Datacom Network Open Programmability
V100R020C00

Plug-in Package API Reference

Issue	02
Date	2020-12-30

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are trademarks 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 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.

Huawei Technologies Co., Ltd.

Address: Huawei Industrial Base
Bantian, Longgang
Shenzhen 518129
People's Republic of China

Website: https://www.huawei.com

Email: support@huawei.com

Contents

1 Programming Interface Overview

1.1 Interface Overview

This section describes the APIs provided by the open programmable framework.

Figure 1-1 API and SPI

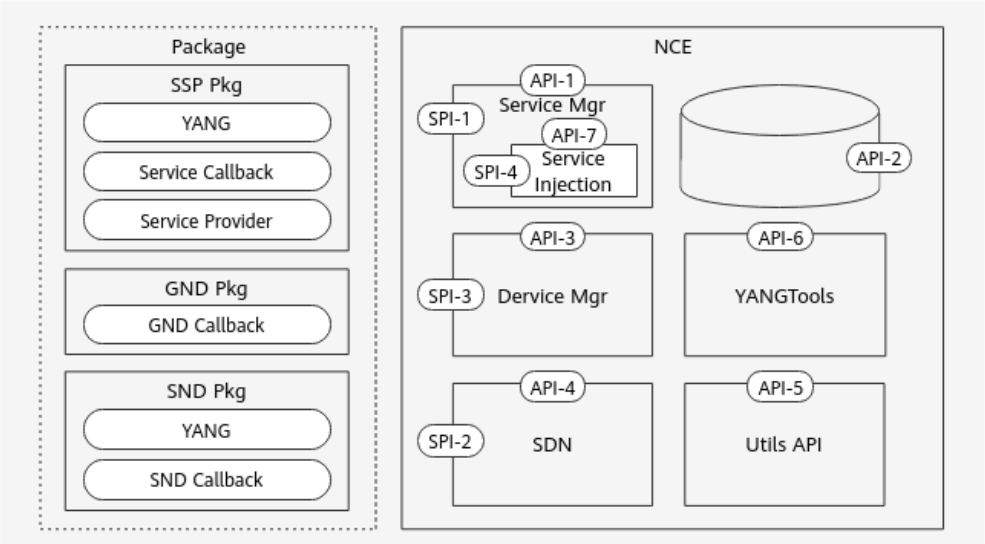


Table 1-1

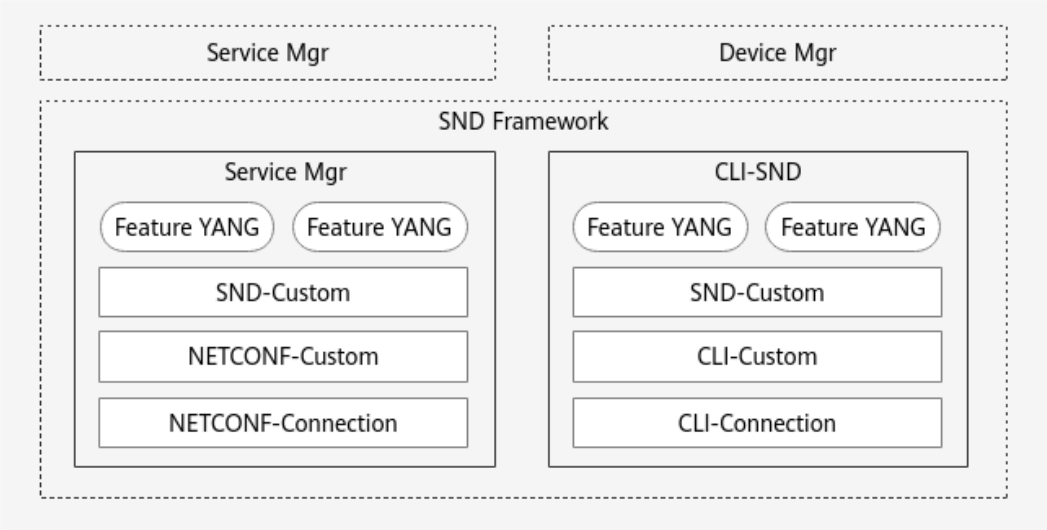
Interface Type	Interface Subtype	Scenario-based Capability
SPI	SPI-1: service function processing callback interface	Provides functions related to a certain type of services. These functions include service mapping, service O&M actions, and service restoration.
	SPI-2: SND function processing callback interface	Abstracts capabilities of vendors' devices and provides functions related to a type of devices. These functions include device function configuration, O&M, configuration consistency, and device connection configuration.
	SPI-3: GND function processing callback interface	Abstracts common device capabilities and provides the mapping between the model data of a certain type of device vendors and common model data. These models include interfaces, links, alarms, and service configurations.
	SPI-4: service injection function callback interface	Provides specific tool functions for the system and can be invoked by other SND, GND, or service functions.
API	API-1: configuration transaction interface	Provides configuration transaction interfaces, including enabling, editing, rolling back, and clearing transactions.
	API-2: data query interface	Provides model database query interfaces, including CDB and RDB query interfaces.
	API-3: basic device information query interface	Provides basic device information query interfaces, including querying device IDs based on device names and querying device MAC addresses based on device IDs.
	API-4: device data query interface and operation interface	Provides query and operation interfaces based on the device YANG model.
	API-5: template function interface	Provides template tool interfaces, such as IP address format conversion, mask address format conversion, and template attribute customization.
	API-6: model data encapsulation interface	Provides model operation interfaces, which are encapsulated into easy-to-use data obtaining interfaces.

Interface Type	Interface Subtype	Scenario-based Capability
	API-7: service injection interface	Provides service ingestion interfaces.

1.2 Interface Invoking Principle

1.2.1 SND Interface Invoking Principle

Figure 1-2 SND Interface Invoking Principle



The SND framework invokes the code of specific NE drivers (SNDs) to customize device functions. The customization includes the following types:

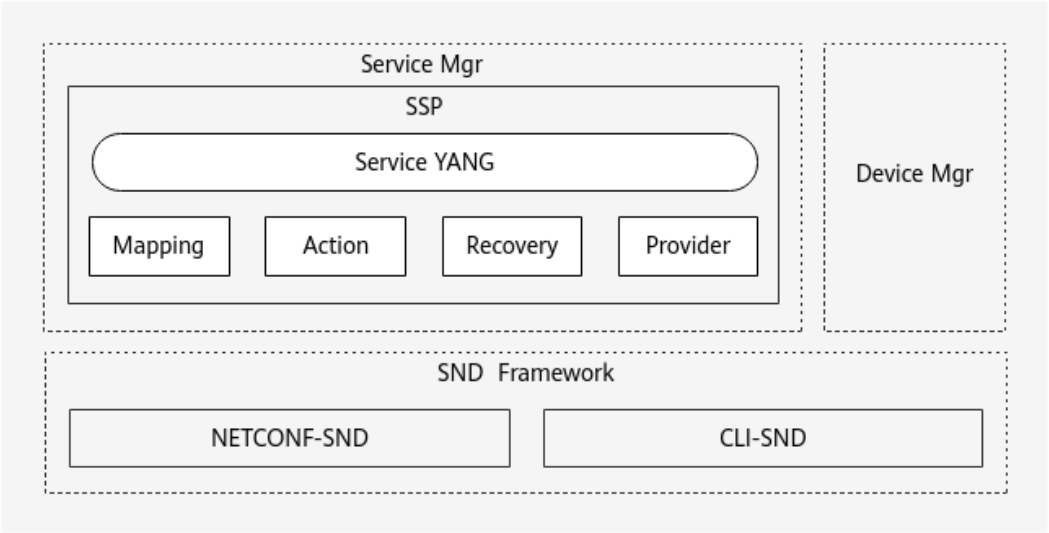
Standard and special SND processing customization: For example, customize the feature collection scope during data synchronization and other data irrelevant to protocols.

Protocol configuration customization: For example, customize standard and special protocol configuration items or the transaction interconnection mode.

Protocol connection mode customization: For example, customize the number of connected channels and protection mode.

1.2.2 SSP Interface Invoking Principle

Figure 1-3 SSP Interface Invoking Principle



The service management module invokes the SSP code to customize service functions. The customization is classified into the following types:

Service mapping customization: Customize the mapping logic to implement service processing capabilities such as service creation, deletion, and update.

Service action customization: Customize the action logic to implement service O&M capabilities.

Service restoration customization: Customize the restoration logic to restore services from devices.

Service provider customization: Customize the service provider logic to provide the tool capability for the system. The logic can be consistent with the system transaction.

2 API Reference

2.1 General APIs

This section describes common APIs, which can be called during the development of SSPs, GND packages, or SND packages.

2.1.1 Device Operation APIs

The APIs described in this section are used to interact with devices in real time.

2.1.1.1 Query Device Configurations

This API is used to query device configurations. The input parameters of this API do not contain southbound protocol information. The return values display device configuration data based on the device YANG model.

Type

API-4: device data query interface and operation interface

Typical Scenarios

When you compile an SSP, GND package, or SND package, some data needs to be obtained from devices in real time.

Functions

The query_data_from_device API provides data query in real time:

1. Obtains the YANG model based on the neid and generates southbound packets based on the input parameter path.
2. Delivers the southbound packets to the device and wait for the result returned by the device.

Constraints

1. Prerequisites: The device has gone online.
2. Precautions: If the device does not return the query result within 30 seconds, the query fails.
3. Usage restrictions: The constructed path must be the same as the path in the YANG model of the device.
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
def query_data_from_device(neid, storetype, path, properties={}, timeout=30000):
    """
    query_data_from_device is used to query device configuration.

    Input:
        string neid
        string storetype, value can be 'OPERATIONAL' or 'CONFIGURATION'.
        string path
        map<string, string> properties, default value is {}
        int timeout, default value is 30000 ms
    Output:
        string result
    """
```

Request Parameters

Table 2-1 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neid	Yes	STRING	-	-	Device ID, in UUID format.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
storetype	Yes	ENUM	-	-	The options are as follows: OPERATIONAL: used to query status data, corresponding to NETCONF get packets. CONFIGURATION: used to query configuration data, corresponding to NETCONF get-config packets.
path	Yes	STRING	-	-	Path of the YANG subtree corresponding to the data to be read from the device.
properties	No	DICT	-	dict{}	Additional field used for subtree filtering.
timeout	No	INT32	Larger than 0	30000	Timeout interval. The default value is 30000 ms.

Sample Request and Sample Response

Python invoking example:

```
# Import the AOC devicemgr.
from aoc import devicemgr

# The input parameters neid and path are transferred from Java or queried from Java through an interface.
neid = '868e778e-153c-3afe-a02c-89678e31e3e4'
path = '/huawei-ac-ne-snmp:snmp'

output = devicemgr.query_data_from_device(neid, 'CONFIGURATION', path)

# The returned result is similar to the following XML packet:
<snmp xmlns=\"urn:huawei:yang:huawei-ac-ne-snmp\">
   <mibViews>
      <mibView>
         <viewName>118</viewName>
         <subtree>iso</subtree>
         <type>excluded</type>
      </mibView>
   </mibViews>
</snmp>
```


Error Code

Error Code	Error Message	Solution
None	python invoke read fail!	Check whether the pkg.json file is correct.
None	input is null!	Check whether the pkg.json file is correct.

2.1.1.2 Perform an RPC Operation

RPC refers to the RPC capability based on the YANG standard on a device. NCE can deliver RPC packets through NETCONF.

Type

API-4: device data query interface and operation interface

Typical Scenarios

An RPC operation needs to be performed on a device.

Functions

The device_rpc API implements the real-time RPC function as follows:

1. Obtains the YANG model based on the neid and generates southbound packets based on the input parameter path.
2. Delivers the southbound packets to the device and wait for the result returned by the device.

Constraints

1. Prerequisites: The device has gone online.
2. Precautions: If the device does not return the query result within 30 seconds, the interface operation fails.
3. Usage restrictions: The rpcData and rpcName parameters must be consistent with those defined in the YANG model of the device.
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
def device_rpc(neid, rpcdata, rpcName, timeout=30000):
    """
    device_rpc is used to config the device.
    Input:
        string neid
        string rpcdata
        string rpcName
        int timeout, default value is 30000 ms
    Output:
```



```
    string result
"""
```

Request Parameters

Table 2-2 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neid	Yes	STRING	-	-	Device ID, in UUID format.
rpcdata	Yes	STRING	-	-	RPC data defined in the YANG model, in XML format.
rpcName	Yes	STRING	-	-	This parameter corresponds to QName of RPC.
timeout	No	INT32	Larger than 0	30000	Timeout interval. The default value is 30000 ms.

Sample Request and Sample Response

Python invoking example:

```
"""
Assume that the following RPC is defined on the device:
YANG Example:
    rpc activate-software-image {
        input {
            leaf image-name {
                type string;
            }
        }
        output {
            leaf status {
                type string;
            }
        }
    }
"""

# Import the AOC devicemgr.
neid = '868e778e-153c-3afe-a02c-89678e31e3e4'
path = '(urn:huawei:yang:huawei-aoc-rpc-test?revision=2018-01-01)activate-software-image'
rpcName = '''
<activate-software-image xmlns="urn:huawei:yang:huawei-aoc-rpc-test">
    <image-name>acmefw-2.3</image-name>
</activate-software-image>
'''
result = devicemgr.device_rpc(neid, rpcdata, rpcName)

# The returned result is similar to the following XML packet:
<activate-software-image xmlns="urn:huawei:yang:huawei-aoc-rpc-test">
```



```
  <status>The image acmefw-2.3 is being installed.</status>
</activate-software-image>
```

Error Code

Error Code	Error Message	Solution
N/A	python invoke read fail!	Check whether the pkg.json file is correct.
None	input is null!	Check whether the pkg.json file is correct.

2.1.1.3 Query Device Configuration or Running Data Through the CLI

This API is used to query device configuration or running data through the CLI on the Python side.

Type

API-4: device data query interface and operation interface

Typical Scenarios

The device data needs to be queried through the CLI.

Functions

The sendCli API on the Python service obtains the device configuration data or running data through the CLI. The processing is as follows:

1. The input parameters of the sendCli include neid, CLI command, and timeout interval.
2. Java invokes the CLI protocol to deliver CLI commands to devices.
3. The device returns the query result of the CLI commands.

Constraints

1. Prerequisites: The device has gone online.
2. Precautions: The default timeout period of the API is 60s.
3. Usage restrictions: The command lines must be supported by the current device.
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
def sendCli(neid, cliCommond, timeOut=60):
    """
sendCli is used to query devices'config or operational information.
    :param neid:str, id of device in aoc, default value is ''
    :param cliCommond:str, cli commond ''
```



```
:param timeOut:int timeout second"
:return response_body:aoc.sys.sys_model_pb2.sendCli_pb2 .SendCliOutput, contains cli response

"""
```

Request Parameters

Table 2-3 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neid	Yes	STRING	-	-	Device ID, in UUID format.
cliCommond	Yes	STRING	-	-	CLI commands delivered to devices. Format: STRING.
timeout	No	INT32	Larger than 0	60000	Timeout interval. The default value is 60000 ms.

Sample Request and Sample Response

Python invoking example:

```
# Import the package.
from aoc.sys import cliproxy

# Invoke the interface to obtain the response result.
response_data = cliproxy.sendCli("8a394835-cb84-38f3-44d5-36a7f2074a77","display this",120)
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.1.2 NCE Datastore APIs

This topic is closely related to two-phase transaction. In two-phase transaction, the configuration in the editing phase is written to the candidate database (CDB) of NCE. After the configuration is submitted, the configuration is transferred from the CDB to the running database (RDB).

2.1.2.1 Read the CDB

This API is used to read data in the CDB in a datastore based on the transaction ID and YANG subtree path.

Type

API-2: data query interface

Typical Scenarios

The service data that has been edited in the transaction needs to be queried during service configuration restoration.

Functions

Query the CDB data based on the transaction ID.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
def read_datastore_cdb(aoccontext, path, query_para=None):
    """
        read_datastore_cdb is a interface for reading cdb.
        :param aoccontext: AocContext, structure defined in aoc.base.aoccontext
        :param path: str, yang path, required field,
        :param query_para: dict type,example1: {"fields": fields=taskGroups1(nametaskGroups1), "depth": 1}
                           example2: {"where":
fields=room(name;taskGroups1(nametaskGroups1;nametaskGroups2);rack/server)}
                           example3: { "depth": 1}
        :return: str, configuration data string, defined in sys_model_pb2.datastore_pb2.ReadResult
    """
```

Request Parameters

Table 2-4 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
aoccontext	Yes	AocContext	-	-	transactionId is mandatory.
path	Yes	STRING	-	-	Subtree path for the YANG model of the device.
query_para	No	DICT	-	-	Filter query parameter.

Sample Request and Sample Response

Python invoking example:

```
# Import the AOC datastore.
from aoc import datastore

neid = '868e778e-153c-3afe-a02c-89678e31e3e4'
path = 'huawei-ac-nes:inventory-cfg/nes/ne/868e778e-153c-3afe-a02c-89678e31e3e4/huawei-ac-ne-snmp:snmp'
aoccontext = {
    'transactionId' = '148e31d3-79a5-4da2-94fd-893adc66080e'
}
output = datastore.read_datastore_ne_cdb(aoccontext, path, neid, {'where':'neId=a1a4ce9c-a24f-11ea-ab05-caf3e5f90b35'})

# The returned result is similar to the following XML packet:
<snmp xmlns=\"urn:huawei:yang:huawei-ac-ne-snmp\">
    <mibViews>
        <mibView>
            <viewName>118</viewName>
            <subtree>iso</subtree>
            <type>excluded</type>
        </mibView>
    </mibViews>
</snmp>
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.1.2.2 Read the RDB

This API is used to read data in the RDB in the datastore through the YANG subtree path.

Type

API-2: data query interface

Typical Scenarios

The submitted configuration data needs to be queried based on the YANG subtree during service configuration restoration.

Functions

Query data in the datastore RDB based on the YANG subtree path.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A

4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
def read_datastore_rdb(aoccontext, path, query_para=None):
    """
    read_datastore_rdb is a interface for reading rdb.

    :param aoccontext:AocContext, structure defined in aoc.base.aoccontext
    :param query_para: dict type,example1: {"fields": fields=taskGroups1(nametaskGroups1), "depth": 1}
                       example2: {"where":
fields=room(name;taskGroups1(nametaskGroups1;nametaskGroups2);rack/server)}
                       example3: { "depth": 1}
    :return document:str, configuration data string, defined in sys_model_pb2.datastore_pb2.ReadResult
    """
```

Request Parameters

Table 2-5 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
aoccontext	Yes	AocContext	-	-	-
path	Yes	STRING	-	-	Subtree path for the YANG model of the device.
query_para	No	DICT	-	-	Filter query parameter.

Sample Request and Sample Response

Python invoking example:

```
# Import the AOC datastore.
from aoc import datastore

path = 'urn:huawei:yang:huawei-ac-applications/bng_binding'
output = datastore.read_datastore_rdb(aoccontext, path)

# The returned result is similar to the following XML packet:
<bng_binding xmlns="https://example.com/example">
    <master_bng>master</master_bng>
    <slave_bng>slave</slave_bng>
    <master_info>
      <master_alias>master-alias</master_alias>
      <master_router_id>master-router-id</master_router_id>
    </master_info>
    <slave_info>
      <slave_alias>slave-alias</slave_alias>
      <slave_router_id>slave-router-id</slave_router_id>
    </slave_info>
 </bng_binding>
```


Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.1.2.3 Write the CDB

This API is used to write data into the datastore CDB through the transaction ID and YANG subtree path.

Type

API-2: data query interface

Typical Scenarios

The edited service data can be written into the CDB during service data restoration.

Functions

Write the edited service data into the datastore CDB based on the global transaction ID.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
def write_datastore(txid, content, path, source):
    """
    write_datastore is a interface for writing cdb in the datastore.
    Input:
        string txid, transactionId can not be None
        string content, required filed
        string path, required field
        string source, default value is None
    Output:
        WriteResult object, contains the following attributes:
            string result
    """
```


Request Parameters

Table 2-6 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
txid	Yes	STRING	-	-	transactionId is mandatory.
content	Yes	STRING	-	-	Content to be written into the datastore.
path	Yes	STRING	-	-	Subtree path for the YANG model of the device.
source	Yes	STRING	-	-	Data source. If the data source is not considered, the value is an empty string.

