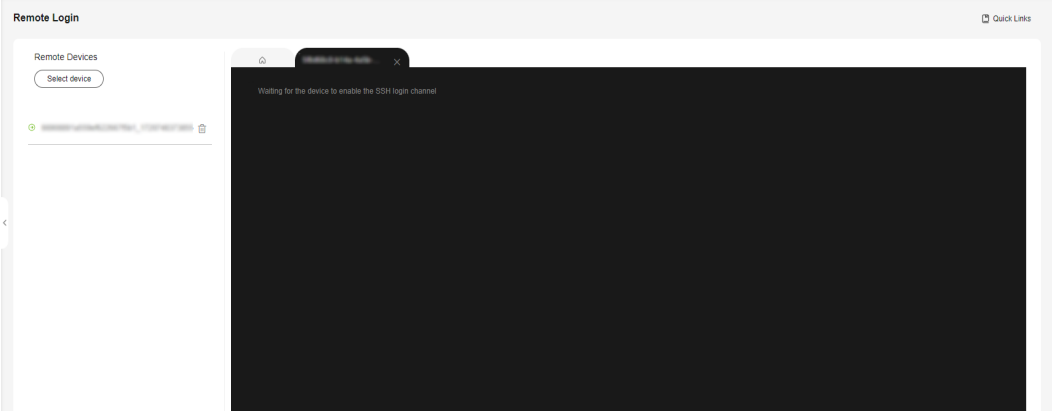
IoTDA does not save the username and password but transparently transfers them to the device.

Figure 8-37 Remote login - Selecting a device



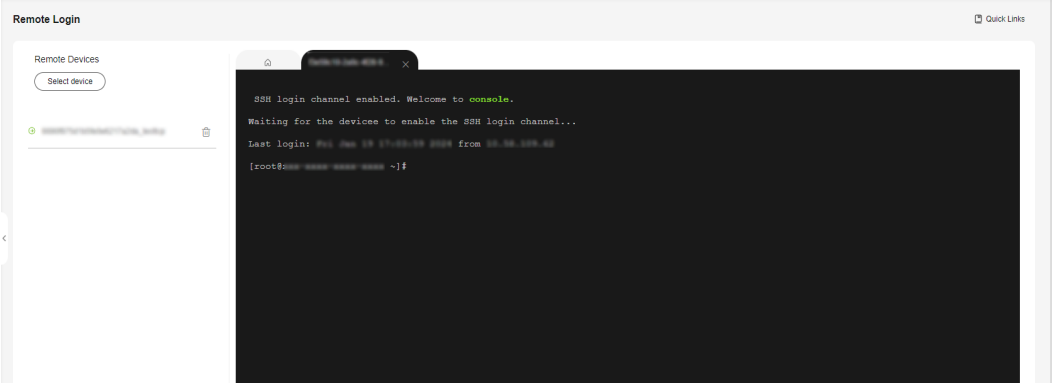
Step 4 Click **OK**, the following remote console page is displayed, prompting you to wait for SSH to be enabled on the device.

Figure 8-38 Remote login - Waiting for the SSH function to be enabled on the device



Step 5 After the login is successful, the following page is displayed. You can manage the device based on its functions.

