IoT Device Access

SDK Reference

Issue 1.0

Date 2025-06-24

HUAWEI

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


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

Notice

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

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

Contents

1 Overview....1

2 SDKs for the Application Side....4

2.1 Application Java SDK....4
2.2 Application Python SDK....7
2.3 Application .NET SDK....9
2.4 Application Go SDK....11
2.5 Application Node.js SDK....13
2.6 Application PHP SDK....15

3 Device SDKs....18

3.1 Overview....18
3.2 IoT Device Java SDK....23
3.3 IoT Device C SDK for Linux/Windows....43
3.4 IoT Device C# SDK....44
3.5 IoT Device Android SDK....45
3.6 IoT Device Go SDK....46
3.7 IoT Device Tiny C SDK for Linux/Windows....46
3.8 IoT Device Python SDK....47
3.9 IoT Device ArkTS (OpenHarmony) SDK....48

1 Overview

IoTDA provides SDKs for the application and device sides so that devices can connect to the platform and applications can call platform APIs to implement secure access, device management, data collection, and command delivery.

NOTE

If you cannot access the GitHub repositories, check whether your network can access the public network.

Resource Package	Description	Download Link
Application Java SDK	You can use Java methods to call **application-side APIs** to communicate with the platform. For details, see **Java SDK**.	**Application Java SDK**
Application .NET SDK	You can use .NET methods to call **application-side APIs** to communicate with the platform. For details, see **.NET SDK**.	**Application .NET SDK**
Application Python SDK	You can use Python methods to call **application-side APIs** to communicate with the platform. For details, see **Python SDK**.	**Application Python SDK**
Application Go SDK	You can use Go methods to call **application-side APIs** to communicate with the platform. For details, see **Go SDK**.	**Application Go SDK**

Resource Package	Description	Download Link
Application Node.js SDK	You can use Node.js methods to call **application-side APIs** to communicate with the platform. For details, see **Node.js SDK**.	**Application Node.js SDK**
Application PHP SDK	You can use PHP methods to call **application-side APIs** to communicate with the platform. For details, see **PHP SDK**.	**Application PHP SDK**
IoT Device Java SDK	Devices can connect to the platform by integrating the IoT Device Java SDK. The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device Java SDK**.	**IoT Device Java SDK**
IoT Device C SDK for Linux/ Windows	Devices can connect to the platform by integrating the IoT Device C SDK for Linux/ Windows. The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device C SDK for Linux/ Windows**.	**IoT Device C SDK for Linux/Windows**
IoT Device C# SDK	Devices can connect to the platform by integrating the IoT Device C# SDK. The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device C# SDK**.	**IoT Device C# SDK**
IoT Device Android SDK	Devices can connect to the platform by integrating the IoT Device Android SDK. The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device Android SDK**.	**IoT Device Android SDK**

Resource Package	Description	Download Link
IoT Device SDK (Go Community Edition)	Devices can connect to the platform by integrating the IoT Device SDK (Go Community Edition). The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device SDK (Go Community Edition)**.	**IoT Device Go SDK (Community Edition)**
IoT Device Tiny C SDK for Linux/ Windows	Devices can connect to the platform by integrating the IoT Device Tiny C SDK for Linux/ Windows. The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device Tiny C SDK for Linux/Windows**.	**IoT Device Tiny C SDK for Linux/Windows**
IoT Device ArkTS (OpenHarmony) SDK	Devices can connect to the platform by integrating the IoT Device ArkTS (OpenHarmony) SDK. The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device ArkTS (OpenHarmony) SDK**.	**IoT Device ArkTS (OpenHarmony) SDK**
IoT Device Python SDK	Devices can connect to the platform by integrating the IoT Device Python SDK. The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device Python SDK**.	**IoT Device Python SDK**

2 SDKs for the Application Side

2.1 Application Java SDK

IoTDA provides an application SDK in Java for developers. This topic describes how to install and configure the Java SDK and how to use it to call **application-side APIs**.

SDK Obtaining and Installing

Step 1 Install the Java development environment.

Visit the **Java website**, and download and install the Java development environment.

NOTE

The Java SDK can be used in Java JDK 1.8 or later.

Step 2 Install Maven.

Download and **install Maven**. After Maven is installed, add the dependencies to the **pom.xml** file of the Java project.

Step 3 Install the Java SDK.

Add Maven dependencies.

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

----End

Code Sample

> **CAUTION**
>
> Use a version range for the Maven dependency versions. If you use a specific version, use 3.0.60 or later.

The following code sample shows how to use the Java SDK to call the API for **querying the device list**.

Step 1 Create a credential.

Step 2 Create and initialize an IoTDAClient instance.

Step 3 Instantiate a request object.

Step 4 Call the API for querying the device list.

```
package com.huaweicloud.sdk.test;

import com.huaweicloud.sdk.core.auth.ICredential;
import com.huaweicloud.sdk.core.exception.ConnectionException;
import com.huaweicloud.sdk.core.exception.RequestTimeoutException;
import com.huaweicloud.sdk.core.exception.ServiceResponseException;
import com.huaweicloud.sdk.core.region.Region;
import com.huaweicloud.sdk.core.auth.BasicCredentials;
import com.huaweicloud.sdk.iotda.v5.*;
import com.huaweicloud.sdk.iotda.v5.model.*;

public class ListDevicesSolution {

    // REGION_ID: If CN East-Shanghai1 is used, enter cn-east-3. If CN North-Beijing4 is used, enter cn-north-4. If CN South-Guangzhou is used, enter cn-south-1.
    private static final String REGION_ID = "<YOUR REGION ID>";
    //ENDPOINT: On the console, choose Overview and click Access Addresses to view the HTTPS application access address.
    private static final String ENDPOINT = "<YOUR ENDPOINT>";

    public static void main(String[] args) {
        // There will be security risks if the AK/SK used for authentication is directly written into code. Encrypt the AK/SK in the configuration file or environment variables for storage, and decrypt the AK/SK when using them;
        // In this example, the AK/SK stored in the environment variables are used. Configure the environment variables HUAWEICLOUD_SDK_AK and HUAWEICLOUD_SDK_SK in the local environment first.
        String ak = System.getenv("HUAWEICLOUD_SDK_AK");
        String sk = System.getenv("HUAWEICLOUD_SDK_SK");
        String projectId = "<YOUR PROJECTID>";

        // Create a credential.
        ICredential auth = new BasicCredentials()
                .withAk(ak)
                .withSk(sk)
                // WithDerivedPredicate is used for the standard or enterprise edition. For the basic edition, delete the line.
                .withDerivedPredicate(BasicCredentials.DEFAULT_DERIVED_PREDICATE)
                .withProjectId(projectId);

        // Create and initialize an IoTDAClient instance.
        IoTDAClient client = IoTDAClient.newBuilder()
                .withCredential(auth)
                // For the standard or enterprise edition, create a region object. For the basic edition, select the region object in IoTDARegion. For example, withRegion(IoTDARegion.CN_NORTH_4).
                .withRegion(new Region(REGION_ID, ENDPOINT))
                // .withRegion(IoTDARegion.CN_NORTH_4)
                // Whether to ignore SSL certificate verification (not ignored by default).
```



```
            // .withHttpConfig(new HttpConfig().withIgnoreSSLVerification(true))
            .build();

        // Instantiate a request object.
        ListDevicesRequest request = new ListDevicesRequest();
        try {
            // Call the API for querying the device list.
            ListDevicesResponse response = client.listDevices(request);
            System.out.println(response.toString());
        } catch (ConnectionException e) {
            e.printStackTrace();
        } catch (RequestTimeoutException e) {
            e.printStackTrace();
        } catch (ServiceResponseException e) {
            e.printStackTrace();
            System.out.println(e.getHttpStatusCode());
            System.out.println(e.getErrorCode());
            System.out.println(e.getErrorMsg());
        }
    }
}
```

----End

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

NOTE

For details on the project source code and usage guide, see **Huawei Cloud Java Software Development Kit (Java SDK)**.

2.2 Application Python SDK

IoTDA provides an application SDK in Python for developers. This topic describes how to install and configure the Python SDK and how to use it to call **application-side APIs**.

SDK Obtaining and Installing

Step 1 Install the Python development environment.

Visit the **Python website**, and download and install the Python development environment.

NOTE

The application Python SDK can be used in Python 3 and later versions.

Step 2 Install PiP.

Visit the **PiP website**, and download and install PiP.

Step 3 Install the Python SDK.

Run the following commands to install the Python SDK core library and related service libraries.

```
# Install the core library.
pip install huaweicloudsdkcore

# Install the IoTDA service library.
pip install huaweicloudsdkiotda
```

----End

Code Sample

The following code sample shows how to use the Python SDK to call API **Querying the Device List**.

Step 1 Create a credential.

Step 2 Create and initialize an IoTDAClient instance.

Step 3 Instantiate a request object.

Step 4 Call the API for querying the device list.

```
import os

from huaweicloudsdkcore.exceptions import exceptions
from huaweicloudsdkcore.region.region import Region
from huaweicloudsdkiotda.v5 import *
from huaweicloudsdkcore.auth.credentials import BasicCredentials
from huaweicloudsdkcore.auth.credentials import DerivedCredentials

if __name__ == "__main__":
    # There will be security risks if the AK/SK used for authentication is directly written into code. Encrypt the AK/SK in the configuration file or environment variables for storage, and decrypt the AK/SK when using them;
    # In this example, the AK/SK stored in the environment variables are used. Configure the environment variables HUAWEICLOUD_SDK_AK and HUAWEICLOUD_SDK_SK in the local environment first.
```



```
    ak = os.environ["HUAWEICLOUD_SDK_AK"]
    sk = os.environ["HUAWEICLOUD_SDK_SK"]
    project_id = "<YOUR PROJECTID>"
    # region_id: If CN East-Shanghai1 is used, enter cn-east-3. If CN North-Beijing4 is used, enter cn-north-4. If CN South-Guangzhou is used, enter cn-south-1.
    region_id = "<YOUR REGION ID>"
    # endpoint: On the console, choose Overview and click Access Addresses to view the HTTPS application access address.
    endpoint = "<YOUR ENDPOINT>"

    # For the standard or enterprise edition, create a region object.
    REGION = Region(region_id, endpoint)

    # Create a credential.
    # Create and initialize a BasicCredentials instance.
    credentials = BasicCredentials(ak, sk, project_id)

    # with_derived_predicate is used for the standard or enterprise edition. For the basic edition, delete the line.
    credentials.with_derived_predicate(DerivedCredentials.get_default_derived_predicate())

    # For the basic edition, select the region object in IoTDAClient. For example, .with_region(IoTDARegion.CN_NORTH_4).
    # For the standard or enterprise edition, create a region object.
    # Whether to ignore SSL certificate verification (not ignored by default): .with_http_config(HttpConfig(ignore_ssl_verification=True) \
    client = IoTDAClient.new_builder() \
        .with_credentials(credentials) \
        .with_region(REGION) \
        .build()

    try:
        # Instantiate a request object.
        request = ListDevicesRequest()
        # Call the API for querying the device list.
        response = client.list_devices(request)
        print(response)
    except exceptions.ClientRequestException as e:
        print(e.status_code)
        print(e.request_id)
        print(e.error_code)
        print(e.error_msg)
```

Parameter	Description
ak	Access key ID of your Huawei Cloud account. You can create and view an AK/SK on the **My Credentials** > **Access Keys** page of the Huawei Cloud console. For details, see **Access Keys**.
sk	Secret access key (SK) of your Huawei Cloud account.
project_id	Project ID. For details on how to obtain a project ID, see **Obtaining a Project ID**.
IoTDARegion.CN_NORTH_4	Region where the IoT platform to be accessed is located. The available regions of the IoT platform have been defined in the SDK code **iotda_region.py**. On the console, you can view the region name of the current service and the mapping between regions and endpoints. For details, see **Platform Connection Information**.

Parameter	Description
region_id	If CN East-Shanghai1 is used, enter **cn-east-3**. If CN North-Beijing4 is used, enter **cn-north-4**. If CN South-Guangzhou is used, enter **cn-south-4**.
endpoint	On the console, choose **Overview** and click **Access Addresses** to view the HTTPS application access address.

----End

Additional Information

For details on the project source code and usage guide, see **Huawei Cloud Python Software Development Kit (Python SDK)**.

2.3 Application .NET SDK

IoTDA provides an application SDK in C# for developers. This topic describes how to install and configure the .NET SDK and how to use it to call **application-side APIs**.

SDK Obtaining and Installing

Step 1 Install the .NET development environment.

Visit the **.NET website**, and download and install the .NET development environment.

NOTE

The .NET SDK can be used in the following environments:

- .NET Framework 4.5 and later
- .NET Standard 2.0 and later
- C# 4.0 and later

Step 2 Use the .NET CLI tool to install the SDK.

```
dotnet add package HuaweiCloud.SDK.Core
dotnet add package HuaweiCloud.SDK.IoTDA
```

----End

Code Sample

The following code sample shows how to use the .NET SDK to call API **Querying the Device List**.

Step 1 Create a credential.

Step 2 Create and initialize a BasicCredentials instance.

Step 3 Instantiate a request object.

Step 4 Call the API for querying the device list.

```
using System;
using System.Collections.Generic;
using HuaweiCloud.SDK.Core;
using HuaweiCloud.SDK.Core.Auth;
using HuaweiCloud.SDK.IoTDA;
using HuaweiCloud.SDK.IoTDA.V5;
using HuaweiCloud.SDK.IoTDA.V5.Model;

namespace ListDevicesSolution
{
    class Program
    {
        static void Main(string[] args)
        {
            var listDevicesRequest = ListDevices();
            var res = JsonUtils.Serialize(listDevicesRequest.Result);
            Console.WriteLine(res);
        }

       private static async Task<ListDevicesResponse> ListDevices()
        {
            // There will be security risks if the AK/SK used for authentication is directly written into code. Encrypt the AK/SK in the configuration file or environment variables for storage, and decrypt the AK/SK when using them;
            // In this example, the AK/SK stored in the environment variables are used. Configure the environment variables HUAWEICLOUD_SDK_AK and HUAWEICLOUD_SDK_SK in the local environment first.
            var ak = Environment.GetEnvironmentVariable("HUAWEICLOUD_SDK_AK", EnvironmentVariableTarget.Machine);
            var sk = Environment.GetEnvironmentVariable("HUAWEICLOUD_SDK_SK", EnvironmentVariableTarget.Machine);
            const string projectId = "<YOUR PROJECTID>";
           // region_id: If CN East-Shanghai1 is used, enter cn-east-3. If CN North-Beijing4 is used, enter cn-north-4. If CN South-Guangzhou is used, enter cn-south-1.
            const string regionId = "<YOUR REGION ID>";
            // endpoint: On the console, choose Overview and click Access Addresses to view the HTTPS application access address.
            const string endpoint = "<YOUR ENDPOINT>";

            // Create a credential.
            var auth = new BasicCredentials(ak, sk, projectId);

           // Used for the standard or enterprise edition. For the basic edition, delete the line.
            auth.WithDerivedPredicate(BasicCredentials.DefaultDerivedPredicate);

            var config = HttpConfig.GetDefaultConfig();
            config.IgnoreSslVerification = false;
            // Create and initialize an IoTDAClient instance.
            var client = IoTDAAsyncClient.NewBuilder()
                  .WithCredential(auth)
                  .WithHttpConfig(config)
                 // For the standard or enterprise edition, create a region object.
                  .WithRegion(new Region(regionId, endpoint))
                 // For the basic edition, select the region object in IoTDARegion.
                 // .WithRegion(IoTDARegion.CN_NORTH_4)
                 // .NET Framework does not support content-type in the get request header.
                // Whether to ignore SSL certificate verification (not ignored by default).
                  // .WithHttpConfig(new HttpConfig().WithIgnoreBodyForGetRequest(true).WithIgnoreSslVerification(true))
                  .Build();

            // Instantiate a request object.
            var req = new ListDevicesRequest
            {
            };

            try
            {
```



```
            // Call the API for querying the device list.
            var resp =await client.ListDevicesAsync(req);
            Console.WriteLine(resp.GetHttpStatusCode());
            return resp;
        }
        catch (RequestTimeoutException requestTimeoutException)
        {
            Console.WriteLine(requestTimeoutException.ErrorMessage);
        }
        catch (ServiceResponseException clientRequestException)
        {
            Console.WriteLine(clientRequestException.HttpStatusCode);
            Console.WriteLine(clientRequestException.ErrorCode);
            Console.WriteLine(clientRequestException.ErrorMsg);
        }
        catch (ConnectionException connectionException)
        {
            Console.WriteLine(connectionException.ErrorMessage);
        }
        return new ListDevicesResponse();
    }
  }
}
```

Parameter	Description
ak	Access key ID of your Huawei Cloud account. You can create and view an AK/SK on the **My Credentials** > **Access Keys** page of the Huawei Cloud console. For details, see **Access Keys**.
sk	Secret access key (SK) of your Huawei Cloud account.
IoTDARegion.CN_NORTH_4	Region where the IoT platform to be accessed is located. The available regions of the IoT platform have been defined in the SDK code **IoTDARegion.cs**. On the console, you can view the region name of the current service and the mapping between regions and endpoints. For details, see **Platform Connection Information**.

