ALIBABA CLOUD

Alibaba Cloud

物联网平台

Communications

Document Version: 20220711

Alibaba Cloud

Legal disclaimer

Alibaba Cloud reminds you to carefully read and fully understand the terms and conditions of this legal disclaimer before you read or use this document. If you have read or used this document, it shall be deemed as your total acceptance of this legal disclaimer.

1. You shall download and obtain this document from the Alibaba Cloud website or other Alibaba Cloud-authorized channels, and use this document for your own legal business activities only. The content of this document is considered confidential information of Alibaba Cloud. You shall strictly abide by the confidentiality obligations. No part of this document shall be disclosed or provided to any third party for use without the prior written consent of Alibaba Cloud.

2. No part of this document shall be excerpted, translated, reproduced, transmitted, or disseminated by any organization, company or individual in any form or by any means without the prior written consent of Alibaba Cloud.

3. The content of this document may be changed because of product version upgrade, adjustment, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and an updated version of this document will be released through Alibaba Cloud-authorized channels from time to time. You should pay attention to the version changes of this document as they occur and download and obtain the most up-to-date version of this document from Alibaba Cloud-authorized channels.

4. This document serves only as a reference guide for your use of Alibaba Cloud products and services. Alibaba Cloud provides this document based on the "status quo", "being defective", and "existing functions" of its products and services. Alibaba Cloud makes every effort to provide relevant operational guidance based on existing technologies. However, Alibaba Cloud hereby makes a clear statement that it in no way guarantees the accuracy, integrity, applicability, and reliability of the content of this document, either explicitly or implicitly. Alibaba Cloud shall not take legal responsibility for any errors or lost profits incurred by any organization, company, or individual arising from download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, take responsibility for any indirect, consequential, punitive, contingent, special, or punitive damages, including lost profits arising from the use or trust in this document (even if Alibaba Cloud has been notified of the possibility of such a loss).

5. By law, all the contents in Alibaba Cloud documents, including but not limited to pictures, architecture design, page layout, and text description, are intellectual property of Alibaba Cloud and/or its affiliates. This intellectual property includes, but is not limited to, trademark rights, patent rights, copyrights, and trade secrets. No part of this document shall be used, modified, reproduced, publicly transmitted, changed, disseminated, distributed, or published without the prior written consent of Alibaba Cloud and/or its affiliates. The names owned by Alibaba Cloud shall not be used, published, or reproduced for marketing, advertising, promotion, or other purposes without the prior written consent of Alibaba Cloud. The names owned by Alibaba Cloud include, but are not limited to, "Alibaba Cloud", "Aliyun", "HiChina", and other brands of Alibaba Cloud and/or its affiliates, which appear separately or in combination, as well as the auxiliary signs and patterns of the preceding brands, or anything similar to the company names, trade names, trademarks, product or service names, domain names, patterns, logos, marks, signs, or special descriptions that third parties identify as Alibaba Cloud and/or its affiliates.

6. Please directly contact Alibaba Cloud for any errors of this document.

Document conventions

Style	Description	Example
Danger	A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	**Danger:** Resetting will result in the loss of user configuration data.
Warning	A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	**Warning:** Restarting will cause business interruption. About 10 minutes are required to restart an instance.
Notice	A caution notice indicates warning information, supplementary instructions, and other content that the user must understand.	**Notice:** If the weight is set to 0, the server no longer receives new requests.
Note	A note indicates supplemental instructions, best practices, tips, and other content.	**Note:** You can use Ctrl + A to select all files.
>	Closing angle brackets are used to indicate a multi-level menu cascade.	Click **Settings**> **Network**> **Set network type**.
Bold	Bold formatting is used for buttons , menus, page names, and other UI elements.	Click **OK**.
`Courier font`	Courier font is used for commands	Run the `cd /d C:/window` command to enter the Windows system folder.
Italic	Italic formatting is used for parameters and variables.	`bae log list --instanceid` *Instance_ID*
[] or [a|b]	This format is used for an optional value, where only one item can be selected.	`ipconfig [-all|-t]`
{} or {a|b}	This format is used for a required value, where only one item can be selected.	`switch {active|stand}`

Table of Contents

1.Overview

After your devices and server are connected to IoT Platform, the devices and server can communicate by using IoT Platform.

Devices send data to IoT Platform

After you connect devices to IoT Platform, the devices can communicate with IoT Platform. The devices can use either of the following ways to send data to IoT Platform:

- Use custom topics to send custom-format data.
  i. On the IoT Platform console, set **Allowed Operations** of a custom topic to **Publish**. Product topics are automatically mapped to the devices under the product.

     You can create custom topics by using either of the following methods:

     - Use the IoT Platform console. For more information, see Custom topics.
     - Use IoT Platform SDK. You can call the CreateProductTopic API operation to create a custom topic.

  ii. Configure devices to send messages to custom topics when you develop the devices.

     You must configure custom topics and message formats for sending messages on devices. For information about how to use Link SDK that is provided by IoT Platform, see The server receives messages from the device.

- Use Thing Specification Language (TSL)-specific topics to send data.

  For more information about TSL features, see What is a TSL model?.

  Devices can submit properties and events.

  Procedure

  i. In the IoT Platform console, define TSL features based on your business requirements. For more information, see Add a TSL feature.

  ii. When you develop your devices, configure the devices to submit properties and events based on the defined TSL features.

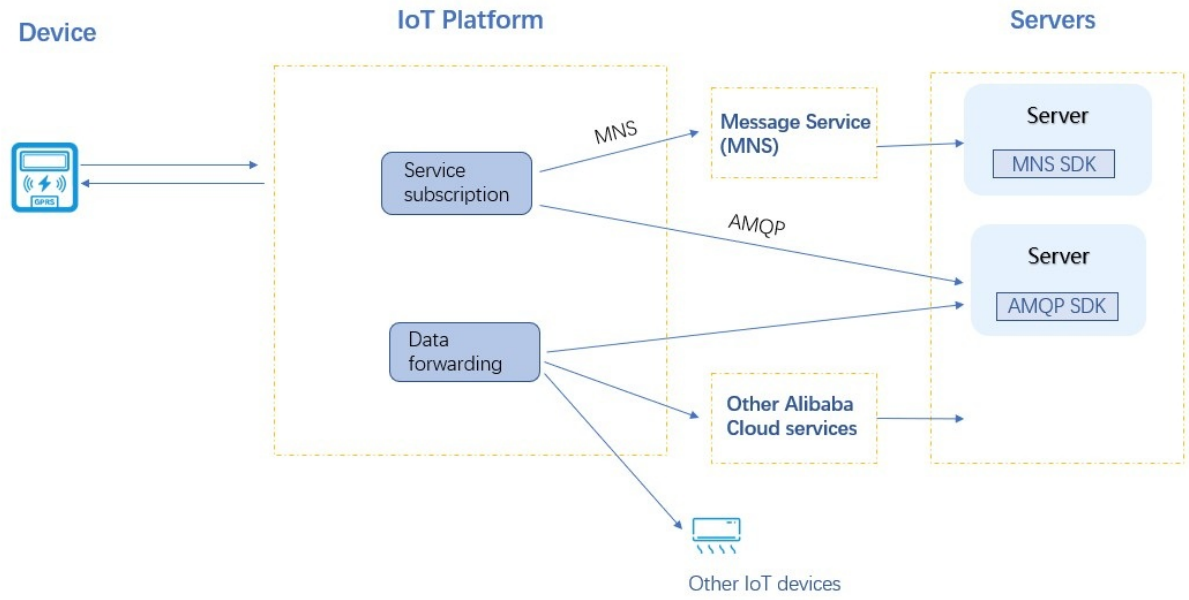
     For more information about the standard data formats for properties and events that are submitted by devices, see Devices submit property information to IoT Platform and Devices submit events to IoT Platform.

     > **Note** IoT Platform provides predefined TSL-specific topics. You can directly use these topics.

     For information about how to use the Link SDK that is provided by IoT Platform, see Submit properties and events by using Link SDK.

IoT Platform forwards data to enterprise servers

You can configure IoT Platform to use either of the following methods to forward data to your server. The data that can be forwarded includes messages submitted by devices, changes of device status, changes of device lifecycles, historical TSL data, firmware update status, information of new sub-devices that are discovered by gateways, and changes of device topologies. Messages are forwarded based on topics. For more information about the data formats of topics, see Data formats.

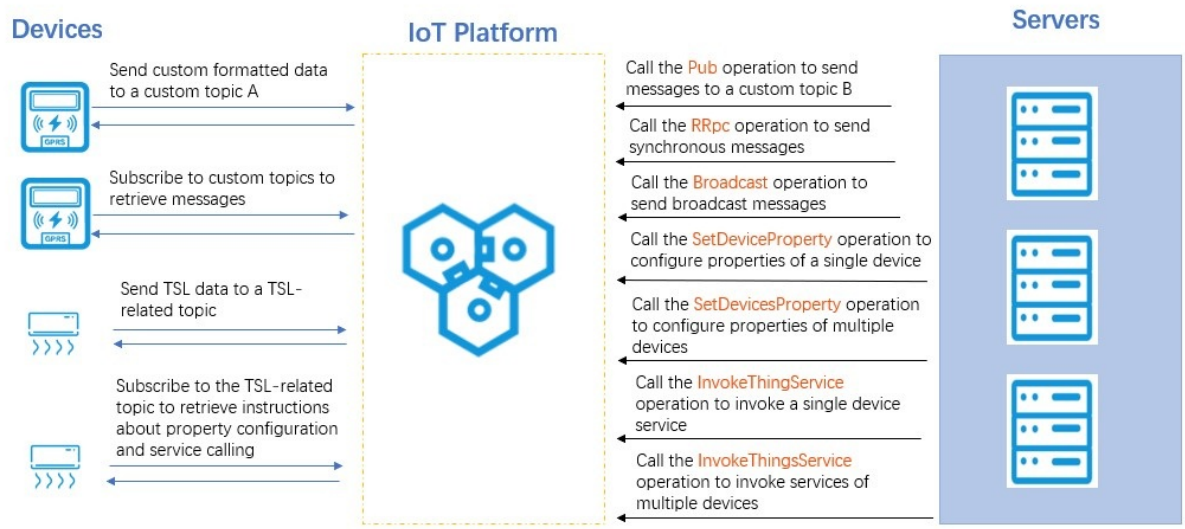


- Server-side subscription: You can use the server-side subscription feature provided by IoT Platform to subscribe to messages of one or more types. IoT Platform can forward messages of specified types from all devices under the product to your server based on your subscription settings. You can use either of the following methods to configure a server-side subscription:
  - Use the AMQP SDK to receive device data that is forwarded by IoT Platform. For more information, see AMQP server-side subscription.
  - Use Message Service (MNS) SDK to receive device data that is forwarded by IoT Platform to MNS queues. For more information, see MNS server-side subscription.
- Data forwarding: You can use the data forwarding feature of the rules engine to forward specified device data to MNS topics or RocketMQ queues based on data forwarding rules. You can receive messages on your server by using MNS SDK or RocketMQ SDK.
  i. Create a rule to forward device data to an MNS topic or RocketMQ queue. For more information, see Configure a data forwarding rule.
  ii. Use MNS SDK to receive messages. For more information, see Manage MNS topics.

For information about the differences between data forwarding and server-side subscription, see Compare data forwarding solutions.

Perform remote control on devices

You can use IoT Platform SDK on your server to achieve remote control on devices. To perform remote control on devices, you must call API operations to send commands from IoT Platform to the devices. The server can use either of the following ways to send commands:



- Use custom topics to control remote devices.
  - Asynchronous control: Call the Pub operation to send custom-format data to a custom topic whose Allowed Operations parameter is set to **Subscribe**. Devices subscribe to this topic to receive messages.

    > **Note** You cannot call the Pub operation to send TSL-related commands.

    For more information about how to use custom topics to control remote devices, see The server sends messages to the device.
  - Synchronous control: Call the RRpc operation to send messages to specified devices and synchronously retrieves the responses.

    For more information about synchronous MQTT communication, see What is RRPC?.

    For more information about how to call the RRpc operation for synchronous communication, see Remotely control a Raspberry Pi server.
  - Batch control: Call the PubBroadcast operation to broadcast messages to all online devices.

    For more information about how to achieve batch control, see Broadcast messages.
- Use TSL-specific topics.

  You can use TSL-specific API operations to send property setting or service calling commands from IoT Platform to devices.

- Control a single device.
  - Call the SetDeviceProperty operation to send a property setting command to a single device.

    After IoT Platform sends the command, the device receives and runs the command in an asynchronous manner. To check whether the device properties are updated, you must view the properties that are later submitted by the device.
  - Call the InvokeThingService operation to send a service calling command to a single device.

    The Invoke Method parameter determines whether a service is called in a synchronous or asynchronous manner. This parameter is specified when you customize the service.

    If the Invoke Method parameter is set to **Synchronous**, the response is synchronously returned after the InvokeThingService operation is called.

    If the Invoke Method parameter is set to **Asynchronous**, the response is asynchronously returned after the InvokeThingService operation is called. You can use the rules engine to retrieve the response. You must set the data source of the SQL query for the rule to **TSL Data Reporting** and set the specific topic to `thing/downlink/reply/message`. For more information about how to configure data forwarding rules, see Configure a data forwarding rule.
- Control multiple devices.
  - Call the SetDevicesProperty operation to send a property setting command to multiple devices.
  - Call the InvokeThingsService operation to send a service calling command to multiple devices.

For more information about how to use TSL-specific topics to control remote devices, see Establish TSL-based communication.

Achieve communication between devices

You can connect devices at two ends to IoT Platform and use IoT Platform to process connection and communication requests between the devices. The following topics describe the two methods that can be used to achieve communication between devices:

- Use the rules engine to establish M2M communication
- Use topic-based message routing to establish M2M communication

Sample scenarios of using device data

Push device data to DingTalk groups

2.Manage consumer groups

A consumer group is used to identify a message consumer. To connect a message consumer to IoT Platform, add the message consumer to a consumer group. Then, the message consumer can receive the messages that are forwarded by IoT Platform to the consumer group. This topic describes how to create, view, and delete a consumer group in the IoT Platform console.

Context

- Feature description:

  You can use the following methods to listen to consumer groups and obtain forwarded messages:

  - Configure an AMQP server-side subscription: You can use an Advanced Message Queuing Protocol (AMQP) server-side subscription to subscribe to a specified type of messages that are sent by all devices of a product. This subscription also allows you to forward these messages to a specified consumer group.
  - Configure a data forwarding rule: You can use the data forwarding feature to forward messages from a specified topic to an AMQP consumer group.

  For more information about the differences between server-side subscriptions and data forwarding, see Compare data forwarding solutions.

- Considerations:

  You can specify a consumer group ID for an AMQP client. Then, the AMQP client can connect to IoT Platform and receive messages. For more information, see Connect an AMQP client to IoT Platform.

  A consumer group can contain up to 64 AMQP clients. These clients share the ID of the consumer group. After a device message is sent to IoT Platform, IoT Platform forwards the message to a random client in the consumer group.

Create a consumer group

1. Log on to the IoT Platform console.
2.
3. In the left-side navigation pane, choose **Rules > Server-side Subscription** and click the **Consumer Groups** tab.
4. Click **Create Consumer Group**.
5. In the **Create Consumer Group** dialog box, enter a group name and click **OK**.

   The consumer group name must be 4 to 30 characters in length, and can contain letters, digits, and underscores (_). Each Chinese character or Japanese character uses 2 characters.

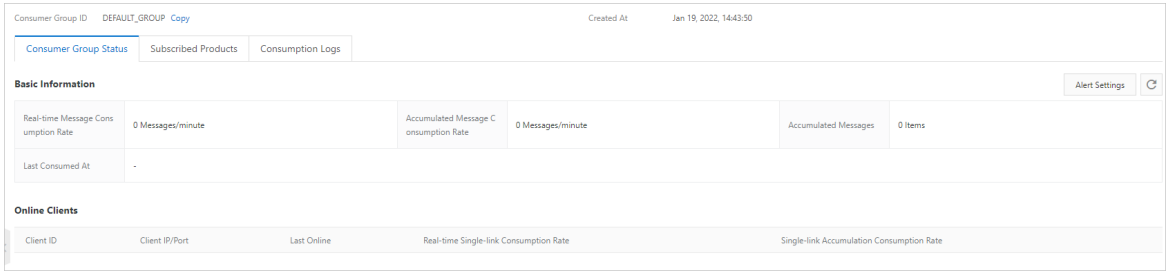
View and monitor consumer groups

You can view the message consumption rate and the number of accumulated messages of a consumer group. To monitor the consumer group, you can configure an alert rule in the CloudMonitor console.

1. On the **Consumer Groups** tab, find the consumer group that you want to manage and click **View**.
2. On the **Consumer Group Status** tab, you can view the values of the **Real-time Message Consumption Rate**, **Accumulated Message Consumption Rate**, **Accumulated Messages**, and **Last Consumed At** parameters. You can also view online clients.

   If the number of accumulated messages is greater than 0, click **Clear** to the right of **Accumulated**

Messages to clear these messages.



3. On the **Consumer Group Status** tab, click **Alert Settings**. In the **Create Alert Rule** panel, configure an alert rule to monitor the number of accumulated messages in the consumer group and the consumption rate, and receive alert notifications.

   Set the **Product** parameter to **IoT Platform-Server subscription**. Configure other parameters based on your business requirements. For more information, see Create a threshold-triggered alert rule.

4. On the **Consumer Group Details** page, click **Consumption Logs**. On the Consumption Logs tab, you can view the details of consumption records.

Delete a consumer group

You can delete a consumer group. However, you cannot delete the default consumer group. After a consumer group is deleted, all clients in the group can no longer receive messages.

1. Disassociate a subscription. If a consumer group is associated with a subscription, you must disassociate the subscription with the consumer group. If the consumer group has no subscriptions, skip this step.
   i. On the **Consumer Groups** tab, find the consumer group and click **View**.
   ii. On the **Subscribed Products** tab of the **Consumer Group Details** page, find the product and click **Unsubscribe**. In the message that appears, click **OK**.

   > **Note** If only one consumer group is associated with a server-side subscription, the unsubscribe action is unavailable on the **Consumer Group Details** page. You must edit or delete the subscription on the **Subscriptions** tab of the **Server-side Subscription** page.

2. On the **Consumer Groups** tab of the **Server-side Subscription** page, find the consumer group and click **Delete**. In the message that appears, click **OK**.

References

For more information about how to specify a consumer group ID for an AMQP client and use the AMQP client to receive messages, see the following topics:

- Connect an AMQP client to IoT Platform
- Connect a client to IoT Platform by using the SDK for Java
- Connect a client to IoT Platform by using the SDK for Node.js
- Connect a client to IoT Platform by using the SDK for .NET
- Connect a client to IoT Platform by using the SDK for Python 2.7
- Connect a client to IoT Platform by using the SDK for Python 3
- Connect a client to IoT Platform by using the SDK for PHP

- Connect a client to IoT Platform by using the SDK for Go

Related API operations

API	Description
CreateConsumerGroup	Creates a consumer group that is required by an AMQP server-side subscription.
UpdateConsumerGroup	Modifies the name of a consumer group.
QueryConsumerGroupByGroupId	Queries the details of a consumer group by consumer group ID.
QueryConsumerGroupList	Queries all consumer groups of an Alibaba Cloud account or performs a fuzzy search by consumer group name.
QueryConsumerGroupStatus	Queries the status of a consumer group when an AMQP server-side subscription is enabled. The status information includes the online client information, message consumption rate, number of accumulated messages, and latest message consumption time.
ResetConsumerGroupPosition	Clears the accumulated messages of a consumer group when an AMQP server-side subscription is enabled.
DeleteConsumerGroup	Deletes a consumer group.

3.Server-side Subscription

3.1. What is server-side subscription?

A server can subscribe to various types of product messages. These messages include device upstream messages, device status changes, gateway and sub-device connections, device lifecycle changes, and device topology changes. After you configure server-side subscription, IoT Platform forwards the subscribed messages from all devices of the product to your server.

Scenarios

Server-side subscription can be applied to the following scenarios:

- Your server receives upstream messages from a large number of devices at the same time.
- Your server receives subscribed messages from all devices of a product.

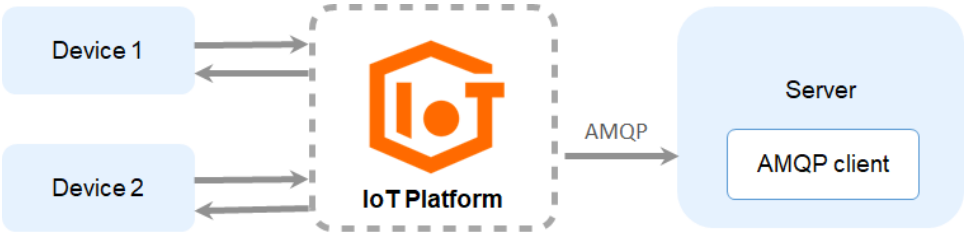
> **Note** If you have multiple servers that consume messages from the same product, the messages are randomly forwarded to one of the servers.

For information about the difference between server-side subscription and data forwarding, see Compare data forwarding solutions.

Use an AMQP server to subscribe to device messages

AMQP refers to Advanced Message Queuing Protocol. After you configure AMQP server-side subscription, IoT Platform forwards all subscribed device messages of a product to your server over AMQP.

The following figure shows how messages are forwarded in AMQP server-side subscription.



Benefits of AMQP server-side subscription:

- Supports multiple consumer groups. You can create multiple consumer groups for different environments. For example, you can create two consumer groups: Group A and Group B. Group A subscribes to the messages of Product A in the development environment and Group B subscribes to the messages of Product B in the production environment.

  > **Note** If multiple consumer groups subscribe to the messages of Product B, the consumer groups can receive the same messages from Product B at the same time.

- Facilitates troubleshooting. Server-side subscription allows you to view the client status, accumulated messages, and message consumption rate.
- Supports linear scalability. You can significantly improve the message forwarding capability by adding consumer clients.

- Pushes real-time messages first. Message accumulation does not affect the service.

  Real-time messages take precedence over accumulated messages if the push rate limit is reached or failures occur.

  Even if messages are accumulated due to client failures or low consumption rates, real-time messages are sent with accumulated messages after clients recover. Then, devices can push messages in real time again.

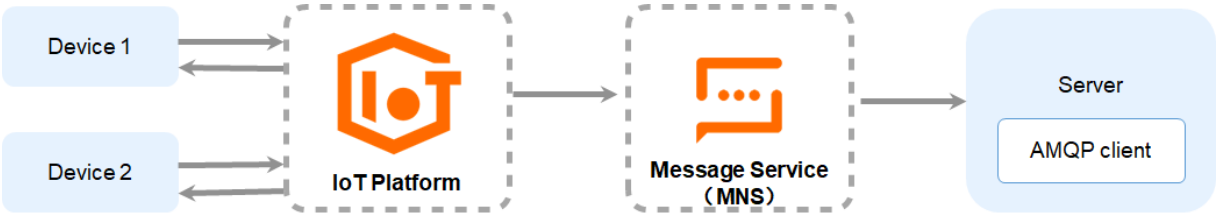
Before you use AMQP server-side subscription, you must configure it in the console. For more information, see Configure an AMQP server-side subscription. You must develop an AMQP client and connect it to IoT Platform. Then, you can use the client to receive messages. For more information, see Connect an AMQP client to IoT Platform.

> **Note** After you enable AMQP server-side subscription in the IoT Platform, you are billed based on the number of forwarded messages. For information about the billing methods, see Messaging fees.

Use the MNS server to subscribe to device messages

After you configure MNS server-side subscription, IoT Platform forwards subscribed messages to Message Service (MNS) queues. The MNS client in your server receives device messages by listening to the MNS queues.

The following figure shows how messages are forwarded in MNS server-side subscription.



For information about how to configure MNS server-side subscription, see Use Message Service (MNS) to subscribe to device messages.

> **Note** You are billed based on the queues that you create and the messages that you receive. For more information about the billing methods, see Message Service documentation.

3.2. Limits of server-side subscriptions

This article describes the limits of server-side subscriptions.

Limits on AMQP subscriptions

Item	Description
Authentication timeout	An authentication request is sent after a connection is established. If the authentication fails within 15 seconds, the server ends the connection.

Item	Description
Data timeout	When a server establishes a connection with IoT Platform, the heartbeat timeout period (the idle-timeout parameter in AMQP) must be specified. The timeout period ranges from 30 to 300 seconds. If no frame is transmitted within the heartbeat timeout period, IoT Platform ends the connection. After the connection is established, the server must send ping packets within the heartbeat timeout period to maintain the connection. Otherwise, IoT Platform ends the connection. **Note** If the connection is established by using an Alibaba Cloud SDK, the server does not need to send ping packets to maintain the connection. During the keep-alive time that is provided by the SDK, make sure that the main process does not exit.
Policy for message pushing retries	Messages may not be consumed in real time due to some issues. In this case, these messages are accumulated. These issues include that consumers disconnect from IoT Platform and the speed at which these messages are consumed is slow. • After these consumers re-connect to IoT Platform and start to consume messages at a stable speed, IoT Platform pushes accumulated messages to these consumers. • If consumers fail to consume these pushed messages, the queue where accumulated messages reside may be blocked. After an interval of about 1 minute, IoT Platform retries to push accumulated messages to consumers.
Maximum number of saved messages	Each consumer group can retain a maximum of 100 million messages.
Message retention period	One day.
Maximum push rate for real-time messages	Each connection can be used to process a maximum of 1,000 transactions per second (TPS). A maximum of 64 connections can be established.
Maximum push rate for accumulated messages	A consumer group can process a maximum of 200 TPS. **Note** To prevent a large number of accumulated messages, make sure that consumers are connected to IoT Platform. You must also make sure that these consumers send ACK responses to messages that are pushed by IoT Platform.
Maximum number of consumer groups with which a product can be associated	10.

Item	Description
Maximum number of products with which a consumer group can be associated	1,000.
Maximum number of consumer groups	Each Alibaba Cloud account can create a maximum of 1,000 consumer groups.
Maximum number of consumers	Each consumer group can have a maximum of 64 consumers.
Maximum connection requests	Each consumer can initiate a maximum of 100 connection requests within 1 minute. **Note** Consumers indicate AMQP clients that are used to receive IoT Platform messages. These consumers are not devices.

Limits on MNS subscriptions

For more information about the limits of Message Service (MNS) server-side subscriptions, see the limits of MNS queues in the MNS limits topic.

Note

- After you create an MNS server-side subscription, you are charged for the usage of MNS queues regardless of whether IoT Platform forwards messages to MNS queues.
- The maximum size of a message that can be received by an MNS queue is 64 KB. If the size of a message exceeds the limit, the message is discarded.

3.3. Use AMQP server-side subscription

3.3.1. Configure an AMQP server-side subscription

IoT Platform allows you to use a consumer group to subscribe to the messages of a topic and then forward the messages to an Advanced Message Queuing Protocol (AMQP) server. This topic describes how to configure and manage an AMQP server-side subscription in the IoT Platform console.

Prerequisites

A consumer group is created. This consumer group is used to subscribe to topic messages. You can use the default consumer group (DEFAULT_GROUP) or create a consumer group in IoT Platform. For more information, see Manage consumer groups.

Configure a subscription

To specify the type of messages to which you want to subscribe, perform the following steps:

1. Log on to the IoT Platform console.
2.
3. In the left-side navigation pane, choose **Rules > Server-side Subscription**.
4. On the **Server-side Subscription** page, click **Create Subscription**.
5. In the **Create Subscription** dialog box, configure the parameters and click **OK**. The following table describes the parameters.

Parameter	Description
Products	Select the product to which the devices belong. The messages submitted by these devices are pushed to consumers.
Subscription Type	Select **AMQP**.
Consumer Group	Select consumer groups. You can specify multiple consumer groups for a product. You can also create multiple subscriptions to products by using a consumer group. IoT Platform provides a default consumer group to consume messages. To group consumers, perform the following steps to create a consumer group: In the lower-right corner of the **Select Consumer Groups** dialog box, click **Create Consumer Group** and set the required parameters. For more information about consumer groups, see Manage consumer groups.
	Select the types of messages. You can subscribe to the following types of device messages: ○ **Device Upstream Notification**: these messages in the topics whose **Allowed Operations** parameter is set to **Publish**. For more information, see What is a topic?. These messages include custom data and Thing Specification Language (TSL) data that is submitted by devices. The upstream TSL data includes property data, event data, responses to property setting requests, and responses to service calls. The TSL data that is pushed to user servers is processed by IoT Platform. For more information about data formats, see Data formats. For example, the following three topic categories are defined for a product: ■ `/${YourProductKey}/${YourDeviceName}/user/get`: The Allowed Operations parameter of this topic category is set to Subscribe. ■ `/${YourProductKey}/${YourDeviceName}/user/update`. The Allowed Operations parameter of this topic category is set to Publish. ■ `/sys/${YourProductKey}/${YourDeviceName}/thing/event/property/post`. The Allowed Operations parameter of this topic category is set to Subscribe. IoT Platform pushes messages of the following topic categories: `/${YourProductKey}/${YourDeviceName}/user/update` and `/sys/${YourProductKey}/${YourDeviceName}/thing/event/property/post`.

Parameter	Description
Message Type	**Notice** You may need to subscribe to response data of asynchronous service calls. In this case, the message ID returned by the device must be the same as the message ID generated when IoT Platform sends downstream data. ○ **Device Status Change Notification**: the notifications that devices send when the online or offline status changes. ○ **Gateway's sub-devices discovery report**: the sub-device data that gateways submit when these gateways detect new sub-devices. The gateways must have the applications that can be used to detect sub-devices. This message type is specific to gateways. ○ **Device Topological Relation Changes**: the notifications that gateways send when topological relationships between sub-devices and the gateways are created or deleted. This message type is specific to gateways. ○ **Device Changes Throughout Lifecycle**: the notifications that devices send when the devices are created, deleted, enabled, or disabled. ○ **TSL Historical Data Reporting**: the historical properties and events that are submitted by devices. ○ **OTA Update Status Notification**: the notifications that devices send during firmware verification and batch update. When a device update task succeeds, fails, is canceled, or is being implemented, a notification is pushed. ○ **Device tag change**: the messages that devices send when device tags change. ○ **Submit a module version number**: the messages that devices send when OTA module versions change. ○ **Batch status notification**: the notifications that devices send when the status of OTA update batches changes. **Note** When you listen to device messages, you may need to filter or process the messages. In this case, you can forward messages to an AMQP consumer group by using data forwarding rules. Then, you can receive the messages by using the AMQP client. For more information, see Data forwarding. For more information about how to forward messages to an AMQP server and the device topics to which you can subscribe, see Topics that are related to the rules engine and device communication.

What's next

Configure an AMQP client: We recommend that you use the AMQP SDK provided by IoT Platform. Alibaba Cloud does not provide technical support for user-developed AMQP SDKs.

Note After the settings are configured and submitted device data is received by an AMQP client, you can view the operation logs in the IoT Platform console.Choose **Maintenance > Device Log > Cloud run log**. The log entries are generated when a device submits data, IoT Platform forwards the data to the AMQP client, and the AMQP client returns an ACK message.

3.3.2. Connect an AMQP client to IoT Platform

This article describes how to connect an Advanced Message Queuing Protocol (AMQP) client to IoT Platform. You can perform this operation after you configure an AMQP server-side subscription in the IoT Platform console. After you connect an AMQP client to IoT Platform, you can receive device messages by using the AMQP client on your business server.

Protocol versions

For more information, see AMQP.

The server-side subscriptions of IoT Platform support only the AMQP 1.0 protocol.

Procedure

1. An AMQP client establishes a TCP connection with IoT Platform by using a three-way handshake. Then, a TLS handshake is performed to authenticate the AMQP client.

   > **Note** To ensure security, AMQP clients must transmit data by using TLS encryption. Data cannot be transmitted over unencrypted TCP channels.

2. The client requests to establish a connection.

   The PLAIN SASL mechanism is used to authenticate the connection. The authentication is based on a username and password. After the username and password-based authentication is passed by IoT Platform, the connection is established.

   AMQP requires that the open frame includes the idle-time-out field when a connection is established. This field specifies the heartbeat timeout period. The heartbeat timeout period ranges from 30,000 to 300,000 ms. If no frame is transmitted over the connection after the heartbeat timeout period expires, IoT Platform ends the connection. The method that is used to configure the idle-time-out field varies based on the programming language of the SDK. For more information, see the sample code of language-specific SDKs.

3. The client sends a request to IoT Platform to establish a receiver link. The receiver link is a one-way channel that you can use to push data from IoT Platform to the client.

   The client must establish the receiver link within 15 seconds after the AMQP connection is established. Otherwise, IoT Platform ends the AMQP connection.

   After the receiver link is established, the client is connected to IoT Platform.

   > **Note**
   > - Only one receiver link can be created for each connection. Sender links are not supported. IoT Platform can push messages to the client, but the client cannot send messages to IoT Platform.
   > - The class name of the receiver link varies based on the programming language of the SDK. For example, the receiver link in some SDKs is named MessageConsumer in some SDKs.

Connect the AMQP client to IoT Platform

This section describes how to specify the endpoint and configure the authentication parameters when you connect the AMQP client to IoT Platform.

- Endpoint: Specify the endpoint of a public instance or an Enterprise Edition instance to which you want to connect an AMQP client. For more information about the supported endpoints, see View the endpoint of an instance.

  > **Note**
  > - The *${YourHost}* variable in the SDK specifies the endpoint.
  > - Before you connect the client to an IoT Platform instance, make sure that your product and device are created in the instance.

- Port number:
  - If you use a Java, .NET, Python 2.7, Node.js, or Go client, the port number is 5671.
  - If you use a Python 3 or PHP client, the port number is 61614.
- Client-side authentication parameters:

```
userName = clientId|iotInstanceId=${iotInstanceId},authMode=aksign,signMethod=hmacsha1,consumerGroupId=${consumerGroupId},authId=${accessKey},timestamp=1573489088171|
password = signMethod(stringToSign, accessSecret)
```

Parameters in userName

Parameter	Required	Description
clientId	Yes	The ID of the client. We recommend that you use a unique identifier, such as the UUID, MAC address, or IP address of the client. The client ID must be 1 to 64 characters in length. Log on to the IoT Platform console and click the card of the instance that you want to manage. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. Find the consumer group that you want to manage and click **View** in the Actions column. The ID of each client is displayed on the **Consumer Group Details** page. You can use client IDs to efficiently identify clients.
iotInstanceId	No	The ID of the instance. You can view the instance **ID** on the **Overview** page in the IoT Platform console. ○ If your instance has an ID, you must specify the ID for the parameter. ○ If your instance does not have an ID, you do not need to configure the parameter.
authMode	Yes	The authentication mode. Valid value: aksign.
signMethod	Yes	The signature algorithm. Valid values: hmacmd5, hmacsha1, and hmacsha256.

Parameter	Required	Description
consumerGroupId	Yes	The ID of the consumer group. To view the ID of the consumer group, perform the following steps: Log on to the IoT Platform console and click the card of the instance that you want to manange. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. The ID is displayed on the Consumer Groups tab.
authId	Yes	The authentication information. In aksign authentication mode, set the authId parameter to your AccessKey ID. Log on to the IoT Platform console, move the pointer over your profile picture, and then click **AccessKey Management**. On the page that appears, obtain the AccessKey ID. **Note** If you use the AccessKey ID of a RAM user, you must grant the RAM user the permissions to configure AMQP server-side subscription. Otherwise, the connection fails. For more information about the authorization methods, see Authorize a RAM user to access IoT Platform.
timestamp	Yes	The current time. The timestamp is a LONG integer. Unit: milliseconds.

Parameters in password

Parameter	Required	Description
signMethod	Yes	The signature algorithm. Use the signature algorithm that is specified in the userName parameter to calculate the signature value. Then, convert the value into a Base64-encoded string.
stringToSign	Yes	The string to sign. Sort the parameters that must be signed in alphabetical order. Concatenate the key and value of each parameter by using an equal sign (=). Concatenate the parameters by using an ampersand (&). The parameters that must be signed include authId and timestamp. Set the stringToSign parameter to a value in the `stringToSign = authId=${accessKey}&timestamp=1573489088171` format.

Parameter	Required	Description
accessSecret	Yes	The AccessKey secret of your Alibaba Cloud account. Log on to the IoT Platform console, move the pointer over the profile picture, and then click **AccessKey Management**. On the page that appears, obtain the required AccessKey ID. **Note** If you use the AccessKey ID of a RAM user, you must grant the RAM user the permissions to configure AMQP server-side subscription. Otherwise, the connection fails. For more information about the authorization methods, see Authorize a RAM user to access IoT Platform.

Examples

You can use one of the following programming language-specific AMQP SDKs to connect a client to IoT Platform. For more information about how to configure the parameters, see Connect the AMQP client to IoT Platform.

Notice We recommend that you use the AMQP SDK provided by IoT Platform. Alibaba Cloud does not provide technical support for user-developed AMQP SDKs.

- Connect a client to IoT Platform by using the SDK for Java
- Connect a client to IoT Platform by using the SDK for .NET
- Connect a client to IoT Platform by using the SDK for Node.js
- Connect a client to IoT Platform by using the SDK for Python 2.7
- Connect a client to IoT Platform by using the SDK for Python 3
- Connect a client to IoT Platform by using the SDK for PHP
- Connect a client to IoT Platform by using the SDK for Go

When you connect a client, you may receive message-related error codes. For more information, see Message-related error codes.

Receive messages that are pushed by IoT Platform

After a receiver link is established between an AMQP client and IoT Platform, IoT Platform pushes messages to the client over the link.

Note A client can receive only messages of the types that are specified in the subscription. If you want to send messages or commands to a device, you can call the required operation. For more information, see List of operations by function.

Each message that is pushed by IoT Platform to the client consists of the following parts:

- Message body. The payload of the message is in the binary format.
- Message attributes, such as topic and message ID. You can obtain the attributes from the application-properties section that is defined in AMQP. The attributes are in the `key:value` format.

Key	Description
topic	The topic of the message.
messageId	The ID of the message.
generateTime	The time when the message was generated. **Note** You cannot determine the order of messages based on the generateTime parameter.

Message receipt:

The AMQP client sends a receipt to IoT Platform to notify IoT Platform that a message is received. The receipt can be manually or automatically sent. We recommend that you use the automatic mode. For more information, see the user guide of the client.

Messaging policies:

- Messages are pushed in real time.
- Accumulated messages

Note

- The amount of traffic that consumers process may fluctuate during a short period of time. In most cases, the fluctuation disappears within 10 minutes. If the number of queries per second (QPS) is high or the message processing is resource-consuming, we recommend that you increase the number of consumers. This way, you can maintain an acceptable level of redundancy in message consumption.
-
- For more information about the limits on messaging, see Limits of server-side subscriptions.
- You can clear accumulated messages in the IoT Platform console. For more information, see View and monitor consumer groups.

Message ordering:

Note The ordering of messages cannot be ensured.

- Upstream and downstream device messages:

  The time when a message is received from a device is different from the time when the device is connected or disconnected. You can sort messages by the value of the Time parameter.

  For example, you obtain the following messages in sequence:

  i. Online time: `2018-08-31 10:02:28.195`
  ii. Offline time: `2018-08-31 10:01:28.195`
  iii. Offline time: `2018-08-31 10:03:28.195`

The preceding three messages indicate that the status of a device changed to offline, online, and then offline.

For more information about the parameters in messages, see Data formats.

- Other types of messages:

  You must add a sequence number to each message at the application layer. Based on the sequence number of a received message, IoT Platform uses an idempotent algorithm to check whether to process the message.

3.3.3. Connect a client to IoT Platform by using the SDK for Java

This topic describes how to connect an AMQP JMS client to Alibaba Cloud IoT Platform and receive messages from IoT Platform.

Prerequisites

The ID of the consumer group that has subscribed to the messages of a topic is obtained.

- Manage consumer groups: You can use the default consumer group (DEFAULT_GROUP) or create a consumer group in the IoT Platform console.
- Configure an AMQP server-side subscription: You can use a consumer group to subscribe to the messages of a topic.

Prepare the development environment

In this example, the development environment consists of the following components:

- Operating system: Windows 10
- Java Development Kit (JDK): JDK 8
- Integrated development environment (IDE): IntelliJ IDEA Community Edition

Download the Apache Qpid JMS client

To download the client and view the instructions, see Qpid JMS 0.57.0.

In this example, add the following dependency to a Maven project to download the Qpid JMS client:

```
<!-- amqp 1.0 qpid client -->
 <dependency>
   <groupId>org.apache.qpid</groupId>
   <artifactId>qpid-jms-client</artifactId>
   <version>0.57.0</version>
 </dependency>
 <!-- util for base64-->
 <dependency>
   <groupId>commons-codec</groupId>
  <artifactId>commons-codec</artifactId>
  <version>1.10</version>
</dependency>
```

Sample code

1. Download the demo package and decompress it.
2. Open IntelliJ IDEA and import the sample project *aiot-java-demo* in the demo package.

   In the pom.xml file, the Maven dependency is added to download the Qpid JMS client.
3. In the *AmqpClient.java* file under the *src/main/java/com.aliyun.iotx.demo* directory, configure the AMQP connection parameters. The following table describes the parameters.

Parameter	Example	Description
accessKey	LTAI4GFGQvKuqHJhFa******	Log on to the IoT Platform console, move the pointer over the profile picture, and then click **AccessKey Management** to obtain the AccessKey ID and AccessKey secret. **Note** If you use a RAM user, you must attach the AliyunIOTFullAccess permission policy to the RAM user. This policy allows the RAM user to manage IoT Platform resources. Otherwise, the connection with IoT Platform fails. For more information about how to authorize a RAM user, see RAM user access.
accessSecret	iMS8ZhCDdfJbCMeA005sieKe******	
consumerGroupId	VWhGZ2QnP7kxWpeSSjt*******	The ID of the consumer group. To view the ID of the consumer group, perform the following steps: Log on to the IoT Platform console and click the card of the instance that you want to manange. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. The ID is displayed on the Consumer Groups tab.

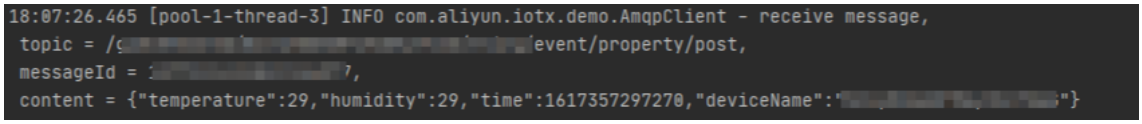
Parameter	Example	Description
iotInstanceId	""	The ID of the instance. You can view the ID of the instance on the **Overview** page in the IoT Platform console. ○ If you have an ID value, you must specify the ID for this parameter. ○ If no **Overview** or ID is generated for your instance, specify an empty string (`iotInstanceId = ""`) for the parameter.
clientId	12345	The ID of the client. We recommend that you use a unique identifier, such as the UUID, MAC address, or IP address of the client. The client ID must be 1 to 64 characters in length. Log on to the IoT Platform console and click the card of the instance that you want to manage. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. Find the consumer group that you want to manage and click **View** in the Actions column. The ID of each client is displayed on the **Consumer Group Details** page. You can use client IDs to efficiently identify clients.
connectionCount	4	The number of connections that are enabled on the AMQP client. Maximum value: 64. This parameter is used for scale-out in real-time message pushing scenarios. On the **Consumer Group Details** page, each connected client is displayed in the format of `${clientId}+"-"+ a number` . The minimum number is 0.
host	198426864******.iot-amqp.cn-shanghai.aliyuncs.com	The AMQP endpoint. For more information about the endpoints that you can specify for the `${YourHost}` variable, see View the endpoint of an instance.