Figure 8-39 Remote login - Successful remote login



----End

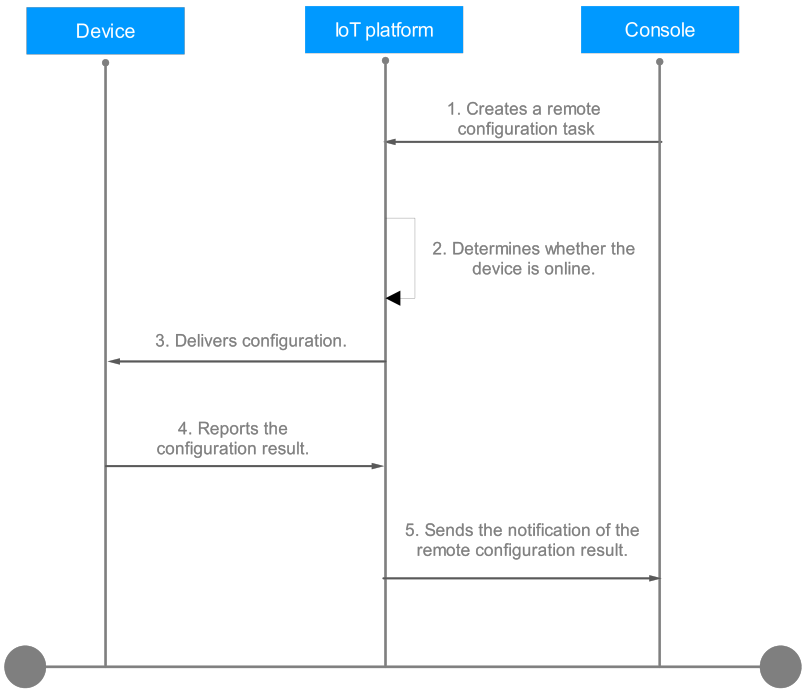
8.9 Remote Device Configuration

Overview

The platform allows you to perform remote configuration. You can remotely update device configuration items such as system and running parameters without interrupting device running.

For example, you can remotely modify system parameters of cashiers running in Windows and the data reporting frequency of T-Boxes in the Internet of Vehicles (IoV) scenarios.

Service Flow



The remote device configuration process is described as follows:

1. A remote configuration task is created on the IoTDA console. Up to 10 remote configuration tasks can run concurrently under an application. Each task can deliver configurations to up to 100,000 devices. If a device is already in an existing remote configuration task and the remote configuration is not complete, a new remote configuration task that contains the device will fail.
2. The platform checks whether the device is online and delivers configurations immediately when the device is online. When the device is offline, the platform waits for the device to go online and subscribe to the **remote configuration topic**. After detecting that the device goes online, the platform delivers configurations. When creating a remote configuration task, you can configure a timeout interval (1 to 30 days). The default timeout interval is 30 days.
3. After configurations are updated, the device calls the API for **reporting the configuration result**.

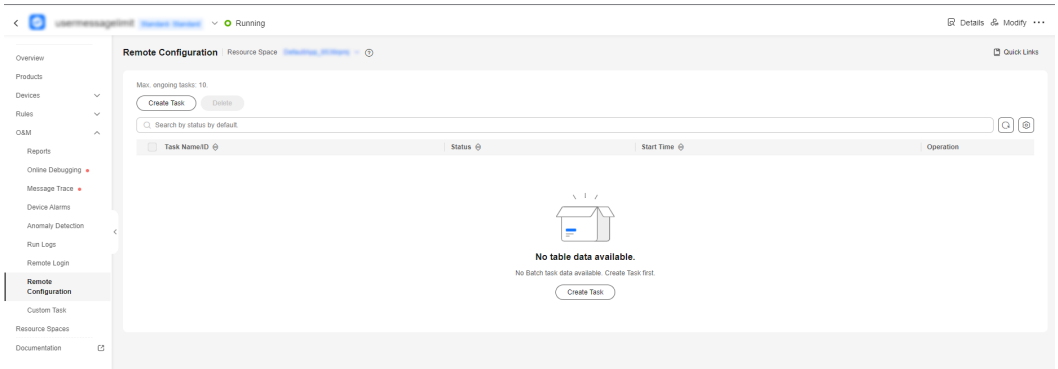
Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **O&M** > **Remote Configuration**.

Step 3 Click **Create Task**.

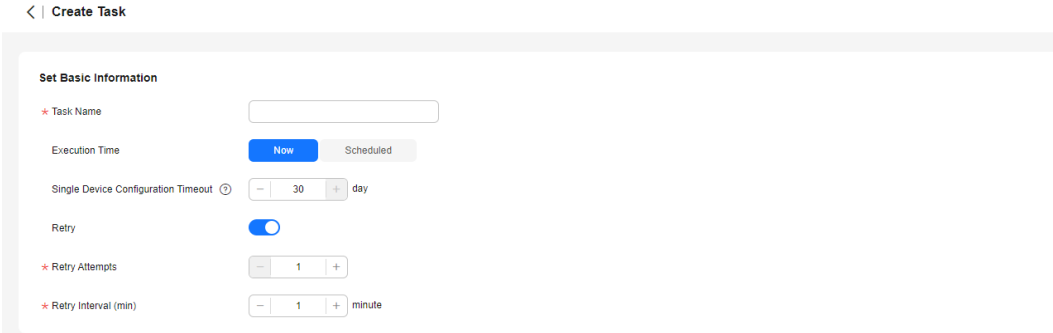
Figure 8-40 Remote configuration - Remote configuration page



Step 4 On the page for creating a remote configuration task, enter a task name, select the execution time, and configure the timeout interval and retry policy.

