开放可编程
V100R021C10
插件包 API 参考

文档版本 01

发布日期 2023-04-24

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： https://www.huawei.com

客户服务邮箱： support@huawei.com

客户服务电话： 4008302118

目 录

1 前言

概述

介绍了开放可编程系统相关功能的操作指导。

读者对象

本文档主要适用于以下工程师：

- 系统管理员
- 维护工程师
- 技术支持工程师

符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

符号	说明
危险	用于警示紧急的危险情形，若不避免，将会导致人员死亡或严重的人身伤害。
警告	用于警示潜在的危险情形，若不避免，可能会导致人员死亡或严重的人身伤害。
注意	用于警示潜在的危险情形，若不避免，可能会导致中度或轻微的人身伤害。
须知	用于传递设备或环境安全警示信息，若不避免，可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。 “注意”不涉及人身伤害。
说明	用于突出重要/关键信息、最佳实践和小窍门等。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害。

图形界面元素引用约定

本文中可能出现下列图形界面元素，它们所代表的含义如下。

格式	意义
“ ”	带双引号“ ”的格式表示各类界面控件名称和数据表，如单击“确定”。
>	多级菜单用“>”隔开。如果选择“文件 > 新建 > 文件夹”，表示选择“文件”菜单下的“新建”子菜单下的“文件夹”菜单项。

命令行格式约定

本文可能出现下列命令行格式，它们所代表的含义如下。

格式	意义
粗体	命令行关键字（命令中保持不变、必须照输的部分）采用**加粗**字体表示。
斜体	命令行参数（命令中必须由实际值进行替代的部分），采用*斜体*表示。
[ ]	表示用“[ ]”括起来的部分在命令配置时是可选的。
{x | y |...}	表示从两个或多个选项中选取一个。
[x | y | ...]	表示从两个或多个选项中选取一个或者不选。
{ x | y | ... } *	表示从两个或多个选项中选取多个，最少选取一个，最多选取所有选项。
[x | y | ...] *	表示从两个或多个选项中选取多个或者不选。

修改记录

文档版本	发布日期	修改说明
03	2022-09-30	第三次正式发布。
02	2022-04-15	第二次正式发布。
01	2022-01-10	第一次正式发布。

2 aoc

2.1 aoc package

class aoc.BaseService(logger, input_types, output_types)

abstract **do_ncs_map**(map_input, aoc_context, template=None)

系统提供给用户的映射接口。

典型场景:

用户调用SSP配置，并走到用户的ncs_map的实现以处理映射。

功能描述:

用户调用SSP配置，并走到用户的ncs_map的实现以处理映射。

接口约束:

如果一个用户异常被抛出，那需要通过使用固定的用户异常类。

注解

此函数具有默认实现。如果映射逻辑中的变量不需要额外处理，可以直接使用默认实现。

- **参数**
  - **map_input** (*obj*) - 请求参数
  - **aoc_context** (*obj*) - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
  - **template** (*str*) - 模板路径
- **返回**
  - 业务映射配置结果，xml格式。
- **返回类型**
  - str

实际案例

1.用户直接将请求数据传递到模板渲染中

```
def ncs_map(self, request, aoc_context=None, template=''):
    result = self.render(template, request.xmldict)
    return result
```

2.用户需要代码来处理请求，并构造模板参数进行渲染

```
def ncs_map(self, request, aoccontext=None, template=None):
    # get isp service input data
    request_dic_node = self.xmltodictnode(request.xml)
    self.logger.info("bngAaaPairs request_dic_node: %s" % (request_dic_node))
    template_context = self.fill_miniaaapair_context(request_dic_node, aoccontext)
    # query bng data from cdb in isp service
    self.logger.info("bngAaaPairs template_syspair_context: %s" % (template_context))
    # render the j2 template with template_context
    strt = self.render('template_Aaa.j2', template_context)
    self.logger.info("bngAaaPairs template_nepair%s" % (strt))
    return strt
#fill context
def fill_miniaaapair_context(self, request_dic_node, aoccontext):
    self.aaamini = request_dic_node.aaaminignd
    servicepath = aoccontext.serviceInst
    nerootpath = aoccontext.serviceInst[servicepath.find("ne/")+3:]
    neid = nerootpath[:nerootpath.find("/")]
    return {'users':self.aaamini,'neid':neid}
```

***abstract* do_ncs_rpc**(rpc_input**,** aoc_context**,** template=None)

提供RPC操作能力。

典型场景:

系统提供给用户的ncs_rpc接口。

功能描述:

提供RPC操作能力。

接口约束:

rpc_input 必须合法.

- **参数**
  - **rpc_input** (*str*) - rpc输入参数
  - **aoc_context** (*obj*) - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
  - **template** - 模板名称
- **返回**
  - rpc的返回内容，如果没有输出，则返回空
- **返回类型**
  - str

示例

这是一个ncs_rpc函数示例

```
rpc_input.inputData = '<input><userName>xxx</userName></input>'
rpc_input.rpcName = '(nrn:com:huawei)activeUser'
```



```
def ncs_rpc(self, rpc_input, aoc_context, template=None):
    sndinputData = convert(rpc_input.inputData)
    neid = getneid(sndinputData)
    sndrpcName = convert(rpc_input.rpcName)
    sndresult = devicemgr.device_rpc(neid, sndinputData, sndrpcName)
    sspresult = convert(sndresult.result)
    return sspresult
>>> print(ncs_rpc(rpc_input))
<output><result>OK</result></output>
```

setup()

类调用前的初始化操作。

典型场景:

此函数用于初始化资源。

功能描述:

此函数用于初始化资源。

接口约束:

无。

实际案例

```
import threading
def fun_timer():
    print('Hello Aoc!')
def setup(self):
    timer = threading.Timer(1, fun_timer)
    timer.start()
```

teardown()

卸载插件包时调用，与setup一起使用。

典型场景:

该函数用于资源释放。

功能描述:

卸载插件包时调用，与setup一起使用。

接口约束:

无。

实际案例

```
def teardown(self):
timer.cancel()
```

class aoc.MapRequest(xml, context, dryrun_context=None)

基类： object

用户映射编码输入数据。

典型场景:

当用户在进行SSP业务映射编码时，使用MapRequest把里面的内容放进去。

功能描述:

此类为用户映射代码的输入数据结构。

接口约束:

数据结构是固定的，里面的原始配置数据是xml格式。

- **参数**
  - **xml** (*str*) - xml格式的数据
  - **context** (*obj*) - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
  - **dryrun_context** (*obj*) - 试运行的上下文信息

***property* xmldict**

系统提供接口供用户获取dict。

典型场景:

用户获取MapRequest后，获取dict格式的数据。

功能描述:

函数xmldict接口，用于获取dict。

接口约束:

xml格式标准可以转换为dict。

返回

用于xml数据转换的dict。

返回类型

dict

实际案例

```
>>> request = MapRequest(map_input.config.data, dict(map_input.contextMap))
>>> print(request.xmldict())
{'aaa':'xxx'}
```

***property* xmldictnode**

系统提供接口供用户获取dictNode。

典型场景:

用户获取MapRequest后，获取dictNode格式的数据。

功能描述:

函数用于获取xmldictnode的接口。

接口约束:

xml格式标准可以转换为dictNode。

返回

xml数据转换的dict

返回类型

dict

实际案例

```
>>> request = MapRequest(map_input.config.data, dict(map_input.contextMap))
>>> print(request.xmldictnode())
{'aaa':'xxx'}
```

class aoc.NcsService(logger=None, resource_dir='')

基类：aoc.base.baseservice.BaseService

用于映射用户继承映射的父类。

典型场景:

当用户实现自己的类继承父类后，实现ncs_map方法处理映射逻辑。

功能描述:

该类提供了一个映射父类供用户继承。

接口约束:

用户代码处理jinja模板的呈现。

- **参数**
  - **logger** (*str, optional*) - 当前包组的日志打印
  - **resource_dir** (*str*) - 资源目录

add_filters()

系统提供了一个接口，供用户复制和添加自定义过滤器。

典型场景:

用户调用用户jinja模板中的用户定义过滤器的函数。

功能描述:

用于用户复制，以添加自定义筛选器接口的函数。

接口约束:

不要重复机制的过滤器名称。

返回

过滤dict内容

返回类型

dict

实际案例

```
>>> def add_filters(self):
>>>     return {'doSomething':doSomething}
>>> def doSomething(self):
>>>     return'something'
```


discover(discover_input**,** aoc_context)

业务恢复接口，由第三方用户实现。

典型场景:

用户需要将网元的配置从现有网络恢复到业务的配置。该函数从网元库读取数据，然后返回相应的业务实例。

功能描述:

提供业务恢复实施。

接口约束:

1. 网元配置须先同步到控制器。
2. 在恢复过程中，根据业务数据量批量处理，机制传递上下文控制恢复是否结束。

说明

当业务量规格较大时，为了避免内存溢出和超时，可以定制该函数被系统多次调用，每次处理少量业务实例。

- **参数**
  - **discover_input** (*obj*) -
    - 业务恢复输入，输入参数以protobuf格式定义

```
message DiscoverInput {
    repeated DeviceInfo deviceInfo; # device info
    string discoveryId; # discovery id
    repeated ServiceContext serviceContext;
    repeated ServiceInstance existService;
    string aocNode;
    map<string, string> cacheContext; # Restore context,
    used for multiple restore processing
}
message ServiceContext {
    string servicePath;
    map<string, string> context;
}
message DeviceInfo {
    string deviceId;
    string deviceName;
    string attributes;
}
```

  - **aoc_context** (*obj*) - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
- **返回**
  - 恢复发现的业务实例的配置数据

```
message DiscoverOutput {
    repeated ServiceConfig serviceConfig;# Business instance data, list
    bool goOn;# Whether to continue to restore, the default is false
    map<string, string> cacheContext; # Restore context,
    used for multiple restore processing
}
message ServiceConfig {
    string servicePath; # Business instance path
    string serviceData; # Business instance data
```



```
    map<string, string> context; # Business instance context,
    generally used for resource occupation
    enum ServiceOperation {
        MERGE;
        REPLACE;
    }
    ServiceOperation operation; # Business instance operation code
}
```

实际案例

1. 当业务量不大时，用户可以选择一次恢复所有业务实例

```
def discover(self, discover_input, aoc_context):
    # Get device information from discover_input
    device_info_list = getDeviceinfolist(discover_input)
    result = DiscoverOutput()
    # Traverse the device information in the input information list
    for device_info in device_info_list:
        device_id = device_info.deviceId
        path ='/huawei-ac-nes:inventory-cfg/nes/ne/'+ deviceId +'/huawei-tunnel:tunnels'
        dbresult = datastore.read_datastore_rdb(aoc_context, path)
        tunnels = self.xmltodictnode(dbresult)
        self.handler_tunnels(tunnels, device_id, result)
    return result
```

2. 当业务量较大时，用户选择批量恢复

```
def discover(self, discover_input, aoc_context):
    # Get device information from discover_input
    device_info_list = getDeviceinfolist(discover_input)
    # Business volume limit
    tunnels_max_num = 100
    cache_context = discoverInput.cacheContext
    result = DiscoverOutput()
    # Traverse the device information in the input information list
    for device_info in device_info_list:
        device_id = device_info.deviceId
        path ='/huawei-ac-nes:inventory-cfg/nes/ne/'+ deviceId +'/huawei-tunnel:tunnels'
        dbresult = datastore.read_datastore_rdb(aoc_context, path)
        tunnels = self.xmltodictnode(dbresult)
        # According to the business mark in cache_context,
        exclude the business that has been restored,
        and obtain the newly restored business
        tunnels_new = getNewTunnels(cache_context, tunnels)
        self.handler_tunnels(tunnels_new, device_id, result)
        if (getTunnelsSize(result)> tunnels_max_num)
            result.goOn = true
            result.cacheContext = updateContext(tunnels_new, cache_context)
            break
    return result
```

get_global(key)

系统提供接口供用户获取全局参数。

典型场景:

系统提供接口供用户编码和获取全局参数。

功能描述:

系统提供接口供用户获取全局参数。

接口约束:

无。

说明

该函数解决了Jinja的原生set和get无法传递跨模板设置变量的问题。

- **参数**
  - **key** - key值
- **返回**
  - 业务映射配置结果，xml格式
- **返回类型**
  - str

实际案例

jinja context:

```
<aaa xmlns="http://www.huawei.com/netconf/vrp/huawei-aaa">
  <tasks>
    <task>
      <taskName>{{GET_GLOBAL('xxx')}}</taskName>
    </task>
  </tasks>
  {% if isis == 'true' %}
    <taskGroups>{% for ww in wws %}
      <taskGroup>
        <taskGroupName>{{ww.xx}}</taskGroupName>
        <taskGroupName1>{{ww.yy}}</taskGroupName1>
      </taskGroup>{% endfor %}
    </taskGroups>
  {% endif %}
</aaa>
```

is_snd(nename, sndid)

根据入参nename和sndid检查设备是否绑定了snd包。

典型场景:

需要检查设备是否绑定了snd包来进行下一步的动作。

功能描述:

根据入参nename和sndid检查设备是否绑定了snd包。

接口约束:

无。

- **参数**
  - **nename** (*str*) - 设备的名称。
  - **sndid** (*str*) - snd包的ID。
- **返回**
  - 如果名称为nename的设备由ID为sndid的插件包管理，则为True，否则为False
- **返回类型**

- bool

实际案例

描述样例：

```
>>> nename = 'test1'
>>> sndid = 'NESPC800'
>>> isSnd = is_snd(neid, sndid)
>>> print isSnd
True
```

ncs_map(request, aoc_context=None, template='')

系统提供给用户的映射接口。

典型场景:

用户调用SSP配置，并走到用户的ncs_map的实现以处理映射。

功能描述:

用户调用SSP配置，并走到用户的ncs_map的实现以处理映射。

接口约束:

如果一个用户异常被抛出，那需要通过使用固定的用户异常类。

注解

此函数具有默认实现。如果映射逻辑中的变量不需要额外处理，可以直接使用默认实现。

- **参数**
  - **request** (*obj*) - 请求参数
  - **aoc_context** - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
  - **template** (*str*) - 模板路径
- **返回**
  - 业务映射配置结果，xml格式
- **返回类型**
  - str

实际案例

1. 用户直接将请求数据传递到模板渲染中

```
def ncs_map(self, request, aoc_context=None, template=''):
    result = self.render(template, request.xmldict)
    return result
```

2. 用户需要代码来处理请求，并构造模板参数进行渲染

```
def ncs_map(self, request, aoccontext=None, template=None):
    # get isp service input data
    request_dic_node = self.xmltodictnode(request.xml)
    self.logger.info("bngAaaPairs request_dic_node: %s" % (request_dic_node))
    template_context = self.fill_miniaaapair_context(request_dic_node, aoccontext)
    # query bng data from cdb in isp service
    self.logger.info("bngAaaPairs template_syspair_context:%s" % (template_context))
    # render the j2 template with template_context
```



```
    strt = self.render('template_Aaa.j2', template_context)
    self.logger.info("bngAaaPairs template_nepair%s" % (strt))
    return strt
#fill context
def fill_miniaaapair_context(self, request_dic_node, aoccontext):
    self.aaamini = request_dic_node.aaaminignd
    servicepath = aoccontext.serviceInst
    nerootpath = aoccontext.serviceInst[servicepath.find("ne/")+3:]
    neid = nerootpath[:nerootpath.find("/")]
    return {'users':self.aaamini,'neid':neid}
```

ncs_rpc(rpc_input**,** aoc_context**,** template=None)

系统提供给用户的ncs_rpc接口。

典型场景:

系统提供给用户的ncs_rpc接口。

功能描述:

系统提供给用户的ncs_rpc接口。

接口约束:

rpc_input入参必须合法。

- **参数**
  - **rpc_input** (*obj*) - rpc输入参数
  - **aoc_context** (*obj*) - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
  - **template** (*str*) - 模板名称
- **返回**
  - 转换后的aoc ncs rpc结果.
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.ncs.aocncsrpc.AocNcsRpcOutput

实际案例

这是一个ncs_rpc函数示例

```
rpc_input.inputData = '<input><userName>xxx</userName></input>'
rpc_input.rpcName = '(nrn:com:huawei)activeUser'
def ncs_rpc(self, rpc_input, aoc_context, template=None):
    sndinputData = convert(rpc_input.inputData)
    neid = getneid(sndinputData)
    sndrpcName = convert(rpc_input.rpcName)
    sndresult = devicemgr.device_rpc(neid, sndinputData, sndrpcName)
    sspresult = convert(sndresult.result)
    return sspresult
sspresult = ncs_rpc(rpc_input)
print(sspresult)
<output><result>OK</result></output>
```

ncs_validate(aoc_context**,** validate_input)

业务恢复接口，由第三方用户实现。

典型场景:

第三方用户需要验证其输入。

功能描述:

验证实现接口，由第三方用户复制。在这里，打开了一个钩子供用户处理，例如资源冲突验证。

接口约束:

无。

- **参数**
  - **aoc_context** (*obj*) - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
  - **validate_input** (*obj*) - 校验入参

```
post_modify(aoc_context, post_modify_request)
```

更新或删除服务后，此方法将在同一事务中调用。

典型场景:

在某些情况下，此业务正在更新或删除，用户也需要更新或删除相关服务实例，用户可以实现此方法。

功能描述:

更新或删除业务后，此方法将在同一事务中调用。

接口约束:

对于某个业务，此方法只在系统中实现一次。

- **参数**
  - **aoc_context** - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
  - **post_modify_request** - 自定义类postModify请求输入， 请参见：类：**PostModifyRequest**。
- **返回**
  - PostModify结果.