For more information, see Connect an AMQP client to IoT Platform.

4. Run the *AmqpClient.java* program.

> **Note** In this example, the `Thread.sleep(60 * 1000);` code snippet is added to end the program after the program starts and runs for 1 minute. You can set the running time based on your business requirements.

○ After you run the sample code, the following log data is returned. The data indicates that the AMQP client is connected to IoT Platform and can receive messages.



Parameter	Example	Description
topic	/**********/******/thing/event/property/post	The topic that is used to submit device properties.
messageId	1324198300680719360	The ID of the message.
content	null	The content of the message.

- If the output that is similar to the following log information appears, the AMQP client fails to connect to IoT Platform.

  You can check the code or network environment based on logs, solve the problem, and then run the code again.

References

For more information about the error codes that are related to server-side subscription, see Message-related error codes.

3.3.4. Connect a client to IoT Platform by using the SDK for .NET

This topic describes how to use the AMQP SDK for .NET to connect a client to Alibaba Cloud IoT Platform and receive messages from IoT Platform.

Prerequisites

The ID of the consumer group that has subscribed to the messages of a topic is obtained.

- Manage consumer groups: You can use the default consumer group (DEFAULT_GROUP) or create a consumer group in the IoT Platform console.
- Configure an AMQP server-side subscription: You can use a consumer group to subscribe to the messages of a topic.

Development Environment

The following table describes the development environment.

Framework	Supported versions
.Net Framework	3.5, 4.0, 4.5, or later
.NET Micro Framework	4.2 or later

Framework	Supported versions
.NET nanoFramework	1.0 or later
.NET Compact Framework	3.9 or later
.Net Core on Windows 10 and Ubuntu 14.04	1.0 or later
Mono	4.2.1 or later

Download the SDK

We recommend that you use the AMQP.Net Lite library. To download the library and view its instructions, see AMQP.Net Lite.

Add a dependency

Add the following dependency to the *packages.config* file:

```
<packages>
  <package id="AMQPNetLite" version="2.2.0" targetFramework="net47" />
</packages>
```

Sample code

```
using System;
using System.Text;
using Amqp;
using Amqp.Sasl;
using Amqp.Framing;
using System.Threading;
using System.Security.Cryptography.X509Certificates;
using System.Net.Security;
using System.Security.Cryptography;
namespace amqp
{
    class MainClass
    {
        // The endpoint. For more information, see Connect an AMQP client to IoT Platform.
        static string Host = "${YourHost}";
        static int Port = 5671;
        static string AccessKey = "${YourAccessKey}";
        static string AccessSecret = "${YourAccessSecret}";
        static string consumerGroupId = "${YourConsumerGroupId}";
        static string clientId = "${YourClientId}";
        // iotInstanceId: the ID of the instance.
        static string iotInstanceId = "${YourIotInstanceId}";
        static int Count = 0;
        static int IntervalTime = 10000;
        static Address address;
        public static void Main(string[] args)
        {
            long timestamp = GetCurrentMilliseconds();
```

```
            string param = "authId=" + AccessKey + "&timestamp=" + timestamp;
            // The structure of the userName parameter. For more information, see Connect an AMQP client to IoT Platform.
            string userName = clientId + "|authMode=aksign,signMethod=hmacmd5,consumerGroupId=" + consumerGroupId
               + ",iotInstanceId=" + iotInstanceId + ",authId=" + AccessKey + ",timestamp=" + timestamp + "|";
            // Calculate a signature. For more information about how to configure the password parameter, see Connect an AMQP client to IoT Platform.
            string password = doSign(param, AccessSecret, "HmacMD5");
            DoConnectAmqp(userName, password);
            ManualResetEvent resetEvent = new ManualResetEvent(false);
            resetEvent.WaitOne();
        }
        static void DoConnectAmqp(string userName, string password)
        {
            address = new Address(Host, Port, userName, password);
            // Create a connection.
            ConnectionFactory cf = new ConnectionFactory();
            // Use a local TLS certificate if required.
            //cf.SSL.ClientCertificates.Add(GetCert());
            //cf.SSL.RemoteCertificateValidationCallback = ValidateServerCertificate;
            cf.SASL.Profile = SaslProfile.External;
            cf.AMQP.IdleTimeout = 120000;
            // Configure the cf.AMQP.ContainerId and cf.AMQP.HostName parameters.
            cf.AMQP.ContainerId = "client.1.2";
            cf.AMQP.HostName = "contoso.com";
            cf.AMQP.MaxFrameSize = 8 * 1024;
            var connection = cf.CreateAsync(address).Result;
            // The connection exception feature is disabled.
            connection.AddClosedCallback(ConnClosed);
            // Receive messages.
            DoReceive(connection);
        }
        static void DoReceive(Connection connection)
        {
            // Create a session.
            var session = new Session(connection);
            // Create a receiver link and receives messages.
            var receiver = new ReceiverLink(session, "queueName", null);
            receiver.Start(20, (link, message) =>
            {
                object messageId = message.ApplicationProperties["messageId"];
                object topic = message.ApplicationProperties["topic"];
                string body = Encoding.UTF8.GetString((Byte[])message.Body);
                // Note: Do not implement time-consuming logic. If you need to specify the business logic, use a new thread. Otherwise, message consumption may be blocked. Messages may be resent due to the latency of consumption.
                Console.WriteLine("receive message, topic=" + topic + ", messageId=" + messageId + ", body=" + body);
                // Send an ACK message.
                link.Accept(message);
            });
        }
        // If an exception occurs, the client attempts to re-connect to IoT Platform.
```

```
        // You can use an exponential backoff algorithm to improve the exception handling mechanism and the reconnection policy.
        static void ConnClosed(IAmqpObject _, Error e)
        {
            Console.WriteLine("ocurr error: " + e);
            if(Count < 3)
            {
                Count += 1;
                Thread.Sleep(IntervalTime * Count);
            }
            else
            {
                Thread.Sleep(120000);
            }
            // Re-establish the connection.
            DoConnectAmqp(address.User, address.Password);
        }
        static X509Certificate GetCert()
        {
            string certPath = Environment.CurrentDirectory + "/root.crt";
            X509Certificate crt = new X509Certificate(certPath);
            return crt;
        }
        static bool ValidateServerCertificate(object sender, X509Certificate certificate, X509Chain chain, SslPolicyErrors sslPolicyErrors)
        {
            return true;
        }
        static long GetCurrentMilliseconds()
        {
            DateTime dt1970 = new DateTime(1970, 1, 1);
            DateTime current = DateTime.Now;
            return (long)(current - dt1970).TotalMilliseconds;
        }
        // The signature algorithm. Valid values: hmacmd5, hmacsha1, and hmacsha256.
        static string doSign(string param, string accessSecret, string signMethod)
        {
            //signMethod = HmacMD5
            byte[] key = Encoding.UTF8.GetBytes(accessSecret);
            byte[] signContent = Encoding.UTF8.GetBytes(param);
            var hmac = new HMACMD5(key);
            byte[] hashBytes = hmac.ComputeHash(signContent);
            return Convert.ToBase64String(hashBytes);
        }
    }
}
```

You can configure the parameters in the preceding code based on the parameter descriptions in the following table. For more information, see Connect an AMQP client to IoT Platform.

Parameter	Example	Description
Host	198426864******.iot-amqp.cn-shanghai.aliyuncs.com	The AMQP endpoint. For more information about the endpoints that you can specify for the `${YourHost}` variable, see View the endpoint of an instance.
AccessKey	LTAI4GFGQvKuqHJhFa******	Log on to the IoT Platform console, move the pointer over the profile picture, and then click **AccessKey Management** to obtain the AccessKey ID and AccessKey secret. **Note** If you use a RAM user, you must attach the AliyunIOTFullAccess permission policy to the RAM user. This policy allows the RAM user to manage IoT Platform resources. Otherwise, the connection with IoT Platform fails. For more information about how to authorize a RAM user, see RAM user access.
AccessSecret	iMS8ZhCDdfJbCMeA005sieKe********	
consumerGroupId	VWhGZ2QnP7kxWpeSSjt******	The ID of the consumer group. To view the ID of the consumer group, perform the following steps: Log on to the IoT Platform console and click the card of the instance that you want to manange. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. The ID is displayed on the Consumer Groups tab.
iotInstanceId	""	The ID of the instance. You can view the ID of the instance on the **Overview** page in the IoT Platform console. • If you have an ID value, you must specify the ID for this parameter. • If no **Overview** or ID is generated for your instance, specify an empty string (`iotInstanceId = ""`) for the parameter.
clientId	12345	The ID of the client. We recommend that you use a unique identifier, such as the UUID, MAC address, or IP address of the client. The client ID must be 1 to 64 characters in length. Log on to the IoT Platform console and click the card of the instance that you want to manage. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. Find the consumer group that you want to manage and click **View** in the Actions column. The ID of each client is displayed on the **Consumer Group Details** page. You can use client IDs to efficiently identify clients.

Sample results

- If the output that is similar to the following log information appears, the AMQP client is connected to IoT Platform and can receive messages.

Parameter	Example	Description
topic	/**********/******/thing/event/property/post	The topic that is used to submit device properties.
messageId	1324198300680719360	The ID of the message.
body	null	The content of the message.

References

For more information about the error codes that are related to server-side subscription, see Message-related error codes.

3.3.5. Connect a client to IoT Platform by using the SDK for Python 2.7

This topic describes how to use the SDK for Python 2.7 to connect a client to IoT Platform and receive messages from IoT Platform.

Prerequisites

The ID of the consumer group that has subscribed to the messages of a topic is obtained.

- Manage consumer groups: You can use the default consumer group (DEFAULT_GROUP) or create a consumer group in the IoT Platform console.
- Configure an AMQP server-side subscription: You can use a consumer group to subscribe to the messages of a topic.

Prepare the development environment

In this example, Python 2.7 is used.

Download the SDK

We recommend that you use the Apache Qpid Proton 0.29.0 library. This library encapsulates the Python API. To download the library and view the instructions, see Qpid Proton 0.29.0.

Install Proton. For more information about how to install Proton, see Installing Qpid Proton.

After you install Proton, run the following Python command to check whether the SSL library is available:

```
import proton;print('%s' % 'SSL present' if proton.SSL.present() else 'SSL NOT AVAILABLE')
```

Sample code

```
# encoding=utf-8
import sys
import logging
import time
from proton.handlers import MessagingHandler
from proton.reactor import Container
import hashlib
import hmac
import base64
reload(sys)
sys.setdefaultencoding('utf-8')
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(
message)s')
logger = logging.getLogger(__name__)
console_handler = logging.StreamHandler(sys.stdout)
def current_time_millis():
    return str(int(round(time.time() * 1000)))
def do_sign(secret, sign_content):
    m = hmac.new(secret, sign_content, digestmod=hashlib.sha1)
    return base64.b64encode(m.digest())
class AmqpClient(MessagingHandler):
    def __init__(self):
        super(AmqpClient, self).__init__()
    def on_start(self, event):
        # The endpoint. For more information, see Connect an AMQP client to IoT Platform.
        url = "amqps://${YourHost}:5671"
        accessKey = "${YourAccessKeyID}"
        accessSecret = "${YourAccessKeySecret}"
        consumerGroupId = "${YourConsumerGroupId}"
        clientId = "${YourClientId}"
        IotInstanceId: the ID of the instance.
        iotInstanceId = "${YourIotInstanceId}"
        // The signature algorithm. Valid values: hmacmd5, hmacsha1, and hmacsha256.
        signMethod = "hmacsha1"
        timestamp = current_time_millis()
        // The structure of the userName parameter. For more information, see Connect an AM
QP client to IoT Platform.
        userName = clientId + "|authMode=aksign" + ",signMethod=" + signMethod \
                        + ",timestamp=" + timestamp + ",authId=" + accessKey \
                        + ",iotInstanceId=" + iotInstanceId + ",consumerGroupId=" + consume
rGroupId + "|"
        signContent = "authId=" + accessKey + "&timestamp=" + timestamp
        # The structure of the signature and password parameters. For more information, see
Connect an AMQP client to IoT Platform.
        passWord = do_sign(accessSecret.encode("utf-8"), signContent.encode("utf-8"))
        conn = event.container.connect(url, user=userName, password=passWord, heartbeat=60)
        self.receiver = event.container.create_receiver(conn)
    # If the connection is established, the following function is called:
    def on_connection_opened(self, event):
        logger.info("Connection established, remoteUrl: %s", event.connection.hostname)
    # If the connection is ended, the following function is called:
    def on_connection_closed(self, event):
```

```
        logger.info("Connection closed: %s", self)
    # If the remote server ends the connection due to an error, the following function is called:
    def on_connection_error(self, event):
        logger.info("Connection error")
    # If an AMQP connection error occurs, such as an authentication error or a socket error, the following function is called:
    def on_transport_error(self, event):
        if event.transport.condition:
            if event.transport.condition.info:
                logger.error("%s: %s: %s" % (
                    event.transport.condition.name, event.transport.condition.description,
                    event.transport.condition.info))
            else:
                logger.error("%s: %s" % (event.transport.condition.name, event.transport.condition.description))
        else:
            logging.error("Unspecified transport error")
    # If a message is received, the following function is called:
    def on_message(self, event):
        message = event.message
        content = message.body.decode('utf-8')
        topic = message.properties.get("topic")
        message_id = message.properties.get("messageId")
        print("receive message: message_id=%s, topic=%s, content=%s" % (message_id, topic, content))
        event.receiver.flow(1)
Container(AmqpClient()).run()
```

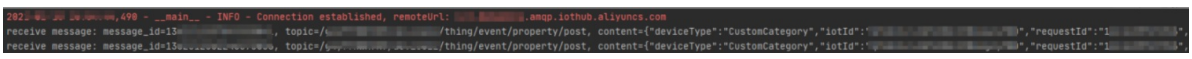
You can configure the parameters in the preceding code based on the parameter descriptions in the following table. For more information, see Connect an AMQP client to IoT Platform.

Parameter	Example	Description
url	amqps://198426864******.iot-amqp.cn-shanghai.aliyuncs.com:5671	The endpoint that the AMQP client uses to connect to IoT Platform. Format: `"amqps://${YourHost}:5671"`. For more information about the endpoints that you can specify for the `${YourHost}` variable, see View the endpoint of an instance.

Parameter	Example	Description
accessKey	LTAI4GFGQvKuqHJhFa******	Log on to the IoT Platform console, move the pointer over the profile picture, and then click **AccessKey Management** to obtain the AccessKey ID and AccessKey secret. **Note** If you use a RAM user, you must attach the AliyunIOTFullAccess permission policy to the RAM user. This policy allows the RAM user to manage IoT Platform resources. Otherwise, the connection with IoT Platform fails. For more information about how to authorize a RAM user, see RAM user access.
accessSecret	iMS8ZhCDdfJbCMeA005sieKe**********	
consumerGroupId	VWhGZ2QnP7kxWpeSSjt******	The ID of the consumer group. To view the ID of the consumer group, perform the following steps: Log on to the IoT Platform console and click the card of the instance that you want to manange. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. The ID is displayed on the Consumer Groups tab.
iotInstanceId	""	The ID of the instance. You can view the ID of the instance on the **Overview** page in the IoT Platform console. • If you have an ID value, you must specify the ID for this parameter. • If no **Overview** or ID is generated for your instance, specify an empty string (`iotInstanceId = ""`) for the parameter.
clientId	12345	The ID of the client. We recommend that you use a unique identifier, such as the UUID, MAC address, or IP address of the client. The client ID must be 1 to 64 characters in length. Log on to the IoT Platform console and click the card of the instance that you want to manage. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. Find the consumer group that you want to manage and click **View** in the Actions column. The ID of each client is displayed on the **Consumer Group Details** page. You can use client IDs to efficiently identify clients.

Sample results

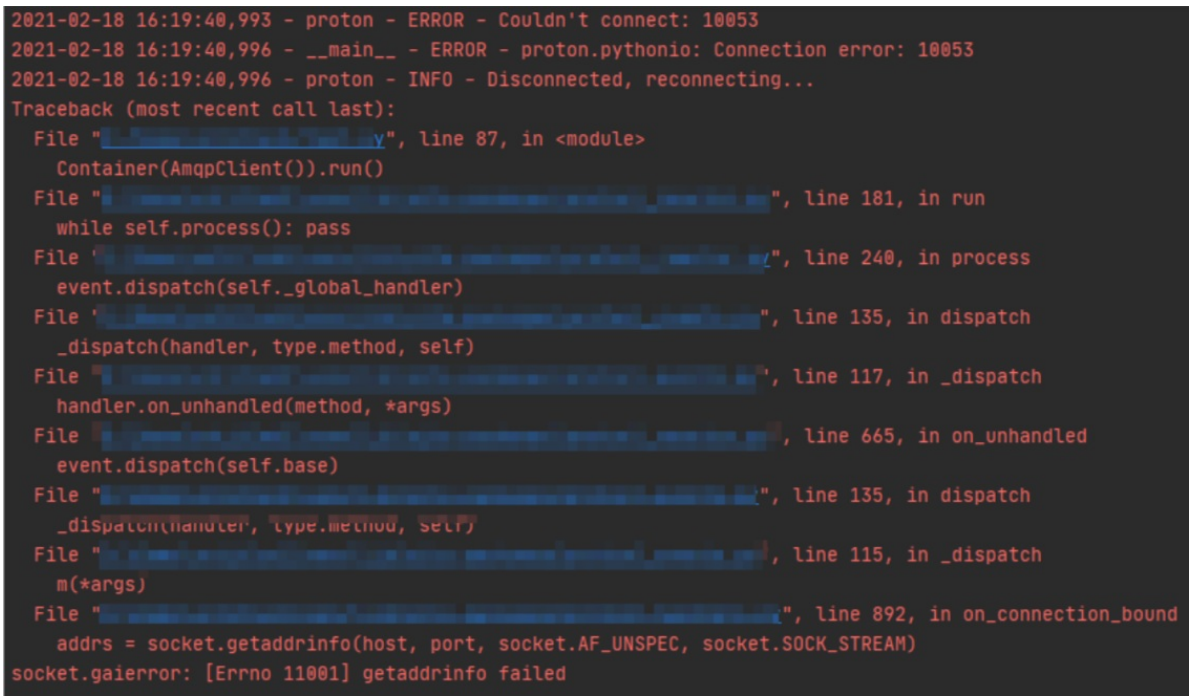
- If the output that is similar to the following log information appears, the AMQP client is connected to IoT Platform and can receive messages.



Parameter	Example	Description
message_id	1324198300680719360	The ID of the message.
topic	/**********/******/thing/event/property/post	The topic that is used to submit device properties.
content	null	The content of the message.

- If the output that is similar to the following log information appears, the AMQP client fails to connect to IoT Platform.

  You can check the code or network environment based on logs, solve the problem, and then run the code again.



References

For more information about the error codes that are related to server-side subscription, see Message-related error codes.

3.3.6. Connect a client to IoT Platform by using the SDK for Python 3

This topic describes how to use the SDK for Python 3 to connect a client to IoT Platform and receive messages from a server-side subscription.

Prerequisites

The ID of the consumer group that has subscribed to the messages of a topic is obtained.

- Manage consumer groups: You can use the default consumer group (DEFAULT_GROUP) or create a consumer group in the IoT Platform console.
- Configure an AMQP server-side subscription: You can use a consumer group to subscribe to the messages of a topic.

Prepare the development environment

You can use Python 3.0 or later. In this example, Python 3.8 is used.

Download the SDK

In this example, the stomp.py and schedule libraries are used. To download the libraries and view the instructions, see stomp.py and schedule.

Install the stomp.py and schedule libraries. For more information, see Installing Packages.

Sample code

This topic provides the sample code in which stomp.py 7.0.0 is used.

```
# encoding=utf-8
import time
import sys
import hashlib
import hmac
import base64
import stomp
import ssl
import schedule
import threading
def connect_and_subscribe(conn):
    accessKey = "${YourAccessKeyId}"
    accessSecret = "${YourAccessKeySecret}"
    consumerGroupId = "${YourConsumerGroupId}"
    IotInstanceId: the ID of the instance.
    iotInstanceId = "${YourIotInstanceId}"
    clientId = "${YourClientId}"
    // The signature algorithm. Valid values: hmacmd5, hmacsha1, and hmacsha256.
    signMethod = "hmacsha1"
    timestamp = current_time_millis()
    // The structure of the userName parameter. For more information, see Connect an AMQP c
lient to IoT Platform.
    // If you need to transmit messages in the binary format, specify encode=base64 in the
userName parameter. Before IoT Platform sends these messages, IoT Platform encodes these me
ssages by using the Base64 algorithm. For more information, see the "Messages in the binary
format" section of this topic.
    username = clientId + "|authMode=aksign" + ",signMethod=" + signMethod \
                    + ",timestamp=" + timestamp + ",authId=" + accessKey \
                    + ",iotInstanceId=" + iotInstanceId \
                    + ",consumerGroupId=" + consumerGroupId + "|"
    signContent = "authId=" + accessKey + "&timestamp=" + timestamp
    # The structure of the signature and password parameters. For more information, see Con
nect an AMQP client to IoT Platform.
    password = do_sign(accessSecret.encode("utf-8"), signContent.encode("utf-8"))
    conn.set_listener('', MyListener(conn))
    conn.connect(username, password, wait=True)
```

```
    conn.connect(username, password, wait=True)
    # Clear historical tasks that are used to check connections and creates tasks to check
connections.
    schedule.clear('conn-check')
    schedule.every(1).seconds.do(do_check,conn).tag('conn-check')
class MyListener(stomp.ConnectionListener):
    def __init__(self, conn):
        self.conn = conn
    def on_error(self, frame):
        print('received an error "%s"' % frame.body)
    def on_message(self, frame):
        print('received a message "%s"' % frame.body)
    def on_heartbeat_timeout(self):
        print('on_heartbeat_timeout')
    def on_connected(self, headers):
        print("successfully connected")
        conn.subscribe(destination='/topic/#', id=1, ack='auto')
        print("successfully subscribe")
    def on_disconnected(self):
        print('disconnected')
        connect_and_subscribe(self.conn)
def current_time_millis():
    return str(int(round(time.time() * 1000)))
def do_sign(secret, sign_content):
    m = hmac.new(secret, sign_content, digestmod=hashlib.sha1)
    return base64.b64encode(m.digest()).decode("utf-8")
# Check the connection. If the connection fails to be established, this method re-establish
es a connection.
def do_check(conn):
    print('check connection, is_connected: %s', conn.is_connected())
    if (not conn.is_connected()):
        try:
            connect_and_subscribe(conn)
        except Exception as e:
            print('disconnected, ', e)
# The method that is used to schedule tasks. Check the connection status.
def connection_check_timer():
    while 1:
        schedule.run_pending()
        time.sleep(10)
// The endpoint. For more information, see Connect an AMQP client to IoT Platform. Do not p
refix the endpoint with amqps://.
conn = stomp.Connection([('${YourHost}', 61614)], heartbeats=(0,300))
conn.set_ssl(for_hosts=[('${YourHost}', 61614)], ssl_version=ssl.PROTOCOL_TLS)
try:
    connect_and_subscribe(conn)
except Exception as e:
    print('connecting failed')
    raise e
# If a scheduled task runs in an asynchronous thread, check the connection status of the sc
heduled task.
thread = threading.Thread(target=connection_check_timer)
thread.start()
```

You can configure the parameters in the preceding code based on the parameter descriptions in the following table. For more information about the parameters, see Connect an AMQP client to IoT Platform.

Parameter	Example	Description
accessKey	LTAI4GFGQvKuqHJhFa******	Log on to the IoT Platform console, move the pointer over the profile picture, and then click **AccessKey Management** to obtain the AccessKey ID and AccessKey secret. **Note** If you use a RAM user, you must attach the AliyunIOTFullAccess permission policy to the RAM user. This policy allows the RAM user to manage IoT Platform resources. Otherwise, the connection with IoT Platform fails. For more information about how to authorize a RAM user, see RAM user access.
accessSecret	iMS8ZhCDdfJbCMeA005sieKe********	
consumerGroupId	VWhGZ2QnP7kxWpeSSjt******	The ID of the consumer group. To view the ID of the consumer group, perform the following steps: Log on to the IoT Platform console and click the card of the instance that you want to manange. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. The ID is displayed on the Consumer Groups tab.
iotInstanceId	""	The ID of the instance. You can view the ID of the instance on the **Overview** page in the IoT Platform console. • If you have an ID value, you must specify the ID for this parameter. • If no **Overview** or ID is generated for your instance, specify an empty string (`iotInstanceId = ""`) for the parameter.
clientId	12345	The ID of the client. We recommend that you use a unique identifier, such as the UUID, MAC address, or IP address of the client. The client ID must be 1 to 64 characters in length. Log on to the IoT Platform console and click the card of the instance that you want to manage. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. Find the consumer group that you want to manage and click **View** in the Actions column. The ID of each client is displayed on the **Consumer Group Details** page. You can use client IDs to efficiently identify clients.

Parameter	Example	Description
conn	stomp.Connection([('198426864******.iot-amqp.cn-shanghai.aliyuncs.com', 61614)])	The Transport Layer Security (TLS) connection that is established between the AMQP client and IoT Platform. For more information about the endpoints that you can specify for the `${YourHost}` variable, see View the endpoint of an instance.
conn.set_ssl	for_hosts=[('198426864******.iot-amqp.cn-shanghai.aliyuncs.com', 61614)], ssl_version=ssl.PROTOCOL_TLS	

Sample results

- Sample success result:

```
successfully connected
successfully subscribe
check connection, is_connected: %s True
check connection, is_connected: %s True
```

- Sample failure result:

  You can check the code or network environment based on logs, solve the problem, and then run the code again.

Messages in the binary format

If you want to transmit messages in the binary format, use the Base64 algorithm to encode the messages. If you do not use the Base64 algorithm to encode the messages, the messages may be truncated because STOMP is a text-based protocol.

The following code shows how to specify encode=base64 in the userName parameter. This setting enables IoT Platform to encode messages by using the Base64 algorithm before IoT Platform send the messages.

```
username = clientId + "|authMode=aksign" + ",signMethod=" + signMethod \
                    + ",timestamp=" + timestamp + ",authId=" + accessKey \
                    + ",iotInstanceId=" + iotInstanceId \
                    + ",consumerGroupId=" + consumerGroupId \
                    + ",encode=base64"+"|"
```

References

For more information about the error codes that are related to server-side subscription, see Message-related error codes.

3.3.7. Connect a client to IoT Platform by using the SDK for Node.js

This topic describes how to use Advanced Message Queuing Protocol (AMQP) SDK for Node.js to connect a client to IoT Platform and receive messages from IoT Platform.

Prerequisites

The ID of the consumer group that has subscribed to the messages of a topic is obtained.

- Manage consumer groups: You can use the default consumer group (DEFAULT_GROUP) or create a consumer group in the IoT Platform console.
- Configure an AMQP server-side subscription: You can use a consumer group to subscribe to the messages of a topic.

Prepare the development environment

You can use Node.js 8.0.0 or later.

Download and install the SDK

We recommend that you use the rhea library. To download the library and view the instructions, see rhea.

In this example, run the `npm install rhea` command to download the rhea library.

Sample code

1. Download and install Node.js on the Windows or Linux operating system. In this example, Windows 10 (64-bit) is used. Download the node-v14.15.1-x64.msi installation package.
2. Open a Command Prompt window and run the following command to view the node version:

   `node --version`

   If the package is decompressed, the following version number appears:

   ```
   v14.15.1
   ```

3. Create a JavaScript file such as *amqp.js* on your PC to store the Node.js sample code.

   When you use the sample code, configure the parameters based on the parameter descriptions in the following table.

```
const container = require('rhea');
const crypto = require('crypto');
// Create a connection.
var connection = container.connect({
    // The endpoint. For more information, see Connect an AMQP client to IoT Platform.
    'host': '${YourHost}',
    'port': 5671,
    'transport':'tls',
    'reconnect':true,
    'idle_time_out':60000,
    // The structure of the userName parameter. For more information, see Connect an AMQP client to IoT Platform.
    'username':'${YourClientId}|authMode=aksign,signMethod=hmacsha1,timestamp=1573489088171,authId=${YourAccessKeyId},iotInstanceId=${YourIotInstanceId},consumerGroupId=${YourConsumerGroupId}|',
    // Calculate a signature. For more information about how to configure the password parameter, see Connect an AMQP client to IoT Platform.
    'password': hmacSha1('${YourAccessKeySecret}', 'authId=${YourAccessKeyId}&timestamp=1573489088171'),
});
// Create a receiver link.
var receiver = connection.open_receiver();
// The callback function that is called when messages are received from IoT Platform.
container.on('message', function (context) {
    var msg = context.message;
    var messageId = msg.message_id;
    var topic = msg.application_properties.topic;
    var content = Buffer.from(msg.body.content).toString();
    // Return message content.
    console.log(content);
    // Send an ACK packet. Do not implement time-consuming logic in the callback function.
    context.delivery.accept();
});
// Calculate the password signature.
function hmacSha1(key, context) {
    return Buffer.from(crypto.createHmac('sha1', key).update(context).digest())
        .toString('base64');
}
```

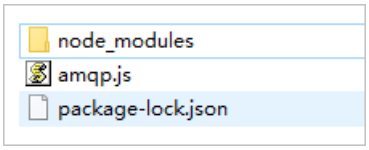
Parameter	Example	Description
host	198426864******.iot-amqp.cn-shanghai.aliyuncs.com	The AMQP endpoint. For more information about the endpoints that you can specify for the `${YourHost}` variable, see View the endpoint of an instance.
username	'test|authMode=aksign,signMethod=hmacsha1,timestamp=1573489088171,authId=LTAI4***QvKu****,iotInstanceId=,c	The authentication information that is required to connect to IoT Platform. Description: o `${YourClientId}`: Replace this variable with your client ID. You can specify a client ID as

Parameter	Example	Description
password	hmacSha1('iMS8ZhCDd****', 'authId=LTAI4GFGQ****&timestamp=1573489088171')	onsumerGroupId=DEFAULT_GROUP|" needed. The client ID cannot exceed 64 characters in length. We recommend that you use a unique identifier, such as the UUID, MAC address, or IP address of the client. Log on to the IoT Platform console and click the card of the instance that you want to manage. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. Find the consumer group that you want to manage and click **View** in the Actions column. The ID of each client is displayed on the **Consumer Group Details** page. You can use client IDs to efficiently identify clients. ○ `${YourAccessKeyId}` and `${YourAccessKeySecret}`: Replace the variables with your AccessKey ID and AccessKey secret. Log on to the IoT Platform console, move the pointer over the profile picture, and then click **AccessKey Management** to obtain the AccessKey ID and AccessKey secret. **Note** If you use a RAM user, you must attach the AliyunIOTFullAccess permission policy to the RAM user. This policy allows the RAM user to manage IoT Platform resources. Otherwise, the connection with IoT Platform fails. For more information about how to authorize a RAM user, see RAM user access. ○ `${YourIotInstanceId}`: Replace this variable with your instance ID. Log on to the IoT Platform console. On the **Overview** page, view the ID of the instance. **Notice** If no **Overview** page or ID is generated for your instance, delete ${YourIotInstanceId}. ○ `${YourConsumerGroupId}`: Replace this variable with your consumer group ID. To view the ID of the consumer group, perform the following steps: Log on to the IoT Platform console and click the card of the instance that you want to manange. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. The ID is displayed on the Consumer Groups tab.

4. Open the Command Prompt and run the cd command to go to the directory where the *amqp.js* file is stored. Then, run the npm command to download the rhea library. The following figure shows

the downloaded file.

```
npm install rhea
```



5. Run the following command in the Command Prompt to run the *amqp.js* file. This way, the AMQP client is started.

```
node amqp.js
```

Sample results

- Sample success result:

- Sample failure result:

  You can check the code or network environment based on logs, solve the problem, and then run the code again.

References

For more information about the error codes that are related to server-side subscription, see Message-related error codes.

3.3.8. Connect a client to IoT Platform by using the SDK for Go

This topic describes how to connect a Go client to IoT Platform over Advanced Message Queuing Protocol (AMQP) and enable the client to receive messages from IoT Platform.

Prerequisites

The ID of the consumer group that has subscribed to the messages of a topic is obtained.

- Manage consumer groups: You can use the default consumer group (DEFAULT_GROUP) or create a consumer group in the IoT Platform console.
- Configure an AMQP server-side subscription: You can use a consumer group to subscribe to the messages of a topic.

Prepare the development environment

In this example, Go 1.12.7 is used.

Download the SDK

You can run the following command to import AMQP SDK for Go:

```
import "pack.ag/amqp"
```

For more information about how to use the SDK, see package amqp.

Sample code

```
package main
import (
    "context"
    "crypto/hmac"
    "crypto/sha1"
    "encoding/base64"
    "fmt"
    "pack.ag/amqp"
    "time"
)
// The parameters. For more information, see Connect an AMQP client to IoT Platform.
const accessKey = "${YourAccessKey}"
const accessSecret = "${YourAccessSecret}"
const consumerGroupId = "${YourConsumerGroupId}"
const clientId = "${YourClientId}"
// iotInstanceId: the ID of the instance.
const iotInstanceId = "${YourIotInstanceId}"
// The endpoint. For more information, see Connect an AMQP client to IoT Platform.
const host = "${YourHost}"
func main() {
    address := "amqps://" + host + ":5671"
    timestamp := time.Now().Nanosecond() / 1000000
    // The structure of the userName parameter. For more information, see Connect an AMQP c
lient to IoT Platform.
    userName := fmt.Sprintf("%s|authMode=aksign,signMethod=Hmacsha1,consumerGroupId=%s,auth
Id=%s,iotInstanceId=%s,timestamp=%d|",
        clientId, consumerGroupId, accessKey, iotInstanceId, timestamp)
    stringToSign := fmt.Sprintf("authId=%s&timestamp=%d", accessKey, timestamp)
    hmacKey := hmac.New(sha1.New, []byte(accessSecret))
    hmacKey.Write([]byte(stringToSign))
    // Calculate a signature. For more information about how to configure the password para
meter, see Connect an AMQP client to IoT Platform.
    password := base64.StdEncoding.EncodeToString(hmacKey.Sum(nil))
    amqpManager := &AmqpManager{
        address:address,
        userName:userName,
        password:password,
    }
    // If you need to enable or disable the message receiving feature, you can create a con
text by using the Background() function.
    ctx := context.Background()
    amqpManager.startReceiveMessage(ctx)
}
// The function that implements your business logic. The function is a user-defined functio
```

```
n that is asynchronously implemented. Before you configure this function, we recommend that
you consider the consumption of system resources.
func (am *AmqpManager) processMessage(message *amqp.Message) {
    fmt.Println("data received:", string(message.GetData()), " properties:", message.Applic
ationProperties)
}
type AmqpManager struct {
    address     string
    userName     string
    password     string
    client       *amqp.Client
    session     *amqp.Session
    receiver     *amqp.Receiver
}
func (am *AmqpManager) startReceiveMessage(ctx context.Context)  {
    childCtx, _ := context.WithCancel(ctx)
    err := am.generateReceiverWithRetry(childCtx)
    if nil != err {
        return
    }
    defer func() {
        am.receiver.Close(childCtx)
        am.session.Close(childCtx)
        am.client.Close()
    }()
    for {
        // Block message receiving. If ctx is the new context that is created based on the
Background() function, message receiving is not blocked.
        message, err := am.receiver.Receive(ctx)
        if nil == err {
            go am.processMessage(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 := am.generateReceiverWithRetry(childCtx)
            if nil != err {
                return
            }
        }
    }
}
func (am *AmqpManager) generateReceiverWithRetry(ctx context.Context) error {
    // Retry with exponential backoff, from 10 ms to 20s.
    duration := 10 * time.Millisecond
    maxDuration := 20000 * time.Millisecond
    times := 1
    // If exceptions occur, retry with exponential backoff.
    for {
```

```
        select {
        case <- ctx.Done(): return amqp.ErrConnClosed
        default:
        }
        err := am.generateReceiver()
        if nil != err {
            time.Sleep(duration)
            if duration < maxDuration {
                duration *= 2
            }
            fmt.Println("amqp connect retry,times:", times, ",duration:", duration)
            times ++
        } 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 (am *AmqpManager) generateReceiver() error {
    if am.session != nil {
        receiver, err := am.session.NewReceiver(
            amqp.LinkSourceAddress("/queue-name"),
            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 {
            am.receiver = receiver
            return nil
        }
    }
    // Delete the previous connection.
    if am.client != nil {
        am.client.Close()
    }
    client, err := amqp.Dial(am.address, amqp.ConnSASLPlain(am.userName, am.password), )
    if err != nil {
        return err
    }
    am.client = client
    session, err := client.NewSession()
    if err != nil {
        return err
    }
    am.session = session
    receiver, err := am.session.NewReceiver(
        amqp.LinkSourceAddress("/queue-name"),
        amqp.LinkCredit(20),
    )
    if err != nil {
        return err
    }
```

```
    am.receiver = receiver
    return nil
}
```

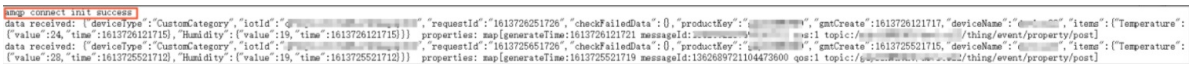
You can configure the parameters in the preceding code based on the parameter descriptions in the following table. For more information, see Connect an AMQP client to IoT Platform.

Parameter	Example	Description
accessKey	LTAI4GFGQvKuqHJhFa******	Log on to the IoT Platform console, move the pointer over the profile picture, and then click **AccessKey Management** to obtain the AccessKey ID and AccessKey secret. **Note** If you use a RAM user, you must attach the AliyunIOTFullAccess permission policy to the RAM user. This policy allows the RAM user to manage IoT Platform resources. Otherwise, the connection with IoT Platform fails. For more information about how to authorize a RAM user, see RAM user access.
accessSecret	iMS8ZhCDdfJbCMeA005sieKe**********	
consumerGroupId	VWhGZ2QnP7kxWpeSSjt******	The ID of the consumer group. To view the ID of the consumer group, perform the following steps: Log on to the IoT Platform console and click the card of the instance that you want to manange. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. The ID is displayed on the Consumer Groups tab.
iotInstanceId	""	The ID of the instance. You can view the ID of the instance on the **Overview** page in the IoT Platform console. • If you have an ID value, you must specify the ID for this parameter. • If no **Overview** or ID is generated for your instance, specify an empty string (`iotInstanceId = ""`) for the parameter.

Parameter	Example	Description
clientId	12345	The ID of the client. We recommend that you use a unique identifier, such as the UUID, MAC address, or IP address of the client. The client ID must be 1 to 64 characters in length. Log on to the IoT Platform console and click the card of the instance that you want to manage. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. Find the consumer group that you want to manage and click **View** in the Actions column. The ID of each client is displayed on the **Consumer Group Details** page. You can use client IDs to efficiently identify clients.
host	198426864******.iot-amqp.cn-shanghai.aliyuncs.com	The AMQP endpoint. For more information about the endpoints that you can specify for the `${YourHost}` variable, see View the endpoint of an instance.

Sample results

- Sample success result:



- Sample failure result:

  You can check the code or network environment based on logs, solve the problem, and then run the code again.

```
amqp connect retry, times: 1 , duration: 20ms
amqp connect retry, times: 2 , duration: 40ms
amqp connect retry, times: 3 , duration: 80ms
amqp connect retry, times: 4 , duration: 160ms
amqp connect retry, times: 5 , duration: 320ms
amqp connect retry, times: 6 , duration: 640ms
amqp connect retry, times: 7 , duration: 1.28s
amqp connect retry, times: 8 , duration: 2.56s
amqp connect retry, times: 9 , duration: 5.12s
amqp connect retry, times: 10 , duration: 10.24s
amqp connect retry, times: 11 , duration: 20.48s
```

References

For more information about the error codes that are related to server-side subscription, see Message-related error codes.

3.3.9. Connect a client to IoT Platform by using the SDK for PHP

This topic describes how to use the SDK for PHP to connect a client to IoT Platform and receive messages from IoT Platform.

Prerequisites

The ID of the consumer group that has subscribed to the messages of a topic is obtained.

- Manage consumer groups: You can use the default consumer group (DEFAULT_GROUP) or create a consumer group in the IoT Platform console.
- Configure an AMQP server-side subscription: You can use a consumer group to subscribe to the messages of a topic.

Download the SDK

The sample code is written based on the Stomp PHP library and allows you to connect a Stomp PHP client to IoT Platform over Simple Text Oriented Message Protocol (STOMP). To download a Stomp PHP client and view the instructions, see Stomp PHP.

Stomp PHP 5.0.0 or earlier may fail to reconnect to IoT Platform after the SDK is disconnected. We recommend that you download Stomp PHP 5.0.0 or later. For more information, see Issues.

In the PHP project directory, run the following command to download the SDK of Stomp PHP 5.0.0:

```
composer require stomp-php/stomp-php 5.0.0
```

Sample code

```
<?php
require __DIR__ . '/vendor/autoload.php';
use Stomp\Client;
use Stomp\Network\Observer\Exception\HeartbeatException;
use Stomp\Network\Observer\ServerAliveObserver;
use Stomp\StatefulStomp;
// The parameters. For more information, see the "Connect an AMQP client to IoT Platform" t
opic.
$accessKey = "${YourAccessKeyId}";
$accessSecret = "${YourAccessKeySecret}";
$consumerGroupId = "${YourConsumerGroupId}";
$clientId = "${YourClientId}";
// iotInstanceId: the ID of the instance.
$iotInstanceId = "${YourIotInstanceId}";
$timeStamp = round(microtime(true) * 1000);
// The signature algorithm. Valid values: hmacmd5, hmacsha1, and hmacsha256.
$signMethod = "hmacsha1";
// The structure of the userName parameter. For more information, see the "Connect an AMQP
client to IoT Platform" topic.
// If you want to transmit messages in the binary format, specify encode=base64 in the user
Name parameter. Before IoT Platform sends these messages, IoT Platform encodes these messag
es by using the Base64 algorithm. For more information, see the "Messages in the binary for
mat" section.
```

```
$userName = $clientId . "|authMode=aksign"
            . ",signMethod=" . $signMethod
            . ",timestamp=" . $timeStamp
            . ",authId=" . $accessKey
            . ",iotInstanceId=" . $iotInstanceId
            . ",consumerGroupId=" . $consumerGroupId
            . "|";
$signContent = "authId=" . $accessKey . "&timestamp=" . $timeStamp;
// Calculate a signature. For more information about how to specify the password parameter,
see the "Connect an AMQP client to IoT Platform" topic.
$password = base64_encode(hash_hmac("sha1", $signContent, $accessSecret, $raw_output = TRUE
));
// The endpoint. For more information, see the "Connect an AMQP client to IoT Platform" top
ic.
$client = new Client('ssl://${YourHost}:61614');
$sslContext = ['ssl' => ['verify_peer' => true, 'verify_peer_name' => false], ];
$client->getConnection()->setContext($sslContext);
// Configure a listener to monitor the status of the connection between the client and IoT
Platform.
$observer = new ServerAliveObserver();
$client->getConnection()->getObservers()->addObserver($observer);
// The heartbeat setting. This setting enables IoT Platform to send a heartbeat packet ever
y 30 seconds.
$client->setHeartbeat(0, 30000);
$client->setLogin($userName, $password);
try {
    $client->connect();
}
catch(StompException $e) {
    echo "failed to connect to server, msg:" . $e->getMessage() , PHP_EOL;
}
// Run the following code if no exceptions occur:
$stomp = new StatefulStomp($client);
$stomp->subscribe('/topic/#');
echo "connect success";
while (true) {
    try {
        // Check the connection status.
        if (!$client->isConnected()) {
            echo "connection not exists, will reconnect after 10s.", PHP_EOL;
            sleep(10);
            $client->connect();
            $stomp->subscribe('/topic/#');
            echo "connect success", PHP_EOL;
        }
        // Specify the business logic to process messages.
        echo $stomp->read();
    }
    catch(HeartbeatException $e) {
        echo 'The server failed to send us heartbeats within the defined interval.', PHP_EO
L;
        $stomp->getClient()->disconnect();
    } catch(Exception $e) {
        echo 'process message occurs error: '. $e->getMessage() , PHP_EOL;
```

```
        $stomp->getClient()->disconnect();
    }
}
```

You can configure the parameters in the preceding code based on the parameter descriptions in the following table. For more information about the parameters, see Connect an AMQP client to IoT Platform.