Sample Request and Sample Response

Python invoking example:

```
# Import the AOC datastore.
from aoc.sys import transaction
from aoc.sys import datastore

ne_id = '8d394835-cb84-38f3-a4d5-16a7f2074b40'
trans_id = transaction.create_transaction().transId
self.logger.info('create_transaction trans_id=%s' % trans_id)
path = '/huawei-ac-nes:inventory-cfg/nes/ne/%s/huawei-snmp:snmp' % ne_id
content = '<snmp xmlns="https://www.huawei.com/netconf/vrp/huawei-snmp">\n  <mibViews>\n
<mibView>\n       <viewName>testmibabc</viewName>\n       <subtree>iso</subtree>\n
<type>excluded</type>\n    </mibView>\n  </mibViews>\n</snmp>'

response = datastore.write_datastore(trans_id, content, path, '')
self.logger.info('write_datastore response=%s' % response)
transaction.commit_transaction(trans_id)
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.1.3 Device Information Query APIs

This topic describes APIs for querying device information.

2.1.3.1 Query Basic Device Information

This API is used to query the basic information about a device. Basic device information has been collected to NCE when the device is imported or goes online.

Type

API-3: basic device information query interface

Typical Scenarios

The basic information about a device needs to be queried.

Functions

The query_basic_data_from_db API queries basic device information in real time.

1. Queries basic device data based on the input parameters neid, nename, and nemanageip.
2. Supports cache acceleration if the input parameter contains neid or nename, improving the query performance. If the input parameter contains only nemanageip, cache acceleration is not supported.

Constraints

1. Prerequisites: N/A
2. Precautions: The three input parameters of this API are optional, but at least one is set. Different combinations of these parameters can be used to filter the devices (managed by NCE) that meet the conditions.
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
def query_basic_data_from_db(neid='', nename='', nemanageip=''):
    """
query_basic_data_from_db is used to query devices'detail information.
    Input:
        string neid, default value is ''
        string nename, default value is ''
        string nemanageip, default value is ''
    Output:
        QueryDevicesPagedResult object, contains the following object:
            queryDevicesPagedEntity object, contains the following attributes:
                neId, name, hardWare, softWare, manageIp, type, manufacturer,
                manageMac, deviceModel, esn, location, description, layer, get_channel_index,
                edit_channel_index
    """
```


Request Parameters

Table 2-7 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neid	No	STRING	N/A	-	Device ID, in UUID format.
nename	No	STRING	N/A	-	Name of a device.
nemanageip	No	STRING	N/A	-	Management interface IP address of a device.

Note: At least one of neid, nename, and nemanageip must be set.

Sample Request and Sample Response

Python invoking example:

```
# Import the AOC devicemgr.
from aoc import devicemgr

# The neid, nename, and nemanageip parameters can be combined for querying device information.
neid = '868e778e-153c-3afe-a02c-89678e31e3e4'
nename = 'HUAWEI_ROUTER'
nemanageip = '{ip:port}'

# The returned result is an object of the QueryDevicesPagedResult class.
result = devicemgr.query_basic_data_from_db(neid, nename, nemanageip)

 """
 The QueryDevicesPagedEntity class has the following members:
 # ME ID
 neId: "868e778e-153c-3afe-a02c-89678e31e3e4",

 # Hardware version number
 hardWare: "CX600-X1-M4",

 # Software version number
 softWare: "V800R012C10SPC680B680",

 # Management IP address
 manageIp: "{ip:port}",

 # Device type
 type: "ROUTER",

 # Device vendor
 manufacturer: "HUAWEI",

 # Device MAC address
 manageMac: "38:BA:16:58:1E:03",

 # Device model
 deviceModel: "CX600-X1-M4",
```



```
 # Device ESN.
 esn: "391091484237311",

 # Device location
 location: "Beijing China",

 # Description.
 description: "Huawei Versatile Routing Platform Software \\r VRP (R) software, Version 8.120 (CX600
 V800R012C10SPC680B680) \\r Copyright (C) 2012-2016 Huawei Technologies Co., Ltd. \\r",

 # Device level:
 layer: "Physical",

# NETCONF read channel ID
 get_channel_index: "b9b51c1d-38cd-35ed-9acd-2cb477762707",

# NETCONF write channel ID
 edit_channel_index: "a377dae5-6f88-3b15-a04d-56b9f769fdbb"
 """
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.1.3.2 Query a Device ID by Device Name

This API is used to query a device ID.

Type

API-3: basic device information query interface

Typical Scenarios

When a user compiles codes of an SSP, the device ID needs to be queried based on the device name.

Functions

The query_neid API implements the function of querying a device ID based on the device name.

Queries the device ID based on the input parameter nename.

Supports cache acceleration to improve query performance.

Constraints

1. Prerequisites: N/A
2. Precautions: The input parameter nename is the **Operation Name** configured when a device is added on the web UI.
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
def query_neid(nename):
    """
    query_neid is used to query neid by nename.

    Input:
        string nename
    Output:
        string neid
    """
```

Request Parameters

Table 2-8 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
nename	Yes	STRING	N/A	-	Name of a device.

Sample Request and Sample Response

Python invoking example:

```
# Import the AOC devicemgr.
from aoc import devicemgr

#Input parameter: device name
nename = 'HUAWEI_ROUTER'

neid = query_neid(nename)

# The returned result is a character string.
"868e778e-153c-3afe-a02c-89678e31e3e4"
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.1.3.3 Query the Device Status

This API is used to query the device status in real time.

Type

API-3: basic device information query interface

Typical Scenarios

When a user compiles codes of an SSP, the device status needs to be queried in real time based on the device ID, device name, and device IP address.

Functions

The query_status API provides device status query in real time.

Queries the real-time status of a device based on the input parameters neid, nename, and nemanageip.

Constraints

1. Prerequisites: N/A
2. Precautions: The three input parameters of this interface are optional, but at least one of them must be set. The devices managed by NCE that meet the search criteria are displayed.
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
def query_status(neid='', nename='', nemanageip=''):
    """
    query_status is used to query device status by neid, nename or nemanageip

    Input:
        string neid, default value is ''
        string nename, default value is ''
        string nemanageip, default value is ''
    Output:
        string status
    """
```

Request Parameters

Table 2-9 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neid	No	STRING	-	-	Device ID, in UUID format.
nename	No	STRING	N/A	-	Name of a device.
nemanageip	No	STRING	N/A	-	Management interface IP address of a device.

Note: At least one of neid, nename, and nemanageip must be set.

Sample Request and Sample Response

Python invoking example:

```
# Import the AOC devicemgr.
from aoc import devicemgr

# The neid, nename, and nemanageip parameters can be combined for querying the device status.
neid = '868e778e-153c-3afe-a02c-89678e31e3e4'
nename = 'HUAWEI_ROUTER'
nemanageip = '{ip:port}'

ne_status = query_status(nename)

# The returned result is a character string.
"OperateDown"
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.2 APIs for SSPs

This chapter describes the APIs used for developing SSPs.

2.2.1 Jinja Template

When mapping from the network layer to the NE layer is developed in the SSP package, the jinja template rendering capability needs to be used to generate NE configurations based on parameters.

2.2.1.1 Jinja Template Filter

For details about Jinja template development, see the official Jinja2 document. To facilitate coding, aoc_api provides some methods and filters that can be used in addition to Jinja filters.

Type

API-5: template function interface

Typical Scenarios

When a user uses rendered Jinjia templates to map data from the network layer to the NE layer, Jinjia filters need to be used for encoding.

Functions

The Jinja template provides a series of filters, including filters for setting global variables, converting NE IDs, and converting IP addresses.

Constraints

1. Prerequisites: N/A
2. Precautions: to_ne_id cannot be tested locally because this API is used to query device information from the AOC database.
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

GET_GLOBAL and SET_GLOBAL

Use the GET_GLOBAL and SET_GLOBAL methods to solve the problem that global variables need to be used in Jinja. The usage method is as follows:

```
<example>
# Initialize the temp global variables.
   {{SET_GLOBAL('temp', 0)}}
   {%- for name in class%}
      <name>{{name}}</name>
      {{GET_GLOBAL('temp') + loop.index0}}
      {%- if loop.lase %}
         {{SET_GLOBAL('temp',GET_GLOBAL('temp' + loop.index0)}}
      {%- endif%}
   {%- endfor%}
   {{GET_GLOBAL('temp')}}
</example>
```

NOTE

The code above is an example of using the GET_GLOBAL and SET_GLOBAL methods to access a for loop index.

Table 2-10 SET_GLOBAL input parameter

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
key	Yes	STRING	N/A	N/A	Key of a global variable.
value	Yes	Any type	N/A	N/A	Content of a global variable.

Table 2-11 GET_GLOBAL input parameter

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
key	Yes	STRING	N/A	N/A	Key of a global variable.

Table 2-12 GET_GLOBAL return value

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
value	Yes	Any type	N/A	N/A	Content of a global variable.

to_ne_id Filter

For details about the API for querying the device ID, see **2.1.3.2 Query a Device ID by Device Name**.

```
Input: {{HUAWEI_ROUTER| to_ne_id}}
Output: 868e778e-153c-3afe-a02c-89678e31e3e4
```

NOTE

The device name and device ID are set based on the actual situation.

Table 2-13 Input parameter to_ne_id

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
ne_name	Yes	STRING	N/A	N/A	Name of a device.

Table 2-14 Return value of to_ne_id

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
ne_id	Yes	STRING	UUID	N/A	Device ID.

ipaddr Filter

The ipaddr filter encapsulates some functions of the netaddr Python library.

- Calculate the IP address:

```
Input: {{'192.168.32.0/24' | ipaddr('0')}}
Output: 192.168.32.0/24
Input: {{'192.168.32.0/24' | ipaddr('1')}}
Output: 192.168.32.1/24
Input: {{'192.168.32.0/24' | ipaddr('-1')}}
Output: 192.168.32.255/24
Input: {{'192.168.32.0/24' | ipaddr('100')}}
Output: 192.168.32.100/24
```

- Calculate masks.

```
Input: {{'192.168.32.0/24' | ipaddr('netmask')}}
Output: 255.255.255.0
Input: {{'192.168.32.0/25' | ipaddr('netmask')}}
Output: 255.255.255.128
```

```
Input: {{'192.168.32.0/32' | ipaddr('netmask')}}
Output: 255.255.255.255
```

- Compute gateway addresses.

```
Input: {{'192.168.32.1/32' | ipaddr('network')}}
Output: 192.168.32.1
Input: {{'192.168.32.1/255.255.255.255' | ipaddr('network')}}
Output: 192.168.32.1
Input: {{'192.168.32.1/24' | ipaddr('network')}}
Output: 192.168.32.0
```

- Calculate the broadcast address.

```
Input: {{'192.168.32.1/24' | ipaddr('broadcast')}}
Output: 192.168.32.255
```

- Calculate the host mask:

```
Input: {{'192.168.32.1/24' | ipaddr('hostmask')}}
Output: 0.0.0.255
Calculate the broadcast address.
Input: {{'192.168.32.1/25' | ipaddr('hostmask')}}
Output: 0.0.0.127
```

2.2.1.2 Customized Deletion Through the no_delete Tag by Jinja Templates

When the action of deleting a service layer configuration is performed and submitted successfully, NCE deletes both the configuration data and data source by default. Jinja templates support two tags: no_delete and no_delete_nested, allowing you to delete only the data source but not the data:

no_delete: Only the data source of the node marked with this tag will be deleted, and the data of sub-nodes can be deleted.

no_delete_nested: indicates that only the data source of the node and subnodes marked with this tag will be deleted.

NOTE

- A data source indicates a reference to real data. For example, if the NE data interface1 is decomposed by service 1 and service 2, two references (data sources) exist. When service 1 is deleted, the interface still exists because the NE data interface1 is referenced by service 2.

Type

API-5: template function interface

Typical Scenarios

When a user map data from the network layer to the NE layer using rendered Jinja templates, the no_delete label in the template can be used to control the configuration deletion logic during service update.

Functions

Customize the configuration deletion logic during service update in a Jinja template.

Constraints

1. Prerequisites: N/A

2. Precautions: For details, see the "Invoking Method" section.
3. Usage restrictions: For details, see the "Invoking Method" section.
4. Relationships between APIs: N/A

Invoking Method

no_delete example:

```
.....
<ntpAuthKeyCfg xmlns:x="urn:huawei:ac" x:tag="no_delete">
.....
```

no_delete_nested example:

```
.....
<system  xmlns:ns0="urn:huawei:ac" ns0:tag="no_delete_nested">
  <inform-timeout>2</inform-timeout>
  <inform-resend-times>2</inform-resend-times>
  <inform-pend-number>111</inform-pend-number>
</system>
.....
```

NOTE

- For container nodes, only no_delete_nested is supported.
- no_delete or no_delete_nested can be nested within no_delete, but no_delete cannot be nested within no_delete_nested.
- neid does not support the no_delete or no_delete_nested tag.

2.2.1.3 Merge, Delete, or Replace Operations by Jinja Templates

Users can customize the deletion and modification operations of NE data in Jinja template.

Type

API-5: template function interface

Typical Scenarios

When a user develops mapping from the network layer to the NE layer using rendered Jinja templates, the operation code label in the Jinja templates is used to directly operate NE data without comparing differences.

Functions

Provides the capability of customizing the logic for deleting and modifying NEs using Jinja templates.

Constraints

1. Prerequisites: N/A
2. Precautions: For details, see the "Invoking Method" section.
3. Usage restrictions: For details, see the "Invoking Method" section.

4. Relationships between APIs: N/A

Invoking Method

merge: Merge with a node if it exists. Otherwise, create the node. The merge operation is the default operation of SSPs.

Example:

```
...
<inform-timeout xmlns:ns0="urn:huawei:ac" ns0:operation="merge">2</inform-timeout>
...
```

delete: Delete an existing node. No error is raised if the node does not exist. If the delete operation is performed on a container node, the following operations cannot be performed on subnodes of the container node, including the merge, replace, no-delete, and no-delete-nested operations. The delete operation cannot be performed on key nodes of instances in the list.

Example:

```
...
<inform-timeout xmlns:ns0="urn:huawei:ac" ns0:operation="delete">2</inform-timeout>
...
```

replace: Replace a node if it exists; otherwise, create the node. If the replace operation is performed on a container node, the merge or replace operation cannot be performed on its subnodes. The replace operation cannot be performed on key nodes of instances in the list or leaflist nodes.

Example:

```
...
<inform-timeout xmlns:ns0="urn:huawei:ac" s0:operation="replace">2</inform-timeout>
...
```

2.2.1.4 Ordering by Jinja Templates

Users can customize the order-by-user attribute in Jinja templates to adjust the sequence of NE data.

Type

API-5: template function interface

Typical Scenarios

When a user develops mapping from the network layer to the NE layer using the jinja template rendering engine, the order-by-user attribute needs to be used to adjust the sequence of different instances.

Functions

Provides the capability of adjusting the sequence of NE data instances using Jinja templates.

Constraints

1. Prerequisites: N/A
2. Precautions: order can be used only in the schema node that declares order-by user.
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

For the list or leaflist nodes marked with the ordered-by user attribute (see RFC7950-YANG1.1-7.7.1), the insert operation can be used to adjust the data sequence in the SSP package.

Example:

```
......
<nacm xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-acm">
<rule-list xmlns:ns0="urn:ietf:params:xml:ns:yang:1" ns0:insert="after" ns0:key="[name='name2']">
      <name>name1</name>
   </rule-list>
</nacm>
......
```

NOTE

If a rule-list node named name2 already exists, a rule-list node named name1 is merged and placed after name2.

2.2.2 Two-Phase Transaction Configuration

Python supports two-phase transaction configuration APIs for two-phase operations, including creating, editing, committing, and deleting a transaction.

Type

API-1: configuration transaction interface

Typical Scenarios

Two-phase transaction operations can be performed during service RPC operations.

Functions

Perform two-phase transaction operations.

Constraints

1. Prerequisites: The device has gone online.
2. Precautions: If a transaction fails to be submitted, the transaction release interface needs to be invoked explicitly.
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
# Create
def create_transaction():
    """
    create_transaction is used to create transaction.

    Input:
        None
    Output:
        NcsServiceConfigCreateTransOutput object, contains the following attributes:
            string transId
    """


# Edit
def edit_transaction(transid, data, path, action, neid=''):
    """
    edit_transaction is used to edit transaction.

    Input:
        string transid
        string data
        string path
        string neid
        string action, values can be create,delete,remove,merge,replace
    Output:
        NcsServiceConfigEditOutput object, contains the following attributes:
            boolean result
            string errorCode
            string errorMsg
    """


# Submit
def commit_transaction(transid):
    """
    commit_transaction is used to commit transaction.

    Input:
        string transid
    Output:
        NcsServiceConfigCommitTransOutput object, contains the following attributes:
            boolean result
            string errorCode
            string errorMsg
    """


# Delete
def delete_transaction(transid):
    """
    delete_transaction is used to reset transaction.

    Input:
        string transid
    Output:
        NcsServiceConfigResetTransOutput object, contains the following attributes:
            boolean result
            string errorCode
            string errorMsg
    """
```


Request Parameters

Table 2-15 Return value of the two-phase transaction creation API

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transId	Yes	STRING	N/A	N/A	Transaction ID, in UUID format.

Table 2-16 Input parameters of the two-phase transaction editing interface

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transid	No	STRING	N/A	N/A	Transaction ID returned by the create interface, in UUID format. If this parameter is not set, the transaction ID is delivered in one phase. The value is in UUID format.
data	Yes	STRING	N/A	N/A	Configuration packets, in XML format.
path	Yes	STRING	N/A	N/A	Path of the YANG subtree.
action	Yes	STRING	create, merge, delete, remove, replace	N/A	Operation type.
neid	No	STRING	N/A	N/A	NE ID, in UUID format. In the case of single-station configuration, you need to specify the device ID. In the case of network configuration, you do not need to specify the device ID. The value is in UUID format.

Table 2-17 Return values of the two-phase transaction editing interface

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
result	Yes	BOOL	true, false	false	Editing result.
errorCode	No	STRING	N/A	N/A	Error code reported during editing.
errorMsg	No	STRING	N/A	N/A	Error message displayed during editing.

Table 2-18 Input parameters of the two-phase transaction submission interface

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transId	Yes	STRING	N/A	N/A	Transaction ID, in UUID format.
onlyService	No	bool	true, false	false	Whether only the network layer takes effect. The value can be true or false.

Table 2-19 Return value of the two-phase transaction submission interface

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
result	Yes	BOOL	true, false	false	Submission result. The value can be true or false.
errorCode	No	STRING	N/A	N/A	Error code reported during submission.
errorMsg	No	STRING	N/A	N/A	Error message displayed during submission.

Table 2-20 Input parameters of the two-phase transaction rollback interface

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transId	Yes	STRING	N/A	N/A	Transaction ID, in UUID format.

Table 2-21 Return value of the two-phase transaction rollback interface

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
result	Yes	BOOL	true, false	false	Rollback result. The value can be true or false.
errorCode	No	STRING	N/A	N/A	Error code reported during rollback.
errorMsg	No	STRING	N/A	N/A	Error message displayed during rollback.

Sample Request and Sample Response

Python invoking example:

```
transid= create_transaction()
# Return the transaction ID.
neid = '868e778e-153c-3afe-a02c-89678e31e3e4'
path = 'huawei-ac-nes:inventory-cfg/nes/ne/868e778e-153c-3afe-a02c-89678e31e3e4/huawei-ac-ne-snmp:snmp'
date ='''
<snmp xmlns=\"urn:huawei:yang:huawei-ac-ne-snmp\">
 <mibViews>
  <mibView>
   <viewName>118</viewName>
   <subtree>iso</subtree>
   <type>excluded</type>
  </mibView>
 </mibViews>
</snmp>
'''

# Invoke the editing interface.
edit_transaction(transid, data, path, 'create',neid)

# Submit the transction.
commit_transaction(transid)
```


Error Code

Error Code	Error Message	Solution
publicinfocode.fp.config.error.0x00c80022	Device {} reports an error. The error information is {}.	Enter correct parameters based on the error information and deliver the configuration again.

2.2.3 Service Mapping

When the Python code is developed for a service package, ncsservice.NcsService must be inherited and the ncs_map method must be rewritten.

Type

SPI-1: service mapping interface

Typical Scenarios

A Python script needs to be developed for a service package. The script maps network service data to NE service data.

Functions

Complete service programming from the service layer to the NE layer.

Constraints

1. Prerequisites: The NE exists and has gone online.
2. Precautions: You need to encode the special characters <, >, and &.
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

- Python-based interface invoking method:

ncsservice.NcsService needs to be inherited and the ncs_map method needs to be implemented. During service decomposition, the framework invokes the ncs_map method and transfers service layer data to the ncs_map method based on the MapRequest input parameter.

```
class MapRequest(object):
    def __init__(self, xml, context):
        self.xml = xml
        self.context = context
        self.__xmldict = None
        self.__xmldictnode = None
```


NOTE

MapRequest contains four member variables:

xml: configuration packets of the str type at the service layer, in XML format

xmldict: result of converting XML files through xmltodict

xmldictnode: result of converting xmldict to dictnode. This type is used more often.

context: dictionary in the Python language, which contains the variables required by the mapping logic

Request Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
xml	Yes	STRING	N/A	N/A	Configuration packets of the str type at the service layer, in XML format.
xmldict	Yes	Dictionary	N/A	N/A	Result of converting XML files through xmltodict.
xmldictnode	Yes	DictNode	N/A	N/A	Result of converting xmldict to dictnode. This type is used more often.
context	Yes	Dictionary	N/A	N/A	Dictionary in the Python language, which contains the variables required by the mapping logic.