----End

Additional Information

For details on the project source code and usage guide, see **Huawei Cloud .NET Software Development Kit (.NET SDK)**.

2.4 Application Go SDK

IoTDA provides an application SDK in Go for developers. This topic describes how to install and configure the Go SDK and how to use it to call **application-side APIs**.

SDK Obtaining and Installing

Step 1 Install the Go development environment.

Visit the **Go website**, and download and install the Go development environment.

> **NOTE**
>
> The Go SDK can be used in Go 1.14 and later versions.

Step 2 Install the Huawei Cloud Go library.

```
go get github.com/huaweicloud/huaweicloud-sdk-go-v3
```

Step 3 Install dependencies.

```
go get github.com/json-iterator/go
```

----End

Code Sample

The following code sample shows how to use the Go SDK to call API **Querying the Device List**.

Step 1 Create a credential.

Step 2 Create and initialize an IoTDAClient instance.

Step 3 Instantiate a request object.

Step 4 Call the API for querying the device list.

```
package main

import (
    "fmt"
    "github.com/huaweicloud/huaweicloud-sdk-go-v3/core/auth"
    "github.com/huaweicloud/huaweicloud-sdk-go-v3/core/auth/basic"
    // For the standard or enterprise edition, use github.com/huaweicloud/huaweicloud-sdk-go-v3/core/region.
    // For the basic edition, use github.com/huaweicloud/huaweicloud-sdk-go-v3/services/iotda/v5/region.
    "github.com/huaweicloud/huaweicloud-sdk-go-v3/core/region"
    //"github.com/huaweicloud/huaweicloud-sdk-go-v3/services/iotda/v5/region"
    iotda "github.com/huaweicloud/huaweicloud-sdk-go-v3/services/iotda/v5"
    "github.com/huaweicloud/huaweicloud-sdk-go-v3/services/iotda/v5/model"
)

func main() {
        // There will be security risks if the AK/SK used for authentication is directly written into code. Encrypt the AK/SK in the configuration file or environment variables for storage, and decrypt the AK/SK when using them;
        // In this example, the AK/SK stored in the environment variables are used. Configure the environment variables HUAWEICLOUD_SDK_AK and HUAWEICLOUD_SDK_SK in the local environment first.
    ak := os.Getenv("HUAWEICLOUD_SDK_AK")
    sk := os.Getenv("HUAWEICLOUD_SDK_SK")
    projectId := "<YOUR PROJECTID>"

    // The regionId and endpoint are used to create regions for the standard edition and enterprise edition. For the basic edition, delete these two lines.
    // region_id: If CN East-Shanghai1 is used, enter cn-east-3. If CN North-Beijing4 is used, enter cn-north-4. If CN South-Guangzhou is used, enter cn-south-1.
    regionId := "<YOUR REGION ID>"
    // endpoint: On the console, choose Overview and click Access Addresses to view the HTTPS application access address.
    endpoint := "<YOUR ENDPOINT>"

    // Create a credential.
    auth := basic.NewCredentialsBuilder().
        WithAk(ak).
        WithSk(sk).
        WithProjectId(projectId).
        // WithDerivedPredicate is used for the standard or enterprise edition. For the basic edition, delete
```



```
the line.
        WithDerivedPredicate(auth.GetDefaultDerivedPredicate()).
        Build()

    // Create and initialize an IoTDAClient instance.
    client := iotda.NewIoTDAClient(
        iotda.IoTDAClientBuilder().
            // For the standard or enterprise edition, create a region object. For the basic edition, select the
region object in IoTDARegion.
            WithRegion(region.NewRegion(regionId, endpoint)).
            // WithRegion(region.CN_NORTH_4).
            WithCredential(auth).
                    // Whether to ignore SSL certificate verification (not ignored by default).
                    // WithHttpConfig(config.DefaultHttpConfig().WithIgnoreSSLVerification(true)).
            Build())
    // Instantiate a request object.
    request := &model.ListDevicesRequest{}
    // Call the API for querying the device list.
    response, err := client.ListDevices(request)
    if err == nil {
        fmt.Printf("%+v\n", response)
    } else {
        fmt.Println(err)
    }
}
```

Parameter	Description
ak	Access key ID of your Huawei Cloud account. You can create and view an AK/SK on the **My Credentials** > **Access Keys** page of the Huawei Cloud console. For details, see **Access Keys**.
sk	Secret access key (SK) of your Huawei Cloud account.
IoTDARegion.CN_NORTH_4	Region where the IoT platform to be accessed is located. The available regions of the IoT platform have been defined in the SDK code **region.go**. On the console, you can view the region name of the current service and the mapping between regions and endpoints. For details, see **Platform Connection Information**.

----End

Additional Information

For details on the project source code and usage guide, see **Huawei Cloud Go Software Development Kit (Go SDK)**.

2.5 Application Node.js SDK

IoTDA provides an application SDK in Node.js for developers. This topic describes how to install and configure the Node.js SDK and how to use it to call **application-side APIs**.

SDK Obtaining and Installing

Step 1 Install the Node.js development environment.

Visit the **Node.js website**, and download and install the Node.js development environment.

NOTE

The Node.js SDK can be used in Node 10.16.1 and later versions.

Step 2 Install dependencies.

```
npm install @huaweicloud/huaweicloud-sdk-core
npm install @huaweicloud/huaweicloud-sdk-iotda
```

----End

Code Sample

The following code sample shows how to use the Node.js SDK to call API **Querying the Device List**.

Step 1 Create a credential.

Step 2 Create and initialize an IoTDAClient instance.

Step 3 Instantiate a request object.

Step 4 Call the API for querying the device list.

```
const core = require('@huaweicloud/huaweicloud-sdk-core');
const iotda = require("@huaweicloud/huaweicloud-sdk-iotda");
// There will be security risks if the AK/SK used for authentication is directly written into code. Encrypt the AK/SK in the configuration file or environment variables for storage, and decrypt the AK/SK when using them;
// In this example, the AK/SK stored in the environment variables are used. Configure the environment variables HUAWEICLOUD_SDK_AK and HUAWEICLOUD_SDK_SK in the local environment first.
const ak = process.env.HUAWEICLOUD_SDK_AK;
const sk = process.env.HUAWEICLOUD_SDK_SK;
// endpoint: On the console, choose Overview and click Access Addresses to view the HTTPS application access address.
// const endpoint = "https://iotda.cn-north-4.myhuaweicloud.com";
const endpoint = "<YOUR ENDPOINT>";
const project_id = "<YOUR PROJECT_ID>";
// region_id: If CN East-Shanghai1 is used, enter cn-east-3. If CN North-Beijing4 is used, enter cn-north-4. If CN South-Guangzhou is used, enter cn-south-1.
const region_id = "<YOUR REGION_ID>";
// (Optional) Skip server certificate verification.
process.env.NODE_TLS_REJECT_UNAUTHORIZED = "0"
// Create a credential.
const credentials = new core.BasicCredentials()
                    .withAk(ak)
                    .withSk(sk)
                    // WithDerivedPredicate is used for the standard or enterprise edition. For the basic edition, delete the line.
                    .withDerivedPredicate(core.BasicCredentials.getDefaultDerivedPredicate)
                    .withProjectId(project_id)
// Create and initialize an IoTDAClient instance.
const client = iotda.IoTDAClient.newBuilder()
                        .withCredential(credentials)
                        .withEndpoint(endpoint)
                        .withRegion(new core.Region(region_id, endpoint))
                        .build();
// Instantiate a request object.
const request = new iotda.ListDevicesRequest();
// Call the API for querying the device list.
```



```
const result = client.listDevices(request);
result.then(result => {
    console.log("JSON.stringify(result)::" + JSON.stringify(result));
}).catch(ex => {
    console.log("exception:" + JSON.stringify(ex));
});
```

Parameter	Description
ak	Access key ID of your Huawei Cloud account. You can create and view an AK/SK on the **My Credentials** > **Access Keys** page of the Huawei Cloud console. For details, see **Access Keys**.
sk	Secret access key (SK) of your Huawei Cloud account.
endpoint	Endpoint of the region where the Huawei Cloud service to be accessed is located. On the console, you can view the region name of the current service and the mapping between regions and endpoints. For details, see **Platform Connection Information**.
project_id	ID of the project where the Huawei Cloud service to be accessed is located. Select a project ID based on the region to which the project belongs.
region_id	If CN East-Shanghai1 is used, enter **cn-east-3**. If CN North-Beijing4 is used, enter **cn-north-4**. If CN South-Guangzhou is used, enter **cn-south-4**.

----End

Additional Information

For details on the project source code and usage guide, see **Huawei Cloud Node.js Software Development Kit (Node.js SDK)**.

2.6 Application PHP SDK

IoTDA provides an application SDK in PHP for developers. This topic describes how to install and configure the PHP SDK and how to use it to call **application-side APIs**.

SDK Obtaining and Installing

Step 1 Install the PHP development environment.

Visit the **PHP website**, and download and install the PHP development environment.

NOTE

Huawei Cloud PHP SDKs support PHP 5.6, PHP 6, and PHP 7, but do not support PHP 8. Before running PHP SDKs, run the **php --version** command to check the current PHP version. If PHP of other languages is installed, an error may occur when you run PHP SDKs.

Step 2 Install Composer.

```
curl -sS https://getcomposer.org/installer | php
```

Step 3 Install the PHP SDK.

```
composer require huaweicloud/huaweicloud-sdk-php
```

Step 4 Introduce the **autoload.php** file of Composer.

```
require 'path/to/vendor/autoload.php';
```

----End

Code Sample

The following code sample shows how to use the PHP SDK to call API **Querying the Device List**.

Step 1 Create a credential.

Step 2 Create and initialize an IoTDAClient instance.

Step 3 Instantiate a request object.

Step 4 Call the API for querying the device list.

```
<?php
namespace HuaweiCloud\SDK\IoTDA\V5\Model;
require_once "vendor/autoload.php";
use HuaweiCloud\SDK\Core\Auth\BasicCredentials;
use HuaweiCloud\SDK\Core\Http\HttpConfig;
use HuaweiCloud\SDK\Core\Auth\Credentials;
use HuaweiCloud\SDK\Core\Exceptions\ConnectionException;
use HuaweiCloud\SDK\Core\Exceptions\RequestTimeoutException;
use HuaweiCloud\SDK\Core\Exceptions\ServiceResponseException;
use HuaweiCloud\SDK\Core\Region\Region;
use HuaweiCloud\SDK\IoTDA\V5\IoTDAClient;
// There will be security risks if the AK/SK used for authentication is directly written into code. Encrypt the AK/SK in the configuration file or environment variables for storage, and decrypt the AK/SK when using them;
// In this example, the AK/SK stored in the environment variables are used. Configure the environment variables HUAWEICLOUD_SDK_AK and HUAWEICLOUD_SDK_SK in the local environment first.
$ak = getenv('HUAWEICLOUD_SDK_AK');
$sk = getenv('HUAWEICLOUD_SDK_SK');
// endpoint: On the console, choose Overview and click Access Addresses to view the HTTPS application access address.
// $endpoint = "https://iotda.cn-north-4.myhuaweicloud.com";
$endpoint = "<YOUR ENDPOINT>";
$projectId = "<YOUR PROJECT_ID>";
// REGION_ID: If CN East-Shanghai1 is used, enter cn-east-3. If CN North-Beijing4 is used, enter cn-north-4. If CN South-Guangzhou is used, enter cn-south-1.
$regionId = "<YOUR REGION ID>";
// Create a credential.
$credential = new BasicCredentials($ak,$sk,$projectId);
// withDerivedPredicate is used for the standard or enterprise edition. For the basic edition, delete the line.
$credential->withDerivedPredicate(Credentials::getDefaultDerivedPredicate());
// Modify the default configuration and skip the server certificate verification.
$config = HttpConfig::getDefaultConfig();
$config->setIgnoreSslVerification(true);
// Create an IoTDAClient instance and initialize it. (If the default configuration is not modified, you do not need to add config.)
$client = IoTDAClient::newBuilder(new IoTDAClient)
  ->withHttpConfig($config)
  ->withEndpoint($endpoint)
  ->withCredentials($credential)
  ->withRegion(new Region($regionId, $endpoint))
  ->build();
// Instantiate a request object.
$request = new ListDevicesRequest();
```



```
try {
// Call the API for querying the device list.
  $response = $client->ListDevices($request);
} catch (ConnectionException $e) {
  $msg = $e->getMessage();
  echo "\n". $msg ."\n";
} catch (RequestTimeoutException $e) {
  $msg = $e->getMessage();
  echo "\n". $msg ."\n";
} catch (ServiceResponseException $e) {
  echo "\n";
  echo $e->getHttpStatusCode(). "\n";
  echo $e->getErrorCode() . "\n";
  echo $e->getErrorMsg() . "\n";
}
echo "\n";
echo $response;
```

Parameter	Description
ak	Access key ID of your Huawei Cloud account. You can create and view an AK/SK on the **My Credentials** > **Access Keys** page of the Huawei Cloud console. For details, see **Access Keys**.
sk	Secret access key (SK) of your Huawei Cloud account.
endpoint	Endpoint of the region where the Huawei Cloud service to be accessed is located. On the console, you can view the region name of the current service and the mapping between regions and endpoints. For details, see **Platform Connection Information**.
projectId	ID of the project where the Huawei Cloud service to be accessed is located. Select a project ID based on the region to which the project belongs.