Parameter	Example	Description
accessKey	LTAI4GFGQvKuqHJhFa******	Log on to the IoT Platform console, move the pointer over the profile picture, and then click **AccessKey Management** to obtain the AccessKey ID and AccessKey secret. **Note** If you use a RAM user, you must attach the AliyunIOTFullAccess permission policy to the RAM user. This policy allows the RAM user to manage IoT Platform resources. Otherwise, the connection with IoT Platform fails. For more information about how to authorize a RAM user, see RAM user access.
accessSecret	iMS8ZhCDdfJbCMeA005sieKe********	
consumerGroupId	VWhGZ2QnP7kxWpeSSjt******	The ID of the consumer group. To view the ID of the consumer group, perform the following steps: Log on to the IoT Platform console and click the card of the instance that you want to manange. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. The ID is displayed on the Consumer Groups tab.
iotInstanceId	""	The ID of the instance. You can view the ID of the instance on the **Overview** page in the IoT Platform console. • If you have an ID value, you must specify the ID for this parameter. • If no **Overview** or ID is generated for your instance, specify an empty string (`iotInstanceId = ""`) for the parameter.

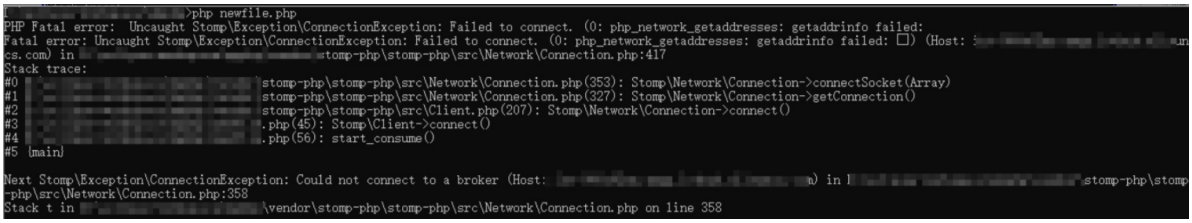
Parameter	Example	Description
clientId	12345	The ID of the client. We recommend that you use a unique identifier, such as the UUID, MAC address, or IP address of the client. The client ID must be 1 to 64 characters in length. Log on to the IoT Platform console and click the card of the instance that you want to manage. Choose **Rules Engine > Server-side Subscription > Consumer Groups**. Find the consumer group that you want to manage and click **View** in the Actions column. The ID of each client is displayed on the **Consumer Group Details** page. You can use client IDs to efficiently identify clients.
client	new Client('ssl://198426864******.iot-amqp.cn-shanghai.aliyuncs.com:61614')	Establish a connection between the AMQP Client and IoT Platform. Format: `$client = new Client('ssl://${YourHost}:61614');` For more information about the endpoints that you can specify for the `${YourHost}` variable, see View the endpoint of an instance.

Sample results

- Sample success result:

- Sample failure result:

  You can check the code or network environment based on logs, solve the problem, and then run the code again.



Messages in the binary format

If you want to transmit messages in the binary format, use the Base64 algorithm to encode the messages. If you do not use the Base64 algorithm to encode the messages, the messages may be truncated because STOMP is a text-based protocol.

The following code shows how to specify encode=base64 in the userName parameter. This setting enables IoT Platform to encode messages by using the Base64 algorithm before IoT Platform send the messages.

```
$userName = $clientId . "|authMode=aksign"
                . ",signMethod=" . $signMethod
                . ",timestamp=" . $timeStamp
                . ",authId=" . $accessKey
                . ",iotInstanceId=" . $iotInstanceId
                . ",consumerGroupId=" . $consumerGroupId
                . ",encode=base64" . "|";
```

References

For more information about the error codes that are related to server-side subscription, see Message-related error codes.

3.4. Configure MNS server-side subscriptions

IoT Platform allows you to send device messages to Message Service (MNS). Cloud applications can obtain the device messages by listening to MNS queues. This article describes how to configure an MNS server-side subscription.

Prerequisites

If you use a RAM user, the RAM user must have the `AliyunIOTAccessingMNSRole` permission.

Procedure

1. In the IoT Platform console, configure an MNS server-side subscription for a product. IoT Platform automatically forwards messages to MNS queues.
   i. Log on to the IoT Platform console.
   ii.
   iii. In the left-side navigation pane, choose **Rules > Server-side Subscription**.
   iv. On the **Subscriptions** tab of the **Server-side Subscription** page, click **Create Subscription**.
   v. In the **Create Subscription** dialog box, set the following parameters and click **OK**.

Parameter	Description
Products	Select the product to which the devices belong. The messages submitted by these devices are pushed to consumers.
Subscription Type	Select **MNS**.

Parameter	Description
Message Type	Select the types of messages. You can subscribe to the following types of device messages: ■ **Device Upstream Notification**: the messages in the topics whose **Allowed Operations** parameter is set to **Publish**. This type of messages includes custom data and Thing Specification Language (TSL) data that is submitted by devices. The upstream TSL data includes property data, event data, responses to property setting requests, and responses to service calling requests. The TSL data that is pushed to user servers is processed by IoT Platform. For more information about data formats, see Data formats. For example, the following three topics are defined for a product: ■ `/${YourProductKey}/${YourDeviceName}/user/get`. The Allowed Operations parameter of this topic is set to Subscribe. ■ `/${YourProductKey}/${YourDeviceName}/user/update`. The Allowed Operations parameter of this topic is set to Publish. ■ `/sys/${YourProductKey}/${YourDeviceName}/thing/event/property/post`. The Allowed Operations parameter of this topic is set to Subscribe. IoT Platform pushes messages in the following topics: `/${YourProductKey}/${YourDeviceName}/user/update` and `/sys/${YourProductKey}/${YourDeviceName}/thing/event/property/post`. ■ **Device Status Change Notification**: the notifications that devices send when the online or offline status changes. ■ **Gateway's sub-devices discovery report**: the sub-device data that gateways submit when these gateways detect new sub-devices. The gateways must have the applications that can be used to detect sub-devices. This message type is specific to gateways. ■ **Device Topological Relation Changes**: the notifications that gateways send when topological relationships between sub-devices and the gateways are created or deleted. This message type is specific to gateways. ■ **Device Changes Throughout Lifecycle**: the notifications that devices send when the devices are created, deleted, enabled, or disabled. ■ **TSL Historical Data Reporting**: The historical properties and events that are submitted by devices. ■ **OTA Update Status Notification**: the notifications that devices send during firmware verification and batch update. When a device update succeeds or fails, a notification is pushed.

vi. In the message that appears, click **Confirm**.

IoT Platform automatically creates an MNS message queue in the format of `aliyun-iot-${yourproductkey}`. When you configure a queue listener, you must specify this message queue.

You are charged for using MNS resources. For more information about billing methods, see MNS billing.

> **Note** If you delete the MNS server-side subscription, the related MNS queue is automatically deleted.

2. Configure an MNS client and listen to the MNS queue.

In this example, the MNS SDK for Java is used to listen to the MNS queue.

For more information about how to download the MNS SDK, see MNS documentation.

i. To install the MNS SDK for Java, add the following dependencies to the pom.xml file:

```
<dependency>
    <groupId>com.aliyun.mns</groupId>
    <artifactId>aliyun-sdk-mns</artifactId>
    <version>1.1.8</version>
    <classifier>jar-with-dependencies</classifier>
</dependency>
```

ii. Specify the following parameters when you configure the MNS SDK:

```
CloudAccount account = new CloudAccount( $AccessKeyId, $AccessKeySecret, $AccountEndpoint);
```

- Replace $AccessKeyId and $AccessKeySecret with the AccessKey ID and AccessKey secret of your Alibaba Cloud account. These parameters are required when you call API operations. To create or view an AccessKey pair, log on to the IoT Platform console, move the pointer over your profile picture, and then click **AccessKey Management**.
- Replace $AccountEndpoint with the MNS endpoint. In the MNS console, click **Get Endpoint**.

iii. Specify the logic to receive device messages.

```
MNSClient client = account.getMNSClient();
CloudQueue queue = client.getQueueRef("aliyun-iot-a1wmrZPO8o9"); //Specify the automatically created queue.
    while (true) {
    //Retrieve messages.
    Message popMsg = queue.popMessage(10);  //The timeout period of long polling requests is 10 seconds.
    if (popMsg ! = null) {
        System.out.println("PopMessage Body: "+ popMsg.getMessageBodyAsRawString()); //Obtain raw messages.
        queue.deleteMessage(popMsg.getReceiptHandle()); //Delete the messages from the queue.
    } else {
        System.out.println("Continuing"); } }
```

iv. Run the program to listen to the MNS queue.

3. Start a device and send a message from the device to IoT Platform.

   For more information about how to develop a device-side SDK, see the Link SDK documentation.

4. Check whether your cloud application retrieves the message. The following code shows the format of a retrieved message:

```
{
"messageid":" ",
"messagetype":"upload",
"topic":"/al12345****/device123/user/update",
"payload":" ",
"timestamp": " "
}
```

Parameter	Description
messageid	The ID of the message. The message ID is generated by IoT Platform.
messagetype	The type of the message. Valid values: ○ upload: submitted device data ○ status: device status changes ○ topo_listfound: the detection of sub-devices by a gateway ○ topo_lifecycle: device topology changes ○ device_lifecycle: device lifecycle changes ○ thing_history: historical TSL data ○ ota_event: firmware update status
topic	The IoT Platform topic from which the message is forwarded.
payload	The base64-encoded message payload. For more information about data formats, see Data formats.
timestamp	The timestamp. It is the number of seconds that have elapsed since the epoch time January 1, 1970, 00:00:00 UTC.

Related information

- Data formats
-

4.Data forwarding

4.1. Overview

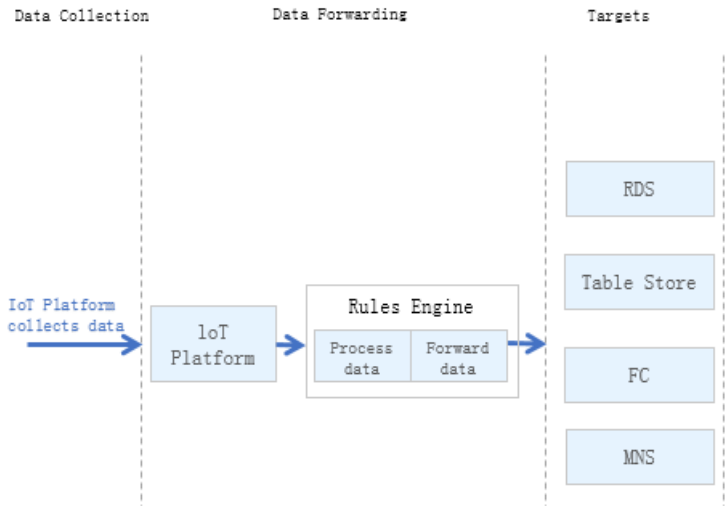
The data forwarding feature of IoT Platform allows you to forward data from a topic to other topics or Alibaba Cloud services for storage or processing.

What is data forwarding?

If your devices use topics to communicate with other devices and IoT Platform, you can write SQL statements to process the data of the topics. You can also configure data forwarding rules to send the processed data to other topics or Alibaba Cloud services.

- You can forward data to the topics of another device to implement machine-to-machine (M2M) communication.
- You can forward data to consumer groups that subscribe to device messages. Your server listens to the messages of consumer groups by using an Advanced Message Queuing Protocol (AMQP) client. For more information, see Connect an AMQP client to IoT Platform.
- You can forward data to ApsaraDB RDS and Tablestore for data storage.
- You can forward data to Function Compute for event-driven computing.
- You can forward data to Message Service (MNS) to consume data in a highly reliable manner.

The data forwarding feature allows you to efficiently collect, compute, and store data. You do not need to purchase servers to deploy a distributed architecture.



Data forwarding usage guide

- Configure a data forwarding rule: describes how to configure a data forwarding rule.
- SQL statements: describes the SQL statements that can be used in a data forwarding rule.
- Functions: describes the functions that can be used in SQL statements.
- Data forwarding procedure: describes the data forwarding process and the data formats at different stages of data forwarding.
- Data formats: describes the data formats of basic communication topics and TSL-based communication topics. The data must be parsed by a TSL model. You must write the SQL statements of the data forwarding rules in the data format that is parsed by a TSL model.

- Regions and zones: describes the Alibaba Cloud services that are supported in different regions and zones.

References

Data forwarding (new version)

4.2. Compare data forwarding solutions

In multiple scenarios, you must process the data that is reported by devices to IoT Platform or use the data for business applications. IoT Platform allows you to forward device data by using one of the following solutions: server-side subscription or data forwarding. This article compares the application scenarios, advantages, and disadvantages of the solutions. You can select a solution based on your business scenarios.

IoT Platform supports the following two data forwarding solutions:

- Data forwarding: provides basic data filtering and processing capabilities. You can configure data forwarding rules to filter and process device data and then forward the data to other Alibaba Cloud services.
- Server-side subscription: allows you to directly obtain device messages by using an Advanced Message Queuing Protocol (AMQP) or Message Service (MNS) client. You can obtain device messages without the need to filter and process the messages. This feature is simple, easy to use, and efficient.

Comparison between data forwarding and server-side subscription

IoT Platform supports the following two data forwarding solutions:

- Data forwarding: You can configure data forwarding rules to filter and process device data and then forward the data to other Alibaba Cloud services.
- Server-side subscription: You can directly obtain device messages by using an AMQP or MNS client.

The following table compares the application scenarios, advantages, and disadvantages of the two solutions.

Solution	Application Scenario	Advantage and disadvantage	Limit

Solution	Application Scenario	Advantage and disadvantage	Limit
Data forwarding	• Scenarios in which data processing is complex. • Scenarios in which data throughput is high.	Advantages: • Allows you to forward data in most scenarios. • Allows you to modify data forwarding rules while the rules are running. • Supports data filtering and processing. • Allows you to forward data to other Alibaba Cloud services. The following table Comparison of data forwarding solutions for different Alibaba Cloud services compares the data forwarding solutions for different Alibaba Cloud services. Disadvantages: Complex to use. You must write SQL statements and configure data forwarding rules.	See Limits on data forwarding.
Server-side subscription	• Scenarios in which device data is only received. • Scenarios in which the server receives all subscribed device data of a product.	Advantages: It is easy to use and efficient. Disadvantages: It does not support data filtering and processing.	See Limits of server-side subscriptions.

Comparison of data forwarding solutions for different Alibaba Cloud services

Data Destination	Application Scenario	Advantages	Disadvantages
Message Service (MNS)	Scenarios in which complex or refined data processing is required in the Internet environment. If the message processing speed is less than 1, 000 queries per second (QPS), we recommend that you forward data to MNS.	• Supports data forwarding based on HTTPS protocols. • Supports data forwarding in the Internet environment.	-

Data Destination	Application Scenario	Advantages	Disadvantages
ApsaraDB RDS	Scenarios in which data is only stored.	Data is directly written to databases.	-
Tablestore (OTS)	Scenarios in which data is only stored.	Data is directly written to Tablestore instances.	-
Function Compute	Scenarios in which the device development process must be simplified and device data must be processed in a flexible way.	• Supports flexible data processing. • Supports multiple features. • No deployment is required.	The cost is slightly higher than other solutions.

Server-side subscription

A server can use AMQP SDK or MNS SDK to receive device messages of the subscribed products. The device messages include upstream messages, status changes, lifecycle changes, historical TSL model statistics, OTA update statuses, gateway and sub-device connections, and topological changes.

Limit	Usage notes	Documentation
• Server-side subscription does not support data filtering. A server receives all messages from the devices of a product. • A single consumer group can process a maximum of 1,000 QPS. For more information, see Limits of server-side subscriptions.	• Server-side subscription prioritizes the processing of real-time messages and degrades the processing of accumulated messages. We recommend that you make sure messages are consumed in a timely manner. • Server-side subscription is not applicable to scenarios in which data filtering and fine-grained processing are required. We recommend that you use a rules engine for these scenarios.	• Overview of server-side subscription • Configure an AMQP server-side subscription • Connect an AMQP client to IoT Platform • Configure MNS server-side subscriptions

Forward data to Message Service (MNS)

The data forwarding feature of IoT Platform allows you to forward messages from specified topics to the topics in MNS. Then, you can receive these messages by using MNS SDK. MNS is compatible with Internet environments. We recommend that you forward data to MNS if the message throughput is less than 1,000 QPS.

Limit	Usage notes	Documentation
For more information about the limits of MNS topics, see Limits of MNS.	If a message fails to be forwarded, the rules engine retries to forward the message. If a specified number of retries fail, the message is discarded. Latency may occur in message-based services. We recommend that you take preventive measures to minimize the impact of data loss or transmission delay.	• Configure a data forwarding rule • Forward data to Message Service • MNS user guide

Forward data to Function Compute

The data forwarding feature of IoT Platform allows you to forward messages from specified topics to Function Compute. Then, you can process the received messages. Function Compute is a fully-managed service that simplifies business development.

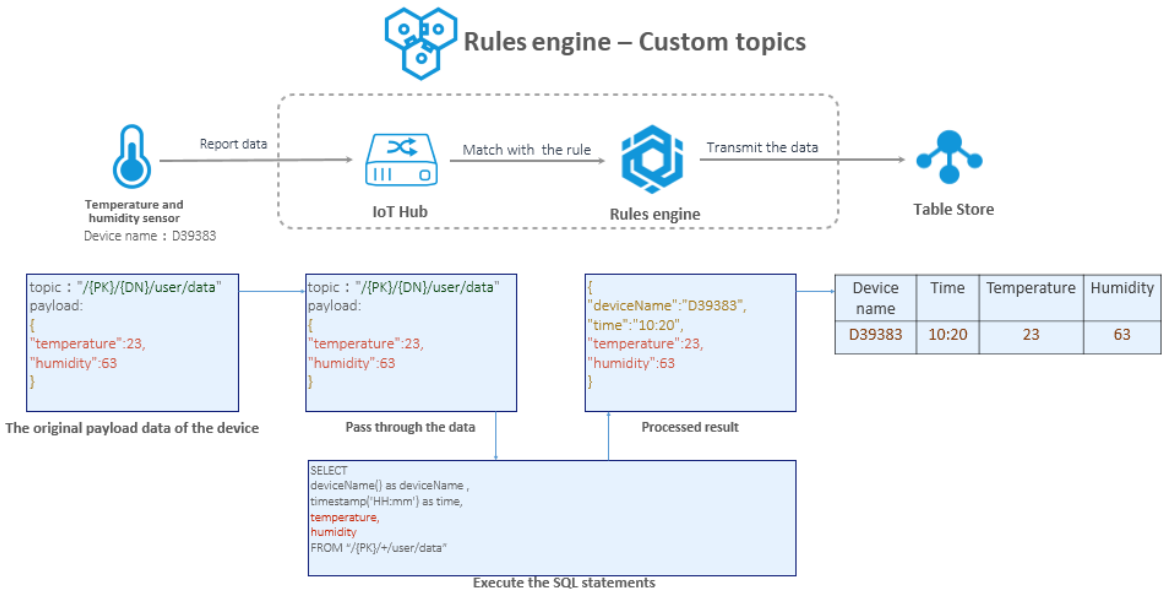
Limit	Usage notes	Documentation
See Limits of Function Compute.	• Function Compute is applicable to scenarios in which you want to customize data processing or simplify the development and maintenance processes. • If a message fails to be forwarded, the rules engine retries to forward the message. If a specified number of retries fail, the message is discarded. We recommend that you take preventive measures to minimize the impact of data loss or transmission delay.	• Configure a data forwarding rule • Forward data to Function Compute • Function Compute user guide

4.3. Data forwarding procedure

The rules engine can only process data that is sent to topics. This topic describes the data forwarding procedure and the data formats at different stages of data forwarding.

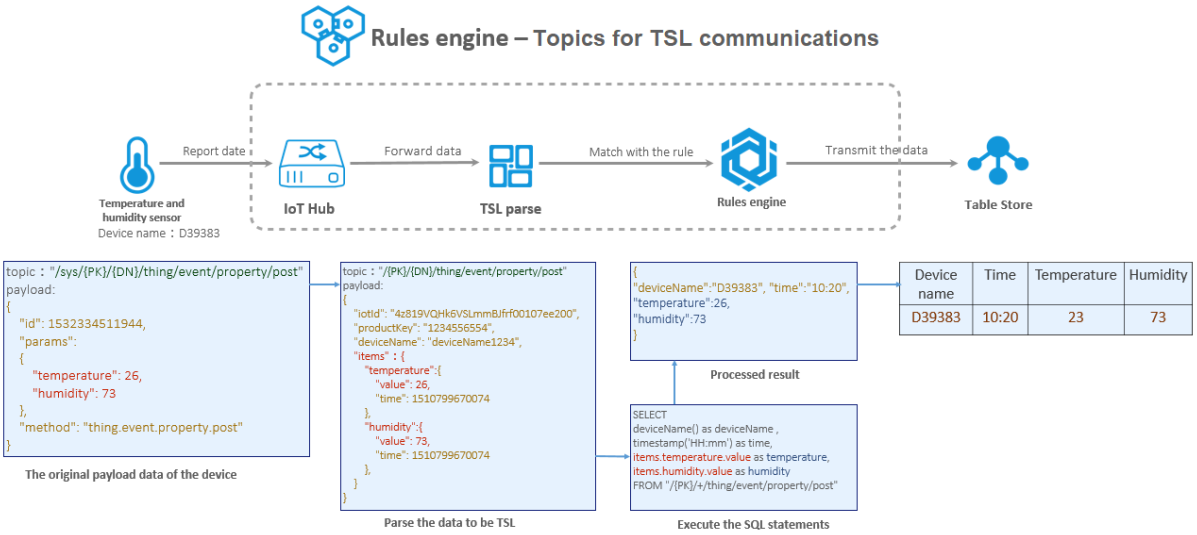
Custom topics

Custom topics are used to submit pass-through data of devices to IoT Platform. The structure of the data is not changed during data submission. The following figure shows the data forwarding procedure:



Topics for TSL communications

Data sent in topics for TSL communications is in the Alink JSON format. During data forwarding, the data is parsed based on the TSL and then processed by the SQL statement of the rules engine. For more information about data formats, see Data formats. The following figure shows the data forwarding procedure:



Note During data forwarding, the params parameter in the payloads changes to the items parameter after the payloads are parsed based on the TSL.

4.4. Configure a data forwarding rule

You can use the rules engine of IoT Platform to forward data from a specified topic to other topics or other Alibaba Cloud services. This topic describes how to configure a data forwarding rule. To configure a data forwarding rule, perform the following steps: Create a rule, write an SQL statement for data processing, specify a data forwarding destination, and specify a destination for error messages if data forwarding fails.

Procedure

1.
2.
3.
4. On the **Data Forwarding** page, click **Create Rule**.

   > **Notice** If the Data Forwarding page of the latest version appears, click **Back to Previous Version** in the upper-right corner of the page. When the Data Forwarding page of the previous version appears, click **Create Rule**.

5. Configure the parameters and click **OK**. The following table describes the parameters.

Parameter	Description
Rule Name	Enter a name for the forwarding rule. The rule name must be 1 to 30 characters in length, and can contain letters, digits, underscores (_), and hyphens (-).
Data Type	The data format that the rules engine can process. Valid values: **JSON** and **Binary**. **Note** ◦ The data forwarding feature processes data based on topics. You must select the format of the data in the topics that you want to process. ◦ If you select Binary, the rule cannot be used to process messages from basic communication topics and Thing Specification Language (TSL) communication topics. The rule also cannot be used to forward data to Tablestore or ApsaraDB RDS.
Rule Description	The description of the rule.

6. After the rule is created, the **Data Forwarding Rule** page appears. You must write an SQL statement to process messages, specify a data forwarding destination, and specify a destination to which error messages are forwarded.
   i. Click **Write SQL** to write an SQL statement. The SQL statement allows you to process message fields.

   For more information about how to write SQL statements, see SQL statements and Functions.

Parameter	Description
Rule Query Expression	The SQL statement. The SQL statement is automatically generated based on the values that you specify for the Field, Topic, and Conditions (Optional) parameters.

Parameter	Description
Field	The field of the message to be processed. This parameter follows the SELECT keyword in the SQL statement. For example, if you enter *deviceName() as deviceName*, the deviceName field in the message is selected. For more information about the functions that can be used in the field, see Functions. **Note** The data of basic communication topics and TSL communication topics is in the Alink JSON format. Before the data is forwarded to the rules engine, the data is parsed based on the corresponding TSL model. For more information about data parsing, see Data forwarding procedure. For more information about the formats of parsed data, see Data formats. When you write an SQL statement, specify the fields based on the format of parsed data.
Topic	The source topic of the message to be processed. This parameter follows the FROM keyword in the SQL statement. For more information about optional topics, see the following *Topics* table. **Notice** If the value of the **Data Type** parameter of the rule is **Binary**, set this parameter to **Custom**.
Condition (Optional)	The trigger condition of the rule. This parameter follows the WHERE keyword in the SQL statement.

Topics

Topic	Description	References
Custom	The topic that is used to forward data of custom formats. The format of this topic is the same as the format of a custom topic. Format: `/${productKey}/${deviceName}/user/${TopicShortName}`. `${TopicShortName}` specifies a custom topic category, which indicates the suffix of the custom topic. The value can contain wildcard characters, including plus signs (+) and number signs (#). ■ All equipment (`+`): indicates all devices of the specified product. ■ `/user/#`: indicates all topics of the specified device.	Custom topics

Topic	Description	References
Device Status Change Notification	The topic that is used to forward notifications when the status of a device changes between online and offline. Format: `/as/mqtt/status/${productKey}/${deviceName}`.	Submit device status
TSL Data Reporting	The following topics are provided: ■ `/${productKey}/${deviceName}/thing/event/property/post`: This topic is used to forward device properties. ■ `/${productKey}/${deviceName}/thing/event/${tsl.event.identifier}/post`: This topic is used to forward device events. ■ `/${productKey}/${deviceName}/thing/event/property/post`: This topic is used to forward multiple device properties at a time. ■ `/${productKey}/${deviceName}/thing/event/batch/post`: This topic is used to forward multiple device events at a time. ■ `/${productKey}/${deviceName}/thing/downlink/reply/message`: This topic is used to forward messages that are sent by a device as responses to IoT Platform commands.	■ Submit device properties ■ Submit device events ■ Submit device properties in batches ■ Submit device events in batches ■ Submit responses to downstream requests
	The preceding topics correspond to the following device topics: ■ `/sys/{productKey}/{deviceName}/thing/event/property/post`: This topic is used to submit device properties. ■ `/sys/${productKey}/${deviceName}/thing/event/${tsl.event.identifier}/post` and `/sys/${productKey}/${deviceName}/thing/event/${tsl.functionBlockId}:{tsl.event.identifier}/post`: These topics are used to submit device events. ■ `/sys/${productKey}/${deviceName}/thing/event/property/batch/post`: This topic is used to submit device properties and events in batches.	■ Devices submit property information to IoT Platform ■ Devices submit events to IoT Platform ■ Devices submit multiple properties and events to IoT Platform at a time

Topic	Description	References
Device Changes Throughout Lifecycle	The topic that is used to forward notifications when a device is created, deleted, disabled, or enabled. Format: `/${productKey}/${deviceName}/thing/lifecycle`	Submit lifecycle changes
Sub-Device Data Report Detected by Gateway	The topic that is used to submit and forward the information about a new sub-device when a gateway detects the sub-device. This topic is specific to gateways. Format: `/${productKey}/${deviceName}/thing/list/found`	Submit information about detected sub-devices
Device Topological Relation Changes	The topic that is used to forward notifications when topological relationships between sub-devices and the gateway are created or deleted. This topic is specific to gateways. Format: `/${productKey}/${deviceName}/thing/topo/lifecycle`.	Submit topology changes
	The preceding topic corresponds to the following topic: `/sys/{productKey}/{deviceName}/thing/topo/change`. This topic is used to submit device data.	Notify gateways of changes of topological relationships
Device tag change	The topic that is used to forward notifications when device tags are changed. Format: `/${productKey}/${deviceName}/thing/deviceinfo/update`.	Submit device tag changes
	The preceding topic corresponds to the following topic: `/sys/{productKey}/{deviceName}/thing/deviceinfo/update`. This topic is used to submit device data.	Report tags
TSL Historical Data Reporting	The following topics are provided: ■ `/${productKey}/${deviceName}/thing/event/property/history/post`: This topic is used to forward historical properties. ■ `/${productKey}/${deviceName}/thing/event/${tsl.event.identifier}/history/post`: This topic is used to forward historical events.	■ Submit historical properties ■ Submit historical events
	The preceding topics correspond to the following topic: `/sys/{productKey}/{deviceName}/thing/event/property/history/post`. This topic is used to submit historical TSL data.	Devices submit historical TSL data to IoT Platform

Topic	Description	References
Device status notification	The following topics are provided: ■ `/${productKey}/${deviceName}/ota/upgrade`: This topic is used to forward over-the-air (OTA) update results. ■ `/${productKey}/${deviceName}/ota/progress/post`: This topic is used to forward update progresses.	■ Submit the status data of over-the-air (OTA) updates ■ Submit the progress data of OTA updates
	The preceding topics correspond to the following topic: `/ota/device/progress/${YourProductKey}/${YourDeviceName}`. This topic is used to submit update progresses.	Submit the update progress to IoT Platform
Submit a module version number	The topic that is used to forward notifications when the version number of an OTA module for a device is changed. Format: `/${productKey}/${deviceName}/ota/version/post`.	Submit OTA module versions
	The preceding topic corresponds to the following topic: `/ota/device/inform/${YourProductKey}/${YourDeviceName}`. This topic is used to submit the version number of an OTA module.	Submit OTA module versions to IoT Platform
Batch status notification	The topic to which IoT Platform sends notifications when the status of an OTA update batch changes. Format: `/${productKey}/${packageId}/${jobId}/ota/job/status`.	Submit the status data of OTA update batches

ii. In the **Data Forwarding** section, click **Add Operation** to specify the Alibaba Cloud service to which you want to forward the processed data. For more information about how to configure the data forwarding feature, see related topics in the Data forwarding examples directory.

> **Note** You can create up to 10 data forwarding operations for each rule.

If data forwarding fails due to exceptions in the destination cloud service, IoT Platform performs one of the following operations:

- When IoT Platform forwards data to cloud services, such as Message Queue for Apache RocketMQ, ApsaraDB RDS, and ApsaraDB for Lindorm, the cloud services may be inaccessible due to resource changes. In this case, IoT Platform stops forwarding data and changes the status of the related rule to **Abnormal**. Then, you must specify a new destination for data forwarding.
- For other exceptions, IoT Platform retries three times at intervals of 1 second, 3 seconds, and 10 seconds. The retry policy may vary based on the scenarios. If all retries fail, the message is discarded. If your business has high requirements for message reliability, you can add an error operation and forward error messages to other cloud services.

iii. In the **Forward Error Data** section, click **Add Error Operation**. Configure the parameters to

forward error messages to the specified destination after all retries fail.

> **Notice**
>
> - You can add only one error operation for each rule.
> - A normal operation and an error operation cannot forward messages to the same destination. For example, normal data and error data cannot be forwarded to Tablestore at the same time.
> - If an error message fails to be forwarded, no retry is performed.
> - Error messages are generated only if the rules engine fails to forward data due to the issues of other cloud services.

If a message fails to be forwarded to a cloud service, IoT Platform retries to forward the message. If the retry fails, an error message is forwarded based on the error operation that you specify for data forwarding.

Sample error message:

```
{
      "ruleName":"",
      "topic":"",
      "productKey":"",
      "deviceName":"",
      "messageId":"",
      "base64OriginalPayload":"",
      "failures":[
        {
          "actionType":"OTS",
          "actionRegion":"cn-shanghai",
          "actionResource":"table1",
          "errorMessage":""
        },
        {
          "actionType":"RDS",
          "actionRegion":"cn-shanghai",
          "actionResource":"instance1/table1",
          "errorMessage":""
        }
      ]
}
```

The following table describes the parameters of error messages.

Parameter	Description
ruleName	The name of the rule.
topic	The source topic of the message.
productKey	The ProductKey of the product.
deviceName	The DeviceName of the device.

Parameter	Description
messageId	The ID of the message that is sent from IoT Platform.
base64OriginalPayload	The Base64-encoded raw data.
failures	The error details. Multiple errors may occur.
actionType	The type of the failed operation.
actionRegion	The region in which the error occurred.
actionResource	The destination service in which the error occurred.
errorMessage	The error message.

7. Go to the **Data Forwarding** page. Find the rule that you configured and click **Start** in the Actions column. After the rule is enabled, data is forwarded based on the rule.

   You can also perform the following operations:

   - Click **View** to go to the **Data Forwarding Rule** page. Then, modify the settings of the rule.
   - Click **Delete** to delete the rule.

     > **Note** You cannot delete a rule that is in the Running state.

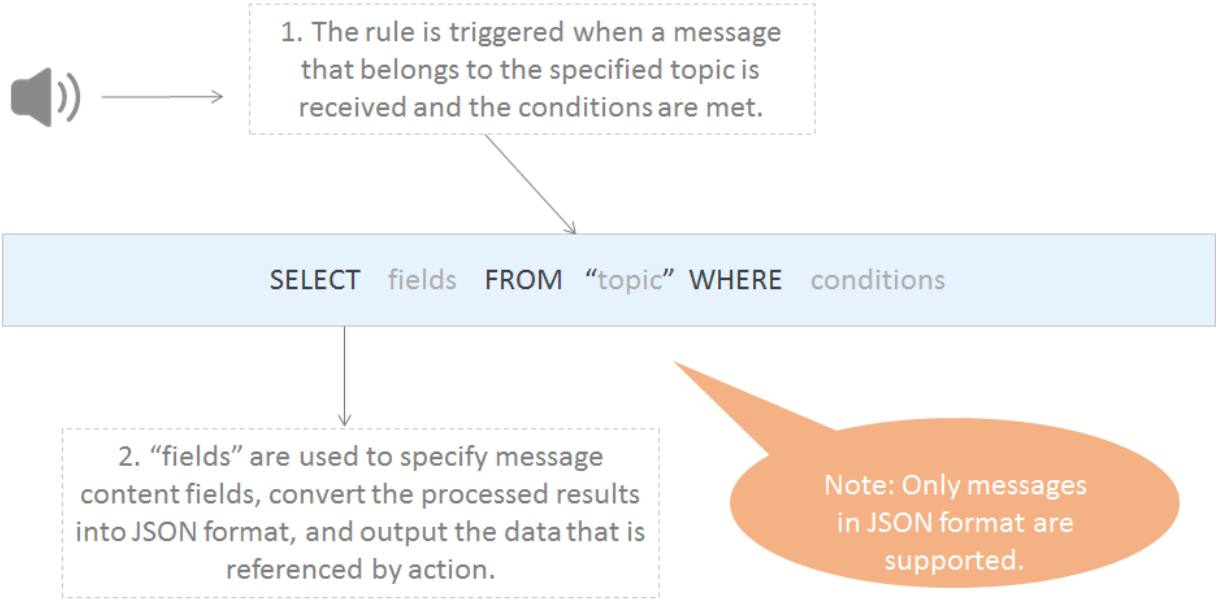
   - Click **Stop** to disable the rule.

4.5. SQL statements

When you create data forwarding rules, you must write SQL statements to parse and process JSON-formatted data that is submitted by devices. IoT Platform does not parse binary data. Binary data is passed through to a specified destination. This topic describes how to write SQL statements for data forwarding rules.

SQL statements

JSON data can be mapped to a virtual table. Keys in a JSON data record correspond to the column names. Values in a JSON data record correspond to the column values. After a JSON data record is mapped to a virtual table, the JSON data record can be processed by using SQL. The following figure shows the format of SQL statements for data forwarding rules.



Examples:

- The following SQL example shows how to process the data of custom topics.

  An environmental sensor can be used to collect temperature, humidity, and atmospheric pressure data. The following code shows the data that is submitted by a device to the /a1hRrzD****/+/user/update custom topic.

  ```
  {
  "temperature":25.1
  "humidity":65
  "pressure":101.5
  "location":"***,***"
  }
  ```

  If the temperature is higher than 38 degrees Celsius, a rule is triggered and the device name, temperature data, and location data are returned. To implement this use case, the following SQL statement is used:

  ```
  SELECT temperature as t, deviceName() as deviceName, location
  FROM "/a1hRrzD****/+/user/update"
  WHERE temperature > 38
  ```

- The following SQL example shows how to process the data of basic communication topics and Thing Specification Language (TSL)-based communication topics. Data can be forwarded from basic communication topics and Thing Specification Language (TSL)-based communication topics to the rules engine. After the data is received, the rules engine parses the data. For more information about the format of parsed data, see Data formats.

  For example, a temperature and humidity sensor has several properties, as shown in the following figure.

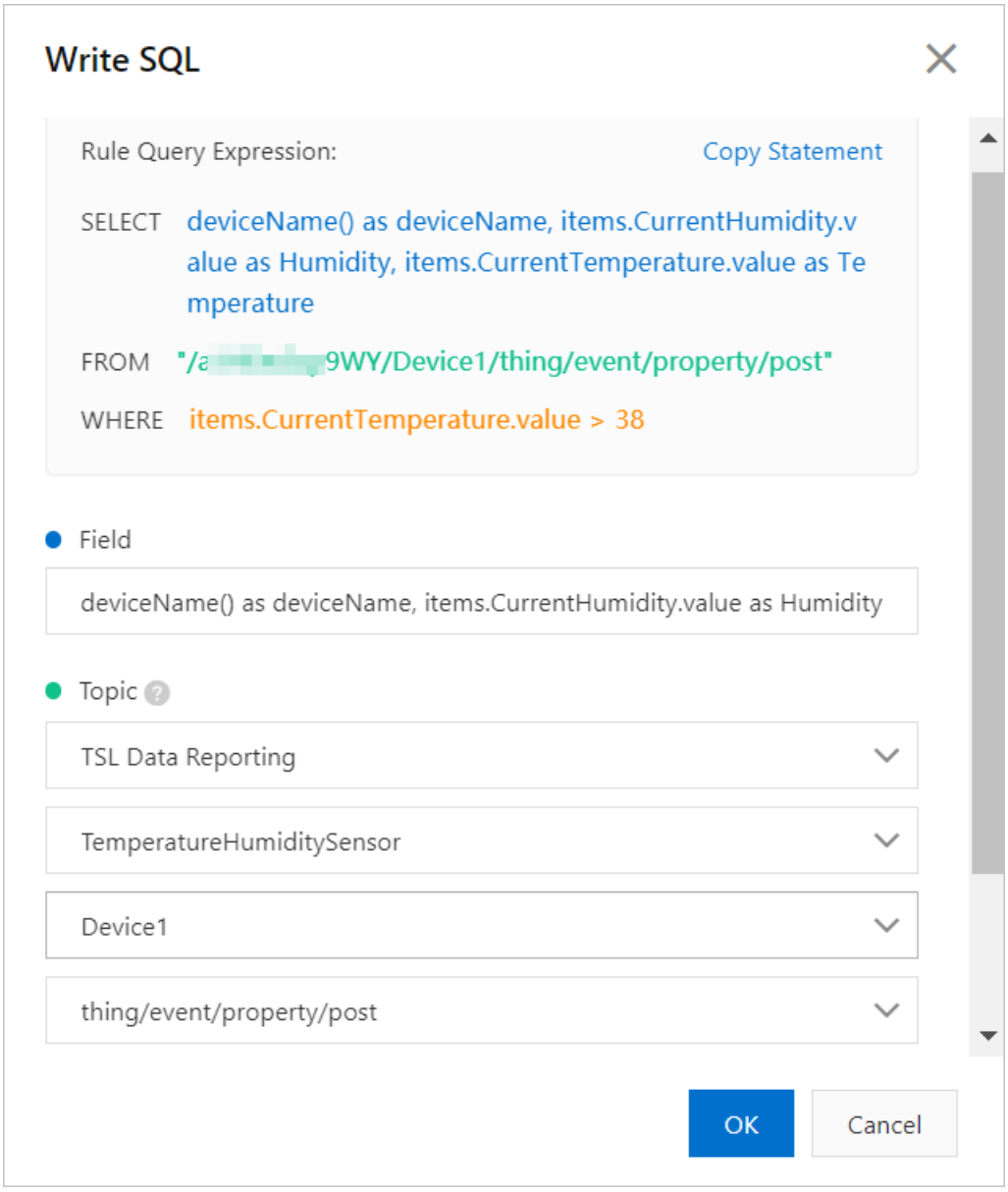
Feature Type	Feature Name	Identifier	Data Type	Data Definition
Properties	Temperature Custom	Temperature	int32	Value Range: -20 ~ 60
Properties	Humidity Custom	Humidity	int32	Value Range: 0 ~ 100

The following sample code shows the result after the rules engine parses the temperature and humidity data that is submitted by the temperature and humidity sensor.

```
{
    "deviceType": "TemperatureHumidityDetector",
    "iotId": "N5KURkKdibnZvSls****000100",
    "productKey": "a15NNfl****",
    "gmtCreate": 1564569974224,
    "deviceName": "N5KURkKdibnZvSls3Yfx",
    "items": {
        "CurrentHumidity": {
            "value": 70,
            "time": 1564569974229
        },
        "CurrentTemperature": {
            "value": 23.5,
            "time": 1564569974229
        }
    }
}
```

Note When you issue SQL queries, you must use the `items.${Property identifier}.value` variable to access the data of a specified property.