Sample Request and Sample Response

Python invoking example:

```
from aoc import NcsService

class AocNcsexample(NcsService):
    def ncs_map(self, request):
        return self.render('example/template_bng.j2', request.xmldictnode)
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

Supplementary: Complex Model Operations

Developing a service package needs to inherit the ncs_map method of the NcsService class. The main logic is to process the service model. NcsService

introduces the required third-party libraries such as jinja2, protobuf, and xmltodict. In addition, NcsService provides auxiliary classes such as DictNode and NoLoopStr, facilitating the development of SSP scripts. This section describes how to use these tools to perform model operations.

Initialize the dictnode and simulate the ncs_map input:

```
xml = '''
   <test>
      <keyName>test</keyName>
      <value>hello world</value>
   </test>
   '''

dictnode = DictNode(xmltodict.parse(self.xml, encoding='utf-8'))
```

- Use the DictNode.member symbol to access the value in DictNode.

  ```
  print(dictnode.test.value) # Use '.' to access the value in the model.
  # hello world
  ```

- Use the for method to traverse a leaf node in DictNode. The traversal is not performed in alphabetical order.

  ```
  i = 0
  for val in dictnode.test.keyName:
      i +=1;
  self.assertEqual(i, 1)
  ```

- Use the for method to traverse a map point in DictNode. The traversal is performed only once.

  ```
  for node in dictnode.test:
      print(node)
  # DictNode({u'keyName': u'test', u'value': u'hello world'})
  ```

- Use the for method to traverse the list in DictNode. The traversal is performed based on each element in the list, which is the same as using the for method to traverse the list.
- If you use the for method to traverse a node that does not exist, no error is reported and no loop occurs.

  ```
  for node in dictnode.noexist:
      print(node)
  # If noexist does not exist under dictnode, print will not be executed.
  ```

- If the if method is used to determine a node that does not exist, the value False is returned.

  ```
  if dictnode.noexist:
       print(dictnode.noexist)
  # If noexist does not exist under dictnode, print will not be executed.
  ```

- If a nonexistent child node is directly referenced, no error is reported.

  ```
  print(dictnode.noexist)
  # dictnode.noexit will return an empty character string.
  ```

- In the YANG model, a schema node is defined as a list. In actual configuration, only one leaf node is configured in the list or no node is configured, and the preceding tips need to be used.
- Use the key() method to obtain the items in the list in the YANG model.

  ```
  nes = [
          {
              'pairName': 'pairName1',
              'hostname': 'hostname1'
          },

          {
              'pairName': 'pairName2',
              'hostname': 'hostname2'
          },
  ```



```
    {
        'pairName': 'pairName2',
        'hostname': 'hostname3'
    },

    {
        'pairName': 'pairName1',
        'hostname': 'hostname4'
    }
]

ne_map = DictNode()
ne_result = ListNode()
for ne in nes:
    ne_dictnode = DictNode(ne)
    ne_map.put(ne['hostname'], ne_dictnode)
    ne_result.append(ne_dictnode)

ne_pare = DictNode()
for hostname, ne in ne_map.items():
    if ne_pare.has_key(ne.pairName):
        pare = ne_pare.get(ne.pairName)
        ne.put('ne_peer', pare)
        pare.put('ne_peer', ne)
    else:
        ne_pare.put(ne.pairName, ne)

    # Use the key method to filter the element of hostname = 'hostname1'.
self.assertEqual(ne_result.key(hostname = 'hostname1').ne_peer.hostname, 'hostname4')
```

2.2.4 Service RPC

When the Python code is developed for a service package, ncsservice.NcsService must be inherited and the ncs_rpc method must be rewritten.

Type

SPI-1: service RPC mapping interface

Typical Scenarios

Some services are non-configuration services. RPC scripts need to be developed to implement certain functions.

Functions

Inherit ncsservice.NcsService and rewrite the ncs_rpc method to complete service programming from the service layer to the NE layer.

Constraints

1. Prerequisites: The NE exists and has gone online.
2. Precautions: You need to encode the special characters <, >, and &.
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

ncsservice.NcsService needs to be inherited and the ncs_rpc method needs to be implemented. During service decomposition, the framework invokes the ncs_rpc method and transfers service layer data to the ncs_rpc method based on the RpcRequest input parameter.

```
class RpcRequest(object):
    def __init__(self, xml, context):
        self.xml = xml
        self.context = context
        self.__xmldict = None
        self.__xmldictnode = None
```

NOTE

RpcRequest contains four member variables:

xml: configuration packets of the str type at the service layer, in XML format

xmldict: result of converting XML files through xmltodict

xmldictnode: result of converting xmldict to dictnode. This type is used more often.

context: dictionary in the Python language, which contains the variables required by the mapping logic

Request Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
xml	Yes	STRING	N/A	N/A	Configuration packets of the str type at the service layer, in XML format.
xmldict	Yes	Dictionary	N/A	N/A	Result of converting XML files through xmltodict.
xmldictnode	Yes	DictNode	N/A	N/A	Result of converting xmldict to dictnode. This type is used more often.
context	Yes	Dictionary	N/A	N/A	Dictionary in the Python language, which contains the variables required by the mapping logic.

Sample Request and Sample Response

Python invoking example:

ncs_rpc example:

```
from aoc import NcsService

class AocNcsDemoRpcService(NcsService):
    def ncs_rpc(self, request, arg1, arg2):
        return self.render('example/template_bng.j2', request.xmldictnode)
```


Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.2.5 SSP Supporting Service-Inject

During the compilation of SSP packages, external interfaces need to be invoked sometimes. Service-inject provides a method of invoking external interfaces.

Type

SPI-4: service injection interface

Typical Scenarios

When a user develops mapping from the network layer to the NE layer using the jinja template rendering engine, some interfaces or capabilities not built in the system may be used, for example, external resource management. The external interfaces or capabilities need to be placed in the system through service injection for SSP scripts to invoke.

Functions

Provides a set of capabilities for service providers to inject and service users to use.

Constraints

1. Prerequisites: N/A
2. Precautions: For details, see the following description.
3. Usage restrictions: For details, see the following description.
4. Relationships between APIs: N/A

run/revert/action Method

The service provider needs to inherit the ServiceProvider API and implement the run/revert/action method as required. The following is an example:

```
import netaddr
import random
from aoc.sys.inject.serviceprovider import ServiceProvider

class IpPool(ServiceProvider):
    ip_pool = set()
    def run(self, input):
        self.log.info('run %s' % input)
        output = {}
        if input.get("operation") == "apply":
            begin = netaddr.IPNetwork('{ip:port}').value
            end = netaddr.IPNetwork('{ip:port}').value
            value = random.randint(begin, end)
            output.setdefault("result", value)
            self.ip_pool.add(str(value))
            self.log.info('apply %s' % value)
```



```
    self.log.info('ip_pool %s' % self.ip_pool)
    return output


def revert(self, input):
    self.log.info('revert %s' % input)
    value = input.get("result")
    self.ip_pool.discard(value)
    output = {}
    output.setdefault("result", "success")
    self.log.info('ip_pool %s' % self.ip_pool)
    return output


def action(self, input):
    self.log.info('action %s' % input)
    if input.get("operation")  == "create":
        request = input.get("request")
        self.log.info('%s' % request.get("url"))
        self.log.info('%s' % request.get("begin"))
        self.log.info('%s' % request.get("end"))
    output = {}
    output.setdefault("result", "success")
    self.log.info('output %s' % output)
    return output
```

NOTE

- The input and output of the run/revert/action method must be the dict type.
- The run method is stateful. When the precommit method fails, the open programmability framework invokes the revert method.
- The input of the run method needs to be provided to users, and the output of the run method can be used as the input of the revert method.
- The action can be performed repeatedly without reporting an error.
- hook-type in **pkg.json** must be set to serviceprovider, and hook-key must be provided to users.

Table 2-22 RUN input parameter

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
input	Yes	STRING	N/A	N/A	JSON packet.

Table 2-23 RUN return value

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
output	Yes	STRING	N/A	N/A	JSON packet.

Table 2-24 REVERT input parameter.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
input	Yes	STRING	N/A	N/A	JSON packet.

Table 2-25 Return value of REVERT

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
output	Yes	STRING	N/A	N/A	JSON packet.

Table 2-26 Action input parameter

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
input	Yes	STRING	N/A	N/A	JSON packet.

Table 2-27 ACTION return value

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
output	Yes	STRING	N/A	N/A	JSON packet.

Example of the Service Invoker Code

```
import netaddr
from aoc.ncs.ncsservice import NcsService
from aoc.sys import datastore
import traceback

class ServiceInjectConsumer(NcsService):
    service_name = 'ippool'
    def gen_request(self):
        begin = netaddr.IPNetwork('{ip:port}').value
        end = netaddr.IPNetwork('{ip:port}').value
        url = "https://aoc.serviceinject.test"
        request = {}
        request.setdefault("url", url)
        request.setdefault("begin", str(begin))
        request.setdefault("end", str(end))

        return request
```



```
def apply_pool(self):
    input = {}
    input.setdefault("operation", "create")
    input.setdefault("request", self.gen_request())
    self.serviceinject.action(self.service_name, input)


def apply_ip(self, key, aoccontext):
    input = {}
    input.setdefault("operation", "apply")
    input.setdefault("request", self.gen_request())
    result = self.serviceinject.run(self.service_name, key, input, aoccontext)
    ip_addr = result.get("result")
    return ip_addr


def ncs_map(self, request, aoccontext=None, template=None):
    self.logger.info('%s' % request)
    try:
        ncs_context = dict(request.context)
        self.logger.info('%s' % ncs_context)

        if "ip_pool" not in ncs_context.keys():
            self.apply_pool();
            ncs_context.setdefault('ip_pool', 'created')

        if "vlan1" not in ncs_context.keys():
            result = self.apply_ip("vlan1", aoccontext);
            ncs_context.setdefault('vlan1', result)

    except Exception as e:
        self.logger.info("%s" % traceback.format_exc())

    return "<inventory-cfg xmlns=\"urn:huawei:yang:huawei-ac-nes\"><nes></nes></inventory-cfg>",
ncs_context
```

NOTE

- Invoke the action method of the service provider through self.serviceinject.action(self.service_name, input). service_name is the hook-key of the service provider. The input needs to be constructed according to the service provider.
- Invoke the run method of the service provider through self.serviceinject.run(self.service_name, key, input, aoccontext). The key is a keyword customized by the user to distinguish other invoking.
- Use ncs_context.setdefault('vlan1', result) to set the context. In the next execution, the open programmable framework carries the context submitted last time through request.context.

Table 2-28 RUN input parameter

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
input	Yes	STRING	N/A	N/A	JSON packet.

Table 2-29 RUN return value

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
output	Yes	STRING	N/A	N/A	JSON packet.

Table 2-30 Action input parameter

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
input	Yes	STRING	N/A	N/A	JSON packet.

Table 2-31 ACTION return value

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
output	Yes	STRING	N/A	N/A	JSON packet.

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3 APIs for SND Packages

SND packages require APIs for device management, configuration management, data consistency, SND CLI framework, and SND RESTCONF framework.

2.3.1 Device Management Customization

This section describes device management APIs.

2.3.1.1 Device sysoid Registration

This API is used to set the sysoid for a device to manage this device model.

Type

API-4: device data query interface and operation interface

Typical Scenarios

The sysoid information identifies the device type, model, and vendor. Before a device is managed, the sysoid information of the device needs to be registered with NCE.

Functions

Set the sysoid for a device to manage this device model.

This interface is optional. If the sysoid field is not carried when a device is added, the triplet in **pkg.json** is used to match the device. If the sysoid field is carried when a device is added, this interface must be overwritten to accurately match the device. If the interface is not overwritten, the device fails to be added.

1. Before managing an NE, you need to register the device type, vendor, and SysOid of the NE.

2. The device type and vendor must be registered.

3. The SysOid can be registered. If the SysOid needs to be verified during management, the SysOid must be available during registration.

4. The registration method is to rewrite this API.

5. Additional description: During device management, the system compares the type, vendor, and SysOid (if registered) entered by the user. The device can be managed only when these parameters are matched. Otherwise, the *** error is reported.

Constraints

1. Prerequisites: N/A
2. Precautions: If the sysoid field is carried when a device is added, this API must be compiled.
3. Usage restrictions: If no sysoid field is carried when a device is added, the triplet in pkg.json is used for matching. If the sysoid field is carried when a device is added, this API needs to be written.
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
def getSysoidInfo(self, aoccontext, request=None):
    """
    Register the device sysoid for a device to manage this device model.
    This interface is not mandatory. If the sysoid field is not carried when a device is added, the triplet in
pkg.json is used to match the device.
    If the sysoid field is carried when a device is added, this interface must be overwritten to accurately
match the device. If the interface is not overwritten, the device fails to be added.
    :param aoccontext: context
    :param request: parameter carried in the method
    :return: device information
    """
    pass
```


Request Parameters

Table 2-32 List of input parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
aoccontext	Yes	REFERENCE	For details, see the AocContext table.	N/A	Context information.

Table 2-33 AocContext

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transactionId	No	STRING	N/A	N/A	Transaction ID, which is used to ensure data consistency and enable the transaction mechanism.
deviceVendor	Yes	STRING	N/A	N/A	Device vendor.
deviceType	Yes	STRING	N/A	N/A	Device type.
deviceVersion	Yes	STRING	N/A	N/A	Device software version number.

Table 2-34 SysoidInfoProtos

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
SysoidEntity	No	ARRAY_REFERENCE	For details, see the SysoidEntity table.	N/A	Device sysoid information.

Table 2-35 SysoidEntity

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
sysoid	No	STRING	N/A	N/A	Sysoid value of a device.
deviceType	Yes	STRING	N/A	N/A	Device type.
deviceModel	Yes	STRING	N/A	N/A	Device model.
deviceVendor	Yes	STRING	N/A	N/A	Device vendor.

Sample Request and Sample Response

Python invoking example:

```
def getSysoidInfo(self, aoccontext, request=None):
    sysoidInfo = SysoidInfo()
    sysoidEntity = sysoidInfo.sysoidEntity.add()

    # sysoid value of a device
    sysoidEntity.sysoid = "1.3.6.1.4.1.2011.2.62.2.18"

    # Device type
    sysoidEntity.deviceType = "ROUTER"

    # Device model
    sysoidEntity.deviceModel = "NE40E-X8A"

    # Device vendor
    sysoidEntity.deviceVendor = "HUAWEI"
    return sysoidInfo
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.1.2 Device Connection Customization

Device connection modes and capabilities need to be customized.

Type

API-4: device data query interface and operation interface

Typical Scenarios

Device connection capabilities need to be customized.

Functions

Customize the device connection mode and connection capabilities.

Constraints

1. Prerequisites: N/A
2. Precautions: This API does not need to be overridden and has been implemented in the parent class by default. If you need to customize the connection mode and capability of a device, refer to the default implementation mode.
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
def getConnectInfo(self, aoccontext, request=None):
    """
    Customize the device connection mode and capability.
    This interface does not need to be overrode and has been implemented in the parent class by default. If you need to customize the connection mode and capability of a device,
    refer to the default implementation mode.
    :param aoccontext: context
    :param request: parameter carried in the method
    :return: connection instance
    """
```

Request Parameters

Table 2-36 List of input parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
aoccontext	Yes	REFERENCE	For details, see the AocContext table.	N/A	Context information.

Table 2-37 AocContext

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transactionId	No	STRING	N/A	N/A	Transaction ID.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
deviceVendor	Yes	STRING	N/A	N/A	Device vendor.
deviceType	Yes	STRING	N/A	N/A	Device type.
deviceVersion	Yes	STRING	N/A	N/A	Device software version number.

Table 2-38 ConnectInfos

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
connectInfo	No	ARRAY_REFERENCE	For details, see the ConnectInfo table.	N/A	Device connection configuration information.

Table 2-39 ConnectInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
protocolEntity	No	ARRAY_REFERENCE	For details, see the ProtocolEntity table.	N/A	Device connection protocol.
connectEntity	Yes	ARRAY_REFERENCE	For details, see the ConnectEntity table.	N/A	Device type.

Table 2-40 ProtocolEntity

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
protocolType	No	ENUM	NETCONF,CLI	N/A	Protocol type, which can be NETCONF or CLI.
helloEntity	Yes	REFERENCE	For details, see the HelloEntity table.	N/A	Hello packet information.

Table 2-41 HelloEntity

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
helloType	No	ENUM	stardardType, extendType, defaultType	defaultType	Hello type. The options are as follows: stardardType: standard packets, which are used for YANG interconnection. extendType: extended packets, which carry the extended capability set and are used for YANG interconnection. defaultType: default packets, which are used for schema interconnection.
extendEntity	Yes	ARRAY_STRING	N/A	N/A	If the packet type is extendType, the extended capability set needs to be carried.

Table 2-42 ConnectEntity

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
methodType	Yes	STRING	Java	N/A	Encoding language of the connection interface.
agentImpl	Yes	STRING	N/A	N/A	Connection interface implementation class.
priority	Yes	STRING	N/A	N/A	Connection invoking priority.

Sample Request and Sample Response

Python invoking example:

```
def getConnectInfo(self, aoccontext, request=None):
    connectInfos = ConnectInfos()
    connectInfo = connectInfos.connectInfo.add()

    # Set the protocol type to NETCONF.
    connectInfo.protocolEntity.protocolType = ProtocolEntity.netconf

    """
    Set connectPolicy to DEFAULT_CONNECT, indicating that the default agent is used.
    The NETCONF protocol system provides NeAgent by default, and the CLI protocol system provides CliAgent by default.
    If connectPolicy is set to COMPATIBLE_CONNECT, the value of type in hooks of pkg.json is snd
    and the value of key is agent in hook of ecs-connect-agent. The agent can be the built-in agent of the system or customized.
    If there is no hook, add it.
    """
    connectInfo.connectPolicy = DEFAULT_CONNECT

    # Set readChannel to SINGLE_CHANNEL, indicating that the read channel is a single channel.
    connectInfo.channelInfo.writeChannel = PROTECTED_MODE

    # Set writeChannel to PROTECTED_MODE, indicating that the write channel is protected in active/standby mode.
    connectInfo.channelInfo.readChannel = SINGLE_CHANNEL

    """
    If is_read_share_write channel is set to false, the read and write channels are not shared. If this parameter is set to true, the read and write channels are shared.
    In this case, you do not need to set readChannel.
    """
    connectInfo.channelInfo.is_read_share_write = False

    """
    Set the packet type.
    defaultType: Establish a NETCONF channel for schema interconnection and leave the capability set empty.
    standardType: Establish a NETCONF channel for the standard YANG capability set and leave the capability set empty.
    extendType: To establish a NETCONF channel for the extended YANG capability set, add a capability set list.
```



```
    """
    connectInfo.protocolEntity.helloEntity.helloType = HelloEntity.defaultType

    # If the channel type is HelloType.extendType, the extended capability set must be carried.
    # helloEntityNewBuilder.addExtendEntity("http://www.huawei.com/netconf/capability/execute-cli/1.0");
    # helloEntityNewBuilder.addExtendEntity("http://www.huawei.com/netconf/capability/discard-commit/
1.0");

    """
    Set this protocol as the primary protocol. In the dual-protocol scenario, you need to add an auxiliary
protocol.
    For details, see the getConnectInfo method of the dual-protocol parent class NetconfCliSnd.
    """
    connectInfo.connectionPriority = PRIMARY_CONNECTION
    return connectInfos
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.2 Configuration Management Customization

Device configurations can be customized.

2.3.2.1 Device Driver Data Configuration

Type

API-4: device data query interface and operation interface

Typical Scenarios

Driver information needs to be configured for a device to communicate with NCE.

Functions

Configure device driver data.

Constraints

1. Prerequisites: The device has gone online.
2. Precautions: The device has gone online.
3. Usage restrictions: The driver parameter settings must be the same as those of the actual device.
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
# Customize a universal driver for a device.
def getCommonDriverInfo(self, aoccontext, request=None):
    pass
```

```
# Customize the NETCONF driver.
def getNetconfDriverInfo(self, aoccontext, request=None):
    pass
```

Request Parameters

Table 2-43 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
aoccontext	Yes	REFERENCE	For details, see the AocContext table.	N/A	Context information.

Table 2-44 AocContext

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transactionId	No	STRING	N/A	N/A	Transaction ID.
deviceVendor	Yes	STRING	N/A	N/A	Device vendor.
deviceType	Yes	STRING	N/A	N/A	Device type.
deviceVersio	Yes	STRING	N/A	N/A	Device software version number.