实际案例

这里有一个例子来说明Post-Modift函数的输入和输出参数。

```
In this example, eva depends on evs
evs Yang definition:
augment "/app:applications"{
   list evs {
      app:application-definition "evs";
      key "evs-id";
      leaf evs-id {
         type string {
            length "1..24";
         }
         description "evs-id";
      }
   }
}
eva Yang definition:
```



```
augment "/app:applications" {
    list eva {
        app:application-definition "eva";
        key "eva-id";
        leaf eva-id {
            type string {
                length "1..24";
            }
            description "eva-id";
        }
        leaf eas-id {
            type string {
                length "1..24";
            }
            description "eas-id";
        }
        leaf evs-id {
            type string {
                length "1..24";
            }
            description "evs-id";
        }
    }
}
register process:
user need to modify package'pkg.json file(Example in evs),
add below hook information
"hooks": [
    {
        "type": "post-modify-action",
        "key": "evs",
        "python-class-name": "kpn.evs.AocNcsEvsServicePoint"
    }
]
def post_modify(self, aoc_context, post_modify_request):
    #  Get device information from post_modify
    for data in post_modify_request.change_data:
        path = data.path
        instance_name = path.split('/')[-1]
        attached_path ="/huawei-ac-applications:applications/huawei-ncs-eva:eva/" +
instance_name
        if data.operation_type == 1:
            service.redeploy(attached_path)
```

```
render(template_name, *args, **kwargs)
```

系统提供的用户模板渲染接口。

典型场景:

用户在代码中调用渲染接口进行模板渲染的函数。

功能描述:

用于模板渲染的函数接口。

接口约束:

模板的参数内容必须与模板变量匹配。模板中只能使用机制提供的过滤器和用户SSP包定义的过滤器。

- **参数**

  - **template_name** (*str*) - 模板的名称
  - **args** (*list, optional*) - 位置参数，模板中使用的变量，必须是字典
  - **kwargs** (*dict, optional*) - 关键字参数，模板中使用的变量
- **返回**
  - 模板渲染后的配置内容
- **返回类型**
  - str

实际案例

```
temppara = {
   'taskName': '111',
   'wws': [{"xx":1,"yy":"xxxxxx"},
      {"xx":7}],
   'isis': 'true'
}
>>> result = render("aaa.j2", temppara, taskName='222')
>>> print(result)
<aaa xmlns="http://www.huawei.com/netconf/vrp/huawei-aaa">
   <tasks>
      <task>
         <taskName>222</taskName>
      </task>
   </tasks>
   <taskGroups>
      <taskGroup>
         <taskGroupName>1</taskGroupName>
         <taskGroupName1>xxxxxx</taskGroupName1>
      </taskGroup>
      <taskGroup>
         <taskGroupName>7</taskGroupName>
         <taskGroupName1></taskGroupName1>
      </taskGroup>
   </taskGroups>
</aaa>
```

set_global(key**,** val)

系统提供接口供用户设置全局参数

典型场景:

系统提供接口供用户编码和设置全局参数。

功能描述:

系统提供接口供用户设置全局参数。

接口约束:

无。

- **参数**
  - **key** (*str*) - key值
  - **val** (*str*) - val值
- **返回**
  - 业务映射配置结果，xml格式

- **返回类型**
  - str

实际案例

template1:

```
<inventory-cfg xmlns="urn:huawei:yang:huawei-ac-nes">
{{SET_GLOBAL('xxx','111')}}
   <nes>
{%- for user in users %}
      <ne>
         <neid>{{ user.nename | to_ne_id }}</neid>
         {%- include 'com/huawei/aaa/hbng-aaa.j2' %}
      </ne>
{%- endfor %}
   </nes>
</inventory-cfg>
```

template2:

```
<aaa xmlns="http://www.huawei.com/netconf/vrp/huawei-aaa">
   <tasks>
      <task>
         <taskName>{{GET_GLOBAL('xxx')}}</taskName>
      </task>
   </tasks>
   {% if isis == 'true' %}
      <taskGroups>{% for ww in wws %}
         <taskGroup>
            <taskGroupName>{{ww.xx}}</taskGroupName>
            <taskGroupName1>{{ww.yy}}</taskGroupName1>
         </taskGroup>{% endfor %}
      </taskGroups>
   {% endif %}
</aaa>
```

***classmethod* to_manage_ip**(ne_name)

系统为用户提供jinja模板，根据ne_name查询管理ip的功能。

典型场景:

系统为用户提供jinja模板，根据ne_name查询管理ip的功能。

功能描述:

系统为用户提供jinja模板，根据ne_name查询管理ip的功能。

接口约束:

neName入参必须合法。

- **参数**
  - **ne_name** (*str*) - 网元名称
- **返回**
  - 管理IP
- **返回类型**
  - str

实际案例

jinja example:

```
<inventory-cfg xmlns="urn:huawei:yang:huawei-ac-nes">
   <nes>
      {%- for ne in nes %}
         <ne>
            <neip>{{ne.deviceName | to_manage_ip}}</neip>
            {{ne.nexml}}
         </ne>
      {%- endfor %}
   </nes>
</inventory-cfg>
```

***classmethod* to_ne_id**(ne_name)

系统为用户提供jinja模板，根据ne_name查询ne_id的函数。

典型场景:

该系统提供给用户使用jinja模板，根据ne_name获取ne_id的功能。

功能描述:

该系统提供给用户使用jinja模板，根据ne_name获取ne_id的功能。

接口约束:

neName必须合法。

- **参数**
  - **ne_name** (*str*) - 网元名称。
- **返回**
  - 网元ID。
- **返回类型**
  - str

实际案例

下面是在jinja模板中使用函数的示例:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
   ~ Copyright (c) Huawei Technologies Co., Ltd. 2020-2021. All rights reserved.
   -->
   <inventory-cfg xmlns="urn:huawei:yang:huawei-ac-nes">
      <nes>
      {%- for ne in nes %}
         <ne>
            <neid>{{ne.deviceName | to_ne_id}}</neid>
            {{ne.xml}}
         </ne>
      {%- end for %}
      </nes>
   </inventory-cfg>
```

static* xmltodict**(*xml_input*, *encoding=None*, *expati=<module 'xml.parsers.expat' from 'c:||users||***||appdata||local||programs||python||python37||lib||xml||parsers||expat.py'>*, *process_namespaces=False*, *namespace_separator=':'*, *disable_entities=True*, ***kwargs*)

系统提供用户将xml转换为dict。

典型场景:

在用户编码过程中，使用xml转换为dict。

功能描述:

在用户编码过程中，使用xml转换为dict。

接口约束:

xml必须为标准格式。

- **参数**
  - **xml_input** (*str*) - 输入xml数据
  - **encoding** (*obj, optional*) - 编码内容
  - **expati** (*obj, optional*) - 通过替代版本
  - **process_namespaces** (*obj, optional*) - 进程命名空间
  - **namespace_separator** (*obj, optional*) - 分离字符串
  - **disable_entities** (*obj, optional*) - 禁用实体
  - **kwargs** (*obj, optional*) - 字典格式输入参数
- **返回**
  - 字典内容
- **返回类型**
  - dict

实际案例

xml_input 可以是 string 也可以是类似文件的对象.

如果xml_attribs为True，则元素属性将放入字典中 在常规子元素中，使用@作为前缀以避免冲突。如果 设置为False，它们只是被忽略。简单的例子

```
>>> doc = xmltodict("""
... <a prop="x">
...   <b>1</b>
...   <b>2</b>
... </a>
... """)
>>> doc['a']['@prop']
u'x'
>>> doc['a']['b']
[u'1', u'2']
```

If item_depth is 0, the function returns a dictionary for the root element (default behavior). Otherwise, it calls item_callback every time an item at the specified depth is found and returns None in the end (streaming mode).

The callback function receives two parameters: the path from the document root to the item (name-attribs pairs), and the item (dict). If the callback’s return value is false-ish, parsing will be stopped with the ParsingInterrupted exception.Streaming example

```
>>> def handle(path, item):
...     print('path:%s item:%s' % (path, item))
...     return True
```



```
...
>>> xmltodict("""
... <a prop="x">
...   <b>1</b>
...   <b>2</b>
... </a>""", item_depth=2, item_callback=handle)
path:[(u'a', {u'prop': u'x'}), (u'b', None)] item:1
path:[(u'a', {u'prop': u'x'}), (u'b', None)] item:2
```

The optional argument postprocessor is a function that takes path, key and value as positional arguments and returns a new (key, value) pair where both key and value may have changed. Usage example

```
>>> def postprocessor(path, key, value):
...     try:
...         return key + ':int', int(value)
...     except (ValueError, TypeError):
...         return key, value
>>> xmltodict('<a><b>1</b><b>2</b><b>x</b></a>',
...                postprocessor=postprocessor)
OrderedDict([(u'a', OrderedDict([(u'b:int', [1, 2]), (u'b', u'x')]))])
```

You can pass an alternate version of expat (such as defusedexpat) by using the expat parameter. E.g:

```
>>> xmltodict('<a>hello</a>', expat=defusedexpat.pyexpat)
OrderedDict([(u'a', u'hello')])
```

You can use the force_list argument to force lists to be created even when there is only a single child of a given level of hierarchy. The force_list argument is a tuple of keys. If the key for a given level of hierarchy is in the force_list argument, that level of hierarchy will have a list as a child (even if there is only one sub-element). The index_keys operation takes precendence over this. This is applied after any user-supplied postprocessor has already run.For example, given this input:

```
<servers>
   <server>
      <name>host1</name>
      <os>Linux</os>
      <interfaces>
         <interface>
            <name>em0</name>
            <ip_address>10.0.0.1</ip_address>
         </interface>
      </interfaces>
   </server>
</servers>
If called with force_list=('interface',), it will produce this dictionary:
{'servers':
   {'server':
      {'name': 'host1',
       'os': 'Linux'},
       'interfaces':
         {'interface':
            [ {'name': 'em0', 'ip_address': '10.0.0.1' } ] } } }
```

force_list can also be a callable that receives path, key and value. This is helpful in cases where the logic that decides whether a list should be forced is more complex.

static* xmltodictnode(*xml_input*, *encoding=None*, *expati=<module 'xml.parsers.expat' from 'c:||users||***||appdata||local||programs||python||python37||lib||xml||parsers||expat.py'>*, *process_namespaces=False*, *namespace_separator=':'*, *disable_entities=True*, ***kwargs*)**

系统提供用户将xml转换为dictNode。

典型场景:

在用户编码过程中，使用xml转换为dictNode。

功能描述:

在用户编码过程中，使用xml转换为dictNode。

接口约束:

xml必须为标准格式。

注解

不要在Args部分中包括self参数。

- **参数**
  - **xml_input** (*str*) - 输入xml数据
  - **encoding** (*obj, optional*) - 编码内容
  - **expati** (*obj, optional*) - 通过替代版本
  - **process_namespaces** (*obj, optional*) - 进程命名空间
  - **namespace_separator** (*obj, optional*) - 分离字符串
  - **disable_entities** (*obj, optional*) - 禁用实体
  - **kwargs** (*obj, optional*) - 字典格式输入参数
- **返回**
  - dictNode内容
- **返回类型**
  - dict

实际案例

xml_input 可以是 string 也可以是类似文件的对象.

如果xml_attribs为True，则元素属性将放入字典中 在常规子元素中，使用@作为前缀以避免冲突。如果 设置为False，它们只是被忽略。简单的例子

```
>>> doc = xmltodictnode("""
... <a prop="x">
...   <b>1</b>
...   <b>2</b>
... </a>
... """)
>>> doc['a']['@prop']
u'x'
>>> doc['a']['b']
[u'1', u'2']
```

If item_depth is 0, the function returns a dictionary for the root element (default behavior). Otherwise, it calls item_callback every time an item at the specified depth is found and returns None in the end (streaming mode).

The callback function receives two parameters: the path from the document root to the item (name-attribs pairs), and the item (dict). If the callback’s return value is false-ish, parsing will be stopped with the ParsingInterrupted exception.Streaming example


```
>>> def handle(path, item):
...     print('path:%s item:%s' % (path, item))
...     return True
...
>>> xmltodictnode("""
... <a prop="x">
...   <b>1</b>
...   <b>2</b>
... </a>""", item_depth=2, item_callback=handle)
path:[(u'a', {u'prop': u'x'}), (u'b', None)] item:1
path:[(u'a', {u'prop': u'x'}), (u'b', None)] item:2
```

The optional argument postprocessor is a function that takes path, key and value as positional arguments and returns a new (key, value) pair where both key and value may have changed. Usage example

```
>>> def postprocessor(path, key, value):
...     try:
...         return key + ':int', int(value)
...     except (ValueError, TypeError):
...         return key, value
>>> xmltodictnode('<a><b>1</b><b>2</b><b>x</b></a>',
...                postprocessor=postprocessor)
OrderedDict([(u'a', OrderedDict([(u'b:int', [1, 2]), (u'b', u'x')]))])
```

You can pass an alternate version of expat (such as defusedexpat) by using the expat parameter. E.g:

```
>>> xmltodictnode('<a>hello</a>', expat=defusedexpat.pyexpat)
OrderedDict([(u'a', u'hello')])
```

You can use the force_list argument to force lists to be created even when there is only a single child of a given level of hierarchy. The force_list argument is a tuple of keys. If the key for a given level of hierarchy is in the force_list argument, that level of hierarchy will have a list as a child (even if there is only one sub-element). The index_keys operation takes precendence over this. This is applied after any user-supplied postprocessor has already run.For example, given this input:

```
<servers>
   <server>
      <name>host1</name>
      <os>Linux</os>
      <interfaces>
         <interface>
            <name>em0</name>
            <ip_address>10.0.0.1</ip_address>
         </interface>
      </interfaces>
   </server>
</servers>
If called with force_list=('interface',), it will produce this dictionary:
{'servers':
   {'server':
      {'name': 'host1',
      'os': 'Linux'},
      'interfaces':
         {'interface':
            [ {'name': 'em0', 'ip_address': '10.0.0.1' } ] } } }
```


force_list can also be a callable that receives path, key and value. This is helpful in cases where the logic that decides whether a list should be forced is more complex.

3 aoc.base

3.1 aoc.base.action module

典型场景:

如果需要定义一个服务的O&M功能，可以在YANG中定义一个action 并在此类中实现这个action操作。

功能描述:

此类是 SSP 或 GND 中action的基类。如果需要定义一个服务的O&M功能， 可以在YANG中定义一个action并在此类中实现这个action操作。

class aoc.base.action.ActionRequest(neid, action_path, action_xml)

基类: object

用户action编码输入数据。

典型场景:

当用户执行SSP或GND action编码时，使用ActionRequest把里面的内容写进去。

功能描述:

此类为用户action代码的输入数据结构。

接口约束:

数据结构是固定的，里面的原始配置数据是xml格式

- **参数**
  - **neid** (*str*) - 设备ID
  - **action_path** (*str*) - action路径

- **action_xml** (*str*) - action的xml数据

class aoc.base.action.ActionService(logger=None, resource_dir=None)

基类：aoc.base.baseservice.BaseService

系统提供给用户的action接口。

典型场景:

如果需要定义一个服务的O&M功能，可以在YANG中定义一个action 并在此类中实现这个action操作。

功能描述:

此类是 SSP 或 GND 中action的基类。 如果需要定义一个服务的O&M功能，可以在YANG中定义一个action 并在此类中实现这个action操作。

接口约束:

无。

- **参数**
  - **logger** (*str, optional*) - 当前包组的日志打印

logger

当前包组的日志打印

Type： str, optional

action(aoc_context, action_input: aoc.base.action.ActionRequest)

用户需要实现的action函数。

典型场景:

如果需要定义一个服务的O&M功能，可以在YANG中定义一个action 并在此类中实现这个action操作。

功能描述:

用户需要实现的action函数。

- **参数**
  - **action_input** - action 输入参数，有关详细信息，请参见：类：**ActionRequest**。
  - **aoc_context** (*obj*) - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
- **返回**
  - 操作的返回内容为xml格式，如果没有输出定义，则返回空。

示例

```
>>> action_input.neid = 'a7f388b6-e867-3f28-86bc-df809918fb96'
>>> action_input.action_xml = '<input><userName>xxx</userName></input>'
>>> action_input.action_path = '/aaamini:aaaminigndjava'
>>> def action(self, action_input, aoc_context):
        inputData = convert(DictNode(action_input.action_xml))
        neid = action_input.neid
        sspresult = dosomething(inputData, neid)
```

```
    return sspresult
>>> sspresult = action(action_input)
>>> return sspresult
```

3.2 aoc.base.aoccontext module

典型场景:

1. 试运行
2. 向设备提交数据
3. 同步数据
4. 从设备查询数据

功能描述:

描述AOC上下文信息。

class aoc.base.aoccontext.AocContext(transid='', nevendor='', netype='', neversion='', servicepoint='', nodeid='', userId='', neid='')

基类：object

描述AOC上下文信息。

典型场景:

1. 试运行
2. 向设备提交数据
3. 同步数据
4. 从设备查询数据

功能描述:

描述AOC上下文信息。

接口约束:

无。

- **参数**
  - **transid**（*str, optional*）- 事务ID。
  - **nevendor**（*str, optional*）- 设备供应商。
  - **netype**（*str, optional*）- 设备硬件类型。
  - **neversion**（*str, optional*）- 设备软件版本。
  - **servicepoint**（*str, optional*）- 微服务路由节点。
  - **nodeid**（*str, optional*）- 用于路由的微服务实例。
  - **userId**（*str, optional*）- 操作用户ID。
  - **neid**（*str, optional*）- 设备ID。

transactionId

事务ID。

Type： str, optional

deviceVendor

设备供应商。

Type： str, optional

netype

设备硬件类型。

Type： str, optional

neversion

设备软件版本。

Type： str, optional

servicepoint

微服务路由节点。

Type： str, optional

nodeid

用于路由的微服务实例。

Type： str, optional

userId

操作用户ID。

Type： str, optional

serviceInst

Type： str, optional

neid

设备ID。

Type： str, optional

3.3 aoc.base.baseservice module

典型场景:

为open api提供基类。

功能描述:

为open api提供基类。

aoc.base.baseservice.INPUT_PARA_INVALID

无效的参数。

Type: str

class aoc.base.baseservice.BaseService(logger, input_types, output_types)

abstract **do_ncs_map**(map_input**,** aoc_context**,** template=None)

系统提供给用户的映射接口。

典型场景:

用户调用SSP配置，并走到用户的ncs_map的实现以处理映射。

功能描述:

用户调用SSP配置，并走到用户的ncs_map的实现以处理映射。

接口约束:

如果一个用户异常被抛出，那需要通过使用固定的用户异常类。

注解

此函数具有默认实现。如果映射逻辑中的变量不需要额外处理，可以直接使用默认实现。

- **参数**
  - **map_input** (*obj*) - 请求参数
  - **aoc_context** (*obj*) - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
  - **template** (*str*) - 模板路径
- **返回**
  - 业务映射配置结果，xml格式。
- **返回类型**
  - str

实际案例

1. 用户直接将请求数据传递到模板渲染中

```
def ncs_map(self, request, aoc_context=None, template=''):
    result = self.render(template, request.xmldict)
    return result
```

2. 用户需要代码来处理请求，并构造模板参数进行渲染

```
def ncs_map(self, request, aoccontext=None, template=None):
    # get isp service input data
    request_dic_node = self.xmltodictnode(request.xml)
    self.logger.info("bngAaaPairs request_dic_node: %s" % (request_dic_node))
    template_context = self.fill_miniaaapair_context(request_dic_node, aoccontext)
    # query bng data from cdb in isp service
    self.logger.info("bngAaaPairs template_syspair_context: %s" % (template_context))
    # render the j2 template with template_context
    strt = self.render('template_Aaa.j2', template_context)
    self.logger.info("bngAaaPairs template_nepair%s" % (strt))
    return strt
#fill context
def fill_miniaaapair_context(self, request_dic_node, aoccontext):
    self.aaamini = request_dic_node.aaaminignd
    servicepath = aoccontext.serviceInst
    nerootpath = aoccontext.serviceInst[servicepath.find("ne/")+3:]
    neid = nerootpath[:nerootpath.find("/")]
    return {'users':self.aaamini,'neid':neid}
```

***abstract* do_ncs_rpc**(rpc_input**,** aoc_context**,** template=None)

提供RPC操作能力。

典型场景:

系统提供给用户的ncs_rpc接口。

功能描述:

提供RPC操作能力。

接口约束:

rpc_input 必须合法.

- **参数**
  - **rpc_input** (*str*) - rpc输入参数
  - **aoc_context** (*obj*) - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
  - **template** - 模板名称
- **返回**
  - rpc的返回内容，如果没有输出，则返回空
- **返回类型**
  - str

示例

这是一个ncs_rpc函数示例

```
rpc_input.inputData = '<input><userName>xxx</userName></input>'
rpc_input.rpcName = '(nrn:com:huawei)activeUser'
def ncs_rpc(self, rpc_input, aoc_context, template=None):
    sndinputData = convert(rpc_input.inputData)
    neid = getneid(sndinputData)
    sndrpcName = convert(rpc_input.rpcName)
    sndresult = devicemgr.device_rpc(neid, sndinputData, sndrpcName)
    sspresult = convert(sndresult.result)
    return sspresult
>>> print(ncs_rpc(rpc_input))
<output><result>OK</result></output>
```

setup()

类调用前的初始化操作。

典型场景:

此函数用于初始化资源。

功能描述:

此函数用于初始化资源。

接口约束:

无。

实际案例

```
import threading
def fun_timer():
    print('Hello Aoc!')
def setup(self):
    timer = threading.Timer(1, fun_timer)
    timer.start()
```

teardown()

卸载插件包时调用，与setup一起使用。

典型场景:

该函数用于资源释放。

功能描述:

卸载插件包时调用，与setup一起使用。

接口约束:

无。

实际案例

```
def teardown(self):
    timer.cancel()
```


4 aoc.common

4.1 aoc.common.upgrade package

4.1.1 aoc.common.upgrade.package_upgrade module

典型场景:

1. 定制插件包升级的预处理。
2. 定制插件包升级的后处理。

功能描述:

提供插件包升级的定制处理能力。

class aoc.common.upgrade.package_upgrade.PackageUpgrade(logger=None, resource_dir='')

基类：aoc.base.baseservice.BaseService

提供插件包升级的定制处理能力。

典型场景:

1. 定制插件包升级的预处理。
2. 定制插件包升级的后处理。

功能描述:

提供插件包升级的定制处理能力。

接口约束:

无。

参数

- **logger** (*Log, optional*) - 当前包组的日志打印。

- **resource_dir** (*str, optional*) - 资源文件的目录。

logger

当前包组的日志打印。

Type： Logger, optional

resource_dir

资源文件的目录。

Type： str, optional

***abstract* post_upgrade**(aoccontext**,** request=None)

定制插件包升级的后处理。

典型场景:

定制插件包升级的后处理。

功能描述:

定制插件包升级的后处理。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**.
  - **request** (*PkgUpgradeInput*) - 插件包升级的输入参数, 详细信息请参考 proto.aoc.common.upgrade.pkgupgrade.PkgUpgradeInput.

实际案例

```
form aoc.sys import datastore
def pre_upgrade(self, aoccontext, request=None):
    if request is None:
        return
    if request.newPkgInfo is not None:
        # backfill the data of the old plug-in package to the new package..
        global doc
        datastore.write_datastore(aoccontext, doc, '/huawei-ac-applications:applications/
aaamini:aaamini', None)
```

***abstract* pre_upgrade**(aoccontext**,** request=None)

定制插件包升级的预处理。

典型场景:

定制插件包升级的预处理。

功能描述:

定制插件包升级的预处理。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**.
  - **request** (*PkgUpgradeInput*) - 插件包升级的输入参数, 详细信息请参考 proto.aoc.common.upgrade.pkgupgrade.PkgUpgradeInput.

实际案例