If the temperature is higher than 38 degrees Celsius, a rule is triggered and the device name, current temperature data, and current humidity data are returned. To implement this use case, the following SQL statement is used:




```
SELECT deviceName() as deviceName, items.CurrentHumidity.value as Humidity, items.Current
Temperature.value as Temperature
FROM "/sysa15NNfl****/N5KUR***/thing/event/property/post"
WHERE items.CurrentTemperature.value > 38
```

If a property belongs to a custom module such as testFB, the format of the property identifier is `Module identifier:Property identifier`. When the expression is used to access the data of a specified property, you must enclose the property in a pair of double quotation marks (""). The following SQL statement provides an example.

```
SELECT deviceName() as deviceName, "items.testFB:CurrentHumidity.value" as Humidity, "ite
ms.testFB:CurrentTemperature.value" as Temperature
FROM "/sysa15NNfl****/N5KUR***/thing/event/property/post"
WHERE "items.testFB:CurrentTemperature.value" > 38
```

SELECT

- The following example shows JSON data that contains labels:

  You can use the parsing result for the payload of a submitted message as the fields of a SELECT statement. The parsing result includes the keys and values of the JSON data. You can also use SQL built-in functions as the fields, for example, `deviceName()`. For more information about the SQL built-in functions of the rules engine, see Functions.

  You can use asterisks (`*`) together with functions. SQL subqueries are not supported.

  The data that is submitted in the JSON format can be arrays or nested JSON data. You can use JSONPath to obtain a property value from an SQL statement. For example, you can use `a.key2` to obtain the `v2` value from the `{a:{key1:v1, key2:v2}}` statement. When you specify variables in SQL statements, take note of the difference between single quotation marks (') and double quotation marks ("). Each constant is enclosed in a pair of single quotation marks ('). Each variable is enclosed in a pair of double quotation marks (""). A variable may alternatively be written without being enclosed by quotation marks. For example, `'a.key2'` indicates a constant whose value is `a.key2`.

  In this SQL example:

  - The `SELECT temperature as t, deviceName() as deviceName, location` statement is used to process the data of a custom topic. The `temperature` and `location` fields are obtained from submitted data. The `deviceName()` SQL built-in function is used as a field.
  - The `SELECT deviceName() as deviceName, items.CurrentHumidity.value as Humidity, items.CurrentTemperature.value as Temperature` statement is used to process the submitted property data of a topic. The `items.CurrentHumidity.value` and `items.CurrentTemperature.value` fields are obtained from the submitted property data of the default module. The `deviceName()` SQL built-in function is used as a field.

    > **Note** The `items.testFB:CurrentHumidity.value` and `items.testFB:CurrentTemperature.value` fields are obtained from the submitted property data of a custom module.

- Binary data
  - Enter an asterisk (`*`) to pass through binary data. After you specify an asterisk (`*`), you can no longer use functions.
  - You can use built-in functions. The `to_base64(*)` function converts an original binary payload into a Base64 string. The `deviceName()` function extracts the name of a device.

> **Note** A SELECT statement can include a maximum of 50 fields.

FROM

You can specify a topic in the FROM clause. This topic is the source from which the device messages to be processed are obtained. In this topic, you can specify a plus sign (+) as a wildcard for the device name category. The plus sign (+) represents all categories at the current level. In this case, the plus sign (+) represents all devices of a specified product. After you specify a custom topic, you can specify a number sign (#) as a wildcard. The number sign (#) represents all categories at the current level and the subsequent levels. For more information about wildcards, see Custom topics.

When the messages of a specified topic are received, the payload data of these messages are converted into the JSON format and processed based on the specified SQL statement. If the format of a message is invalid, the message is ignored. You can use the `topic()` function to specify a topic.

In the preceding SQL example:

- The `FROM "/a1hRrzD****/+/user/update"` clause indicates that the related SQL statement processes only the messages of the /a1hRrzD****/+/user/update custom topic.
- The `FROM "/sys/a15NNfl****/N5KURkKdibnZvSls3Yfx/thing/event/property/post"` clause indicates that the related SQL statement processes only the messages from the topic of the N5KURkKdibnZvSls3Yfx device. The topic is used by the device to submit property data.

WHERE

- JSON data

  The WHERE clause is used as the condition to trigger the rule. SQL subqueries are not supported. The fields that can be used in the WHERE clause are the same as the fields that can be used in the SELECT statement. When a message of the related topic is received, the results that are obtained by using the WHERE clause are used to check whether a rule is triggered. For more information, see the "Supported WHERE expressions" section of this topic.

  In the preceding two examples, the `WHERE temperature > 38` condition indicates that the rules are triggered only when the temperature is higher than 38 degrees Celsius.

- Binary data

  If the reported message is composed of binary data, you can use only built-in functions and conditions in a WHERE clause. You cannot use the fields in the payload of the reported message.

SQL results

After the SQL statement is executed, you can forward the query result. If an error occurs when IoT Platform parses the payload of the reported message, the rule execution fails.

You can forward data to Tablestore. In this case, you must use the `${expression}` variable to specify the required value when you specify the data forwarding destination.

In the preceding two SQL examples, if you want to use the related rules to forward data to Tablestore tables, you can specify the following variables as the primary keys:

- *${t}*, *${deviceName}*, and *${loaction}*.
- *${deviceName}*, *${Humidity}*, and *${Temperature}*.

Arrays

Enclose each array expression in a pair of double quotation marks (""). Use `$.` to obtain a JSON object. `$.` can be removed. Use `.` to obtain a JSON array.

If a device message is `{"a":[{"v":0},{"v":1},{"v":2}]}`, the following results are obtained based on the specified expressions:

- The result of `"a[0]"` is `{"v":0}`
- The result of `"$.a[0]"` is `{"v":0}`
- The result of `".a[0]"` is `[{"v":0}]`
- The result of `"a[1].v"` is `1`

- The result of `"$.a[1].v"` is `1`
- The result of `".a[1].v"` is `[1]`

Supported WHERE conditions

Operator	Description	Example
=	Equal to	color = 'red'
<>	Not equal to	color <> 'red'
AND	Logical AND	color = 'red' AND siren = 'on'
OR	Logical OR	color = 'red' OR siren = 'on'
+	Add	4 + 5
-	Subtract	5 - 4
/	Divide	20 / 4
*	Multiply	5 * 4
%	Returns the remainder	20 % 6
<	Less than	5 < 6
<=	Less than or equal to	5 <= 6
>	Greater than	6 > 5
>=	Greater than or equal to	6 >= 5
Function calls	Functions. For more information, see Functions.	deviceId()
Fields that are specified in the JSON format	You can extract fields from a message payload and specify these fields in the JSON format.	state.desired.color,a.b.c[0].d
CASE ... WHEN ... THEN ... ELSE ...END	CASE expressions. Nested expressions are not supported.	CASE col WHEN 1 THEN 'Y' WHEN 0 THEN 'N' ELSE '' END as flag
IN	Only enumeration is supported. Subqueries are not supported.	For example, you can use WHERE a IN(1, 2, 3). However, you cannot use WHERE a IN(select xxx).
like	This operator is used to match a character. When you use a LIKE operator, you can use only a percent sign (`%`) as a wildcard to specify an arbitrary character string.	where c1 like '%abc', where c1 not like '%def%'

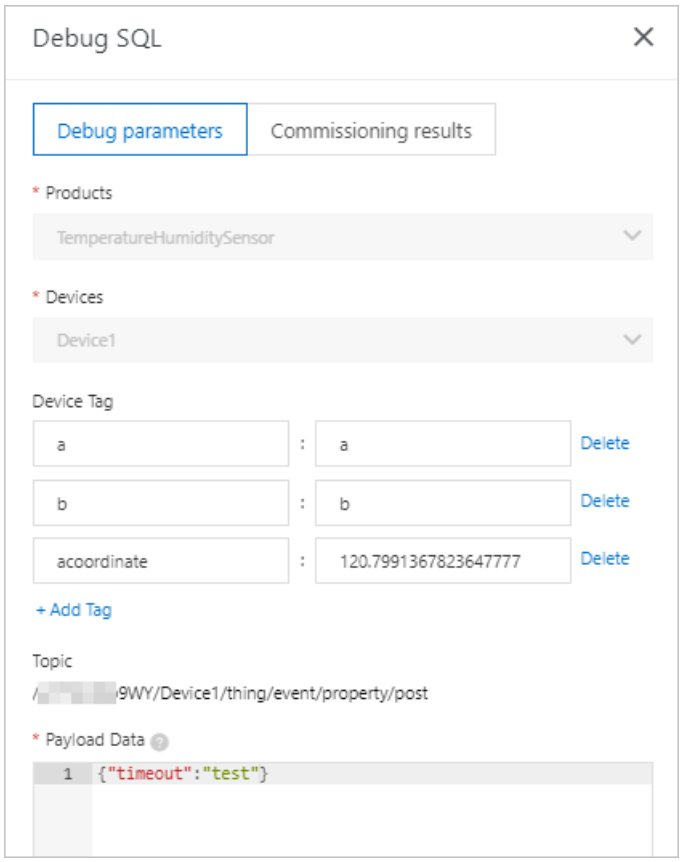
Debug SQL statements

If you select **JSON** in the **Data Type** field when you create a data forwarding rule, you can debug an SQL statement in the IoT Platform console. Procedure:

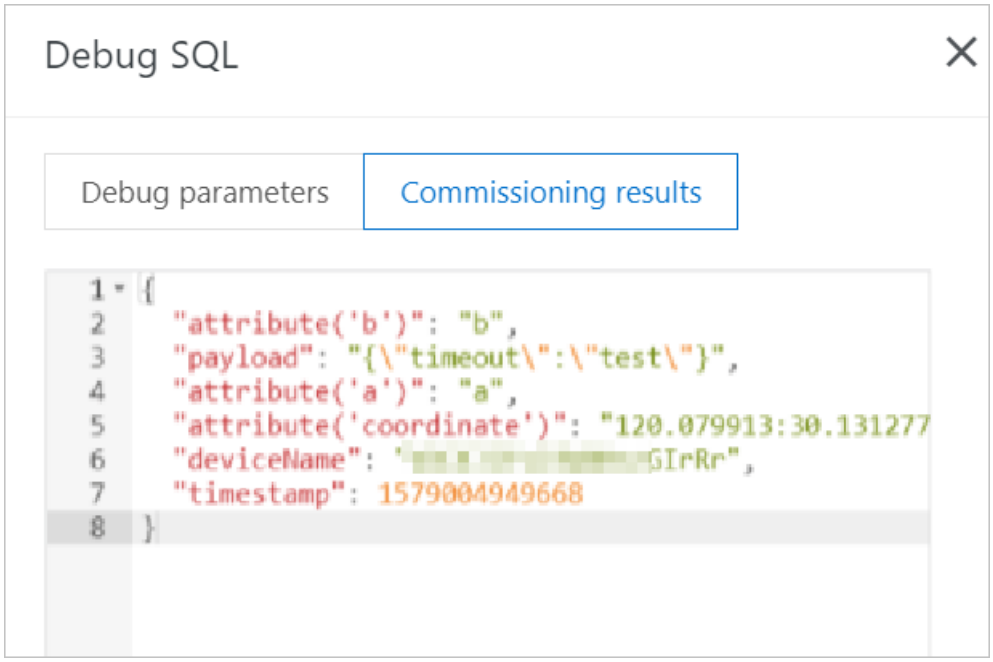
1. After an SQL statement is written, click **Debug SQL**.
2. In the **Debug SQL** panel, click the **Debug parameters** tab. On the tab that appears, enter the required debugging data and click **Debugging**.

   Enter the required payload data for debugging based on the type of the data that is submitted by a topic. Description of data types:

   - If you use a custom topic, the type of the specified payload data must be the same as the type of the data that is submitted by the custom topic.
   - If you use a basic communication topic or a TSL-based communication topic, see Data formats.



3. Click the **Commissioning results** tab to view the result.



4.6. Functions

The rules engine provides functions that you can use in SQL statements to process data.

Functions supported by the data forwarding feature

Function	Description
abs(number)	Returns the absolute value of the number.
asin(number)	Returns the arcsine of the number.
attribute(key)	Returns the device tag value of the tag key. If the tag with the specified key is not attached to a device, no tag value is returned. When you debug your SQL statement, a null string is returned because no actual device or tag exists.
concat(string1, string2)	Concatenates the strings. A concatenated string is returned. Example: `concat(field,'a')`.
cos(number)	Returns the cosine of the number.
cosh(number)	Returns the hyperbolic cosine of the number.
crypto(field,String)	Encrypts the value of the field. The String parameter specifies an encryption algorithm. Available algorithms include MD2, MD5, SHA1, SHA-256, SHA-384, and SHA-512.
deviceName()	Returns the name of the current device. When you debug your SQL statement, a null string is returned because no actual device exists.
endswith(input, suffix)	Checks whether the input string ends with the suffix string.

Function	Description
exp(number)	Returns the value of the mathematical constant e that is raised to the power of a specified number.
floor(number)	Returns the largest integer that is less than or equal to the number.
log(n, m)	Returns the logarithm of number n to base m. If you do not specify m, the default base 10 is used. In this case, `log(n)` is returned.
lower(string)	Converts all letters in the specified string into lowercase and returns the lowercase string.
mod(n, m)	Returns the remainder after number n is divided by divisor m.
nanvl(value, default)	Returns the value of the property. The value parameter specifies the name of the property. If the value of the property is null, the function returns the value of the default parameter.
newuuid()	Returns a random universally unique identifier (UUID).
payload(textEncoding)	Returns the message payload that is sent by a device. The payload is encoded by using the encoding scheme that is specified by the textEncoding parameter. The default encoding scheme is UTF-8. This indicates that `payload()` and `payload('utf-8')` return the same result.
power(n,m)	Returns number n raised to the power of m.
rand()	Returns a random number that is greater than or equal to 0 and less than 1.
replace(source, substring, replacement)	Replaces the substring in the source column with the replacement string. Example: `replace(field,'a','1')`.
sin(n)	Returns the sine of number n.
sinh(n)	Returns the hyperbolic sine of number n.
tan(n)	Returns the tangent of number n.
tanh(n)	Returns the hyperbolic tangent of number n.

Function	Description
thingPropertyFlatMap(property)	Returns the values of a property in a Thing Specification Language (TSL) model. If a property has multiple values, separate the values with underscores (_). If a TSL model contains more than 50 properties, the data forwarding feature cannot forward the entire TSL model. You can use this function to extract property data from the TSL model. This way, all properties of the TSL model can be forwarded to other Alibaba Cloud services. You can specify multiple properties as the input parameters of the function. If you do not specify a property, all the values of the properties are extracted. For example, the function `thingValueFlat("Power", "Position")` adds `"Power": "On", "Position_latitude": 39.9, "Position_longitude": 116.38` to a message body.
timestamp(format)	Returns the timestamp of the current system time in the specified format. The format parameter is optional. If you do not specify the format parameter, the timestamp of the current system time is returned. For example, `timestamp()` returns `1543373798943` and `timestamp('yyyy-MM-dd\'T\'HH:mm:ss\'Z\'')` returns `2018-11-28T10:56:38Z`.
timestamp_utc(format)	Returns the UTC timestamp of the current system time in the specified format. The format parameter is optional. If you do not specify the format parameter, the 13-digit timestamp of the current system time is returned. For example, `timestamp_utc()` returns `1543373798943` and `timestamp_utc('yyyy-MM-dd\'T\'HH:mm:ss\'Z\'')` returns `2018-11-28T02:56:38Z`.
topic(number)	Returns the topic information at a specified level. For example, a topic is named `/alDbcLe****/TestDevice/user/`. `topic()` returns the complete topic name /alDbcLe****/TestDevice/user/set. `topic(1)` returns the first level `alDbcLe****`. `topic(2)` returns the second level `TestDevice`.
upper(string)	Converts all letters in the specified string into uppercase and returns the uppercase string. For example, `upper(alibaba)` returns `ALIBABA`.
to_base64(*)	Converts the message payload from binary data into a Base64-encoded string, and returns the message payload after conversion. You can use this function if the original message payload is of the binary data type.
messageId()	Returns the message ID that is generated by IoT Platform.

Function	Description
substring(target, start, end)	Returns part of the specified string. Parameters • target: the original string. This parameter is required. • start: the position from which the returned characters start. The character at the position is also returned. This parameter is required. • end: the position at which the returned characters end. The character at the position is not returned. This parameter is optional. **Note** • Only data of the string and integer types is supported. An integer is converted into a string before the integer is processed. • Enclose a constant string in a pair of single quotation marks (''). The data that is enclosed in double quotation marks (") is parsed as integers. • If an input parameter is invalid, for example, the data type is not supported, SQL parsing fails and the rules are not triggered. Examples: • substring('012345', 0) = "012345" • substring('012345', 2) = "2345" • substring('012345', 2.745) = "2345" • substring(123, 2) = "3" • substring('012345', -1) = "012345" • substring(true, 1.2) error • substring('012345', 1, 3) = "12" • substring('012345', -50, 50) = "012345" • substring('012345', 3, 1) = ""
to_hex(*)	Converts the message payload from binary data into a hexadecimal string, and returns the message payload after conversion. You can use this function if the original message payload is of the binary data type.
user_property()	Obtains the value of the UserProperty parameter when MQTT 5.0 is used by a device. • `user_property()` is used to obtain all property data. • `user_property('${Key}')` is used to obtain the data of the specified key. For example, the value of the UserProperty parameter that is reported by a device is `{"a": "1", "b": "2"}`. • `user_property()` returns `"{\"a\": \"1\", \"b\": \"2\"}"`. • `user_property('a')` returns `"{\"a\": \"1\"}"`.

Function	Description
things_function_type()	Obtains the type of the reported TSL data. This function is used only to query TSL data. The return value of this function varies based on the type of the TSL feature that is queried. • If you query a property, the value `property` is returned. • If you query an event, the identifier of the event is returned. • If you query a service, the identifier of the service is returned. For example, the `/SdfgeW***/device1/thing/event/BrokenInfo/post` topic is used to forward the BrokenInfo event. If you use the `things_function_type()` function to query the event, the identifier of the reported event `BrokenInfo` is returned.
things_property('${Parameter name}')	Obtains the value of a TSL property, service, or event. This function is used only to query TSL data. For example, the `/${productKey}/${deviceName}/thing/event/property/post` topic is used to forward the CurrentTemperature property. `things_property('CurrentTemperature')` returns the reported value of the CurrentTemperature property.

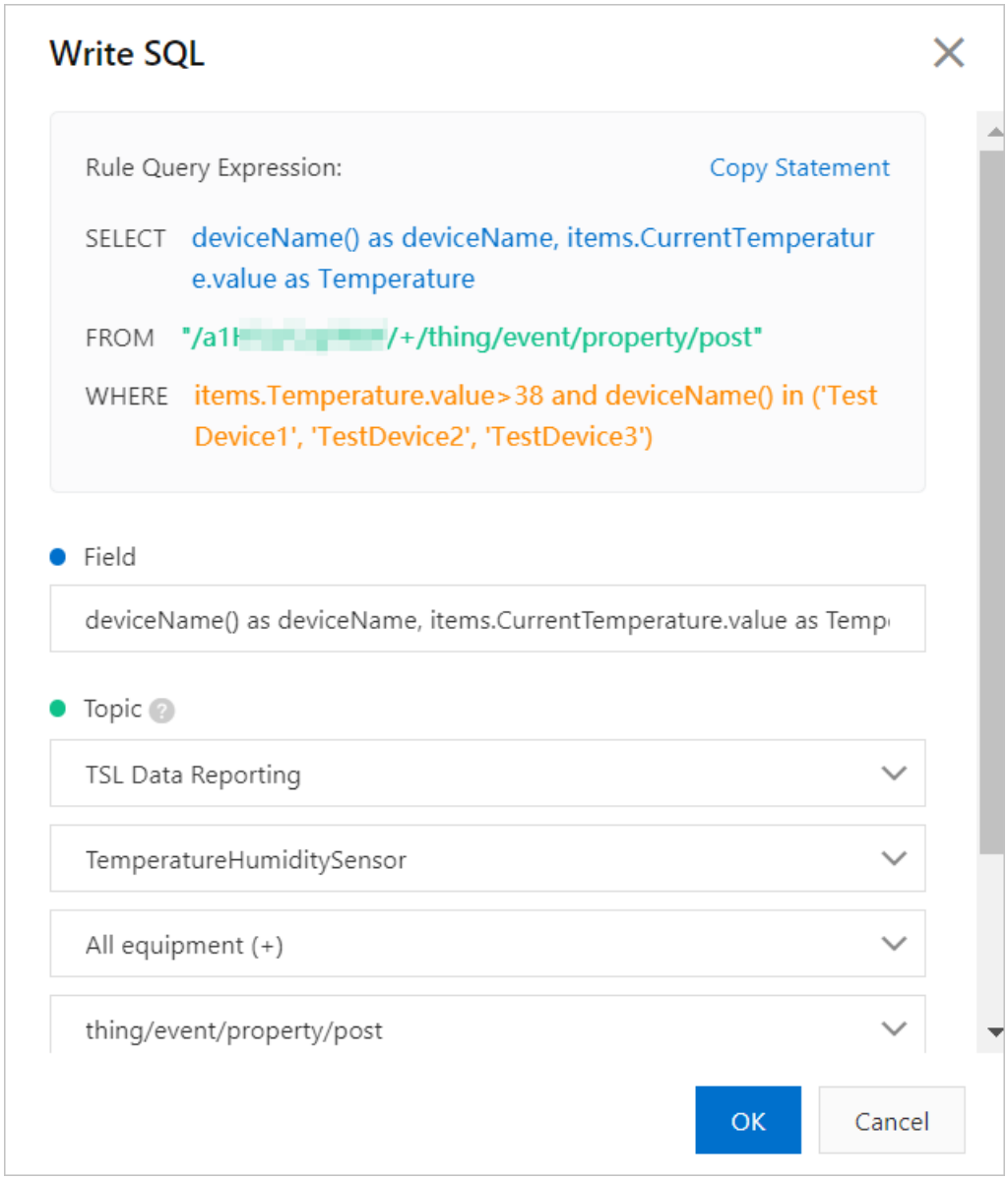
Examples

You can call functions to obtain or process data in the SELECT and WHERE clauses of SQL statements.

For example, a temperature sensor product has the Temperature property. The following script shows the TSL property data that is submitted by a device.

```
{
    "deviceType": "Custom",
    "iotId": "H5KURkKdibnZvSls****000100",
    "productKey": "a1HHrkm****",
    "gmtCreate": 1564569974224,
    "deviceName": "TestDevice1",
    "items": {
        "Temperature": {
            "value": 23.5,
            "time": 1564569974229
        }
    }
}
```

The temperature sensor product has multiple devices. The device names are TestDevice1, TestDevice2, and TestDevice3. The temperature property is forwarded to Function Compute for processing only if the submitted property value is greater than 38. The following figure shows the SQL statement of the rule that is used to filter submitted device data.



SQL statement:

```
SELECT deviceName() as deviceName,things_property('Temperature') as Temperature
FROM "/g5or0***/+thing/event/property/post"
WHERE things_property('Temperature')>38 and deviceName() in ('TestDevice1', 'TestDevice2',
'TestDevice3')
```

In this example, the deviceName() and things_property('Temperature') functions are used.

- These functions are used in the SELECT clause to filter the submitted dat to obtain a device name and the value of the Temperature parameter.
- These functions are used in the WHERE clause to specify a condition. `>38` and `in ('TestDevice1', 'TestDevice2', 'TestDevice3')` indicate that the property data is forwarded only if the temperature is greater than 38 degrees Celsius and the device name is TestDevice1, TestDevice2, or TestDevice3.

For more information about how to write the SELECT and WHERE clauses in an SQL statement and which conditional expressions are supported by the rules engine, see SQL statements.

4.7. Regions and zones

IoT Platform allows you to use the rules engine to forward device data to other Alibaba Cloud services. Before you forward device data, make sure that the destination cloud service is available in the specified region and supports the format of the forwarded data.

Background information

By default, the public instances of IoT Platform are available in the China (Shanghai), Singapore (Singapore), Japan (Tokyo), US (Silicon Valley), US (Virginia), and Germany (Frankfurt) regions.The Enterprise Edition instances of IoT Platform are available only in the Japan (Tokyo) region.

- For regions in which Enterprise Edition instances are available, you can view a public instance and the Enterprise Edition instances that you purchased on the **Overview** page in the IoT Platform console. You can click the card of the instance that you want to manage to go to the Instance Details page. The features of the instance are displayed on the page.
- For regions in which Enterprise Edition instances are unavailable, IoT Platform does not provide the **Overview** page. By default, the features of a public instance are displayed in the left-side navigation pane.

Public instances

The following tables list the data forwarding destinations and data formats that are supported by each region.

- China (Shanghai)

Data forwarding destination	Data format	
	JSON	Binary
Message Service (MNS)	Supported	Supported
Message Queue for Apache RocketMQ	Supported	Supported
Function Compute	Supported	Not supported
Tablestore	Supported	Not supported
ApsaraDB RDS	Supported	Not supported

- Japan (Tokyo)

Data forwarding destination	Data format	
	JSON	Binary
Message Service (MNS)	Supported	Not supported
Function Compute	Supported	Supported
Tablestore	Supported	Not supported

Data forwarding destination	Data format	
	JSON	Binary
ApsaraDB RDS	Supported	Not supported

- Singapore (Singapore)

Data forwarding destination	Data format	
	JSON	Binary
Message Service (MNS)	Supported	Supported
Message Queue for Apache RocketMQ	Supported	Supported
Function Compute	Supported	Supported
Tablestore	Supported	Not supported
ApsaraDB RDS	Supported	Not supported

- US (Silicon Valley) and US (Virginia)

Data forwarding destination	Data format	
	JSON	Binary
Message Service (MNS)	Supported	Supported
Tablestore	Supported	Not supported
ApsaraDB RDS	Supported	Not supported

- Germany (Frankfurt)

Data forwarding destination	Data format	
	JSON	Binary
Message Service (MNS)	Supported	Supported
Tablestore	Supported	Not supported
ApsaraDB RDS	Supported	Not supported

Enterprise Edition instances

Enterprise Edition instances are available only in the Japan (Tokyo) region. The following table list the data forwarding destinations and data formats that are supported by each region.

Notice Enterprise Edition instances do not support cross-region data forwarding. ExampleYou can forward the instance data of IoT Platform in the region only to ApsaraDB RDS tables in the Japan (Tokyo) region.

Data forwarding destination	Data format	
	JSON	Binary
Message Service (MNS)	Supported	Supported
Message Queue for Apache RocketMQ	Supported	Supported
Function Compute	Supported	Supported
Tablestore	Supported	Not supported
ApsaraDB for Lindorm	Supported	Not supported
ApsaraDB RDS	Supported	Not supported

4.8. Data forwarding examples

4.8.1. Forward data to a topic

You can forward data that is processed by an SQL rule to a topic. This implements machine-to-machine (M2M) communication or other communication scenarios.

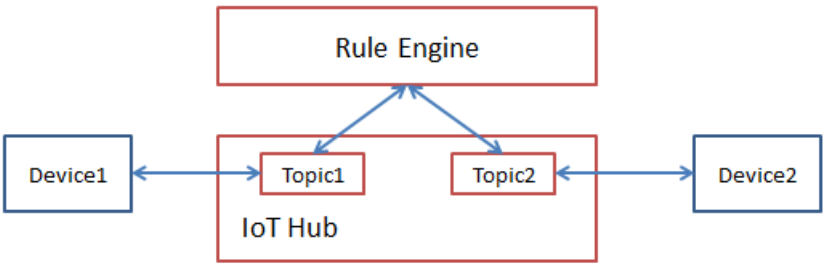
Prerequisites

A data forwarding rule is created and an SQL statement that is used to process data is written. For more information, see Configure a data forwarding rule.

Context

The data forwarding feature of the rules engine allows you to forward data from Topic 1 to Topic 2.

The following figure shows the data forwarding process.

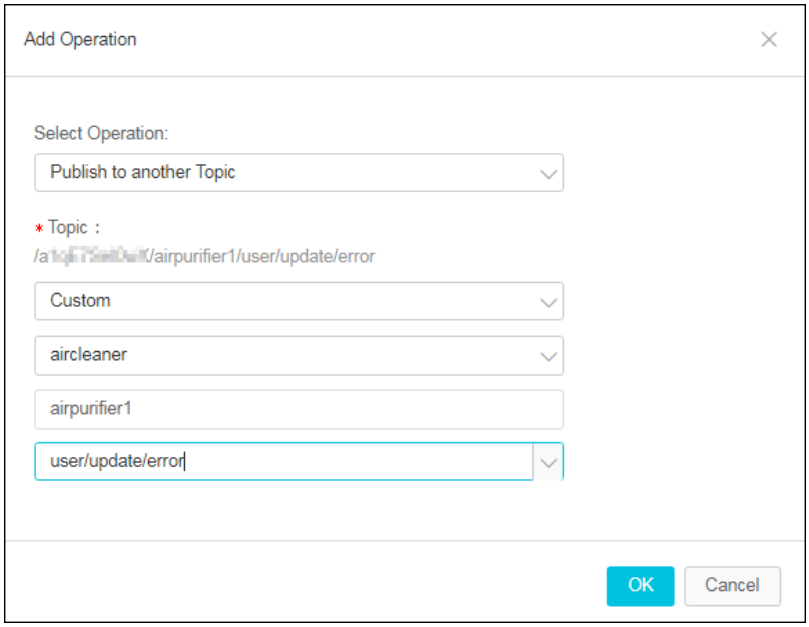


Procedure

1. Log on to the IoT Platform console.
2.
3. In the left-side navigation pane, choose **Rules Engine > Data Forwarding**.
4. Click **View** next to the rule that you want to manage. The **Data Forwarding Rule** page appears.

   > **Notice** If the new version of the Data Forwarding page is displayed, you must click **Back to Previous Version** in the upper-right corner, and then click **View** in the Actions column corresponding to the rule that you want to manage.

5. In the **Forward Data** section, click **Add Operation**.
6. In the **Add Operation** dialog box, select **Publish Data to Another Topic** from the Operation drop-down list. Follow the instructions on the page to set other parameters and click **OK**.



Parameter	Description
Operation	Select **Publish Data to Another Topic**.
Topic	Select the destination topic to which data is forwarded. Valid values: ○ Custom: specifies a custom topic as the destination topic. You must set the permission of the custom topic to **subscribe**. This allows devices to subscribe to the topic and obtain forwarded messages from the topic. ○ Send Downstream TSL Data: specifies a topic that is used by devices to receive commands as the destination topic. These commands are used to set device properties. By default, the topic is `thing/service/property/set`. A device receives forwarded data from the topic. Then, the device sets the required property based on the received data. You can set the Topic parameter to this value when you want to specify the properties for devices of the destination topic based on forwarded data. After you select a topic type, you must select a product, device, and topic.

7. Go to the **Data Forwarding** page, find the rule that you managed, and then click **Start** in the Actions column of the rule.

Use the rules engine to establish M2M communication

4.8.2. Forward data to an AMQP consumer group

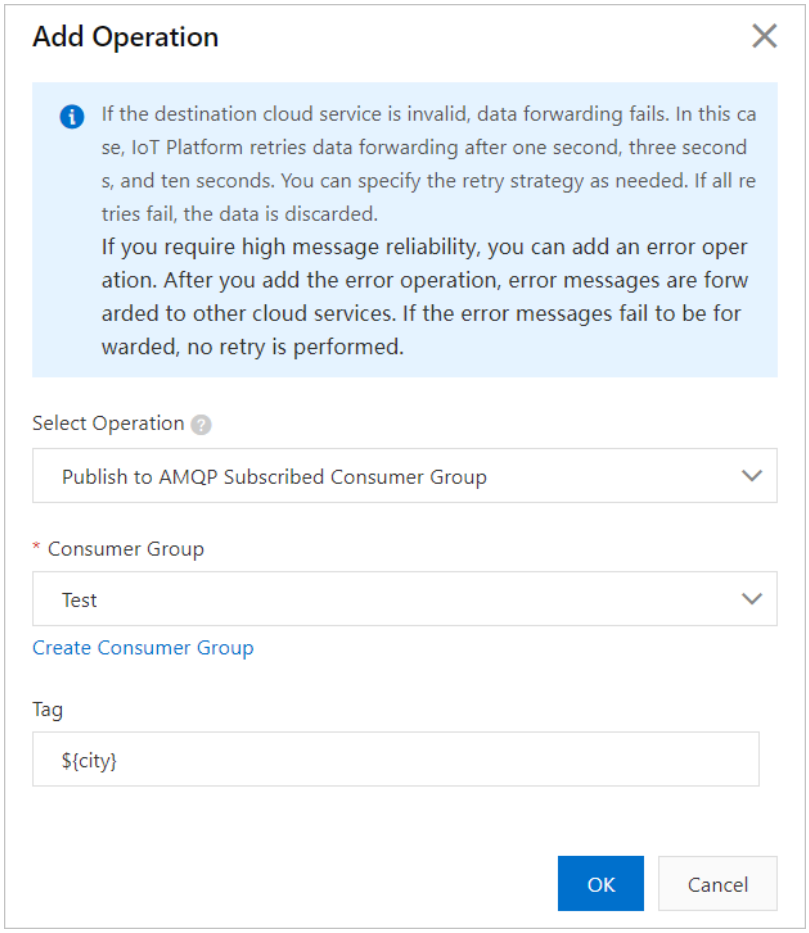
You can use the rules engine to forward messages that are sent by devices to IoT Platform. These messages can be processed by using SQL statements, forwarded to Advanced Message Queuing Protocol (AMQP) consumer groups, and then consumed by AMQP clients.

Prerequisites

- A data forwarding rule is created and an SQL statement that is used to process data is written. For more information, see Configure a data forwarding rule.
- An AMQP consumer group is created and used as the data forwarding destination. For more information about how to create and manage consumer groups, see Manage consumer groups.

Procedure

1.
2.
3.
4.
5.
6. In the **Add Operation** dialog box, select **Publish to AMQP Subscribed Consumer Group**. Set the required parameters as prompted and click **OK**.



Parameter	Description
Select Operation	Select **Publish to AMQP Subscribed Consumer Group**.
Consumer Group	Select an existing consumer group as the data forwarding destination. Click **Create Consumer Group** to create a consumer group.
Tag	If you set a tag, the tag will be added to all messages that are forwarded by this operation to the AMQP consumer group. The tag must be 1 to 128 characters in length. You can specify a constant or a variable. ○ A constant can contain letters and digits. ○ If you specify a variable, use the format of ${key}. ${key} indicates that the variable references the value of a key in the JSON result of an SQL statement. If the value is unavailable, a tag is not added to the messages.

7.

Configure an AMQP client to consume messages

After the data is forwarded to the AMQP consumer group, your server consumes the messages by using the AMQP client. For more information about how to configure an AMQP client, see Connect an AMQP client to IoT Platform.

For sample code of the AMQP client implementation, see the following topics:

- Connect a client to IoT Platform by using the SDK for Java
- Connect a client to IoT Platform by using the SDK for .NET
- Connect a client to IoT Platform by using the SDK for Node.js
- Connect a client to IoT Platform by using the SDK for Python 2.7
- Connect a client to IoT Platform by using the SDK for Python 3
- Connect a client to IoT Platform by using the SDK for PHP
- Connect a client to IoT Platform by using the SDK for Go

4.8.3. Forward data to Message Queue for Apache RocketMQ

You can use the rules engine to forward data from IoT Platform to Message Queue for Apache RocketMQ. This ensures reliable end-to-end transmission from devices, IoT Platform, and Message Queue for Apache RocketMQ to application servers.

Prerequisites

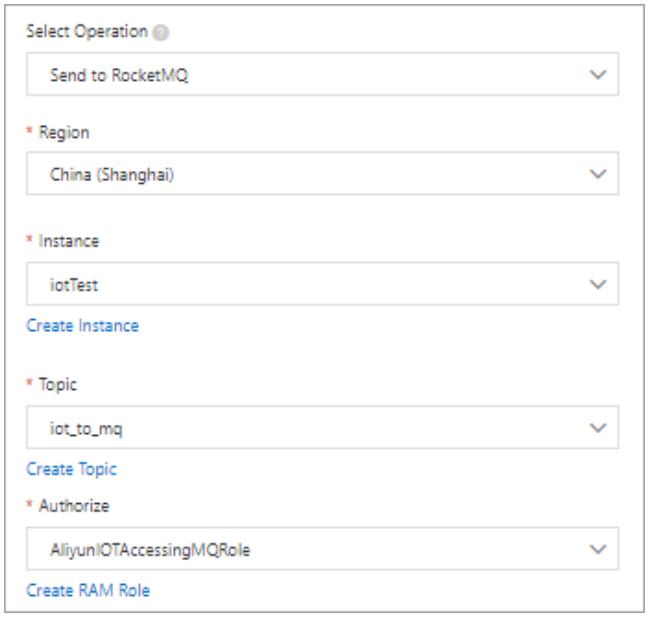
- A Message Queue for Apache RocketMQ instance and a topic that is used to receive data are created. For more information, see What is Message Queue for Apache RocketMQ?.

  > **Notice** The Message Queue for Apache RocketMQ instance must reside in the same region as the IoT Platform instance on which you want to configure a data forwarding rule. This limit does not apply to the existing data forwarding rules that are used to forward data across instances in different regions.

- A data forwarding rule is created and an SQL statement that is used to process data is written. For more information, see Configure a data forwarding rule.

Procedure

1.
2.
3.
4.
5.
6. In the **Add Operation** dialog box, select **Send to RocketMQ** from the Select Operation drop-down list. Configure the parameters and click **OK**.



Parameter	Description
Select Operation	Select **Send to RocketMQ**.
Region	Select the region where the Message Queue for Apache RocketMQ instance is deployed.
Instance	Select the Message Queue for Apache RocketMQ instance. You can click **Create Instance** to go to the Message Queue for Apache RocketMQ console and create a Message Queue for Apache RocketMQ instance. For more information, see Message Queue for Apache RocketMQ documentation.
Topic	Select the Message Queue for Apache RocketMQ topic that is used to receive data from IoT Platform. You can click **Create Topic** to go to the Message Queue for Apache RocketMQ console and create a Message Queue for Apache RocketMQ topic.
Tag	Optional. Specify a tag. If you specify a tag, the tag is attached to all messages that are forwarded to the Message Queue for Apache RocketMQ topic. You can filter messages by tag on your consumer client. The maximum length of a tag is 128 bytes. You can enter a constant or a variable. Enter variables in the ${key} format. ${key} specifies that the variable references the value of a key in the JSON result of an SQL statement.
Authorize	Grant IoT Platform the permissions to write data to Message Queue for Apache RocketMQ. If no RAM roles exist, click **Create RAM Role** to go to the Resource Access Management (RAM) console, create a RAM role, and then grant permissions to the RAM role. For more information, see Create a RAM role.

7.
8. Perform a test.

   Publish a message to the topic defined in the SQL statement that is configured in the data forwarding rule. Then, go to the Message Queue for Apache RocketMQ console to check whether the message is received.

4.8.4. Forward data to Tablestore

You can use the data forwarding feature of the rules engine to forward data to Tablestore.

Prerequisites

- A Tablestore instance and a table that is used to receive data are created. For more information, see the Tablestore documentation.

  > **Notice** If you use an IoT Platform instance of the Enterprise Edition, the region where the Tablestore instance is deployed must be the same as that of the IoT Platform instance.

- A data forwarding rule is created and an SQL statement that is used to process data is written. For more information, see Configure a data forwarding rule.

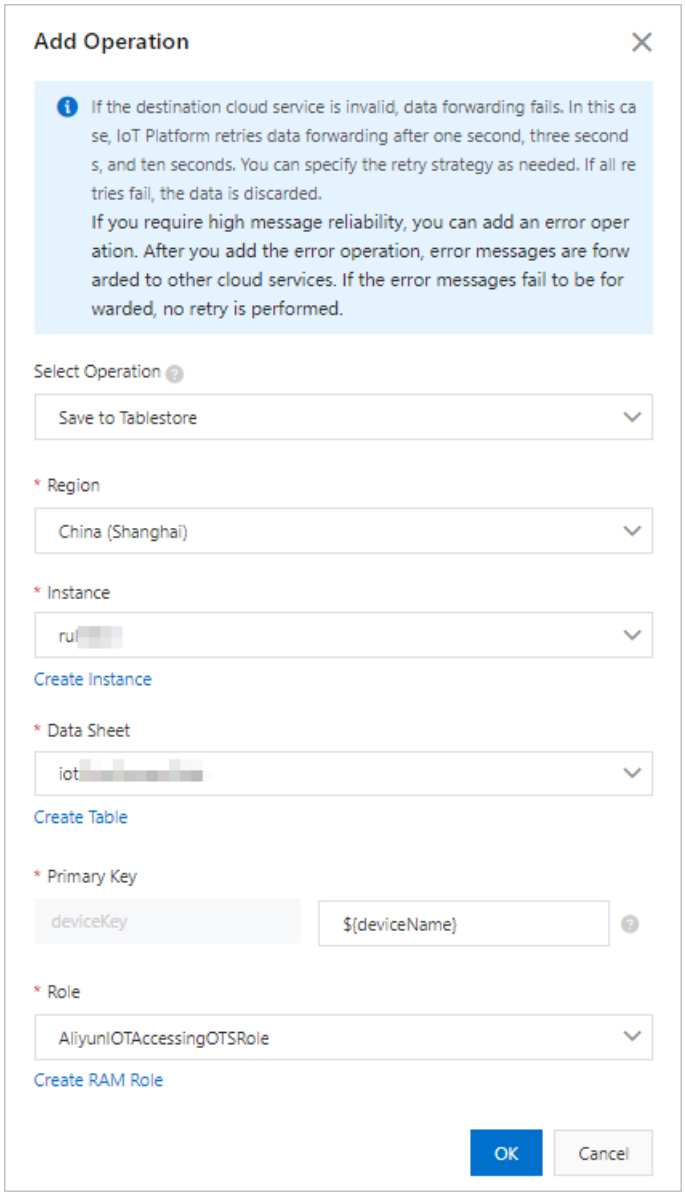
  In the data forwarding rule that is used in this example, the following SQL statement is defined:

```
SELECT items.temperature.value as temperature, items.humidity.value as humidity,deviceNam
e() as deviceName
FROM "/sys/a1ktuxe****/BZoyHO***/thing/event/property/post"
```

Procedure

1.
2.
3.
4.
5.
6. In the **Add Operation** dialog box, follow the instructions on the page to set parameters and click **OK**.

   > **Note** You can forward data only in the JSON format.