----End

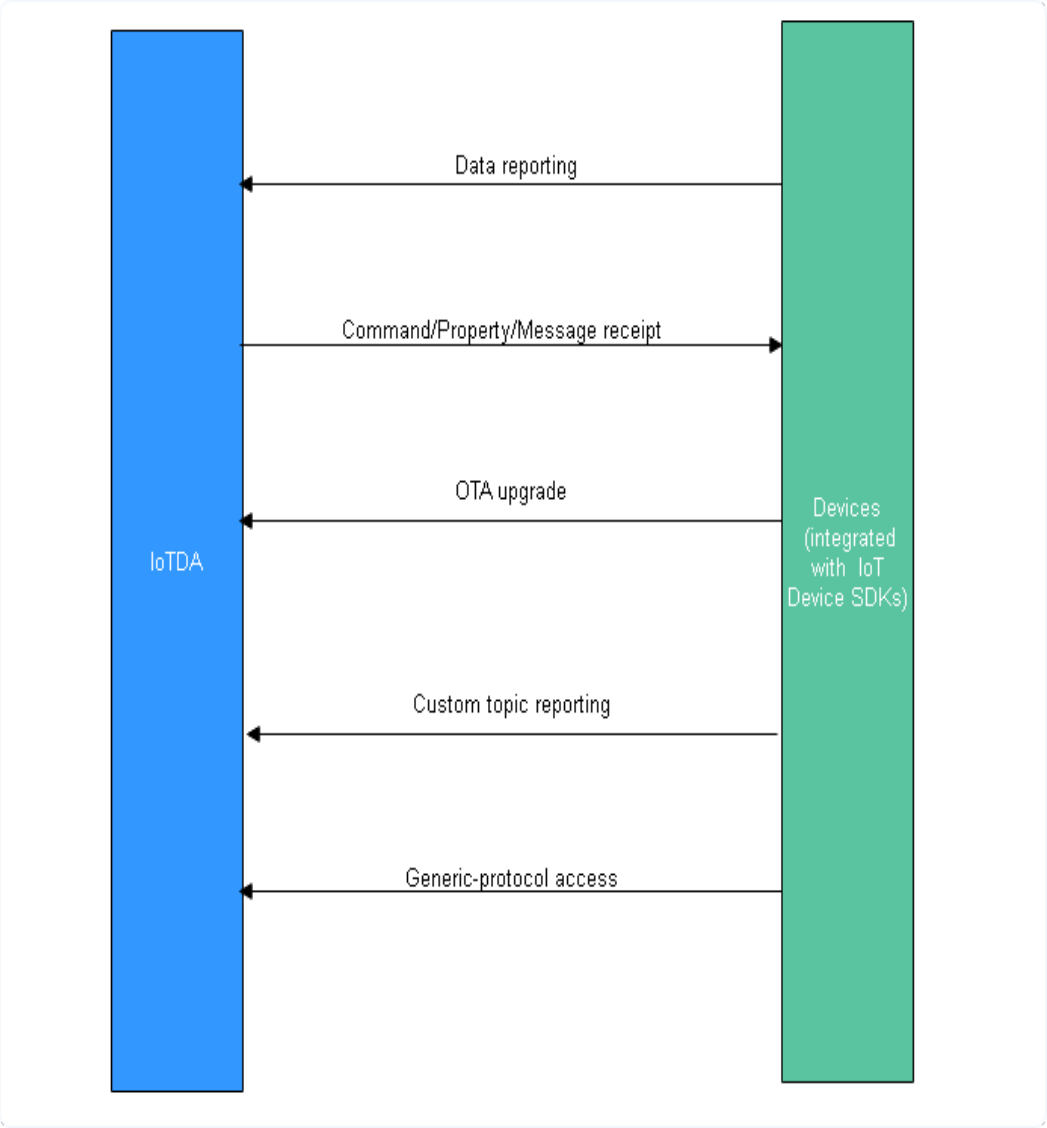
Additional Information

For details on the project source code and usage guide, see **Huawei Cloud PHP Software Development Kit (PHP SDK)**.

3 Device SDKs

3.1 Overview

You can use Huawei IoT Device SDKs to quickly connect devices to the IoT platform. After being integrated with an IoT Device SDK, devices that support the TCP/IP protocol stack can directly communicate with the platform. Devices that do not support the TCP/IP protocol stack, such as Bluetooth and ZigBee devices, need to use a gateway integrated with the IoT Device SDK to communicate with the platform.



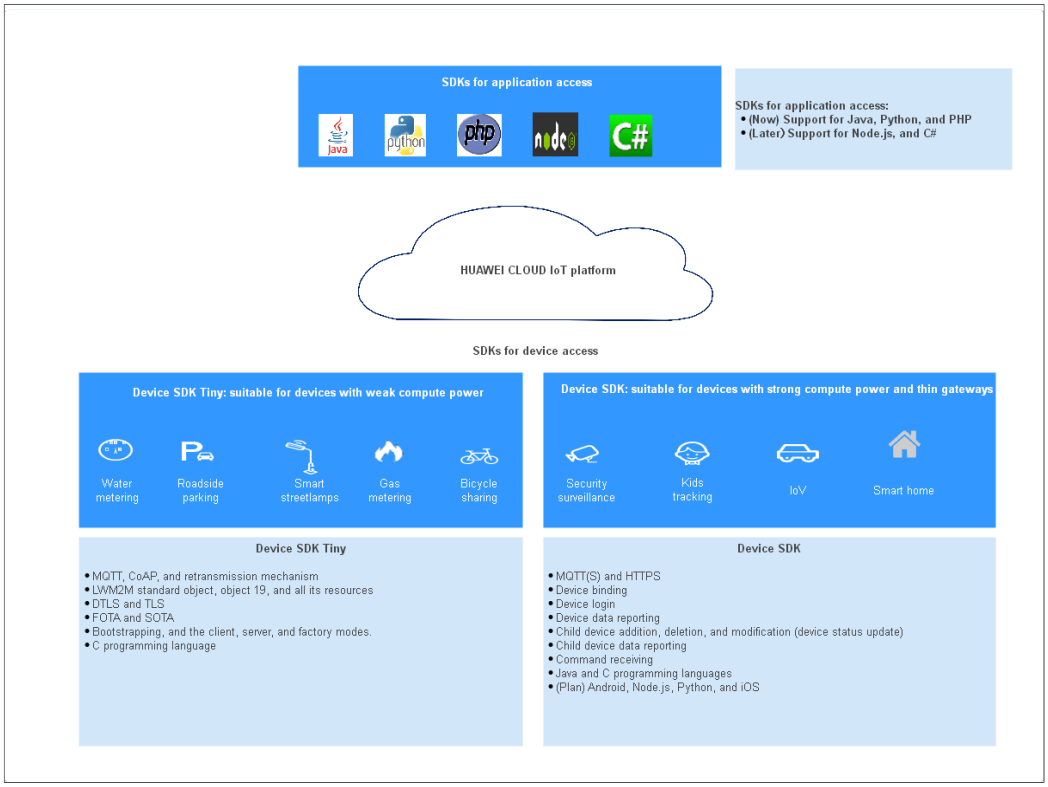
1. Create a product on the IoTDA console or by calling the API for **creating a product**.
2. Register a device on the IoTDA console or by calling the API **Creating a Device**.
3. Implement the functions demonstrated in the preceding figure, including reporting messages/properties, receiving commands/properties/messages, OTA upgrades, topic customization, and generic-protocol access (see **Developing a Protocol Conversion Gateway for Access of Generic-Protocol Devices**).

The platform provides two types of SDKs. The table below describes their differences.

SDK Type	Pre-integration Solution	IoT Protocols Supported
IoT Device SDK	Embedded devices with strong computing and storage capabilities, such as gateways and collectors	MQTT
IoT Device SDK Tiny	Devices that have strict restrictions on power consumption, storage, and computing resources, such as single-chip microcomputer and modules	LwM2M over CoAP and MQTT

The table below describes hardware requirements for devices.

SDK	RAM Capacity	Flash Memory	CPU Frequency	OS Type	Programming Language
IoT Device SDK	> 4 MB	> 2 MB	> 200 MHz	C (Linux), Java (Linux/ Windows), C# (Windows), Android (Android), Go Community Edition (Linux/ Windows/Unix-like OS), and OpenHarmony	C, Java, C#, Android, and Go
IoT Device SDK Tiny	> 32 KB	> 128 KB	> 100 MHz	It adapts to LiteOS, Linux, macOS, and FreeRTOS. You can modify the SDK to **adapt to other environments**.	C



For details on the SDK usage, visit the following links:

- **IoT Device C SDK for Linux/Windows**
- **IoT Device Java SDK**
- **IoT Device C# SDK**
- **IoT Device Android SDK**
- **IoT Device Go SDK**
- **IoT Device Tiny C SDK for Linux/Windows**
- **IoT Device Python SDK**

The following table shows the main function matrix of the SDK.

Table 3-1 SDK function matrix

Function	C	Java	C#	Android	Go	Python	C Tiny	ArkTS
Property reporting	√	√	√	√	√	√	√	√
Message reporting and delivery	√	√	√	√	√	√	√	√

Function	C	Java	C#	Android	Go	Python	C Tiny	ArkTS
Event reporting and delivery	√	√	√	√	√	√	√	×
Command delivery and response	√	√	√	√	√	√	√	√
Device shadow	√	√	√	√	√	√	√	√
OTA upgrade	√	√	√	√	√	√	√	×
bootstrap	√	√	√	√	√	√	√	×
Time synchronization	√	√	√	√	√	√	√	×
Gateway and child device management	√	√	√	√	√	√	√	×
Device-side Rules	√	×	√	×	×	×	√	×
Remote SSH	√	×	√	×	×	×	×	×
Anomaly detection	√	×	√	×	×	×	×	×

Function	C	Java	C#	Android	Go	Python	C Tiny	ArkTS
Device-cloud secure communication (soft bus)	√	×	√	×	×	×	×	×
M2M function	√	×	√	×	×	×	×	×
Generic-protocol access	√	√	√	√	×	√	×	×

FAQ

IoT Device SDKs

Device Integration

3.2 IoT Device Java SDK

Maven Reference

```
<dependencies>
    <dependency>
        <groupId>com.huaweicloud</groupId>
        <artifactId>iot-device-sdk-java</artifactId>
        <version>1.2.0</version>
    </dependency>
</dependencies>
```

Requirements

- JDK (version 1.8 or later) and Maven have been installed.
- **Download the SDK**. The project contains the following subprojects:

  iot-bridge-demo
  iot-bridge-sample-tcp-protocol
  iot-bridge-sdk
  iot-device-code-generator
  iot-device-demo
  iot-device-sdk-java
  iot-gateway-demo

iot-device-sdk-java: SDK code

iot-device-demo: demo code for common directly connected devices

iot-gateway-demo: demo code for gateways

iot-bridge-sdk: SDK code for the bridge

iot-bridge-demo: demo code for the bridge, which is used to bridge a TCP device to the platform

iot-bridge-sample-tcp-protocol: sample code of a child device using TCP to connect to a bridge

iot-device-code-generator: device code generator, which can automatically generate device code for different product models

- Go to the SDK root directory and run the **mvn install** command to build and install the SDK.

Creating a Product

A smoke detector product model is provided to help you understand the product model. This smoke detector can report the smoke density, temperature, humidity, and smoke alarms, and execute the ring alarm command. The following uses the smoke detector as an example to introduce the procedures of message reporting and property reporting.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card. Check and save the MQTTS device access domain name.

Step 2 Choose **Products** in the navigation pane and click **Create Product**.

Step 3 Set the parameters as prompted and click **OK**.

Set Basic Info	
Resource Space	The platform automatically allocates the created product to the default resource space. If you want to allocate the product to another resource space, select the resource space from the drop-down list box. If a **resource space** does not exist, create it first.
Product Name	Customize the product name. The name can contain letters, numbers, underscores (_), and hyphens (-).
Protocol	Select **MQTT**.
Data Type	Select **JSON**.
Device Type Selection	Select **Custom**.
Device Type	Select **smokeDetector**.
Advanced Settings	
Product ID	Leave this parameter blank.
Description	Set this parameter based on service requirements.

----End

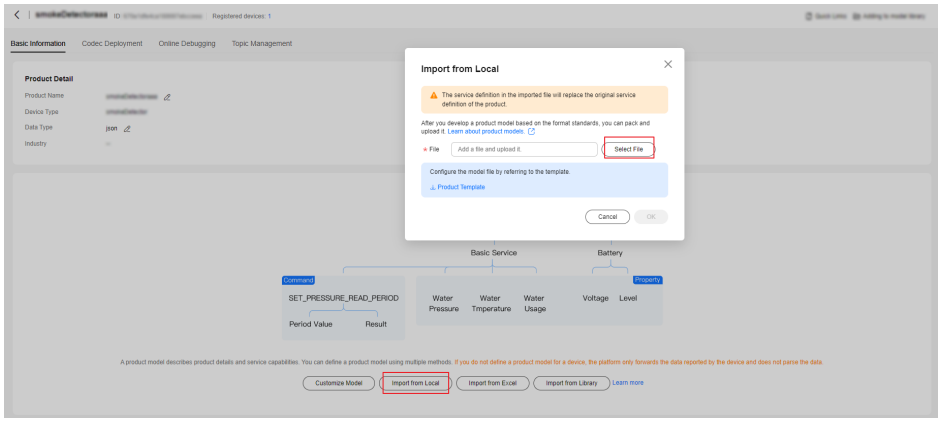
Uploading a Product Model

Step 1 Download the **smokeDetector product model file**.

Step 2 Click the name of the product created in **3** to access its details.

Step 3 On the **Basic Information** tab page, click **Import from Local** to upload the product model file obtained in **1**.

Figure 3-1 Product - Uploading a product model



----End

Registering a Device

Step 1 In the navigation pane, choose **Devices** > **All Devices**, and click **Register Device**.

Step 2 Set the parameters as prompted and click **OK**.

Parameter	Description
Resource Space	Ensure that the device and the product created in **3** belong to the same resource space.
Product	Select the product created in **3**.
Node ID	This parameter specifies the unique physical identifier of the device. The value can be customized and consists of letters and numbers.
Device Name	Customize the device name.
Authentication Type	Select **Secret**.
Secret	Customize the device secret. If this parameter is not set, the platform automatically generates a secret.

After the device is registered, save the node ID, device ID, and secret.

----End

Initializing a Device

1. Enter the device ID and secret obtained in **Registering a Device** and the device connection information obtained in **1**. The format is *ssl://Domain name:Port* or *ssl://IP address:Port*.

```
// Obtaining the certificate path: Load the CA certificate of the platform and use the default ca.jks of the SDK for server verification.
    URL resource = MessageSample.class.getClassLoader().getResource("ca.jks");
    File file = new File(resource.getPath());
    //For example, modify the following parameters in MessageSample.java in the iot-device-demo file:
    IoTDevice device = new IoTDevice("ssl://Domain name:8883",
            "5e06bfee334dd4f33759f5b3_demo", "mysecret", file);
```

   NOTE

   All files that involve device IDs and passwords must be modified accordingly.

2. Establish a connection. Call **init** of the IoT Device SDK. The thread is blocked until a result is returned. If the connection is established, **0** is returned.

```
    if (device.init() != 0) {
        return;
    }
```

   If the connection is successful, information similar to the following is displayed:

```
2023-07-17 17:22:59 INFO MqttConnection:105 - Mqtt client connected. address :ssl://Domain name:8883
```

3. After the device is created and connected, it can be used for communication. Call the **getClient** method of the IoT Device SDK to obtain the device client. The client provides communication APIs for processing messages, properties, and commands.

Reporting a Message

Message reporting is the process in which a device reports messages to the platform.