```
form aoc.sys import datastore
def pre_upgrade(self, aoccontext, request=None):
    if request is None:
        return
    if request.oldPkgInfo is not None:
        # Obtain the old plug-in package data before the upgrade and cache the data.
        global doc
        doc = datastore.read_datastore_rdb(aoccontext, '/huawei-ac-applications:applications/
aaamini:aaamini')
```

5 aoc.event

5.1 aoc.event package

5.1.1 aoc.event.event_receiver module

典型场景:

接收到事件后自定义通知函数。

功能描述:

自定义接收事件的处理。

class aoc.event.event_receiver.EventReciever(logger=None, resource_dir='')

基类： aoc.base.baseservice.BaseService

自定义接收事件的处理。

典型场景:

接收到事件后自定义通知函数。

功能描述:

自定义接收事件的处理，和 :class: aoc.sys.event_mgr.EventMgr配合使用。

接口约束:

无。

- **参数**
  - **logger** (*Log, optional*) - 当前包组的日志打印。
  - **resource_dir** (*str, optional*) - 资源文件的目录。

logger

通知北向的日志打印。

Type： Logger, optional

resource_dir

资源文件的目录。

Type： str, optional

abstract **notify**(aoc_context**,** event_handler)

设备告警变更通知北向接口。

典型场景:

设备告警变更通知北向接口。

功能描述:

设备告警变更通知北向接口。

接口约束:

无。

- **参数**
  - **aoc_context** (*AocContext*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**.
  - **event_handler** (*AocEventHandler*) - aoc事件信息，详细信息请参考 proto.aoc.event.AocEventHandler.AocEventHandler。

实际案例

```
from aoc.sys.alarm_mgr import AlarmMgr
class EventReceiverTest(EventReciever):
    def notify(self, aoc_context, event_handler):
        alarm_sender = AlarmMgr(self.logger)
        entity_list = event_handler.eventEntity
        if entity_list:
            for entity in entity_list:
                alarm_sender.write_alarm(entity.alarm)
```

6 aoc.exception

6.1 aoc.exception package

6.1.1 aoc.exception.aocexceptions module

典型场景:

本模块提供不同的异常场景。

功能描述:

classes: AOCException, DBException, SysException, DeviceException, TimeoutException, InvalidAugments, UserException。

exception aoc.exception.aocexceptions.AOCException(errcode, *paras)

基类: Exception

典型场景:

这是open-api中所有异常的基本异常。

功能描述:

继承自Exception，提供不同场景的异常。

exception aoc.exception.aocexceptions.DBException(errcode, *paras)

基类：**aoc.exception.aocexceptions.AOCException**

典型场景:

当错误代码与数据库相关时使用。

功能描述

继承自AOCException，提供与数据库相关的异常使用。

exception aoc.exception.aocexceptions.DeviceException(errcode, *paras)

基类：aoc.exception.aocexceptions.AOCException

典型场景:

当错误代码与设备相关时使用。

功能描述:

继承自AOCException，提供与设备相关的异常使用。

exception aoc.exception.aocexceptions.IllegalParameterException(errcode, *paras)

基类：aoc.exception.aocexceptions.AOCException

典型场景:

场景: aoc_context中输入的事务ID为空。 场景2: The input transactionId in aoc_context is not exited.

exception aoc.exception.aocexceptions.InnerException(errcode, *paras)

基类：aoc.exception.aocexceptions.AOCException

典型场景:

当错误与AOC内部相关时，用于引发内部异常的异常。 例如： 根据查询条件在数据库中查找设备信息失败。

功能描述:

此错误发生在AOC内部处理期间，而不是因为用户的脚本。

- **参数**
  - **errcode** (*str*) - 内部异常的自定义错误代码。
  - **paras** (*list*) - 内部异常的错误信息描述。

exception aoc.exception.aocexceptions.InvalidArguments(errcode, *paras)

基类：aoc.exception.aocexceptions.AOCException

典型场景:

在error为无效参数时使用

功能描述:

继承自AOCException。

exception aoc.exception.aocexceptions.LockException(*paras)

基类：aoc.exception.aocexceptions.AOCException

典型场景:

如果路径被其他人锁定，则此函数将等待。如果出现，函数将返回此异常。

功能描述:

继承自AOCException。

接口约束:

用户不能抛出此异常，只能捕获。

exception aoc.exception.aocexceptions.MigrateException(*paras)

基类：aoc.exception.aocexceptions.AOCException

典型场景:

snd包不兼容升级中使用的异常。

功能描述:

此异常发是在snd包不兼容处理时使用

参数

paras (*list*) - snd不兼容升级的失败原因。

接口约束:

用户不能抛出此异常，只能捕获。

exception aoc.exception.aocexceptions.SysException(errcode, *paras)

基类：aoc.exception.aocexceptions.AOCException

典型场景:

当错误代码与系统错误相关时使用。

功能描述:

继承自AOCException，提供与系统相关的异常使用。

exception aoc.exception.aocexceptions.TimeoutException(errcode, *paras)

基类：aoc.exception.aocexceptions.AOCException

典型场景:

当错误代码与超时相关时使用。

功能描述:

继承自AOCException，提供与超时相关的异常使用。

exception aoc.exception.aocexceptions.UserException(*paras)

基类：aoc.exception.aocexceptions.AOCException

典型场景:

这是用户代码使用的异常。

功能描述:

继承自AOCException。

接口约束:

用户不能抛出此异常，只能捕获。

aoc.exception.aocexceptions.err_to_exception(errcode, *paras)

典型场景:

用于特定的错误码转换异常。

功能描述:

该函数用于构造不同错误码的特定异常。

注:

不要在“Arguments”部分中包括“self”参数。

- **参数**
  - **errcode**（*str*）- 错误码。
  - ***paras**（*list*）- V错误信息描述。
- **返回**
  - 当错误码不在_ERROR_CODE_MAP中，将获得特定的异常。
- **返回类型**
  - AOCException

实际案例

```
raise err_to_exception(“publicinfocode.input.para.invalid.0x00c80035”,“ne_id”)
```

6.1.2 aoc.exception.public_info_code module

典型场景:

该模块用于构造特定的异常。

功能描述:

该模块提供不同的错误码和ERROR_CODE_MAP。

7 aoc.gnd

7.1 aoc.gnd package

7.1.1 aoc.gnd.GndRpc module

典型场景:

该模块提供GND rpc挂钩。用户定义的GND YANG模型包含rpc。当在北向api中调用gnd的rpc时，机制会自动回调用户python脚本实现的rpc处理回调。

功能描述:

此模块提供用户实现的rpc接口。

class aoc.gnd.GndRpc.GndRpc(logger=None, resource_dir='')

基类：aoc.base.baseservice.BaseService

GND层rpc处理接口

典型场景:

此类提供GND rpc钩子。用户定义的GND Yang模型包含rpc。当在北向api中调用gnd的rpc时， 机制会自动回调用户python脚本实现的rpc处理回调。

功能描述:

该类提供rpc接口供用户实现。

接口约束:

rpc接口的输入输出参数为xml格式。

- **参数**
  - **logger** (*str, optional*) - 当前包组的日志打印
  - **resourceDir** (*str*) - 资源所在目录

```
logger
```


当前包组的日志打印

Type： str, optional

resourceDir

资源所在的目录

Type： str

***abstract* rpc**(aoccontext, request=None)

系统提供给用户的rpc方法。

典型场景:

该方法提供GND rpc挂钩。用户定义的GND YANG模型包含rpc。当在北向api中调用gnd的rpc时，机制会自动回调用户python脚本实现的rpc处理回调。

功能描述:

提供gnd rpc操作能力。

接口约束:

rpc接口的输入输出参数为xml格式。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*str, optional*) - rpc接口的输入参数，包括rpc的名称、网元的ID和rpc的输入xml格式。
- **返回**
  - rpc的输出数据是xml格式的，如果没有输出，则为空。
- **返回类型**
  - str

示例

这里有一个例子来说明rpc函数的输入和输出参数。

```
from aoc.sys import devicemgr
def rpc(self, aoccontext, request=None):
    sndinputData = convert(request.inputData)
    sndrpcName = convert(request.rpcName)
    sndresult = devicemgr.device_rpc(request.neid, sndinputData,sndrpcName)
    gndresult = convert(sndresult.result)
    output = RpcResult()
    output.outputData = gndresult
    return output
```

7.1.2 aoc.gnd.alarm_receiver module

典型场景:

1. AOC无法自动处理的设备告警。

功能描述:

提供设备告警处理能力。

class aoc.gnd.alarm_receiver.AlarmReceiver(logger=None, resource_dir='')

基类： aoc.base.baseservice.BaseService

提供设备告警处理能力。

典型场景:

自定义处理告警数据的方法。

功能描述:

提供设备告警处理能力。

接口约束:

无。

- **参数**
  - **logger** (*Log, optional*) - 当前包组的日志打印。
  - **resource_dir** (*str, optional*) - 资源文件的目录。

logger

告警的日志打印。

Type： Logger, optional

resource_dir

资源文件的目录。

Type： str, optional

***abstract* load_configuration**(aoc_context)

获取告警配置。

典型场景:

从系统加载配置。

功能描述:

获取告警配置。

- **参数**
  - **aoc_context** (*AocContext*) - 上下文信息，详细信息请参考 aoc.base.aoccontext.
- **返回**
  - 告警配置信息。
- **返回类型**
  - 它由proto文件定义，路径为 proto.aoc.event.AocConfiguration.AocAlarmConfigEntity

实际案例

```
def load_configuration(self, aoc_context):
    fp = open(os.path.join(self.resource_dir, "resources", "alarm_cfg.json"))
    cfg_json = json.load(fp)
    subscription = AocNcfNotificationSubscription()
```



```
    subscription.isSupported = cfg_json['AocNcfNotificationSubscription.isSupported']
    subscription.filter = cfg_json['AocNcfNotificationSubscription.filter']
    config = AocAlarmConfigEntity()
    config.subscription = subscription
    return config
```

***abstract* netconf_notification_handle**(aoc_context, netconf_notification)

自定义处理来自设备的netconf通知。

典型场景:

自定义告警处理的实现方法。

功能描述:

自定义处理来自设备的netconf通知。

接口约束:

此抽象方法是为netconf协议设计的。

- **参数**
  - **aoc_context** (*AocContext*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**.
  - **netconf_notification** (*List*) - netconf通知列表，详细信息请参考 proto.aoc.gnd.AocNcfNotification.AocNcfNotificationsEntity.

实际案例

```
def netconf_notification_handle(self, aoc_context, netconf_notification):
    alarm_data = AocAlarmEntity()
    if netconf_notification.key == '0x0de42013':
        alarm_data.alarmId = '0x0de42013'
        alarm_data.alarmName = 'hwSpsUcssLinkDownTrap'
        alarm_data.additionalInformation = AocNcfNotificationsEntity.notification
        alarm_mgr.write_alarm(alarm_data)
    else:
        logger.info('this is not hwSpsUcssLinkDownTrap alarm')
```

***abstract* trap_handle**(aoc_context, trap_entity_list)

snmp Trap的自定义处理。

典型场景:

snmp Trap的自定义实现方法。

功能描述:

snmp Trap的自定义处理。

接口约束:

这个抽象方法是为snmp Trap设计的。

- **参数**
  - **aoc_context** (*AocContext*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**.
  - **trap_entity_list** (*List*) - snmp Trap列表，详细信息请参考 proto.aoc.gnd.AocSnmpTrap.AocSnmpTrapsEntity.

实际案例

```
from aoc.event.event_model_pb2.AocAlarm_pb2.py import AocAlarmEntity
from aoc.gnd.gnd_model_pb2.AocSnmpTrap_pb2 import AocSnmpTrapsEntity
class AlarmReceiverTest(AlarmReceiver):
    def trap_handle(self, aoc_context, trap_entity_list):
        if isinstance(trap_entity_list, list):
            for snmp_tramp in trap_entity_list:
                if (snmp_tramp.trapOidEntity.oid == '1.3.6.1.4.1.2011.6.122.20.2.2.15'):
                    alarm_data = AocAlarmEntity()
                    alarm_data.alarmId = '0x0de42013'
                    alarm_data.alarmName = 'hwSpsUcssLinkDownTrap'
                    alarm_mgr.write_alarm(alarm_data)
```

7.1.3 aoc.gnd.gnd_action module

典型场景:

如果需要定义服务的运维功能，可以在YANG中定义action，并在类中实现action。

功能描述:

这是一个GND Action类，涉及网元级action操作。 如果需要定义服务的运维功能，可以在YANG中定义action，并在类中实现action。

class aoc.gnd.gnd_action.GndAction(logger=None, resource_dir=None)

基类：aoc.base.action.ActionService

GND层action处理接口。

典型场景:

如果需要定义服务的运维功能，可以在YANG中定义action，并在类中实现action。

功能描述:

这是一个GND Action类，涉及网元级action操作。 如果需要定义服务的运维功能，可以在YANG中定义action，并在类中实现action。

接口约束:

无。

- **参数**
  - **logger** (*str, optional*) - 当前包组的日志打印

7.1.4 aoc.gnd.linkservice module

典型场景:

设备上线时，使用SNMP或定制采集设备侧的链路信息。

功能描述:

提供采集设备侧链路信息方法，提供以下函数： is_link_discovery_by_snmp, link_discovery_customized.

class aoc.gnd.linkservice.LinkService(logger=None, resource_dir='')

基类：aoc.base.baseservice.BaseService

典型场景:

设备上线时，使用SNMP或定制采集设备侧的链路信息。

功能描述:

提供采集设备侧链路信息方法，提供以下函数： is_link_discovery_by_snmp, link_discovery_customized。

***abstract* is_link_discovery_by_snmp**(aoccontext, device_discovery)

是否使用SNMP收集设备侧链路信息。

典型场景:

1. 当设备上线时，使用SNMP去采集设备侧的链路信息。
2. 设备上线后，周期性通过SNMP采集设备链路信息。

功能描述:

该接口用于定制时候通过SNMP采集设备链路信息。

接口约束:

1. 设设备支持SNMP且已配置SNMP协议信息
2. 设备管理时，配置SNMP协议信息。
3. 当前SNMP通用链路采集只支持一下列表中的MIB对象。
4. 以下列表罗列了支持SNMP通用链路采集的MIB对象。

MIB	NAME	OID
LLDP-MIB	lldpRemChassisIdSubtype	1.0.8802.1.1.2.1.4.1.1.4
	lldpRemChassisId	1.0.8802.1.1.2.1.4.1.1.5
	lldpRemPortIdSubtype	1.0.8802.1.1.2.1.4.1.1.6
	lldpRemPortId	1.0.8802.1.1.2.1.4.1.1.7
	lldpRemSysName	1.0.8802.1.1.2.1.4.1.1.9
	lldpRemSysDesc	1.0.8802.1.1.2.1.4.1.1.10

说明

该接口用于定制是否通过SNMP收集设备链路信息。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，有关详细信息，请参见 aoc.base.aoccontext.
  - **device_discovery** (*obj*) - 收集设备信息，具体操作请参见 proto.aoc.gnd.DeviceDiscovery.DiscoveryDeviceInfo.

- **返回**
  - true: 使用SNMP采集；false: 表示不使用SNMP采集数据。
- **返回类型**
  - 由proto文件定义，路径为 proto.aoc.gnd.LinkDiscovery.LinkDiscoveryConfig

实际案例

如果使用SNMP采集数据，示例如下:

```
# If SNMP is used for data collection, the following is an example:
def is_link_discovery_by_snmp(self, aoccontext, device_discovery):
    config = LinkDiscoveryConfig()
    config.discoverBySnmp = true
    return config
```

***abstract* link_discovery_customized**(aoccontext**,** device_discovery)

通过自定义方式收集设备侧链路信息。

典型场景:

1. 设备上线时，设备侧会采集用户指定的链路信息。
2. 设备上线后，设备链路信息设备侧链路信息会周期性的以用户指定模式进行采集。

功能描述:

该接口用于定制设备链路信息的采集。

接口约束:

当前仅支持采集二层链路信息（对端设备标识符类型为 MAC 地址）。

说明

该接口用于定制设备侧链路采集。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，有关详细信息，请参见 **aoc.base.aoccontext**.
  - **device_discovery** (*obj*) - 收集设备信息，具体操作请参见 proto.aoc.gnd.DeviceDiscovery.DiscoveryDeviceInfo.
- **返回**
  - 设备链路信息列表。
- **返回类型**
  - 由proto文件定义，路径为proto.aoc.gnd.LinkDiscovery.LinkDiscoveryOutput

实际案例

以自定义方式收集设备侧链路信息，示例如下:

```
from aoc import devicemgr
def link_discovery_customized(self, aoccontext, device_discovery):
    neid = '868e778e-153c-3afe-a02c-89678e31e3e4'
    path = 'huawei-lldp:lldp'
    response = devicemgr.query_data_from_device(neid,'CONFIGURATION', path)
```



```
>>> print(response)
<lldp xmlns="urn:huawei:params:xml:ns:yang:huawei-lldp">
   <remote-port>
      <index>1</index>
      <chassis-id>2944-aa03-9126</chassis-id>
      <port-id-sub-type>5</port-id-sub-type>
      <remote-management>
         <subtype>1</subtype>
         <address>172.16.40.92</address>
         <if-subtype>2</if-subtype>
         <if-id>210</if-id>
         <oid>0.6.15.43.6.1.4.1.2011.5.25.41.1.2.1.1.1</oid>
      </remote-management>
      <chassic-id-sub-type>4</chassic-id-sub-type>
      <system-description>Huawei</system-description>
      <port-description/>
      <port-id>GigabitEthernet0/0/0</port-id>
      <system-name>AP4050DN-E-039126</system-name>
      <system-cap-supported>cap-bridge</system-cap-supported>
      <system-cap-enabled>cap-bridge</system-cap-enabled>
   </remote-port>
</lldp>
# Then convert the xml message to LinkDiscoveryOutput
```

7.1.5 aoc.gnd.notification module

典型场景:

设备上报的netconf通知的自定义处理。

功能描述:

处理设备netconf通知。

class aoc.gnd.notification.Notification(logger=None, resource_dir='')

基类： aoc.base.baseservice.BaseService

处理设备netconf通知。

典型场景:

设备上报的netconf通知的自定义处理。

功能描述:

处理设备netconf通知。

接口约束:

无。

- **参数**
  - **logger** (*Log, optional*) - 当前包组的日志打印。
  - **resource_dir** (*str, optional*) - 资源文件的目录。

logger

通知北向的日志打印。

Type： Logger, optional

resource_dir

资源文件的目录。

Type： str, optional

abstract **receive**(aoc_context**,** request)

提供处理设备通知的能力。

典型场景:

设备上报的netconf通知的自定义处理。

功能描述:

提供处理设备通知的能力。

接口约束:

无。

- **参数**
  - **aoc_context** (*AocContext*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**.
  - **request** (*NotificationData*) - 通知数据, 详细信息请参考 proto.aoc.gnd.Notification.NotificationData.

实际案例