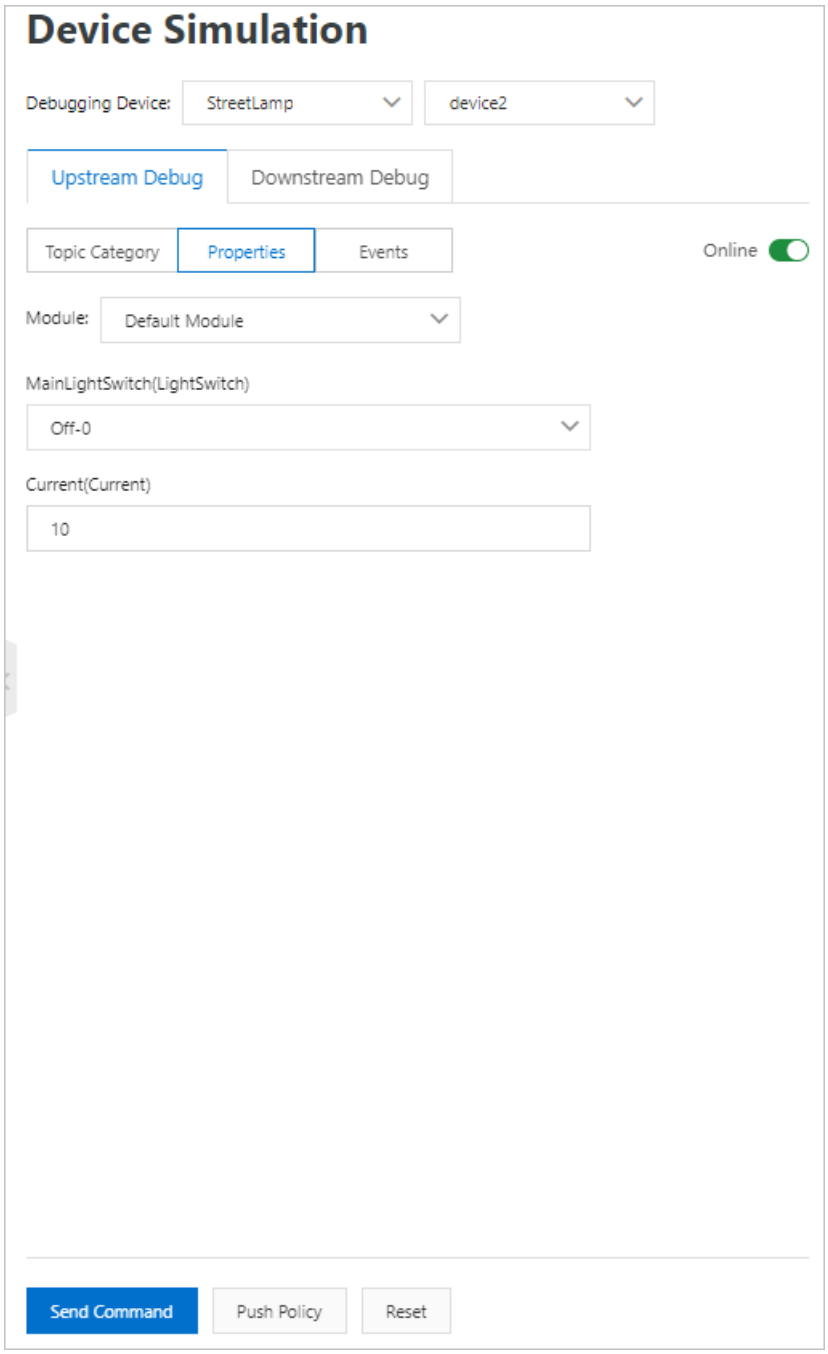
Parameter	Description
Operation	Select **Store Data to Tablestore**.
Region	Select a region where your Tablestore instance is deployed.
Instance	Select a Tablestore instance. You can click **Create Instance** to go to the Tablestore console and create a Tablestore instance. For more information, see the Tablestore documentation.
Data Table	Select a Tablestore table that is used to receive data. You can click **Create Data Table** to go to the Tablestore console and create a Tablestore table.

Parameter	Description
Primary Key	Specify a value for the primary key of the table. You must set this value to a field value that is specified in the SELECT statement configured in the data forwarding rule. When data is forwarded, this value is saved as the value of the primary key. **Note** ◦ You can set this parameter in the `${}` format. For example, set the timestamp to ${deviceName}. ${deviceName} specifies that the value of the primary key is the value of the `deviceName` parameter in a message. ◦ If the primary key corresponds to an **auto-increment column**, you do not need to specify a value for the primary key. Tablestore automatically generates values for this primary key column. By default, the value of an auto-increment primary key column is set to `AUTO_INCREMENT`. You cannot modify the value. For more information about auto-increment primary key columns, see Auto-increment of primary key columns.
Role	Grant IoT Platform the permissions to write data to Tablestore. If no RAM roles exist, click **Create RAM Role** to go to the Resource Access Management (RAM) console, create a RAM role, and then grant permissions to the RAM role. For more information, see Create a RAM role.

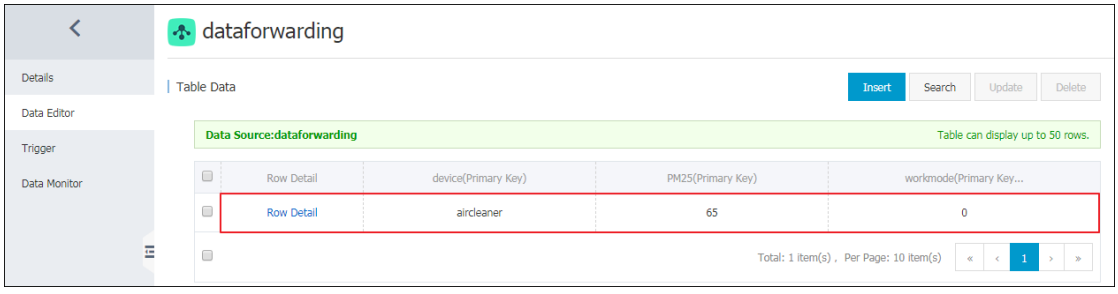
7.

8. Perform a test.
   i. Log on to the IoT Platform console.
   ii. In the left-side navigation pane, choose **Maintenance > Device Simulation**.
   iii. Select the required device, and click **Upstream Debug**, **Properties**, and **Start Device Simulation** one after one.

iv. Below **Default Module**, set test data and click **Send Command**.



v. After the data is forwarded, log on to the Tablestore console. Go to the **Query Data** page of the table that is used to receive data and check whether the specified data is received.



4.8.5. Forward data to ApsaraDB RDS

You can use the data forwarding feature of the rules engine to forward processed data to ApsaraDB RDS instances for storage. This article describes how to forward data to a destination.

Prerequisites

- An ApsaraDB RDS instance is created in the region where your IoT Platform instance resides. The database engine of the instance must be MySQL or Microsoft SQL Server and the network type of the instance must be VPC. A database and table are created. For more information about how to use ApsaraDB RDS, see the ApsaraDB RDS documentation.
- A data forwarding rule is created and an SQL statement that is used to process data is written. For more information, see Configure a data forwarding rule.

Limits

- Data forwarding is supported between IoT Platform instances and ApsaraDB RDS instances that reside in the same region. For example, you can forward the data of an IoT Platform instance in the China (Shanghai) region to an ApsaraDB RDS table that resides only in the China (Shanghai) region.
- You can forward data to ApsaraDB RDS instances that resides only in VPCs.
- You can forward data to MySQL and SQL Server instances.
- Data forwarding is supported between standard databases and privileged databases.
- You can forward data only in the JSON format.

Procedure

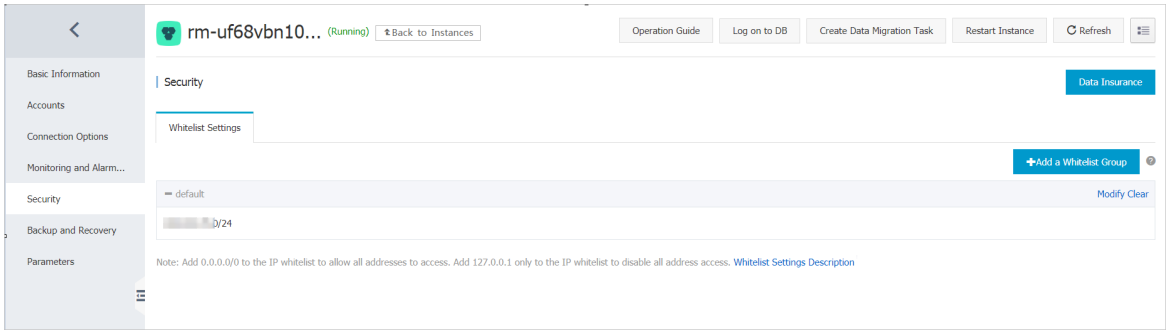
1.
2.
3.
4.
5.
6. In the **Add Operation** dialog box, select **Save to RDS** in the Select Operation field. Set the parameters and click **OK**.

Parameter	Description
Select Operation	Select **Save to RDS**.
Select a region	Select the region where your IoT Platform instance resides.
RDS instance	Select the ApsaraDB RDS instance.
Database management	Enter the database name. **Note** If you use a privileged database, you must enter a database name.

Parameter	Description
Account	Enter the account of the ApsaraDB RDS instance. This account must have the read and write permissions on the database. Otherwise, the rules engine cannot write data to the ApsaraDB RDS instance. **Note** After the rules engine obtains the account, the rules engine writes only the data that matches the specified rule to the database. No extra operations are performed.
Password	Enter the password that is used to log on to the ApsaraDB RDS instance.
Table Name	Enter the name of the table that is created in the database. The rules engine writes data to the table.
Key	Enter the name of the field in the table. The rules engine writes data to the field.
Value	Enter the value of the field in the topic of the SQL statement as the value of the key. **Note** ◦ The data type of the key must be the same as the data type of the value. Otherwise, you may fail to save this parameter. ◦ You can enter a variable, for example, `${deviceName}`.
Role	Authorize IoT Platform to write data to ApsaraDB RDS. If no RAM roles exist, click **Create RAM Role** to go to the Resource Access Management (RAM) console, create a RAM role, and then grant permissions to the RAM role. For more information, see Create a RAM role.

7.

8. Log on to the ApsaraDB RDS console. On the **Data Security** page, configure a whitelist or view whitelists. After you set the parameters, the rules engine adds the following IP addresses to the ApsaraDB RDS whitelist. Then, the rules engine can communicate with ApsaraDB RDS. If one of the following IP addresses is not displayed in the whitelist, you must manually add the IP address.
   - China (Shanghai): `100.104.53.192/26,100.104.148.64/26,100.104.6.192/26,100.104.143.128/26,100.104.76.0/24,100.104.73.128/26,100.104.200.64/26,100.104.40.64/26,100.104.3.0/26,100.104.29.128/26,100.104.121.0/26,100.104.84.64/26`
   - Singapore (Singapore): `100.104.106.0/24`
   - US (Silicon Valley): `100.104.8.0/24`
   - US (Virginia): `100.104.133.64/26`
   - Germany (Frankfurt): `100.104.160.192/26`
   - Japan (Tokyo): `100.104.160.192/26`



4.8.6. Forward data to Message Service

You can use the data forwarding feature of the rules engine to forward data to a Message Service (MNS) topic. A server subscribes to messages of the topic. This way, high-performance message transmission is implemented between the server and devices.

Prerequisites

- An MNS topic is created. A subscription whose Push Type parameter is set to HTTP or Queue is created for the topic. For more information, see the MNS documentation.

  > **Notice** If you use an IoT Platform instance of the Enterprise Edition, the region where the MNS topic resides must be the same as that of the IoT Platform instance.

- A data forwarding rule is created and an SQL statement that is used to process data is written. For more information, see Configure a data forwarding rule.

Data forwarding procedure

- A device sends data to a server.

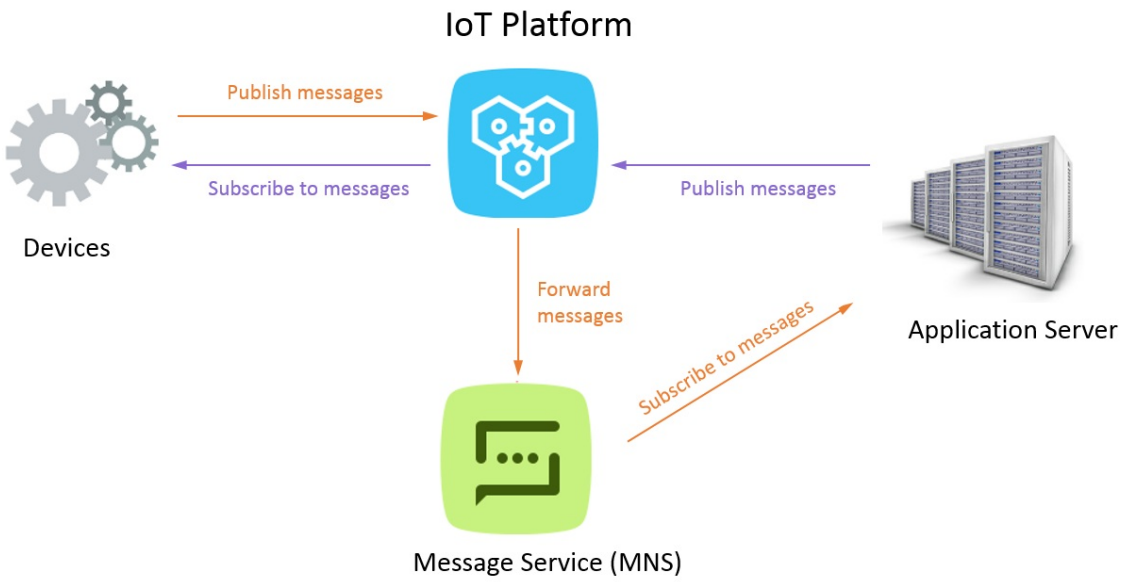
  The device publishes messages to IoT Platform. IoT Platform uses the rules engine to process these messages and forward these messages to an MNS topic. The server then calls an MNS operation to subscribe to messages in the topic.

  MNS is used to ensure the availability of messages. This prevents data loss when the server is unavailable. In addition, when a large number of concurrent messages are processed, MNS allows you to balance the workloads of the server. This prevents service unavailability that is caused by an abrupt increase in workloads. IoT Platform and MNS are used together to implement high-performance message transmission between the server and the device.

- The server sends data to the device.

  The server calls an IoT Platform operation to publish messages to IoT Platform. Then, the device subscribes to the messages of IoT Platform.

The following figure shows the data forwarding procedure.



Procedure

1.
2.
3.
4.
5.
6. In the **Add Operation** dialog box, select **Send Data to MNS** from the Operation drop-down list. Set the parameters and click **OK**.

Parameter	Description
Operation	Select **Send Data to MNS**.
Region	Select a region where MNS runs.
Topic	Select an MNS topic that is used to receive data. MNS sends received messages to a subscription of the topic. The Push Type parameter value of the subscription is HTTP or Queue. You can subscribe to messages that IoT Platform pushes to MNS through only the HTTP or queue method. To create an MNS topic and subscription in the MNS console, click **Create Topic**. For more information, see the MNS documentation.
Role	Authorize IoT Platform to write data to MNS. If no RAM roles exist, click **Create RAM Role** to go to the Resource Access Management (RAM) console, create a RAM role, and then grant permissions to the RAM role. For more information, see Create a RAM role.

7.

4.8.7. Forward data to Lindorm Time Series Database

You can configure data forwarding rules to forward processed data to Lindorm Time Series Database (TSDB) for storage.

Prerequisites

- A TSDB instance that is deployed in a virtual private cloud (VPC) is created in the Japan (Tokyo) region. For more information, see TSDB documentation.
- A data forwarding rule is created and an SQL statement that is used to process data is written. For more information, see Configure a data forwarding rule.

Limits

- Data can be forwarded only within a region. For example, you can only forward data from an IoT Platform instance that is deployed in the Japan (Tokyo) region to a TSDB instance that is deployed in the Japan (Tokyo) region.
- You can forward data only to a TSDB instance that is deployed in a VPC.
- You can forward data only in the JSON format.
- Only TSDB is supported. Time Series Database for InfluxDB® and Ganos are not supported.
- In the forwarded messages, all fields are written to TSDB as the metric, except for the fields that are configured as the timestamp and tag value. The data types of the metric parameter support numeric and string. If you use other data types, data may fail to be written to the database.

Procedure

1.
2.
3.
4.
5.
6. In the **Add Operation** dialog box, select **Store Data to TSDB** from the Operation drop-down list. Follow the instructions on the page to set other parameters and click **OK**.

Parameter	Description
Operation	Select **Store Data to TSDB**.
Region	The region where your IoT Platform instance is deployed: Japan (Tokyo)
TSDB Instance	Specify the TSDB instance in the VPC that you have created as the destination to which data is forwarded.

Parameter	Description
metric Data Type	Specify the data type of the metric parameter. Valid values: **Numeric Type** and **String**. Click the icon to obtain more information.
timestamp	Timestamp. The following configurations are supported: ○ Use an escape character expression `${}`. For example, set the parameter to `${time}`. This specifies that the parameter value corresponds to the value of the time field in the message in the data source topic. ○ Use the rules engine function `timestamp()`. This specifies that the parameter value is set to the timestamp of the rules engine server. ○ Enter a value. The input value must be a UNIX timestamp. For example, you can enter 1404955893000.
Tag Name	Specify the tag name that is used to identify data. The tag name can contain letters, digits, and the following special characters: :,.'/-_()[]
Tag Value	Specify the value of the tag. The following configurations are supported: ○ Use an escape character expression `${}`. For example, if the structure of a message in a data source topic contains a location property whose key is city, you can set the tag value to `${city}`. This specifies that the parameter value corresponds to the value of the city field in the message. We recommend that you use this configuration. ○ Use the functions specified in the rules engine, such as `deviceName()`. This specifies that the tag value is the name of the device. For more information about the supported functions, see Functions. ○ Enter a constant. For example, you can enter the constant beijing. The value can contain letters, digits, and the following special characters: :,.'/-_()[] **Note** ○ You can add a maximum of eight tag names and eight tag values. ○ Make sure that TSDB can obtain the configured tag names and tag values. If TSDB cannot obtain the name or value of a tag, the data cannot be written to the database.
Role	Grant IoT Platform the permissions to write data to TSDB.

7.

Data forwarding examples

The following code block provides a sample SQL statement in the data forwarding rule:

```
SELECT time,city,power,distance FROM "/alprodu****/myDevice/user/update";
```

The following section shows how the rules engine processes data and writes data to TSDB based on the SQL statement.

1. The sample SQL statement shows that the rules engine filters the time, city, power, and distance fields from the messages in the `/alprodu****/myDevice/user/update` topic and uses the content of the filtered fields as the content of the message to forward.

   The following example shows the content of the message to forward after data is processed based on the preceding SQL statement:

   ```
   {
   "time": 1513677897,
   "city": "beijing",
   "distance": 8545,
   "power": 93.0
   }
   ```

2. The rules engine writes two rows of data to TSDB based on the configured data forwarding operation.

   In the example, the following data is written to TSDB:

   ```
   Data: timestamp:1513677897, [metric:power value:93.0]
   tag: cityName=beijing
   ```

   ```
   Data: timestamp:1513677897, [metric:distance value:8545]
   tag: cityName=beijing
   ```

   The following section describes how data is written to TSDB:

   In the preceding forwarded messages, except for the time field that is configured as the timestamp and the city field that is configured as the tag value, other fields (power and distance) are written to TSDB as the metric.

4.8.8. Forward data to Function Compute

You can use the data forwarding feature of the rules engine to forward data to Function Compute. Then, Function Compute runs function scripts to process data.

Prerequisites

- A Function Compute service and a function are created. The function is configured and can run as expected after verification. For more information, see the Function Compute documentation.

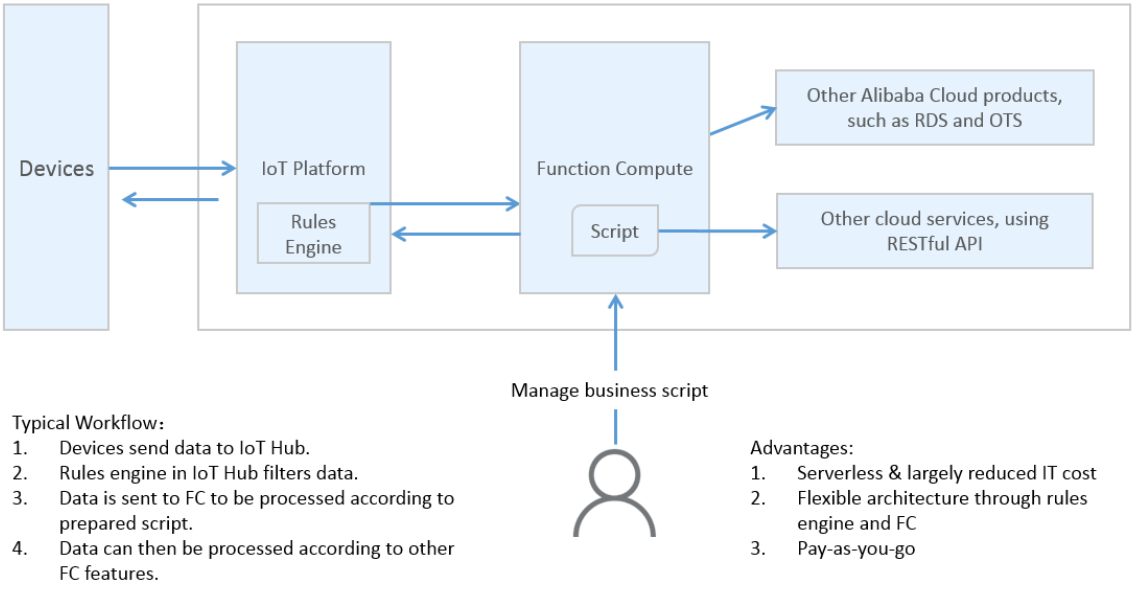
  > **Notice** If you use an IoT Platform Enterprise Edition instance, Function Compute must be deployed in the region where the Enterprise Edition instance resides.

- A data forwarding rule is created and an SQL statement that is used to process data is written. For more information, see Configure a data forwarding rule.

Context

The rules engine uses the data forwarding feature to forward device data to Function Compute. Then, Function Compute runs business-specific function scripts to implement various business features.

The following figure shows the data forwarding process.



Procedure

1.
2.
3.
4.
5.
6. In the **Add Operation** dialog box, select **Send Data to Function Compute** from the Operation drop-down list. Follow the instructions as prompted to configure other parameters and click **OK**.

Parameter	Description
Operation	Select **Send Data to Function Compute**.
Regions	Select the region in which your Function Compute service is deployed.
Services	Select a Function Compute service. You can click **Create Service** to go to the Function Compute console and create a service. For more information, see Manage services.

Parameter	Description
Function Version	Valid values: ○ **Use Default Version**: Use the default version of the Function Compute service. The default version is LATEST. ○ **Select Version**: Select the version of the Function Compute service that you want to release. You can click **Create Version** to go to the Function Compute console and create a version. For more information, see Manage versions. ○ **Select Alias**: Select the alias that you want to specify for the version of the Function Compute service. You can click **Create Alias** to go to the Function Compute console and create an alias. For more information, see Manage aliases.
Function	Select the function that you want to use to receive data. You can click **Create Function** to go to the Function Compute console and create a function. For more information, see Manage functions.
Authorize	Grant IoT Platform the permissions to write data to Function Compute. If no RAM roles exist, click **Create RAM Role** to go to the Resource Access Management (RAM) console, create a RAM role, and then grant permissions to the RAM role. For more information, see Create a RAM role.

7.
8. Perform a test.
    i. Send messages to the destination topic based on the topic category that is defined in the SQL statement that is specified in the data forwarding rule. For information about how to use topics, see What is a topic?. For more information about how to debug upstream or downstream messaging between devices and IoT Platform, see Device simulation.
    ii. Log on to the Function Compute console. On the details page of the function, the **Logs** tab displays the execution logs of the function. In the upper-right corner of the details page, click **Monitoring Dashboard** to view the monitoring statistics of the function.

> **Notice** The monitoring statistics of a function are generated 5 minutes after the function is executed.

Examples

Push device data to DingTalk groups

5.Data Forwarding v2.0

5.1. Overview

The data forwarding feature of IoT Platform allows you to forward data from a topic to other topics or Alibaba Cloud services for storage or processing.

Background information

Compared with Data Forwarding v1.0, Data Forwarding v2.0 provides a parser that can process complex data and achieve interactions with other cloud services. When you configure a parser, you can add multiple topics to the data source from which data is forwarded to other topics or cloud services.

For more information about data forwarding, see What is data forwarding?.

Features

Feature	Description
Create a data source	Add topics to the data source from which data is forwarded. You can add multiple topics to a data source.
Create a data destination	Add topics or cloud services to the data destination to which data is forwarded.
Configure a parser	Create a parser, associate the parser with the created data source and data destination, and then write the parser script to implement data forwarding.

Limits

Item	Description	Limit
Parser	The maximum number of parsers that an instance can contain.	1,000
Data Sources	The maximum number of data sources that can be associated with a parser.	1
	The maximum number of topics that a data source can contain.	1,000
Data destination	The maximum number of data destinations that can be associated with a parser.	10
	The maximum number of operations that a data destination can contain.	1
	The maximum number of error data destinations that can be associated with a parser.	1
	The maximum size of a parser script.	120 KB

Item	Description	Limit
Parser script	The maximum number of times that a data forwarding function can be executed in a loop in a parser script. For more information about data forwarding functions, see Forward data to destinations.	100

Procedure

1. Create a data source.
2. Create a data destination.
3. Configure and start a parser.

References

- Script syntax: describes the syntax of a parser script.
- Functions: describes the functions that can be used in a parser script.
- Data formats: describes the data formats of basic communication topics and TSL-based communication topics. A TSL model is used to parse the data that is sent to a topic. When you write a parser script, you must specify fields based on the parsed data format.

5.2. Configure a data forwarding parser

5.2.1. Create a data source

You can use the rules engine of IoT Platform to forward data from specified topics to other topics or other Alibaba Cloud services. This article describes how to add topics to a data source.

Context

When you configure a data forwarding parser, you must associate the parser with topics from which data is forwarded. You can add a maximum of 1,000 topics to each data source.

You can use the `topic(n)` function in a script to obtain the topic from which data is forwarded. For more information, see Functions supported by the data forwarding feature.

Create a data source

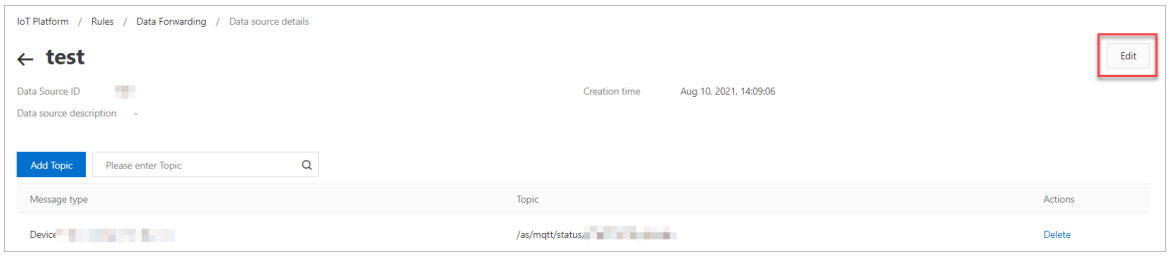
1.
2.
3.
4. On the **Data Forwarding** page, click **Go to New Version** in the upper-right corner to go to the new version.

> **Note** If you have performed this step, the **Data Forwarding** page of the new version appears after you choose Rules > Data Forwarding.

5. Click the **Data Source** tab. On this tab, click **Create Data Source**.
6. Enter a data source name and description, and then click **OK**.

   The name must be 1 to 30 characters in length, and can contain letters, digits, underscores (_), and hyphens (-).

   The **Data Source Details** page appears. You can click **Edit** in the upper-right corner to modify the name and description of the data source.



7. On the **Data Source Details** page, click **Add Topic**. In the dialog box that appears, select the topic that you want to process, and then click **OK**.

   Valid values:Topics

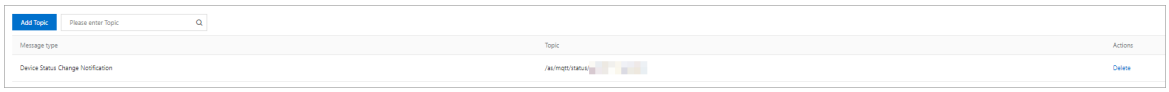
Topic	Description	References
Custom	The topic that is used to forward data of custom formats. The format of this topic is the same as the format of a custom topic. Format: `/${productKey}/${deviceName}/user/${TopicShortName}`. `${TopicShortName}` specifies a custom topic category, which indicates the suffix of the custom topic. The value can contain wildcard characters, including plus signs (+) and number signs (#). ○ All equipment (`+`): indicates all devices of the specified product. ○ `/user/#` : indicates all topics of the specified device.	Custom topics
Device Status Change Notification	The topic that is used to forward notifications when the status of a device changes between online and offline. Format: `/as/mqtt/status/${productKey}/${deviceName}`.	Submit device status

Topic	Description	References
TSL Data Reporting	The following topics are provided: ○ `/${productKey}/${deviceName}/thing/event/property/post` : This topic is used to forward device properties. ○ `/${productKey}/${deviceName}/thing/event/${tsl.event.identifier}/post` : This topic is used to forward device events. ○ `/${productKey}/${deviceName}/thing/event/property/post` : This topic is used to forward multiple device properties at a time. ○ `/${productKey}/${deviceName}/thing/event/batch/post` : This topic is used to forward multiple device events at a time. ○ `/${productKey}/${deviceName}/thing/downlink/reply/message` : This topic is used to forward messages that are sent by a device as responses to IoT Platform commands.	○ Submit device properties ○ Submit device events ○ Submit device properties in batches ○ Submit device events in batches ○ Submit responses to downstream requests
	The preceding topics correspond to the following device topics: ○ `/sys/{productKey}/{deviceName}/thing/event/property/post` : This topic is used to submit device properties. ○ `/sys/${productKey}/${deviceName}/thing/event/${tsl.event.identifier}/post` and `/sys/${productKey}/${deviceName}/thing/event/${tsl.functionBlockId}:{tsl.event.identifier}/post` : These topics are used to submit device events. ○ `/sys/${productKey}/${deviceName}/thing/event/property/batch/post` : This topic is used to submit device properties and events in batches.	○ Devices submit property information to IoT Platform ○ Devices submit events to IoT Platform ○ Devices submit multiple properties and events to IoT Platform at a time
Device Changes Throughout Lifecycle	The topic that is used to forward notifications when a device is created, deleted, disabled, or enabled. Format: `/${productKey}/${deviceName}/thing/lifecycle`	Submit lifecycle changes
Sub-Device Data Report Detected by Gateway	The topic that is used to submit and forward the information about a new sub-device when a gateway detects the sub-device. This topic is specific to gateways. Format: `/${productKey}/${deviceName}/thing/list/found`	Submit information about detected sub-devices
Device Topological	The topic that is used to forward notifications when topological relationships between sub-devices and the gateway are created or deleted. This topic is specific to gateways. Format: `/${productKey}/${deviceName}/thing/topo/lifecycle` .	Submit topology changes

Topological Relation Changes	Description	References
	The preceding topic corresponds to the following topic: `/sys/{productKey}/{deviceName}/thing/topo/change`. This topic is used to submit device data.	Notify gateways of changes of topological relationships
Device tag change	The topic that is used to forward notifications when device tags are changed. Format: `/${productKey}/${deviceName}/thing/deviceinfo/update`.	Submit device tag changes
	The preceding topic corresponds to the following topic: `/sys/{productKey}/{deviceName}/thing/deviceinfo/update`. This topic is used to submit device data.	Report tags
TSL Historical Data Reporting	The following topics are provided: ○ `/${productKey}/${deviceName}/thing/event/property/history/post`: This topic is used to forward historical properties. ○ `/${productKey}/${deviceName}/thing/event/${tsl.event.identifier}/history/post`: This topic is used to forward historical events.	○ Submit historical properties ○ Submit historical events
	The preceding topics correspond to the following topic: `/sys/{productKey}/{deviceName}/thing/event/property/history/post`. This topic is used to submit historical TSL data.	Devices submit historical TSL data to IoT Platform
Device status notification	The following topics are provided: ○ `/${productKey}/${deviceName}/ota/upgrade`: This topic is used to forward over-the-air (OTA) update results. ○ `/${productKey}/${deviceName}/ota/progress/post`: This topic is used to forward update progresses.	○ Submit the status data of over-the-air (OTA) updates ○ Submit the progress data of OTA updates
	The preceding topics correspond to the following topic: `/ota/device/progress/${YourProductKey}/${YourDeviceName}`. This topic is used to submit update progresses.	Submit the update progress to IoT Platform
Submit a module version number	The topic that is used to forward notifications when the version number of an OTA module for a device is changed. Format: `/${productKey}/${deviceName}/ota/version/post`.	Submit OTA module versions
	The preceding topic corresponds to the following topic: `/ota/device/inform/${YourProductKey}/${YourDeviceName}`. This topic is used to submit the version number of an OTA module.	Submit OTA module versions to IoT Platform

Topic	Description	References
Batch status notification	The topic to which IoT Platform sends notifications when the status of an OTA update batch changes. Format: `/${productKey}/${packageId}/${jobId}/ota/job/status`.	Submit the status data of OTA update batches

8. (Optional)Reperform the previous step to add multiple topics to the data source.
   You can delete the added topics on the **Data Source Details** page based on your business requirements.



9. (Optional)Reperform Step 5 to 8 to add multiple data sources.

What's next

Create a data destination

5.2.2. Create a data destination

You can use the rules engine of IoT Platform to forward data from specified topics to other topics or other Alibaba Cloud services.This article describes how to create a data destination.

Context

When you configure a data forwarding parser, you must associate the parser with a data destination to which normal or error data is forwarded. You can specify only one data forwarding operation for each data destination.

You can use functions in the parser script to forward data to other topics or cloud services. For more information, see Forward data to destinations.

Create a data destination

1.
2.
3.
4. (Optional)On the **Data Forwarding** page, click **Go to New Version** in the upper-right corner to go to the new version.

   > **Note** If you have performed this step, the **Data Forwarding** page of the new version appears after you choose Rules > Data Forwarding.

5. Click the **Data Destination** tab. On this tab, click **Create Data Destination**.
6. In the **Create Data Destination** dialog box, set the parameters and then click **OK**. The following table describes the parameters.

Parameter	Description
Data Destination Name	Enter a custom name. The name must be 1 to 30 characters in length, and can contain letters, digits, underscores (_), and hyphens (-).
Data Destination Description	Enter a description of the data destination.
Select Operation	Specify a data forwarding destination. For more information, see Data forwarding examples.

7. (Optional)Perform Step 5 and Step 6 to create multiple data destinations.

What's next

Configure a parser

5.2.3. Configure a parser

You can use the rules engine of IoT Platform to forward data from specified topics to other topics or other Alibaba Cloud services.This article describes how to create a parser, associate the parser with a data source and a data destination, and then configure the parser script.

Prerequisites

A data source and a data destination are created. For more information, see the following articles:

- Create a data source.
- Create a data destination.

Create a parser

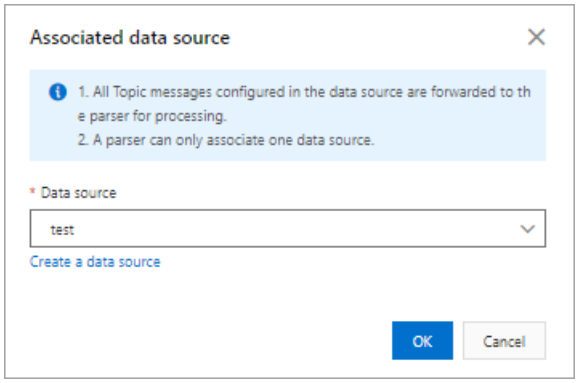
1.
2.
3.
4. (Optional)
5. On the **Parser** tab, click **Create Parser**.
6. Enter a parser name and description, and then click **OK**.

   The name must be 1 to 30 characters in length, and can contain letters, digits, underscores (_), and hyphens (-).

   The **Parser Details** page appears. You can click **Edit** in the upper-right corner to modify the name and description of the parser.

Associate the parser with the data source and data destination

1. On the **Parser Details** page, click **Associate Data Source**.
2. In the dialog box that appears, select the created data source from the **Data Source** drop-down list, and then click **OK**.



3. On the **Parser Details** page, click **Data Destination**.
4. Click **Associate Data Destination**. In the dialog box that appears, select the created data destination from the **Data Destination** drop-down list, and then click **OK**.
5. In the **Error Data Destination** section, click **Associate Data Destination** to specify an error data destination. If retries to forward data fail, error messages are sent to the specified destination.

   > **Notice**
   > - You can add only one error operation for each rule.
   > - A normal operation and an error operation cannot forward messages to the same destination. For example, normal data and error data cannot be forwarded to Tablestore at the same time.
   > - If an error message fails to be forwarded, no retry is performed.
   > - Error messages are generated only if the rules engine fails to forward data due to the issues of other cloud services.

   If a message fails to be forwarded to a cloud service, IoT Platform tries again. If the retry fails, an error message is forwarded based on the configured error operation for data forwarding.

   Sample error message:

```
{
      "ruleName":"",
      "topic":"",
      "productKey":"",
      "deviceName":"",
      "messageId":"",
      "base64OriginalPayload":"",
      "failures":[
        {
          "actionType":"OTS",
          "actionRegion":"cn-shanghai",
          "actionResource":"table1",
          "errorMessage":""
        },
        {
          "actionType":"RDS",
          "actionRegion":"cn-shanghai",
          "actionResource":"instance1/table1",
          "errorMessage":""
        }
      ]
}
```

The following table describes the parameters of error messages.

Parameter	Description
ruleName	The name of the rule.
topic	The source topic of the message.
productKey	The ProductKey of the product.
deviceName	The DeviceName of the device.
messageId	The ID of the message that is sent from IoT Platform.
base64OriginalPayload	The Base64-encoded raw data.
failures	The error details. Multiple errors may occur.
actionType	The type of the failed operation.
actionRegion	The region in which the error occurred.
actionResource	The destination service in which the error occurred.
errorMessage	The error message.

Configure and start a parser

1. On the **Parser Details** page, click **Parser**.
2. In the code editor, enter a script. For more information about how to write a script, see Sample

script.

3. Click **Debugging**. In the dialog box that appears, select a product and a device, specify a topic, and enter payload data to verify whether the script can run properly.
   - **Topic**: The data format of the topic must match the parsing logic of the script.
   - **Payload Data**: The format of the input data must match the data format of the rules engine.
     - If you use a custom topic, the payload data is the original data that is submitted by a device.
     - For information about the data formats of basic communication topics and TSL communication topics, see Data formats.

   The variables declared in the script and the data forwarding functions that are implemented are displayed on the **Result** tab.

   If the debugging is successful, the data is written to the destination cloud service. You can log on to the console of the destination cloud service and view the forwarded data.
4. Click **Publish**.
5. On the **Parser** tab of the **Data Forwarding** page, find the parser and click **Start** in the Actions column. After the parser is started, data is forwarded based on the specified logic of the script.

   You can also perform the following operations:
   - Click **View** to go to the **Parser Details** page. On this page, you can modify the settings of the parser.
   - Click **Delete** to delete the parser.

     > **Note** You cannot delete a parser that is in the Running state.

   - Click **Stop** to stop data forwarding.

Import previous rules

You can import the data forwarding rules that are configured in the previous version, and then reconfigure the data source, data destination, and parser script. To import previous rules, perform the following steps:

1. On the **Parser** tab, click **Import Previous Rules**.
2. In the right-side panel, select the rules that you want to import and click **OK**.
   After the rules are imported, the rules are displayed in the parser list and renamed in the format of `"connector_" +${Previous rule name}` .

   On the **Parser Details** page, you can modify the name of the parser. You can also reconfigure the data source, data destination, and parser script.

5.3. Script syntax

IoT Platform provides a parser that uses similar syntax as JavaScript. Compared with SQL expressions, the parser can process complex data and achieve interactions with other cloud services. The parser is used to obtain message content, convert data formats, and forward data. The parser can process strings, JSON-formatted data, and binary data. This article describes how to write a parser script.

Background information

IoT Platform processes and transmits data based on data formats of topics. For more information about the data formats, see Data formats.

Sample script

The following property data is submitted:

```
{
    "deviceType": "CustomCategory",
    "iotId": "JCp9***",
    "requestId": "1626948228247",
    "checkFailedData": {
    },
    "productKey": "a1o***",
    "gmtCreate": 1626948134445,
    "deviceName": "Device1",
    "items": {
        "Temperature": {
            "value": 38,
            "time": 1626948134319
        },
        "Humidity": {
            "value": 25,
            "time": 1626948134319
        }
    }
}
```

The following script is used to parse and process the submitted data:

```
// Use the payload() function to obtain the data that is submitted by devices and convert t
he data by using the JSON format.
var data = payload("json");
// Filter the submitted temperature and humidity values.
var h = getOrNull(data, "items", "Humidity", "value");
var t = data.items.Temperature.value;
// Set a rule to send data to ApsaraDB RDS if the temperature value is greater than 38.
// An ApsaraDB RDS table includes the following columns: id (auto-increment primary key), d
eviceName, temperature, humidity, and time. You can use the writeRds() method to write valu
es to columns by using the format of column:value.
if (t > 38) {
    writeRds(1000, {"devcieName":deviceName(), "temperature":t, "time":timestamp(), "humidi
ty":h});
}
```

The source data that is parsed and processed must be converted to JSON arrays or nested JSON data.

You can use JSONPath expressions or the `getOrNull()` function in a script to obtain field values. For more information, see LanguageManual UDF and getOrNull().

For example, you can use `getOrNull(data, "items", "Humidity", "value");` to obtain the value `25`, use `data.items.Temperature.value` to obtain the value `38`, and use `data.iotId` to obtain the value `JCp9***`.

Notice The following logic is implemented if the field to query does not exist.

- If you use the function, the value `null` is returned and the script can continue to run.
- If you use a JSONPath expression, a null pointer appears and the script stops running.

Identifiers

You must use identifiers to define constants, variables, and custom fields in the code. Each identifier can contain uppercase and lowercase letters, digits, and underscores (_). It cannot start with a digit.

The following keywords and reserved words cannot be used as identifiers:

- Keywords: `for`, `break`, `continue`, `if`, `else`, `true`, `false`, `var`, `new`, `null`, and `return`.
- Reserved words: `breakdo`, `instanceof`, `typeof`, `case`, `catch`, `finally`, `void`, `switch`, `while`, `debugger`, `function`, `this`, `with`, `default`, `throw`, `delete`, `in`, `try`, `as`, `from`, `classenum`, `extends`, `super`, `const`, `export`, `import`, `await`, `implementslet`, `let`, `private`, `public`, `interface`, `package`, `protected`, `static`, and `yield`.

Data types

The data types of constants, variables, and custom fields in the code can be number, Boolean, string, byte, map, and array.

The value of a constant can be null. The data types of numeric constants can be decimal integer, hexadecimal integer, and floating point.

Process control statements

IoT Platform supports `for` and `if...else` statements. `for` statements support the `break` and `continue` keywords.

Operators

- Logical operators: `&&` and `||`.

  For non-Boolean values that are specified in logical conditions, null indicates false and other values indicate true. For example, `null && "x"` returns false, and `null || "x"` returns true.

- Mathematical operators: `*`, `/`, `%`, `+`, and `-`.

  Only the numeric data type is supported. Otherwise, an error occurs.

- Relational operators: `>`, `=>`, `<`, `<=`, `==`, and `!=`. The equality operator (==) supports only the comparison of key values.

Comments

Multi-line comments (`/* ${comments}*/`) and single-line comments (`// ${comments}`) are supported.

References

For information about available functions, see Functions.