1. Call **getClient** of the IoT Device SDK to obtain the client from the device.
2. Call **reportDeviceMessage** to enable the client to report a device message. In the sample below, messages are reported periodically.

```
    while (true) {
        device.getClient().reportDeviceMessage(new DeviceMessage("hello"), new ActionListener() {
            @Override
            public void onSuccess(Object context) {
                log.info("reportDeviceMessage ok");
            }

            @Override
            public void onFailure(Object context, Throwable var2) {
                log.error("reportDeviceMessage fail: " + var2);
            }
        });

        // Report messages through custom topics, which must be configured on the platform first.
        String topic = "$oc/devices/" + device.getDeviceId() + "/user/wpy";
        device.getClient().publishRawMessage(new RawMessage(topic, "hello raw message "),
            new ActionListener() {
                @Override
```



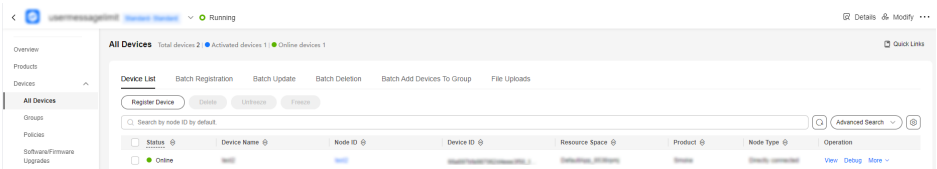
```
            public void onSuccess(Object context) {
                log.info("publishRawMessage ok: ");
            }

            @Override
            public void onFailure(Object context, Throwable var2) {
                log.error("publishRawMessage fail: " + var2);
            }
        });

        Thread.sleep(5000);
    }
```

3. Replace the device parameters with the actual values in the **main** function of the **MessageSample** class, and run this class. Then view the logs about successful connection and message reporting.

```
2024-04-16 16:43:09  INFO AbstractService:103 - create device, the deviceId is
5e06bfee334dd4f33759f5b3_demo
2024-04-16 16:43:09 INFO MqttConnection:233 - try to connect to ssl://Domain name: 8883
2024-04-16 16:43:10 INFO MqttConnection:257 - connect success, the uri is ssl://Domain name: 8883
2024-04-16 16:43:11  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"5e06bfee334dd4f33759f5b3_demo","services":[{"paras":
{"type":"DEVICE_STATUS","content":"connect
success","timestamp":"1713256990817"},"service_id":"$log","event_type":"log_report","event_time":"20
240416T084310Z","event_id":null}]}
2024-04-16 16:43:11 INFO MqttConnection:140 - Mqtt client connected. address is ssl://Domain
name: 8883
2024-04-16 16:43:11  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"5e06bfee334dd4f33759f5b3_demo","services":[{"paras":
{"device_sdk_version":"JAVA_v1.2.0","fw_version":null,"sw_version":null},"service_id":"$sdk_info","event
_type":"sdk_info_report","event_time":"20240416T084311Z","event_id":null}]}
2024-04-16 16:43:11 INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo /sys/events/up, msg = {"object_device_id":
"5e06bfee334dd4f33759f5b3_demo ","services": [{"paras":
{"type":"DEVICE_STATUS","content":"connect complete, the url is ssl://Domain name:
8883","timestamp":"1713256991263"},"service_id":"$log","event_type":"log_report","event_time":"2024
0416T084311Z","event_id":null}]}
2024-04-16 16:43:11  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/messages/up, msg =
{"name":null,"id":null,"content":"hello","object_device_id":null}
2024-04-16 16:43:11  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/user/wpy, msg =  hello raw message
2024-04-16 16:43:11  INFO MessageSample:98 - reportDeviceMessage ok
2024-04-16 16:43:11  INFO MessageSample:113 - publishRawMessage ok:
```

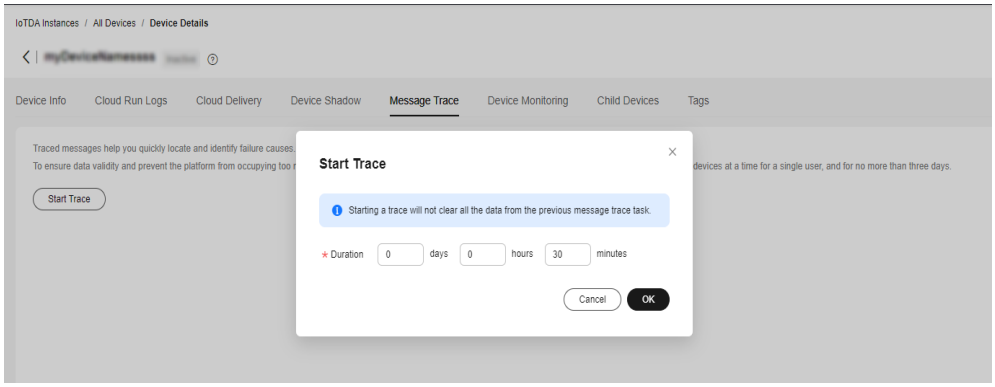
4. On the IoTDA console, choose **Devices** > **All Devices** and check whether the device is online.

Figure 3-2 Device list - Device online status



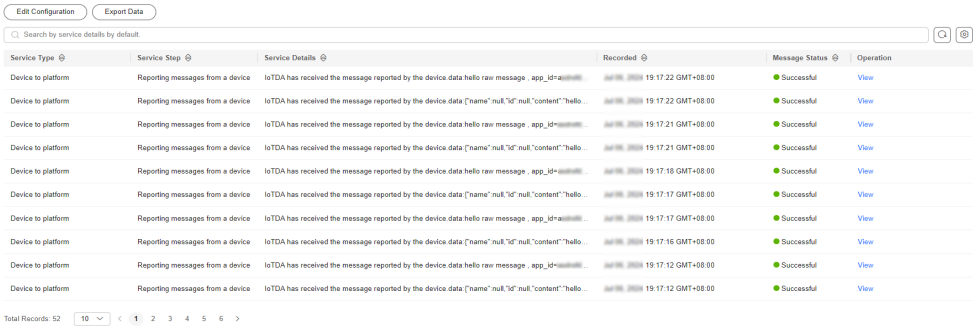
5. Select the device, click **View**, and enable message trace on the device details page.

Figure 3-3 Message tracing - Starting message tracing



6. View the messages received by the platform.

Figure 3-4 Message tracing - Viewing device_sdk_java tracing result



Note: Message trace may be delayed. If no data is displayed, wait for a while and refresh the page.

Reporting Properties

Open the **PropertySample** class. In this example, the **alarm**, **temperature**, **humidity**, and **smokeConcentration** properties are periodically reported to the platform.

```
    // Report properties periodically.
      while (true) {

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

          // Set properties based on the product model.
          json.put("alarm", 1);
          json.put("temperature", rand.nextFloat()*100.0f);
          json.put("humidity", rand.nextFloat()*100.0f);
          json.put("smokeConcentration", rand.nextFloat() * 100.0f);

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


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



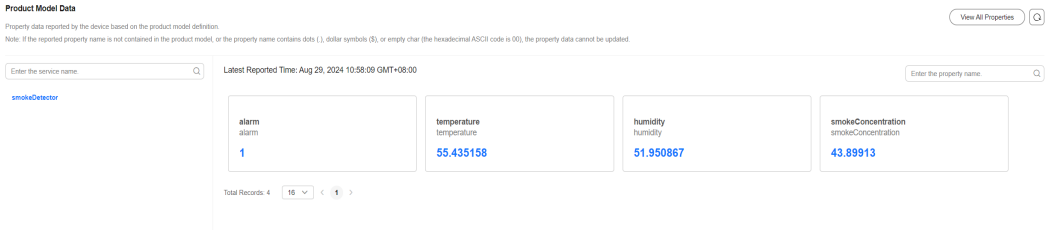
```
            @Override
            public void onFailure(Object context, Throwable var2) {
                log.error("reportProperties failed" + var2.toString());
            }
        });

        Thread.sleep(10000);
    }
}
```

Modify the **main** function of the **PropertySample** class and run this class. Then view the logs about successful property reporting.

```
2024-04-17 15:38:37  INFO AbstractService:103 - create device, the deviceId is
5e06bfee334dd4f33759f5b3_demo
2024-04-17 15:38:37 INFO MqttConnection:233 - try to connect to ssl://Domain name: 8883
2024-04-17 15:38:38 INFO MqttConnection:257 - connect success, the uri is ssl://Domain name: 8883
2024-04-17 15:38:38  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"661e35467bdccc0126d1a595_feng-sdk-test3","services":[{"paras":
{"type":"DEVICE_STATUS","content":"connect
success","timestamp":"1713339518043"},"service_id":"$log","event_type":"log_report","event_time":"2024041
7T073838Z","event_id":null}]}
2024-04-17 15:38:38 INFO MqttConnection:140 - Mqtt client connected. address is ssl://Domain name: 8883
2024-04-17 15:38:38  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"661e35467bdccc0126d1a595_feng-sdk-test3","services":[{"paras":
{"device_sdk_version":"JAVA_v1.2.0","fw_version":null,"sw_version":null},"service_id":"$sdk_info","event_type"
:"sdk_info_report","event_time":"20240417T073838Z","event_id":null}]}
2024-04-17 15:38:38 INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo /sys/events/up, msg = {"object_device_id":
"5e06bfee334dd4f33759f5b3_demo ","services": [{"paras":{"type":"DEVICE_STATUS","content":"connect
complete, the url is ssl://Domain
name :8883","timestamp":"1713339518464"},"service_id":"$log","event_type":"log_report","event_time":"202
40417T073838Z","event_id":null}]}
2024-04-17 15:38:38  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/properties/report, msg =  {"services":[{"properties":
{"alarm":1,"temperature":55.435158,"humidity":51.950867,"smokeConcentration":43.89913},"service_id":"sm
okeDetector","event_time":null}]}
2024-04-17 15:38:38  INFO PropertySample:144 - pubMessage success
```

The latest property values are displayed on the device details page of the platform.

Figure 3-5 Product model - Property reporting



Reading and Writing Properties

Call the **setPropertyListener** method of the client to set the property callback. In **PropertySample**, the property reading/writing API is implemented.

Property reading: Only the **alarm** property can be written.

Property reading: Assemble the local property value based on the API format.

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



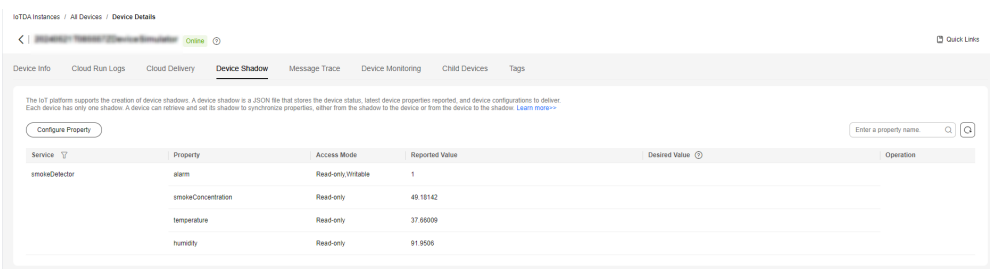
```
        // Process property writing.
        @Override
        public void onPropertiesSet(String requestId, List<ServiceProperty> services) {
            // Traverse services.
            for (ServiceProperty serviceProperty : services) {

                log.info("OnPropertiesSet, serviceId is {}", serviceProperty.getServiceId());

                // Traverse properties.
                for (String name : serviceProperty.getProperties().keySet()) {
                    log.info("property name is {}", name);
                    log.info("set property value is {}", serviceProperty.getProperties().get(name));
                }

            }
            // Change the local property value.
            device.getClient().respondPropsSet(requestId, IotResult.SUCCESS);
        }

        /**
         * Process property reading. In most scenarios, you can directly read the device shadow on the
platform, so this interface does not need to be implemented.
         * To read device properties in real time, implement this method.
         */
        @Override
        public void onPropertiesGet(String requestId, String serviceId) {
            log.info("OnPropertiesGet, the serviceId is {}", serviceId);
            Map<String, Object> json = new HashMap<>();
            Random rand = new SecureRandom();
            json.put("alarm", 1);
            json.put("temperature", rand.nextFloat() * 100.0f);
            json.put("humidity", rand.nextFloat() * 100.0f);
            json.put("smokeConcentration", rand.nextFloat() * 100.0f);

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

            device.getClient().respondPropsGet(requestId, Arrays.asList(serviceProperty));
        }
    });
```

NOTE

1. The property reading/writing API must call the **respondPropsGet** and **respondPropsSet** methods to report the operation result.
2. If the device does not allow the platform to proactively read data from the device, **onPropertiesGet** can be left not implemented.

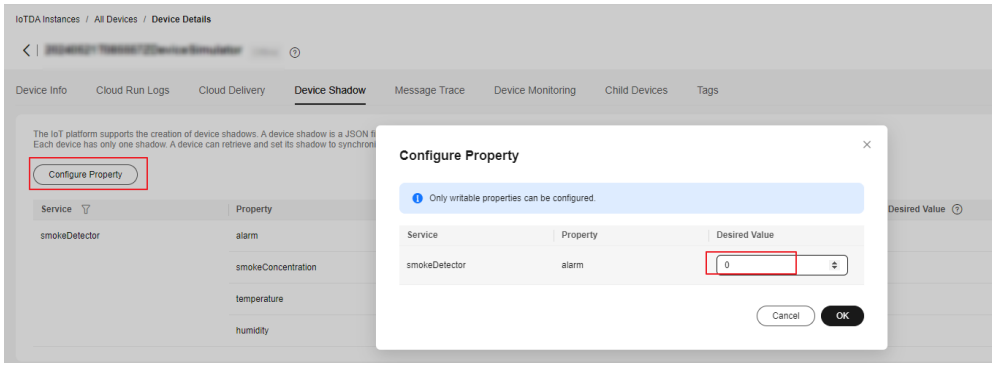
Run the **PropertySample** class and check whether the value of the **alarm** property is **1** on the **Device Shadow** tab page.

Figure 3-6 Device shadow - Viewing property (Alarm)



Change the value of the **alarm** property to **0**.

Figure 3-7 Device shadow - Configuring property (alarm)



In the device logs, the value of **alarm** is **0**.

```
INFO MqttConnection:66 - messageArrived topic =  $oc/devices/[redacted]_demo/sys/properties/set/reque
INFO PropertySample:53 - OnPropertiesSet, serviceId =  smokeDetector
INFO PropertySample:57 - property name = alarm
INFO PropertySample:58 - set property value = 0
```

Delivering a Command

You can set a command listener to receive commands delivered by the platform. The callback API needs to process the commands and report responses.

Command processing in the **CommandSample** example: Print the received command, and then call **respondCommand** method to report the response.

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

            // Process the command.

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

    });
```

Run the **CommandSample** class and deliver a command on the platform. In the command, set **serviceId** to **smokeDetector**, **name** to **ringAlarm**, and **paras** to **duration=20**.

The log shows that the device receives the command and reports a response.

```
INFO MqttConnection:66 - messageArrived topic =  $oc/devices/test_testDevice/sys/commands/request_id=4, msg = {"paras":{"duration":20},"service_id":"smo
INFO CommandSample:62 - onCommand, serviceId = smokeDetector
INFO CommandSample:63 - onCommand , name = ringAlarm
INFO CommandSample:64 - onCommand, paras =  {duration=20}
INFO MqttConnection:213 - publish message topic =  $oc/devices/test_testDevice/sys/commands/response/request_id=4, msg = {"paras":null,"result_code":0,"
```

Object-oriented Programming

Calling device client APIs to communicate with the platform is flexible but requires you to properly configure each API.

The SDK provides a simpler method, object-oriented programming. You can use the product model capabilities provided by the SDK to define device services and call the property reading/writing API to access the device services. In this way, the SDK can automatically communicate with the platform to synchronize properties and call commands.

Object-oriented programming simplifies the complexity of device code and enables you to focus only on services rather than the communications with the platform. This method is much easier than calling client APIs and suitable for most scenarios.

The following uses **smokeDetector** to demonstrate the process of object-oriented programming.

1. Define the service class and properties based on the product model. (If there are multiple services, define multiple service classes.)

```
public static class SmokeDetectorService extends AbstractService {

    // Define properties based on the product model. Ensure that the device name and type are the same as those in the product model. writeable indicates whether the property can be written, and name indicates the property name.
    @Property(name = "alarm", writeable = true)
    int smokeAlarm = 1;

    @Property(name = "smokeConcentration", writeable = false)
    float concentration = 0.0f;

    @Property(writeable = false)
    int humidity;

    @Property(writeable = false)
    float temperature;
```

   @Property indicates a property. You can use **name** to specify a property name. If no property name is specified, the field name is used.

   You can add **writeable** to a property to control permissions on it. If the property is read-only, add **writeable = false**. If **writeable** is not added, the property can be read and written.

2. Define service commands. The SDK automatically calls the commands when the device receives commands from the platform.

   The type of input parameters and return values for APIs cannot be changed. Otherwise, a runtime error occurs.

   The following code defines a ring alarm command named **ringAlarm**. The delivered parameter is **duration**, which indicates the duration of the ringing alarm.

```
// Define the command. The type of input parameters and return values for APIs cannot be changed. Otherwise, a runtime error occurs.
    @DeviceCommand(name = "ringAlarm")
    public CommandRsp alarm(Map<String, Object> paras) {
        int duration = (int) paras.get("duration");
        log.info("ringAlarm  duration = " + duration);
        return new CommandRsp(0);
    }
```

3. Define the getter and setter methods.
   - The device automatically calls the **getter** method after receiving the commands for querying and reporting properties from the platform. The getter method reads device properties from the sensor in real time or from the local cache.