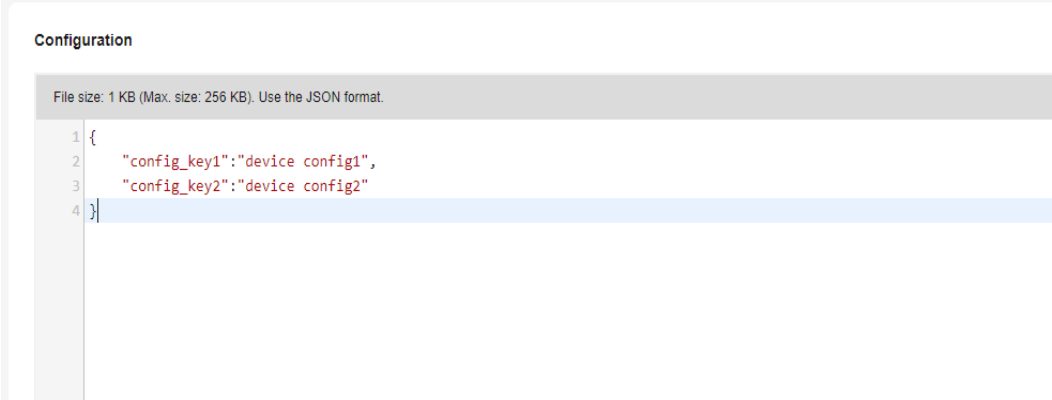
If **Retry** is enabled, you can set the number of retry attempts and retry interval. You are advised to set **Retry Attempts** to **2** and **Retry Interval (min)** to **5**. That is, if the remote configuration fails, the remote configuration will be retried in 5 minutes. (The maximum number of retry attempts is 5 and the maximum retry interval is 1,440 minutes.)

Figure 8-41 Adding remote configuration - Basic information



Step 5 Enter the configuration content in JSON format.

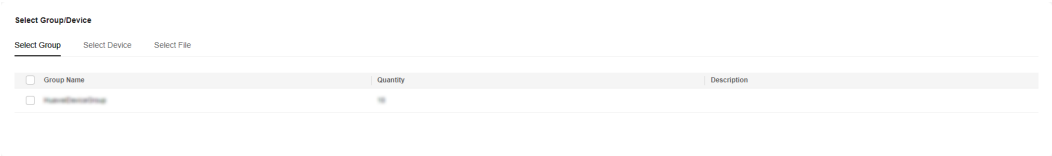
Figure 8-42 Adding remote configuration - Configuration content



Step 6 Select the devices to which the configuration to deliver. You can select a group, upload a file (up to 100,000 devices), or select target devices (up to 30,000

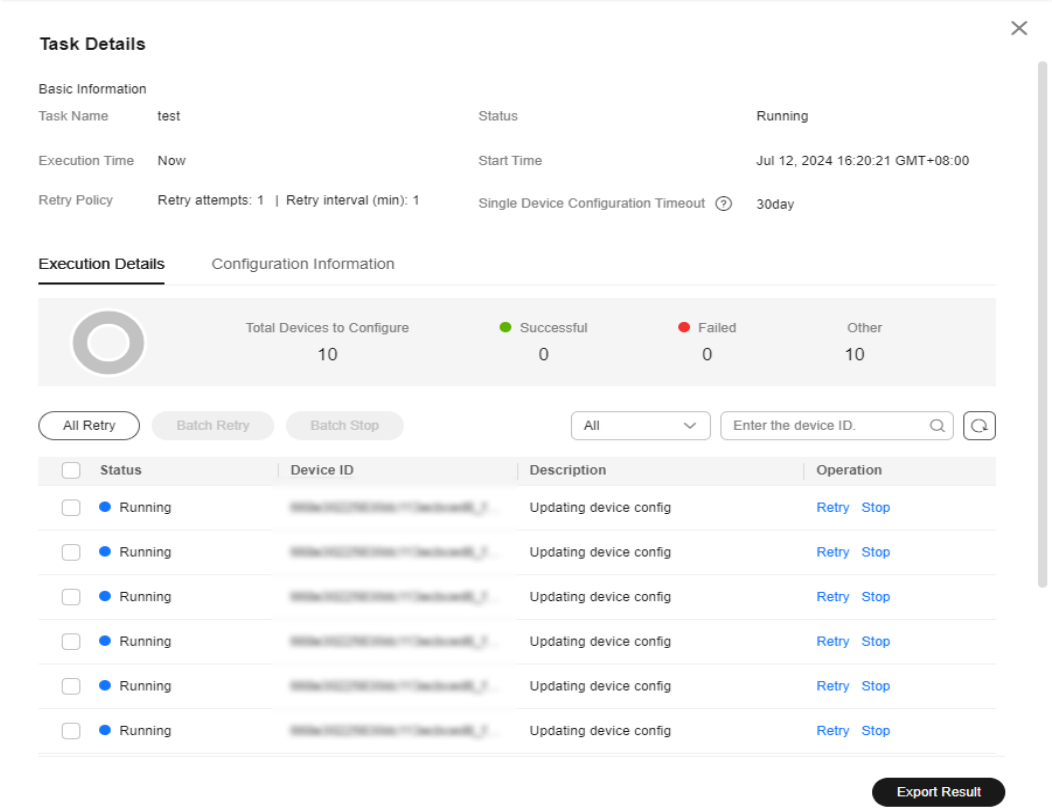
devices) manually. If a large number of devices need to be configured, select a group or upload a file.