5.4. Functions

IoT Platform provides various functions that can be used in a parser script to process data. You can use the functions to convert data types, forward data to destinations, and process payloads of specific data types. This topic describes the functions that are provided by IoT Platform.

You can use the functions that are supported by Data Forwarding v1.0 in parser scripts of Data Forwarding v2.0. For more information, see Functions supported by the data forwarding feature.

Convert data types

Function	Description
toBoolean(Object)	Converts the Object parameter to a Boolean value. The parameter supports the following data types: • Boolean: The function returns the same value as the parameter. • Number: If the parameter is set to 0, the function returns false. Otherwise, the function returns true. • String: If the parameter is set to `"true"`, the function returns true. Otherwise, the function returns false. • If the parameter is set to null, the function returns false.
toNumber(Object)	Converts the Object parameter to a number. The parameter supports the following data types: • Boolean: If the parameter is set to true, the function returns 1. If the parameter is set to false, the function returns 0. • Number: The function returns the same value as the parameter. • String: The function parses the string to a number. • If the parameter is set to null, the function returns 0.
toString(Object)	Converts the Object parameter to a string. If the parameter is set to a value other than null, the function returns the value as a string. If the parameter is set to null, the function returns an empty string. If the parameter is of the binary type, the function returns a UTF-8 decoded value.
toMap(Object)	Converts the Object parameter to a map. The parameter supports the following data types: • Map: The function returns the same value as the parameter. • String: The function parses the string to a map based on the JSON format. • If the parameter is set to null, the function returns an empty map.

Function	Description
toArray(Object)	Converts the Object parameter to an array. The parameter supports the following data types: • Array: The function returns the same value as the parameter. • String: The function parses the string to an array based on the JSON format. • If the parameter is set to null, the function returns an empty array.
toBinary(Object)	Converts the Object parameter to a binary value. The parameter supports the following data types: • Binary: The function returns the same value as the parameter. • String: The function returns a UTF-8 encoded value. • If the parameter is set to null, the function returns an empty binary value.

Convert time formats

Function	Description
format_date(timestamp, patten, timeZone)	Converts a timestamp in milliseconds to a time in a specified format. A string is returned. • timestamp: the timestamp in milliseconds. • patten: the required time format. Example: `yyyy-MM-dd HH:mm:ss`. • timeZone: the time zone. Examples: GMT, UTC, and CST. If you do not configure this parameter, the time zone of the current system is used.
to_timestamp(dateString, patten, timeZone)	Converts a time in a specified format to a timestamp in milliseconds. A numeric value is returned. • dateString: the time string. • patten: the time format. Example: `yyyy-MM-dd HH:mm:ss`. • timeZone: the time zone. Examples: GMT, UTC, and CST. If you do not configure this parameter, the time zone of the current system is used.

Forward data to destinations

In the following functions, the destinationId parameter specifies the ID of a data destination and the payload parameter specifies the message content.

Function	Description	Examples

Function	Description	Examples
writeAmqp(destinationId, payload, tag)	Forwards data to an Advanced Message Queuing Protocol (AMQP) consumer group. tag: This parameter is optional. If you specify a tag, the tag is added to all messages that are forwarded to the AMQP consumer group. The tag must be 1 to 128 characters in length. You can specify a constant or a variable in the tag. • The constant can contain letters and digits. • The variable specifies the value of a key in the JSON data that is parsed from payloads by using a script. If the value is unavailable, a tag is not added to the messages.	Forward data to an AMQP consumer group
writeFc(destinationId, data)	Forwards data to Function Compute. data: the data that you want to forward to Function Compute.	Forward data to Function Compute
writeMns(destinationId, payload)	Forwards data to Message Service (MNS).	Forward data to Message Service (MNS)
writeMq(destinationId, payload, tag)	Forwards data to Message Queue for Apache RocketMQ. tag: This parameter is optional. If you specify a tag, the tag is added to all messages that are forwarded to Message Queue for Apache RocketMQ. You can filter messages based on tags on your consumer client. The tag must be 1 to 128 characters in length. You can specify a constant or a variable in the tag. • The constant can contain letters and digits. • The variable specifies the value of a key in the JSON data that is parsed from payloads by using a script. If the value is unavailable, a tag is not added to the messages.	Forward data to Message Queue for Apache RocketMQ

Function	Description	Examples
writeTableStore(destinationId, data, flowType)	Forwards data to Tablestore. • data: the data that you want to write to Tablestore. Only the map data type is supported. The key specifies the table column name, and the value specifies the column value. The data parameter must contain the primary key of Tablestore. • flowType: This parameter is optional. It specifies the data type of non-primary fields. ◦ If the value of the flowType parameter is true, the system converts the values of non-primary fields into strings and then forwards the strings to Tablestore. ◦ If the value of the flowType parameter is false, the system forwards the values of non-primary fields to Tablestore without converting the data types.	Forward data to Tablestore
writeRds(destinationId, data)	Forwards data to ApsaraDB RDS. data: the data that you want to write to ApsaraDB RDS. Only the map data type is supported. The key specifies the table column name, and the value specifies the column value.	Forward data to ApsaraDB RDS
writeTsdb(destinationId, timestamp, metricName, value, tag)	Forwards data to ApsaraDB for Lindorm. • timestamp: the timestamp. • metricName: the name of the metric that is stored in Lindorm. • value: the value of the data point that is stored in Lindorm. The string and map data types are supported. • tag: the tag that consists of a key-value pair. The data type is map.	Forward data to Time Series Database (TSDB)

Function	Description	Examples
writeIotTopic(destinationId, topic, payload)	Forwards data to another topic. Topic: the destination topic to which you want to forward data. The following topics are supported: • Custom topic: You must set the permission of the custom topic to subscribe. This allows devices to subscribe to the topic and obtain forwarded messages from the topic. • Topic to which downstream TSL data is sent: `/sys/${productKey}/${deviceName}/thing/service/property/set`. IoT Platform forwards messages to this topic. The messages include commands to configure device properties. Devices receive the forwarded messages from the topic and configure device properties based on the payload of the messages. If you want to specify the properties for devices of the destination topic based on forwarded data, you can set the Topic parameter to this value. The value of the `${productKey}` variable must be the same as the ProductKey of the product that you specified in the destinationId parameter. You must specify a topic for the function. The topic name cannot contain wildcards. For more information, see Create a data destination.	Forward data to another topic

Functions that are supported by each basic data type

- Map

Function	Description
[Object]	Retrieves the value of a specified key.
size()	Retrieves the number of key-value pairs in a map.
containKey(String)	Checks whether a map contains a specified key.
keySet()	Retrieves keys in a map. An array is returned.
remove(Object)	Removes a key-value pair in a map based on a specified key.
put(Object, Object)	Adds key-value pairs to a map.
putAll(map)	Adds another map to a map.

- Array

Function	Description
[int]	Retrieves the value at a specified index. The index of the first element in an array is 0.
contains(Object)	Checks whether an array contains a specified element.
remove(Object)	Removes a specified element from an array.
removeAt(int)	Removes an element at a specified index in an array.
add(Object)	Adds an element to the end of an array.
add(index, Object)	Adds an element to a specified index.
addAll(array)	Adds another array to an array.
size()	Retrieves the number of elements in an array.

- String

Function	Description
substring(start, end)	Extracts a string from the `start` position to the `end-1` position. If you do not configure the `end` parameter, the extraction continues until the end of the string.
length()	Queries the length of a string.
split(String)	Splits a string by using a specified separator.
startsWith(String)	Checks whether a string starts with a specified substring.
endsWith(String)	Checks whether a string ends with a specified substring.
indexOf(String, index)	Queries the position where a substring first appears from a specified index of a string. The default value of the index parameter is 0.

Other functions

Function	Description
endWith(input, suffix)	Checks whether the input string ends with a specified suffix. A Boolean value is returned.
getDeviceTag(key)	Returns the device tag value of a tag `key`. If the tag with the specified `key` is not attached to a device, no tag value is returned.

Function	Description
getOrNull(data, "items", ...)	Returns the value of a specified field in the JSON-formatted data parameter. You can specify one or more fields in the function. Multiple fields must be specified based on levels. The function returns the value of the last field. If the last field does not exist or the value is empty, the function returns null. Example of data: `"items": { "Humidity": { "value": 25, "time": 1626948134319 } }` Examples of the function: • `getOrNull(data, "items")` : returns the value of the items field `"Humidity": {"value": 25,"time": 1626948134319}` . • `getOrNull(data, "items", "Humidity", "value")` : returns the value of the value field `25` . • `getOrNull(data, "items", "Temperature")` : returns the value null because the Temperature field does not exist.
payload(textEncoding)	Encodes the message payload that is sent by a device. The textEncoding parameter specifies an encoding scheme. • If you do not configure the parameter, the message payload is converted to a string by using the UTF-8 encoding. `payload()` is equivalent to `payload('utf-8')` . • If you set the parameter to `'json'` , the message payload is converted to a map. If the payload is not in the JSON format, an error occurs. • If you set the parameter to `'binary'` , the message payload is converted to pass-through binary data.

References

If you want to forward data to a destination, you can write a script by using the supported syntax to call specific functions. For more information, see Script syntax.

5.5. Data forwarding examples

5.5.1. Forward data to another topic

You can forward data that is processed by a parser script to another topic to achieve machine-to-machine (M2M) communication. This article describes the data forwarding process. In this example, a Thing Specification Language (TSL) communication topic is used as the source topic.

Prerequisites

A data source named *DataSource* is created and a TSL communication topic is added the data source. For more information, see Create a data source.

Context

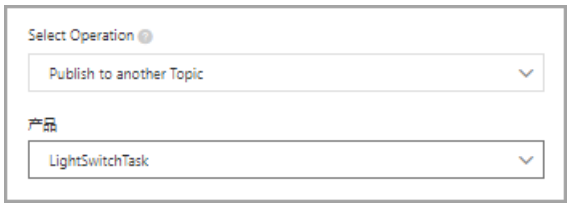
Create a data destination

1.
2.
3.
4. (Optional)On the **Data Forwarding** page, click **Go to New Version** in the upper-right corner to go to the new version.

   > **Note** If you have performed this step, the **Data Forwarding** page of the new version appears after you choose Rules > Data Forwarding.

5. Click the **Data Destination** tab. On this tab, click **Create Data Destination**.
6. In the **Create Data Destination** dialog box, enter a data destination name. In this example, enter *DataPurpose*. Set the parameters and then click **OK**.

   The following figure shows the parameters.



Parameter	Description
Select Operation	Select **Publish to another Topic**.
Product	Select the product to which the destination topic belongs. You must use the `writeIotTopic(destinationId, topic, payload)` function to specify a topic in your parser script. For more information about how to use the function, see Functions.

Configure and start a parser

1. Create a parser named *DataParser*. For more information, see Create a parser.
2. On the **Parser Details** page, associate the parser with the created data source.
   i. In the **Data Source** step of the wizard, click **Associate Data Source**.
   ii. In the dialog box that appears, select *DataSource* from the **Data Source** drop-down list, and then click **OK**.
3. On the **Parser Details** page, associate the parser with the created data destination.
   i. Click **Data Destination** in the wizard. In the **Data Destination** section, click **Associate Data Destination**.

ii. In the dialog box that appears, select *DataPurpose* from the **Data Destination** drop-down list, and then click **OK**.

iii. In the Data Destination section, view and save the **data destination ID**. In this example, the ID is **1000**.

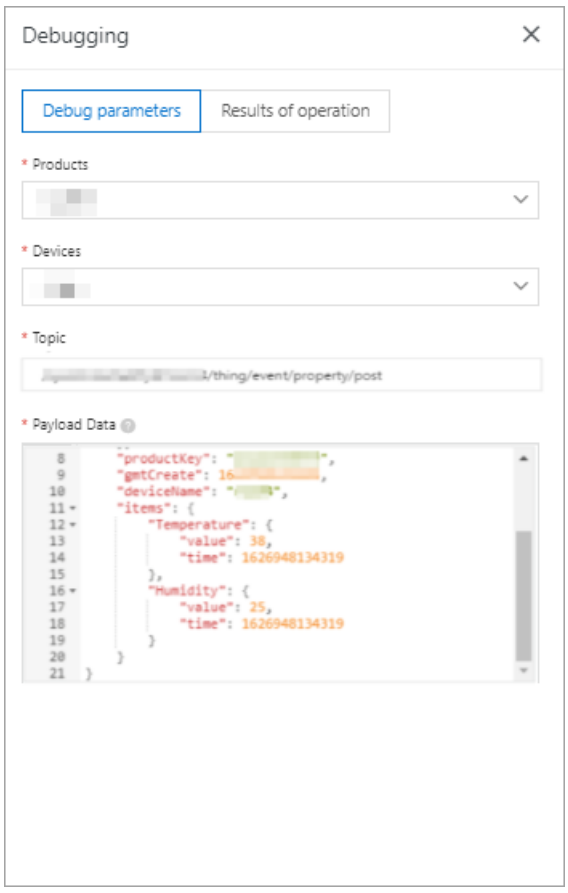
When you write the parser script, you must use the **data destination ID**.

4.

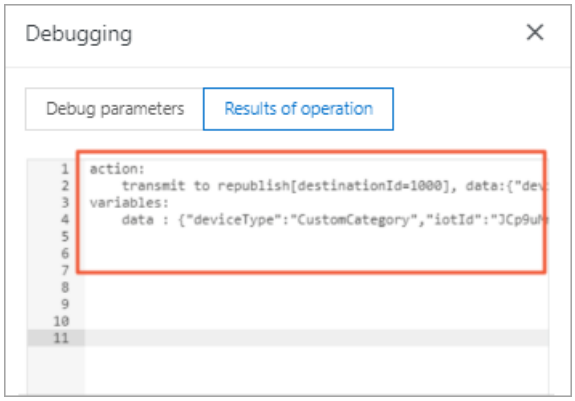
5. For more information about the function parameters, see Functions.

```
// Use the payload() function to obtain the data that is submitted by devices and conve
rt the data by using the JSON format.
var data = payload("json");
// Forward submitted TSL data.
writeIotTopic(1000, "/sys/a1I***/room3/thing/service/property/set", data)
```

6. The following figure shows the parameters.



The following result indicates that the script is implemented.



7.
8. Go to the **Parser** tab of the **Data Forwarding** page. Find the *DataParser* parser and click **Start** in the Actions column to start the parser.

5.5.2. Forward data to an AMQP consumer group

You can use the rules engine to forward messages from devices to IoT Platform. The messages are processed by using data parsing scripts, forwarded to AMQP consumer groups, and then consumed by AMQP clients.This article describes the data forwarding process. In this example, a Thing Specification Language (TSL) communication topic is used as the source topic.

Prerequisites

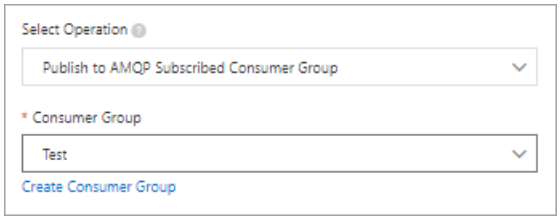
- A data source named *DataSource* is created and a TSL communication topic is added the data source. For more information, see Create a data source.
- A destination AMQP consumer group is created. For more information, see Create a consumer group.

Create a data destination

1.
2.
3.
4. (Optional)On the **Data Forwarding** page, click **Go to New Version** in the upper-right corner to go to the new version.

   > **Note** If you have performed this step, the **Data Forwarding** page of the new version appears after you choose Rules > Data Forwarding.

5. Click the **Data Destination** tab. On this tab, click **Create Data Destination**.
6. In the **Create Data Destination** dialog box, enter a data destination name. In this example, enter *DataPurpose*. Set the parameters and then click **OK**.



Parameter	Description
Select Operation	Select **Publish to AMQP Subscribed Consumer Group**.
Consumer Group	Select an existing consumer group as the data forwarding destination. Click **Create Consumer Group** to create a consumer group.

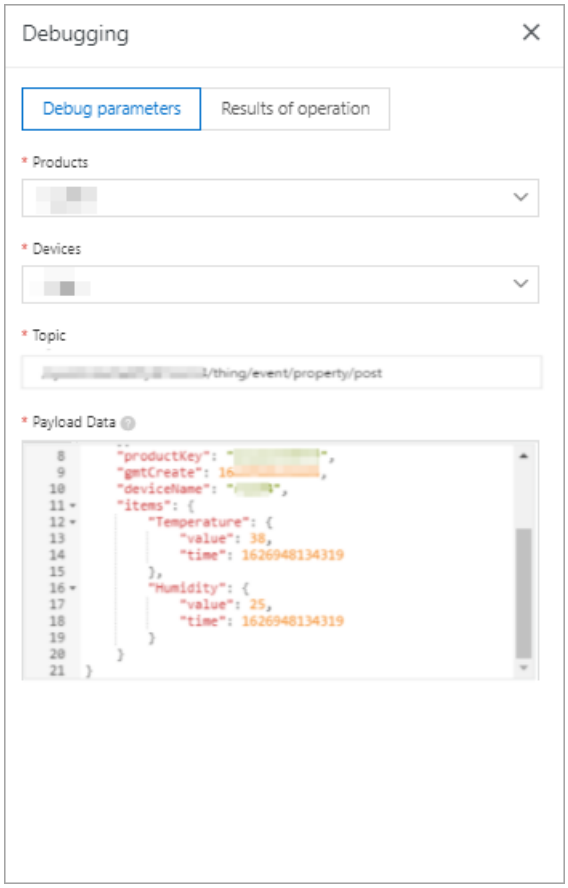
Configure and start a parser

1. Create a parser named *DataParser*. For more information, see Create a parser.
2. On the **Parser Details** page, associate the parser with the created data source.
   i. In the **Data Source** step of the wizard, click **Associate Data Source**.
   ii. In the dialog box that appears, select *DataSource* from the **Data Source** drop-down list, and then click **OK**.
3. On the **Parser Details** page, associate the parser with the created data destination.
   i. Click **Data Destination** in the wizard. In the **Data Destination** section, click **Associate Data Destination**.
   ii. In the dialog box that appears, select *DataPurpose* from the **Data Destination** drop-down list, and then click **OK**.
   iii. In the Data Destination section, view and save the **data destination ID**. In this example, the ID is **1000**.

   When you write the parser script, you must use the **data destination ID**.
4.
5. For more information about the function parameters, see Functions.

```
// Use the payload() function to obtain the data that is submitted by devices and conve
rt the data by using the JSON format.
var data = payload("json");
// Forward submitted TSL data.
writeAmqp(1000, data, "debug");
```

6. The following figure shows the parameters.



The following result indicates that the script is implemented.

7.

8. Go to the **Parser** tab of the **Data Forwarding** page. Find the *DataParser* parser and click **Start** in the Actions column to start the parser.

Configure an AMQP client to consume messages

After the data is forwarded to the AMQP consumer group, your server consumes the messages by using the AMQP client. For more information about how to configure an AMQP client, see Connect an AMQP client to IoT Platform.

For sample code of the AMQP client implementation, see the following topics:

- Connect a client to IoT Platform by using the SDK for Java
- Connect a client to IoT Platform by using the SDK for .NET
- Connect a client to IoT Platform by using the SDK for Node.js
- Connect a client to IoT Platform by using the SDK for Python 2.7

- Connect a client to IoT Platform by using the SDK for Python 3
- Connect a client to IoT Platform by using the SDK for PHP
- Connect a client to IoT Platform by using the SDK for Go

5.5.3. Forward data to Message Queue for Apache RocketMQ

You can use the rules engine to forward data from IoT Platform to Message Queue for Apache RocketMQ for storage. This ensures high-reliability data transmission from devices to IoT Platform, Message Queue for Apache RocketMQ, and application servers.This article describes the data forwarding process. In this example, a Thing Specification Language (TSL) communication topic is used as the source topic.

Prerequisites

- A data source named *DataSource* is created and a TSL communication topic is added the data source. For more information, see Create a data source.
- A Message Queue for Apache RocketMQ instance and a topic that is used to receive data are created. For more information, see What is Message Queue for Apache RocketMQ?.

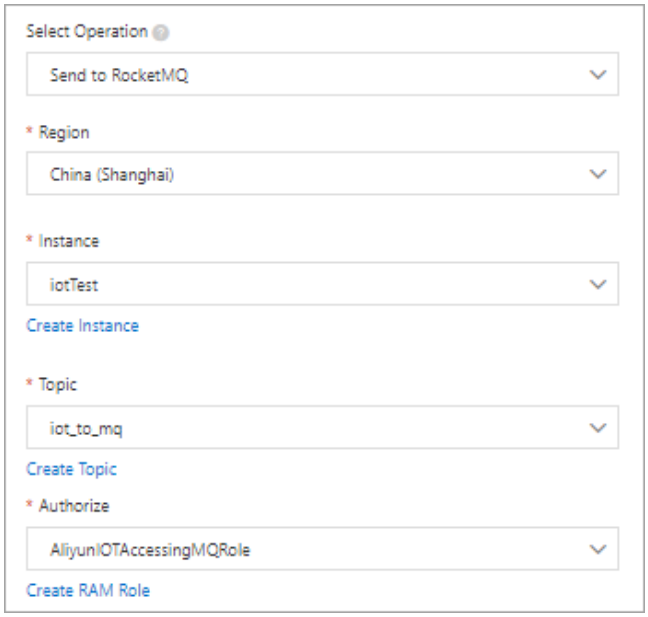
> **Notice** The Message Queue for Apache RocketMQ instance must reside in the same region as the IoT Platform instance on which you want to configure a data forwarding rule. This limit does not apply to the existing data forwarding rules that are used to forward data across instances in different regions.

Create a data destination

1.
2.
3.
4. (Optional)On the **Data Forwarding** page, click **Go to New Version** in the upper-right corner to go to the new version.

   > **Note** If you have performed this step, the **Data Forwarding** page of the new version appears after you choose Rules > Data Forwarding.

5. Click the **Data Destination** tab. On this tab, click **Create Data Destination**.
6. In the **Create Data Destination** dialog box, enter a data destination name. In this example, enter *DataPurpose*. Set the parameters and then click **OK**.



Parameter	Description
Select Operation	Select **Send to RocketMQ**.
Region	Select the region where the Message Queue for Apache RocketMQ instance is deployed.
Instance	Select the Message Queue for Apache RocketMQ instance. You can click **Create Instance** to go to the Message Queue for Apache RocketMQ console and create a Message Queue for Apache RocketMQ instance. For more information, see Message Queue for Apache RocketMQ documentation.
Topic	Select the Message Queue for Apache RocketMQ topic that is used to receive data from IoT Platform. You can click **Create Topic** to go to the Message Queue for Apache RocketMQ console and create a Message Queue for Apache RocketMQ topic.
Authorize	Grant IoT Platform the permissions to write data to Message Queue for Apache RocketMQ. If no RAM roles exist, click **Create RAM Role** to go to the Resource Access Management (RAM) console, create a RAM role, and then grant permissions to the RAM role. For more information, see Create a RAM role.

Configure and start a parser

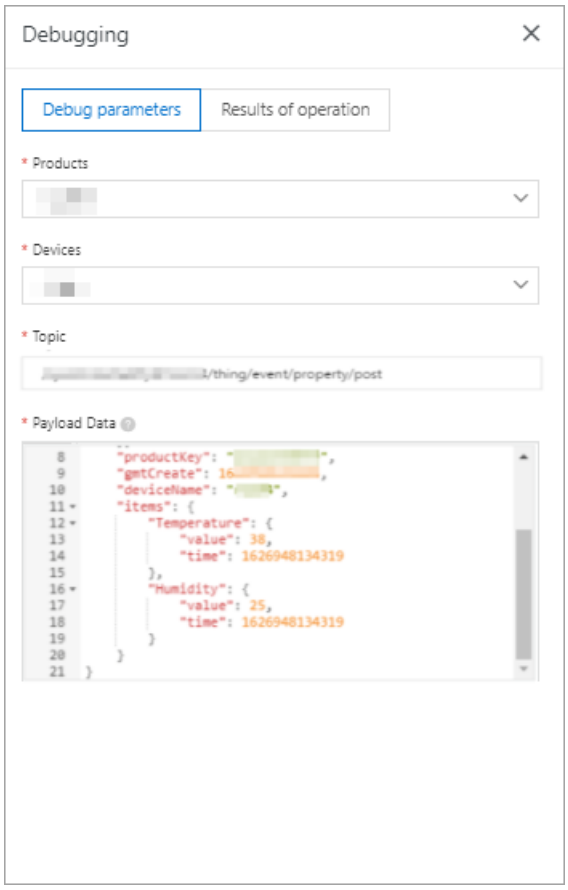
1. Create a parser named *DataParser*. For more information, see Create a parser.
2. On the **Parser Details** page, associate the parser with the created data source.
   i. In the **Data Source** step of the wizard, click **Associate Data Source**.
   ii. In the dialog box that appears, select *DataSource* from the **Data Source** drop-down list, and then click **OK**.

3. On the **Parser Details** page, associate the parser with the created data destination.
   i. Click **Data Destination** in the wizard. In the **Data Destination** section, click **Associate Data Destination**.
   ii. In the dialog box that appears, select *DataPurpose* from the **Data Destination** drop-down list, and then click **OK**.
   iii. In the Data Destination section, view and save the **data destination ID**. In this example, the ID is **1000**.

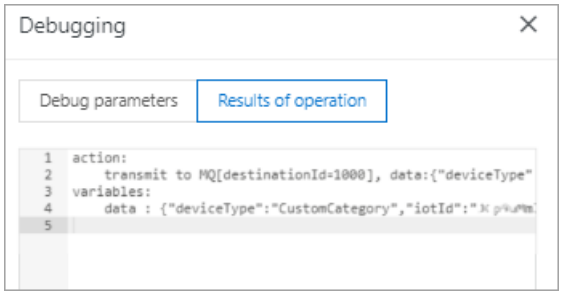
   When you write the parser script, you must use the **data destination ID**.
4.
5. For more information about the function parameters, see Functions.

```
// Use the payload() function to obtain the data that is submitted by devices and conve
rt the data by using the JSON format.
var data = payload("json");
// Forward submitted TSL data.
writeMq(1000, data, "debug");
```

6. The following figure shows the parameters.



The following result indicates that the script is implemented.



7.
8. Go to the **Parser** tab of the **Data Forwarding** page. Find the *DataParser* parser and click **Start** in the Actions column to start the parser.
9. In the Message Queue for Apache RocketMQ console, check whether the message is received.

5.5.4. Forward data to Tablestore

You can use the data forwarding feature of the rules engine to forward data to Tablestore.This article describes the data forwarding process. In this example, a Thing Specification Language (TSL) communication topic is used as the source topic.

Prerequisites

- A data source named *DataSource* is created and a TSL communication topic is added the data source. For more information, see Create a data source.
- A Tablestore instance and a table that is used to receive data are created. For more information, see the Tablestore documentation.

> **Notice** If you use an IoT Platform instance of the Enterprise Edition, the region where the Tablestore instance resides must be the same as that of the IoT Platform instance.

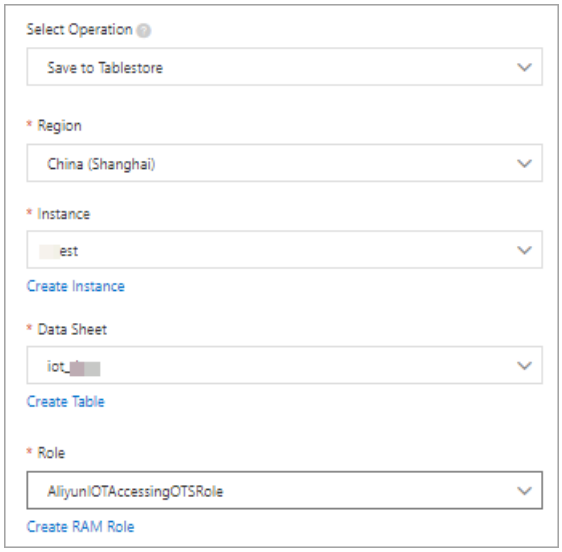
Create a data destination

1.
2.
3.
4. (Optional)On the **Data Forwarding** page, click **Go to New Version** in the upper-right corner to go to the new version.

   > **Note** If you have performed this step, the **Data Forwarding** page of the new version appears after you choose Rules > Data Forwarding.

5. Click the **Data Destination** tab. On this tab, click **Create Data Destination**.
6. In the **Create Data Destination** dialog box, enter a data destination name. In this example, enter *DataPurpose*. Set the parameters and then click **OK**.

   > **Note** Only JSON-formatted data can be forwarded.



Parameter	Description
Select Operation	Select **Store Data to Tablestore**.
Region	Select a region where your Tablestore instance is deployed.
Instance	Select a Tablestore instance. You can click **Create Instance** to go to the Tablestore console and create a Tablestore instance. For more information, see the Tablestore documentation.
Table	Select a Tablestore table that is used to receive data. You can click **Create Data Table** to go to the Tablestore console and create a Tablestore table.
Role	Grant IoT Platform the permissions to write data to Tablestore. If no RAM roles exist, click **Create RAM Role** to go to the Resource Access Management (RAM) console, create a RAM role, and then grant permissions to the RAM role. For more information, see Create a RAM role.

Configure and start a parser

1. Create a parser named *DataParser*. For more information, see Create a parser.
2. On the **Parser Details** page, associate the parser with the created data source.
   i. In the **Data Source** step of the wizard, click **Associate Data Source**.
   ii. In the dialog box that appears, select *DataSource* from the **Data Source** drop-down list, and then click **OK**.
3. On the **Parser Details** page, associate the parser with the created data destination.

i. Click **Data Destination** in the wizard. In the **Data Destination** section, click **Associate Data Destination**.

ii. In the dialog box that appears, select *DataPurpose* from the **Data Destination** drop-down list, and then click **OK**.

iii. In the Data Destination section, view and save the **data destination ID**. In this example, the ID is **1000**.

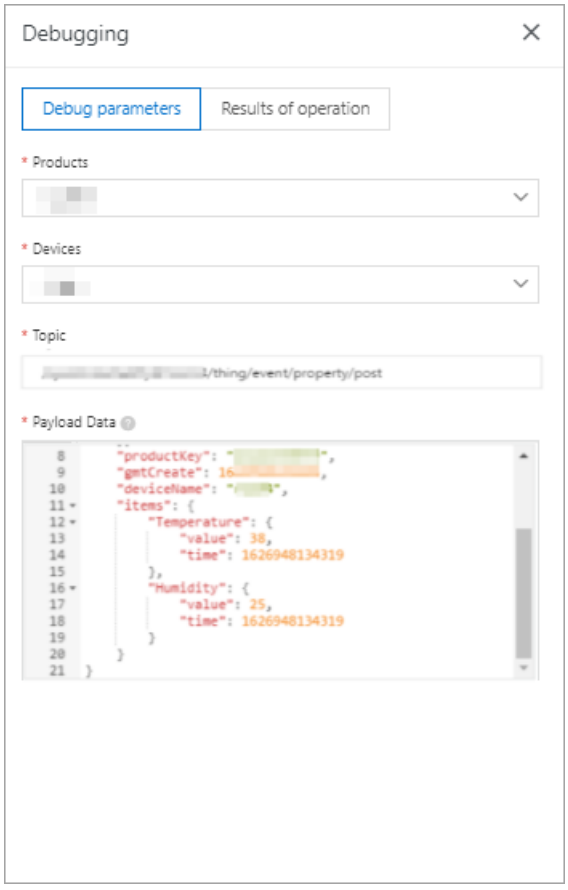
When you write the parser script, you must use the **data destination ID**.

4.

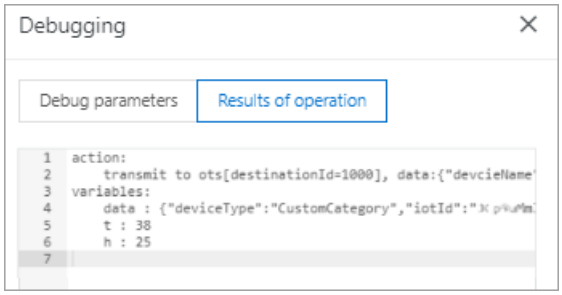
5. For more information about the function parameters, see Functions.

```
// Use the payload() function to obtain the data that is submitted by devices and conve
rt the data by using the JSON format.
var data = payload("json");
// Obtain the submitted property values.
var h = data.items.Humidity.value;
var t = data.items.Temperature.value;
// Add the deviceName and id primary keys to the table. You can use the writeTableStore
() method to write values to columns by using the format of column:value.
writeTableStore(1000, {"devcieName":deviceName(), "id":timestamp(), "temperature":t, "h
umidity":h});
```

6. The following figure shows the parameters.

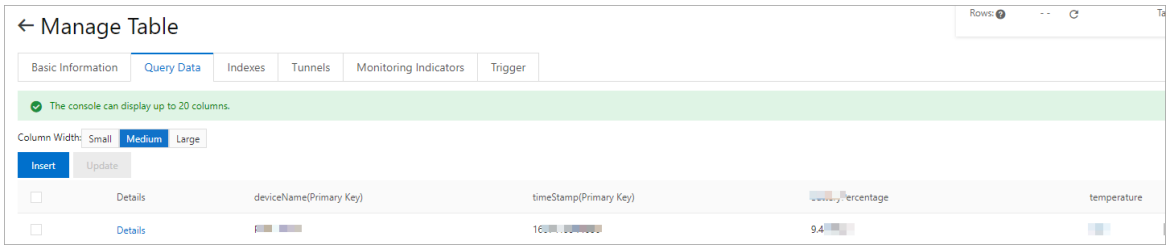


The following result indicates that the script is implemented.



7.

8. Go to the **Parser** tab of the **Data Forwarding** page. Find the *DataParser* parser and click **Start** in the Actions column to start the parser.
9. After the data is pushed, log on to the Tablestore console. Go to the **Manage Editor** page of the table that is used to receive data and check whether the specified data is received.



5.5.5. Forward data to ApsaraDB RDS

You can use the data forwarding feature of the rules engine to forward processed data to ApsaraDB RDS instances for storage.This article describes the data forwarding process. In this example, a Thing Specification Language (TSL) communication topic is used as the source topic.

Prerequisites

- A data source named *DataSource* is created and a TSL communication topic is added the data source. For more information, see Create a data source.
- An ApsaraDB RDS instance, a database, and a table are created in the region where your IoT Platform instance resides. The database engine of the instance must be MySQL or Microsoft SQL Server and the network type of the instance must be VPC. For more information, see the ApsaraDB RDS documentation.

Limits

- Data forwarding is supported between IoT Platform instances and ApsaraDB RDS instances that reside in the same region. For example, you can forward the data of an IoT Platform instance in the China (Shanghai) region to an ApsaraDB RDS table that resides only in the China (Shanghai) region.
- You can forward data to ApsaraDB RDS instances that resides only in VPCs.
- You can forward data to MySQL and SQL Server instances.
- Data forwarding is supported between standard databases and privileged databases.
- You can forward data only in the JSON format.

Create a data destination

1.

2.
3.
4. (Optional)On the **Data Forwarding** page, click **Go to New Version** in the upper-right corner to go to the new version.

   > **Note** If you have performed this step, the **Data Forwarding** page of the new version appears after you choose Rules > Data Forwarding.

5. Click the **Data Destination** tab. On this tab, click **Create Data Destination**.
6. In the **Create Data Destination** dialog box, enter a data destination name. In this example, enter *DataPurpose*. Set the parameters and then click **OK**.

Parameter	Description
Select Operation	Select **Save to RDS**.
Region	Select the region where your IoT Platform instance resides.
RDS Instance	Select the ApsaraDB RDS instance.
Database	Enter the database name. **Note** If you use a privileged database, you must enter a database name.
Account	Enter the account of the ApsaraDB RDS instance. This account must have the read and write permissions on the database. Otherwise, the rules engine cannot write data to the ApsaraDB RDS instance. **Note** After the rules engine obtains the account, the rules engine writes only the data that matches the specified rule to the database. No extra operations are performed.
Password	Enter the password that is used to log on to the ApsaraDB RDS instance.
Table Name	Enter the name of the table that is created in the database. The rules engine writes data to the table.
Role	Authorize IoT Platform to write data to ApsaraDB RDS. If no RAM roles exist, click **Create RAM Role** to go to the Resource Access Management (RAM) console, create a RAM role, and then grant permissions to the RAM role. For more information, see Create a RAM role.

Configure and start a parser

1. Create a parser named *DataParser*. For more information, see Create a parser.
2. On the **Parser Details** page, associate the parser with the created data source.
   i. In the **Data Source** step of the wizard, click **Associate Data Source**.

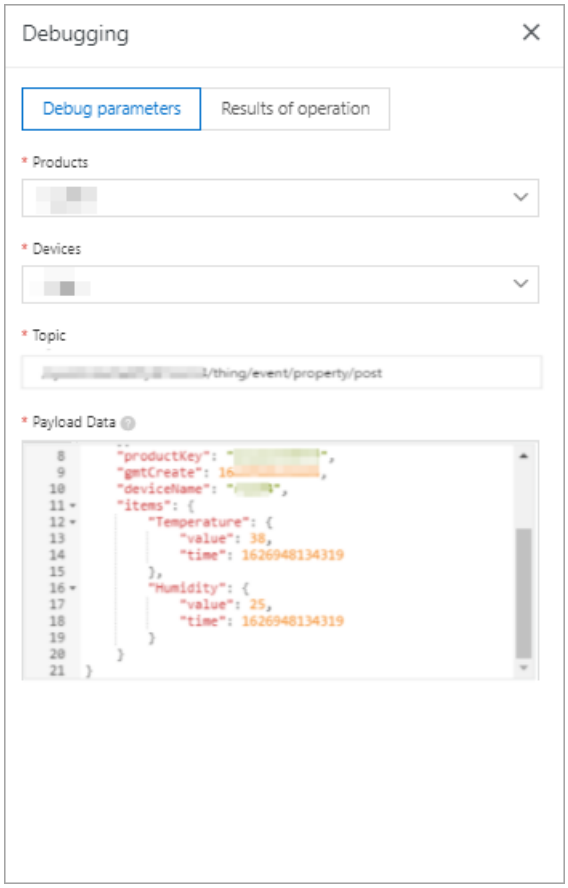
ii. In the dialog box that appears, select *DataSource* from the **Data Source** drop-down list, and then click **OK**.

3. On the **Parser Details** page, associate the parser with the created data destination.
   i. Click **Data Destination** in the wizard. In the **Data Destination** section, click **Associate Data Destination**.
   ii. In the dialog box that appears, select *DataPurpose* from the **Data Destination** drop-down list, and then click **OK**.
   iii. In the Data Destination section, view and save the **data destination ID**. In this example, the ID is **1000**.

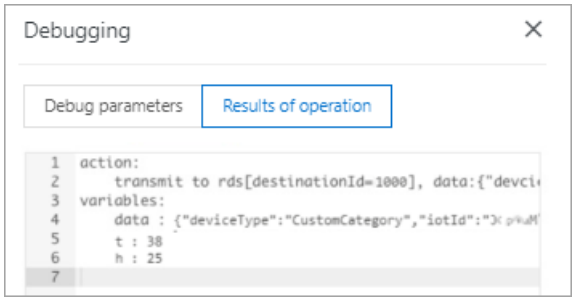
   When you write the parser script, you must use the **data destination ID**.
4. On the **Parser Details** page, click **Parser**.
5. For more information about the function parameters, see Functions.

```
// Use the payload() function to obtain the data that is submitted by devices and conve
rt the data by using the JSON format.
var data = payload("json");
// Filter the submitted temperature and humidity values.
var h = data.items.Humidity.value;
var t = data.items.Temperature.value;
// An ApsaraDB RDS table includes the following columns: id (auto-increment primary key
), deviceName, temperature, humidity, and time. You can use the writeRds() method to wr
ite values to columns by using the format of column:value.
// Set a rule that is triggered if the temperature value is greater than 38.
if (t > 38) {
    writeRds(1000, {"devcieName":deviceName(), "temperature":t, "time":timestamp(), "hu
midity":h});
}
```

6. The following figure shows the parameters.

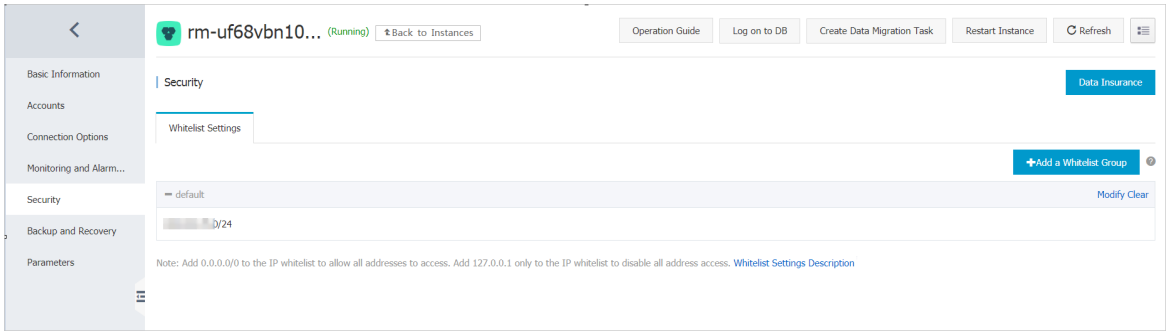


The following result indicates that the script is implemented.



7. Click **Publish**.
8. Go to the **Parser** tab of the **Data Forwarding** page. Find the *DataParser* parser and click **Start** in the Actions column to start the parser.
9. Log on to the ApsaraDB RDS console. On the **Data Security** page, configure a whitelist or view whitelists. After you set the parameters, the rules engine adds the following IP addresses to the ApsaraDB RDS whitelist. Then, the rules engine can communicate with ApsaraDB RDS. If one of the following IP addresses is not displayed in the whitelist, you must manually add the IP address.
   - China (Shanghai): `100.104.53.192/26,100.104.148.64/26,100.104.6.192/26,100.104.143.128/26,100.104.76.0/24,100.104.73.128/26,100.104.200.64/26,100.104.40.64/26,100.104.3.0/26,100.104.29.128/26,100.104.121.0/26,100.104.84.64/26`
   - Singapore (Singapore): `100.104.106.0/24`
   - US (Silicon Valley): `100.104.8.0/24`
   - US (Virginia): `100.104.133.64/26`

- Germany (Frankfurt): `100.104.160.192/26`
- Japan (Tokyo): `100.104.160.192/26`



5.5.6. Forward data to Message Service (MNS)

You can use the data forwarding feature of the rules engine to forward device data to an MNS topic. Your business server subscribes to the messages of the topic. This implements high-performance message transmission between your business server and devices.This article describes the data forwarding process. In this example, a Thing Specification Language (TSL) communication topic is used as the source topic.

Prerequisites

- A data source named *DataSource* is created and a TSL communication topic is added the data source. For more information, see Create a data source.
- An MNS topic is created. A subscription whose Push Type parameter is set to HTTP or Queue is created for the topic. For more information, see the MNS documentation.

> **Notice** If you use an IoT Platform instance of the Enterprise Edition, the region where the MNS topic resides must be the same as that of the IoT Platform instance.

Context

For more information about the data forwarding process, see Data forwarding procedure.

Create a data destination

1.
2.
3.
4. (Optional)On the **Data Forwarding** page, click **Go to New Version** in the upper-right corner to go to the new version.