- The device automatically calls the setter method after receiving the commands for setting properties from the platform. The setter method updates the local values of the device. If a property is not writable, leave the setter method not implemented.

```
// Ensure that the names of the setter and getter methods comply with the JavaBean specifications so that the APIs can be automatically called by the SDK.
    public int getHumidity() {

        // Simulate the action of reading data from the sensor.
        humidity = new Random().nextInt(100);
        return humidity;
    }

    public void setHumidity(int humidity) {
        // You do not need to implement this method for read-only fields.
    }

    public float getTemperature() {

        // Simulate the action of reading data from the sensor.
        temperature = new Random().nextInt(100);
        return temperature;
    }

    public void setTemperature(float temperature) {
        // Read-only fields do not need to implement the set method.
    }

    public float getConcentration() {

        // Simulate the action of reading data from the sensor.
        concentration = new Random().nextFloat()*100.0f;
        return concentration;
    }

    public void setConcentration(float concentration) {
        // Read-only fields do not need to implement the set method.
    }

    public int getSmokeAlarm() {
        return smokeAlarm;
    }

    public void setSmokeAlarm(int smokeAlarm) {

        this.smokeAlarm = smokeAlarm;
        if (smokeAlarm == 0){
            log.info("alarm is cleared by app");
        }
    }
```

4. Create a service instance in the **main** function and add the service instance to the device.

```
    // Create a device.
    IoTDevice device = new IoTDevice(serverUri, deviceId, secret);

    // Create a device service.
    SmokeDetectorService smokeDetectorService = new SmokeDetectorService();
    device.addService("smokeDetector", smokeDetectorService);

    if (device.init() != 0) {
        return;
    }
```

5. Enable periodic property reporting.

```
// Enable periodic property reporting.
smokeDetectorService.enableAutoReport(10000);
```

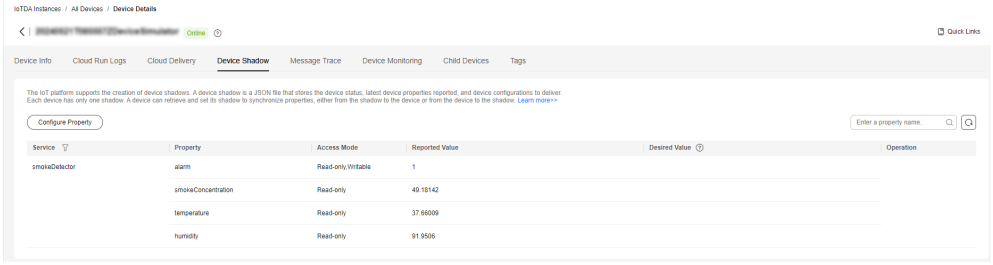

If you do not want to report properties periodically, you can call **firePropertiesChanged** to manually report them.

Run the **SmokeDetector** class to view the logs about property reporting.

```
INFO MqttConnection:140 - try to connect to ssl://[redacted].myhuaweicloud.com:8883
INFO MqttConnection:147 - connect success ssl://[redacted].myhuaweicloud.com:8883
INFO MqttConnection:87 - Mqtt client connected. address :ssl://[redacted].myhuaweicloud.com:8883
connect ok
INFO MqttConnection:213 - publish message topic =  $oc/devices/5e06bfee334dd4f33759f5b3_demo/sys/properties/report, msg = {"service
```

View the device shadow on the platform.

Figure 3-8 Device shadow - Viewing property (Alarm)



Modify the **alarm** property on the platform and view the device logs about property modification.

```
INFO MqttConnection:66 - messageArrived topic =  $oc/devices/test_testDevice/sys/properties/set/request_id=2, msg = {"services":[{"pro
INFO AbstractService:187 - write property ok:alarm
```

Deliver the **ringAlarm** command on the platform.

View the logs about calling the **ringAlarm** command and reporting a response.

```
INFO MqttConnection:66 - messageArrived topic =  $oc/devices/test_testDevice/sys/commands/request_id=1, msg = {"paras":{"duration":20},
INFO DeviceServiceSample$SmokeDetectorService:53 - ringAlarm  duration = 20
INFO MqttConnection:213 - publish message topic =  $oc/devices/test_testDevice/sys/commands/response/request_id=1, msg = {"paras":null,
```

Using the Code Generator

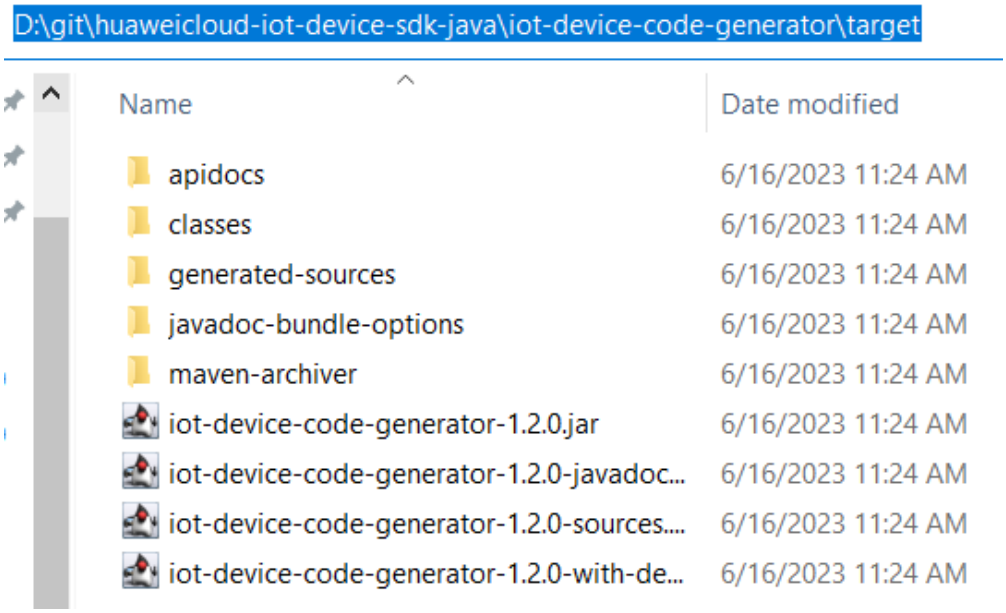
The SDK provides a code generator, which allows you to automatically generate a device code framework only using a product model. The code generator parses the product model, generates a service class for each service defined in the model, and generates a device main class based on the service classes. In addition, the code generator creates a device and registers a service instance in the **main** function.

To use the code generator to generate device code, proceed as follows:

1. Download the **huaweicloud-iot-device-sdk-java** project, decompress it, go to the **huaweicloud-iot-device-sdk-java** directory, and run the **mvn install** command.


```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary for huaweicloud iot device sdk project for java 1.2.0:
[INFO]
[INFO] huaweicloud iot device sdk project for java ........ SUCCESS [  0.802 s]
[INFO] iot-device-sdk-java ................................ SUCCESS [  3.976 s]
[INFO] iot-device-demo .................................... SUCCESS [  4.112 s]
[INFO] iot-bridge-sdk ..................................... SUCCESS [ 17.187 s]
[INFO] iot-bridge-demo .................................... SUCCESS [  4.168 s]
[INFO] iot-gateway-demo ................................... SUCCESS [  2.852 s]
[INFO] iot-device-code-generator .......................... SUCCESS [  2.658 s]
[INFO] iot-bridge-sample-tcp-protocol ..................... SUCCESS [  4.122 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  39.978 s
[INFO] Finished at: 2023-06-16T11:25:00+08:00
[INFO] ------------------------------------------------------------------------
```

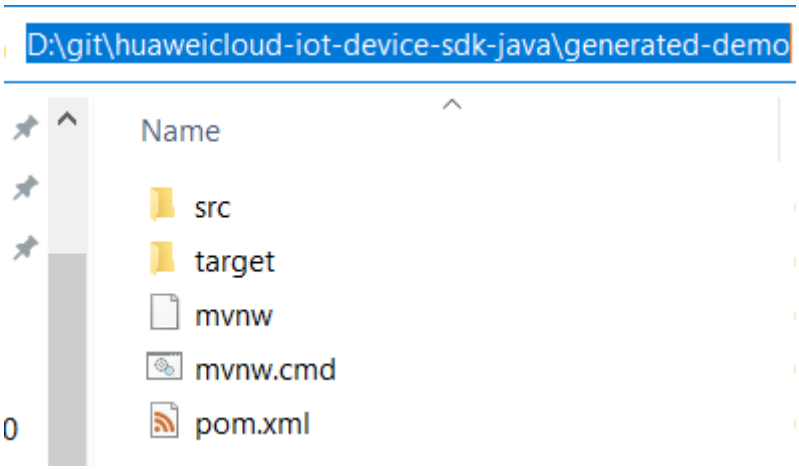
2. Check whether an executable JAR package is generated in the **target** folder of **iot-device-code-generator**.



3. Save the product model to a local directory. For example, save the **smokeDetector.zip** file to disk D.
4. Access the SDK root directory and run the **java -jar .\iot-device-code-generator\target\iot-device-code-generator-1.2.0-with-deps.jar D:\smokeDetector.zip** command.

```
PS D:\git\huaweicloud-iot-device-sdk-java> java -jar .\iot-device-code-generator\target\iot-device-code-generator-1.2.0-with-deps.jar D:\smokeDetector.zip
2023-06-16 11:30:47  INFO DeviceCodeGenerator:147 - the file generation path is :D:\git\huaweicloud-iot-device-sdk-java\generated-demo\src\main\java\com\huaweicloud\sdk\iot\device\demo\smokeDetectorService.java
2023-06-16 11:30:47  INFO DeviceCodeGenerator:73 - demo code generated to: D:\git\huaweicloud-iot-device-sdk-java\generated-demo
```

5. Check whether the **generated-demo** package is generated in the **huaweicloud-iot-device-sdk-java** directory.



The device code is generated.

To compile the generated code, proceed as follows:

1. Go to the **huaweicloud-iot-device-sdk-java\generated-demo** directory, and run the **mvn install** command to generate a JAR package in the **target** folder.

```
PS D:\git\huaweicloud-iot-device-sdk-java> cd .\generated-demo\
PS D:\git\huaweicloud-iot-device-sdk-java\generated-demo> mvn install
```

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.386 s
[INFO] Finished at: 2023-06-16T11:31:47+08:00
[INFO] ------------------------------------------------------------------------
PS D:\git\huaweicloud-iot-device-sdk-java\generated-demo> dir target


    Directory: D:\git\huaweicloud-iot-device-sdk-java\generated-demo\target


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         6/16/2023  11:31 AM                apidocs
d-----         6/16/2023  11:31 AM                classes
d-----         6/16/2023  11:31 AM                generated-sources
d-----         6/16/2023  11:31 AM                javadoc-bundle-options
d-----         6/16/2023  11:31 AM                maven-archiver
-a----         6/16/2023  11:31 AM          29924 iot-device-demo-ganerated-1.2.0-javado
                                                  c.jar
-a----         6/16/2023  11:31 AM           6728 iot-device-demo-ganerated-1.2.0-source
                                                  s.jar
-a----         6/16/2023  11:31 AM       11530020 iot-device-demo-ganerated-1.2.0-with-d
                                                  eps.jar
-a----         6/16/2023  11:31 AM           8031 iot-device-demo-ganerated-1.2.0.jar
```

2. Run the **java -jar .\target\iot-device-demo-ganerated-1.2.0-with-deps.jar ssl://*Domain name*:8883 *device_id secret*** command. The three parameters are the device access address, device ID, and password, respectively. Run the generated demo.

```
D:\git\huaweicloud-iot-device-sdk-java\generated-demo> java -jar .\target\iot-device-demo-ganerated-1.2.0-with-deps.jar ssl://Domain name:8883 5e06bfee334dd4f33759f5b3_demo secret
2024-04-17 15:50:53  INFO AbstractService:73 - create device, the deviceId is 5e06bfee334dd4f33759f5b3_demo
2024-04-17 15:50:54 INFO MqttConnection:204 - try to connect to ssl://Domain name: 8883
2024-04-17 15:50:55 INFO MqttConnection:228 - connect success, the uri is ssl://Domain name: 8883
2024-04-17 15:50:55  INFO MqttConnection:268 - publish message topic is $oc/devices/5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg = {"object_device_id":"5e06bfee334dd4f33759f5b3_demo","services":[{"paras":
```


{"type":"DEVICE_STATUS","content":"connect success","timestamp":"1713340255148"},"service_id":"$log","event_type":"log_report","event_time":"20240417T075055Z","event_id":null}]}
2024-04-17 15:50:55 INFO MqttConnection:111 - Mqtt client connected. address is ssl://***Domain name***: 8883
2024-04-17 15:50:55 INFO MqttConnection:268 - publish message topic is $oc/devices/***5e06bfee334dd4f33759f5b3_demo***/sys/events/up, msg = {"object_device_id":"***5e06bfee334dd4f33759f5b3_demo***","services":[{"paras":{"device_sdk_version":"JAVA_v1.2.0","fw_version":null,"sw_version":null},"service_id":"$sdk_info","event_type":"sdk_info_report","event_time":"20240417T075055Z","event_id":null}]}
2024-04-17 15:50:55 INFO MqttConnection:268 - publish message topic is $oc/devices/***5e06bfee334dd4f33759f5b3_demo*** /sys/events/up, msg = {"object_device_id":"***5e06bfee334dd4f33759f5b3_demo*** ","services": [{"paras":{"type":"DEVICE_STATUS","content":"connect complete, the url is ssl://***Domain name*** :8883","timestamp":"1713340255496"},"service_id":"$log","event_type":"log_report","event_time":"20240417T075055Z","event_id":null}]}
2024-04-17 15:51:03 INFO smokeDetectorService:78 - report property alarm value = 50
2024-04-17 15:51:03 INFO smokeDetectorService:104 - report property temperature value = 0.3648571367849047
2024-04-17 15:51:03 INFO smokeDetectorService:91 - report property smokeConcentration value = 0.679772877336927
2024-04-17 15:51:03 INFO smokeDetectorService:117 - report property humidity value = 15
2024-04-17 15:51:03 INFO MqttConnection:268 - publish message topic is $oc/devices/***5e06bfee334dd4f33759f5b3_demo***/sys/properties/report, msg = {"services":[{"properties":{"alarm":50,"temperature":0.3648571367849047,"smokeConcentration":0.679772877336927,"humidity":15},"service_id":"smokeDetector","event_time":"20240417T075103Z"}]}

To modify the extended code, proceed as follows:

Service definition and registration have already been completed through the generated code. You only need to make small changes to the code.

1. Command API: Add specific implementation logic.

```
/********** ommands **********/
@DeviceCommand
public CommandRsp ringAlarm (Map<String, Object> paras) {
    //todo Add command processing code here.
    return new CommandRsp(0);
}
```

2. **getter** method: Change the value return mode of the generated code from returning a random value to reading from the sensor.
3. **setter** method: Add specific processing logic, such as delivering instructions to the sensor, because the generated code only modifies and saves the properties.

Developing a Gateway

Gateways are special devices that provide child device management and message forwarding in addition to the functions of common devices. The SDK provides the **AbstractGateway** class to simplify gateway implementation. This class can collect and save child device information (with a data persistence API), forward message responses (with a message forwarding API), and report child device list, properties, statuses, and messages.

- **AbstractGateway Class**

  Inherit this class to provide APIs for persistently storing device information and forwarding messages to child devices in the constructor.