Figure 8-43 Adding remote configuration - Device selection



Step 7 After a remote configuration task is created and the device goes online, the device can receive a configuration notification delivered by the platform. After the device updates its configuration and reports the result, you can view the remote configuration result on the task details page. You can stop an executing remote configuration task for a single device or multiple devices (up to 100 devices at a time) in batches. You can also retry a task for a single device or multiple devices (up to 100 devices at a time), or retry all failed remote configuration tasks.

Figure 8-44 Remote configuration - Viewing tasks



----End

9 Granting Permissions Using IAM

9.1 Agency Authorization

Some functions provided by IoTDA need to access user resources. Therefore, you need to create an agency to authorize the access. For details, see **Table 9-1**.

Table 9-1 Agency authorization scenarios

Scenario	Authorization
Uploading a file	obs:object:PutObject obs:bucket:HeadBucket obs:object:GetObject obs:bucket:GetBucketCustomDomainConfiguration KMS Administrator (encryption scenario)
Upgrading software/firmware	obs:object:GetObject KMS Administrator (encryption scenario)
Forwarding data to DIS	DIS Administrator
Forwarding data to FunctionGraph	FunctionGraph:function:list FunctionGraph:function:invokeAsync
Forwarding data to OBS	obs:bucket:ListAllMyBuckets obs:object:GetObject obs:object:PutObject KMS Administrator (encryption scenario)
Forwarding data to LTS	lts:groups:get lts:topics:get

Scenario	Authorization
Forwarding data to BCS Fabric	bcs:fabricInstance:getDetail bcs:fabricInstance:downloadSdkCfg bcs:fabricInstance:downloadCert
Forwarding data to BCS HW	bcs:huaweiCloudChain-Chain:downloadSdkConfig bcs:huaweiCloudChainChain:getChain bcs:huaweiCloudChainContract:get
Using codecs	FunctionGraph:function:invoke FunctionGraph:function:getConfig
Using custom authentication functions	FunctionGraph:function:invoke FunctionGraph:function:getConfig
Using SMN notifications of linkage rules	smn:topic:list smn:topic:publish
Using private images for generic-protocol plug-in	swr:repo:listRepos swr:repo:createRepoDomain
Using instance maintenance window notifications	smn:topic:list smn:topic:publish
Configuring private connections	vpcep:permissions:update vpcep:epservices:create vpcep:epservices:list vpcep:connections:update
Configuring private connections to DMS	dms:instance:get dms:instance:modify vpcep:permissions:update vpcep:epservices:create vpcep:epservices:list vpcep:connections:update
Creating an enterprise edition instance	vpc:securityGroups:get vpc:ports:delete vpc:subnets:get vpc:subnets:update vpc:vpcs:get vpcep:endpoints:create vpcep:endpoints:get vpcep:endpoints:delete

Authorization Scenarios

When you use **Table 9-1** for the first time, the page for creating agency authorization is displayed, showing the function list and scope of authorization. After you agree to the authorization, IoTDA creates an agency named **iotda_admin_trust** in IAM, after the authorization is successful, you can view the created agency in the agency list on the IAM console.