```
def receive(self, aoc_context, request):
    alarm_data = AocAlarmEntity()
    if request.key == '0x0de42013':
        alarm_data.alarmId = '0x0de42013'
        alarm_data.alarmName = 'hwSpsUcssLinkDownTrap'
        alarm_data.additionalInformation = AocNcfNotificationsEntity.notification
        alarm_mgr.write_alarm(alarm_data)
    else:
        logger.info('this is not hwSpsUcssLinkDownTrap alarm')
```

7.1.6 aoc.gnd.portservice module

典型场景:

设备上线时，使用该模块定制端口采集设备端口信息。

功能描述:

该模块提供定制端口采集的方法，主要提供方法：is_port_discovery_by_snmp, port_discovery_customized.

class aoc.gnd.portservice.PortService(logger=None, resource_dir='')

基类： **aoc.base.baseservice.BaseService**

典型场景:

设备上线时，使用该模块定制端口采集设备端口信息。

功能描述:

该模块提供定制端口采集的方法，主要提供方法： is_port_discovery_by_snmp, port_discovery_customized.

***abstract* is_port_discovery_by_snmp**(aoccontext, device_discovery)

该接口用于定制是否通过SNMP收集设备接口信息。

典型场景:

1. 设备上线时，使用SNMP采集设备端口信息。
2. 设备上线后，周期性通过SNMP采集设备端口信息。

功能描述:

该接口用于定制是否通过SNMP采集设备端口信息。

接口约束:

1. 设备支持SNMP且已配置SNMP协议信息。
2. 设备管理时，配置SNMP协议信息。
3. 当前SNMP通用端口采集只支持一下列表中的MIB对象。
4. 以下列表罗列了支持SNMP通用端口采集的MIB对象。

MIB	NAME	OID
IF-MIB	ifIndex	1.3.6.1.2.1.2.2.1.1
	ifDescr	1.3.6.1.2.1.2.2.1.2
	ifType	1.3.6.1.2.1.2.2.1.3
	ifMtu	1.3.6.1.2.1.2.2.1.4
	ifPhysAddress	1.3.6.1.2.1.2.2.1.6
	ifAdminStatus	1.3.6.1.2.1.2.2.1.7
	ifOperStatus	1.3.6.1.2.1.2.2.1.8
IP-MIB	ipAdEntIfIndex	1.3.6.1.2.1.4.20.1.2
	ipAdEntNetMask	1.3.6.1.2.1.4.20.1.3
IPV6-MIB	ipv6AddrPfxLength	1.3.6.1.2.1.55.1.8.1.2
	ipv6AddrType	1.3.6.1.2.1.55.1.8.1.3

说明

该接口用于定制是否通过SNMP收集设备接口信息。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，有关详细信息，请参见 **aoc.base.aoccontext**.
  - **device_discovery** (*obj*) - 采集设备信息，具体操作请参见 proto.aoc.gnd.DeviceDiscovery.DiscoveryDeviceInfo.
- **返回**

- true: 使用SNMP收集; false: 表示不使用SNMP采集数据。

- **返回类型**
  - 由proto文件定义，路径为 proto.aoc.gnd.PortDiscovery.PortDiscoveryConfig

实际案例

如果使用SNMP采集数据，示例如下:

```
# If SNMP is used for data collection, the following is an example:
def is_port_discovery_by_snmp(self, aoccontext, device_discovery):
    config = PortDiscoveryConfig()
    config.discoverBySnmp = true
    return config
```

abstract **port_discovery_customized**(aoccontext**,** device_discovery)

自定义方式收集设备侧接口信息。

典型场景:

1. 设备上线时，设备端口信息采集通过用户定制模式执行。
2. 设备上线后，用户定制的采集模式将周期性采集设备端口信息。

功能描述:

通过用户定制模式采集设备端口信息

接口约束:

端口信息采集范围被PortDiscoveryOutput中定义的范围所限制。

说明

该接口用于定制设备侧接口集合。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，有关详细信息，请参见 **aoc.base.aoccontext**.
  - **device_discovery** (*obj*) - 采集设备信息，具体操作请参见 proto.aoc.gnd.DeviceDiscovery.DiscoveryDeviceInfo.
- **返回**
  - 设备接口信息列表。
- **返回类型**
  - 由proto文件定义，路径为 proto.aoc.gnd.PortDiscovery.PortDiscoveryOutput

实际案例

以自定义方式收集设备侧接口信息为例:

```
from aoc import devicemgr
def port_discovery_customized(self, aoccontext, device_discovery):
neid = '868e778e-153c-3afe-a02c-89678e31e3e4'
path = 'ietf-interfaces:interfaces-state'
response = devicemgr.query_data_from_device(neid,'CONFIGURATION', path)
>>> print(response)
<interfaces-state xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
```



```
<interface>
<name>GigabitEthernet0/0/0</name>
<if-index>1</if-index>
<type>other</type>
<admin-status>up</admin-status>
<oper-status>down</oper-status>
<phys-address>00:51:61:6f:c4:28</phys-address>
</interface>
</interfaces-state>
# Then convert the xml message to PortDiscoveryOutput
```


8 aoc.ncs

8.1 aoc.ncs package

8.1.1 aoc.ncs.filter module

典型场景:

当用户进行SSP映射时，它通过jinja模板渲染网元配置。 可以在jinja模板中调用该模块的方法来转换IP或子网。

功能描述:

本模块提供了过滤方法，供用户通过jinja模板调用。

class aoc.ncs.filter.Filter

aoc.ncs.filter.ipaddr(value, query='', version=False)

提供IP地址转换能力。

典型场景:

当用户对服务进行编码时，他们通常需要转换或计算IP地址并调用此函数方法。

功能描述:

提供IP地址转换能力。

接口约束:

ipaddr仅支持地址、广播、ip、网关、前缀、子网、类型、路由、大小、主机掩码。

- **参数**
  - **value** (*str*) - 输入参数的内容，如地址
  - **query** (*str, optional*) - 要执行的转换方法的名称(仅可以采用以下值：’address’,’broadcast’,’ip’,’net’,’netmask’,’network’,’prefix’,’subnet’,’type’,’cidr ‘,’size’,’hostmask’) 或数字。

- **version** (*bool, optional*) - True if ipv6, False if ipv4.

- **返回**
  - 如果值检查失败，则为False，否则为转换后的数据内容。
- **返回类型**
  - str

示例

这里有一个例子来说明调用方法。

```
>>> address = '192.168.32.1/24'
>>> print(ipaddr(address, 'address'))
192.168.32.1
>>> address = '10.1.1.1/25'
>>> print(ipaddr(address, 'address'))
10.1.1.1
>>> address = '10.12.1.34/24'
>>> print(ipaddr(address, 'broadcast'))
10.12.1.255
>>> address = '10.12.1.34/16'
>>> print(ipaddr(address, 'broadcast'))
10.12.255.255
>>> address = '10.1.1.1/25'
>>> print(ipaddr(address, 'net'))
None
>>> address = '10.1.1.0/24'
>>> print(ipaddr(address, 'net'))
10.1.1.0/24
>>> address = '192.168.32.0/24'
>>> print(ipaddr(address, 'netmask'))
255.255.255.0
>>> address = '10.1.1.1/25'
>>> print(ipaddr(address, 'netmask'))
255.255.255.128
>>> address = '10.12.1.34/32'
>>> print(ipaddr(address, 'network'))
10.12.1.34
>>> address = '10.12.1.34/255.255.255.255'
>>> print(ipaddr(address, 'network'))
10.12.1.34
>>> address = '192.168.32.1/24'
>>> print(t.ipaddr(address, 'prefix'))
24
>>> address = '10.1.1.1/25'
>>> print(t.ipaddr(address, 'prefix'))
25
>>> address = ('10.1.1.1/24', '30')
>>> print(t.ipaddr(*address, 'subnet'))
64
>>> address = ('10.1.1.1/25', '24')
>>> print(t.ipaddr(*address, 'subnet'))
0
>>> address = '192.168.32.0/24'
>>> print(ipaddr(address, '0'))
192.168.32.0/24
>>> print(ipaddr(address, '1'))
192.168.32.1/24
>>> address = '10.1.1.1/25'
>>> print(ipaddr(address, 'hostmask'))
0.0.0.127
```



```
>>> address = '10.12.1.34/32'
>>> print(ipaddr(address, 'hostmask'))
0.0.0.0
```

aoc.ncs.filter.ipsubnet(value, query='', index='x')

系统提供给用户的IP子网处理功能。

典型场景:

当用户对服务进行编码时，他们通常需要转换或计算IP子网并调用此函数方法。

功能描述:

ipsubnet函数可用于ip子网处理方法。

接口约束:

Value参数的格式必须为’10.1.1.1/24’，query参数格式必须为’30’

注解

索引默认值为x。

- **参数**
  - **value** (*str*) - 入参内容，如地址、掩码等
  - **query** (*str, optional*) - 要执行的转换方法的名称(仅可以采取以下值：’address’,’broadcast’,’ip’,’net’,’netmask’,’network’,’prefix’,’subnet’,’type’,’cidr ‘,’size’,’hostmask’) 或数字.
- **返回**
  - 如果值检查失败，则为False，否则为转换后的数据内容。
- **引发**:
  - **BaseException** - 1.您必须传递有效的子网或IP地址，IP地址无效，用户输入的参数与IP地址不匹配或子网规格。 2.IP地址不在子网中，用户输入的参数与子网不一致规格。
  - **InvalidArguments** - 参数无效。
  - **ValueError** - 值无效或查询无效。
  - **StopIteration** - 非法输入

示例

这里有一个例子来说明调用方法。

```
>>> address = ('10.1.1.1/24', '30')
>>> print(t.ipsubnet(*address))
64
>>> address = ('10.1.1.1/25', '24')
>>> print(t.ipsubnet(*address))
0
```

8.1.2 aoc.ncs.ncs_action module

典型场景:

如果需要定义一个服务的O&M功能，可以在YANG中定义一个action

并在此类中实现这个action操作。

功能描述:

这是一个SSP Action类，涉及网络级action操作。 如果需要定义服务的运维功能，可以在YANG中定义action，并在类中实现action。

class aoc.ncs.ncs_action.NcsAction(logger=None, resource_dir=None)

基类：aoc.base.action.ActionService

典型场景:

如果需要定义一个服务的O&M功能，可以在YANG中定义一个action 并在此类中实现这个action操作。

功能描述:

这是一个SSP Action类，涉及网络级action操作。 如果需要定义服务的运维功能，可以在YANG中定义action，并在类中实现action。

接口约束:

无。

- **参数**
  - **logger** (*str, optional*) - 当前包组的日志打印

8.1.3 aoc.ncs.ncsservice module

典型场景:

当用户进行SSP映射时，他们输入网络层数据，输出网络元素层数据。该模块提供用户实现的接口。

功能描述:

该模块提供用户编码的映射接口。

class aoc.ncs.ncsservice.ChangeData(path, new_data, old_data, operation_type)

基类：object

用户执行后修改输入数据。

典型场景:

当用户执行postModify业务时，它将操作一个业务实例，其中包括该业务实例数据。

功能描述:

此类为用户postModify代码的输入数据结构。

接口约束:

无。

- **参数**
  - **path** (*str*) - 变更业务实例路径

- **new_data** (*str*) - 变更业务实例修改后数据
- **old_data** (*str*) - 变更业务实例修改前数据
- **operation_type** (*int*) - 业务实例操作类型， 操作类型包含: Create, Merge, Delete

path

变更业务实例路径

Type： str

new_data

更业务实例修改后数据

Type： str

old_data

变更业务实例修改前数据

Type： str

operation_type

业务实例操作类型， 操作类型包含: Create, Merge, Delete

Type： int

get_new_data()

获取ChangeData新数据。

典型场景:

调用此方法以获取此操作的新数据。

功能描述:

调用此方法以获取此操作的新数据。

接口约束:

无。

- **返回**
  - 此次操作业务实例新数据
- **返回类型**
  - new_data(str)

实际案例

这里有一个例子来说明Change-data获取新数据函数的输入和输出参数。

```
>>> print ChangeData.get_new_data
'data context'
```

get_old_data()

获取ChangeData旧数据。

典型场景:

调用此方法以获取此操作的旧数据。

功能描述:

调用此方法以获取此操作的旧数据。

接口约束:

无。

- **返回**
  - 此次操作业务实例旧数据
- **返回类型**
  - old_data(str)

实际案例

这里有一个例子来说明Change-data获取旧数据函数的输入和输出参数。

```
>>> print ChangeData.get_old_data
'data context'
```

get_operation_type()

获取ChangeData操作类型。

典型场景:

调用此方法以获取操作类型。

功能描述:

调用此方法以获取操作类型。

接口约束:

无。

- **返回**
  - CREATE = 0; MERGE = 1; DELETE = 2;
- **返回类型**
  - Operation type(int)

实际案例

这里有一个例子来说明Change-data获取操作类型函数的输入和输出参数。

```
>>> print ChangeData.get_operation_type
1
```

get_path()

获取ChangeData服务路径。

典型场景:

调用此方法以获取此操作的业务路径。

功能描述:

调用此方法以获取此操作的业务路径。

接口约束:

无。

- **返回**
  - 此次操作业务实例路径
- **返回类型**
  - path(str)

实际案例

这里有一个例子来说明Change-data获取业务路径函数的输入和输出参数。

```
>>> print ChangeData.get_path
'/huawei-ac-applications:applications/huawei-ncs-eas:eas/eas_1'
```

class aoc.ncs.ncsservice.DictNode(mapping=None)

基类：dict

用户映射过程中可能使用的数据格式。

典型场景:

在用户定义SSP YANG模型后，传输的数据格式由机制是xml。xml格式不容易获取值的字段，使用DictNode方便处理。

功能描述:

该类别为用户提供了易于使用的YANG数据承载格式操作。

接口约束:

输入数据格式为xml格式。

- **参数**
  - **mapping**（*str,optional*）- xml格式的数据

key(map_content=None, **kwds)

系统为用户提供过滤获取到的DictNode值的能力。

典型场景:

用户按键过滤dict的内容。

功能描述:

系统将功能键提供给用户的DictNode，以获取值过滤能力。

接口约束:

无。

- **参数**
  - **map_content**（*dict, optional*）- 过滤器的内容，字典格式
  - **kwds**（*dict, optional*）- 多个关键字参数作为过滤条件，每个参数作为单独的条件， 同时满足多个条件。
- **返回**
  - 筛选后的词典的内容。

实际案例


```
xml = '''
<company xmlns="http://example.com/demotemplates">
    <address>nanjing</address>
    <employee>
        <wang>
            <name>wangwu</name>
            <age>28</age>
        </wang>
        <li>
            <name>lisi</name>
            <age>26</age>
        </li>
        <wang>
            <name>wangliu</name>
            <age>21</age>
        </wang>
        <wang>
            <name>wangwu</name>
            <age>20</age>
        </wang>
        <s-s>special</s-s>
    </employee>
</company>'''
def test_case(self):
    a = xmltodict.parse(self.xml, encoding='utf-8')
    b = DictNode(a)
    i = 0
    for val in b.company.address:
        i +=1;
        self.assertTrue(len(val) == 1)
    self.assertEqual(i, 1)
    self.assertEqual(len(b.company.employee), 1)
    self.assertEqual(len(b.company.employee.wang), 3)
    self.assertEqual(b['company']['employee']['wang'].key({'name': 'wangliu'}).age, '21')
    self.assertEqual(b['company']['employee']['wang'][0].age, '28')
    self.assertEqual(b['company']['employee']['wang'].key({'name': 'wangwu'})[1].age, '20')
    self.assertEqual(b.company.employee['s-s'], 'special')
```

```
put(key, value)
```

系统提供给用户的DictNode分配函数。

典型场景:

用户在使用dict中添加内容。

功能描述:

系统提供给用户的DictNode分配函数。

接口约束:

键和值都不能为空。

- **参数**
  - **key** - 字典的key值
  - **value** - 字典的value值

实际案例

```
>>> ne.put('nexml', nexml)
```


class aoc.ncs.ncsservice.MapRequest(xml, context, dryrun_context=None)

基类： object

用户映射编码输入数据。

典型场景:

当用户在进行SSP业务映射编码时，使用MapRequest把里面的内容放进去。

功能描述:

此类为用户映射代码的输入数据结构。

接口约束:

数据结构是固定的，里面的原始配置数据是xml格式。

- **参数**
  - **xml** (*str*) - xml格式的数据
  - **context** (*obj*) - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
  - **dryrun_context** (*obj*) - 试运行的上下文信息

***property* xmldict**

系统提供接口供用户获取dict。

典型场景:

用户获取MapRequest后，获取dict格式的数据。

功能描述:

函数xmldict接口，用于获取dict。

接口约束:

xml格式标准可以转换为dict。

- **返回**
  - 用于xml数据转换的dict。
- **返回类型**
  - dict

实际案例

```
>>> request = MapRequest(map_input.config.data, dict(map_input.contextMap))
>>> print(request.xmldict())
{'aaa':'xxx'}
```

***property* xmldictnode**

系统提供接口供用户获取dictNode。

典型场景:

用户获取MapRequest后，获取dictNode格式的数据。

功能描述:

函数用于获取xmldictnode的接口。

接口约束:

xml格式标准可以转换为dictNode。

- **返回**
  - xml数据转换的dict
- **返回类型**
  - dict

实际案例

```
>>> request = MapRequest(map_input.config.data, dict(map_input.contextMap))
>>> print(request.xmldictnode())
{'aaa':'xxx'}
```

***class* aoc.ncs.ncsservice.NcsService**(logger=None**,** resource_dir='')

基类：**aoc.base.baseservice.BaseService**

用于映射用户继承映射的父类。

典型场景:

当用户实现自己的类继承父类后，实现ncs_map方法处理映射逻辑。

功能描述:

该类提供了一个映射父类供用户继承。

接口约束:

用户代码处理jinja模板的呈现。

- **参数**
  - **logger** (*str, optional*) - 当前包组的日志打印
  - **resource_dir** (*str*) - 资源目录

add_filters()

系统提供了一个接口，供用户复制和添加自定义过滤器。

典型场景:

用户调用用户jinja模板中的用户定义过滤器的函数。

功能描述:

用于用户复制，以添加自定义筛选器接口的函数。

接口约束:

不要重复机制的过滤器名称。

- **返回**
  - 过滤dict内容
- **返回类型**
  - dict

实际案例

```
>>> def add_filters(self):
>>>     return {'doSomething':doSomething}
>>> def doSomething(self):
>>>     return'something'
```

discover(discover_input, aoc_context)

业务恢复接口，由第三方用户实现。

典型场景:

用户需要将网元的配置从现有网络恢复到业务的配置。该函数从网元库读取数据，然后返回相应的业务实例。

功能描述:

提供业务恢复实施。

接口约束:

1. 网元配置须先同步到控制器。
2. 在恢复过程中，根据业务数据量批量处理，机制传递上下文控制恢复是否结束。

说明

当业务量规格较大时，为了避免内存溢出和超时，可以定制该函数被系统多次调用，每次处理少量业务实例。

- **参数**
  - **discover_input** (*obj*) -
    - 业务恢复输入，输入参数以protobuf格式定义

```
message DiscoverInput {
    repeated DeviceInfo deviceInfo; # device info
    string discoveryId; # discovery id
    repeated ServiceContext serviceContext;
    repeated ServiceInstance existService;
    string aocNode;
    map<string, string> cacheContext; # Restore context,
    used for multiple restore processing
}
message ServiceContext {
    string servicePath;
    map<string, string> context;
}
message DeviceInfo {
    string deviceId;
    string deviceName;
    string attributes;
}
```

  - **aoc_context** (*obj*) - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
- **返回**
  - 恢复发现的业务实例的配置数据

```
message DiscoverOutput {
    repeated ServiceConfig serviceConfig; # Business instance data, list
    bool goOn; # Whether to continue to restore, the default is false
    map<string, string> cacheContext; # Restore context, used for multiple restore
processing
}
```



```
message ServiceConfig {
    string servicePath; # Business instance path
    string serviceData; # Business instance data
    map<string, string> context; # Business instance context,generally used for
resource occupation
    enum ServiceOperation {
        MERGE;
        REPLACE;
    }
    ServiceOperation operation; # Business instance operation code
}
```

实际案例

1. 当业务量不大时，用户可以选择一次恢复所有业务实例

```
def discover(self, discover_input, aoc_context):
    # Get device information from discover_input
    device_info_list = getDeviceinfolist(discover_input)
    result = DiscoverOutput()
    # Traverse the device information in the input information list
    for device_info in device_info_list:
        device_id = device_info.deviceId
        path ='/huawei-ac-nes:inventory-cfg/nes/ne/'+ deviceId +'/huawei-tunnel:tunnels'
        dbresult = datastore.read_datastore_rdb(aoc_context, path)
        tunnels = self.xmltodictnode(dbresult)
        self.handler_tunnels(tunnels, device_id, result)
    return result
```

2. 当业务量较大时，用户选择批量恢复

```
def discover(self, discover_input, aoc_context):
    # Get device information from discover_input
    device_info_list = getDeviceinfolist(discover_input)
    # Business volume limit
    tunnels_max_num = 100
    cache_context = discoverInput.cacheContext
    result = DiscoverOutput()
    # Traverse the device information in the input information list
    for device_info in device_info_list:
        device_id = device_info.deviceId
        path ='/huawei-ac-nes:inventory-cfg/nes/ne/'+ deviceId +'/huawei-tunnel:tunnels'
        dbresult = datastore.read_datastore_rdb(aoc_context, path)
        tunnels = self.xmltodictnode(dbresult)
        # According to the business mark in cache_context,
        # exclude the business that has been restored,
        # and obtain the newly restored business
        tunnels_new = getNewTunnels(cache_context, tunnels)
        self.handler_tunnels(tunnels_new, device_id, result)
        if (getTunnelsSize(result)> tunnels_max_num)
            result.goOn = true
            result.cacheContext = updateContext(tunnels_new, cache_context)
            break
    return result
```

get_global(key)

系统提供接口供用户获取全局参数。

典型场景:

系统提供接口供用户编码和获取全局参数。

功能描述:

系统提供接口供用户获取全局参数。

接口约束:

无。

说明

该函数解决了Jinja的原生set和get无法传递跨模板设置变量的问题。

- **参数**
  - **key** - key值
- **返回**
  - 业务映射配置结果，xml格式
- **返回类型**
  - str

实际案例

jinja context:

```
<aaa xmlns="http://www.huawei.com/netconf/vrp/huawei-aaa">
  <tasks>
    <task>
      <taskName>{{GET_GLOBAL('xxx')}}</taskName>
    </task>
  </tasks>
  {% if isis == 'true' %}
    <taskGroups>{% for ww in wws %}
      <taskGroup>
        <taskGroupName>{{ww.xx}}</taskGroupName>
        <taskGroupName1>{{ww.yy}}</taskGroupName1>
      </taskGroup>{% endfor %}
    </taskGroups>
  {% endif %}
</aaa>
```

is_snd(nename, sndid)

根据入参nename和sndid检查设备是否绑定了snd包。

典型场景:

需要检查设备是否绑定了snd包来进行下一步的动作。

功能描述:

根据入参nename和sndid检查设备是否绑定了snd包。

接口约束:

无。

- **参数**
  - **nename** (*str*) - 设备的名称。
  - **sndid** (*str*) - snd包的ID。
- **返回**
  - 如果名称为nename的设备由ID为sndid的插件包管理，则为True，否则为False

- **返回类型**
  - bool

实际案例

描述样例：

```
>>> nename = 'test1'
>>> sndid = 'NESPC800'
>>> isSnd = is_snd(neid, sndid)
>>> print isSnd
True
```

ncs_map(request, aoc_context=None, template='')

系统提供给用户的映射接口。

典型场景:

用户调用SSP配置，并走到用户的ncs_map的实现以处理映射。

功能描述:

用户调用SSP配置，并走到用户的ncs_map的实现以处理映射。

接口约束:

如果一个用户异常被抛出，那需要通过使用固定的用户异常类。

说明

此函数具有默认实现。如果映射逻辑中的变量不需要额外处理，可以直接使用默认实现。

- **参数**
  - **request** (*obj*) - 请求参数
  - **aoc_context** - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
  - **template** (*str*) - 模板路径
- **返回**
  - 业务映射配置结果，xml格式
- **返回类型**
  - str

实际案例

1. 用户直接将请求数据传递到模板渲染中

```
def ncs_map(self, request, aoc_context=None, template=''):
    result = self.render(template, request.xmldict)
    return result
```

2. 用户需要代码来处理请求，并构造模板参数进行渲染

```
def ncs_map(self, request, aoccontext=None, template=None):
    # get isp service input data
    request_dic_node = self.xmltodictnode(request.xml)
    self.logger.info("bngAaaPairs request_dic_node: %s" % (request_dic_node))
    template_context = self.fill_miniaaapair_context(request_dic_node, aoccontext)
    # query bng data from cdb in isp service
    self.logger.info("bngAaaPairs template_syspair_context: %s" % (template_context))
    # render the j2 template with template_context
```

```
    strt = self.render('template_Aaa.j2', template_context)
    self.logger.info("bngAaaPairs template_nepair%s" % (strt))
    return strt
#fill context
def fill_miniaaapair_context(self, request_dic_node, aoccontext):
    self.aaamini = request_dic_node.aaaminignd
    servicepath = aoccontext.serviceInst
    nerootpath = aoccontext.serviceInst[servicepath.find("ne/")+3:]
    neid = nerootpath[:nerootpath.find("/")]
    return {'users':self.aaamini,'neid':neid}
```

ncs_rpc(rpc_input**,** aoc_context**,** template=None)

系统提供给用户的ncs_rpc接口。

典型场景:

系统提供给用户的ncs_rpc接口。

功能描述:

系统提供给用户的ncs_rpc接口。

接口约束:

rpc_input入参必须合法。

- **参数**
  - **rpc_input** (*obj*) - rpc输入参数
  - **aoc_context** (*obj*) - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
  - **template** (*str*) - 模板名称
- **返回**
  - 转换后的aoc ncs rpc结果.
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.ncs.aocncsrpc.AocNcsRpcOutput

实际案例

这是一个ncs_rpc函数示例

```
rpc_input.inputData = '<input><userName>xxx</userName></input>'
rpc_input.rpcName = '(nrn:com:huawei)activeUser'
def ncs_rpc(self, rpc_input, aoc_context, template=None):
    sndinputData = convert(rpc_input.inputData)
    neid = getneid(sndinputData)
    sndrpcName = convert(rpc_input.rpcName)
    sndresult = devicemgr.device_rpc(neid, sndinputData, sndrpcName)
    sspresult = convert(sndresult.result)
    return sspresult
sspresult = ncs_rpc(rpc_input)
print(sspresult)
<output><result>OK</result></output>
```

ncs_validate(aoc_context**,** validate_input)

业务恢复接口，由第三方用户实现。

典型场景:

第三方用户需要验证其输入。

功能描述:

验证实现接口，由第三方用户复制。在这里，打开了一个钩子供用户处理，例如资源冲突验证。

接口约束:

无。

- **参数**
  - **aoc_context** (*obj*) - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
  - **validate_input** (*obj*) - 校验入参

```
post_modify(aoc_context, post_modify_request)
```

更新或删除服务后，此方法将在同一事务中调用。

典型场景:

在某些情况下，此业务正在更新或删除，用户也需要更新或删除相关服务实例，用户可以实现此方法。

功能描述:

更新或删除业务后，此方法将在同一事务中调用。

接口约束:

对于某个业务，此方法只在系统中实现一次。

- **参数**
  - **aoc_context** - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
  - **post_modify_request** - 自定义类postModify请求输入， 请参见：类：**PostModifyRequest**。
- **返回**
  - PostModify结果.

实际案例

这里有一个例子来说明Post-Modift函数的输入和输出参数。

```
In this example, eva depends on evs
evs Yang definition:
augment "/app:applications"{
   list evs {
      app:application-definition "evs";
      key "evs-id";
      leaf evs-id {
         type string {
            length "1..24";
         }
         description "evs-id";
      }
   }
}
eva Yang definition:
```

```
augment "/app:applications" {
    list eva {
        app:application-definition "eva";
        key "eva-id";
        leaf eva-id {
            type string {
                length "1..24";
            }
            description "eva-id";
        }
        leaf eas-id {
            type string {
                length "1..24";
            }
            description "eas-id";
        }
        leaf evs-id {
            type string {
                length "1..24";
            }
            description "evs-id";
        }
    }
}
register process:
user need to modify package'pkg.json file(Example in evs),
add below hook information
"hooks": [
    {
        "type": "post-modify-action",
        "key": "evs",
        "python-class-name": "kpn.evs.AocNcsEvsServicePoint"
    }
]
def post_modify(self, aoc_context, post_modify_request):
    #  Get device information from post_modify
    for data in post_modify_request.change_data:
        path = data.path
        instance_name = path.split('/')[-1]
        attached_path = "/huawei-ac-applications:applications/huawei-ncs-eva:eva/" +
instance_name
        if data.operation_type == 1:
            service.redeploy(attached_path)
```

render(template_name**,** *args**,** **kwargs)

系统提供的用户模板渲染接口。

典型场景:

用户在代码中调用渲染接口进行模板渲染的函数。

功能描述:

用于模板渲染的函数接口。

接口约束:

模板的参数内容必须与模板变量匹配。模板中只能使用机制提供的过滤器和用户SSP包定义的过滤器。

- **参数**

  - **template_name** (*str*) - 模板的名称
  - **args** (*list, optional*) - 位置参数，模板中使用的变量，必须是字典
  - **kwargs** (*dict, optional*) - 关键字参数，模板中使用的变量
- **返回**
  - 模板渲染后的配置内容
- **返回类型**
  - str

实际案例

```
temppara = {
   'taskName': '111',
   'wws': [{"xx":1,"yy":"xxxxxx"},
      {"xx":7}],
   'isis': 'true'
}
>>> result = render("aaa.j2", temppara, taskName='222')
>>> print(result)
<aaa xmlns="http://www.huawei.com/netconf/vrp/huawei-aaa">
   <tasks>
      <task>
         <taskName>222</taskName>
      </task>
   </tasks>
   <taskGroups>
      <taskGroup>
         <taskGroupName>1</taskGroupName>
         <taskGroupName1>xxxxxx</taskGroupName1>
      </taskGroup>
      <taskGroup>
         <taskGroupName>7</taskGroupName>
         <taskGroupName1></taskGroupName1>
      </taskGroup>
   </taskGroups>
</aaa>
```

set_global(key, val)

系统提供接口供用户设置全局参数

典型场景:

系统提供接口供用户编码和设置全局参数。

功能描述:

系统提供接口供用户设置全局参数。

接口约束:

无。

- **参数**
  - **key** (*str*) - key值
  - **val** (*str*) - val值
- **返回**
  - 业务映射配置结果，xml格式

- **返回类型**
  - str

实际案例

template1:

```
<inventory-cfg xmlns="urn:huawei:yang:huawei-ac-nes">
{{SET_GLOBAL('xxx','111')}}
   <nes>
{%- for user in users %}
      <ne>
         <neid>{{ user.nename | to_ne_id }}</neid>
         {%- include 'com/huawei/aaa/hbng-aaa.j2' %}
      </ne>
{%- endfor %}
   </nes>
</inventory-cfg>
```

template2:

```
<aaa xmlns="http://www.huawei.com/netconf/vrp/huawei-aaa">
   <tasks>
      <task>
         <taskName>{{GET_GLOBAL('xxx')}}</taskName>
      </task>
   </tasks>
   {% if isis == 'true' %}
      <taskGroups>{% for ww in wws %}
         <taskGroup>
            <taskGroupName>{{ww.xx}}</taskGroupName>
            <taskGroupName1>{{ww.yy}}</taskGroupName1>
         </taskGroup>{% endfor %}
      </taskGroups>
   {% endif %}
</aaa>
```

***classmethod* to_manage_ip(ne_name)**

系统为用户提供jinja模板，根据ne_name查询管理ip的功能。

典型场景:

系统为用户提供jinja模板，根据ne_name查询管理ip的功能。

功能描述:

系统为用户提供jinja模板，根据ne_name查询管理ip的功能。

接口约束:

neName入参必须合法。

- **参数**
  - **ne_name** (*str*) - 网元名称
- **返回**
  - 管理IP
- **返回类型**
  - str

实际案例

jinja example:

```
<inventory-cfg xmlns="urn:huawei:yang:huawei-ac-nes">
   <nes>
      {%- for ne in nes %}
         <ne>
            <neip>{{ne.deviceName | to_manage_ip}}</neip>
            {{ne.nexml}}
         </ne>
      {%- endfor %}
   </nes>
</inventory-cfg>
```

***classmethod* to_ne_id**(ne_name)

系统为用户提供jinja模板，根据ne_name查询ne_id的函数。

典型场景:

该系统提供给用户使用jinja模板，根据ne_name获取ne_id的功能。

功能描述:

该系统提供给用户使用jinja模板，根据ne_name获取ne_id的功能。

接口约束:

neName必须合法。

- **参数**
  - **ne_name** (*str*) - 网元名称。
- **返回**
  - 网元ID。
- **返回类型**
  - str

实际案例

下面是在jinja模板中使用函数的示例:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
   ~ Copyright (c) Huawei Technologies Co., Ltd. 2020-2021. All rights reserved.
   -->
   <inventory-cfg xmlns="urn:huawei:yang:huawei-ac-nes">
   <nes>
   {%- for ne in nes %}
      <ne>
         <neid>{{ne.deviceName | to_ne_id}}</neid>
         {{ne.xml}}
      </ne>
   {%- end for %}
   </nes>
</inventory-cfg>
```

static* xmltodict**(*xml_input*, *encoding=None*, *expati=<module 'xml.parsers.expat' from 'c:||users||***|*
|appdata||local||programs||python||python37||lib||xml||parsers||expat.py'>,
process_namespaces=False, *namespace_separator=':'*, *disable_entities=True*, ***kwargs*)

系统提供用户将xml转换为dict。

典型场景:

在用户编码过程中，使用xml转换为dict。

功能描述:

在用户编码过程中，使用xml转换为dict。

接口约束:

xml必须为标准格式。

- **参数**
  - **xml_input** (*str*) - 输入xml数据
  - **encoding** (*obj, optional*) - 编码内容
  - **expati** (*obj, optional*) - 通过替代版本
  - **process_namespaces** (*obj, optional*) - 进程命名空间
  - **namespace_separator** (*obj, optional*) - 分离字符串
  - **disable_entities** (*obj, optional*) - 禁用实体
  - **kwargs** (*obj, optional*) - 字典格式输入参数
- **返回**
  - 字典内容
- **返回类型**
  - dict

实际案例

xml_input 可以是 string 也可以是类似文件的对象.

如果xml_attribs为True，则元素属性将放入字典中 在常规子元素中，使用@作为前缀以避免冲突。如果 设置为False，它们只是被忽略。简单的例子

```
>>> doc = xmltodict("""
... <a prop="x">
...   <b>1</b>
...   <b>2</b>
... </a>
... """)
>>> doc['a']['@prop']
u'x'
>>> doc['a']['b']
[u'1', u'2']
```

If item_depth is 0, the function returns a dictionary for the root element (default behavior). Otherwise, it calls item_callback every time an item at the specified depth is found and returns None in the end (streaming mode).

The callback function receives two parameters: the path from the document root to the item (name-attribs pairs), and the item (dict). If the callback’s return value is false-ish, parsing will be stopped with the ParsingInterrupted exception.Streaming example

```
>>> def handle(path, item):
...     print('path:%s item:%s' % (path, item))
...     return True
```



```
...
>>> xmltodict("""
... <a prop="x">
...   <b>1</b>
...   <b>2</b>
... </a>""", item_depth=2, item_callback=handle)
path:[(u'a', {u'prop': u'x'}), (u'b', None)] item:1
path:[(u'a', {u'prop': u'x'}), (u'b', None)] item:2
```

The optional argument postprocessor is a function that takes path, key and value as positional arguments and returns a new (key, value) pair where both key and value may have changed. Usage example

```
>>> def postprocessor(path, key, value):
...     try:
...         return key + ':int', int(value)
...     except (ValueError, TypeError):
...         return key, value
>>> xmltodict('<a><b>1</b><b>2</b><b>x</b></a>',
...                postprocessor=postprocessor)
OrderedDict([(u'a', OrderedDict([(u'b:int', [1, 2]), (u'b', u'x')]))])
```

You can pass an alternate version of expat (such as defusedexpat) by using the expat parameter. E.g:

```
>>> xmltodict('<a>hello</a>', expat=defusedexpat.pyexpat)
OrderedDict([(u'a', u'hello')])
```

You can use the force_list argument to force lists to be created even when there is only a single child of a given level of hierarchy. The force_list argument is a tuple of keys. If the key for a given level of hierarchy is in the force_list argument, that level of hierarchy will have a list as a child (even if there is only one sub-element). The index_keys operation takes precendence over this. This is applied after any user-supplied postprocessor has already run.For example, given this input:

```
<servers>
   <server>
      <name>host1</name>
      <os>Linux</os>
      <interfaces>
         <interface>
            <name>em0</name>
            <ip_address>10.0.0.1</ip_address>
         </interface>
      </interfaces>
   </server>
</servers>
```

If called with force_list=('interface',), it will produce this dictionary:

```
{'servers':
   {'server':
      {'name': 'host1',
      'os': 'Linux'},
      'interfaces':
         {'interface':
            [ {'name': 'em0', 'ip_address': '10.0.0.1' } ] } } }
```

force_list can also be a callable that receives path, key and value. This is helpful in cases where the logic that decides whether a list should be forced is more complex.

static* xmltodictnode(*xml_input*, *encoding=None*, *expati=<module 'xml.parsers.expat' from 'c:||users||***||appdata||local||programs||python||python37||lib||xml||parsers||expat.py'>*, *process_namespaces=False*, *namespace_separator=':'*, *disable_entities=True*, ***kwargs*)**

系统提供用户将xml转换为dictNode。

典型场景:

在用户编码过程中，使用xml转换为dictNode。

功能描述:

在用户编码过程中，使用xml转换为dictNode。

接口约束:

xml必须为标准格式。

注解

不要在Args部分中包括self参数。

- **参数**
  - **xml_input** (*str*) - 输入xml数据
  - **encoding** (*obj, optional*) - 编码内容
  - **expati** (*obj, optional*) - 通过替代版本
  - **process_namespaces** (*obj, optional*) - 进程命名空间
  - **namespace_separator** (*obj, optional*) - 分离字符串
  - **disable_entities** (*obj, optional*) - 禁用实体
  - **kwargs** (*obj, optional*) - 字典格式输入参数
- **返回**
  - dictNode内容
- **返回类型**
  - dict

实际案例

xml_input 可以是 string 也可以是类似文件的对象.

如果xml_attribs为True，则元素属性将放入字典中 在常规子元素中，使用@作为前缀以避免冲突。如果 设置为False，它们只是被忽略。简单的例子

```
>>> doc = xmltodictnode("""
... <a prop="x">
...   <b>1</b>
...   <b>2</b>
... </a>
... """)
>>> doc['a']['@prop']
u'x'
>>> doc['a']['b']
[u'1', u'2']
```

If item_depth is 0, the function returns a dictionary for the root element (default behavior). Otherwise, it calls item_callback every time an item at the specified depth is found and returns None in the end (streaming mode).

The callback function receives two parameters: the path from the document root to the item (name-attribs pairs), and the item (dict). If the callback’s return value is false-ish, parsing will be stopped with the ParsingInterrupted exception.Streaming example

```
>>> def handle(path, item):
...     print('path:%s item:%s' % (path, item))
...     return True
...
>>> xmltodictnode("""
... <a prop="x">
...   <b>1</b>
...   <b>2</b>
... </a>""", item_depth=2, item_callback=handle)
path:[(u'a', {u'prop': u'x'}), (u'b', None)] item:1
path:[(u'a', {u'prop': u'x'}), (u'b', None)] item:2
```

The optional argument postprocessor is a function that takes path, key and value as positional arguments and returns a new (key, value) pair where both key and value may have changed. Usage example

```
>>> def postprocessor(path, key, value):
...     try:
...         return key + ':int', int(value)
...     except (ValueError, TypeError):
...         return key, value
>>> xmltodictnode('<a><b>1</b><b>2</b><b>x</b></a>',
...                postprocessor=postprocessor)
OrderedDict([(u'a', OrderedDict([(u'b:int', [1, 2]), (u'b', u'x')]))])
```

You can pass an alternate version of expat (such as defusedexpat) by using the expat parameter. E.g:

```
>>> xmltodictnode('<a>hello</a>', expat=defusedexpat.pyexpat)
OrderedDict([(u'a', u'hello')])
```

You can use the force_list argument to force lists to be created even when there is only a single child of a given level of hierarchy. The force_list argument is a tuple of keys. If the key for a given level of hierarchy is in the force_list argument, that level of hierarchy will have a list as a child (even if there is only one sub-element). The index_keys operation takes precendence over this. This is applied after any user-supplied postprocessor has already run.For example, given this input:

```
<servers>
   <server>
      <name>host1</name>
      <os>Linux</os>
      <interfaces>
         <interface>
            <name>em0</name>
            <ip_address>10.0.0.1</ip_address>
         </interface>
      </interfaces>
   </server>
</servers>
```

If called with force_list=('interface',), it will produce this dictionary:

```
{'servers':
   {'server':
      {'name': 'host1',
      'os': 'Linux'},
```



```
'interfaces':
{'interface':
   [ {'name': 'em0', 'ip_address': '10.0.0.1' } ] } } }
```

force_list can also be a callable that receives path, key and value. This is helpful in cases where the logic that decides whether a list should be forced is more complex.

class aoc.ncs.ncsservice.PostModifyRequest(properties, change_data)

基类：object

post modify请求参数。

典型场景:

该类用于将原型定义的postModifyInput转换为请求Input，作为post_modify方法的入参。

功能描述:

该类用于将原型定义的postModifyInput转换为请求Input，作为post_modify方法的入参。

接口约束:

无。

- **参数**
  - **properties** (*dict*) - PostModify请求属性.
  - **change_data** (*list*) - 自定义数据结构，包括修改后的服务实例数据。

service_path

Post modify 业务实例路径

Type：str

properties

PostModify 请求属性.