```
public abstract void onSubdevCommand(String requestId, Command command);

public abstract void onSubdevPropertiesSet(String requestId, PropsSet propsSet);

public abstract void onSubdevPropertiesGet(String requestId, PropsGet propsGet);

public abstract void onSubdevMessage(DeviceMessage message);
```

- **iot-gateway-demo Code**

  The **iot-gateway-demo** project implements a simple gateway with **AbstractGateway** to connect TCP devices. The key classes include:

  SimpleGateway: inherited from **AbstractGateway** to manage child devices and forward messages to child devices.

  StringTcpServer: implements a TCP server based on Netty. In this example, child devices support the TCP protocol, and the first message is used for authentication.

  SubDevicesFilePersistence: persistently stores child device information in a JSON file and caches the file in the memory.

  Session: stores the mapping between device IDs and TCP channels.

- **SimpleGateway Class**

  Adding or Deleting a Child Device

  Adding a child device: **onAddSubDevices** of **AbstractGateway** can store child device information. Additional processing is not required, and **onAddSubDevices** does not need to be overridden for **SimpleGateway**.

  Deleting a child device: You need to modify persistently stored information of the child device and disconnect the device from the platform. Therefore, **onDeleteSubDevices** is overridden to add the link release logic, and **onDeleteSubDevices** in the parent class is called.

```
@Override
  public int onDeleteSubDevices(SubDevicesInfo subDevicesInfo) {

      for (DeviceInfo subdevice : subDevicesInfo.getDevices()) {
          Session session = nodeIdToSesseionMap.get(subdevice.getNodeId());
          if (session != null) {
              if (session.getChannel() != null) {
                  session.getChannel().close();
                  channelIdToSessionMap.remove(session.getChannel().id().asLongText());
                  nodeIdToSesseionMap.remove(session.getNodeId());
              }
          }
      }
      return super.onDeleteSubDevices(subDevicesInfo);

  }
```

- **Processing Messages to Child Devices**

  The gateway needs to forward messages received from the platform to child devices. The messages from the platform include device messages, property reading/writing, and commands.

  – **Device messages**: Obtain the nodeId based on the deviceId, and then obtain the session of the device to get a channel for sending messages. You can choose whether to convert messages during forwarding.

```
@Override
  public void onSubdevMessage(DeviceMessage message) {

      // Each platform API carries a deviceId, which consists of a nodeId and productId.
```



```
        //deviceId = productId_nodeId
        String nodeId = IotUtil.getNodeIdFromDeviceId(message.getDeviceId());
        if (nodeId == null) {
            return;
        }

        // Obtain the session based on the nodeId for a channel.
        Session session = nodeIdToSesseionMap.get(nodeId);
        if (session == null) {
            log.error("subdev is not connected " + nodeId);
            return;
        }
        if (session.getChannel() == null){
            log.error("channel is null " + nodeId);
            return;
        }

        // Directly forward messages to the child device.
        session.getChannel().writeAndFlush(message.getContent());
        log.info("writeAndFlush " + message);
    }
```

- **Property Reading and Writing**

  Property reading and writing include property setting and query.

  Property setting:

```
@Override
    public void onSubdevPropertiesSet(String requestId, PropsSet propsSet) {

        if (propsSet.getDeviceId() == null) {
            return;
        }

        String nodeId = IotUtil.getNodeIdFromDeviceId(propsSet.getDeviceId());
        if (nodeId == null) {
            return;
        }

        Session session = nodeIdToSesseionMap.get(nodeId);
        if (session == null) {
            return;
        }

        // Convert the object into a string and send the string to the child device. Encoding/
Decoding may be required in actual situations.
        session.getChannel().writeAndFlush(JsonUtil.convertObject2String(propsSet));

        // Directly send a response. A more reasonable method is to send a response after the
child device processes the request.
        getClient().respondPropsSet(requestId, IotResult.SUCCESS);

        log.info("writeAndFlush " + propsSet);

    }
```

  Property query:

```
@Override
    public void onSubdevPropertiesGet(String requestId, PropsGet propsGet) {

        // Send a failure response. It is not recommended that the platform directly reads the
properties of the child device.
        log.error("not support onSubdevPropertiesGet");
        deviceClient.respondPropsSet(requestId, IotResult.FAIL);
    }
```

- **Commands**: The procedure is similar to that of message processing. Different types of encoding/decoding may be required in actual situations.


```
@Override
    public void onSubdevCommand(String requestId, Command command) {

        if (command.getDeviceId() == null) {
            return;
        }

        String nodeId = IotUtil.getNodeIdFromDeviceId(command.getDeviceId());
        if (nodeId == null) {
            return;
        }

        Session session = nodeIdToSesseionMap.get(nodeId);
        if (session == null) {
            return;
        }

        // Convert the command object into a string and send the string to the child device.
Encoding/Decoding may be required in actual situations.
        session.getChannel().writeAndFlush(JsonUtil.convertObject2String(command));

        // Directly send a response. A more reasonable method is to send a response after the
child device processes the request.
        getClient().respondCommand(requestId, new CommandRsp(0));
        log.info("writeAndFlush " + command);
    }
```

- **Upstream Message Processing**

  Upstream message processing is implemented by **channelRead0** of **StringTcpServer**. If no session exists, create a session.

  If the child device information does not exist, the session cannot be created and the connection is rejected.

```
@Override
    protected void channelRead0(ChannelHandlerContext ctx, String s) throws Exception {
        Channel incoming = ctx.channel();
        log.info("channelRead0" + incoming.remoteAddress() + " msg :" + s);

        // Create a session for the first message.
// Create a session for the first message.
        Session session = simpleGateway.getSessionByChannel(incoming.id().asLongText());
        if (session == null) {
            String nodeId = s;
            session = simpleGateway.createSession(nodeId, incoming);

            // The session fails to create and the connection is rejected.
            if (session == null) {
                log.info("close channel");
                ctx.close();
            }
        }
```

  If the session exists, the message is forwarded.

```
else {
            // Call reportSubDeviceProperties to report properties of the child device.
            DeviceMessage deviceMessage = new DeviceMessage(s);
            deviceMessage.setDeviceId(session.getDeviceId());
            simpleGateway.reportSubDeviceMessage(deviceMessage, null);

        }
```

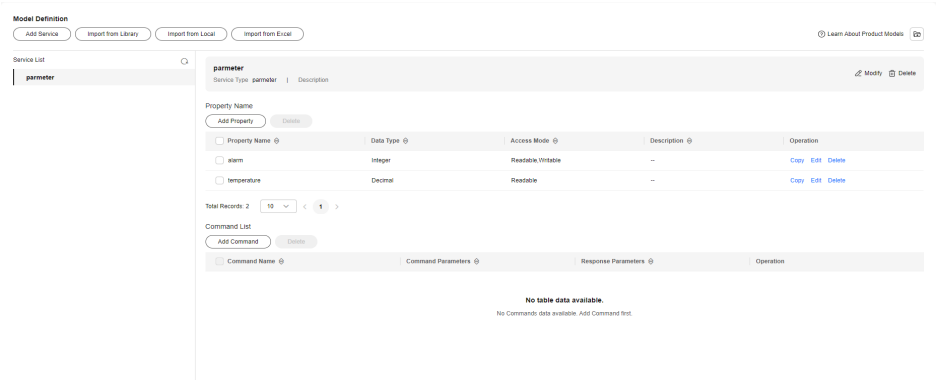
  For details about the gateway, view the source code. The demo is open-source and can be extended as required. For example, you can modify the persistence mode, add message format conversion during forwarding, and support other device access protocols.

- **Using iot-gateway-demo**

  a. Create a product for the child device. For details, see **Creating a Product**.

  b. Define a model in the created product and add a service whose ID is **parameter**. Add **alarm** and **temperature** properties, as shown in the following figure.

  Figure 3-9 Model definition - Child device product



  c. Modify the **main** function of **StringTcpServer** by replacing the constructor parameters, and run this class.

```
simpleGateway = new SimpleGateway(new SubDevicesFilePersistence(),
                "ssl://iot-acc.cn-north-4.myhuaweicloud.com:8883",
                "5e06bfee334dd4f33759f5b3_demo", "mysecret");
```

  d. After the gateway is displayed as **Online** on the platform, add a child device.

  Figure 3-10 Device - Adding a child device

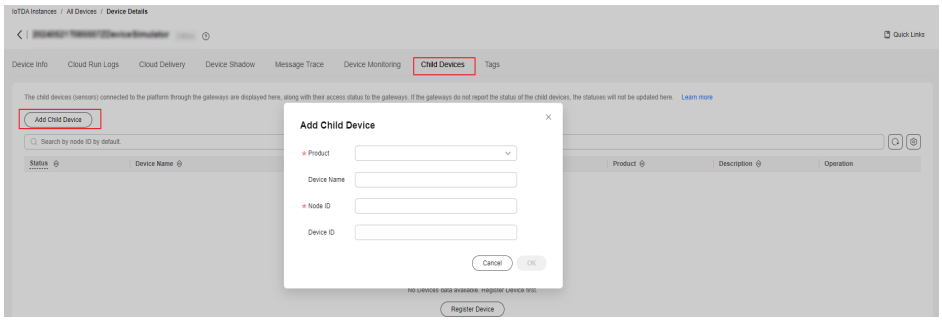


Table 3-2 Child device parameters

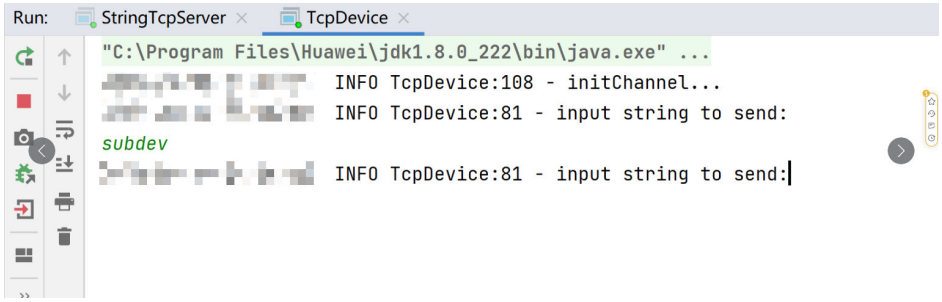
Parameter	Description
Product	Product to which the child device belongs. Select the product created in **1**.
Device Name	Customize a device name, for example, **subdev_name**.
Node ID	Enter **subdev**.
Device ID	This parameter is optional and is automatically generated.

A log similar to the following is displayed on the gateway:

2024-04-16 21:00:01 INFO SubDevicesFilePersistence:112 - add subdev, the nodeId is subdev

e. Run the TcpDevice class. After the connection is set up, enter the node ID of the child device registered in step 3, for example, **subdev**.

Figure 3-11 Child device connection

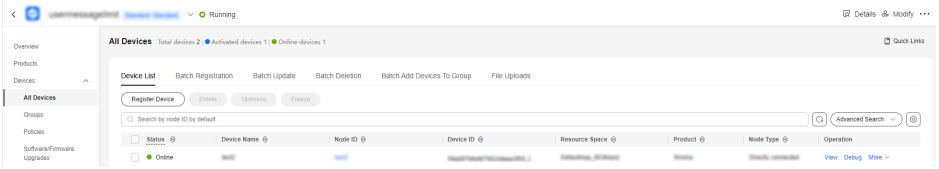


A log similar to the following is displayed on the gateway:

2024-04-16 21:00:54 INFO StringTcpServer:196 - initChannel: /127.0.0.1:21889
2024-04-16 21:01:00 INFO StringTcpServer:137 - channelRead0 is /127.0.0.1:21889, the msg is ***subdev***
2024-04-16 21:01:00 INFO SimpleGateway:100 - create new session ok, the session is Session{nodeId='***subdev***', channel=[id: 0xf9b89f78, L:/127.0.0.1:8080 - R:/127.0.0.1:21889], deviceId='***subdev_deviceId***'}

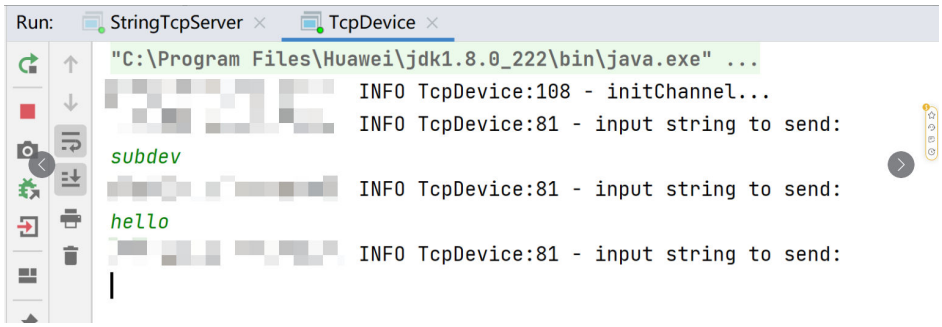
f. Check whether the child device is online on the platform.

Figure 3-12 Device list - Device online status



g. Enable the child device to report messages.

Figure 3-13 Enable the child device to report messages.



Logs similar to the following show that the message is reported.

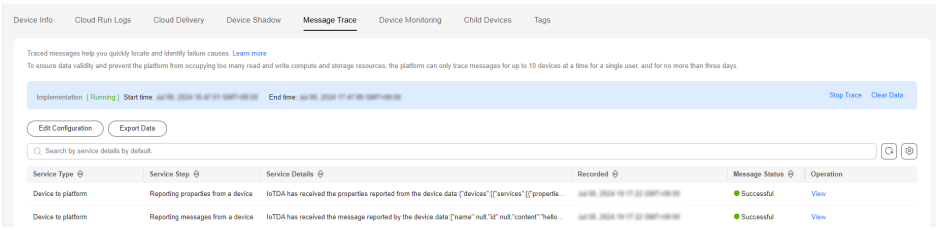
2024-04-16 21:02:36 INFO StringTcpServer:137 - channelRead0 is /127.0.0.1:***21889***, the msg is hello