   > **Note** If you have performed this step, the **Data Forwarding** page of the new version appears after you choose Rules > Data Forwarding.

5. Click the **Data Destination** tab. On this tab, click **Create Data Destination**.
6. In the **Create Data Destination** dialog box, enter a data destination name. In this example, enter *DataPurpose*. Set the parameters and then click **OK**.

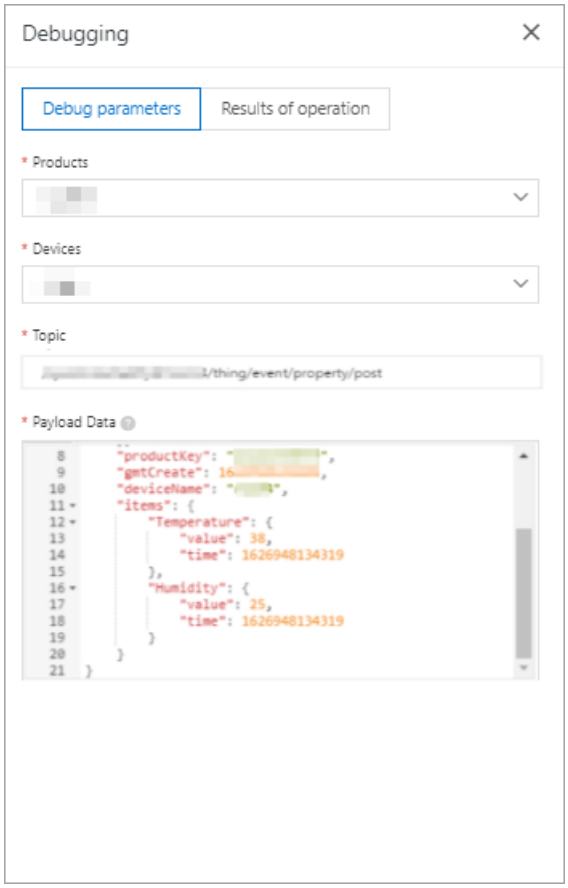
Parameter	Description
Operation	Select **Send Data to MNS**.
Region	Select a region where MNS runs.
Topic	Select an MNS topic that is used to receive data. MNS sends received messages to a subscription of the topic. The Push Type parameter value of the subscription is HTTP or Queue. You can subscribe to messages that IoT Platform pushes to MNS through only the HTTP or queue method. To create an MNS topic and subscription in the MNS console, click **Create Topic**. For more information, see the MNS documentation.
Role	Authorize IoT Platform to write data to MNS. If no RAM roles exist, click **Create RAM Role** to go to the Resource Access Management (RAM) console, create a RAM role, and then grant permissions to the RAM role. For more information, see Create a RAM role.

Configure and start a parser

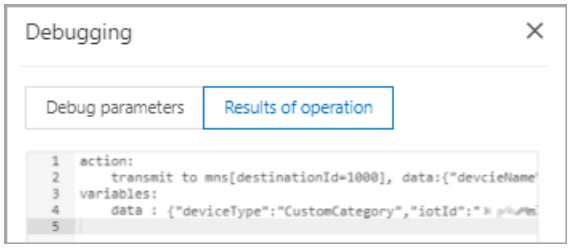
1. Create a parser named *DataParser*. For more information, see Create a parser.
2. On the **Parser Details** page, associate the parser with the created data source.
   i. In the **Data Source** step of the wizard, click **Associate Data Source**.
   ii. In the dialog box that appears, select *DataSource* from the **Data Source** drop-down list, and then click **OK**.
3. On the **Parser Details** page, associate the parser with the created data destination.
   i. Click **Data Destination** in the wizard. In the **Data Destination** section, click **Associate Data Destination**.
   ii. In the dialog box that appears, select *DataPurpose* from the **Data Destination** drop-down list, and then click **OK**.
   iii. In the Data Destination section, view and save the **data destination ID**. In this example, the ID is **1000**.
   When you write the parser script, you must use the **data destination ID**.
4. On the **Parser Details** page, click **Parser**.
5. For more information about the function parameters, see Functions.

```
// Use the payload() function to obtain the data that is submitted by devices and conve
rt the data by using the JSON format.
var data = payload("json");
// Forward submitted TSL data.
writeMns(1000, data)
```

6. The following figure shows the parameters.



The following result indicates that the script is implemented.



7. Click **Publish**.
8. Go to the **Parser** tab of the **Data Forwarding** page. Find the *DataParser* parser and click **Start** in the Actions column to start the parser.

5.5.7. Forward data to Time Series Database (TSDB)

If you use a public IoT Platform instance, you can configure data forwarding rules to forward processed data to TSDB. This article describes the data forwarding process. In this example, a Thing Specification Language (TSL) communication topic is used as the source topic.

Prerequisites

- A data source named *DataSource* is created and a TSL communication topic is added the data source. For more information, see Create a data source.
- A TSDB instance that resides in a virtual private cloud (VPC) is created in the Japan (Tokyo) region.

For more information about TSDB, see Overview.

Context

In this example, the following data is written to TSDB:

Limits

- Data can be forwarded only within a region. For example, you can only forward data from an IoT Platform instance that is deployed in the Japan (Tokyo) region to a TSDB instance that is deployed in the Japan (Tokyo) region.
- You can forward data only to a TSDB instance that is deployed in a VPC.
- You can forward data only in the JSON format.
- Only TSDB is supported. Time Series Database for InfluxDB® and Ganos are not supported.
- In the forwarded messages, all fields are written to TSDB as the metric, except for the fields that are configured as the timestamp and tag value. The data types of the metric parameter support numeric and string. If you use other data types, data may fail to be written to the database.

Create a data destination

1.
2.
3.
4. (Optional)On the **Data Forwarding** page, click **Go to New Version** in the upper-right corner to go to the new version.

   > **Note** If you have performed this step, the **Data Forwarding** page of the new version appears after you choose Rules > Data Forwarding.

5. Click the **Data Destination** tab. On this tab, click **Create Data Destination**.
6. In the **Create Data Destination** dialog box, enter a data destination name. In this example, enter *DataPurpose*. Set the parameters and then click **OK**.

Parameter	Description
Select Operation	Select **Save to TSDB**.
Region	Specify the region where your IoT Platform instance resides.
TSDB Instance	Specify the TSDB instance that you have created.
metric Data Type	Specify the data type of the metric parameter. Valid values: **numeric** and **string**. For more information, click the icon.
Role	Grant IoT Platform the permissions to write data to TSDB.

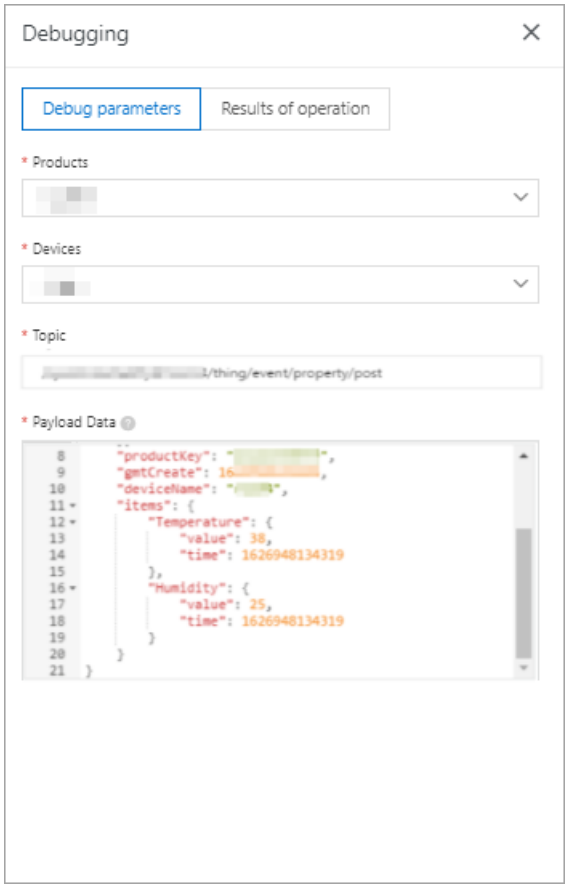
Configure and start a parser

1. Create a parser named *DataParser*. For more information, see Create a parser.
2. On the **Parser Details** page, associate the parser with the created data source.
   i. In the **Data Source** step of the wizard, click **Associate Data Source**.
   ii. In the dialog box that appears, select *DataSource* from the **Data Source** drop-down list, and then click **OK**.
3. On the **Parser Details** page, associate the parser with the created data destination.
   i. Click **Data Destination** in the wizard. In the **Data Destination** section, click **Associate Data Destination**.
   ii. In the dialog box that appears, select *DataPurpose* from the **Data Destination** drop-down list, and then click **OK**.
   iii. In the Data Destination section, view and save the **data destination ID**. In this example, the ID is **1000**.

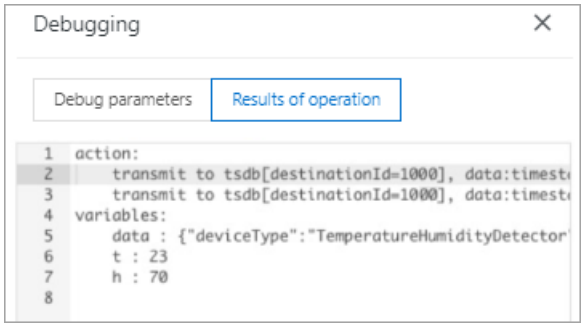
   When you write the parser script, you must use the **data destination ID**.
4.
5. For more information about the function parameters, see Functions.

```
// Use the payload() function to obtain the data that is submitted by devices and conve
rt the data by using the JSON format.
var data = payload("json");
// Filter the submitted temperature and humidity values.
var h = data.items.Humidity.value;
var t = data.items.Temperature.value;
// Forward submitted TSL data.
writeTsdb(1000,timestamp(),"temperature", t , {"deviceName":deviceName()});
writeTsdb(1000,timestamp(),"humidity", h , {"deviceName":deviceName()});
```

6. The following figure shows the parameters.



The following result indicates that the script is implemented.



7.
8. Go to the **Parser** tab of the **Data Forwarding** page. Find the *DataParser* parser and click **Start** in the Actions column to start the parser.

5.5.8. Forward data to Function Compute

You can use the data forwarding feature of the rules engine to forward data to Function Compute. Then, Function Compute uses function scripts to process business data.This article describes the data forwarding process. In this example, a Thing Specification Language (TSL) communication topic is used as the source topic.

Prerequisites

- A data source named *DataSource* is created and a TSL communication topic is added the data source. For more information, see Create a data source.

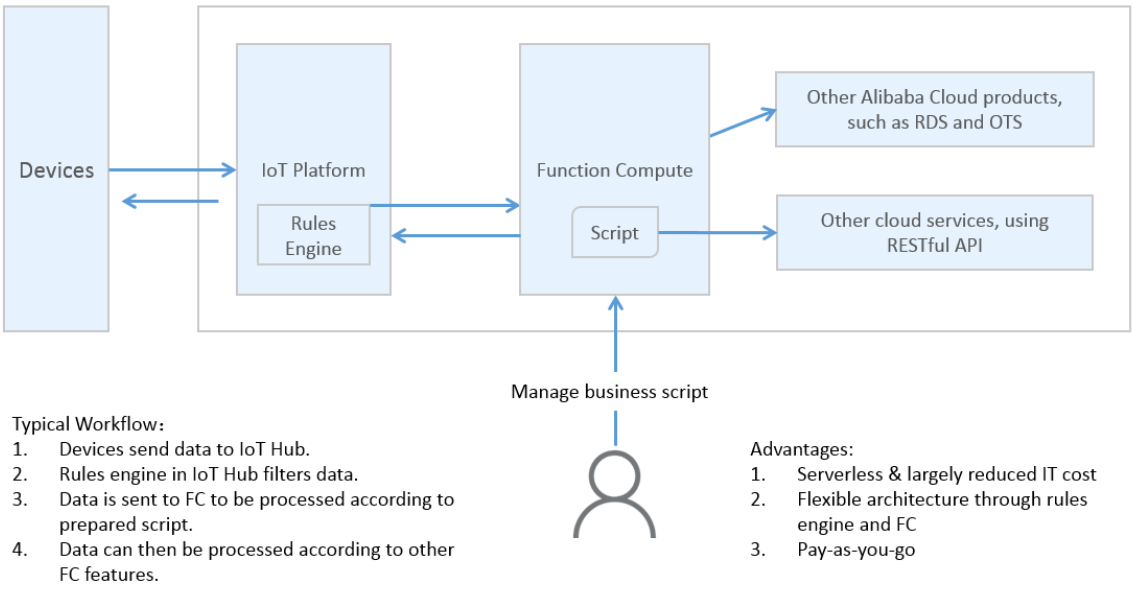
- A Function Compute service and a function are created. The function is configured and can run as expected after verification. For more information, see the Function Compute documentation.

> **Notice** If you use an IoT Platform instance of the Enterprise Edition, the region where the Function Compute service resides must be the same as that of the IoT Platform instance.

Context

The rules engine uses the data forwarding feature to forward device data to Function Compute. Then, Function Compute runs business-specific function scripts to implement various business features.

The following figure shows the data forwarding process.



Create a data destination

1.
2.
3.
4. (Optional)On the **Data Forwarding** page, click **Go to New Version** in the upper-right corner to go to the new version.

> **Note** If you have performed this step, the **Data Forwarding** page of the new version appears after you choose Rules > Data Forwarding.

5. Click the **Data Destination** tab. On this tab, click **Create Data Destination**.
6. In the **Create Data Destination** dialog box, enter a data destination name. In this example, enter *DataPurpose*. Set the parameters and then click **OK**.

Parameter	Description
Operation	Select **Send Data to Function Compute**.

Parameter	Description
Regions	Select the region in which your Function Compute service is deployed.
Services	Select a Function Compute service. You can click **Create Service** to go to the Function Compute console and create a service. For more information, see Manage services.
Function Version	Valid values: ○ **Use Default Version**: Use the default version of the Function Compute service. The default version is LATEST. ○ **Select Version**: Select the version of the Function Compute service that you want to release. You can click **Create Version** to go to the Function Compute console and create a version. For more information, see Manage versions. ○ **Select Alias**: Select the alias that you want to specify for the version of the Function Compute service. You can click **Create Alias** to go to the Function Compute console and create an alias. For more information, see Manage aliases.
Function	Select the function that you want to use to receive data. You can click **Create Function** to go to the Function Compute console and create a function. For more information, see Manage functions.
Authorize	Grant IoT Platform the permissions to write data to Function Compute. If no RAM roles exist, click **Create RAM Role** to go to the Resource Access Management (RAM) console, create a RAM role, and then grant permissions to the RAM role. For more information, see Create a RAM role.

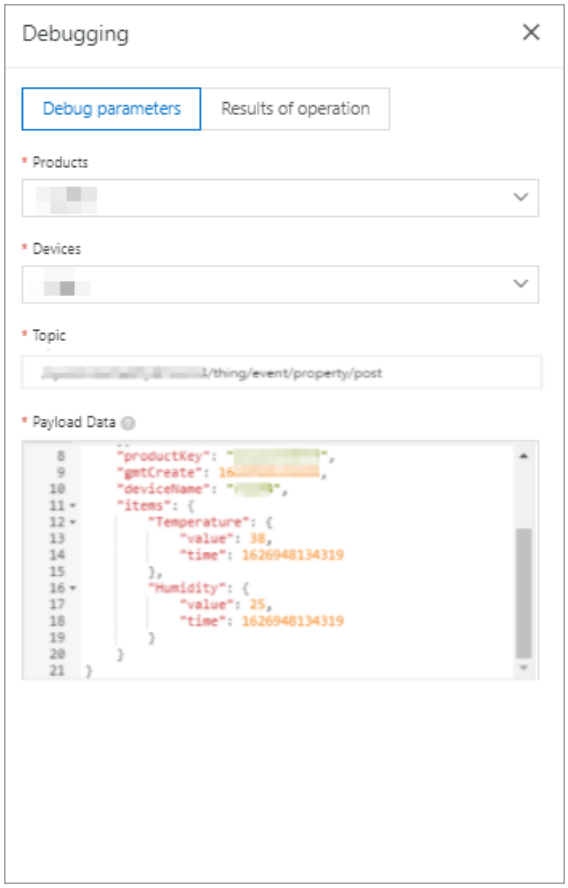
Configure and start a parser

1. Create a parser named *DataParser*. For more information, see Create a parser.
2. On the **Parser Details** page, associate the parser with the created data source.
   i. In the **Data Source** step of the wizard, click **Associate Data Source**.
   ii. In the dialog box that appears, select *DataSource* from the **Data Source** drop-down list, and then click **OK**.
3. On the **Parser Details** page, associate the parser with the created data destination.
   i. Click **Data Destination** in the wizard. In the **Data Destination** section, click **Associate Data Destination**.
   ii. In the dialog box that appears, select *DataPurpose* from the **Data Destination** drop-down list, and then click **OK**.
   iii. In the Data Destination section, view and save the **data destination ID**. In this example, the ID is **1000**.
   When you write the parser script, you must use the **data destination ID**.
4. On the **Parser Details** page, click **Parser**.

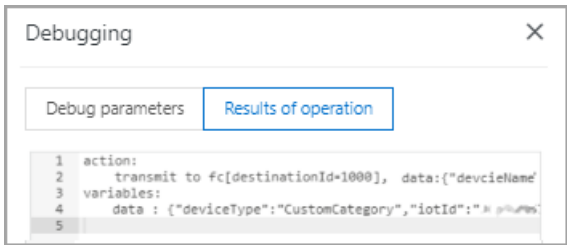
5. For more information about the function parameters, see Functions.

```
// Use the payload() function to obtain the data that is submitted by devices and conve
rt the data by using the JSON format.
var data = payload("json");
// Forward submitted TSL data.
writeFc(1000, data)
```

6. The following figure shows the parameters.



The following result indicates that the script is implemented.



7. Click **Publish**.
8. Go to the **Parser** tab of the **Data Forwarding** page. Find the *DataParser* parser and click **Start** in the Actions column to start the parser.
9. Log on to the Function Compute console. On the **Logs** tab of the Function Details page, view the running records of the function. Click **Monitoring Dashboard** in the upper-right corner to view the monitoring statistics on the function.

Notice The monitoring statistics on a function are generated 5 minutes after the function runs.

6.Data formats

The data forwarding and server-side subscription features of IoT Platform process and forward data based on the data formats of topics. Topics can be classified into custom topics, basic communication topics, and Thing Specification Language (TSL)-based topics. This topic describes the data formats of basic communication topics and TSL-based communication topics.

Topics that are related to the rules engine and device communication

In this example, the format of the data that is submitted by devices is converted by the rules engine. For more information about the formats of original data, see Alink protocol. The following table describes the topics.

Topics

Topic	Description	References
Custom	The topic that is used to forward data of custom formats. The format of this topic is the same as the format of a custom topic. Format: `/${productKey}/${deviceName}/user/${TopicShortName}`. `${TopicShortName}` specifies a custom topic category, which indicates the suffix of the custom topic. The value can contain wildcard characters, including plus signs (+) and number signs (#). • All equipment (`+`): indicates all devices of the specified product. • `/user/#` : indicates all topics of the specified device.	Custom topics
Device Status Change Notification	The topic that is used to forward notifications when the status of a device changes between online and offline. Format: `/as/mqtt/status/${productKey}/${deviceName}`.	Submit device status

Topic	Description	References
TSL Data Reporting	The following topics are provided: • `/${productKey}/${deviceName}/thing/event/property/post`: This topic is used to forward device properties. • `/${productKey}/${deviceName}/thing/event/${tsl.event.identifier}/post`: This topic is used to forward device events. • `/${productKey}/${deviceName}/thing/event/property/post`: This topic is used to forward multiple device properties at a time. • `/${productKey}/${deviceName}/thing/event/batch/post`: This topic is used to forward multiple device events at a time. • `/${productKey}/${deviceName}/thing/downlink/reply/message`: This topic is used to forward messages that are sent by a device as responses to IoT Platform commands.	• Submit device properties • Submit device events • Submit device properties in batches • Submit device events in batches • Submit responses to downstream requests
	The preceding topics correspond to the following device topics: • `/sys/{productKey}/{deviceName}/thing/event/property/post`: This topic is used to submit device properties. • `/sys/${productKey}/${deviceName}/thing/event/${tsl.event.identifier}/post` and `/sys/${productKey}/${deviceName}/thing/event/${tsl.functionBlockId}:{tsl.event.identifier}/post`: These topics are used to submit device events. • `/sys/${productKey}/${deviceName}/thing/event/property/batch/post`: This topic is used to submit device properties and events in batches.	• Devices submit property information to IoT Platform • Devices submit events to IoT Platform • Devices submit multiple properties and events to IoT Platform at a time
Device Changes Throughout Lifecycle	The topic that is used to forward notifications when a device is created, deleted, disabled, or enabled. Format: `/${productKey}/${deviceName}/thing/lifecycle`	Submit lifecycle changes
Sub-Device Data Report Detected by Gateway	The topic that is used to submit and forward the information about a new sub-device when a gateway detects the sub-device. This topic is specific to gateways. Format: `/${productKey}/${deviceName}/thing/list/found`	Submit information about detected sub-devices
Device Topological Relation Changes	The topic that is used to forward notifications when topological relationships between sub-devices and the gateway are created or deleted. This topic is specific to gateways. Format: `/${productKey}/${deviceName}/thing/topo/lifecycle`.	Submit topology changes

Topic	Description	References
	The preceding topic corresponds to the following topic: `/sys/{productKey}/{deviceName}/thing/topo/change`. This topic is used to submit device data.	Notify gateways of changes of topological relationships
Device tag change	The topic that is used to forward notifications when device tags are changed. Format: `/${productKey}/${deviceName}/thing/deviceinfo/update`.	Submit device tag changes
	The preceding topic corresponds to the following topic: `/sys/{productKey}/{deviceName}/thing/deviceinfo/update`. This topic is used to submit device data.	Report tags
TSL Historical Data Reporting	The following topics are provided: • `/${productKey}/${deviceName}/thing/event/property/history/post`: This topic is used to forward historical properties. • `/${productKey}/${deviceName}/thing/event/${tsl.event.identifier}/history/post`: This topic is used to forward historical events.	• Submit historical properties • Submit historical events
	The preceding topics correspond to the following topic: `/sys/{productKey}/{deviceName}/thing/event/property/history/post`. This topic is used to submit historical TSL data.	Devices submit historical TSL data to IoT Platform
Device status notification	The following topics are provided: • `/${productKey}/${deviceName}/ota/upgrade`: This topic is used to forward over-the-air (OTA) update results. • `/${productKey}/${deviceName}/ota/progress/post`: This topic is used to forward update progresses.	• Submit the status data of over-the-air (OTA) updates • Submit the progress data of OTA updates
	The preceding topics correspond to the following topic: `/ota/device/progress/${YourProductKey}/${YourDeviceName}`. This topic is used to submit update progresses.	Submit the update progress to IoT Platform
Submit a module version number	The topic that is used to forward notifications when the version number of an OTA module for a device is changed. Format: `/${productKey}/${deviceName}/ota/version/post`.	Submit OTA module versions
	The preceding topic corresponds to the following topic: `/ota/device/inform/${YourProductKey}/${YourDeviceName}`. This topic is used to submit the version number of an OTA module.	Submit OTA module versions to IoT Platform

Topic	Description	References
Batch status notification	The topic to which IoT Platform sends notifications when the status of an OTA update batch changes. Format: `/${productKey}/${packageId}/${jobId}/ota/job/status`.	Submit the status data of OTA update batches

Submit device status

Topic: `/as/mqtt/status/${productKey}/${deviceName}`

You can use this topic to obtain the online status or offline status of devices.

Format of data for an online device:

```
{
    "status":"online",
    "iotId":"4z819VQHk6VSLmmBJfrf00107e****",
    "productKey":"al12345****",
    "deviceName":"deviceName1234",
    "time":"2018-08-31 15:32:28.205",
    "utcTime":"2018-08-31T07:32:28.205Z",
    "lastTime":"2018-08-31 15:32:28.195",
    "utcLastTime":"2018-08-31T07:32:28.195Z",
    "clientIp":"192.0.2.1"
}
```

Format of data for an offline device:

```
{
    "status":"offline",
    "iotId":"4z819VQHk6VSLmmBJfrf00107e****",
    "offlineReasonCode":427,
    "productKey":"al12345****",
    "deviceName":"deviceName1234",
    "time":"2018-08-31 15:32:28.205",
    "utcTime":"2018-08-31T07:32:28.205Z",
    "lastTime":"2018-08-31 15:32:28.195",
    "utcLastTime":"2018-08-31T07:32:28.195Z",
    "clientIp":"192.0.2.1"
}
```

The following table describes the parameters.

Parameter	Category	Description
status	String	The device status. Valid values: • *online*: The device is online. • *offline*: The device is offline.
iotId	String	The ID of the device in IoT Platform.

Parameter	Category	Description
offlineReasonCode	Integer	The error code returned if the device goes offline. For more information, see Device behavior-related error codes.
productKey	String	The ProductKey of the product to which the device belongs.
deviceName	String	The DeviceName of the device.
lastTime	String	These parameters are no longer valid.
utcLastTime	String	
Time	String	The time when the device was connected or disconnected. The time when a message is received from a device is different from the time when the device is connected or disconnected. You can sort messages by the value of the Time parameter. For example, you obtain the following messages in sequence: 1. Online time: `2018-08-31 10:02:28.195` 2. Offline time: `2018-08-31 10:01:28.195` 3. Offline time: `2018-08-31 10:03:28.195` The preceding three messages indicate that the status of a device changed to offline, online, and then offline.
utcTime	String	The time when the device was connected or disconnected. The time is in the UTC format.
clientIp	String	The public IP address of the device.

Submit device properties

Topic: `/${productKey}/${deviceName}/thing/event/property/post`

You can use this topic to obtain the properties that are submitted by devices.

Data format:

```
{
    "iotId":"4z819VQHk6VSLmmBJfrf00107e****",
    "requestId":"2",
    "productKey":"al12345****",
    "deviceName":"deviceName1234",
    "gmtCreate":1510799670074,
    "deviceType":"Ammeter",
    "items":{
        "Power":{
            "value":"on",
            "time":1510799670074
        },
        "Position":{
            "time":1510292697470,
            "value":{
                "latitude":39.9,
                "longitude":116.38
            }
        }
    },
    "checkFailedData":{
        "attribute_8":{
            "time": 1510292697470,
            "value": 715665571,
            "code":6304,
            "message":"tsl parse: params not exist -> attribute_8"
        }
    }
}
```

The following table describes the parameters.

Parameter	Category	Description
iotId	String	The ID of the device in IoT Platform.
requestId	String	The ID of the message. Valid values: 0 to 4294967295. Each message ID must be unique for the device.
productKey	String	The ProductKey of the product to which the device belongs.
deviceName	String	The DeviceName of the device.
gmtCreate	Long	The time when the message was generated.
deviceType	String	The product category of the device.
items	Object	The data that is submitted by the device.

Parameter	Category	Description
Power	String	The identifier of the property. For more information about properties, see the TSL model of the product. If you use the properties of a custom module, the identifier of each property is in the `Module identifier:Property identifier` format. A colon is used to connect the two parts of this expression. The following example shows the data format that is used if the identifier of a TSL custom module is named test: `"items":{ "test:Power":{ "value":"on", "time":1510799670074 }, "test:Position":{ "time":1510292697470, "value":{ "latitude":39.9, "longitude":116.38 } } }`
Position		
attribute_8		
checkFailedData	Object	The data that failed to be verified.
value	Subject to the TSL definition	The value of the resource property.
time	Long	The time when the property was submitted. If the device does not submit a timestamp, the timestamp when IoT Platform receives the message is used.
code	Integer	The error code returned if the data fails to be verified. For more information, see Error codes for devices.
message	String	The error message returned if the data fails to be verified. The message includes the failure cause and the invalid parameters.

Submit device events

Topic: `/${productKey}/${deviceName}/thing/event/${tsl.event.identifier}/post`

You can use this topic to obtain the events that are submitted by devices.

Data format:

```
{
    "identifier":"BrokenInfo",
    "name":"Damage rate report",
    "type":"info",
    "iotId":"4z819VQHk6VSLmmBJfrf00107e****",
    "requestId":"2",
    "productKey":"X5eCzh6****",
    "deviceName":"5gJtxDVeGAkaEztpisjX",
    "gmtCreate":1510799670074,
    "value":{
        "Power":"on",
        "Position":{
            "latitude":39.9,
            "longitude":116.38
        }
    },
    "checkFailedData":{
    },
    "time":1510799670074
}
```

The following table describes the parameters.

Parameter	Category	Description
identifier	String	The identifier of the event. If you use the events of a custom module, the identifier of each property is in the `Module identifier:Event identifier` format. A colon is used to connect the two parts of this expression. The following example shows the data format if the identifier of a custom TSL module is test: `"test:identifier":"BrokenInfo",`
name	String	The name of the event.
type	String	The type of the event. For more information about the supported event types, see the TSL model of the product.
iotId	String	The ID of the device in IoT Platform.
requestId	String	The ID of the message. Valid values: 0 to 4294967295. Each message ID must be unique for the device.
productKey	String	The ProductKey of the product to which the device belongs.
deviceName	String	The DeviceName of the device.
gmtCreate	Long	The time when the message was generated.

Parameter	Category	Description
value	Object	The output parameters of the event. Examples: Power and Position. `{ "Power":"on", "Position":{ "latitude":39.9, "longitude":116.38 } }` **Notice** • Only the output parameters that pass verification are included in the value parameter. In this case, the checkFailedData parameter is empty. • Only the output parameters that fail to verified are included in the checkFailedData parameter. In this case, the value parameter is empty.

Parameter	Category	Description

checkFailedData	Object	The data that failed to be verified.
		If the output parameters fail to be verified, the checkFailedData parameter includes the following parameters:
		<pre>{
		"value":{
		"Power":"on",
		"Position":{
		"latitude":39.9,
		"longitude":116.38
		}
		}
		"time":1524448722000,
		"code":6304,
		"message":"tsl parse: params not exist ->
		type"
		}</pre>
		Description:
		• value: Object type. The output parameters of the event.
		• time: Long type. The timestamp when the event was generated.
		• code: Integer type. The error code returned if the data fails to be verified. For more information, see Error codes for devices.
		• message: String type. The error message returned if the data fails to be verified. The message includes the failure cause and the invalid parameters.

time	Long	The time when the event was submitted. If the device does not submit a timestamp, the timestamp when IoT Platform receives the message is used.

Submit device properties in batches

Topic: `/${productKey}/${deviceName}/thing/property/batch/post`

You can use this topic to obtain the properties that are submitted by devices in batches.

Data format:

```
{
    "productKey": "al12345****",
    "deviceName": "deviceName1234",
    "instanceId": "iot-0***",
    "requestId": "2",
    "payload": {
        "Power": [{
                "value": "on",
                "time": 1524448722000
            },
            {
                "value": "off",
                "time": 1524448722001
            }
        ],
        "WF": [{
                "value": 3,
                "time": 1524448722000
            },
            {
                "value": 4,
                "time": 1524448722009
            }
        ]
    }
}
```

The following table describes the parameters.

Parameter	Category	Description
productKey	String	The ProductKey of the product to which the device belongs.
deviceName	String	The DeviceName of the device.
instanceId	String	The ID of the instance to which the device belongs.
requestId	String	The ID of the message. Valid values: 0 to 4294967295. Each message ID must be unique for the device.
payload	Object	The data that is submitted by the device.

Parameter	Category	Description
Power	String	The identifier of the property. For more information about properties, see the TSL model of the product. If you use the properties of a custom module, the identifier of each property is in the `Module identifier:Property identifier` format. A colon is used to connect the two parts of this expression. The following example shows the data format that is used if the identifier of a TSL custom module is named test: `"payload":{ "test:Power":[ { "value":"on", "time":1510799670074 }, { "value": "off", "time": 1524448722001 }], "test:WF":[ { "value": 3, "time": 1524448722000 }, { "value": 4, "time": 1524448722009 }] } }`
WF		
value	Subject to the TSL definition	The value of the resource property.
time	Long	The time when the property was submitted. If the device does not submit a timestamp, the timestamp when IoT Platform receives the message is used.

Submit device events in batches

Topic: `/${productKey}/${deviceName}/thing/event/batch/post`

You can use this topic to obtain the events that are submitted by devices in batches.

Data format:

```
{
    "productKey": "al12345****",
    "deviceName": "deviceName1234",
    "instanceId": "iot-0***",
    "requestId": "2",
    "payload": {
        "alarmEvent": [{
                "value": {
                    "Power": "on",
                    "WF": "2"
                },
                "time": 1524448722000
            },
            {
                "value": {
                    "Power": "on",
                    "WF": "2"
                },
                "time": 1524448723000
            }
        ]
    }
}
```

The following table describes the parameters.

Parameter	Category	Description
productKey	String	The ProductKey of the product to which the device belongs.
deviceName	String	The DeviceName of the device.
instanceId	String	The ID of the instance to which the device belongs.
requestId	String	The ID of the message. Valid values: 0 to 4294967295. Each message ID must be unique for the device.
payload	Object	The data that is submitted by the device.
alarmEvent	List	The identifier of the event.
value	Object	The parameters of the event. Examples: Power and WF.
time	Long	The time when the event was submitted. If the device does not submit a timestamp, the timestamp when IoT Platform receives the message is used.

Submit lifecycle changes

Topic: `/${productKey}/${deviceName}/thing/lifecycle`

You can use this topic to receive notifications when devices are created, deleted, enabled, or disabled.

Data format:

```
{
    "action": "create|delete|enable|disable",
    "iotId": "4z819VQHk6VSLmmBJfrf00107e****",
    "productKey": "al5eCzh****",
    "deviceName": "5gJtxDVeGAkaEztpisjX",
    "deviceSecret": "wsde***",
    "messageCreateTime": 1510292739881
}
```

The following table describes the parameters.

Parameter	Category	Description
action	String	• create: creates a device • delete: deletes a device • enable: enables a device • disable: disables a device
iotId	String	The ID of the device in IoT Platform.
productKey	String	The ProductKey of the product to which the device belongs.
deviceName	String	The DeviceName of the device.
deviceSecret	String	The DeviceSecret of the device. This parameter is available only if the action parameter is set to create.
messageCreateTime	Integer	The timestamp that was generated for the message. Unit: milliseconds.

Submit topology changes

Topic: `/${productKey}/${deviceName}/thing/topo/lifecycle`

You can use this topic to receive notifications when topological relationships between sub-devices and gateways are created or deleted.

Data format:

```
{
    "action" : "create|delete|enable|disable",
    "gwIotId": "dfaejVQHk6VSLmmBJfrf00107e****",
    "gwProductKey": "al5eCzh****",
    "gwDeviceName": "deviceName1234",
    "devices": [
        {
          "iotId": "4z819VQHk6VSLmmBJfrf00107e****",
          "productKey": "ala4Czh****",
          "deviceName": "deviceName1234"
       }
    ],
    "messageCreateTime": 1510292739881
}
```

The following table describes the parameters.

Parameter	Category	Description
action	String	• create: creates a topology • delete: deletes a topology • enable: enables a topology • disable: disables a topology
gwIotId	String	The ID of the gateway in IoT Platform.
gwProductKey	String	The ProductKey of the product to which the gateway belongs.
gwDeviceName	String	The name of the gateway.
devices	Object	The sub-devices whose topologies are changed.
iotId	String	The ID of each sub-device in IoT Platform.
productKey	String	The ProductKey of the product to which the sub-device belongs.
deviceName	String	The name of the sub-device.
messageCreateTime	Integer	The timestamp that was generated for the message. Unit: milliseconds.

Submit information about detected sub-devices

Topic: `/${productKey}/${deviceName}/thing/list/found`

In some cases, gateways can detect sub-devices and submit sub-device information. You can use this topic to obtain the submitted information.

Data format:

```
{
    "gwIotId":"dfaew9VQHk6VSLmmBJfrf00107e****",
    "gwProductKey":"al12345****",
    "gwDeviceName":"deviceName1234",
    "devices":[
        {
            "iotId":"4z819VQHk6VSLmmBJfrf00107e****",
            "productKey":"alr56g9****",
            "deviceName":"deviceName1234"
        }
    ]
}
```

The following table describes the parameters.

Parameter	Category	Description
gwIotId	String	The ID of the gateway in IoT Platform.
gwProductKey	String	The ProductKey of the product to which the gateway belongs.
gwDeviceName	String	The name of the gateway.
devices	Object	The sub-devices that are detected by the gateway.
iotId	String	The ID of each sub-device in IoT Platform.
productKey	String	The ProductKey of the product to which the sub-device belongs.
deviceName	String	The name of the sub-device.

Submit responses to downstream requests

Topic: `/${productKey}/${deviceName}/thing/downlink/reply/message`

You can use this topic to obtain the results that are returned after devices process downstream requests. IoT Platform sends the downstream requests to the devices in an asynchronous manner. If errors occur when IoT Platform sends the downstream requests, you can also use this topic to obtain error messages.

Data format:

```
{
    "gmtCreate":1510292739881,
    "iotId":"4z819VQHk6VSLmmBJfrf00107e****",
    "productKey":"al12355****",
    "deviceName":"deviceName1234",
    "requestId":"2",
    "code":200,
    "message":"success",
    "topic":"/sys/al12355****/deviceName1234/thing/service/property/set",
    "data":{
    },
    "checkFailedData":{
        "value": {
            "PicID": "15194139"
        },
        "code":6304,
        "message":"tsl parse: params not exist -> PicID"
    }
}
```

The following table describes the parameters.

Parameter	Category	Description
gmtCreate	Long	The timestamp in UTC.
iotId	String	The ID of the device in IoT Platform.
productKey	String	The ProductKey of the product to which the device belongs.
deviceName	String	The DeviceName of the device.
requestId	String	The ID of the message. Valid values: 0 to 4294967295. Each message ID must be unique for the device.
code	Integer	The response code that is returned by the device. For more information, see the following Response codes table.
message	String	The message that is returned by the device.
topic	String	The information about the topic that is used to send the downstream request.
data	Object	The data that is returned by the device. If the device returns Alink data, you do not need to parse the data. If the device returns pass-through data, you must parse the data by using a data parsing script.
checkFailedData	Object	The data that failed to be verified.

Parameter	Category	Description
value	Subject to the TSL definition	The property or service parameter whose value failed to be verified. Example: PicID.
code	Integer	The error code returned if the data fails to be verified. For more information, see Error codes for devices.
message	String	The error message returned if the data fails to be verified. The message includes the failure cause and the invalid parameters.

Response codes

code	message	Description
200	success	The message returned because the request is successful.
400	request error	The error message returned because an internal error occurred.
460	request parameter error	The error message returned because the request parameter is invalid and the device failed to verify the parameter.
429	too many requests	An excessive number of requests are submitted within a specific period of time.
9200	device not actived	The device is not activated.
9201	device offline	The device is offline.
403	request forbidden	The error message returned because the request is rejected due to overdue payments.

For more information about how to troubleshoot errors, see Error codes for devices.

Submit historical properties

Topic: `/${productKey}/${deviceName}/thing/event/property/history/post`

You can use this topic to obtain historical properties that are submitted by devices.

Data format:

```
{
    "iotId":"4z819VQHk6VSLmmBJfrf00107e****",
    "requestId":"2",
    "productKey":"12345****",
    "deviceName":"deviceName1234",
    "gmtCreate":1510799670074,
    "deviceType":"Ammeter",
    "items":{
        "Power":{
            "value":"on",
            "time":1510799670074
        },
        "Position":{
            "time":1510292697470,
            "value":{
                "latitude":39.9,
                "longitude":116.38
            }
        }
    },
    "checkFailedData":{
        "attribute_8":{
            "time": 1510292697470,
            "value": 715665571,
            "code":6304,
            "message":"tsl parse: params not exist -> attribute_8"
        }
    }
}
```

The following section describes the parameters:

Parameter	Category	Description
iotId	String	The ID of the device in IoT Platform.
requestId	String	The ID of the message. Valid values: 0 to 4294967295. Each message ID must be unique for the device.
productKey	String	The ProductKey of the product to which the device belongs.
deviceName	String	The DeviceName of the device.
gmtCreate	Long	The time when the message was generated.
deviceType	String	The product category of the device.
items	Object	The data that is submitted by the device.

Parameter	Category	Description
Power	String	The identifier of the property. For more information about properties, see the TSL model of the product. If you use the properties of a custom module, the identifier of each property is in the `Module identifier:Property identifier` format. A colon is used to connect the two parts of this expression. The following example shows the data format that is used if the identifier of a TSL custom module is named test: `"items":{ "test:Power":{ "value":"on", "time":1510799670074 }, "test:Position":{ "time":1510292697470, "value":{ "latitude":39.9, "longitude":116.38 } } }`
Position		
attribute_8		
checkFailedData	Object	The data that failed to be verified.
value	Subject to the TSL definition	The value of the resource property.
time	Long	The time when the property was submitted. If the device does not submit a timestamp, the timestamp when IoT Platform receives the message is used.
code	Integer	The error code returned if the data fails to be verified. For more information, see Error codes for devices.
message	String	The error message returned if the data fails to be verified. The message includes the failure cause and the invalid parameters.

Submit historical events

Topic: `/${productKey}/${deviceName}/thing/event/${tsl.event.identifier}/history/post`

You can use this topic to obtain historical events that are submitted by devices.

Data format:

```
{
    "identifier":"BrokenInfo",
    "name":"Damage rate report",
    "type":"info",
    "iotId":"4z819VQHk6VSLmmBJfrf00107e***",
    "requestId":"2",
    "productKey":"X5eCzh6***",
    "deviceName":"5gJtxDVeGAkaEztpisjX",
    "value":{
        "Power":"on",
        "Position":{
            "latitude":39.9,
            "longitude":116.38
        }
    },
    "checkFailedData":{
    },
    "time":1510799670074
}
```

The following section describes the parameters:

Parameter	Category	Description
identifier	String	The identifier of the event. If you use the events of a custom module, the identifier of each property is in the `Module identifier:Event identifier` format. The following example shows the data format if the identifier of a custom TSL module is test: `"test:identifier":"BrokenInfo",`
name	String	The name of the event.
type	String	The type of the event. For more information about the supported event types, see the TSL model of the product.
iotId	String	The ID of the device in IoT Platform.
requestId	String	
productKey	String	The ProductKey of the product to which the device belongs.
deviceName	String	The DeviceName of the device.
gmtCreate	Long	The time when the message was generated.

Parameter	Category	Description
value	Object	The output parameters of the event. Examples: Power and Position. `{ "Power":"on", "Position":{ "latitude":39.9, "longitude":116.38 } }` **Notice** • Only the output parameters that pass verification are included in the value parameter. In this case, the checkFailedData parameter is empty. • Only the output parameters that fail to verified are included in the checkFailedData parameter. In this case, the value parameter is empty.