Table 2-45 CommonDriverInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
rollbackLastErrPackage	No	BOOL	true/false	false	Whether to deliver the reverse packets of configuration error packets upon rollback when the device returns an error during the configuration delivery. Assume that five packets are delivered, the first two packets are delivered successfully, and the third packet fails to be delivered. If this field is set to **false**, reverse packets of the first two packets are delivered in the rollback phase. If this field is set to **true**, the reverse packets of the first three packets are delivered in the rollback phase.
pathNeedPreProcess	No	STRING	N/A	N/A	QName of the feature requiring packet pre-processing. If this field is set to the corresponding QName, netconfPreprocess is invoked to process the query result packet returned by the device before the packet is processed. After the packet is processed, it is converted.
pathNeedPostProcess	No	STRING	N/A	N/A	QName of the feature requiring packet post-processing. If this field is set to the corresponding QName, netconfPostprocess is invoked to process a packet before delivering the packet to the device.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
pathNeedEcsMapping	No	STRING	N/A	N/A	QName of the feature requiring service packet customized processing. If this field is set to the corresponding QName, toDocument and toDataObject are invoked to convert the packets sent to and returned by a device.
pathNeedEcsRpcMapping	No	STRING	N/A	N/A	RPC QName requiring rpc packet customized processing. If this field is set to the corresponding QName, toRpcDocument and toRpcDataObject are invoked to convert the rpc packets sent to and returned by a device.
para	No	EcsDeviceDriverPara	N/A	N/A	This field is an extended field. It is a key-value pair used for data consistency.
isNeedGlobalPush	No	STRING	"true"/"false"	"false"	Whether to perform global configurations for packet delivery. If the value is **false**, the packets sent by the same device in the same transaction are packed and then delivered. If the value is **true**, the packets sent by the same device in the same transaction are not packed.
isPathNeedPush	No	STRING	"true"/"false"	"false"	Whether to pack the packets of this feature for delivery. If the value is **false**, the packets sent by the same device in the same transaction are packed and then delivered. If the value is **true**, the packets sent by the same device in the same transaction are not packed with subsequent packets for delivery.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
pathNeedEcsDBHook	No	STRING	N/A	N/A	After configurations are delivered to devices, new configurations will be automatically generated on the devices. However, NCE only stores the delivered configurations but not the new configurations. As a result, the configurations on NCE are inconsistent with those on the devices. Therefore, before the configuration delivery, an association processing is performed to generate new configurations on NCE.
checkSync	No	BOOL	true/ false	-	Indicates whether to check the consistency between the configuration delivered by NCE and the device configuration. If so, an error is reported in case of configuration inconsistency. If not, configuration consistency will not be checked.
unsupportedOperations	No	STRING	"create"/ "delete"/ "create,delete"	N/A	Operations that devices do not support and will be automatically converted after being configured. If the create operation is configured, the delivered create operation packet is automatically converted into the merge operation. If the delete operation is configured, the delivered delete operation packet is automatically converted into the remove operation.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
pathUnsupportedOperations	No	MAP<STRING,STRING>	N/A	N/A	Operations that some features do not support will be automatically converted after being configured. If the create operation is configured, the create operation packet delivered in a feature is automatically converted into the merge operation. If the delete operation is configured, the delete operation packet delivered in a feature is automatically converted into the remove operation.
deleteStrategy	No	DeleteStrategy	PATH_ADD/ CONTAINER_WITH_PRESENCE	N/A	Whether to delete detailed information in the packets of the delete and remove operations. If the value is **CONTAINER_WITH_PRESENCE**, content in the delivered packets is not deleted.
uiSupportAbility	No	UiSupportAbility	N/A	N/A	GUI operation capability and connection protocol customized based on the device version.
configDeliveryStrategy	No	ConfigDeliveryStrategy	SYNC_DB_AND_SYNC_DEVICE/ SYNC_DB_AND_ASYNC_DEVICE	N/A	Policy to be delivered. SYNC_DB_AND_SYNC_DEVICE: Save the configuration to the database and deliver it to the device immediately. SYNC_DB_AND_ASYNC_DEVICE: After the configuration is saved to the database, the configuration is not delivered to a device immediately. Instead, the configuration is delivered when the device is idle and online.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
outOfSyncDeviceStrategyInHASwitching	No	OutOfSyncDeviceStrategyInHASwitching	N/A	N/A	Processing policy customized based on the device version after the NCE active/standby switchover. ALARM: Generate an alarm. SYNV_TO_DEVICE: Notify data consistency verification. CUSTOM_HOOK: Service mounting processing hook
SwitchHACustomHook	No	STRING	N/A	N/A	Service processing policy. After an NCE active/standby switchover, you can use this processing policy customized based on the device version.
ecsCheckHook	No	EcsCheckHook	N/A	N/A	Resource monitoring customized based on path.
ecsBehaviourHook	No	EcsBehaviourHook	N/A	N/A	Behavior driver.

Table 2-46 EcsCheckHook

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
moduleName	Yes	STRING	N/A	N/A	Service feature name.
appClass	Yes	STRING	N/A	N/A	Path of the package where the implementation class is located.

Table 2-47 EcsBehaviourHook

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
apiClass	Yes	STRING	N/A	N/A	API class.
implClass	Yes	STRING	N/A	N/A	Implementation class.

Table 2-48 NetconfDriverInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
classification	No	STRING	"huawei-v5"/"huawei-v8"/"cisio"	N/A	Device prototype.
phase	No	STRING	"one"/"two"	N/A	Delivery phase supported by a device.
maxPkgLength	No	INT32	> 0	N/A	Maximum size of a single packet supported by a device.
netconfLock	No	BOOL	true/false	false	Whether to obtain the device lock before delivery. If the device is a Juniper device, the value of this field must be **true**.
errorOption	No	STRING	"rollback-on-error"/"stop-on-error"/"continue-on-error"	"rollback-on-error"	Policy for processing incorrect device configuration data.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
testOption	No	STRING	"set"	N/A	Whether to add the <test-option> tag to the packet to be delivered to a device. If the value of this field is **set**, the <test-option>set</test-option> tag is added to the packet.
netconfHelloEntity	No	NetconfHelloEntity	N/A	N/A	Hello packet customization for device management: standardType: standard packet extendType: user-defined packet defaultType: default packet (YANG interconnection)
modelDiff	No	STRING	"same"/"match"	N/A	NETCONF packet conversion type. If the value of this field is **same**, namespace of the delivered packet is converted into https://www.huawei.com/netconf/vrp.
rpcPrefix	No	RpcPrefix	N/A	N/A	Whether to add a prefix to the delivered rpc packet.

Table 2-49 RpcPrefix

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
rpcQname	Yes	STRING	N/A	N/A	Feature QName.
prefix	Yes	STRING	N/A	N/A	Prefix node name of rpc packets.

Sample Request and Sample Response

Python invoking example:

```
# Customize a universal driver for a device.
def getCommonDriverInfo(self, aoccontext, request=None):
    self.logger.info('getCommonDriverInfo start.')
    common_driver = CommonDriverInfo()

    """
    Whether to delete detailed information in the packets of the delete and remove operations.
     If the value is CONTAINER_WITH_PRESENCE (the value is 1), content in the delivered packets is not deleted.
    """
    common_driver.deleteStrategy = 1
    syncToDel = common_driver.para.add()

    """
    This parameter is optional. It specifies whether southbound device configuration instances can be deleted during data consistency verification.
    The value can be false (default value, not supported) or true (supported).
    Note: Configuration instances of southbound devices can be deleted during data consistency verification. If data consistency verification is performed by mistake, southbound devices may be disconnected.
    """
    syncToDel.key = "sync-to-del-enable"
    syncToDel.value = "true"

    self.logger.info('getCommonDriverInfo end.')
    return common_driver


# Customize the NETCONF driver.
def getNetconfDriverInfo(self, aoccontext, request=None):
    self.logger.info('getNetconfDriverInfo start.')
    netconf_driver = NetconfDriverInfo()
    netconf_driver.phase = "two"
    netconf_driver.classification = "huawei-v5"
    self.logger.info('getNetconfDriverInfo end.')
    return netconf_driver
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.2.2 Data Association

Type

API-4: device data query interface and operation interface

Typical Scenarios

After configurations are delivered to devices, new configurations will be automatically generated on the devices. However, NCE only stores the delivered configurations but not the new configurations. As a result, the configurations on NCE are inconsistent with those on the devices. Therefore, before the configuration delivery, an association processing is performed to generate new configurations on NCE.

Functions

Generate new configurations on NCE before configurations are delivered to devices.

Constraints

1. Prerequisites: The device has gone online.
2. Precautions: The generated data is consistent with the device behavior.
3. Usage restrictions: This function can be used only when associated data is generated during configuration delivery.
4. Relationships between APIs: Depend on the pathNeedEcsDBHook field in **2.3.2.1 Device Driver Data Configuration**.

Invoking Method

Python-based interface invoking method:

```
# Data association processing
# @param aoccontext: context
# @param request: data to be associated
def dbPostProcess(self, aoccontext, request):
    pass
```

Request Parameters

Table 2-50 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
aoccontext	Yes	REFERENCE	For details, see the AocContext table.	N/A	Context information.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
request	Yes	REFERENCE	For details, see the EcsDbConfig table.	N/A	Data association input parameter.

Table 2-51 AocContext

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transactionId	No	STRING	N/A	N/A	Transaction ID.
deviceVendor	Yes	STRING	N/A	N/A	Device vendor.
deviceType	Yes	STRING	N/A	N/A	Device type.
deviceVersio	Yes	STRING	N/A	N/A	Device software version number.

Table 2-52 EcsDbConfig

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
path	Yes	STRING	N/A	N/A	Path.
data	Yes	STRING	N/A	N/A	Data type.
opType	Yes	STRING	N/A	N/A	Operation type.

Sample Request and Sample Response

Python invoking example:

```
# Data association processing
# @param aoccontext: context
# @param request: data to be associated
def dbPostProcess(self, aoccontext, request):
```



```
self.logger.info('dbPostProcess start.')
ecsData = request.data
ecsPath = request.path
ecsopType = request.opType

#  /huawei-ifm:interfaces/huawei-ifm:interface/asd/huawei-ifm:mpls/huawei-ifm:l2vc/primary
ecsDbConfigOut = EcsDbConfigOut()

# /huawei-ifm:interfaces/huawei-ifm:interface/Giga0000001/huawei-ifm:mpls/huawei-ifm:l2vc/secondary
if "huawei-ifm:l2vc/primary" in ecsPath and ecsopType == "DELETE":
    ifmKey = self.get_key(ecsPath)
    ecsDbconfig = ecsDbConfigOut.ecsDbConfig.add()
    ecsDbconfig.path = "/huawei-ifm:interfaces/huawei-ifm:interface/" + ifmKey + \
                       "/huawei-ifm:mpls/huawei-ifm:l2vpn/huawei-ifm:service-name"
elif "huawei-ifm:l2vc/secondary" in ecsPath and ecsopType == "DELETE":
    ifmKey = self.get_key(ecsPath)
    ecsDbconfig1 = ecsDbConfigOut.ecsDbConfig.add()
    ecsDbconfig2 = ecsDbConfigOut.ecsDbConfig.add()
    ecsDbconfig1.path = "/huawei-ifm:interfaces/huawei-ifm:interface/" + ifmKey + \
                        "/huawei-ifm:mpls/huawei-ifm:l2vpn/reroute"
    ecsDbconfig2.path = "/huawei-ifm:interfaces/huawei-ifm:interface/" + ifmKey + \
                        "/huawei-ifm:mpls/huawei-ifm:l2vpn/redundancy"
self.logger.info('dbPostProcess end.')
return ecsDbConfigOut
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.2.3 Service Customization

Type

API-4: device data query interface and operation interface

Typical Scenarios

The device does not support common data, and services need to be customized.

Functions

Provide the data matching devices.

Constraints

1. Prerequisites: The device has gone online.
2. Precautions: The user-defined conversion rules must be the same as those on the device.
3. Usage restrictions: The rules must be defined by users to ensure the correctness of rule conversion.
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
# User-defined processing
# @param aoccontext: context
# @param request: data to be processed
def netconfTransformer(self, aoccontext, request):
    pass
```

Request Parameters

Table 2-53 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
aoccontext	Yes	REFERENCE	For details, see the AocContext table.	N/A	Context information.
request	Yes	REFERENCE	For details, see the AtomicConfig table.	N/A	Service input parameters to be customized.

Table 2-54 AocContext

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transactionId	No	STRING	N/A	N/A	Transaction ID.
deviceVendor	Yes	STRING	N/A	N/A	Device vendor.
deviceType	Yes	STRING	N/A	N/A	Device type.
deviceVersio	Yes	STRING	N/A	N/A	Device software version number.

Table 2-55 AtomicConfig

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
path	Yes	STRING	N/A	N/A	YANG model path.
data	Yes	STRING	N/A	N/A	YANG model data.
optype	Yes	STRING	N/A	N/A	Operation type.

Table 2-56 NetconfMsg

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
msg	Yes	STRING	N/A	N/A	Packet.
isRpc	Yes	BOOL	N/A	N/A	Whether the packet is an RPC configuration packet.

Sample Request and Sample Response

Python invoking example:

```
def netconfTransformer(self, aoccontext, input=None):
    self.logger.info('netconfTransformer start.')
    netconf_msg = NetconfMsg()
    if input.data.find('xmlns="urn:huawei:yang:huawei-ac-ne-snmp"') > -1:
        netconf_msg.msg = input.data.replace('xmlns="urn:huawei:yang:huawei-ac-ne-snmp"',
                                             'xmlns="http://www.huawei.com/netconf/vrp" content-version="1.0"  '
                                             'format-version="1.0"')
    else:
         netconf_msg.msg = input.data.replace('http://www.huawei.com/netconf/vrp',
                                             'urn:huawei:yang:huawei-ac-ne-snmp')
         self.logger.info('netconfTransformer end.')
    return netconf_msg
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.2.4 Pre-processing

Type

API-4: device data query interface and operation interface

Typical Scenarios

When data is synchronized from a device to NCE, some fields of the data may need to be modified to make them be identified by NCE.

Functions

Adjust the data synchronized from a device to NCE so that NCE can identify the data.

Constraints

1. Prerequisites: The device has gone online.
2. Precautions: Ensure that packets are pre-processed correctly.
3. Usage restrictions: This API applies only to NETCONF.
4. Relationships between APIs: Depend on the pathNeedPreProcess field in **2.3.2.1 Device Driver Data Configuration**.

Invoking Method

Python-based interface invoking method:

```
#Pre-processing
# @param aoccontext: context
# @param request: data to be processed
# @return: processed data
def netconfPreprocess(self, aoccontext, request):
    pass
```

Request Parameters

Table 2-57 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
aoccontext	Yes	REFERENCE	For details, see the AocContext table.	N/A	Context information.
request	Yes	REFERENCE	For details, see the NetconfAtomicConfig table.	N/A	Input parameter for pre-processing.

Table 2-58 AocContext

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transactionId	No	STRING	N/A	N/A	Transaction ID.
deviceVendor	Yes	STRING	N/A	N/A	Device vendor.
deviceType	Yes	STRING	N/A	N/A	Device type.
deviceVersio	Yes	STRING	N/A	N/A	Device software version number.

Table 2-59 NetconfAtomicConfig

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
data	Yes	STRING	N/A	N/A	NETCONF packet, in XML format.

Sample Request and Sample Response

Python invoking example:

```
def netconfPreprocess(self, aoccontext, request=None):
    self.logger.info('netconfPreprocess start.')
    doc = request.data
    new_doc = doc.replace("<viewName>", "<viewName>pre")
    netconfAtomicConfig = NetconfAtomicConfig()
    netconfAtomicConfig.data = new_doc
    self.logger.info('netconfPreprocess end.')
    return netconfAtomicConfig
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.2.5 Post-processing

Type

API-4: device data query interface and operation interface

Typical Scenarios

When data is synchronized from NCE to a device, some fields of the data may need to be converted to make them be identified by the device.

Functions

Adjust the data synchronized from NCE to a device so that the device can identify the data.

Constraints

1. Prerequisites: The device has gone online.
2. Precautions: Ensure that the packets are post-processed correctly.
3. Usage restrictions: This API applies only to NETCONF.
4. Relationships between APIs: Depend on the pathNeedPostProcess field in **2.3.2.1 Device Driver Data Configuration**.

Invoking Method

Python-based interface invoking method:

```
# Post-processing
# @param context: context
# @param input: data to be processed
# @return: processed data
def netconfPostprocess(self, aoccontext, request):
    pass
```

Request Parameters

Table 2-60 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
aoccontext	Yes	REFERENCE	For details, see the AocContext table.	N/A	Context information.
request	Yes	REFERENCE	For details, see the NetconfAtomicConfig table.	N/A	Post-processing input parameter.

Table 2-61 AocContext

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transactionId	No	STRING	N/A	N/A	Transaction ID.
deviceVendor	Yes	STRING	N/A	N/A	Device vendor.
deviceType	Yes	STRING	N/A	N/A	Device type.
deviceVersio	Yes	STRING	N/A	N/A	Device software version number.

Table 2-62 NetconfAtomicConfig

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
data	Yes	STRING	N/A	N/A	NETCONF packet, in XML format.

Sample Request and Sample Response

Python invoking example:

```
def netconfPostprocess(self, aoccontext, request=None):
    self.logger.info('netconfPostprocess start.')
    doc = request.data
    new_doc = doc.replace("<viewName>", "<viewName>post")
    netconfAtomicConfig = NetconfAtomicConfig()
    netconfAtomicConfig.data = new_doc
    self.logger.info('netconfPostprocess end.')
    return netconfAtomicConfig
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.3 Data Consistency Customization

2.3.3.1 Feature Customization

Type

API-4: device data query interface and operation interface

Typical Scenarios

Feature scope for inconsistency discovery, data consistency verification, and synchronization needs to be customized based on service requirements.

The data consistency module can analyze the YANG model in an SND package and generate data paths for data inconsistency discovery, data consistency verification, and synchronization based on the top-layer container and top-layer list configuration node of each module.

Functions

Customize the data paths used for data inconsistency discovery, data consistency verification, and synchronization.

Constraints

1. Prerequisites: The device has gone online.
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: depend on **2.3.1.2 Device Connection Customization** and **2.3.2.1 Device Driver Data Configuration**.

Invoking Method

Python-based interface invoking method:

```
    def getFeatures(self, aoccontext, request=None):
        """
        Customize the data paths used for data inconsistency discovery, data consistency verification, and
synchronization.
        :param aoccontext: context
        :param request: parameter carried in the method
        :return: customized feature information
        """
```

Request Parameters

Table 2-63 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
aoccontext	Yes	REFERENCE	For details, see the AocContext table.	N/A	Context information.

Table 2-64 AocContext

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transactionId	No	STRING	N/A	N/A	Transaction ID, which is used to ensure data consistency and enable the transaction mechanism.
deviceVendor	Yes	STRING	N/A	N/A	Device vendor.
deviceType	Yes	STRING	N/A	N/A	Device type.
deviceVersion	Yes	STRING	N/A	N/A	Device software version number.

Table 2-65 FeatureCfgsMsg

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
replace	No	BOOLEAN	N/A	false	Whether to enable feature customization. The default value is false. If the value is true, the return value of the feature customization method takes effect.
features	Yes	List<Feature>	N/A	N/A	Feature list.

Table 2-66 Feature

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
name	Yes	STRING	N/A	N/A	Feature name (module name: for example, huawei-ifm).

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
operType	No	Oper	MERGE, REPLACE, DELETE	MERGE	Rules for combining features and YANG files. MERGE (default value): The customized fields are merged into the features generated by YANG by default. REPLACE: This feature is customized. DELETE: The feature is deleted.
depends	No	List<STRING>	N/A	N/A	List of dependent features (Name field of the feature).
functions	No	List<Function>	N/A	N/A	Data path list of a feature.

Table 2-67 Function

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
value	Yes	STRING	N/A	N/A	Data path for inconsistency comparison. For example: (https://www.huawei.com/netconf/vrp/huawei-ifm?revision=2018-06-11)ifm
collectPath	No	STRING	N/A	N/A	Path for collecting configuration data from southbound devices. The default value is the same as that of the value field. For example, (https://www.huawei.com/netconf/vrp/huawei-ifm?revision=2018-06-11)ifm

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
collectFilter	No	STRING	N/A	N/A	Filter for collecting configuration data from southbound devices. All configurations corresponding to the default data collection path are collected.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
supportDel	No	Switch	ENABLE, DISABLE	DISABLE	Whether southbound device configuration instances can be deleted. If this parameter is not specified, the global setting takes effect. Global setting reference: getCommonDriverInfo() method in the device driver. Note: Configuration instances of southbound devices can be deleted during data consistency verification. If data consistency verification is performed by mistake, southbound devices may be disconnected.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
preSyncToNe	No	BOOLEAN	false,true	false	Whether to customize features before data consistency verification. If this parameter is set to false, customization before data consistency verification is not supported. Set this parameter to true. For details, see **2.3.3.2 Data Consistency Verification Customization**.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
preSyncFromNe	No	BOOLEAN	false, true	false	Whether to customize features before synchronization. If this parameter is set to false, customization before synchronization is not supported. Set this parameter to true. For details, see **2.3.3.3 Data Synchronization Customization**.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
postSyncFromNe	No	BOOLEAN	false,true	false	Whether to customize features after synchronization. If this parameter is set to false, customization after synchronization is not supported. Set this parameter to true. For details, see **2.3.3.4 Post-processing for Synchronization**.