```
2024-04-16 21:02:36  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/messages/up, msg =
{"name":null,"id":null,"content":"hello","object_device_id":"subdev_deviceId"]
2024-04-16 21:02:36  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/gateway/sub_devices/properties/report, msg =
{"devices":[{"services":[{"properties":
{"temperature":2,"alarm":1},"service_id":"parameter","event_time":null}],"device_id":"subdev_de
viceId"}]]
```

h. View the messages traced.

Click **Message Trace** on the gateway details page. Send data from the child device to the platform, and view the messages after a while.

Figure 3-14 Message tracing - Directly connected device



3.3 IoT Device C SDK for Linux/Windows

The IoT Device C SDK for Linux/Windows provides abundant demo code for devices to communicate with the platform and implement device, gateway, and Over-The-Air (OTA) services. For details about the integration guide, see **IoT Device C SDK for Linux/Windows**.

Requirements

- The SDK runs on Linux.
- The SDK depends on the OpenSSL and Paho libraries. If you have your own compilation chain, compile library files such as OpenSSL, Paho, zlib, and Huawei secure function library.
- For some devices that are connected in MCU+module mode, use the C Tiny SDK for development.

NOTE

For details, see **README**.

Change History

Table 3-3 Change history

Version	Change	Description
1.2.0	Function enhancement	Added the SDK test code and demo, and optimized the code usage.
1.1.5	Function enhancement	Updated the OTA upgrade transmission format.

Version	Change	Description
1.1.4	Function enhancement	Fixed the issue of remote login packet reporting timeout.
1.1.3	Function enhancement	Updated the conf\rootcert.pem certificates.
1.1.2	New function	Added device rules, M2M, GN compilation file, anomaly detection, timestamp printed in logs, MQTT_DEBUG, Chinese cryptographic algorithm, remote configuration, and device-cloud secure communication (soft bus).
1.1.1	New function	Added SSH remote O&M.
1.1.0	New function	Supported MQTT 5.0. Optimized the cod to resolve the memory overflow issue.
1.0.1	Function enhancement	Added application scenarios, where MQTTS does not verify the platform public key, using TLS version is V1.2, and adding message storage examples.
0.9.0	New function	Added the API for the gateway to update the child device status.
0.8.0	Function enhancement	Added the access domain name (iot-mqtts.cn-north-4.myhuaweicloud.com) and root certificates. If the device uses the old domain name (iot-acc.cn-north-4.myhuaweicloud.com) for access, use the v0.5.0 SDK.
0.5.0	Function enhancement	Preset the device access address and the matching CA certificate in the SDK to support interconnection with the Huawei Cloud IoT platform.

3.4 IoT Device C# SDK

The IoT Device C# SDK provides abundant demo code for devices to communicate with the platform and implement advanced services such as device, gateway, and OTA services. For details about the integration guide, see **IoT Device C# SDK**.

Requirements

- .NET SDK 8.0 has been installed.
  - **.NET installation guide**
  - **.NET 8.0.**
- The corresponding IDE (Visual Studio Code 2017+, Rider 17.0.6+) has been installed. This SDK does not depend on the IDE. You can select the IDE or directly use the CLI as required.

NOTE

For details, see **README**.

Change History

Table 3-4 Change history

Version	Change	Description
1.3.4	Function enhancement	1. Optimized the log printing function. 2. Modified the topic returned by SubscribeTopic starting with oc. 3. Optimized demos. 4. Fixed the bug of the gateway interface. 5. Upgraded the target framework. 6. Optimized other features.
1.3.3	New function	Supported gateway mode for OTA upgrade.
1.3.2	Function enhancement	Updated the CA certificate for the server.
1.3.1	Fixing	Resolved issues such as null pointer exceptions and MQTT object release failures.
1.3.0	New function	Supported OBS-based upgrade of software and firmware packages.
1.2.0	New function	Added the generic-protocol function.
1.1.1	Function enhancement	Added the function of deleting child devices from a gateway and optimized the description.
1.1.0	New function	Added the gateway and product model functions.
1.0.0	First release	Provided basic device access capabilities. Preset the device access address and the CA certificate matching Huawei IoTDA in the SDK.

3.5 IoT Device Android SDK

The IoT Device Android SDK provides abundant demo code for devices to communicate with the platform and implement advanced services such as device,

gateway, and OTA services. For details about the integration guide, see **IoT Device Android SDK**.

Requirements

Android Studio has been installed.

NOTE

For details, see **README**.

Change History

Table 3-5 Change history

Version	Change	Description
1.0.0	First release	Provided basic device access capabilities.

3.6 IoT Device Go SDK

The IoT Device Go SDK provides abundant demo code for devices to communicate with the platform and implement advanced services such as device and OTA services. For details about the integration guide, see **IoT Device Go SDK**.

Requirements

- Go 1.18 or later has been installed.
- The dependencies have been installed based on go.mod.

NOTE

For details, see **README**.

Change History

Table 3-6 Change history

Version	Change	Description
v1.0.0	New function	Provided capabilities for connections to the Huawei Cloud IoT platform to facilitate service scenarios such as secure access, device management, data collection, command delivery, device provisioning, and device rules.

3.7 IoT Device Tiny C SDK for Linux/Windows

The IoT Device Tiny C SDK for Linux/Windows is lightweight interconnection middleware deployed on devices that have wide area network (WAN) capabilities

and limited power consumption, storage, and computing resources. After the SDK is deployed on such devices, you only need to call APIs to enable the devices to connect to the IoT platform, report data, and receive commands. For details about the integration, see **development guide on device-cloud communication components**.

NOTE

The IoT Device SDK Tiny can run on devices that do not run Linux OS, and can also be integrated into modules. However, it does not provide gateway services.

Requirements

- It adapts to LiteOS, Linux, macOS, and FreeRTOS. You can modify the SDK to **adapt to other environments**.
- For details about different modules, see the **SDK development board porting list**.

FAQ

LwM2M/CoAP Device Access

3.8 IoT Device Python SDK

The IoT Device Python SDK provides abundant demo code for devices to communicate with the platform and implement device, gateway, and OTA services. For details, see **IoT Device Python SDK**.

Requirements

- Python 3.11.4 has been installed.
- The third-party class library paho-mqtt 2.0.0 has been installed (mandatory).
- The third-party class library schedule 1.2.2 has been installed (mandatory).
- The third-party class library APScheduler 3.10.4 has been installed (mandatory).
- The third-party class library requests 2.32.2 has been installed (optional, used in the demo of gateway and child device management).
- The third-party class library tornado 6.3.3 has been installed (optional, used in the demo of gateway and child device management).

NOTE

For details about how to install the components, see **IoT Device SDK (Python) Usage Guide**.

Change History

Table 3-7 Change history

Version	Change	Description
1.2.0	New function	Added the functions of rule engine, device provisioning, customized reconnection upon disconnection, and component version upgrade.
1.1.4	New function	Supported gateway mode for OTA upgrade.
1.1.3	Function enhancement	Updated the CA certificate for the server.
1.1.2	New function	Supported MicroPython and the corresponding demo, OTA downloading from OBS, and description documents.
1.1.1	New function	Provided capabilities for connections to the Huawei Cloud IoT platform to facilitate service scenarios such as secure access, device management, data collection, and command delivery.

3.9 IoT Device ArkTS (OpenHarmony) SDK

The IoT Device ArkTS (OpenHarmony) SDK provides abundant demo code for devices to communicate with the platform.

Preparations

- DevEco Studio 5.0.0 or later has been installed.
  - **Installing DevEco Studio**
  - **DevEco Studio Downloading**
- The matching Node.js has been installed.

Requirements

- Download and installation: In DevEco Studio, run the following command to import and install the SDK.
  ```
  ohpm install @huaweicloud/iot-device-sdk
  ```
- Permission configuration: To use the SDK, add the **ohos.permission.INTERNET** permission to **requestPermissions** in the **module.json5** file.
  ```
  {
    "module": {
      "requestPermissions": [
        {
          "name": "ohos.permission.INTERNET"
        }
      ]
  ```



```
    }
}
```

Creating a Product

A smoke detector product model is provided to help you understand the product model. This smoke detector can report the smoke density, temperature, humidity, and smoke alarms, and execute the ring alarm command. The following uses the smoke detector as an example to introduce the procedures of message reporting and property reporting.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card. Check and save the MQTTS device access domain name.

Step 2 Choose **Products** in the navigation pane and click **Create Product**.

Step 3 Set the parameters as prompted and click **OK**.

Set Basic Info	
Resource Space	The platform automatically allocates the created product to the default resource space. If you want to allocate the product to another resource space, select the resource space from the drop-down list box. If a **resource space** does not exist, create it first.
Product Name	Customize the product name. The name can contain letters, numbers, underscores (_), and hyphens (-).
Protocol	Select **MQTT**.
Data Type	Select **JSON**.
Device Type Selection	Select **Custom**.
Device Type	Select **smokeDetector**.
Advanced Settings	
Product ID	Leave this parameter blank.
Description	Set this parameter based on service requirements.

----End

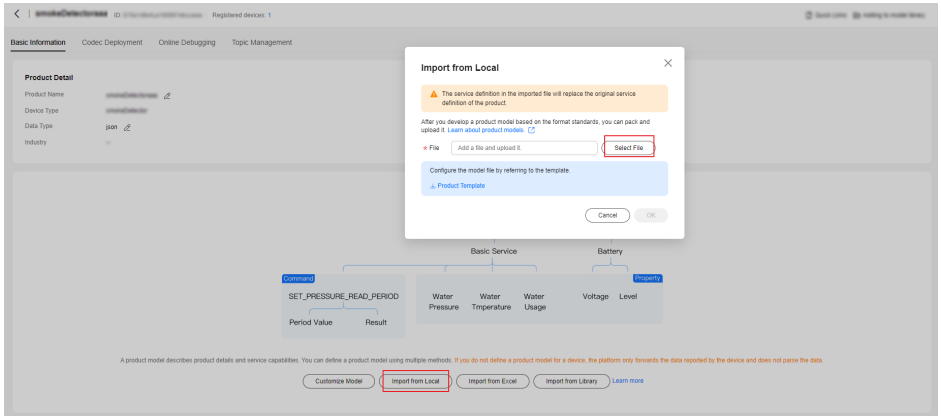
Uploading a Product Model

Step 1 Download the **smokeDetector product model file**.

Step 2 Click the name of the product created in **3** to access its details.

Step 3 On the **Basic Information** tab page, click **Import from Local** to upload the product model file obtained in **1**.

Figure 3-15 Product - Uploading a product model



----End

Registering a Device

Step 1 In the navigation pane, choose **Devices** > **All Devices**, and click **Register Device**.

Step 2 Set the parameters as prompted and click **OK**.

Parameter	Description
Resource Space	Ensure that the device and the product created in **3** belong to the same resource space.
Product	Select the product created in **3**.
Node ID	This parameter specifies the unique physical identifier of the device. The value can be customized and consists of letters and numbers.
Device Name	Customize the device name.
Authentication Type	Select **Secret**.
Secret	Customize the device secret. If this parameter is not set, the platform automatically generates a secret.

After the device is registered, save the node ID, device ID, and secret.

----End

Initializing a Device

NOTE

Demo: **entry/src/main/ets/pages/Index.ets**

1. Enter the device ID and secret obtained in **Registering a Device** and the device connection information obtained in **1**. The format is *ssl://Domain name:Port* or *ssl://IP address:Port*.


```
private device: IoTDevice | null = null;

// Replace the access address, device ID, device secret, and certificate path with your own ones. (Use the corresponding certificate when connecting to Huawei Cloud. The certificate file is stored in resource/resfile. Download the certificate file.)
this.device = new IoTDevice ("ssl://Domain name:8883","deviceId","mySecret","filePath");
```

2. Call the **init** method to establish a connection in either asynchronous or synchronous initialization mode.

```
// Perform initialization in asynchronous mode.
this.device.init().then((data: boolean) => {
// Connection succeeded.
}).catch((err: string) => {
// Connection failed.
})

// Perform initialization in synchronous mode.
// await this.device.init();
```

3. Check device logs. The device is successfully connected.

```
IoTDA_SDK# connect result is {"code":0,"message":"Connect Success"}
```

4. After the device is created and connected, it can be used for communication. Call the **client** method of the IoT Device SDK to obtain the device client. The client provides communication APIs for processing messages, properties, and commands.

Reporting a Message

NOTE

Demo: **entry/src/main/ets/pages/MessageSample.ets**

Message reporting is the process in which a device reports messages to the platform.

1. After the device is initialized and connected to the platform, call the **reportDeviceMessage** method of the client to report device messages and call the **publishRawMessage** method to report messages through custom topics.

```
 // Report messages through system topics.
 const reportMessage: DeviceMessage = { content: this.message }
 this.device.client.reportDeviceMessage(reportMessage)
   .then((data: IoTMqttResponse) => {
     LogUtil.info(TAG, `report deviceMessage success ${JSON.stringify(data)}. DeviceMessage is ${JSON.stringify(reportMessage)}`);
   })
   .catch((error: IoTMqttResponse | string) => {
     LogUtil.error(TAG, `report deviceMessage failed ${JSON.stringify(error)}`);
   })

 // Report messages through custom topics starting with $oc, which must be configured on the platform first.
 const topic = `$oc/devices/${this.device?.deviceId}/user/test`;
 const rawMessage: RawMessage = {
   topic: topic,
   qos: 0,
   payload: this.message
 }
 this.device?.client.publishRawMessage((rawMessage)).then((res: IoTMqttResponse) => {
   LogUtil.info(TAG, `publish rawMessage(${rawMessage.topic}) success, message is ${JSON.stringify(rawMessage)}}`);
 }).catch((error: IoTMqttResponse | string) => {
   LogUtil.error(TAG, `publish rawMessage(${rawMessage.topic}) failed, error is ${JSON.stringify(error)}}`);
 });
```



```
// Report messages through custom topics not starting with $oc (controlled by device topic policies).
const topic = "hello/world";
const rawMessage: RawMessage = {
  topic: topic,
  qos: 0,
  payload: this.message
}
this.device?.client.publishRawMessage((rawMessage)).then((res: IoTMqttResponse) => {
  LogUtil.info(TAG, `publish rawMessage(${rawMessage.topic}) success, message is ${JSON.stringify(rawMessage)}}`);
}).catch((error: IoTMqttResponse | string) => {
  LogUtil.error(TAG, `publish rawMessage(${rawMessage.topic}) failed, error is ${JSON.stringify(error)}}`);
});
```

2. The logs show that the message is successfully sent.

Figure 3-16 Logs for reporting messages through system topics

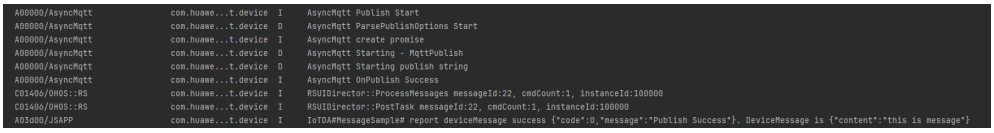


Figure 3-17 Logs for reporting messages through custom topics (starting with $oc)

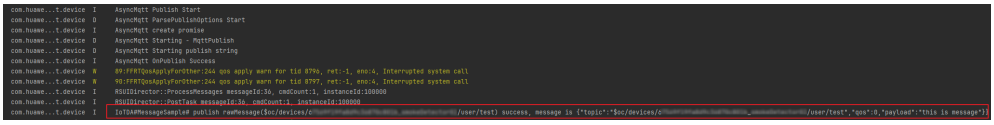
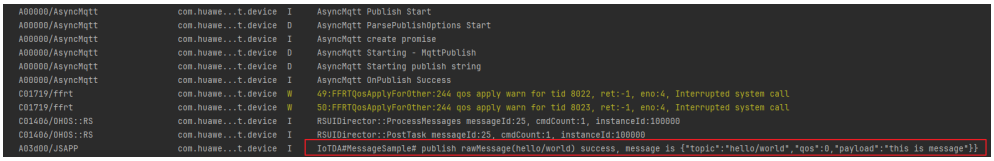
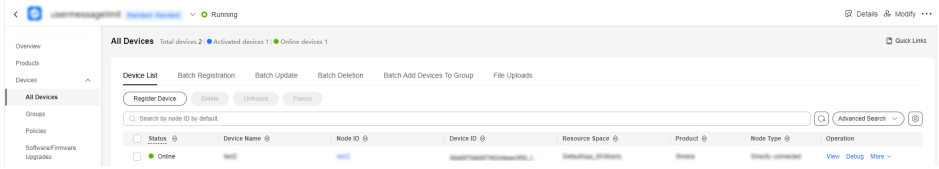


Figure 3-18 Logs for reporting messages through custom topics (not starting with $oc)



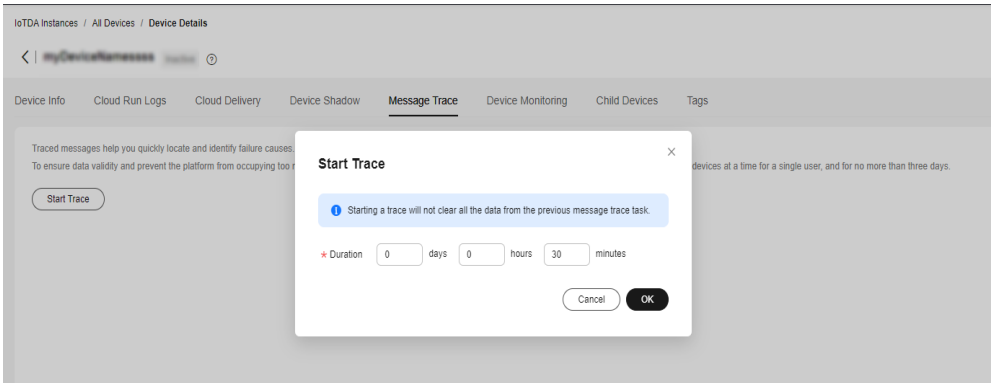
3. On the IoTDA console, choose **Devices** > **All Devices** and check whether the device is online.

Figure 3-19 Device list - Device online status



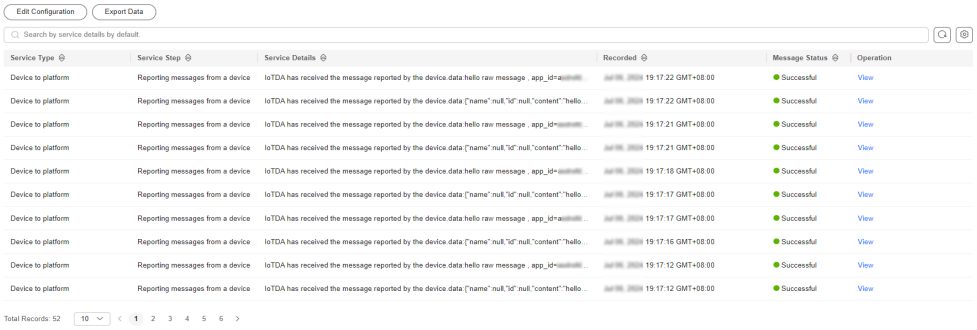
4. Select the device, click the button to access its details page, and enable message tracing.

Figure 3-20 Message tracing - Starting message tracing



5. The message tracing results show that the platform successfully receives the message from the device.

Figure 3-21 Message tracing - Viewing device_sdk_java tracing result



NOTICE

Message tracing may be delayed. If no data is displayed, wait for a while and refresh the page.

Reporting Properties

NOTE

Demo: **entry/src/main/ets/pages/PropertySample.ets**

1. After the device is initialized and connected to the platform, call the **reportProperties** method of the client to report device properties.