Parameter	Category	Description

checkFailedData	Object	The data that failed to be verified.
		If the output parameters fail to be verified, the checkFailedData parameter includes the following parameters:
		<pre>{
		"value":{
		"Power":"on",
		"Position":{
		"latitude":39.9,
		"longitude":116.38
		}
		}
		"time":1524448722000,
		"code":6304,
		"message":"tsl parse: params not exist ->
		type"
		}</pre>
		Description:
		• value: Object type. The output parameters of the event.
		• time: Long type. The timestamp when the event was generated.
		• code: Integer type. The error code returned if the data fails to be verified. For more information, see Error codes for devices.
		• message: String type. The error message returned if the data fails to be verified. The message includes the failure cause and the invalid parameters.

time	Long	The time when the event was submitted. If the device does not submit a timestamp, the timestamp when IoT Platform receives the message is used.

Submit the status data of over-the-air (OTA) updates

Topic: `/${productKey}/${deviceName}/ota/upgrade`

You can use this topic to receive notifications when OTA updates succeed or fail.

> **Note** If an update task on a device is pending and you initiate another batch update task on the device, the most recent update task fails. In this case, the notification that is generated when the most recent update task fails is not sent to IoT Platform.

Data format:

```
{
    "iotId": "4z819VQHk6VSLmmBJfrf00107e****",
    "productKey": "X5eCzh6****",
    "deviceName": "deviceName1234",
    "moduleName": "default",
    "status": "SUCCEEDED|FAILED|CANCELED",
    "messageCreateTime": 1571323748000,
    "srcVersion": "1.0.1",
    "destVersion": "1.0.2",
    "desc": "success",
    "jobId": "wahVIzGkCMuAUE2gDERM02****",
    "taskId": "y3tOmCDNgpR8F9jnVEzC01****"
}
```

The following table describes the parameters.

Parameter	Category	Description
iotId	String	The ID of the device.
productKey	String	The ProductKey of the product to which the device belongs.
deviceName	String	The DeviceName of the device.
moduleName	String	The name of the OTA module.
status	String	The status of the update. Valid values: • *SUCCEEDED*: The update is successful. • *FAILED*: The update failed. • *CANCELED*: The update task is canceled.
messageCreateTime	Long	The timestamp that was generated for the message. Unit: milliseconds.
srcVersion	String	The firmware version before the update.
destVersion	String	The firmware version after the update.
desc	String	The description of the update.
jobId	String	The ID of the update batch. This parameter is used to uniquely identify the update batch.
taskId	String	The ID of the device update record.

Submit the progress data of OTA updates

Topic: `/${productKey}/${deviceName}/ota/progress/post`

You can use this topic to obtain the progress data of OTA updates.

Data format:

```
{
    "iotId": "4z819VQHk6VSLmmBJfrf00107e****",
    "productKey": "X5eCzh6****",
    "deviceName": "deviceName1234",
    "moduleName":"default",
    "status":"IN_PROGRESS",
    "step": "90",
    "messageCreateTime": 1571323748000,
    "srcVersion":"1.0.1",
    "destVersion":"1.0.2",
    "desc": "success",
    "jobId": "wahVIzGkCMuAUE2gDERM02****",
    "taskId": "y3tOmCDNgpR8F9jnVEzC01****"
}
```

The following table describes the parameters.

Parameter	Category	Description
iotId	String	The ID of the device.
productKey	String	The ProductKey of the product to which the device belongs.
deviceName	String	The DeviceName of the device.
moduleName	String	The name of the OTA module.
status	String	The status of the update. Set the value to IN_PROGRESS.
step	Integer	The progress of the update.
messageCreateTime	Long	The timestamp that was generated for the message. Unit: milliseconds.
srcVersion	String	The firmware version before the update.
destVersion	String	The firmware version after the update.
desc	String	The description of the update.
jobId	String	The ID of the update batch. This parameter is used to uniquely identify the update batch.
taskId	String	The ID of the device update record.

Submit OTA module versions

Topic: `/${productKey}/${deviceName}/ota/version/post`

You can use this topic to obtain OTA module versions that are submitted by devices. If the submitted versions are different from the previous versions, the messages that are sent to the topic are forwarded.

Data format:

```
{
    "iotId": "4z819VQHk6VSLmmBJfrf00107e****",
    "deviceName": "deviceName1234",
    "productKey": "X5eCzh6****",
    "moduleName": "BarcodeScanner",
    "moduleVersion": "1.0.3",
    "messageCreateTime": 1571323748000
}
```

The following table describes the parameters.

Parameter	Category	Description
iotId	String	The ID of the device.
productKey	String	The ProductKey of the product to which the device belongs.
deviceName	String	The DeviceName of the device.
moduleName	String	The name of the module.
moduleVersion	String	The version number of the module.
messageCreateTime	Long	The timestamp that was generated for the message. Unit: milliseconds.

Submit the status data of OTA update batches

Topic: `/${productKey}/${packageId}/${jobId}/ota/job/status`

You can use this topic to obtain the status data of OTA update batches.

Data format:

```
{
    "productKey": "X5eCzh6****",
    "moduleName": "BarcodeScanner",
    "packageId": "wahVIzGkCMuAUE2***",
    "jobId": "wahVIzGkCMuAUE2gDERM02****",
    "state": "IN_PROGRESS",
    "messageCreateTime": 1571323748000
}
```

The following table describes the parameters.

Parameter	Category	Description
productKey	String	The ProductKey of the product to which the device belongs.
moduleName	String	The name of the module.

Parameter	Category	Description
packageId	String	The ID of the update package. The value of this parameter is the same as the value of the FirmwareId parameter that is returned when you call the CreateOTAFirmware operation to create an update package.
jobId	String	The ID of the update batch. This parameter is used to uniquely identify the update batch.
state	String	The status of the update batch. Valid values: • PLANNED: The update batch is not started. • IN_PROGRESS: The update batch is running. • COMPLETED: The update is complete. • CANCELED: The update is canceled.
messageCreateTime	Long	The timestamp that was generated for the message. Unit: milliseconds.

Submit device tag changes

Topic: `/${productKey}/${deviceName}/thing/deviceinfo/update`

You can use this topic to receive notifications when a device tag is changed.

Data format:

```
{
  "action": "UPDATE|DELETE|DELETEALL"
  "iotId": "4z819VQHk6VSLmmBJfrf00107e****",
  "productKey": "X5eCzh6****",
  "deviceName": "deviceName1234",
  "deletedAttrKeyList": ["abc", "def", "rng"],
  "value": [
    {
      "attrKey": "tagKey",
      "attrValue": "tagValue"
    }
  ],
  "messageCreateTime": 1510799670074
}
```

The following table describes the parameters.

Parameter	Category	Description
action	String	The type of the tag change. Valid values: • UPDATE: The tag is updated or created. • DELETE: The tag is deleted. • DELETEALL: All tags are deleted.

Parameter	Category	Description
iotId	String	The ID of the device.
productKey	String	The ProductKey of the product to which the device belongs.
deviceName	String	The DeviceName of the device.
deletedAttrKeyList	List	The keys of the deleted tags. This parameter is available only if the action parameter is set to DELETE.
value	List	The data of the tag.
attrKey	String	The key of the tag.
attrValue	String	The value of the tag.
messageCreateTime	Long	The timestamp that was generated for the message. Unit: milliseconds.

7.Scene orchestration

7.1. What is scene orchestration?

Scene orchestration is a feature that is provided by the rules engine of IoT Platform. Scene orchestration allows you to create automated workflows in a visualized manner. You can define scene orchestration rules to configure complex interactions between devices and then deploy the rules in the cloud or at the edge.

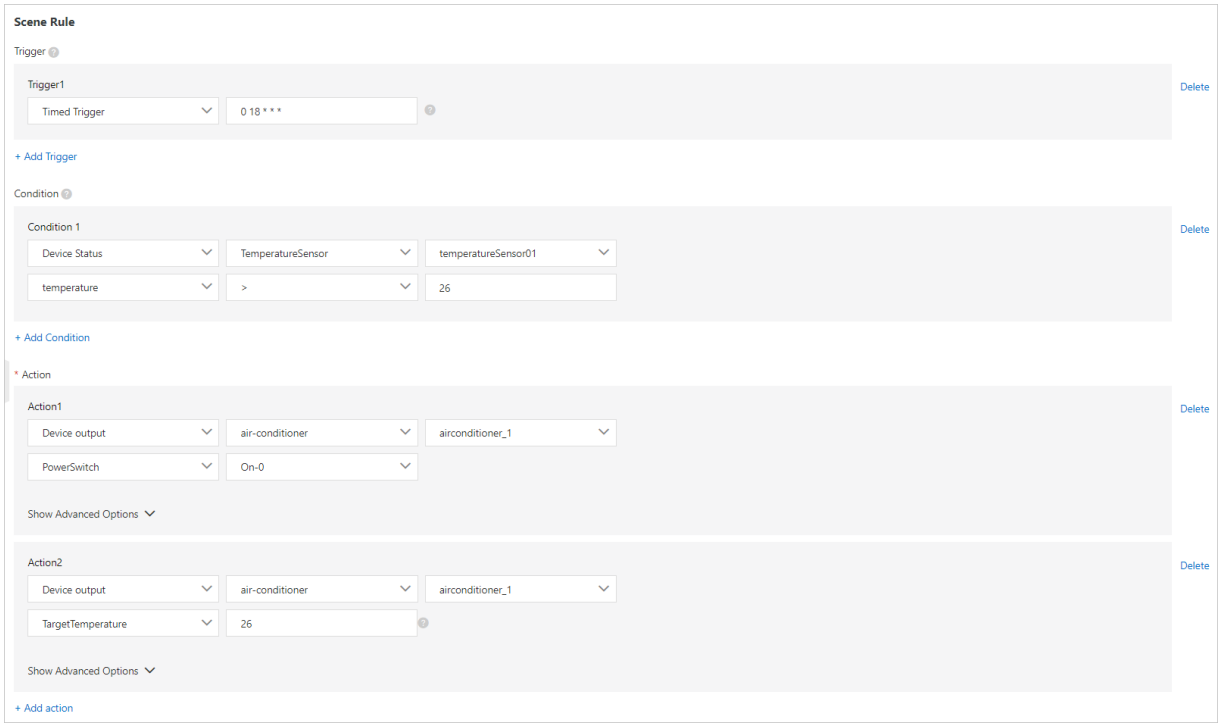
To create a scene orchestration rule, perform the following steps: Log on to the IoT Platform console and choose **Rules > Scene Orchestration**. On the page that appears, create a scene orchestration rule. Each scene orchestration rule consists of triggers, conditions, and actions. This rule model is called the TCA model. TCA is short for triggers, conditions, and actions.

> **Note** Only public instances that were activated before July 30, 2021 and Enterprise Edition instances in the China (Shanghai) region support the scene orchestration feature.

When an event or property change that is defined in a trigger occurs, IoT Platform determines whether to perform the actions that are defined in the rule based on the defined conditions. If the conditions are met, IoT Platform performs the defined actions. Otherwise, IoT Platform does not perform the actions.

Assume that you arrive home at 18:00 every day. In the summer, you hope that the indoor temperature is cool and comfortable when you get home. In this case, you can create a rule to automate the air conditioner.

The following figure shows the parameters.



The following table describes the parameters.

Parameter	Description	Logical operator
Trigger	This rule is triggered at 18:00 every day. For more information about how to write a CRON expression, see CRONTAB.	Or (||)
Condition	If the indoor temperature that is reported by a temperature sensor is higher than 26°C, the action is performed.	And (&&)
Action	Turn on the air conditioner and set the temperature to 26°C.	And (&&)

For more information about how to create scene orchestration rules, see Scene orchestrations in the cloud.

7.2. Scene orchestrations in the cloud

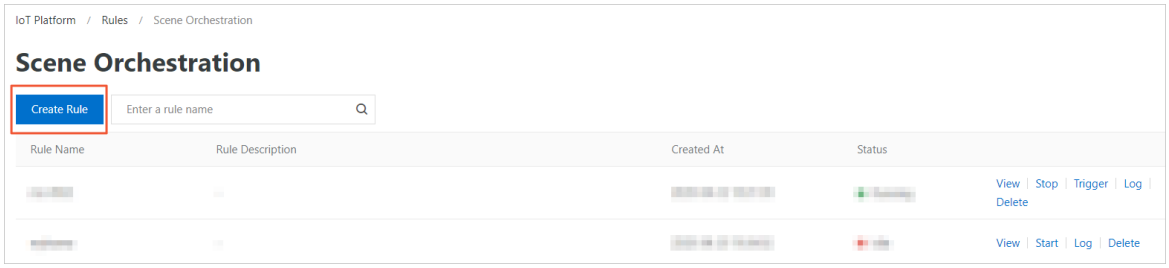
The rules for scene orchestration allow you to automate workflows based on the specified business logic in a visualized way. You can configure time or device-based triggers and conditions for rules. These rules can execute actions that change the statuses of other rules, devices, or functions. This allows you to implement scene orchestration for a large number of devices.

Prerequisites

An edge instance is created. For more information, see Set up environments.

Create scene rules

1. Log on to the IoT Platform console.
2. In the left-side navigation pane, choose **Rules > Scene Orchestration**.
3. Click **Create Rule**.



4. Set the parameters as required and click **OK**.

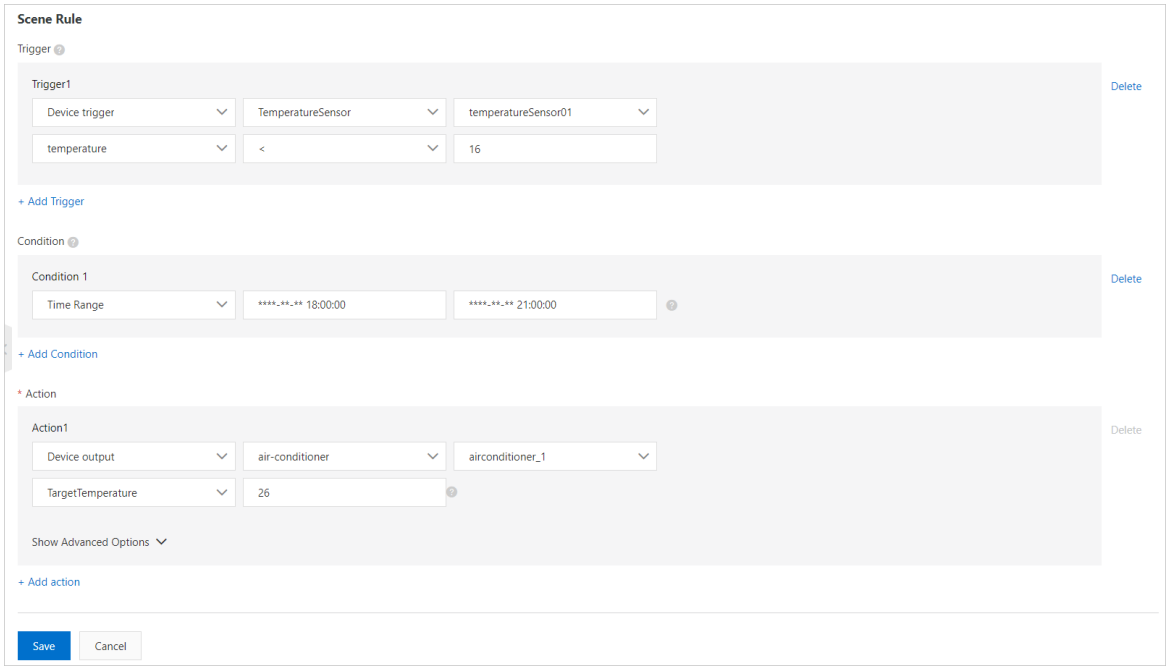
Parameter	Description
Rule Name	The name of the rule. The rule name must be 1 to 30 characters in length, and can contain letters, digits, underscores (_), and hyphens (-).
Rule Description	Optional. The description of the rule.

5. After the scene rule is created, click **Edit** in the message that appears to configure the rule.

   You can also configure the scene rule by clicking **View** next to the rule name.

   For example, you can use a scene rule to automate an air conditioner. When the indoor temperature that is reported by the temperature sensor between 12:00 and 23:59 is lower than 16°C, the air conditioner automatically raises the indoor temperature to 26°C.

The following figure shows the parameter configurations.



In the upper-right corner of the page, click **Edit** to change the name of the scene rule. For more information about other parameters, see the following table.

Parameter	Description
Trigger	The triggers that set off the rule. You can set this parameter to **Device Trigger** or **Timed Trigger**. When the reported device data or the current time meets the triggers, the system checks whether the conditions to trigger the rule are met. You can create one or more triggers for a rule. The rules are related by logic OR operations. ○ If you set this parameter to **Device Trigger**, you must select an existing product, one or all devices, and one or all properties or one or all events. ○ If you set this parameter to **Timed Trigger**, you must specify a point in time. The time must be specified by using a CRON expression. A CRON expression consists of the following five fields: the minute, hour, day, month, and day of a week. For the day of a week, a value of 0 or 7 indicates Sunday and values of 1 to 6 indicate Monday to Saturday. Separate these fields with spaces. For example, you can write a CRON expression to trigger a rule every day at 18:00 in the `0 18 * * *` format. You can also write a CRON expression to trigger a rule every Friday at 18:00 in the `0 18 * * 5` format. The asterisks (*) are wildcards. For more information about how to write a CRON expression, see CRONTAB. In the preceding example, this parameter is set to **Device Trigger**. In this case, the trigger refers to a scenario where the indoor temperature reported by the temperature sensor is lower than 16°C.

Parameter	Description
Condition	The set of conditions. The rule is triggered only when the data meets all conditions. You can set this parameter to **Device Status** or **Time Range**. You can create one or more conditions for a rule. The conditions are related by logical AND operations. ◦ If you set this parameter to **Device Status**, you must select an existing product, a device of the product, and a property or event of the device feature. ◦ If you set this parameter to **Time Range**, you must specify the start time and end time in the `yyyy-mm-dd hh24:mi:ss` format. In the preceding example, this parameter is set to **Time Range** and the rule can be triggered between 12:00 and 23:59.
Action	The action to be performed. You can create one or more actions. When an action fails, it does not affect other actions. ◦ If you set this parameter to **Device Output**, you must select an existing product, a device of the product, and a property or service of the device feature. Only writable properties or services can be selected to perform the action. IoT Platform performs the actions based on the defined device properties or services when both the triggers and the conditions for the rule are met. ◦ If you set this parameter to **Rule Output**, you must select another rule and the action will invoke the selected rule. The triggers for the invoked rule are skipped and only the conditions of the rule are checked. If the conditions are met, the actions that are defined in the invoked rule is performed. For example, if Rule A is triggered, the triggers of Rule A are skipped. The conditions of Rule A are checked. If all conditions of Rule A are met, IoT Platform performs the actions of Rule A. The conditions are related by logical AND operations. In the preceding example, this parameter is set to **Device Output**. In this case, when the rule is triggered, the air conditioner will raise the temperature to 26°C.
Delayed Execution	This parameter is displayed after you show the advanced options. After you set this parameter, the specified actions are executed after a specified period of time. Valid values: 0 to 86400. Unit: seconds.

Start scene rules

After a scene rule is created, you can start the scene rule on the **Scene Orchestration** page.

You can perform the following steps to start a scene rule:

1. Log on to the IoT Platform console. In the left-side navigation pane, choose **Rules > Scene Orchestration**.
2. Find the scene rule that you want to start and click **Start**. The rule status changes to **Running**.

After you start the rule:

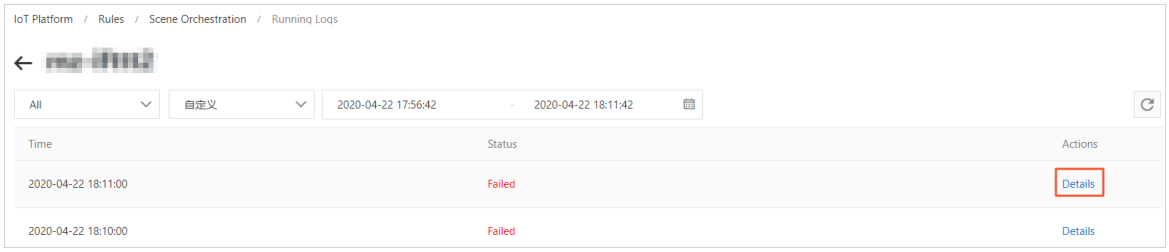
- If the scene rule runs on the cloud, you must configure message routing for the devices of the scene rule. This allows the device properties and events to be sent to IoT Hub that runs on the cloud. For more information about how to configure a message route, see Configure message routing.
- If the scene rule runs at the edge, you must stop running the scene rule in the cloud and associate the scene rule with the edge instance. For more information, see Other operations for scene orchestration.

View logs

You can view the logs of scene rules and view the results on the details page.

> **Note** A scene rule can run on the cloud and at the edge at the same time. In this case, to view all the logs of the scene rule, perform the following steps: Log on to the IoT Platform console and choose **Rules > Scene Orchestration**.

1. Log on to the IoT Platform console. In the left-side navigation pane, choose **Rules > Scene Orchestration**.
2. Find the scene rule that you want to view and click **Log** on the right.
3. Click **Details** to view the log details.



> **Note** If the status of a log is Failed, you can click **Details** in the Actions column to view the details of execution failures.

Other operations for scene orchestration

- Delete scene rules:
  i. On the **Scene Orchestration** page, find the rule that you want to delete.
  ii. Click **Delete** next to the rule name. In the message that appears, click **OK** to delete the scene rule.
- Trigger scene rules:

  You can trigger a scene rule after you start the rule.

  i. On the **Scene Orchestration** page, find the rule that you want to trigger.
  ii. Click **Trigger** next to the rule name, and the rule is manually triggered once. All actions of the rule are performed regardless of the specified triggers.

- Run a scene rule on an edge instance:

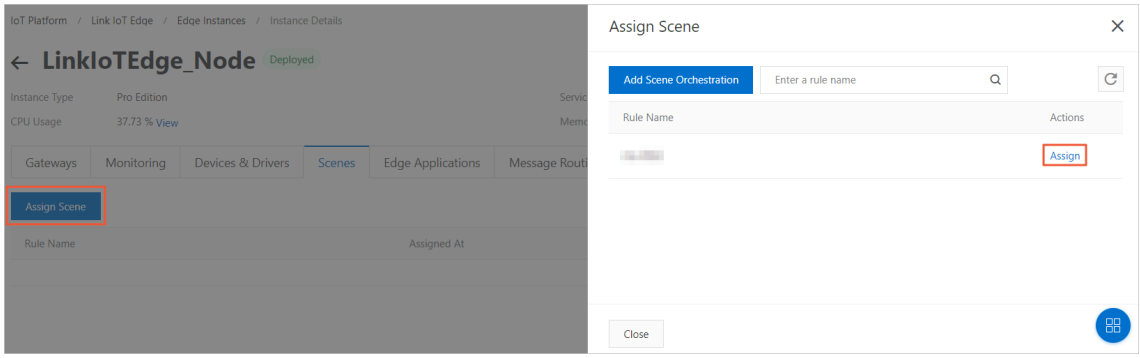
  Perform the following steps to deploy a scene rule on the edge instance.

  > **Notice** Make sure that you have stopped running the scene rule in the cloud.

  i. Log on to the Link IoT Edge console. In the left-side navigation pane, click **Edge Instances**. Find the edge instance that you created and click **View** in the Actions column.

  ii. On the **Instance Details** page, click the **Scenes** tab. Then, click **Assign Scene**.

  iii. In **Assign Scene** panel, click **Assign** next to the rule name. Then, click **Close**.



  iv. After the scene rule is assigned, redeploy the edge instance.

8.MQTT-based synchronous communication (RRPC)

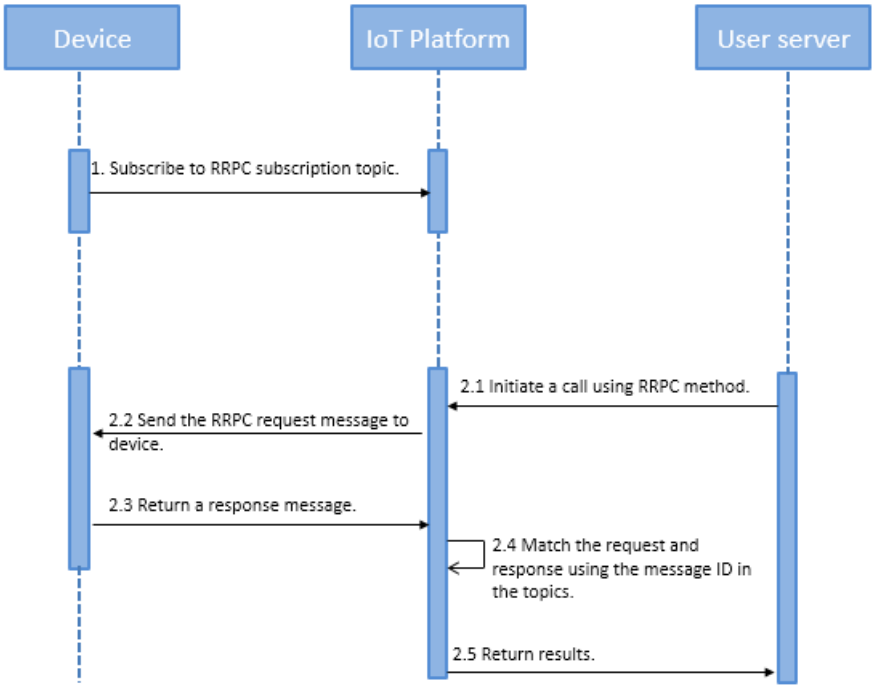
8.1. What is RRPC?

The Message Queuing Telemetry Transport (MQTT) protocol uses the asynchronous publish/subscribe model. This model is not applicable to the scenarios in which user servers need to synchronously control devices and obtain responses. IoT Platform provides a synchronous communication mechanism based on MQTT. You do not need to modify the MQTT protocol. IoT Platform provides the RRpc operation for user servers to send requests to devices. The devices only need to respond to the requests by using the specified topics and then the servers can synchronously obtain the responses.

Terms

Term	Description
RRPC	RRPC is short for revert-RPC. A remote procedure call (RPC) uses the client-server model. This model allows you to request a remote service without the need to understand the underlying protocol. An RRPC allows you to send a request from a server to a device and receive a response from the device.
RRPC subscription-specific topic	The topic to which devices subscribe to receive RRPC messages. The topic includes a wildcard.
RRPC request message	The message that IoT Platform sends to a device.
RRPC response message	The message that a device sends to IoT Platform as a response.
RRPC message ID	The ID that is generated by IoT Platform for each RRPC message

RRPC procedure



Procedure:

1. A device subscribes to the RRPC subscription-specific topic.
2. A user server calls the RRpc operation.
3. IoT Platform receives a server-side RRPC request and sends an RRPC request message to the device. The message body includes the payload that is sent from the user server. The topic that is used by the device to receive RRPC messages is predefined in IoT Platform. The topic includes the ID of the message.
4. After the device receives the message, the device sends an RRPC response message to IoT Platform by using the specified topic. The topic includes the ID of the request message.
5. IoT Platform extracts the response message ID to match the previous RRPC request message.
6. IoT Platform returns the response to the user server.

> **Note** When you call the RRpc operation on the user server, the following errors may occur:
> - The device is offline. In this case, IoT Platform returns an error message to the user server.
> - The device does not respond to the RRPC request within the timeout period (8 seconds). In this case, IoT Platform returns an error message to the user server.

RRPC-specific topics

You can implement RRPCs based on the following types of topics:

- Basic communication topics. For more information, see Use RRPC-specific topics.
- Custom topics. For more information, see Use custom topics.

For information about the example of implementing an RRPC, see Remotely control a Raspberry Pi server.

8.2. Use RRPC-specific topics

Revert-RPCs (RRPCs) support using RRPC-specific topics for communication between IoT Platform and devices. This article describes RRPC-specific topics and how to initiate an RRPC request.

RRPC-specific topics

This following table describes the syntax of the RRPC-specific topics.

Topic	Format	Description
RRPC subscription topic	`/sys/${YourProductKey}/${YourDeviceName}/rrpc/request/+`	Subscribes to RRPC request messages that are sent from IoT Platform.
RRPC request topic	`/sys/${YourProductKey}/${YourDeviceName}/rrpc/request/${messageId}`	Sends RRPC request messages from IoT Platform.
RRPC response topic	`/sys/${YourProductKey}/${YourDeviceName}/rrpc/response/${messageId}`	Sends RRPC response messages from devices.

Variables:

- *${YourProductKey}*: Replace this variable with the **ProductKey** of the product to which your device belongs.
- *${YourDeviceName}*: Replace this variable with the DeviceName of your device.
- *${messageId}*: Replace this variable with the ID of the RRPC message. When a server calls the RRpc operation to send a message to a device, IoT Platform generates a unique ID for the message.

Initiate an RRPC request

1. IoT Platform sends an RRPC message.

   A server calls the RRpc operation to send a message to a device. For more information, see RRpc.

   The following example shows how to use Link SDK for Java to call the RRpc operation:

   ```
   RRpcRequest request = new RRpcRequest();
   request.setProductKey("testProductKey");
   request.setDeviceName("testDeviceName");
   request.setRequestBase64Byte(Base64.getEncoder().encodeToString("hello world"));
   request.setTimeout(3000);
   RRpcResponse response = client.getAcsResponse(request);
   ```

   > **Note** Log on to the OpenAPI Developer Portal and call the RRpc operation. The sample code for multiple programming languages is provided.

2. The device sends an RRPC response.

   After the device receives the RRPC request, the device returns a message to the response topic.

   The device extracts the messageId parameter from the /sys/${YourProductKey}/${YourDeviceName}/rrpc/request/${messageId} request topic, and generates an RRPC response topic to send the response to IoT Platform.

   > **Note** Devices can return only the RRPC messages whose quality of service (QoS) level is 0.

Examples: Remotely control a Raspberry Pi server

8.3. Use custom topics

Revert-RPCs (RRPCs) support using custom topics for communication between IoT Platform and devices. An RRPC-specific topic includes an entire custom topic that is defined in IoT Platform. This article describes RRPC-specific topics that include custom topics and how to initiate an RRPC request.

Prerequisites

Devices are integrated with Link SDK for C.

Custom topics

Topic	Format	Description
RRPC subscription topic	`/ext/rrpc/+/${topic}`	Subscribes to RRPC request messages that are sent from IoT Platform.
RRPC request topic	`/ext/rrpc/${messageId}/${topic}`	Sends RRPC request messages from IoT Platform.
RRPC response topic	`/ext/rrpc/${messageId}/${topic}`	Sends RRPC response messages from devices.

Variables:

- *${topic}*: Replace this variable with your custom topic. For more information, see Custom topics.
- *${messageId}*: Replace this variable with the ID of the RRPC message. When a server calls the RRpc operation to send a message to a device, IoT Platform generates a unique ID for the message.

Initiate an RRPC request

1. IoT Platform sends an RRPC message.

   A server calls the RRpc operation to send a message to a device. For more information, see RRpc.

   In this example, IoT Platform SDK for Java is used as an example.

   To use a custom topic, make sure that the version of your SDK for Java (aliyun-java-sdk-iot) is 6.0.0 or later.

   ```
   <dependency>
       <groupId>com.aliyun</groupId>
       <artifactId>aliyun-java-sdk-iot</artifactId>
       <version>6.0.0</version>
   </dependency>
   ```

   The following example shows how to call the RRpc operation.

```
RRpcRequest request = new RRpcRequest();
request.setProductKey("testProductKey");
request.setDeviceName("testDeviceName");
request.setRequestBase64Byte(Base64.getEncoder().encodeToString("hello world"));
request.setTopic("/testProductKey/testDeviceName/user/get");// If you need to use a cus
tom topic for communication, specify the custom topic in this function.
request.setTimeout(3000);
RRpcResponse response = client.getAcsResponse(request);
```

Note Log on to the OpenAPI Developer Portal and call the RRpc operation. The sample code for multiple programming languages is provided.

2. Connect the device to IoT Platform.
   - For devices that use Link SDK for C, no additional configuration is required.
   -

```
clientId:`${this.clientId}|securemode=${this.securemode },signmethod=hmac${this.signA
lgorithm},timestamp=${this.timestamp},${extra}`,
```

```
clientId:`${this.clientId}|securemode=${this.securemode },signmethod=hmac${this.signA
lgorithm},timestamp=${this.timestamp},${extra},ext=1`,
```

3. The device sends an RRPC response.

   The format of an RRPC response topic is the same as the format of an RRPC request topic. Therefore, you can use an RRPC request topic as an RRPC response topic.

Note Devices can return only the RRPC messages whose quality of service (QoS) level is 0.

9.Broadcast messages

IoT Platform supports broadcast communication. You can broadcast a message to all devices under a product. In this case, the devices do not need to subscribe to a broadcast topic to receive the message. You can also broadcast a message to all devices that subscribe to a specified topic. A device must be online to receive the message that is sent by the server. This article describes how to broadcast a message to all online devices under a product.

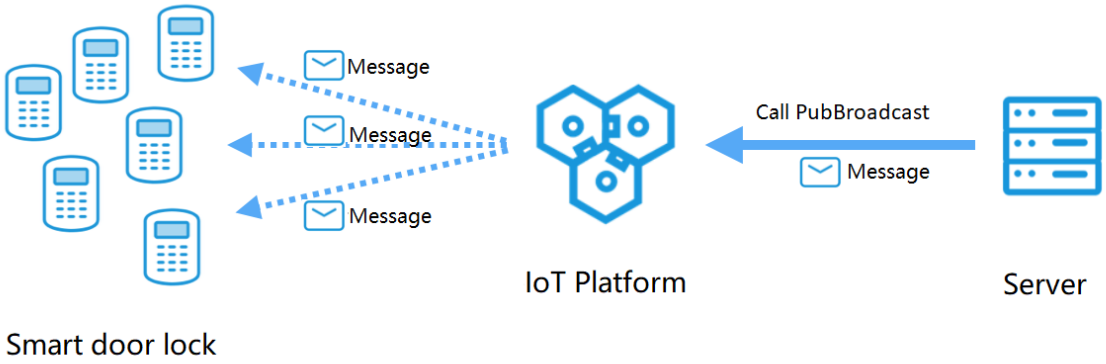
Context

- Broadcast a message to all online devices under a product

  A business server calls the PubBroadcast operation and specifies the ProductKey and **MessageContent** parameters. Then, all online devices receive the message from the following broadcast topic: `/sys/${productKey}/${deviceName}/broadcast/request/${MessageId}`.

  The message ID in the broadcast topic is generated by IoT Platform. After the message is sent, the message ID is returned to the business server that calls the PubBroadcast operation.

  For example, a manufacturer has multiple smart door locks that are connected to IoT Platform. The manufacturer uses a business server to send a command to all online devices to invalidate a password.



- Broadcast a message to all devices that subscribe to a specified topic

  The devices subscribe to the same broadcast topic. A business server calls the PubBroadcast operation and specifies the ProductKey, **MessageContent**, and Topic parameters. The format of a broadcast topic is `/broadcast/${productKey}/Custom field`. Then, all online devices receive the message from the topic.

  > **Notice**
  > - When you develop devices, use code to define a broadcast topic. You do not need to create a topic in the IoT Platform console.
  > - A maximum of 1,000 devices can subscribe to the same broadcast topic. If the number of devices exceeds the limit, you can divide the devices into groups. For example, you can divide 5,000 devices into five groups. Each group contains 1,000 devices. You must call the PubBroadcast operation five times, and each time set the custom field in the broadcast topic to group1, group2, group3, group4, and group5. Make sure that each group of devices subscribes to the required broadcast topic.

For more information about how to call the PubBroadcast operation, see PubBroadcast.

Limits

- Broadcast messages can be pushed only to online devices under a product.
- When you broadcast a message to specified online devices, specify the broadcast topic to which the devices subscribe. In this case, the maximum frequency of calling the PubBroadcast operation is once per second.
- When you broadcast a message to all online devices, the devices do not need to subscribe to a broadcast topic. In this case, the maximum frequency of calling the PubBroadcast operation is once per minute.
- The size of a message body can be up to 64 KB.

Notice Broadcast messages are not throttledby the limit of transactions per second (TPS) in messaging. For example, the limit of TPS in messaging is 100 per second and the number of current online devices is 500. Each time you call the PubBroadcast operation, the message is sent to 500 online devices.

Prepare the development environment

In this example, both devices and IoT Platform use SDKs for Java. You need to prepare the Java development environment first. You can download Java from the Java official website and install the Java development environment.

Add Maven dependencies

Create a project. In the *pom.xml* file, add the following Maven dependencies:

```
<dependencies>
  <dependency>
     <groupId>com.aliyun.alink.linksdk</groupId>
     <artifactId>iot-linkkit-java</artifactId>
     <version>1.2.0.1</version>
     <scope>compile</scope>
  </dependency>
  <dependency>
      <groupId>com.aliyun</groupId>
      <artifactId>aliyun-java-sdk-core</artifactId>
      <version>3.7.1</version>
  </dependency>
  <dependency>
      <groupId>com.aliyun</groupId>
      <artifactId>aliyun-java-sdk-iot</artifactId>
      <version>7.6.0</version>
  </dependency>
  <dependency>
    <groupId>com.aliyun.openservices</groupId>
    <artifactId>iot-client-message</artifactId>
    <version>1.1.2</version>
  </dependency>
</dependencies>
```

Create a product and devices

1. Log on to the IoT Platform console.
2.
3. In left-side navigation pane, choose **Devices > Products**.
4. Click **Create Product** to create a smart door lock product. For more information, see Create a product.
5. In the left-side navigation pane, choose **Devices > Devices**. Then, you can create three smart door lock devices under the product that is created. For more information, see Create multiple devices at a time.

Configure the device SDK

- Connect devices to IoT Platform.
  - Configure device authentication information.

    ```
    final String productKey = "<yourProductKey>";
    final String deviceName = "<yourDeviceName>";
    final String deviceSecret = "<yourDeviceSecret>";
    final String region = "<yourRegionID>";
    ```

    The productKey, deviceName, and deviceSecret parameters specify the device certificate information. To view the information, go to the IoT Platform console.In the left-side navigation pane, choose **Devices > Devices**. On the page that appears, find the device and click **View** in the Actions column. The information is displayed on the **Device Details** page.

    region: the ID of the region in which the device resides. For more information about region IDs, see Regions and zones.

  - Set the parameters to initialize a connection. These parameters include the MQTT connection parameters, device information, and initial device status.

    ```
    LinkKitInitParams params = new LinkKitInitParams();
    // Set the MQTT connection parameters. Link SDK uses MQTT as the underlying protocol.
    IoTMqttClientConfig config = new IoTMqttClientConfig();
    config.productKey = productKey;
    config.deviceName = deviceName;
    config.deviceSecret = deviceSecret;
    config.channelHost = productKey + ".iot-as-mqtt." + region + ".aliyuncs.com:1883";
    // Set the device information.
    DeviceInfo deviceInfo = new DeviceInfo();
    deviceInfo.productKey = productKey;
    deviceInfo.deviceName = deviceName;
    deviceInfo.deviceSecret = deviceSecret;
    // Set the initial status of the device.
    Map<String, ValueWrapper> propertyValues = new HashMap<String, ValueWrapper>();
    params.mqttClientConfig = config;
    params.deviceInfo = deviceInfo;
    params.propertyValues = propertyValues;
    ```

- Initialize the connection.

```
// Initialize the connection and configure the callback function that is used after the
initialization succeeds.
LinkKit.getInstance().init(params, new ILinkKitConnectListener() {
     @Override
     public void onError(AError aError) {
         System.out.println("Init error:" + aError);
     }
     // Implement the callback function.
     @Override
     public void onInitDone(InitResult initResult) {
         System.out.println("Init done:" + initResult);
     }
 });
```

- The onInitDone() callback function uses the prefix to identify broadcast topics. The syntax of a topic prefix is `/sys/${productKey}/${deviceName}/broadcast/request/` .

```
public void onInitDone(InitResult initResult) {
                // Configure a callback function that is used when downstream messages ar
e sent.
                IConnectNotifyListener notifyListener = new IConnectNotifyListener() {
                    // Configure the callback function that is used after the downstream
messages are received.
                    @Override
                    public void onNotify(String connectId, String topic, AMessage aMessag
e) {
                        // Filter the received broadcast messages.
                        if(topic.startsWith(broadcastTopic)){
                            System.out.println(
                                    "received broadcast message from topic=" + topic + ",
\npayload=" + new String((byte[])aMessage.getData()));
                        }
                    }
                    @Override
                    public boolean shouldHandle(String s, String s1) {
                        return false;
                    }
                    @Override
                    public void onConnectStateChange(String s, ConnectState connectState)
{

                    }
                };
                LinkKit.getInstance().registerOnNotifyListener(notifyListener);
            }
```

Configure IoT Platform SDK

Configure IoT Platform SDK for Java to broadcast a message.

- Specify identity information for verification.

```
String regionId = "<yourRegionID>";
String accessKey = "<yourAccessKey>";
String accessSecret = "<yourAccessSecret>";
final String productKey = "<yourProductKey>";
```

- Call the PubBroadcast operation of IoT Platform to broadcast a message.

```
// Set the parameters of the client.
DefaultProfile profile = DefaultProfile.getProfile(regionId, accessKey, accessSecret);
IAcsClient client = new DefaultAcsClient(profile);
PubBroadcastRequest request = new PubBroadcastRequest();
// Set the productKey parameter of the product.
request.setProductKey(productKey);
// Set the MessageContent parameter. The message content must be encoded in Base64. Other
wise, the message content appears as garbled characters.
request.setMessageContent(Base64.encode("{\"pwd\":\"2892nd6Y\"}"));
```

- Broadcast the message.

```
try {
    PubBroadcastResponse response = client.getAcsResponse(request);
System.out.println("broadcast pub success: broadcastId =" + response.getMessageId());
} catch (Exception e) {
    System.out.println(e);
}
```

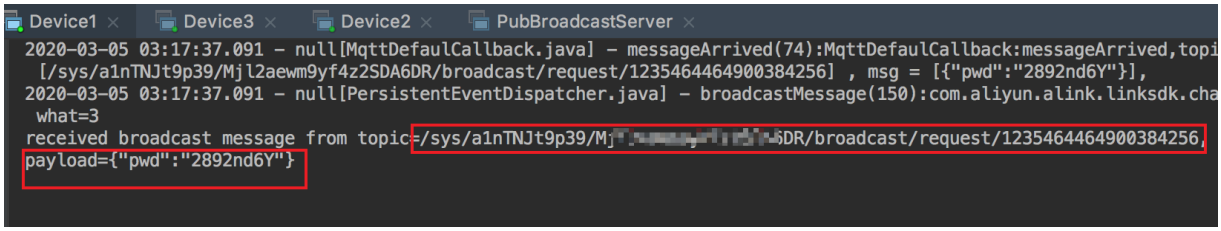
Verify the operation

Configure Link SDK on devices to connect the devices to IoT Platform. Then, configure IoT Platform SDK to call the PubBroadcast operation to broadcast a message to the devices.

Use IoT Platform SDK to send the following message to devices: `"{\"pwd\":\"2892nd6Y\"}"` .

The following message appears in the on-premises logs of the devices: `{\"pwd\":\"2892nd6Y\"}` .

Door Lock 1:



Door Lock 2:

Door Lock 3:

Appendix: Sample code

You can view the following sample codes of IoT Platform SDK and Link SDK:

- PubBroadcast Demo. The sample code is used to broadcast a message to all online devices.
- Broadcast Demo. The sample code is used to broadcast a message to all devices that subscribe to a specified topic.