Type：dict

change_data

自定义数据结构，包括修改后的服务实例数据。

Type：list

get_change_data()

在PostModify操作中获取更改的数据。

典型场景:

在PostModify操作中获取更改的数据。

功能描述:

在PostModify操作中获取更改的数据。

接口约束:

无。

- **返回**
  - 自定义数据结构，包括修改后的服务实例数据。

实际案例

这里有一个例子来说明Post-Modify获取变更数据函数的输入和输出参数。

```
>>> ChangeData =  PostModifyRequest.get_change_data
>>> print ChangeData[0].path
'/huawei-ac-applications:applications/huawei-ncs-eas:eas/eas_1'
```

get_path()

获取PostModify业务路径。

典型场景:

获取PostModify业务路径。

功能描述:

获取PostModify业务路径。

接口约束:

无。

- **返回**
  - Post modify 业务实例路径

实际案例

这里有一个例子来说明Post-Modify获取业务路径函数的输入和输出参数。

```
>>> print PostModifyRequest.get_path
'/huawei-ac-applications:applications/huawei-ncs-eas:eas/eas_1'
```

get_properties()

获取PostModify属性。

典型场景:

获取PostModify属性。

功能描述:

获取PostModify属性。

接口约束:

无。

- **返回**
  - PostModify请求属性.

实际案例

这里有一个例子来说明Post-Modify获取属性函数的输入和输出参数。


```
>>> postModify_Property= PostModifyRequest.get_properties
>>>print postModify_Property['name']
'property'
```

class aoc.ncs.ncsservice.RawTypeStr(string)

基类： str

str is put to the dictnode and special characters are not escaped

aoc.ncs.ncsservice.to_manage_ip(ne_name)

根据操作名称查询设备IP。

典型场景:

编译SSP包代码时，需要根据设备操作名称查询设备IP。

功能描述:

to_manage_ip接口用于根据设备管理操作名称查询设备IP。处理如下： 根据入参操作名称查询设备IP, to_manage_ip接口支持缓存加速，提高查询性能。

接口约束:

无。

- **参数**
  - **ne_name** (*str*) - 设备名称
- **返回**
  - 管理IP
- **返回类型**
  - str
- **引发**
  - **AOCException** - 设备不存在

实际案例

```
>>> from aoc.sys import NcsService
>>> nename = 'HUAWEI_ROUTER'
>>> neip = to_ne_id(nename)
>>> print(neip)
192.168.1.1
```

aoc.ncs.ncsservice.to_ne_id(ne_name)

根据操作名称查询设备ID。

典型场景:

编译SSP包代码时，需要根据设备操作名称去获取设备id。

功能描述:

该接口用于根据设备操作名称获取设备ID。流程如下： 根据入参operateName获取设备ID。 该接口支持缓存加速以提升性能。

- **参数**

- **ne_name** (*str*) - 设备操作名称。

- **返回**
  - 设备 id.
- **返回类型**
  - str
- **引发**
  - **InvalidArguments** - 输入参数为空或输入参数无效。
  - **InnerException** - 设备不存在。

实际案例

```
>>> # Invoke the interface to obtain the device ID corresponding to response result.
>>> print to_ne_id('HUAWEI_ROUTER')
868e778e-153c-3afe-a02c-89678e31e3e4
```

8.1.4 aoc.ncs.validateexception module

典型场景:

当用户调用SSP或GND配置并步行到用户的ncs_map实现以处理映射时， 此模块提供插件包自定义验证失败时的异常。

功能描述:

验证插件包的验证异常失败。

exception aoc.ncs.validateexception.ValidateException(*paras)

基类：aoc.exception.aocexceptions.AOCException

典型场景:

当用户调用SSP或GND配置并步行到用户的ncs_map实现以处理映射时，此模块提供插件包自定义验证失败时的异常。**功能描述:**

验证插件包的验证异常失败。

接口约束:

无。

9 aoc.snd

9.1 aoc.snd package

9.1.1 aoc.snd.clicapability module

典型场景:

当aoc基于cli协议下发配置时，设备或aoc无法识别默认转换规则传递的命令或yang数据。 基于上述场景，该模块允许用户更改默认的AOC转换规则。

功能描述:

该模块允许用户更改默认的aoc转换规则。提供的方法如下：yangToCli,cliToYang, configCliPostProcessor, readCliPreProcessor,configRollbackCliPostProcessor, configCliByLinePostProcessor, getPassthroughCommands.

class aoc.snd.clicapability.CliCapability(logger, input_types, output_types)

基类：aoc.snd.clicommoncapability.CliCommonCapability

典型场景:

当aoc基于cli协议下发配置时，设备或aoc无法识别默认转换规则传递的命令或yang数据。 基于上述场景，该模块允许用户更改默认的AOC转换规则。

功能描述:

该模块允许用户更改默认的aoc转换规则。提供的方法如下：yangToCli,cliToYang, configCliPostProcessor, readCliPreProcessor,configRollbackCliPostProcessor, configCliByLinePostProcessor, getPassthroughCommands.

接口约束:

无。

***abstract* cliToYang**(aoccontext, request)

提供自定义处理接口，用于将cli转换为yang。

典型场景:

1. aoc读取设备数据，并将cli数据包根据默认转换规则转换为yang模型数据。
2. 在某些特殊场景下，命令无法根据yang模型结构的默认规则转换为yang数据， 用户可以自定义转换实现逻辑。

功能描述:

提供自定义处理接口，用于将cli转换为yang。

接口约束:

注解cli-custom-read需要在yang文件中配置。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.cliTransform.CliCustomTransformInput
- **返回**
  - 转换后的Cli命令信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.cliTransform.CliCustomTransformOutput

实际案例

描述样例:

```
module cli-custom {
    container c55 {
        cli:cli-custom-read 'two-step';
        leaf l55 {
            type string;
        }
    }
}
def cliToYang(self, aoccontext, request):
    out = CliCustomTransformOutput()
    if request.path.startswith("/cli-custom:c55"):
        out.data = "<c55 xmlns="http://huawei.com/cli-custom"><l55>hello_world</l55></c55>"
        out.endOffset = str(request.data).__len__()
    return out
request = {}
request.setdefault("path", "/cli-custom:c55:l55")
request.setdefault("data", "hello_world")
request.setdefault("oper_type", "CREATE")
request.setdefault("neId", "123")
>>> print cliToYang (aoccontext, request)
{"data":"<c55 xmlns="http://huawei.com/cli-custom"><l55>hello_world</l55></c55>",
"endOffset":128}
```

***abstract* configCliByLinePostProcessor**(aoccontext, request)

在后处理的基础上，不支持事务的设备可以回退。

典型场景:

1. 设备不支持自动回退，当aoc下发多个配置给设备时，其他配置因其中一个配置失败而回退， 默认的aoc转换规则不符合用户要求。

2. 配置成功下发到设备后，由于其他设备故障，下发成功的设备的配置会回退。
3. aoc根据回退规则生成报文，如果生成的报文不符合业务预期，可以通过后处理修改回滚命令。

功能描述:

在后处理的基础上，不支持事务的设备可以回退。

接口约束:

注解cli-custom-processor的’post-config-by-line’属性需要在yang文件中配置。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.cliTransform.CliCustomTransformInput。
- **返回**
  - 转换后的Cli命令信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.cliTransform.CliCustomTransformOutput

实际案例

描述样例:

```
cli-custom.yang:
    module cli-custom {
        container c1 {
            cli:cli-custom-processor 'post-config-by-line';
            leaf l8 {
                type string;
            }
        }
    }
def configCliByLinePostProcessor(self, aoccontext, request):
    out = CliCustomTransformOutput()
    req_path = request.path
    if req_path == '/cli-custom:c1':
        for item in request.commandLines:
            command = out.commandLines.add()
            command.index = item.index
            if 'interface a' in item.commandLine:
                command.commandLine = item.commandLine.replace('interface a','interface b')
    return out
request = {}
request.setdefault("path", "/cli-custom:c1")
request.setdefault("commandLines", [{"index":0, "commandLine": "no interface a"}])
>>> print configCliByLinePostProcessor (aoccontext, request)
[{"index":0, "commandLine": "no interface b"}]
```

***abstract* configCliPostProcessor**(aoccontext**,** request)

aoc将yang模型数据转换为命令后，用户可以修改命令。

典型场景:

aoc配置下发时，aoc根据默认的yang模型转换规则，将yang模型数据转换为cli报文。

但是，在某些特殊场景下，aoc生成的cli命令与设备上的实际命令不同， 您可以自定义转换后的命令。

功能描述:

aoc将YANG模型数据转换为命令后，用户可以修改命令。

接口约束:

注解cli-custom-processor的’post-config’属性需要在yang文件中配置。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.cliTransform.CliCustomTransformInput。
- **返回**
  - 转换后的Cli命令信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.cliTransform.CliCustomTransformOutput

实际案例

描述样例:

```
cli-custom.yang:
    module cli-custom {
        container c1 {
            cli:cli-custom-processor 'post-config';
            leaf l8 {
                type string;
            }
        }
    }
def configCliPostProcessor(self, aoccontext, request):
    out = CliCustomTransformOutput()
    cli_str = request.data
    if request.path == "/cli-custom:c1":
        cli_str = cli_str + "with c1"
    elif request.path == "/cli-custom:c1/cli-custom:c2":
        cli_str = cli_str + "with c2"
    elif request.path == "/cli-custom:c1/cli-custom:c2/cli-custom:c3":
        cli_str = cli_str + "with c3"
    out.data = cli_str
    return out
request = {}
request.setdefault("path", "/cli-custom:c1/c1:l8")
request.setdefault("data", "hello_word")
request.setdefault("oper_type", "CREATE")
request.setdefault("neId", "123")
>>> print configCliPostProcessor (aoccontext, request)
{"data":"<c1 xmlns="http://huawei.com/cli-custom"><l8>hello_world with c1</l8></c1>"}
```

```
abstract configRollbackCliPostProcessor(aoccontext, request)
```

aoc生成回滚报文后，修改后续报文。

典型场景:

1. 配置成功下发到设备后，由于其他设备故障，下发成功的设备的配置会回退。
2. Aoc根据回滚规则生成报文，如果生成的报文不符合业务预期， 可以通过后处理修改回滚命令。

功能描述:

aoc根据默认规则生成回滚报文后，用户可以修改后续报文。

接口约束:

1. 如果场景1不支持回退能力，则控制器状态标记为不一致。
2. 当AOC下发多个配置给设备时，设备不支持自动回退。
3. 其他配置因其中一个配置失败而回退。
4. 默认的aoc转换规则不符合用户要求。

说明

cli-custom-processor的’post-config-rollback’属性需要在yang文件中配置。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.cliTransform.CliCustomTransformInput。
- **返回**
  - 转换后的Cli命令信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.cliTransform.CliCustomTransformOutput

实际案例

描述样例:

```
cli-custom.yang:
    module cli-custom {
        container c1 {
            cli:cli-custom-processor 'pre-read';
            leaf l8 {
                type string;
            }
        }
    }
def configRollbackCliPostProcessor(self, aoccontext, request):
    out = CliCustomTransformOutput()
    if request.path.startswith('/cli-custom:c1'):
        out.data = 'undo cli-custom c1'
    return out
request = {}
request.setdefault("path", "/cli-custom:c1/c1:l8")
request.setdefault("data", "c1 leaf1 value1")
request.setdefault("oper_type", "CREATE")
request.setdefault("neId", "123")
>>> print configRollbackCliPostProcessor (self, aoccontext, request)
{"data":"undo cli-custom c1"}
```

***abstract* getPassthroughCommands**(aoccontext)

为了防止用户在设备上操作任何命令，您可以指定将命令透传下发到设备的条件。

典型场景:

1. aoc提供cli接口供北向用户调用。
2. 但是，需要定制接口的权限，并设置设备命令的访问范围。

功能描述:

为了防止用户在设备上操作任何命令，您可以指定将命令透传下发到设备的条件。

接口约束:

1. snd包的资源目录中包含CliPasstoCommands.xml配置文件。
2. xml配置文件中配置的命令为正则表达式。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
- **返回**
  - 命令信任列表。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.cliPassthrough.CliPassthroughOutput

实际案例

描述样例:

```
Obtains the command trustlist configured in the snd package CliPassthroughCommands.xml.
def getPassthroughCommands(self, aoccontext):
    output = CliPassthroughOutput()
    file_path = os.path.join(self.resourceDir, "resources/CliPassthroughCommands.xml")
    tree = ET.parse(file_path)
    root = tree.getroot()
    for child in root:
        value = codecs.getdecoder("unicode_escape")(child.text)[0]
        commandEntity = output.add()
        commandEntity.command = value
    return output
>>> print configCliByLinePostProcessor (aoccontext)
["command":"interface.*"]
```

***abstract* readCliPreProcessor**(aoccontext, request)

在aoc将命令行转换为yang数据之前，用户可以修改命令行。

典型场景:

1. aoc读取设备数据，并根据默认的转换规则将cli报文转换为yang模型数据。
2. 但在某些特殊场景下，命令无法转换为yang数据，自定义转换实现逻辑。

功能描述:

在aoc将命令行转换为yang数据之前，用户可以修改命令行。

说明

cli-custom-processor的’pre-read’属性需要在yang文件中配置。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.cliTransform.CliCustomTransformInput。
- **返回**
  - 转换后的Cli命令信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.cliTransform.CliCustomTransformOutput

实际案例

描述样例:

```
cli-custom.yang:
    module cli-custom {
        container c1 {
            cli:cli-custom-processor 'pre-read';
            leaf l8 {
                type string;
            }
        }
    }
def readCliPreProcessor(self, aoccontext, request):
    out = CliCustomTransformOutput()
    cli_str = str(request.data)
    if request.path.startswith("/cli-custom:c1"):
        cli_str = cli_str + "\npreread test\n"
    out.endOffset = -1
    out.data = cli_str
    self.logger.info("readCliPreProcessor output data: %s" % str(out.data))
    return out
request = {}
request.setdefault("path", "/cli-custom:c1/c1:l8")
request.setdefault("data", "hello_word")
request.setdefault("oper_type", "CREATE")
request.setdefault("neId", "123")
>>> print readCliPreProcessor (aoccontext, request)
{"data":"hello_word \npreread test\n","endOffset":-1}
```

***abstract* yangToCli**(aoccontext, request)

提供自定义处理接口，用于将yang转换为cli。

典型场景:

aoc配置下发时，aoc将YANG模型数据转换为cli基于默认的yang模型转换规则的数据包。

但是，在一些特殊场景下，aoc生成的cli命令是与设备上的实际命令不同， 您可以自定义从yang向cli的转变。

功能描述::

提供自定义处理接口，用于将yang转换为cli。

接口约束:

注解cli-custom-transform需要在yang文件中配置。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.cliTransform.cliCustomTransformInput。
- **返回**
  - 转换后的Cli命令信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.cliTransform.CliCustomTransformOutput

实际案例

描述样例：

```
cli-custom.yang:
    module cli-custom {
        container c100 {
            cli:cli-custom-transform 'create,update,delete'
            leaf l100 {
                type string;
            }
        }
    }
def yangToCli(self, aoccontext, request):
    if request.path.startswith("/cli-custom:c100") and request.oper_type == 'DELETE':
    return yangtocli_c100(self.logger, aoccontext, request)
def yangtocli_c100(logger, aoccontext, request):
    out = CliCustomTransformOutput()
    out.data =
        {
            OperType.Value('CREATE'): processCreate,
            OperType.Value('MERGE'): processMerge,
            OperType.Value('DELETE'): processDelete
        }.get(request.oper_type, "")(logger, aoccontext, request)
    return out
def processCreate(logger, aoccontext, request):
    return "c100 l100 value_override_create"
def processMerge(logger, aoccontext, request):
    root = ET.fromstring(request.data)
    new_data = root[0].text
    logger.info('new_data %s' % new_data)
    rdb_document = read_rdb(aoccontext, request)
    old_root = ET.fromstring(rdb_document)
    old_data = old_root[0].text
    logger.info('old_data %s' % old_data)
    return "c100 l100" + old_data + "_" + new_data
def processDelete(logger, aoccontext, request):
    return "c100 l100 value_override_delete"
request = {}
request.setdefault("path", "/cli-custom:c100")
request.setdefault("oper_type", "DELETE")
request.setdefault("neId", "123")
>>> print yangToCli (aoccontext, request)
{"data":"c100 l100 value_override_delete"}
```

9.1.2 aoc.snd.clicommoncapability module

典型场景:

当aoc根据netconf下发配置时，需要设置设备连接建立参数和配置下发参数。

功能描述:

本模块提供了使用netconf建立关联的方法和设置参数的方法。

class aoc.snd.clicommoncapability.CliCommonCapability(logger, input_types, output_types)

基类：aoc.base.baseservice.BaseService

典型场景:

当aoc根据netconf下发配置时，需要设置设备连接建立参数和配置下发参数。

功能描述:

本模块提供了使用netconf建立关联的方法和设置参数的方法。

接口约束:

无。

***abstract* getCliDriverInfo**(aoccontext, request=None)

根据驱动程序信息获取参数和数据包交换信息。

典型场景:

当设备建立连接并交换报文时，需要设置参数。例如:
userModePrompt,privilegedModePrompt,userNameResponse,passwordResponse,
keepAliveCommand,errorMessagePattern,ignoreMessagePattern,promptSkipRegex,
judgeContinueCondition,continueReadingCommand,negateCommand,exitSubmode
Command,
configModeCommand,partialReadCommand,defaultDeleteOperationWithValue,
contextSupportsSinglelineCommand,supportsSinglelineCommand,partialReadSubm
ode.

功能描述:

1. 定义与设备关联建立和报文交互相关的参数。
2. 您可以在配置文件中设置驱动程序信息。

接口约束:

某些驱动程序信息是必填的，如果没有配置驱动信息，包可能激活失败。 必填字段为:userModePrompt,privilegedModePrompt,userNameResponse,
passwordResponse,keepAliveCommand,errorMessagePattern,
ignoreMessagePattern,promptSkipRegex,judgeContinueCondition,
continueReadingCommand,negateCommand,exitSubmodeCommand,configModeC
ommand,
partialReadCommand,defaultDeleteOperationWithValue,contextSupportsSingleline
Command, supportsSinglelineCommand,partialReadSubmode.