Sample Request and Sample Response

Python invoking example:

```
    def getFeatures(self, aoccontext, request=None):
        self.logger.info('getFeatures start.')
        feature_msg = FeatureCfgsMsg()
        feature_msg.replace = True

        # Add a feature customization.
        feature_msg.features.extend(self.build_feature('huawei-l3vpn', 'huawei-rtp', '(http://www.huawei.com/
netconf/vrp/huawei-l3vpn?revision=2018-06-11)l3vpn'))
        self.logger.info('getFeatures end.')
        return feature_msg


    def build_feature(self, name, depends, path):
        feature = Feature()

        # Feature name
        feature.name = name

        # Operation type
        feature.operType = Feature.MERGE
        feature.depends.extend([depends])
        function = Function()

        # Set the inconsistency comparison path.
        function.value = path
```



```
    # Set the path for collecting data from southbound devices.
    function.collectPath = path
    feature_name = name.replace('huawei-', '')

    # Set customization before data consistency verification.
    function.preSyncToNe = False

    # Set customization before synchronization.
    function.preSyncFromNe = False

    # Set customization after synchronization.
    function.postSyncFromNe = False
    feature.functions.extend([function])
    return [feature]
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.3.2 Data Consistency Verification Customization

Type

API-4: device data query interface and operation interface

Typical Scenarios

Data delivered to southbound devices during data consistency verification needs to be modified. For example, if the configuration A of a southbound device does not need to be verified, you can delete the configuration from the data to be verified.

Functions

Customize the data used for data consistency verification.

Constraints

1. Prerequisites: The device has gone online.
2. Precautions: N/A
3. Usage restrictions: The method takes effect only when the preSyncToNe attribute of feature customization is set to true.
4. Relationships between APIs: depend on **2.3.1.2 Device Connection Customization**, **2.3.2.1 Device Driver Data Configuration**, and **2.3.3.1 Feature Customization**.

Invoking Method

Python-based interface invoking method:

```
def preSyncToNe(self, request, aoccontext):
    """
```



```
    Customize the data used for data consistency verification.
    :param request: parameter carried in the method
     :param aoccontext: context
     :return: customized feature information
    """
```

Request Parameters

Table 2-68 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
aoccontext	Yes	REFERENCE	For details, see the AocContext table.	N/A	Context information.
diffDataInMsg	Yes	REFERENCE	For details, see the DiffDataInMsg table.	N/A	Inconsistent data information.
neId	Yes	STRING	N/A	N/A	Device ID.
featureId	Yes	STRING	N/A	N/A	Feature ID.
ChangeData	Yes	REFERENCE	N/A	N/A	Inconsistent data. Forwarder data and controller data can be compared to generate differences.

Table 2-69 AocContext

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transactionId	No	STRING	N/A	N/A	Transaction ID, which is used to ensure data consistency and enable the transaction mechanism.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
deviceVendor	Yes	STRING	N/A	N/A	Device vendor.
deviceType	Yes	STRING	N/A	N/A	Device type.
deviceVersion	Yes	STRING	N/A	N/A	Device software version number.

Table 2-70 DiffDataInMsg

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
feature	Yes	STRING	N/A	N/A	Name of a feature.
regPath	No	STRING	N/A	N/A	Data path for service registration.
transId	Yes	STRING	N/A	N/A	Transaction ID.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
diffDatas	Yes	List<STRING>	N/A	N/A	List of inconsistent data. Inconsistent data is in XML format. The <diff> label identifies inconsistent data. <left> indicates data on NCE, and <right> indicates data on a southbound device. Modify southbound device data based on the data on NCE.

Table 2-71 DiffDataOutMsg

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
diffDatas	Yes	LIST<REFERENCE>	For details, see the DiffData table.	N/A	Data returned after service processing.

Table 2-72 DiffData

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
singleMsg	No	BOOLEAN	true/false	false	Whether inconsistent data needs to be separately submitted to southbound devices.
diffData	Yes	STRING	N/A	N/A	Inconsistent data in XML format. The <diff> label identifies inconsistent data. <left> indicates data on NCE, and <right> indicates data on a southbound device. Modify southbound device data based on the data on NCE.

Sample Request and Sample Response

Python invoking example:

```
def preSyncToNe(self, request, aoccontext=None):
    self.logger.info('preSyncToNe start.')
    dataIn = request

     # Obtain the data path for service registration.
    regPath = dataIn.regPath

    # Obtain the feature name.
    featureName = dataIn.feature

    # Obtain the transaction ID.
    transId = dataIn.transId
    self.logger.info("preSyncToNe begin regPath ={}, feature={}, transId={}", regPath, featureName, transId)
```



```
        # Obtain the inconsistent data returned after service processing.
        diffDatas = dataIn.diffDatas

        # Obtain the capacity of the inconsistent data list.
        diffSize = len(diffDatas)

        # If the inconsistency is null or the inconsistent data capacity is 0, no inconsistency exists and the
function directly returns null.
        if diffDatas is None or diffSize == 0:
            return None

       # Customize the data used for data consistency verification.
        dataOut = DiffDataOutMsg()

        # Traverse the inconsistent data list and process the inconsistent data.
        for data in diffDatas:
            # Obtain the inconsistent data in XML format.
            diffXml = data

            # TODO Process inconsistent data in XML format.
            diffDataOut = diffData()

            # Whether the inconsistent data needs to be separately submitted to a southbound device. false:
The inconsistent data does not need to be separately submitted to the southbound device.
            diffDataOut.singleMsg = False
            diffDataOut.diffData = diffXml
            dataOut.diffDatas.extend([diffDataOut])
        self.logger.info('preSyncToNe end.')
        return dataOut
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.3.3 Data Synchronization Customization

Type

API-4: device data query interface and operation interface

Typical Scenarios

Data saved to NCE during data consistency synchronization needs to be modified. For example, the password field collected from a southbound device is encrypted and saved to NCE.

Functions

Customize the data used for data consistency synchronization.

Constraints

1. Prerequisites: The device has gone online. The method takes effect when the preSyncFromNe attribute of feature customization is set to true.

2. Precautions: The following two interfaces can be used to implement data consistency customization: AbstractSND (existing interface) and YangSND (model rule interface).
3. Usage restrictions: N/A
4. Relationships between APIs: depend on **2.3.1.2 Device Connection Customization**, **2.3.2.1 Device Driver Data Configuration**, and **2.3.3.1 Feature Customization**.

Invoking Method

Python-based interface invoking method:

```
    def preSyncFromNe(self, request, aoccontext):
        """
Data used for user-defined data consistency synchronization.
:param request: parameter carried in the method
:param aoccontext: context
:return: customized feature information
        """
```

Request Parameters

Table 2-73 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
aoccontext	Yes	REFERENCE	For details, see the AocContext table.	N/A	Context information.
diffDataInMsg	Yes	REFERENCE	For details, see the DiffDataInMsg table.	N/A	Inconsistent data information.
neId	Yes	STRING	N/A	N/A	Device ID.
featureId	Yes	STRING	N/A	N/A	Feature ID.
ChangeData	Yes	REFERENCE	N/A	N/A	Inconsistent data. Forwarder data and controller data can be compared to generate differences.

Table 2-74 AocContext

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transactionId	No	STRING	N/A	N/A	Transaction ID, which is used to ensure data consistency and enable the transaction mechanism.
deviceVendor	Yes	STRING	N/A	N/A	Device vendor.
deviceType	Yes	STRING	N/A	N/A	Device type.
deviceVersion	Yes	STRING	N/A	N/A	Device software version number.

Table 2-75 DiffDataInMsg

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
feature	Yes	STRING	N/A	N/A	Name of a feature.
regPath	No	STRING	N/A	N/A	Data path for service registration.
transId	Yes	STRING	N/A	N/A	Transaction ID.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
diffDatas	Yes	List<STRING>	N/A	N/A	List of inconsistent data. Inconsistent data in XML format. The <diff> label identifies inconsistent data. <left> indicates data on NCE, and <right> indicates data on a southbound device. During data consistency synchronization, NCE data is modified based on southbound device data.

Table 2-76 DiffDataOutMsg

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
diffDatas	Yes	LIST<REFERENCE>	For details, see the DiffData table.	N/A	Data returned after service processing.

Table 2-77 DiffData

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
singleMsg	No	BOOLEAN	true/false	false	Whether inconsistent data needs to be separately submitted to southbound devices. This field is valid only during data verification.
diffData	Yes	STRING	N/A	N/A	Inconsistent data in XML format. The <diff> label identifies inconsistent data. <left> indicates data on NCE, and <right> indicates data on a southbound device. During data consistency synchronization, NCE data is modified based on southbound device data.

Sample Request and Sample Response

Python invoking example:

```
def preSyncFromNe(self, request, aoccontext):
    self.logger.info('preSyncFromNe start.')
```



```
    dataIn = request

     # Obtain the data path for service registration.
    regPath = dataIn.regPath

    # Obtain the feature name.
    featureName = dataIn.feature

    # Obtain the transaction ID.
    transId = dataIn.transId
    self.logger.info("preSyncFromNe begin regPath ={}, feature={}, transId={}", regPath, featureName,
transId)

    # Obtain the inconsistent data returned after service processing.
    diffDatas = dataIn.diffDatas

    # Obtain the capacity of the inconsistent data list.
    diffSize = len(diffDatas)

    # If the inconsistency is null or the inconsistent data capacity is 0, no inconsistency exists and the
function directly returns null.
    if diffDatas is None or diffSize == 0:
        return None

    # Customize the data used for data consistency verification.
    dataOut = DiffDataOutMsg()

    # Traverse the inconsistent data list and process the inconsistent data.
    for data in diffDatas:
        # Obtain the inconsistent data in XML format.
        diffXml = data

        # TODO Process inconsistent data in XML format.
        diffDataOut = diffData()

        // Indicates whether the inconsistent data needs to be submitted to a southbound device
separately. This field is valid only during data consistency verification.
        diffDataOut.singleMsg = False
        diffDataOut.diffData = diffXml
        dataOut.diffDatas.extend([diffDataOut])
    self.logger.info('preSyncFromNe end.')
    return dataOut
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.3.4 Post-processing for Synchronization

Type

API-4: device data query interface and operation interface

Typical Scenarios

Service resources need to be occupied after data consistency synchronization. For example, after an L3VPN instance is synchronized from a southbound device, label resources need to be occupied.

Functions

Customize post-synchronization processing.

Constraints

1. Prerequisites: The device has gone online. The method takes effect when the postSyncFromNe attribute of feature customization is set to true.
2. Precautions: The following two interfaces can be used to implement data consistency customization: AbstractSND (existing interface) and YangSND (model rule interface).
3. Usage restrictions: N/A
4. Relationships between APIs: depend on **2.3.1.2 Device Connection Customization**, **2.3.2.1 Device Driver Data Configuration**, and **2.3.3.1 Feature Customization**.

Invoking Method

Python-based interface invoking method:

```
    def postSyncFromNe(self, request, aoccontext):
        """
        Customize post-synchronization processing.
:param request: parameter carried in the method
        :param aoccontext: context
        :return: customized feature information
        """
```

Request Parameters

Table 2-78 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
aoccontext	Yes	REFERENCE	For details, see the AocContext table.	N/A	Context information.
diffDataInMsg	Yes	REFERENCE	For details, see the DiffDataInMsg table.	N/A	Inconsistent data information.
neId	Yes	STRING	N/A	N/A	Device ID.
featureId	Yes	STRING	N/A	N/A	Feature ID.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
ChangeData	Yes	REFERENCE	N/A	N/A	Inconsistent data. Forwarder data and controller data can be compared to generate differences.

Table 2-79 AocContext

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transactionId	No	STRING	N/A	N/A	Transaction ID, which is used to ensure data consistency and enable the transaction mechanism.
deviceVendor	Yes	STRING	N/A	N/A	Device vendor.
deviceType	Yes	STRING	N/A	N/A	Device type.
deviceVersion	Yes	STRING	N/A	N/A	Device software version number.

Table 2-80 DiffDataInMsg

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
feature	Yes	STRING	N/A	N/A	Name of a feature.
regPath	No	STRING	N/A	N/A	Data path for service registration.
transId	Yes	STRING	N/A	N/A	Transaction ID.
diffDatas	Yes	List<STRING>	N/A	N/A	List of inconsistent data. Inconsistent data in XML format. The <diff> label identifies inconsistent data. <left> indicates data on NCE, and <right> indicates data on a southbound device. During data consistency synchronization, NCE data is modified based on southbound device data.

Sample Request and Sample Response

Python invoking example:

```
def postSyncFromNe(self, request, aoccontext):
    self.logger.info('postSyncFromNe start.')
```



```
    dataIn = request

     # Obtain the data path for service registration.
    regPath = dataIn.regPath

    # Obtain the feature name.
    featureName = dataIn.feature

    # Obtain the transaction ID.
    transId = dataIn.transId
    self.logger.info("postSyncFromNe begin regPath ={}, feature={}, transId={}", regPath, featureName,
transId)

    # Obtain the inconsistent data returned after service processing.
    diffDatas = dataIn.diffDatas

    # Obtain the capacity of the inconsistent data list.
    diffSize = len(diffDatas)

    # If the inconsistency is null or the inconsistent data capacity is 0, no inconsistency exists and the
function directly returns null.
    if diffDatas is None or diffSize == 0:
        return

    # Traverse the inconsistent data list and process the inconsistent data.
    for data in diffDatas:
        # Obtain the inconsistent data in XML format.
        diffXml = data

        # TODO
    self.logger.info('postSyncFromNe end.')
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.3.5 Customization of Inconsistency Comparison Methods

Type

API-4: device data query interface and operation interface

Typical Scenarios

Data inconsistency comparison modes are classified into E2E mode and non-E2E mode (WHOLE_DIFF) based on data sources. E2E inconsistencies are compared based on controller data, and non-E2E inconsistencies are compared based on forwarder data.

Functions

Customize the device type and inconsistency comparison mode.

Constraints

1. Prerequisites: The device has gone online.

2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: depend on **2.3.1.2 Device Connection Customization** and **2.3.2.1 Device Driver Data Configuration**.

Invoking Method

Python-based interface invoking method:

```
    def getEcsConfigParams(self, aoccontext, request=None):
        """
Customize the inconsistency comparison mode.
        :param aoccontext: context
        :param request:None
        :return:EcsConfigOut
        """
```

Request Parameters

Table 2-81 Parameters

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
aoccontext	Yes	REFERENCE	For details, see the AocContext table.	N/A	Context information.
EcsConfigOut	Yes	REFERENCE	N/A	N/A	Consistent configuration data.

Table 2-82 AocContext

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
transactionId	No	STRING	N/A	N/A	Transaction ID, which is used to ensure data consistency and enable the transaction mechanism.
deviceVendor	Yes	STRING	N/A	N/A	Device vendor.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
deviceType	Yes	STRING	N/A	N/A	Device type.
deviceVersion	Yes	STRING	N/A	N/A	Device software version number.

Table 2-83 EcsConfigOutPara

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
EcsConfigOutPara	No	REFERENCE	N/A	N/A	This is an extended field and is a key-value pair.

Sample Request and Sample Response

Python invoking example:

```
def getEcsConfigParams(self, aoccontext, request=None):
    """
   Customize the inconsistency comparison mode.
    :param aoccontext: context
    :param request:None
    :return:EcsConfigOut
    """
    return None
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.4 SND CLI Framework Customization

2.3.4.1 Obtaining the Whitelist of CLI Commands to Be Transparently Transmitted

Type

API-4: device data query interface and operation interface

Typical Scenarios

A whitelist of CLI commands that can be transparently transmitted to devices needs to be customized.

Functions

Specify the CLI commands that can be transparently transmitted to devices to ensure service security.

Constraints

1. Prerequisites: The **CliPassthroughCommands.xml** configuration file exists in the resources directory of an SND package.
2. Precautions: In the XML configuration file, each item is a regular expression.
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def getPassthroughCommands(self, aoccontext):
    """
    Obtain the CLI command whitelist from the XML configuration file in the resources directory of the SND package.
    :param aoccontext: context
    :return: CLI command whitelist
    """
    pass
```

Request Parameters

Table 2-84 CliPassthroughOutput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
commands	No	CliPassthroughCommand	For details, see the CliPassthroughCommand table.	N/A	CLI command whitelist.

Table 2-85 CliPassthroughCommand

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
command	No	STRING	N/A	N/A	Regular expression of a command.

Sample Request and Sample Response

Python invoking example:

```
def getPassthroughCommands(self, aoccontext):
    self.logger.info("getPassthroughCommands begin")
    output = CliPassthroughOutput()

    # Create a file object to reference the XML configuration file.
    file_path = os.path.join(self.resourceDir, "resources/CliPassthroughCommands.xml")

    # Parse the XML configuration file.
    tree = ET.parse(file_path)
    root = tree.getroot()

    # Construct the return parameter based on the configuration file content.
    for child in root:
        value = codecs.getdecoder("unicode_escape")(child.text)[0]
        commandEntity = output.add()
        commandEntity.command = value
    self.logger.info('getPassthroughCommands end.')
    return output
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.4.2 YANG-to-CLI Conversion

Type

API-4: device data query interface and operation interface

Typical Scenarios

If the conversion logic of the CLI driver is not supported, the customized YANG-to-CLI conversion method needs to be used for conversion.

Functions

Customize the YANG-to-CLI conversion logic.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def yangToCli(self, aoccontext, request):
    """
    Customize the YANG-to-CLI conversion logic.
    :param aoccontext: context
    :param request: input parameter for conversion
    :return: conversion result
    """
    pass
```

Request Parameters

Table 2-86 CliCustomTransformInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
path	Yes	STRING	N/A	N/A	Request path.
data	Yes	STRING	N/A	N/A	Request data.
oper_type	Yes	OperType	For details, see the OperType table.	N/A	Operation type of a request.
startOffset	Yes	Integer	≥ 0	N/A	Offset of a request.
neId	Yes	STRING	N/A	N/A	ID of the requested NE.
commandLines	No	CommandLine	For details, see the CommandLine table.	N/A	Requested CLI command.

Table 2-87 OperType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	READ_CFG READ_OPR CREATE MERGE DELETE RPC	N/A	Operation type.

Table 2-88 CommandLine

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
index	No	Integer	≥ 0	N/A	Serial number of a CLI command.
commandLine	No	STRING	N/A	N/A	Content of a CLI command.

Table 2-89 CliCustomTransformOutput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
data	Yes	STRING	N/A	N/A	Modified data.
endOffset	Yes	Integer	≥ -1	N/A	End offset.
commandLines	Yes	Integer	For details, see the CommandLine table.	N/A	Output commands by line.

Sample Request and Sample Response

Python invoking example:

```
def yangToCli(self, aoccontext, request):
    res_data = None

    # Obtain the request path and process services based on the path.
    if request.path.startswith("/path1"):
        # Perform service processing based on the request operation type.
        if request.oper_type == OperType.Value('CREATE'):
            res_data = 'c1 leaf1 value1'
        elif request.oper_type == OperType.Value('MERGE'):
            res_data = 'c1 leaf1 value2'
        elif request.oper_type == OperType.Value('DELETE'):
            res_data = 'undo c1 leaf1'

    # Construct the return parameter.
    out = CliCustomTransformOutput()
    out.data = res_data
    return out
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.4.3 CLI-to-YANG Conversion

Type

API-4: device data query interface and operation interface

Typical Scenarios

If the conversion logic of the CLI driver is not supported, the customized CLI-to-YANG conversion method needs to be used to resolve the conversion problem.

Functions

Customize the CLI-to-YANG conversion logic.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def cliToYang(self, aoccontext, request):
```



```
"""
Customize the CLI-to-YANG conversion logic.
:param aoccontext: context
:param request: input parameter for conversion
:return: conversion result
"""
pass
```

Request Parameters

Table 2-90 CliCustomTransformInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
path	Yes	STRING	N/A	N/A	Request path.
data	Yes	STRING	N/A	N/A	Request data.
oper_type	Yes	OperType	For details, see the OperType table.	N/A	Operation type of a request.
startOffset	Yes	Integer	≥ 0	N/A	Offset of a request.
neId	Yes	STRING	N/A	N/A	ID of the requested NE.
commandLines	No	CommandLine	For details, see the CommandLine table.	N/A	Requested CLI command.

Table 2-91 OperType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	READ_CFG READ_OPR CREATE MERGE DELETE RPC	N/A	Operation type.

Table 2-92 CommandLine

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
index	No	Integer	≥ 0	N/A	Serial number of a CLI command.
commandLine	No	STRING	N/A	N/A	Content of a CLI command.

Table 2-93 CliCustomTransformOutput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
data	Yes	STRING	N/A	N/A	Modified data.
endOffset	Yes	Integer	≥ -1	N/A	End offset.
commandLines	Yes	Integer	For details, see the CommandLine table.	N/A	Output commands by line.