```
const properties: ServiceProperty[] =
[
  {
    "service_id": "smokeDetector",
    "properties": {
      "alarm": 1,
      "temperature": Math.random() * 100,
      "humidity": Math.random() * 100,
      "smokeConcentration": Math.random() * 100,
    }
  }
]
;
```



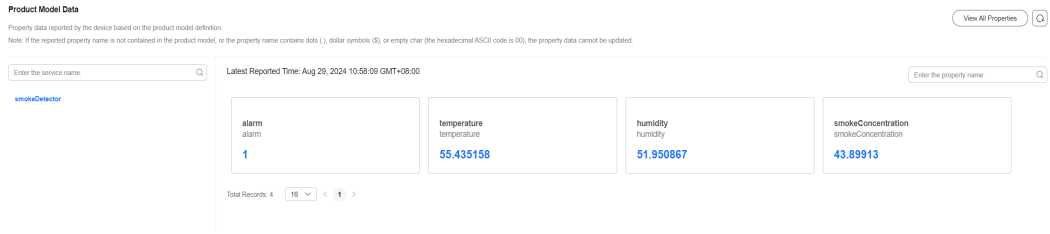
```
this.device.client.reportProperties(properties)
  .then((data: IoTMqttResponse) => {
    LogUtil.info(TAG, `report properties success ${JSON.stringify(data)}, properties is $
{JSON.stringify(properties)}}`);  })
  .catch((error: IoTMqttResponse | string) => {
    LogUtil.error(TAG, `report properties failed ${JSON.stringify(error)}`);
  })
```

2. The logs show that the properties are successfully sent.

Figure 3-22 Logs for reporting properties

3. On the IoTDA console, choose **Devices** > **All Devices**, and click the target device to access its details page. The latest reported property values are displayed.

Figure 3-23 Product model - Property reporting



Reading and Writing Properties

NOTE

Demo: **entry/src/main/ets/pages/PropertySample.ets**

1. After the device is successfully initialized, call the **propertyListener** method of the client to set the property callback interface.
   - Property reading: Only the **alarm** property can be written.
   - Property reading: Assemble the local property value based on the API format.

```
let propertyListener: PropertyListener = {
  onPropertiesSet: (requestId: string, services: ServiceProperty[]): void => {
    this.logArr.unshift(`${new Date()}: onPropertiesSet requestId is ${requestId}, services is $
{JSON.stringify(services)}`)
    // Traverse services.
    services.forEach(serviceProperty => {
      LogUtil.info("onPropertiesSet, serviceId is ", serviceProperty.service_id);
      // Traverse properties.
      Object.keys(serviceProperty.properties).forEach(name => {
        LogUtil.log(TAG, `property name is ${name}`);
        LogUtil.log(TAG, `set property value is ${serviceProperty.properties[name]}`);
      })
    })

    // Change the local properties.
    this.device?.client.respondPropsSet(requestId, IotResult.SUCCESS);
  },
  onPropertiesGet: (requestId: string, serviceId?: string): void => {
    this.logArr.unshift(`${new Date()}: onPropertiesGet requestId is ${requestId}, serviceId is $
{serviceId} and respondPropsGet`)
    LogUtil.info(TAG, `onPropertiesGet, the serviceId is ${serviceId}`);
```



```
    const serviceProperties: ServiceProperty[] = [
      {
        "service_id": "smokeDetector",
        "properties": {
          "alarm": 1,
          "temperature": Math.random() * 100,
          "humidity": Math.random() * 100,
          "smokeConcentration": Math.random() * 100,
        }
      }
    ];
    this.device?.client.respondPropsGet(requestId, serviceProperties);
  }
}
// Set the property listener.
this.device.client.propertyListener = propertyListener;
```

NOTE

- The property reading/writing API must call the **respondPropsGet** and **respondPropsSet** methods to report the operation result.
- If the device does not allow the platform to proactively read data from the device, the **onPropertiesGet** method can be left not implemented.

2. Run the preceding code to set the property listener. On the device shadow page of the platform, check the value of **alarm**, which is 1. Change the value to **0**, and check the device logs. The results show that the device receives a request for setting the value of **alarm** to **0**.

Figure 3-24 Device shadow - Viewing property (Alarm)

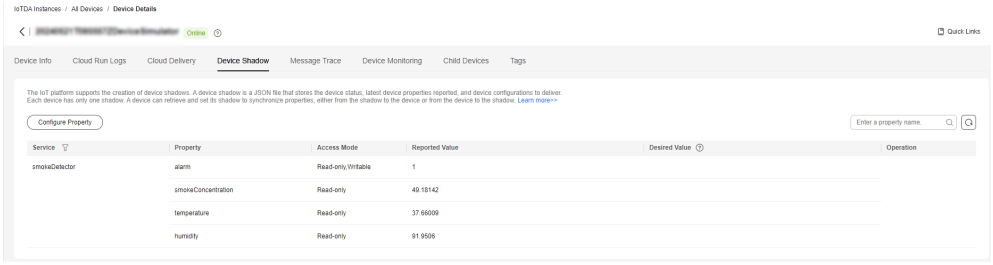


Figure 3-25 Device shadow - Configuring property (alarm)

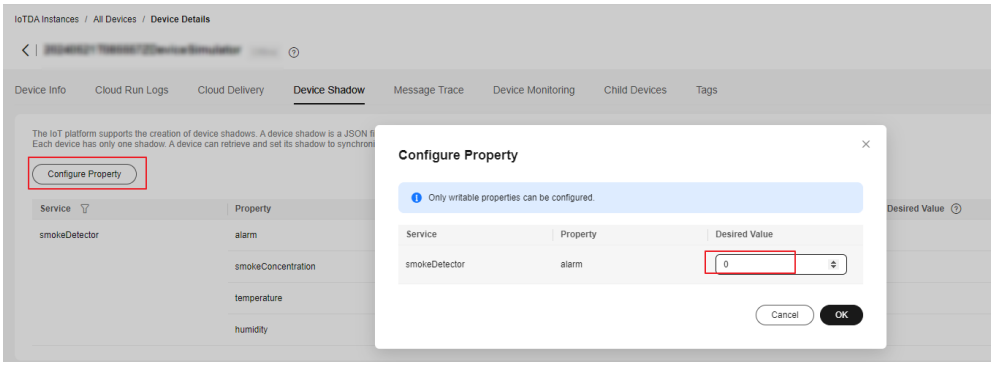
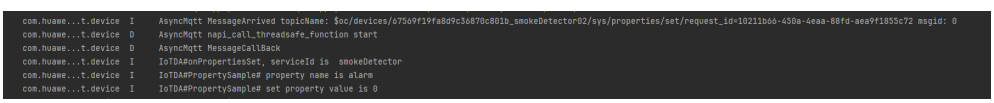


Figure 3-26 Checking property setting results



Delivering a Command

NOTE

Demo: **entry/src/main/ets/pages/CommandSample.ets**

You can set a command listener to receive commands delivered by the platform. The callback API needs to process the commands and report responses.

1. Command processing in the **CommandSample** example: Print the received command, and then call **respondCommand** method to report the response.

```
let commandListener: CommandListener = {
  onCommand: (requestId: string, serviceId: string, commandName: string, paras: object): void => {
    const command = `requestId is ${requestId}, serviceId is ${serviceId}, commandName is $
{commandName}, paras is ${JSON.stringify(paras)}`;
    LogUtil.info(TAG, `received command is ${command}`);
    // Process commands.

    const commandRsp: CommandRsp = {
      result_code: 0
    }
    this.device?.client.respondCommand(requestId, commandRsp).then((data: IoTMqttResponse) => {
      LogUtil.info(TAG, `respond command success ${JSON.stringify(data)}, commandRsp is $
{commandRsp}}`);
    }).catch((err: IoTMqttResponse | string) => {
      LogUtil.error(TAG, `respond command failed ${JSON.stringify(err)}`);
    })
  }
}
this.device.client.commandListener = commandListener;
```

2. After the preceding code is executed to set the command listening, deliver a command on the platform, in which **serviceId** is **smokeDetector**, command name is **ringAlarm**, and **duration** is **20**.
3. Check the log, which indicates that the device receives the command and reports a response successfully.

```
com.huawe...t.device  I    IoTDA#CommandSample# received command is requestId is dc0992cc-6212-4eaa-bef8-682545133d12, serviceId is smokeDetector, commandName is ringAlarm, paras is {"duration":20}
com.huawe...t.device  I    AsyncMqtt Publish Start
com.huawe...t.device  D    AsyncMqtt ParsePublishOptions Start
com.huawe...t.device  I    AsyncMqtt create promise
com.huawe...t.device  D    AsyncMqtt Starting - MqttPublish
com.huawe...t.device  D    AsyncMqtt Starting publish string
com.huawe...t.device  I    AsyncMqtt OnPublish Success
com.huawe...t.device  W    26:FFRTQosApplyForOther:244 qos apply warn for tid 7133, ret:-1, eno:4, Interrupted system call
com.huawe...t.device  W    27:FFRTQosApplyForOther:244 qos apply warn for tid 7134, ret:-1, eno:4, Interrupted system call
com.huawe...t.device  I    IoTDA#CommandSample# respond command success {"code":0,"message":"Publish Success"}, commandRsp is {"result_code":0}}
```

Object-oriented Programming

NOTE

Demo: **entry/src/main/ets/pages/ProfileSample.ets**

Calling device client APIs to communicate with the platform is flexible but requires you to properly configure each API.

The SDK provides a simpler method, object-oriented programming. You can use the product model capabilities provided by the SDK to define device services and call the property reading/writing API to access the device services. In this way, the SDK can automatically communicate with the platform to synchronize properties and call commands.

Object-oriented programming simplifies the complexity of device code and enables you to focus only on services rather than the communications with the platform. This method is much easier than calling client APIs and suitable for most scenarios.

ProfileSample demonstrates the process of object-oriented programming.

1. Define a smoke detector service class, which is inherited from **AbstractService**. (If there are multiple services, define multiple service classes.)

```
class SmokeDetector extends AbstractService  {

}
```

2. Define service properties. Private variables start with underscores (_). Use **@Reflect.metadata ("Property", { name: "string", writeable: boolean})** to indicate a property. The name must be the same as the property name in the product model. **writeable** indicates whether the property is writable.

```
  @Reflect.metadata("Property", { name: "alarm", writeable: true })
  private _smokeAlarm: number = 1;

  @Reflect.metadata("Property", { name: "smokeConcentration", writeable: false })
  private _concentration: number = 0;

  @Reflect.metadata("Property", { name: "humidity", writeable: false })
  private _humidity: number = 0;

  @Reflect.metadata("Property", { name: "temperature", writeable: false })
  private _temperature: number = 10;
```

3. Define service commands. The SDK automatically calls the commands below when the device receives commands from the platform. The **name** corresponds to **command_name** of the product model, and **method** corresponds to the method for receiving the command. The input parameter and return value type of the command cannot be changed.

   The following code defines a ring alarm command named **ringAlarm**. The delivered parameter is **duration**, which indicates the duration of the ringing alarm.

```
@Reflect.metadata("DeviceCommand", {
  name: "ringAlarm",
  method: (paras: object): CommandRsp => {
    let duration: number = paras['duration'];
    LogUtil.log(TAG, `duration is ${duration}`);
        return IotResult.SUCCESS;
    }
})
private _alarm: Function = () => {};
```

4. Define the getter and setter methods.
   - The device automatically calls the getter method after receiving the commands for querying and reporting properties from the platform. The getter method reads device properties from the sensor in real time or from the local cache.
   - The device automatically calls the setter method after receiving the commands for setting properties from the platform. The setter method updates the local values of the device. If a property is not writable, leave the setter method not implemented.
   - Click **Generate** on the DevEco Studio and choose **Getter and Setter**, and then modify the method. The setter and getter interfaces are automatically generated.

```
public set smokeAlarm(value: number) {
  this._smokeAlarm = value;
  if (value == 0) {
    LogUtil.info(TAG, "alarm is cleared by app");
  }
}
```



```
public get smokeAlarm(): number {
  return this._smokeAlarm;
}

public set concentration(value: number) {
  // Read-only fields do not need to implement the set method.
}

public get concentration(): number {
  return Math.floor(Math.random() * 100);
}

public set humidity(value: number) {
  // Read-only fields do not need to implement the set method.
}

public get humidity(): number {
  return Math.floor(Math.random() * 100);
}

public set temperature(value: number) {
  // Read-only fields do not need to implement the set method.
}

public get temperature(): number {
  return Math.floor(Math.random() * 100);
}
```

5. Implement the constructor to initialize properties and commands.

```
constructor() {
  super();
  const fields = Object.getOwnPropertyNames(this);
  this.init(fields);
}
```

6. Create a device and register the smoke sensor service.

```
// Create a device service.
const smokeDetector = new SmokeDetector();
this.device.addService("smokeDetector", smokeDetector);
```

7. Enable periodic property reporting.

```
// Enable periodic property reporting.
this.device.getService("smokeDetector")?.enableAutoReport(10000);
```

8. Execute the preceding code to check the logs of reported properties.

```
com.huawe...t.device  I     AsyncMqtt Publish Start
com.huawe...t.device  D     AsyncMqtt Starting publish string
com.huawe...t.device  D     AsyncMqtt ParsePublishOptions Start
com.huawe...t.device  I     AsyncMqtt create promise
com.huawe...t.device  I     AsyncMqtt OnPublish Success
com.huawe...t.device  D     AsyncMqtt Starting - MqttPublish
com.huawe...t.device  D     AsyncMqtt Starting publish string
com.huawe...t.device  D     IoTDA_SDK# report properties success {"code":0,"message":"Publish Success"}
```

9. Check the value of the **alarm** property in the device shadow on the platform. The value is **1**. Change the value to **0** and check the device logs.

Figure 3-27 Device shadow - Viewing property (Alarm)

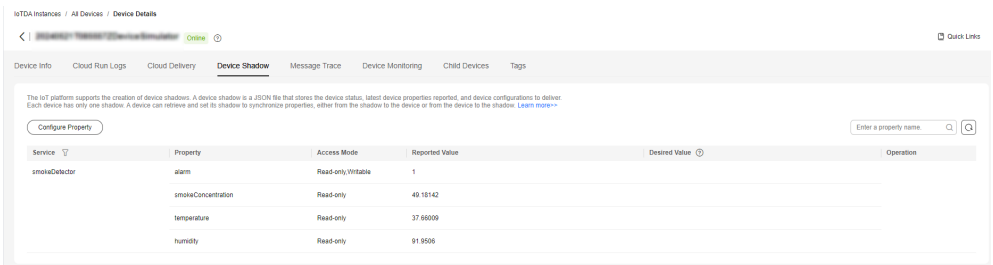


Figure 3-28 Logs for successful device property setting

10. Deliver the **ringAlarm** command on the platform. Check the device logs, which indicate that the **ringAlarm** command is called and a response is successfully reported.

Change History

Table 3-8 Change history

Version	Change	Description
0.0.1	New function	Added the capability of connecting to the Huawei Cloud IoT platform to facilitate service scenarios such as access, device management, and command delivery.