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*dict, optional*) - 自定义参数的扩展使用。
- **返回**

- 转换后的Cli命令信息。

- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.cliDriver.CliDriverInfo

实际案例

描述样例:

```
def getCliDriverInfo(self, aoccontext, request=None):
    cliDriverInfo = CliDriverInfo()
    cliDriverEntity = cliDriverInfo.cliDriverEntity.add()
    cliDriverEntity.key = "protocol"
    cliDriverEntity.value = "Ssh"
    cliDriverEntity2 = cliDriverInfo.cliDriverEntity.add()
    cliDriverEntity2.key = "userModePrompt"
    cliDriverEntity2.value = ".*>.*"
    return cliDriverInfo
>>> print getCliDriverInfo (aoccontext, None)
[{"protocol":"Ssh"},{"userModePrompt":".*>.*"}]
```

9.1.3 aoc.snd.clisnd module

class aoc.snd.clisnd.CliSND(logger=None, resourceDir='')

基类：aoc.snd.snd.SND, aoc.snd.clicapability.CliCapability

cli默认实现

cliToYang(aoccontext, request)

引用

具体操作请参见 **aoc.snd.clicapability.CliCapability.cliToYang()**。

configCliByLinePostProcessor(aoccontext, request)

引用

具体操作请参见 **aoc.snd.clicapability.CliCapability.configCliByLinePostProcessor()**。

configCliPostProcessor(aoccontext, request)

引用

具体操作请参见 **aoc.snd.clicapability.CliCapability.configCliPostProcessor()**。

configRollbackCliPostProcessor(aoccontext, request)

引用

具体操作请参见 **aoc.snd.clicapability.CliCapability.configRollbackCliPostProcessor()**。

getCliDriverInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.clicommoncapability.CliCommonCapability**

getConnectInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.commonsnd.CommonSND.getConnectInfo()**。

getDiscoverInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.commonsnd.CommonSND.getDiscoverInfo()**。

getPassthroughCommands(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.clicapability.CliCapability.getPassthroughCommands()**。

getPrimaryProtocolInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.snd.SND.getPrimaryProtocolInfo()**。

getProtocolInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.commonsnd.CommonSND.getProtocolInfo()**。

readCliPreProcessor(aoccontext, request)

引用

具体操作请参见 **aoc.snd.clicapability.CliCapability.readCliPreProcessor()**。

yangToCli(aoccontext, request)

引用

具体操作请参见 **aoc.snd.clicapability.CliCapability.yangToCli()**。

9.1.4 aoc.snd.commonsnd module

class aoc.snd.commonsnd.CommonSND(logger=None, resourceDir='')

基类：aoc.base.baseservice.BaseService

abstract **callDiffJson**(aoccontext, request)

定制数据一致性差异显示。

典型场景:

发现数据一致性差异后，您可以修改生成的数据，并在页面上显示修改后的数据。 例如，在别名替换中，静态rt可以替换huawei-staticrt。

功能描述:

定制不一致的数据在Web UI上显示。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。

  - **request** (*obj*) - 要下发的差分数据。 定义在proto文件中，路径为proto.aoc.snd.DiffDataIn.DiffDataInMsg。

- **返回**
  - 修改页面显示的差异数据。
- **返回类型**
  - 在proto文件中定义, 路径为 proto.aoc.snd.DiffDataOut.DiffDataOutMsg

实际案例

可以这样实现：

```
def callDiffJson(self, aoccontext, request):
    dataIn = request
    regPath = dataIn.regPath
    featureName = dataIn.feature
    transId = dataIn.transId
    diffDatas = dataIn.diffDatas
    diffSize = len(diffDatas)
    if diffDatas is None or diffSize == 0:
        return None
    dataOut = DiffDataOutMsg()
    for data in diffDatas:
        diffXml = data
        # modify diffXml
        diffDataOut = diffData()
        diffDataOut.diffData = diffXml
        dataOut.diffDatas.extend([diffDataOut])
    return dataOut
```

***abstract* getCommonDriverInfo**(aoccontext**,** request=None)

引用

具体操作请参见 **aoc.snd.snd.SND.getCommonDriverInfo()**。

***abstract* getConnectInfo**(aoccontext**,** request=None)

定制设备的连接方式和能力

典型场景:

1. 定制设备连接能力。
2. 定制主协议和辅助协议的连接建立方式和连接能力。主协议用于下发配置、建立连接和设备状态，辅助协议用于查询设备。
3. 自定义NETCONF hello报文。如果对接了华为设备，则使用hello报文自定义模式或YANG对接模式。
   a. 自定义协议连接通道。目前，仅支持自定义长连接来建立设备。长连接通道能力可定制，包括查询通道和配置通道。
   b. 自定义通道时，读写可以共享同一个通道。
4. 每个通道可以配置为主备通道或单个通道。
5. 定制连接建立策略。您可以使用预置的连接建立策略，也可以自定义连接建立策略。

功能描述:

该接口用于自定义设备的连接方式和能力。

接口约束:

1. 此接口不是必填接口。父类提供了默认实现。如果您对设备连接方式和连接能力有定制要求，请参考父类注释复制此接口。
   a. 如果使用NETCONF协议，默认使用系统neagent。配置通道和查询通道不共享。 配置通道为主备模式，查询通道为单主模式。设备以模式连接。
   b. 如果使用CLI协议，默认使用系统cliagent。通道在单活模式下配置和查询。
   c. 如果NETCONF作为主备协议，默认使用系统NETCONFcliagent。NETCONF的配置通道和查询通道不共享。 配置通道为主备模式，查询通道为单主模式。设备以模式连接。CLI协议的配置通道和查询通道共享，通道为单主模式。
2. 目前仅支持NETCONF作为主协议，CLI作为辅协议。

说明

该接口用于自定义设备的连接方式和能力。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，有关详细信息，请参见 **aoc.base.aoccontext**.
  - **request** (*str, optional*) - 目前未使用。
- **返回**
  - 设备连接定制信息。
- **返回类型**
  - 由proto文件定义，路径为 proto.aoc.snd.connectinfo.ConnectInfos

实际案例

如果使用单CLI协议，则使用以下示例：

```
def getConnectInfo(self, aoccontext, request=None):
    connect_infos = ConnectInfos()
    connect_info = connect_infos.connectInfo.add()
    # Set the connection protocol type to CLI.
    connect_info.protocolEntity.protocolType = ProtocolEntity.cli
    connect_info.connectPolicy = DEFAULT_CONNECT
    connect_info.channelInfo.writeChannel = SINGLE_CHANNEL
    # Set IsReadShareWrite to true, indicating that the read and write channels are shared.
    connect_info.channelInfo.is_read_share_write = True
    # CLI protocol. The value is fixed to true.
    connect_info.connectionPriority = PRIMARY_CONNECTION
    return connect_infos
```

***abstract* getDiscoverInfo**(aoccontext, request=None)

获取自定义设备发现信息。

典型场景:

在设备管理之前，需要了解设备的基本信息。

功能描述:

获取定制设备发现信息。

接口约束:

1. 前置条件: 设备通过SNMP扫描设备的sysoid，获取硬件版本、厂商和设备类型。
2. 注意事项:
   a. 默认情况下，使用华为SNMP进行发现。用户不需要复制此方法。
   b. 标准SNMP发现：用户需要实现该方法，并定义一些私有MIB。
   c. 自定义发现：用户需要实现该方法，并实现发现方法。
   d. 兼容的发现模式：需要实现该方法，实现控制器内部实现类，并在pkg.json中注册类。
3. 限制条件: 此方法仅适用于NCE-FABRIC产品，不适用于NCE-IP和Lite。

说明

获取自定义设备发现信息。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，有关详细信息，请参见 **aoc.base.aoccontext**.
  - **request** (*str, optional*) - 目前未使用。
- **返回**
  - 设备发现自定义信息。
- **返回类型**
  - 由proto文件定义，路径为 proto.aoc.snd.discoverinfo.DiscoverInfo

实际案例

如果使用默认_DISCOVER_HUAWEI_MIB发现设备，示例如下:

```
def getDiscoverInfo(self, aoccontext, request=None):
    discover_info = DiscoverInfo()
    discover_entity = discover_info.discoverEntity.add()
    discover_entity.discoverPolicy = DEFAULT_DISCOVER_HUAWEI_MIB
    return discover_info
```

***abstract* getEcsConfigParams**(aoccontext**,** request=None)

定制设备类型和差异比较方式。

典型场景:

有些设备支持数据一致性，有些设备不支持。

部分设备需要区分数据源，部分设备不需要数据源。

功能描述:

定制支持数据一致性处理的设备类型和差异比较方式。

接口约束:

无。

说明

1. 根据数据源的不同，数据一致性比较模式分为E2E模式和非E2E模式（WHOLE_DIFF）。
2. 在E2E模式下，差分数据分为转发器产生和控制器产生的差异。
3. 非E2E模式下，差异数据不区分数据源。
4. 默认使用非E2E模式。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*str, optional*) - 字符串类型的保留字段。
- **返回**
  - 自定义配置项的实体类。
- **返回类型**
  - 在proto文件中定义, 路径为 proto.aoc.snd.ecsdriver.EcsConfigOut

实际案例

可以这样实现：

```
def getEcsConfigParams(self, aoccontext, request=None):
    ecsConfigOut = EcsConfigOut()
    ecsConfigOutParas = []
    # syncEnable: Set whether to support consistency check.
    paraSyncEnable = EcsConfigOut.EcsConfigOutPara()
    paraSyncEnable.key = 'syncEnable'
    paraSyncEnable.value = 'true'
    ecsConfigOutParas.add(paraSyncEnable)
    # syncToNe: Setting whether reconciliation is supported
    paraSyncToNe = EcsConfigOut.EcsConfigOutPara()
    paraSyncToNe.key = 'syncEnable'
    paraSyncToNe.value = 'true'
    ecsConfigOutParas.add(paraSyncToNe)
    # syncFromNe : Setting whether synchronization is supported
    paraSyncFromNe = EcsConfigOut.EcsConfigOutPara()
    paraSyncFromNe.key = 'syncFromNe'
    paraSyncFromNe.value = 'true'
    ecsConfigOutParas.add(paraSyncFromNe)
    # Set the difference comparison mode. E2E: E2E mode; WHOLE_DIFF: non-E2E mode
    paraSyncWay = EcsConfigOut.EcsConfigOutPara()
    paraSyncWay.key = 'syncWay'
    paraSyncWay.value = 'WHOLE_DIFF'
    ecsConfigOutParas.add(paraSyncWay)
    ecsConfigOut.para = ecsConfigOutParas
    return ecsConfigOut
```

***abstract* getFeatures**(aoccontext**,** request=None)

定制差异发现、对账和同步的范围。

典型场景:

用户由于业务需要，自定义差异发现、对账和同步使用的特性范围。数据一致性支 持分析SND包中的YANG模型，根据每个模块的顶层容器和顶层List配置节点，生成 差异发现、对账和同步使用的数据路径。

功能描述:

定制数据一致性差异发现、对账和同步使用的数据路径。

接口约束:

无。

说明

定制差异发现、对账和同步的范围。

如果数据没有复制，则可以分析SND包中的YANG模型，并根据每个模块的顶级容器 和顶级列表配置节点生成用于差异发现、对账和同步的数据路径。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*str, optional*) - 保留字段。
- **返回**
  - 自定义特性。
- **返回类型**
  - 在proto文件中定义, 路径为 proto.aoc.snd.FeatureCfgs.FeatureCfgsMsg

实际案例

可以这样实现：

```
def getFeatures(self, aoccontext, request=None):
    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 = name
    feature.operType = Feature.MERGE
    feature.depends.extend([depends])
    function = Function()
    function.value = path
    function.collectPath = path
    feature_name = name.replace('huawei-', '')
    function.preSyncToNe = False
    function.preSyncFromNe = False
    function.postSyncFromNe = False
    feature.functions.extend([function])
    return [feature]
```

***abstract* getProtocolInfo**(aoccontext**,** request=None)

获取主协议信息。

典型场景:

在设备连接建立时，使用南向双协议（NETCONF+cli）获取南向主协议信息。

功能描述:

获取主协议信息.

接口约束:

仅限于NETCONF作为主协议，CLI作为辅协议

说明

获取主协议信息。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，有关详细信息，请参见 **aoc.base.aoccontext**.
  - **request** (*str, optional*) - 目前未使用。
- **返回**
  - 协议信息。
- **返回类型**
  - 由proto文件定义，路径为 proto.aoc.snd.protocolinfo.ProtocolInfo

实际案例

如果主协议为NETCONF，则示例如下:

```
def getProtocolInfo(self, aoccontext, request=None):
    self.logger.info('getProtocolInfo start.')
    protocol_info = ProtocolInfo()
    protocol_info.protocolType = ProtocolInfo.netconf
    return protocol_info
```

***abstract* getSyncId**(aoccontext, request)

此方法从设备获取commitId。

典型场景:

为了验证设备和控制器的commitId，需要自定义此方法来获取设备的commitId。

功能描述:

此方法从设备获取commitId。

接口约束:

设备已上线。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.syncidinfo.SyncIdInfoInput。
- **返回**
  - 获取到的 SyncId 的信息。
- **返回类型**
  - 在 proto 文件中定义，路径为 proto.aoc.snd.syncidinfo.SyncIdInfoOutput

实际案例

描述样例:

```
def getSyncId(self, aoccontext, request):
    device_id = request.deviceId
    path = '/huawei-cfg:cfg'
    properties = {'subtree': self.CFG_SUBTREE.strip()}
```

```
    data = devicemgr.query_data_from_device(device_id, 'OPERATIONAL', path, properties, 60000)
    data_dict = {}
    if data.isSuccessful == "success" and data.result is not None:
        data_dict = NcsService.xmltodictnode(data.result)
    sync_id_info_output = SyncIdInfoOutput()
    if 'cfg' not in data_dict or 'checkpoints' not in data_dict['cfg'] or 'checkpoint' not in
data_dict['cfg']['checkpoints']:
        return sync_id_info_output
    commit_ids = []
    for checkpoint in data_dict['cfg']['checkpoints']['checkpoint']:
        if checkpoint['commit-id']:
            commit_ids.append(checkpoint['commit-id'])
    sync_id_info_output.SyncId = max(commit_ids)
    return sync_id_info_output
request = {}
request.setdefault("deviceId", "123")
request.setdefault("logicSessionId", "1234")
>>> print getSyncId (aoccontext, request)
{"SyncId":"123"}
```

***abstract* getSysoidInfo**(aoccontext**,** request=None)

设置设备sysoid与设备类型、设备型号和供应商的映射关系。

典型场景:

如果要使用SNMP发现新设备，需要根据sysoid与设备类型、设备型号、厂商的映射关系获取设备的基本属性。目前该参数仅在NCE-Fabric上使用。

功能描述:

设置sysoid与设备类型、设备型号和供应商的映射关系，以管理设备。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，有关详细信息，请参见 **aoc.base.aoccontext**.
  - **request** (*str, optional*) - 目前未使用。
- **返回**
  - 设备连接定制信息。
- **返回类型**
  - 由proto文件定义，路径为 proto.aoc.snd.sysoidinfo.SysoidInfo

实际案例

以下是设置设备sysoid与设备类型、设备型号和供应商的映射关系的示例：

```
def getSysoidInfo(self, aoccontext, request=None):
    sysoidInfo = SysoidInfo()
    sysoidEntity = sysoidInfo.sysoidEntity.add()
    # Device sysoid value
    sysoidEntity.sysoid = "1.3.6.1.4.1.2011.2.62.2.18"
    sysoidEntity.deviceType = "ROUTER"
    sysoidEntity.deviceModel = "NE40E-X8A"
    sysoidEntity.deviceVendor = "HUAWEI"
    return sysoidInfo
```

***abstract* postSyncFromNe**(aoccontext**,** request)

定制数据一致性同步后处理。

典型场景:

数据一致性同步后，用户需要使用业务资源。例如，从南向设备同步L3VPN实例后，标签资源被占用。

功能描述:

定制数据一致性同步后处理操作。

接口约束:

要使用此自定义能力，请在getFeatures中将相应特性的postSyncFromNe设置为true。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 要下发的差分数据。 定义在proto文件中，路径为proto.aoc.snd.DiffDataIn.DiffDataInMsg。

实际案例

可以这样实现：

```
def postSyncFromNe(self, request, aoccontext):
    dataIn = request
    regPath = dataIn.regPath
    featureName = dataIn.feature
    transId = dataIn.transId
    diffDatas = dataIn.diffDatas
    diffSize = len(diffDatas)
    if diffDatas is None or diffSize == 0:
        return
    for data in diffDatas:
        diffXml = data
        # modify
```

***abstract* postSyncToNe**(aoccontext, request)

定制对账后的后处理。

典型场景:

数据一致性校验后的处理需要定制。

功能描述:

定制对账后的后处理操作。

接口约束:

要使用此定制能力，请在getFeatures中将相应特性的postSyncToNe设置为true。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 要下发的差分数据。 定义在proto文件中，路径为proto.aoc.snd.DiffDataIn.DiffDataInMsg。

实际案例

可以这样实现：

```
def postSyncToNe(self, request, aoccontext=None):
    dataIn = request
    regPath = dataIn.regPath
    featureName = dataIn.feature
    transId = dataIn.transId
    diffDatas = dataIn.diffDatas
    diffSize = len(diffDatas)l
    if diffDatas is None or diffSize == 0:
        dataOut = DiffDataOutMsg()
        for data in diffDatas:
            diffXml = data
            # modify
            diffDataOut = diffData()
            diffDataOut.diffData = diffXml
            dataOut.diffDatas.extend([diffDataOut])
```

***abstract* preSyncFromNe**(aoccontext**,** request)

定制数据一致性同步预处理操作。

典型场景:

差异比较后，您可以在差异数据写入控制器数据库之前修改写入控制器数据库的数据。

功能描述:

定制数据一致性同步使用的数据。

接口约束:

要使用此定制能力，请在getFeatures中将相应特性的preSyncFromNe属性设置为true。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 要下发的差分数据。 定义在proto文件中，路径为proto.aoc.snd.DiffDataIn.DiffDataInMsg。
- **返回**
  - 修改待下发数据。
- **返回类型**
  - 在proto文件中定义, 路径为 proto.aoc.snd.DiffDataIn.DiffDataOutMsg

实际案例

可以这样实现：

```
def preSyncFromNe(self, request, aoccontext):
    dataIn = request
    regPath = dataIn.regPath
    featureName = dataIn.feature
    transId = dataIn.transId
    diffDatas = dataIn.diffDatas
    diffSize = len(diffDatas)
    if diffDatas is None or diffSize == 0:
        return None
    dataOut = DiffDataOutMsg()
    for data in diffDatas:
        diffXml = data
```



```
    # modify diffXml
    diffDataOut = diffData()
    diffDataOut.singleMsg = False
    diffDataOut.diffData = diffXml
    dataOut.diffDatas.extend([diffDataOut])
return dataOut
```

***abstract* preSyncToNe**(aoccontext**,** request)

定制数据一致性对账预处理。

典型场景:

差异比较后和验证下发前，您可以修改下发的数据，用于数据一致性对账。

功能描述:

定制数据一致性对账使用的数据。

接口约束:

要使用此定制能力，请在getFeatures中将相应特性的preSyncToNe属性设置为true。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 要下发的差分数据。定义在proto文件中，路径为 proto.aoc.snd.DiffDataIn.DiffDataInMsg。
- **返回**
  - 修改待下发数据。
- **返回类型**
  - 在proto文件中定义, 路径为 proto.aoc.snd.DiffDataIn.DiffDataOutMsg

实际案例

可以这样实现：

```
def preSyncToNe(self, request, aoccontext=None):
    dataIn = request
    regPath = dataIn.regPath
    featureName = dataIn.feature
    transId = dataIn.transId
    diffDatas = dataIn.diffDatas
    diffSize = len(diffDatas)
    if diffDatas is None or diffSize == 0:
        return None
    dataOut = DiffDataOutMsg()
    for data in diffDatas:
        diffXml = data
        # modify diffXml
        diffDataOut = diffData()
        diffDataOut.singleMsg = False
        diffDataOut.diffData = diffXml
        dataOut.diffDatas.extend([diffDataOut])
    return dataOut
```

9.1.5 aoc.snd.custom_migrate_capability module

典型场景:

当snd pkg不兼容升级时，数据存储区数据应从一个域迁移到另一个域。 在某些情况下，NCE迁移框架无法处理复杂的数据迁移。

功能描述:

该接口提供钩子功能，供客户处理复杂的数据迁移。

class aoc.snd.custom_migrate_capability.CustomMigrateCapability(logger=None, resourceDir='')

基类：aoc.snd.commonsnd.CommonSND

该接口提供钩子功能，供客户处理复杂的数据迁移。

典型场景:

当snd pkg不兼容升级时，数据存储区数据应从一个域迁移到另一个域。 在某些情况下，NCE迁移框架无法处理复杂的数据迁移。

功能描述:

该接口提供钩子功能，供客户处理复杂的数据迁移。

接口约束:

无。

***abstract* custom_migrate**(aoccontext, aoc_input)

此spi用于处理复杂的数据迁移。

典型场景:

此spi用于snd 包不兼容升级，客户可以通过直接使用 数据库操作工具来覆盖此方法来处理复杂的数据迁移。

功能描述:

此spi用于处理复杂的数据迁移。

接口约束:

此spi用于snd pkg不兼容升级。

- **参数**
  - **aoc_context** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **aoc_input** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.migrate_info.CustomMigrateInput。
- **返回**
  - 受影响源名称列表。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.migrate_info.CustomMigrateOutput

示例

描述样例:

```
def custom_migrate(self, aoccontext, aoc_input):
    ne_id = aoc_input.ne_id;
    gt_id = aoc_input.gt_id;
    out = CustomMigrateOutput()
    out.affected_source.append('www.huawei.com/example')
    return out
aoc_input = {}
aoc_input.setdefault("ne_id", "8a394835-cb84-38f3-44d5-36a7f2074a77")
aoc_input.setdefault("gt_id", "G_8a394835_cb84-38f3_44d5_36a7f2074a77")
>>> print custom_migrate (self, aoccontext, aoc_input)
{"effected_source": "http://huawei.com/nce/netconf-driver"}
```

***abstract* value_change**(aoccontext**,** aoc_input)

此spi用于处理简单的不兼容数据值更改迁移。

典型场景:

此抽象方法用于snd pkg不兼容升级，客户可以覆盖此方法以处理简单的不兼容数据值更改迁移。

功能描述:

此spi用于处理简单的不兼容数据值更改迁移。

接口约束:

此spi用于snd pkg不兼容升级。

- **参数**
  - **aoc_context** (*dict*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **aoc_input** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.migrate_info.ValueChangeInput。
- **返回**
  - 受影响源名称列表。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.migrate_info.ValueChangeOutput

示例

描述样例:

```
def value_change(self, aoccontext, aoc_input):
    ne_id = aoc_input.ne_id;
    path = aoc_input.path;
    old_data = aoc_input.old_data
    out = ValueChangeOutput()
    out.new_data = 'example'
    return out
aoc_input = {}
aoc_input.setdefault("ne_id", "8a394835-cb84-38f3-44d5-36a7f2074a77")
aoc_input.setdefault("path", "http://huawei.com/nce/netconf-driver")
aoc_input.setdefault("old_data", "456")
>>> print value_change (self, aoccontext, aoc_input)
{"new_data": "123"}
```