Sample Request and Sample Response

Python invoking example:

```
def cliToYang(self, aoccontext, request):
    out = CliCustomTransformOutput()

    # Obtain the request path and process services based on the path.
    if request.path.startswith("/cli-custom:c1"):
        out.data = "<c1 xmlns=\"http://huawei.com/cli-custom\"><f1>hello_world</f1></c1>"
        out.endOffset = str(request.data).__len__()
    return out
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.4.4 Pre-processing

Type

API-4: device data query interface and operation interface

Typical Scenarios

After a device returns a configuration packet, the configuration packet can be processed before being sent to the CLI driver.

Functions

The CLI driver performs CLI-to-YANG conversion based on the YANG model by default. However, in some special cases, the CLI commands returned by the device cannot be directly used as the conversion input parameters of the CLI driver and need to be modified for CLI-to-YANG conversion.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: The annotation extension item must be added to the top-level YANG node.
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def readCliPreProcessor(self, aoccontext, request):
    """
    Customize the pre-processing method.
    :param aoccontext: context
```



```
    :param request: input parameter for conversion
     :return: conversion result
    """
    pass
```

Request Parameters

Table 2-94 CliCustomTransformInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
path	Yes	STRING	N/A	N/A	Request path.
data	Yes	STRING	N/A	N/A	Request data.
oper_type	Yes	OperType	For details, see the OperType table.	N/A	Operation type of a request.
startOffset	Yes	Integer	≥ 0	N/A	Offset of a request.
neId	Yes	STRING	N/A	N/A	ID of the requested NE.
commandLines	No	CommandLine	For details, see the CommandLine table.	N/A	Requested CLI command.

Table 2-95 OperType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	READ_CFG READ_OPR CREATE MERGE DELETE RPC	N/A	Operation type.

Table 2-96 CommandLine

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
index	No	Integer	≥ 0	N/A	Serial number of a CLI command.
commandLine	No	STRING	N/A	N/A	Content of a CLI command.

Table 2-97 CliCustomTransformOutput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
data	Yes	STRING	N/A	N/A	Modified data.
endOffset	Yes	Integer	≥ -1	N/A	End offset.
commandLines	Yes	Integer	For details, see the CommandLine table.	N/A	Output commands by line.

Sample Request and Sample Response

Python invoking example:

```
def readCliPreProcessor(self, aoccontext, request):
    # Obtain the CLI commands returned by the device.
    req_data = str(request.data)

    # Customize the CLI commands.
    if "c1 leaf1 value1" in req_data:
        req_data.replace("c1 leaf1 value1", "c1 leaf1 value2")

    # Construct the return parameter.
    out = CliCustomTransformOutput()
    out.data = req_data
    return out
```


Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.4.5 Post-processing

Type

API-4: device data query interface and operation interface

Typical Scenarios

After the CLI driver generates CLI commands that need to be delivered to a device, the commands can be processed by the CLI driver.

Functions

The CLI driver performs default YANG-to-CLI conversion based on the YANG model. However, in some special cases, the CLI commands generated by the CLI driver cannot meet the device requirements and need to be modified to meet the device requirements.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: The annotation extension item must be added to the top-level YANG node.
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def configCliPostProcessor(self, aoccontext, request):
    """
     Customized post-processing
    :param aoccontext: context
    :param request: input parameter for conversion
    :return: conversion result
    """
    pass
```


Request Parameters

Table 2-98 CliCustomTransformInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
path	Yes	STRING	N/A	N/A	Request path.
data	Yes	STRING	N/A	N/A	Request data.
oper_type	Yes	OperType	For details, see the OperType table.	N/A	Operation type of a request.
startOffset	Yes	Integer	≥ 0	N/A	Offset of a request.
neId	Yes	STRING	N/A	N/A	ID of the requested NE.
commandLines	No	CommandLine	For details, see the CommandLine table.	N/A	Requested CLI command.

Table 2-99 OperType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	READ_CFG READ_OPR CREATE MERGE DELETE RPC	N/A	Operation type.

Table 2-100 CommandLine

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
index	No	Integer	≥ 0	N/A	Serial number of a CLI command.
commandLine	No	STRING	N/A	N/A	Content of a CLI command.

Table 2-101 CliCustomTransformOutput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
data	Yes	STRING	N/A	N/A	Modified data.
endOffset	Yes	Integer	≥ -1	N/A	End offset.
commandLines	Yes	Integer	For details, see the CommandLine table.	N/A	Output commands by line.

Sample Request and Sample Response

Python invoking example:

```
def configCliPostProcessor(self, aoccontext, request):
    # Obtain the CLI data that requires post-processing.
   req_data = request.data
   req_path = request.path

   # Obtain the request path and process data based on the path.
   if req_path == '/path':
      req_data = req_data + 'a'
   else:
      req_data = req_data + 'b'

   # Construct the return parameter.
   out = CliCustomTransformOutput()
   out.data = req_data
   return out
```


Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.4.6 Post-Processing by Line

Type

API-4: device data query interface and operation interface

Typical Scenarios

After the CLI driver generates CLI commands that need to be delivered to a device, the commands can be processed by the CLI driver by line. If the device does not support two-phase transactions, the CLI driver can roll back the device.

Functions

On the basis of post-processing, roll back devices that do not support transactions.

Constraints

1. Prerequisites: N/A
2. Precautions: This method can be used to automatically roll back devices that do not support transactions. Do not change the number of commands and the command sequence.
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def configCliByLinePostProcessor(self, aoccontext, request):
    """
    Customize the logic for post-processing by line
    :param aoccontext: context
    :param request: input parameter for conversion
    :return: conversion result
    """
    pass
```


Request Parameters

Table 2-102 CliCustomTransformInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
path	Yes	STRING	N/A	N/A	Request path.
data	Yes	STRING	N/A	N/A	Request data.
oper_type	Yes	OperType	For details, see the OperType table.	N/A	Operation type of a request.
startOffset	Yes	Integer	≥ 0	N/A	Offset of a request.
neId	Yes	STRING	N/A	N/A	ID of the requested NE.
commandLines	No	CommandLine	For details, see the CommandLine table.	N/A	Requested CLI command.

Table 2-103 OperType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	READ_CFG READ_OPR CREATE MERGE DELETE RPC	N/A	Operation type.

Table 2-104 CommandLine

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
index	No	Integer	≥ 0	N/A	Serial number of a CLI command.
commandLine	No	STRING	N/A	N/A	Content of a CLI command.

Table 2-105 CliCustomTransformOutput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
data	Yes	STRING	N/A	N/A	Modified data.
endOffset	Yes	Integer	≥ -1	N/A	End offset.
commandLines	Yes	Integer	For details, see the CommandLine table.	N/A	Output commands by line.

Sample Request and Sample Response

Python invoking example:

```
def configCliByLinePostProcessor(self, aoccontext, request):
    out = CliCustomTransformOutput()

    # Obtain the request path and process services based on the path.
    req_path = request.path
    if req_path == '/path1':
        # Create a return CLI command object for each request CLI command object.
        for item in request.commandLines:
            command = out.commandLines.add()

            # Set the index of a CLI command object.
            command.index = item.index

            # Customize the CLI command.
            if 'a' in item.commandLine:
                command.commandLine = item.commandLine.replace('a', 'b')
    return out
```


2.3.4.7 Post-Processing for Rollback Commands

Type

API-4: device data query interface and operation interface

Typical Scenarios

For devices involved in one-phase transactions, if a command generated by the CLIDriver fails to be delivered, the CLIDriver rollback mechanism is automatically triggered to generate a rollback command. If the automatically generated rollback command does not meet the device rollback requirements, you can perform post-processing on the generated rollback command.

Functions

Performs post-processing on the generated rollback command.

Constraints

1. Prerequisites: Configurations fail to be delivered to devices involved in one-phase transactions, and automatic rollback is triggered.
2. Precautions: N/A
3. Usage restrictions: The corresponding configuration delivery command meets automatic rollback conditions of the CLIDriver.
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def configRollbackCliPostProcessor(self, aoccontext, request):
    """
    Customize the logic for post-processing by line
    :param aoccontext: context
    :param request: input parameter for conversion
    :return: conversion result
    """
    pass
```

Request Parameters

Table 2-106 CliCustomTransformInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
path	Yes	STRING	N/A	N/A	Request path.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
data	Yes	STRING	N/A	N/A	Request data.
oper_type	Yes	OperType	For details, see the OperType table.	N/A	Operation type of a request.
startOffset	Yes	Integer	≥ 0	N/A	Offset of a request.
neId	Yes	STRING	N/A	N/A	ID of the requested NE.
commandLines	No	CommandLine	For details, see the CommandLine table.	N/A	Requested CLI command.

Table 2-107 OperType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	READ_CFG READ_OPR CREATE MERGE DELETE RPC	N/A	Operation type.

Table 2-108 CommandLine

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
index	No	Integer	≥ 0	N/A	Serial number of a CLI command.
commandLine	No	STRING	N/A	N/A	Content of a CLI command.

Table 2-109 CliCustomTransformOutput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
data	Yes	STRING	N/A	N/A	Modified data.
endOffset	Yes	Integer	≥ –1	N/A	End offset.
commandLines	Yes	Integer	For details, see the CommandLine table.	N/A	Output commands by line.

Sample Request and Sample Response

Python invoking example:

```
def configRollbackCliPostProcessor(self, aoccontext, request):
    out = CliCustomTransformOutput()
    # Obtain the request path and process services based on the path.
    req_path = request.path
    if req_path == '/path1':
        out.data = 'no interface aaa'
    return out
```

2.3.5 SND RESTCONF Framework Customization

2.3.5.1 YANG-to-RESTCONF Conversion for RPC Operations

Type

API-4: device data query interface and operation interface

Typical Scenarios

If the default conversion logic of the RestconfDriver is not supported, the customized YANG-to-RESTCONF conversion method needs to be used to resolve the RPC operation conversion problem.

Functions

Customize the YANG-to-RESTCONF conversion logic for RPC operations.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def rpcCustomYangToRestconf(self, aoccontext, request):
    """
    Customize the YANG-to-RESTCONF conversion logic for RPC operations.
    :param aoccontext: context
    :param request: input parameter for conversion
    :return: conversion result
    """
    pass
```

Request Parameters

Table 2-110 RpcRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neId	Yes	STRING	N/A	N/A	Device ID.
rpcQName	Yes	STRING	N/A	N/A	QName of an RPC request.
input	Yes	STRING	N/A	N/A	Input parameter of an RPC request.

Table 2-111 RestconfRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
url	Yes	STRING	N/A	N/A	Request URL.
requestBody	Yes	STRING	N/A	N/A	Request packet.
HttpMethod	Yes	HttpMethod	For details, see the HttpMethod table.	N/A	Request method.
parameter	No	Map<string,string>	N/A	N/A	GET request parameter.
restfulClient	No	Boolean	N/A	N/A	Whether to invoke the RESTful interface.

Table 2-112 HttpMethod

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	GET POST PUT DELETE PATCH	N/A	Request method.

Sample Request and Sample Response

Python invoking example:

```
def rpcCustomYangToRestconf(self, aoccontext, request):
    url = None
    httpMethod = None

    # Obtain the request path and process services based on the path.
    if request.rpcQName == "/path1":
        url = '/sample/url_1'
        httpMethod = POST

    # Construct the return parameter.
    requestInfo = RestconfRequestInfo()
    requestInfo.url = url
    requestInfo.httpMethod = httpMethod
    return requestInfo
```


Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.5.2 RESTCONF-to-YANG Conversion for RPC Operations

Type

API-4: device data query interface and operation interface

Typical Scenarios

If the default conversion logic of the RestconfDriver is not supported, the customized RESTCONF-to-YANG conversion method needs to be used to resolve the RPC operation conversion problem.

Functions

Customize the RESTCONF-to-YANG conversion logic for RPC operations.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def rpcCustomRestconfToYang(self, aoccontext, input):
    """
    RESTCONF-to-YANG conversion logic customized for RPC operations: restconf:custom-restconf-to-yang
'rpc'
    :param aoccontext: context
    :param input: input parameter for RESTCONF-to-YANG conversion
    :return: YANG data corresponding to the output
    """
    pass
```


Request Parameters

Table 2-113 RpcCustomRestconfToYangInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
rpcRequestInfo	Yes	RpcRequestInfo	For details, see the RpcRequestInfo table.	N/A	RPC request information.
restconfReponseInfo	Yes	RestconfReponseInfo	For details, see the RestconfReponseInfo table.	N/A	RESTCONF response information.

Table 2-114 RpcRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neId	Yes	STRING	N/A	N/A	Device ID.
rpcQName	Yes	STRING	N/A	N/A	QName of an RPC request.
input	Yes	STRING	N/A	N/A	Input parameter of an RPC request.

Table 2-115 RestconfReponseInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
responseBody	Yes	STRING	N/A	N/A	Response packet.

Table 2-116 RpcReponseInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
ouput	Yes	STRING	N/A	N/A	RPC output packet, in XML format.

Sample Request and Sample Response

Python invoking example:

```
def rpcCustomRestconfToYang(self, aoccontext, input):
    rpcRequestInfo = input.rpcRequestInfo
    output = None
    if rpcRequestInfo.rpcQname == 'testqname':
        output = "<xml></xml>"
    response = RpcResponseInfo()
    response.output = output
    return response
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.5.3 Pre-processing for RPC Operations

Type

API-4: device data query interface and operation interface

Typical Scenarios

After an RPC operation returns a response packet, the response packet needs to be processed before being transferred to the RestconfDriver.

Functions

The RestconfDriver performs default RESTCONF-to-YANG conversion based on the YANG model. However, in some special cases, the RESTCONF packets returned by the device cannot be directly used as the conversion input parameters of the RestconfDriver and need to be modified to meet the requirements for RESTCONF-to-YANG conversion.

Constraints

1. Prerequisites: N/A

2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
def rpcCustomPreProcess(self, aoccontext, input):
    """
     Pre-processing logic for RPC operations: restconf:custom-pre-process'rpc'
    :param aoccontext: context
    :param input: RPC request information and information returned by the RESTCONF protocol
    :return: RESTCONF information processed by the SND package
    """
    pass
```

Request Parameters

Table 2-117 RpcCustomPreProcessInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
rpcRequestInfo	Yes	RpcRequestInfo	For details, see the RpcRequestInfo table.	N/A	RPC request information.
restconfReponseInfo	Yes	RestconfReponseInfo	For details, see the RestconfReponseInfo table.	N/A	RESTCONF response information.

Table 2-118 RpcRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neId	Yes	STRING	N/A	N/A	Device ID.
rpcQName	Yes	STRING	N/A	N/A	QName of an RPC request.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
input	Yes	STRING	N/A	N/A	Input parameter of an RPC request.

Table 2-119 RestconfReponseInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
responseBody	Yes	STRING	N/A	N/A	Response packet.
status	Yes	INT	N/A	N/A	Response status.

Sample Request and Sample Response

Python invoking example:

```
def rpcCustomPreProcess(self, aoccontext, input):
    requestInfo = input.rpcReqeustInfo
    respopnseInfo = input.restResponseInfo
    responseBody = None
    if requestInfo.rpcQname == "testqname":
        # Customize the processing logic.
        responseInfo.responseBody = responseBody
    return responseInfo
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.5.4 Post-processing for RPC Operations

Type

API-4: device data query interface and operation interface

Typical Scenarios

After the RestconfDriver generates a request packet that needs to be delivered to a device, the request packet needs to be processed.

Functions

The RestconfDriver performs default YANG-to-RESTCONF conversion based on the YANG model. However, in some special cases, the RESTCONF packets generated by the RestconfDriver cannot meet the device requirements and need to be modified to meet the device requirements.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python invoking example:

```
def rpcCustomPostProcess(self, aoccontext, input):
    """
    Post-processing logic for RPC operations: restconf:custom-post-process'rpc'
    :param aoccontext: context
    :param input: RPC request information and RESTCONF request information generated by the default conversion algorithm of the SND framework mechanism
    :return: RESTCONF request information processed by the SND package
    """
    pass
```

Request Parameters

Table 2-120 RpcCustomPostProcessInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
rpcRequestInfo	Yes	RpcRequestInfo	For details, see the RpcRequestInfo table.	N/A	RPC request information.
restconfRequestInfo	Yes	RestconfRequestInfo	For details, see the RestconfRequestInfo table.	N/A	RESTCONF request information.

Table 2-121 RpcRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neId	Yes	STRING	N/A	N/A	Device ID.
rpcQName	Yes	STRING	N/A	N/A	QName of an RPC request.
input	Yes	STRING	N/A	N/A	Input parameter of an RPC request.

Table 2-122 RestconfRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
url	Yes	STRING	N/A	N/A	Request URL.
requestBody	Yes	STRING	N/A	N/A	Request packet.
HttpMethod	Yes	HttpMethod	For details, see the HttpMethod table.	N/A	Request method.
parameter	No	Map<string,string>	N/A	N/A	GET request parameter.
restfulClient	No	Boolean	N/A	N/A	Whether to invoke the RESTful interface.

Table 2-123 HttpMethod

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	GET POST PUT DELETE PATCH	N/A	Request method.

Sample Request and Sample Response

Python invoking example:

```
def rpcCustomPostProcess(self, aoccontext, input):
    requestInfo = input.rpcRequestInfo
    restconfRequestInfo = input.restconfRequestInfo
    requestBody = None
    httpMethod = None
    if requstInfo.rpcQname == "testqname":
        # Customize the processing logic.
        restconfRequestInfo.requestBody = requestBody
        restconfRequestInfo.httpMethod = httpMethod
    return restconfRequest
```

Error Code

Error Code	Error Message	Solution
N/A	N/A	N/A

2.3.5.5 YANG-to-RESTCONF Conversion for the Read Operation

Type

API-4: device data query interface and operation interface

Typical Scenarios

If the default conversion logic of the RestconfDriver is not supported, the customized YANG-to-RESTCONF conversion method needs to be used to resolve the query operation conversion problem.

Functions

Customize the YANG-to-RESTCONF conversion logic for query operations.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
def readCustomYangToRestconf(self, aoccontext, request):
    """
     * YANG-to-RESTCONF conversion logic customized for the read operation: restconf:custom-yang-to-
restconf 'read'
    :param aoccontext: context
    :param request: input parameter of the read operation
    :return: output parameter of the read operation
    """
    pass
```

Request Parameters

Table 2-124 ReadRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neId	Yes	STRING	N/A	N/A	Device ID.
path	Yes	STRING	N/A	N/A	Query path.
properties	No	Map<STRING,STRING>	N/A	N/A	Query parameters.
logicalDatastor-eType	Yes	LogicalDatastoreType	For details, see the LogicalDatastoreType table.	N/A	Logical data type.

Table 2-125 LogicalDatastoreType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	OPERATIONAL CONFIGURATION	N/A	Logical data type.

Table 2-126 RestconfRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
url	Yes	STRING	N/A	N/A	Request URL.
requestBody	Yes	STRING	N/A	N/A	Request packet.
HttpMethod	Yes	HttpMethod	For details, see the HttpMethod table.	N/A	Request method.
parameter	No	Map<string,string>	N/A	N/A	GET request parameter.
restfulClient	No	Boolean	N/A	N/A	Whether to invoke the RESTful interface.

Table 2-127 HttpMethod

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	GET POST PUT DELETE PATCH	N/A	Request method.

Sample Request and Sample Response

Python invoking example:

```
def readCustomYangToRestconf(self, aoccontext, request):
    restconfRequestInfo = RestconfRequest()
    url = None
    if request.path == "testreadpath":
        # Customize the conversion logic.
        restconfRequestInfo.url = url
    return restconfRequestInfo
```

2.3.5.6 RESTCONF-to-YANG Conversion for the Read Operation

Type

API-4: device data query interface and operation interface

Typical Scenarios

If the default conversion logic of the RestconfDriver is not supported, the customized RESTCONF-to-YANG conversion method needs to be used to resolve the query operation conversion problem.

Functions

Customize the RESTCONF-to-YANG conversion logic for the query operation.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def readCustomRestconfToYang(self, aoccontext, input):
    """
    RESTCONF-to-YANG conversion logic customized for the read operation: restconf:custom-restconf-to-
yang 'read'
    :param aoccontext: context
    :param input: input parameter of the read operation
    :return: output parameter of the read operation
    """
    pass
```


Request Parameters

Table 2-128 ReadCustomRestconfToYangInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
readRequestInfo	Yes	ReadRequestInfo	For details, see the ReadRequestInfo table.	N/A	Read request information.
restconfReponseInfo	Yes	RestconfReponseInfo	For details, see the RestconfReponseInfo table.	N/A	RESTCONF response information.

Table 2-129 ReadRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neId	Yes	STRING	N/A	N/A	Device ID.
path	Yes	STRING	N/A	N/A	Query path.
properties	No	Map<STRING,STRING>	N/A	N/A	Query parameters.
logicalDatastoreType	Yes	LogicalDatastoreType	For details, see the LogicalDatastoreType table.	N/A	Logical data type.

Table 2-130 LogicalDatastoreType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	OPERATIONAL CONFIGURATION	N/A	Logical data type.

Table 2-131 RestconfReponseInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
responseBody	Yes	STRING	N/A	N/A	Response packet.

Table 2-132 ReadReponseInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
data	Yes	STRING	N/A	N/A	Query result, in XML format.

Sample Request and Sample Response

Python invoking example:

```
def readCustomRestconfToYang(self, aoccontext, input):
    rpcRequestInfo = input.readRequestInfo
    data = None
    if rpcRequestInfo.path == 'testreadpath':
        data = "<xml></xml>"
    response = ReadReponseInfo()
    response.data= data
    return response
```

2.3.5.7 Pre-processing for the Read Operation

Type

API-4: device data query interface and operation interface

Typical Scenarios

After the read operation returns a response packet, the packet needs to be processed before being transferred to the RestconfDriver.

Functions

The RestconfDriver performs default RESTCONF-to-YANG conversion based on the YANG model. However, in some special cases, the RESTCONF packets returned by the device cannot be directly used as the conversion input parameters of the RestconfDriver and need to be modified to meet the requirements for RESTCONF-to-YANG conversion.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def readCustomPreProcess(self, aoccontext, input):
    """
    Pre-processing logic for the read operation: restconf:custom-pre-process 'read'
    :param aoccontext: context
    :param input: input parameter of the read operation
    :return: output parameter of the read operation
    """
    pass
```

Request Parameters

Table 2-133 ReadCustomPreProcessInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
readRequestInfo	Yes	ReadRequestInfo	For details, see the ReadRequestInfo table.	N/A	Read request information.
restconfReponseInfo	Yes	RestconfReponseInfo	For details, see the RestconfReponseInfo table.	N/A	RESTCONF response information.

Table 2-134 ReadRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neId	Yes	STRING	N/A	N/A	Device ID.
path	Yes	STRING	N/A	N/A	Query path.
properties	No	Map<STRING,STRING>	N/A	N/A	Query parameters.
logicalDatastoreType	Yes	LogicalDatastoreType	For details, see the LogicalDatastoreType table.	N/A	Logical data type.

Table 2-135 LogicalDatastoreType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	OPERATIONAL CONFIGURATION	N/A	Logical data type.

Table 2-136 RestconfReponseInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
responseBody	Yes	STRING	N/A	N/A	Response packet.
status	Yes	INT	N/A	N/A	Response status.

Sample Request and Sample Response

Python invoking example:

```
def readCustomPreProcess(self, aoccontext, input):
    readRequestInfo = input.readRequestInfo
    restconfResponse = input.restconfResponse
    responseBody = restconfResponse.responseBody
```



```
    if readRequestInfo .path == 'testpath':
        # Customize the processing logic.
        restconfResponse.responseBody = responseBody
    return restconfResponse
```

2.3.5.8 Post-processing for the Read Operation

Type

API-4: device data query interface and operation interface

Typical Scenarios

After the RestconfDriver generates a request packet that needs to be delivered to a device, the request packet needs to be processed.

Functions

The RestconfDriver performs default YANG-to-RESTCONF conversion based on the YANG model. However, in some special cases, the RESTCONF packets generated by the RestconfDriver cannot meet the device requirements and need to be modified to meet the device requirements.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def readCustomPostProcess(self, aoccontext, input):
    """
    Post-processing logic for the read operation: restconf:custom-post-process'read'
    :param aoccontext: context
    :param input: input parameter for post-processing
    :return: output parameter for post-processing
    """
    pass
```


Request Parameters

Table 2-137 ReadCustomPostProcessInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
readRequestInfo	Yes	ReadRequestInfo	For details, see the LogicalDatastoreType table.	N/A	Read request information.
restconfRequestInfo	Yes	RestconfRequestInfo	For details, see the LogicalDatastoreType table.	N/A	RESTCONF request information.

Table 2-138 ReadRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neId	Yes	STRING	N/A	N/A	Device ID.
path	Yes	STRING	N/A	N/A	Query path.
properties	No	Map<STRING,STRING>	N/A	N/A	Query parameters.
logicalDatastoreType	Yes	LogicalDatastoreType	For details, see the LogicalDatastoreType table.	N/A	Logical data type.

Table 2-139 LogicalDatastoreType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	OPERATIONAL CONFIGURATION	N/A	Logical data type.

Table 2-140 RestconfRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
url	Yes	STRING	N/A	N/A	Request URL.
requestBody	Yes	STRING	N/A	N/A	Request packet.
HttpMethod	Yes	HttpMethod	For details, see the HttpMethod table.	N/A	Request method.
parameter	No	Map<string,string>	N/A	N/A	GET request parameter.
restfulClient	No	Boolean	N/A	N/A	Whether to invoke the RESTful interface.

Table 2-141 HttpMethod

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	GET POST PUT DELETE PATCH	N/A	Request method.

Sample Request and Sample Response

Python invoking example:

```
def readCustomPostProcess(self, aoccontext, input):
    readReqeustInfo = input.readRequestInfo
    restconfRequestInfo = input.restconfRequestInfo
    url = restconfRequestInfo.url
    httpMethod = restconfRequestInfo.httpMethod
    if readRequestInfo.path == 'testpath':
        # Customize the conversion logic.
        restconfRequestInfo.url = url
        restconfRequestInfo.httpMethod = httpMethod
    return restconfRequestInfo
```

2.3.5.9 YANG-to-RESTCONF Conversion for the CONFIG Operation

Type

API-4: device data query interface and operation interface

Typical Scenarios

If the default conversion logic of RestconfDriver is not supported, the customized YANG-to-RESTCONF conversion method needs to be used to resolve the CONFIG operation conversion problem.

Functions

Customize the YANG-to-RESTCONF conversion logic for the CONFIG operation.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def configCustomYangToRestconf(self, aoccontext, request):
    """
     * YANG-to-RESTCONF conversion logic customized for the CONFIG operation: restconf:custom-yang-to-
restconf 'merge,replace,create,delete'
    :param aoccontext: context
    :param request: input parameter of the CONFIG operation
    :return: output parameter of the CONFIG operation
    """
    pass
```


Request Parameters

Table 2-142 ConfigRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neId	Yes	STRING	N/A	N/A	Device ID.
path	Yes	STRING	N/A	N/A	Configuration path.
data	No	STRING	N/A	N/A	Configuration packets, in XML format.
requestType	Yes	RequestType	For details, see the RequestType table.	N/A	Operation type.
logicalDatastor-eType	Yes	LogicalDatastoreType	For details, see the LogicalDatastoreType table.	N/A	Logical data type.

Table 2-143 RequestType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	CREATE MERGE REPLACE DELETE	N/A	Operation type.

Table 2-144 LogicalDatastoreType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	OPERATIONAL CONFIGURATION	N/A	Logical data type.

Table 2-145 RestconfRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
url	Yes	STRING	N/A	N/A	Request URL.
requestBody	Yes	STRING	N/A	N/A	Request packet.
HttpMethod	Yes	HttpMethod	For details, see the HttpMethod table.	N/A	Request method.
parameter	No	Map<string,string>	N/A	N/A	GET request parameter.
restfulClient	No	Boolean	N/A	N/A	Whether to invoke the RESTful interface.

Table 2-146 HttpMethod

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	GET POST PUT DELETE PATCH	N/A	Request method.

Sample Request and Sample Response

Python invoking example:

```
def configCustomYangToRestconf(self, aoccontext, request):
    url = None
    httpMethod = None
    requestBody = None
    restconfRequestInfo = RestconfRequestInfo()
    if request.path == 'testconfigpath':
        # Customize the conversion logic.
        restconfRequestInfo.url = url
        restconfRequestInfo.httpMethod = httpMethod
        restconfRequestInfo.requestBody = requestBody
    else:
        # Other processing logic
    return restconfRequestInfo
```

2.3.5.10 RESTCONF-to-YANG Conversion for the CONFIG Operation

Type

API-4: device data query interface and operation interface

Typical Scenarios

If the default conversion logic of RestconfDriver is not supported, the customized RESTCONF-to-YANG conversion method needs to be used to resolve the CONFIG operation conversion problem.

Functions

Customize the RESTCONF-to-YANG conversion logic for the CONFIG operation.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def configCustomRestconfToYang(self, aoccontext, input):
    """
    * RESTCONF-to-YANG conversion logic customized for the CONFIG operation: restconf:custom-restconf-
to-yang 'merge,replace,create,delete'
     :param aoccontext: context
    :param input: input parameter of the CONFIG operation
    :return: output parameter of the CONFIG operation
    """
    pass
```


Request Parameters

Table 2-147 ConfigCustomRestconfToYangInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
configRequestInfo	Yes	ConfigRequestInfo	For details, see the ConfigRequestInfo table.	N/A	Configuration request information.
restconfReponseInfo	Yes	RestconfReponseInfo	For details, see the RestconfReponseInfo table.	N/A	RESTCONF response information.

Table 2-148 ConfigRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neId	Yes	STRING	N/A	N/A	Device ID.
path	Yes	STRING	N/A	N/A	Configuration path.
data	No	STRING	N/A	N/A	Configuration packets, in XML format.
requestType	Yes	RequestType	For details, see the RequestType table.	N/A	Operation type.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
logicalDatastor-eType	Yes	LogicalDatastoreType	For details, see the LogicalDatastoreType table.	N/A	Logical data type.

Table 2-149 RequestType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	CREATE MERGE REPLACE DELETE	N/A	Operation type.

Table 2-150 LogicalDatastoreType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	OPERATIONAL CONFIGURATION	N/A	Logical data type.

Table 2-151 RestconfReponseInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
responseBody	Yes	STRING	N/A	N/A	Response packet.

Table 2-152 ConfigResponseInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	N/A	N/A	N/A	N/A	N/A

Sample Request and Sample Response

Python invoking example:

```
def configCustomRestconfToYang(self, aoccontext, input):
    configRequestInfo = input.configRequestInfo
    restconfResponseIno = input.restconfReponseInfo
    responseBody = restconfResponseInfo.responseBody
    configReponseInfo = ConfigReponseInfo()
    if 'testconfigpath' == configRequestInfo.path:
        # Customize the conversion logic.
       configReponseInfo.responseBody = responseBody
    return configReponseInfo
```

2.3.5.11 Pre-processing for the CONFIG Operation

Type

API-4: device data query interface and operation interface

Typical Scenarios

After the CONFIG operation returns a response packet, the packet needs to be processed before being transferred to the RestconfDriver.

Functions

The RestconfDriver performs default RESTCONF-to-YANG conversion based on the YANG model. However, in some special cases, the RESTCONF packets returned by the device cannot be directly used as the conversion input parameters of the RestconfDriver and need to be modified to meet the requirements for RESTCONF-to-YANG conversion.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def configCustomPreProcess(self, aoccontext, input):
    """
  Pre-processing logic for the CONFIG operation: restconf:custom-pre-process 'merge,replace,create,delete'
```



```
 :param aoccontext: context
:param input: input parameter of the CONFIG operation
 :return: output parameter of the CONFIG operation
"""
pass
```

Request Parameters

Table 2-153 ConfigCustomPreProcessInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
configRequestInfo	Yes	ConfigRequestInfo	For details, see the ConfigRequestInfo table.	N/A	Read request information.
restconfReponseInfo	Yes	RestconfReponseInfo	For details, see the RestconfReponseInfo table.	N/A	RESTCONF response information.

Table 2-154 ConfigRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neId	Yes	STRING	N/A	N/A	Device ID.
path	Yes	STRING	N/A	N/A	Configuration path.
data	No	STRING	N/A	N/A	Configuration packets, in XML format.
requestType	Yes	RequestType	For details, see the RequestType table.	N/A	Operation type.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
logicalDatastor-eType	Yes	LogicalDatastoreType	For details, see the LogicalDatastoreType table.	N/A	Logical data type.

Table 2-155 RequestType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	CREATE MERGE REPLACE DELETE	N/A	Operation type.

Table 2-156 LogicalDatastoreType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	OPERATIONAL CONFIGURATION	N/A	Logical data type.

Table 2-157 RestconfReponseInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
responseBody	Yes	STRING	N/A	N/A	Response packet.
status	Yes	INT	N/A	N/A	Response status.

Sample Request and Sample Response

Python invoking example:

```
def configCustomPreProcess(self, aoccontext, input):
    configRequestInfo = input.configRequestInfo
    restconfResponseInfo = input.restconfResponseInfo
    responseBody = restconfResponseInfo.responseBody
    if configRequestInfo.path == 'testconfigpath':
        # Customize the processing logic.
        restconfResponseInfo.responseBody = responseBody
    return restconfResponseInfo
```

2.3.5.12 Post-processing for the CONFIG Operation

Type

API-4: device data query interface and operation interface

Typical Scenarios

After the RestconfDriver generates a request packet that needs to be delivered to a device, the request packet needs to be processed.

Functions

The RestconfDriver performs default YANG-to-RESTCONF conversion based on the YANG model. However, in some special cases, the RESTCONF packets generated by the RestconfDriver cannot meet the device requirements and need to be modified to meet the device requirements.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def configCustomPostProcess(self, aoccontext, input):
    """
     Post-processing logic for the CONFIG operation: custom-post-process 'merge,replace,create,delete'
    :param aoccontext: context
    :param input: input parameter of the CONFIG operation
    :return: output parameter of the CONFIG operation
    """
    pass
```


Request Parameters

Table 2-158 ConfigCustomPostProcessInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
configRequestInfo	Yes	ConfigRequestInfo	For details, see the ConfigRequestInfo table.	N/A	Read request information.
restconfRequestInfo	Yes	RestconfRequestInfo	For details, see the RestconfRequestInfo table.	N/A	RESTCONF request information.

Table 2-159 ConfigRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
neId	Yes	STRING	N/A	N/A	Device ID.
path	Yes	STRING	N/A	N/A	Configuration path.
data	No	STRING	N/A	N/A	Configuration packets, in XML format.
requestType	Yes	RequestType	For details, see the RequestType table.	N/A	Operation type.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
logicalDatastor-eType	Yes	LogicalDatastoreType	For details, see the LogicalDatastoreType table.	N/A	Logical data type.

Table 2-160 RequestType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	CREATE MERGE REPLACE DELETE	N/A	Operation type.

Table 2-161 LogicalDatastoreType

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	OPERATIONAL CONFIGURATION	N/A	Logical data type.

Table 2-162 RestconfRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
url	Yes	STRING	N/A	N/A	Request URL.
requestBody	Yes	STRING	N/A	N/A	Request packet.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
HttpMethod	Yes	HttpMethod	For details, see the Table HttpMethod table.	N/A	Request method.
parameter	No	Map<string,string>	N/A	N/A	GET request parameter.
restfulClient	No	Boolean	N/A	N/A	Whether to invoke the RESTful interface.

Table 2-163 HttpMethod

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	GET POST PUT DELETE PATCH	N/A	Request method.

Sample Request and Sample Response

Python invoking example:

```
def configCustomPostProcess(self, aoccontext, input):
    configRequestInfo = input.configReqeustInfo
    restconfRequestInfo = input.restconfRequestInfo
    url = restconfRequestInfo.url
    httpMethod = restconfRequestInfo.httpMethod
    requestBody = restconfRequestInfo.requestBody
    if configRequestInfo.path == 'testconfigpath':
        # Customize the processing logic.
        restconfRequestInfo.url = url
        restconfRequestInfo.httpMethod = httpMethod
        restconfRequestInfo.requestBody = requestBody
    return restconfRequestInfo
```

2.3.5.13 Pre-processing for Abnormal Packets

Type

API-4: device data query interface and operation interface

Typical Scenarios

When a device returns abnormal packets, the abnormal packets need to be pre-processed.

Functions

After obtaining an error response from a device, the RestconfDriver obtains the error information from the response packet. If the error information cannot be obtained from the packet returned by the device using the default method, the returned packet needs to be processed so that the RestconfDriver can parse it. In this way, the corresponding error information can be displayed on the page.

Constraints

1. Prerequisites: N/A
2. Precautions: N/A
3. Usage restrictions: N/A
4. Relationships between APIs: N/A

Invoking Method

Python-based interface invoking method:

```
@abstractmethod
def errorCustomPreProcess(self, aoccontext, input):
    """
     Pre-processing for abnormal packets: restconf:custom-pre-process 'error'
    :param aoccontext: context
    :param input: input parameter of an operation on an abnormal packet
    :return: output parameter of an operation on an abnormal packet
    """
    pass
```

Request Parameters

Table 2-164 ErrorCustomPreProcessInput

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
restconfRequestInfo	Yes	RestconfRequestInfo	For details, see the RestconfRequestInfo table.	N/A	RESTCONF request information.

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
RestconfResponseInfo	Yes	RestconfResponseInfo	For details, see the RestconfReponseInfo table.	N/A	RESTCONF response information.

Table 2-165 RestconfRequestInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
url	Yes	STRING	N/A	N/A	Request URL.
requestBody	Yes	STRING	N/A	N/A	Request packet.
HttpMethod	Yes	HttpMethod	For details, see the HttpMethod table.	N/A	Request method.
parameter	No	Map<string,string>	N/A	N/A	GET request parameter.
restfulClient	No	Boolean	N/A	N/A	Whether to invoke the RESTful interface.

Table 2-166 HttpMethod

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
N/A	Yes	Enumerated	GET POST PUT DELETE PATCH	N/A	Request method.

Table 2-167 RestconfReponseInfo

Parameter (Python)	Mandatory	Category	Value Range	Default Value	Description
responseBody	Yes	STRING	N/A	N/A	Response packet.
status	Yes	INT	N/A	N/A	Response status.

Sample Request and Sample Response

Python invoking example:

```
def errorCustomPreProcess(self, aoccontext, input):
    rpcRequestInfo = input.restconfRequestInfo
    errorMsg= None
     #Customize the processing logic.
    response = RestconfReponseInfo()
    response.responseBody= errorMsg
    return response
```


3 Customized CLI Extension

3.1 Context

To enable CLI Driver to become a general engine rather than a service engine and to reduce unnecessary special codes and compatibility codes, you can develop the following customized extension items to transfer the forward and reverse conversion logic between YANG and CLI to the specific NE driver (SND). You can customize the conversion logic based on the CLI features of different devices through coding. In this way, CLI Driver can flexibly parse the CLI and YANG.

3.2 Extension

3.2.1 cli-custom-transform

Application Scope

container, list, and leaf nodes.

Function

The SND can be used to customize the forward YangToCli conversion logic of the container, list, and leaf nodes marked with **cli-custom-transform**. This extension may involve three parameters, which can be combined with commas (,), namely, create, update, and delete. These parameters specify the operation type for the extension to take effect. If the current node has extension items, operations for three parameters are described as follows.

1. create:

   a. If the path points to the current node, the conversion logic of the current node is provided by the SND.

b. If the path points to the parent node, nodes through which other paths of the parent node pass are not affected. The conversion logic of the current path is provided by the SND.

c. If the path points to a child node, the conversion logic of the child node is provided by the SND.

The following two examples are provided by referring to section **Example**:

- Example 1: In the existing YANG file structure c0/c1/c2/l2, c0/c1/l1, c0/c4/l4, c2 in red has the create extension item, the path points to c1, and c0/c4/l3 is irrelevant to c1. Therefore, c0/c4/l3 does not participate in YangToCli conversion. If both l1 and l2 have values, CLI Driver performs YangToCli conversion because no **cli-custom-transform** is provided for the l1 path. The SND performs YangToCli conversion for l2.
- Example 2: In the existing YANG file structure c0/c1/c2/c3/l3, the c2 and c3 in red have the create extension item, and the path points to l3. CLI Driver checks from c0 from top to bottom. If c2 has **cli-custom-transform**, the customized code corresponding to the c2 path in the SND will be executed, and the codes below c2 will not be parsed. This is because c0 and c1 are parent nodes of the container type, the prefixes are not generated by the CLI, but are provided by the customized logic of c2.

2. delete:
   a. If the path points to the parent node, the node is not affected.
   b. If the path points to the current node or child node, the conversion logic is provided by the SND.
3. update:
   a. If the current node has extension items and the path points to the current node, the conversion logic of the current node is provided by the SND.
   b. If the current node has extension items and the path points to the parent node, nodes through which other paths of the parent node pass are not affected. The conversion logic of the current path is provided by the SND.
   c. If the current node has extension items and the path points to a child node, the conversion logic of the child node is provided by the SND.

Example

```
container c0 {
  container c4 {
    leaf l4 {
      type string;
    }
  }
  container c1 {
    leaf l1 {
      type string;
    }
    container c2 {
    cli:cli-custom-transform 'create,update,delete'
      leaf l2 {
        type string;
      }
      container c3 {
      cli:cli-custom-transform 'create,update,delete'
        leaf l3 {
          type string;
```



```
      }
    }
   }
  }
}
```

In the YANG file, both c2 and c3 are marked with **cli-custom-transform** extension. c3 is the child node of c2, c2 is the child node of c1, c0 is the top-level node, and c1 and c4 are sibling nodes. According to the extended parameters, the forward YangToCli conversion logic is provided by the SND if the operation type is create, update, or delete, and the last node of the path is the parent node of the c2/c3, c2/c3, or the child node of the c2/c3.

Assuming that SND is as follows:

```
if path=/c0/c1/c2       return c1 c2custom  # The c1 prefix is provided by the SND.
if path=/c0/c1/c2/c3    return c1 c2 c3custom  # Front containers need to be provided for c1 and c2.
```

Assume that the path is as follows:

```
path=/c0/c1
```

Analysis: The last node of the path is c1. Therefore, c4 and l4 are not affected. c1 and all its child nodes perform forward YangToCli conversion. There is no **cli-custom-transform** annotation on the path of l1. Therefore, the conversion of l1 is provided by CLI Driver. If c2 has a **cli-custom-transform** annotation, the SND executes the customized conversion logic corresponding to c2, that is, return c1 c2custom. After the c2 logic is provided, the SND does not execute the conversion logic of the c2 child node. That is, c3 is not processed by the SND.

Normal output parameter:

```
c1 l1 l1test
c1 c2 l2 l2test
c1 c2 c3 l3 l3test
```

Customized output parameter:

```
c1 l1 l1test
c1 c2custom
```

```
def yangToCli(self, aoccontext, request):
    out = CliCustomTransformOutput()
    if request.path == "/cli-custom:c1/cli-custom:c2" and request.oper_type == OperType.Value('CREATE'):
        out.data = "c1 c2custom"
    return out
```

The preceding example is the forward customized conversion code of the SND. The code is in the YangToCli method. The correct request is intercepted by matching **request.path** and operation type **request.oper_type**. The **out.data** file stores generated CLIs.

3.2.2 cli-custom-read

Application Scope

container, list, and leaf nodes.

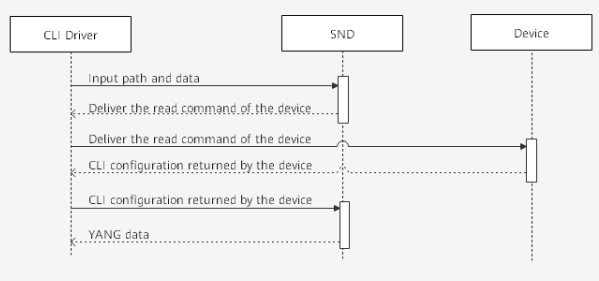
Function

The SND can be used to customize the reverse CliToYang conversion logic of nodes marked with cli-custom-read. This extension may involve three parameters: two-

step, one-step, and one-step-no-parse. Only one of the three parameters can be used at a time. The differences are as follows:

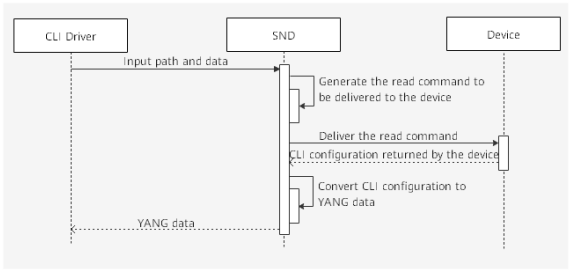
1. **two-step**

   The SND provides two conversion logics for the read operation, that is, forward CLI generation and the reverse device configuration resolution. During the process, CLI Driver interacts with the SND twice. CLI Driver invokes the YangToCli method of the SND to generate the CLI read command delivered to the device. After obtaining configuration data returned by the device, CLI Driver invokes the cliToYang method of the SND to convert the configuration data into YANG data. Then the SND returns the YANG data to CLI Driver.



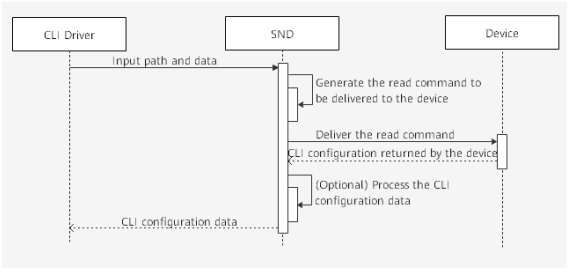
2. **one-step**

   CLI Driver directly sends the path parameter of the read command to the SND, and invokes the CliToYang method of the SND. The SND is responsible for generating the read command to be delivered to the device. After delivering the command to the device and obtaining configuration data returned by the device, the SND converts the data into YANG data, and returns the data to CLI Driver. In the one-step process, CLI Driver interacts with the SND only once, involving only the CliToYang method. In addition, the SND interacts with the device instead of CLI Driver.



3. **one-step-no-parse**

   Similar to one-step, CLI Driver sends the path parameter of the read command to the SND and invokes the CliToYang method of the SND. However, after delivering a command to the device and obtaining configuration data returned by the device, the SND processes the configuration data, and then directly returns the configuration data to CLI Driver. CLI Driver is responsible for CliToYang conversion. The SND is only responsible for generating commands, invoking devices, and obtaining and processing returned configuration data.



NOTE

- **cli-custom-read 'one-step'** and **'one-step-no-parse'** can be added only to the top-level node. They do not take effect on non-top-level nodes.
- When the out object is returned in the CliToYang method, the **endOffset** field must be set, indicating the number of consumed input parameter character strings and the offset value after consumption. For one-step and one-step-no-parse, the value of **endOffset** is -1. In other cases, if all input parameter character stings are consumed, **str(request.data).__len__()** is returned. If not all input parameter character stings are consumed, the value of endOffset is the number of actually consumed character strings plus the value of **startOffset** in the request object.

Example

1. **two-step**

```
leaf router-id {
    cli:cli-custom-read 'two-step';
    type string;
}
```

In the YANG file, the router-id leaf node is marked with **cli-custom-read 'two-step'**. Therefore, the SND is responsible for generating the forward CLI delivery of the read operation of leaf node and converting CLI configuration data returned by the device into YANG data.

```
def yangToCli(self, aoccontext, request):
    out = CliCustomTransformOutput()
    if request.path == "tellabs:router-id":
        out.data = "show running config | block router-id"
    return out
```

The preceding is the code for the SND to generate the customized read command for two-step. The SND matches the required path based on **request.path** and generates the command for delivering the customized read command to the device.

```
def cliToYang(self, aoccontext, request):
    out = CliCustomTransformOutput()
    if request.path == "tellabs:router-id":
        lines = request.data.splitlines()
        if len(lines) > 1:
            router_id_content = str(lines[1]).split()[1]
            out.data = "<router-id xmlns=\"https://huawei.com/tellabs\">" + router_id_content + "</
router-id>"
    out.endOffset = str(request.data).__len__()
    return out
```

The preceding is the operation of converting configuration data returned by the device into YANG data through the CliToYang method of the SND for two-step. The customized code matches the path through **request.path**, obtains device configuration data through **request.data**, processes the obtained router-id data, stores the data in the **router_id_content** variable, assembles YANG data, and returns the data to CLI Driver.

2. **one-step**

```
leaf router-id {
    cli:cli-custom-read 'one-step';
    type string;
}
```

In the YANG file, if the router-id leaf node is marked with **cli-custom-read 'one-step'**, the SND provides the CliToYang conversion logic of the read operation on the leaf node through the CliToYang method.

```
def cliToYang(self, aoccontext, request):
    out = CliCustomTransformOutput()
```



```
    proxy = CommandProxy(request.neId, logger)
    proxy.send_command(["enable", "end", "configure terminal"])
    response_data = proxy.send_command(["show running-config | block router-id"], 120)
    response_body = response_data.response
    lines = response_body.splitlines()
    if len(lines) > 1:
        router_id_content = str(lines[1]).split()[1]
        out.data = "<router-id xmlns=\"https://huawei.com/tellabs\">" + router_id_content + "</router-
id>"
    out.endOffset = -1
    return out
```

In the SND, you can obtain the NE ID from the **neId** attribute of the input request parameter, invoke the **CommandProxy** proxy class to deliver the customized read command to the device, obtain configuration data returned by the device, and convert the configuration data into YANG data. CLI Driver invokes the SND once and completes the customized CliToYang conversion of the read operation through the CliToYang method.

3. **one-step-no-parse**

```
leaf router-id {
    cli:cli-custom-read 'one-step-no-parse';
    type string;
}
```

In the preceding, router-id is marked with **cli-custom-read 'one-step-no-parse'**. Compared with one-step, the SND only returns CLI configuration data through the CliToYang method, without the need to perform CliToYang conversion.

```
def cliToYang(self, aoccontext, request):
    out = CliCustomTransformOutput()
    proxy = CommandProxy(request.neId, logger)
    proxy.send_command(["enable", "end", "configure terminal"])
    response_data = proxy.send_command(["show running-config | block router-id"], 120)
    response_body = response_data.response
    lines = response_body.splitlines()
    if len(lines) > 1:
        router_id_content = str(lines[1]).split()[1]
        out.data = "router-id" + router_id_content
    out.endOffset = -1
    return out
```

Similar to one-step, one-step-no-parse compiles customized code through the CliToYang method. You can deliver the customized read command to the device through the **CommandProxy** proxy class, obtain configuration data returned by the device, process the data, and return the CLI configuration data similar to **router-id 1.2.3.4** to CLI Driver. CLI Driver is responsible for definition according to the YANG model and converting the CLI configuration data into YANG data. one-step-no-parse enables the SND to customize code without CliToYang conversion.

3.2.3 cli-custom-processor

Application Scope

Top-level container and top-level list nodes.

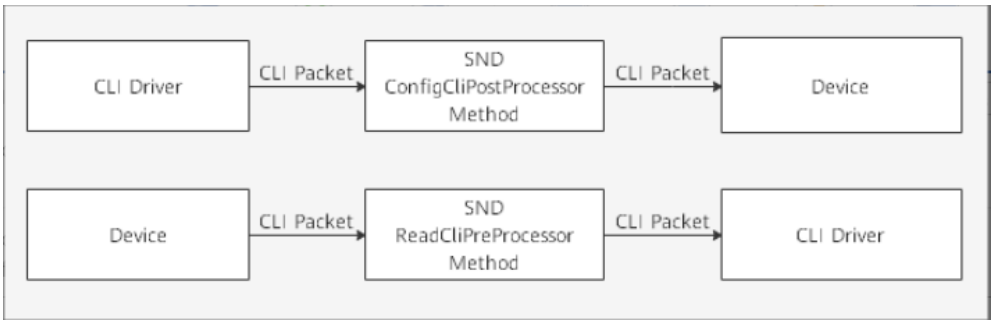
Function

This extension supports pre- and post-processing and involves four parameters: pre-read, post-config, post-config-by-line, and post-config-rollback, which can be separated by commas (,).

- pre-read: processes configuration data before the device sends the configuration data to CLI Driver. Generally, this parameter is used in the read scenario.
- post-config: processes CLI data before CLI Driver sends the CLI data to the device. This parameter is mainly used in the CUD delivery scenario, and is occasionally used in the read command delivery scenario.
- post-config-by-line: performs postprocessing for commands by line. If this parameter is used, the number of lines and sequences of commands in user-defined code cannot be adjusted.
- post-config-rollback: generates rollback commands based on customer requirements, if the default rollback mechanism of the CLIDriver cannot meet the rollback requirements of one-phase devices.

  The process is as follows:

  CLI Driver forward > **CLI** > **(SND) post-processing** > **device** > **(SND) pre-processing** > **Config** > **CLI Driver reverse**

  The annotation is configured on the top-level node. Due to the parent-child inheritance mechanism, if the input parameter path points to any child node, you can enter the SND preprocessing and postprocessing method. You can take different measures based on different paths and input parameters.

Example

YANG file

```
container interfaces {
   cli:cli-custom-processor 'post-config,pre-read';
   list interface {
      container mpls {
         list l2vc {
            key role;
            leaf role {
               typoe enumeration {
                  enum "primary";
                  enum "secondary";
               }
            }
         }
      }
   }
}
```

In the YANG file, top-level node **interfaces** is marked with **cli:cli-custom-processor 'post-config,pre-read'**. If the input parameter path points to the node **interfaces** or its child nodes, the SND performs customized preprocessing or postprocessing for the conversion operation. The current customization requirements are as follows:

- If the value of l2vc in the CLI data contains **primary** during CLI delivery, delete **primary**.
- If the value of l2vc is null during device configuration reading, add **primary** to the proper position of the CLI data.

1. **pre-read**

```
def readCliPreProcessor(self, aoccontext, request):
    out = CliCustomTransformOutput()
    cli_str = str(request.data)
    re_mpls = r'.*(undo\s+)?mpls l2vc.*'
    if re.search(re_mpls, cli_str):
        newlines = []
        lines = cli_str.splitlines()
        for line in lines:
            if re.search(re_mpls, line) and 'secondary' not in line:
                line += ' primary'
            newlines.append(line)
        cli_str = '\n'.join(newlines)
    out.data = cli_str
    return out
```

   As shown in the preceding figure, the customized code of **pre-read** is written in the readCliPreProcessor method. The customized code matches the CLI data returned by the device using the regular expression. If the value of l2vc is null and does not contain **secondary**, add **primary** to the end of the line.

2. **post-config**

```
def configCliPostProcessor(self, aoccontext, request):
    out = CliCustomTransformOutput()
    cli_str = request.data
    re_mpls = r'.*(undo\s+)?mpls l2vc.+primary.*'
    if re.search(re_mpls, cli_str):
        newlines = []
        lines = cli_str.splitlines()
            for line in lines:
                if re.search(re_mpls, line):
                    line = line.replace('primary', '')
                newlines.append(line)
            cli_str = '\n'.join(newlines)
    out.data = cli_str
    return out
```

   As shown in the preceding figure, the customized code of **post-config** is written in the configCliPostProcessor method. You can obtain the CLI data to be processed from **request.data**, and then process the CLI data using the regular expression or other measures. If the data contains **mpls l2vc primary**, replace **primary** with an empty character string. After the processing is complete, the new CLI data is assigned to **out.data**.

3.2.4 cli-custom-rpc

Application Scope

Top-level node of rpc.

Function

The forward and reverse conversion of YANG and CLI of rpc is implemented using YangToCli and CliTtoYang methods in the SND.

1. A user invokes the input interface of the rpc to import the input path and data. CLI Driver invokes the YangToCli method of the SND to generate commands delivered to the device.

2. CLI Driver issues commands to the device and obtains data returned by the device.
3. CLI Driver invokes the CliTtoYang method in the SND to import the output path and data returned by the device.

Example

1. **YANG file**

```
rpc resetIpStatistics {
    cli:cli-custom-rpc;
    input {
        leaf interface-name {
            type string;
        }
    }
    output {
        leaf response {
            type string;
        }
    }
}
```

In the YANG file, resetIpStatistics is the top-level node of rpc. cli-custom-rpc applies only to the top-level node. The input parameter **path=/resetIpStatistics/input** invokes the YangToCli method of the SND to generate the CLI to be sent to the device. The output parameter **path=/resetIpStatistics/output** invokes the CliTtoYang method of the SND to generate YANG data.

2. **SND package**

```
def yangToCli(self, aoccontext, request):
    out = CliCustomTransformOutput()
    if request.path == "/cli-custom:resetIpStatistics/cli-custom:input":
        out.data = "show running-config | block router-id"
    return out


def cliToYang(self, aoccontext, request):
    out = CliCustomTransformOutput()
    if request.path == "/cli-custom:resetIpStatistics/cli-custom:output":
        out.data = "<resetIpStatistics xmlns=\"https://huawei.com/cli-custom\">" \
                   "<response>hello_world</response>" \
                   "</resetIpStatistics>"
        out.endOffset = str(request.data).__len__()
    return out
```

In the preceding figure, the forward customized codes of rpc are written in the YangToCli method. **request.path** points to the input node of rpc YANG. The content of **out.data** is the CLI to be issued to the device. The reverse customized code of rpc is written in the CliTtoYang method. **request.path** points to the output node of rpc YANG, and **out.data** corresponds to the YANG data to be returned.

4 RESTCONF Customization Extension

4.1 Context

The RestconfDriver is a general engine. To reduce unnecessary special code and compatibility code, the following customized extension items are developed to transfer the forward and reverse conversion logic between YANG and RESTCONF to the specific NE driver (SND) package. You can customize the conversion logic based on the RESTCONF packet features of different devices through encoding to implement the flexibility of the RESTCONFDriver in parsing RESTCONF and YANG functions.

4.2 Extension

4.2.1 custom-yang-to-restconf

Application Scope

Container, list, and leaf nodes

NOTE

RPC operations can be performed only on the top nodes.

Function

The YANG-to-REASTCONF conversion logic of the container nodes, list nodes, and leaf nodes which are marked with custom-yang-to-restconf is defined in the SND package. Six parameters can be added to the end of custom-yang-to-restconf. The parameters are separated by commas (,). The create, update, delete, merge, read, and rpc parameters specify the operation types. The SND package provides the YANG-to-RESTCONF conversion logic for the operations on the nodes and their sub-nodes marked with custom-yang-to-restconf.

Example

1. **YANG file**

```
rpc resetIpStatistics {
    restconf:custom-yang-to-restconf 'rpc';
    input {
        leaf interface-name {
            type string;
        }
    }
    output {
        leaf response {
            type string;
        }
    }
}
```

In the preceding YANG file, resetIpStatistics is a top node of RPC. The input parameter path=/resetIpStatistics/input invokes the YangToRestconf method of the corresponding operation in the SND package to generate a RESTCONF request packet to be sent to the device.

2. **SND package**

For details, see the implementation of the rpcCustomYangToRestconf method in **2.3.5.1 YANG-to-RESTCONF Conversion for RPC Operations**.

4.2.2 custom-restconf-to-yang

Application Scope

Container, list, and leaf nodes

Function

The RESTCONF-to-YANG conversion logic of the container nodes, list nodes, and leaf nodes which are marked with custom-restconf-to-yang container is defined in the SND package. Six parameters can be added to the end of custom-restconf-to-yang. The parameters are separated by commas (,). The create, update, delete, merge, read, and rpc parameters specify the operation types. The SND package provides the RESTCONF-to-YANG conversion logic for the operations on the nodes and their sub-nodes marked with custom-restconf-to-yang.

Example

1. **YANG file**

```
container l3vpn-svcs {
    restconf:custom-restconf-to-yang 'read';
    description
      "Service information set.";
    uses gp-l3vpn-svcs;
  }
}
```

In the preceding YANG file, l3vpn-svcs is a top node. The readCustomRestconfToYang method in an SND package is invoked for the read operation on the node and all its child nodes to convert RESTCONF packets of all read operations on the node and its child nodes to YANG data.

2. **SND package**

For details, see the implementation of the readCustomRestconfToYang method in **2.3.5.6 RESTCONF-to-YANG Conversion for the Read Operation**.

4.2.3 custom-post-process

Application Scope

Container, list, and leaf nodes

Function

The SND package performs post-processing on the request information generated by the RestconfDriver on the container nodes, leaf nodes, and list nodes marked with custom-post-process, including modifying requestBody and HTTPMethod and determining whether to invoke the RESTful interface. Six parameters can be added to the end of custom-post-process. The parameters are separated by commas (,). The create, update, delete, merge, read, and rpc parameters specify the operation types.

Example

1. **YANG file**

```
grouping gp-l3vpn-svcs {
    description
        "Service information set.";
    list l3vpn-svc {
        restconf:custom-post-process 'create';
        key "vpn-id";
        description
            "Service information set.";
        uses gp-l3vpn-svc;
    }
}
```

In the preceding YANG file, when an l3vpn-svc is created in the list, the configCustomPostProcess method in the SND package is invoked to perform post-processing on the request information generated by the RestconfDriver.

2. **SND package**

For details, see the implementation of the configCustomPostProcess method in **2.3.5.12 Post-processing for the CONFIG Operation**.

4.2.4 custom-pre-process

Application Scope

Container, list, and leaf nodes

Function

The SND package performs pre-processing on the RESTCONF information returned by devices on the container nodes, leaf nodes, and list nodes marked with custom-pre-process, converts the information into the response packets that meet the RESTCONFDriver requirements, and then sends the response packets to the RESTCONFDriver for processing. Seven parameters can be added to the end of custom-pre-process. The parameters are separated by commas (,). The create, update, delete, merge, read, rpc, and error parameters specify the operation types.

Example

1. **YANG file**

```
grouping gp-l3vpn-svcs {
    description
        "Service information set.";
    list l3vpn-svc {
        restconf:custom-pre-process 'create';
        key "vpn-id";
        description
            "Service information set.";
        uses gp-l3vpn-svc;
    }
}
```

   In the preceding YANG file, when an l3vpn-svc is created in the list, the configCustomPreProcess method in the SND package is invoked to perform pre-processing on the response information generated by the device.

2. **SND package**

   For details, see the implementation of the configCustomPreProcess method in **2.3.5.11 Pre-processing for the CONFIG Operation**.