9.1.6 aoc.snd.customcapability module

典型场景:

1. 如果您需要使用cli、netconf、restconf和restful以外的协议管理设备， 您可以使用此API定制设备驱动程序信息，系统为YANG模型和协议提供默认的转换驱动程序。
2. 您可以自定义驱动程序以与设备交换数据包，而不使用默认协议驱动程序。
3. 目前的限制是设备和系统必须使用系统支持的协议建立：cli协议、netconf协议、restconf协议、restful协议。

功能描述:

用户定义的驱动程序。

class aoc.snd.customcapability.CliCapability(logger, input_types, output_types)

基类：aoc.snd.customcommoncapability.CustomCommonCapability

典型场景:

1. 如果您需要使用cli、netconf、restconf和restful以外的协议管理设备， 您可以使用此API自定义设备驱动程序信息。系统为yang模型和协议提供默认的转换驱动程序。
2. 您可以自定义驱动程序以与设备交换数据包，而不使用默认协议驱动程序。
3. 目前的限制是设备和系统必须使用系统支持的协议建立： cli协议、netconf协议、restconf协议、restful协议。

功能描述:

用户定义的驱动程序

接口约束:

设备与系统之间的连接必须基于系统支持的协议。（ cli、netconf、Restconf和Restful ）

***abstract* config**(aoccontext, request)

用户定义的设备报文下发接口。

典型场景:

配置下发需要两个步骤。将yang模型数据转换为报文，并将转换后的报文下发给设备。 该接口是第二步。

功能描述:

用户定义的设备报文下发接口。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。

  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.customdriver.CustomConfigData。
- **返回**
  - 自定义配置结果的信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.customdriver.CustomConfigResult

实际案例

描述样例:

```
def config(self, aoccontext, request):
    ne_id = request.neId
    for config in request.items:
        pkg_data = config.pkgData
        datas = config.datas
    response = CustomConfigResult()
    response.replyPkg= "success"
    response.result = SUCCESS
    return response
request = {}
request.setdefault("items",[{"pkgData":"pkgData","data":"data"}])
request.setdefault("neId", "123")
>>> print query (self, aoccontext, request)
{"replyPkg":"success", "result":SUCCESS}
```

***abstract* query**(aoccontext, request)

定制从设备读取数据的逻辑，并将数据转换为系统yang模型。

典型场景:

1. snd-yang模型提供了相应的模型查询功能，包括状态数据或配置数据。
2. 该函数由aoc触发，用于从设备同步和查询数据。

功能描述:

用户需要实现该接口，自定义从设备读取数据的逻辑，并将数据转换为系统yang模型。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.customdriver.CustomQueryParameter。
- **返回**
  - 自定义查询结果的信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.customdriver.CustomQueryResult

实际案例

描述样例:

```
def query(self, aoccontext, request):
    ne_id = request.neId
    path = request.path
    query_type = request.queryType
    response = CustomQueryResult()
    response.yangData= "<xml>data</xml>"
    response.result = SUCCESS
    return response
request = {}
request.setdefault("path", "/path1")
request.setdefault("queryType", "GETCONFIG")
request.setdefault("neId", "123")
>>> print query (aoccontext, request)
{"yangData":"<xml>data</xml>", "result":SUCCESS}
```

***abstract* rpc**(aoccontext**,** request)

定制rpcs下发到设备的逻辑，并返回结果。

典型场景:

1. rpc函数是基于snd-yang模型提供的。
2. 该函数由AOC触发，用于从aoc下发rpc操作。

功能描述:

用户需要实现该接口，定制rpcs下发到设备的逻辑，并返回结果。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.customdriver.CustomRpcParameter。
- **返回**
  - 自定义rpc结果的信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.customdriver.CustomRpcResult

实际案例

描述样例:

```
def rpc(self, aoccontext, request):
    ne_id = request.neId
    rpc_qname = request.rpcQname
    rpc_input = request.rpcInput
    response = CustomRpcResult()
    response.rpcOutput= "<output></output>"
    response.result = SUCCESS
    return response
request = {}
request.setdefault("rpcQname", "/path1")
request.setdefault("rpcInput", "/path1: input")
```



```
request.setdefault("neId", "123")
>>> print query (aoccontext, request)
{"rpcOutput":"<output></output>","result":SUCCESS}
```

***abstract* transform**(aoccontext, request)

定制试运行报文转换接口。

典型场景:

配置下发需要两个步骤。将yang模型数据转换为报文，并将转换后的报文下发给设备。 该接口是第二步。

功能描述:

定制试运行报文转换接口。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.customdriver.CustomTransformParameter。
- **返回**
  - 自定义转换结果的信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.customdriver.CustomTransformResult

实际案例

描述样例:

```
def transform(self, aoccontext, request):
    ne_id = request.neId
    yang_data = request.yangData
    data = yang_data.data
    path = yang_data.path
    op = yang_data.op
    response = CustomTransformResult()
    response.data = transform(ne_id,yang_path,data,op)
    return response
request = {}
request.setdefault("yangData",{'path':"/path1", 'data':"data1", 'op':"CREATE"})
request.setdefault("neId", "123")
>>> print query (aoccontext, request)
{"data":{"pkgData":"pkgData","pkgLength":"0", "datas":[{"<output></output>"}]}}
```

9.1.7 aoc.snd.customcommoncapability module

典型场景:

1. 如果您需要使用cli、netconf、restconf和restful以外的协议管理设备， 您可以使用此API定制设备驱动程序信息。
2. 系统为YANG模型和协议提供默认的转换驱动程序，您可以自定义驱动程序以与设备交换数据包， 而不使用默认协议驱动程序。

3. 目前的限制是设备和系统必须使用系统支持的协议建立： cli协议、netconf协议、restconf协议、restful协议。

功能描述:

该模块提供cli-snd定义与设备关联建立和报文交互相关的参数。

class aoc.snd.customcommoncapability.CustomCommonCapability(logger, input_types, output_types)

基类：aoc.base.baseservice.BaseService

典型场景:

1. 如果您需要使用cli、netconf、restconf和restful以外的协议管理设备， 您可以使用此API定制设备驱动程序信息。
2. 系统为YANG模型和协议提供默认的转换驱动程序，您可以自定义驱动程序以与设备交换数据包， 而不使用默认协议驱动程序。
3. 目前的限制是设备和系统必须使用系统支持的协议建立： cli协议、netconf协议、restconf协议、restful协议。

功能描述:

该模块提供cli-snd定义与设备关联建立和报文交互相关的参数。

接口描述:

无。

***abstract* getCustomDriverInfo**(aoccontext, request=None)

定义与设备关联建立和报文交互相关的参数。

典型场景:

1. 如果您需要使用cli、netconf、restconf、restful以外的协议管理设备， 您可以使用该接口自定义设备驱动程序信息。
2. 系统为YANG模型和协议提供默认的转换驱动程序，您可以自定义驱动程序以与设备交换数据包， 而不使用默认协议驱动程序。
3. 目前的限制是设备和系统必须使用系统支持的协议建立： cli协议、netconf协议、restconf协议、restful协议。

功能描述:

该模块提供cli-snd定义与设备关联建立和报文交互相关的参数。

接口描述:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*dict, optional*) - 自定义参数的扩展使用。
- **返回**
  - 自定义驱动程序的信息。
- **返回类型**

- 在 proto 文件中定义， 路径为 proto.aoc.snd.ecsdriver.CustomDriverInfo

实际案例

描述样例:

```
def getCustomDriverInfo(self, aoccontext, request=None):
    cliDriverInfo = CliDriverInfo()
    cliDriverEntity = cliDriverInfo.cliDriverEntity.add()
    cliDriverEntity.key = "protocol"
    cliDriverEntity.value = "Ssh"
    cliDriverEntity2 = cliDriverInfo.cliDriverEntity.add()
    cliDriverEntity2.key = "userModePrompt"
    cliDriverEntity2.value = ".*>.*"
    return cliDriverInfo
>>> print getCustomDriverInfo (aoccontext, None)
[{"protocol":"Ssh"},{"userModePrompt":".*>.*"}]
```

9.1.8 aoc.snd.customsnd module

Custom SND

class aoc.snd.customsnd.CustomSND(logger=None, resourceDir='')

基类： aoc.snd.snd.SND, aoc.snd.customcapability.CliCapability

getCustomDriverInfo(aoccontext, request=None)

引用

具体操作请参见 aoc.snd.customcommoncapability.CustomCommonCapability.getCustomDriverInfo()。

9.1.9 aoc.snd.netconfcapability module

典型场景:

当aoc根据netconf下发配置时，设备或aoc无法识别默认转换规则传递的xml报文或yang数据， 基于上述场景，本模块允许用户修改默认的AOC转换规则。

功能描述:

该模块允许用户更改默认的aoc转换规则。提供以下函数，如getNetconfDriverInfo、netconfTransformer、 netconfPreprocess、netconfPostprocess、to Yang、to NetconfRpc、to YangRpc。

class aoc.snd.netconfcapability.NetconfCapability(logger, input_types, output_types)

基类： aoc.snd.netconfcommoncapability.NetconfCommonCapability

典型场景:

当aoc根据netconf下发配置时，设备或aoc无法识别默认转换规则传递的xml报文或yang数据， 基于上述场景，本模块允许用户修改默认的AOC转换规则。

功能描述:

该模块允许用户更改默认的aoc转换规则。提供以下函数，如getNetconfDriverInfo、netconfTransformer、 netconfPreprocess、netconfPostprocess、to Yang、to NetconfRpc、to YangRpc。

接口约束:

无。

***abstract* getNetconfDriverInfo**(aoccontext**,** request)

Netconf-snd驱动程序参数定制。

典型场景:

当设备建立连接并交换报文时，需要设置参数。 例如:
classification,phase,maxPkgLength,netconfLock ana等。

功能描述:

Netconf-snd驱动程序参数定制。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*str, optional*) - 目前未使用。
- **返回**
  - Netconf驱动程序信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.ecsdriver.NetconfDriverInfo

实际案例

描述样例:

```
def getNetconfDriverInfo(self, aoccontext, request=None):
    netconf_driver = NetconfDriverInfo()
    netconf_driver.phase = "two"
    netconf_driver.classification = "huawei-v5"
    return netconf_driver
>>> print getNetconfDriverInfo (aoccontext, None)
{"phase":"two","classification":"huawei-v5"}
```

netconfPostprocess(aoccontext**,** request)

当系统将yang模型数据转换为netconf报文后，修改netconf报文。

典型场景:

1. 在下发aoc配置时，aoc根据yang模型的默认转换规则，将yang模型数据转换为netconf命令。
2. 但是，在某些特殊场景下，aoc生成的netconf报文与设备上的实际命令不同， 您可以自定义转换后的netconf报文。

功能描述:

当系统将yang模型数据转换为netconf报文后，修改netconf报文。

接口约束:

在CommonDriverInfo函数中需要设置pathNeedPostProcess字段。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.netconfProcess.NetconfAtomicConfig。
- **返回**
  - 修改设备报文后的XML信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.netconfProcess.NetconfAtomicConfig

实际案例

描述样例:

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

netconfPreprocess(aoccontext**,** request)

在系统将yang模型数据转换为netconf报文后，修改xml报文。

典型场景:

1. aoc读取设备数据，根据yang模型的默认转换规则，将设备的netconf报文转换为yang模型数据。
2. 但是，在某些特殊场景下，netconf报文无法根据yang模型结构的默认规则转换为yang数据， 您可以自定义转换实现逻辑。

功能描述:

在系统将yang模型数据转换为netconf报文后，修改xml报文。

接口约束:

在CommonDriverInfo函数中需要设置pathNeedPreProcess字段。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.netconfProcess.NetconfAtomicConfig。
- **返回**
  - 修改设备报文后的XML信息。

- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.netconfProcess.NetconfAtomicConfig

实际案例

描述样例:

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

netconfTransformer(aoccontext**,** request)

提供定制XML和YANG之间的转换接口。

典型场景:

1. 在下发aoc配置时，aoc会根据yang模型的默认转换规则，将yang模型数据转换为netconf报文， 但在某些特殊场景下，aoc生成的netconf报文与设备上的实际报文不一致， 您可以自定义netconftransformer函数的转换。
2. aoc读取设备数据，并根据yang模型的默认转换规则将netconf报文转换为yang模型数据， 但在某些特殊场景下，设备返回的报文不能根据yang模型结构的默认规则转换为yang数据， 您可以自定义转换实现逻辑。

功能描述:

提供定制XML和YANG之间的转换接口。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.netconfTfm.AtomicConfig。
- **返回**
  - Netconf消息的信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.netconfMsg.NetconfMsg

实际案例

描述样例:

```
def netconfTransformer(self, aoccontext, request=None):
    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')
>>> print netconfTransformer (self, aoccontext,
>>>       {"path":"/huawei:yang:huawei-ac-ne-snmp",
>>>       /"data":"<c55 xmlns="http://huawei.com/cli-custom">
>>>       /<l55>hello_world</l55></c55>"})
{"msg":"<c55 xmlns="http://huawei.com/cli-custom"><l55>hello_world</l55></c55>"}
```

toNetconfRpc(aoccontext**,** request)

当系统将yang模型数据转换为netconf rpc报文后，修改netconf rpc报文。

典型场景:

1. aoc与netconf-rpc设备交互分两步。
   a. 将rpc的输入数据转换为设备的分组数据。
   b. 将rpc输出转换为yang模型数据，但在某些特殊场景下，rpc报文不符合设备要求， 您可以自定义转换逻辑。

功能描述:

当系统将yang模型数据转换为netconf rpc报文后，修改netconf rpc报文。

接口约束:

需要在CommonDriverInfo函数中设置pathNeedEcsRpcMapping字段。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.netconfdriver.YangRpcData。
- **返回**
  - 已处理的数据。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.netconfdriver.NetconfConfigMessage

实际案例

描述样例:

```
def toNetconfRpc(self, aoccontext, request):
    ne_id = request.neId
    xml_input= request.xmlInput
    response = NetconfRpcMessage()
    response.msgInput= ""
    return response
request = {}
request.setdefault("neId", "123")
request.setdefault("xmlInput", "<xml>input</xml>")
>>> print toNetconfRpc (aoccontext, request)
{"msgInput":""}
```

toYang(aoccontext**,** request)

Internal function, not exposed to users.

toYangRpc(aoccontext, request)

修改netconf rpc报文，系统netconf rpc报文转换为yang模型数据后。

典型场景:

aoc与netconf-rpc设备交互分两步。 1. 将RPC的输入数据转换为设备的报文数据。 #. 将RPC输出转换为YANG模型数据。

功能描述:

系统netconf rpc报文转换为yang模型数据后，用户可以修改netconf rpc报文。

接口约束:

需要在CommonDriverInfo函数中设置pathNeedEcsRpcMapping字段。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.netconfdriver.NetconfConfigMessage。
- **返回**
  - 已处理的数据。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.netconfdriver.YangRpcData

实际案例

描述样例:

```
def toYangRpc(self, aoccontext, request):
    ne_id = request.neId
    msg_output = request.msgOutput
    response = YangRpcData()
    response.xmlOutput = "<xml>data</xml>"
    return response
request = {}
request.setdefault("neId", "123")
request.setdefault("msgOutput", "<xml>input</xml>")
>>> print toYangRpc (aoccontext, request)
{"xmlOutput":"<xml>data</xml>"}
```

9.1.10 aoc.snd.netconfclisnd module

NetConf SND

class aoc.snd.netconfclisnd.NetconfCliSND(logger=None, resourceDir='')

基类： **aoc.snd.snd.SND**, **aoc.snd.netconfcapability.NetconfCapability**, **aoc.snd.clicapability.CliCapability**

cliToYang(aoccontext, request)

引用

具体操作请参见 **aoc.snd.clicapability.CliCapability.cliToYang()**。

configCliByLinePostProcessor(aoccontext, request)

引用

具体操作请参见 **aoc.snd.clicapability.CliCapability.configCliByLinePostProcessor()**。

configCliPostProcessor(aoccontext, request)

引用

具体操作请参见 **aoc.snd.clicapability.CliCapability.configCliPostProcessor()**。

configRollbackCliPostProcessor(aoccontext, request)

引用

具体操作请参见 **aoc.snd.clicapability.CliCapability.configRollbackCliPostProcessor()**。

getCliDriverInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.clicommoncapability.CliCommonCapability.getCliDriverInfo()**。

getCommonDriverInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.snd.SND.getCommonDriverInfo()**。

getConnectInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.commonsnd.CommonSND.getConnectInfo()**。

getDiscoverInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.commonsnd.CommonSND.getDiscoverInfo()**。

getKeepAliveInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.snd.SND.getKeepAliveInfo()**。

getPassthroughCommands(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.clicapability.CliCapability.getPassthroughCommands()**。

getPrimaryProtocolInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.snd.SND.getPrimaryProtocolInfo()**。

getProtocolInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.commonsnd.CommonSND.getProtocolInfo()**。

readCliPreProcessor(aoccontext, request)

引用

具体操作请参见 **aoc.snd.clicapability.CliCapability.readCliPreProcessor()**。

yangToCli(aoccontext, request)

引用

具体操作请参见 **aoc.snd.clicapability.CliCapability.yangToCli()**。

9.1.11 aoc.snd.netconfcommoncapability module

典型场景:

当设备建立连接并交换报文时，需要设置参数。 例如:
classification,phase,maxPkgLength,netconfLock ana等。

功能描述:

Netconf-snd驱动程序参数定制。

class aoc.snd.netconfcommoncapability.NetconfCommonCapability(logger, input_types, output_types)

基类：aoc.base.baseservice.BaseService

典型场景:

当设备建立连接并交换报文时，需要设置参数。 例如:
classification,phase,maxPkgLength,netconfLock ana等。

功能描述:

Netconf-snd驱动程序参数定制。

接口约束:

无。

abstract **getNetconfDriverInfo**(aoccontext, request)

引用

具体操作请参见
aoc.snd.netconfcapability.NetconfCapability.getNetconfDriverInfo()。

9.1.12 aoc.snd.netconfsnd module

NetConf SND

class aoc.snd.netconfsnd.NetconfSND(logger=None, resourceDir='')

基类：aoc.snd.snd.SND, aoc.snd.netconfcapability.NetconfCapability

getCommonDriverInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.snd.SND.getCommonDriverInfo()**。

getConnectInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.commonsnd.CommonSND.getConnectInfo()**。

getDiscoverInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.commonsnd.CommonSND.getDiscoverInfo()**。

getKeepAliveInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.snd.SND.getKeepAliveInfo()**。

getNetconfDriverInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.netconfcapability.NetconfCapability.getNetconfDriverInfo()**。

getPrimaryProtocolInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.snd.SND.getPrimaryProtocolInfo()**。

getProtocolInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.commonsnd.CommonSND.getProtocolInfo()**。

9.1.13 aoc.snd.restcapability module

典型场景:

如果南向协议为REST协议，则可以读取REST协议.

功能描述:

提供获取REST协议的token定制信息的方法.主要方法:getRestProtocolInfo

class aoc.snd.restcapability.RestCapability(logger, input_types, output_types)

基类：aoc.base.baseservice.BaseService

典型场景:

如果南向协议为REST协议，则可以读取REST协议.

功能描述:

提供获取REST协议的token定制信息的方法.主要方法:getRestProtocolInfo

***abstract* getRestProtocolInfo**(aoccontext, request)

获取REST协议的token定制信息。

典型场景:

1. 如果南向协议为REST协议，则可以使用此方法设置REST协议的令牌定制信息。
2. 如果不实现该方法，可以在rest-protocol.xml文件中配置REST协议的token定制信息。

功能描述:

获取REST协议的token定制信息。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，有关详细信息，请参见 **aoc.base.aoccontext**.
  - **request** (*str, optional*) - 目前未使用。
- **返回**
  - REST协议定制信息。
- **返回类型**
  - 由proto文件定义，路径为 proto.aoc.snd.restProtocolInfo.RestProtocolInfo

实际案例

在rest-protocol.xml文件中配置token定制信息。示例如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<rest-protocol-info xmlns="http://huawei.com/nce/rest-agent">
   <tokenAuth>
      <isTokenAuth>true</isTokenAuth>
      <tokenFreshTime>100</tokenFreshTime>
      <tokenUrl>https://rest/plat/smapp/v1/oath/token</tokenUrl>
      <tokenMethod>POST</tokenMethod>
      <tokenPayLoad>
         {"userName":$userName,"grantType":"password","value":$password}
      </tokenPayLoad>
      <tokenValuePath>$.accessSession</tokenValuePath>
      <headerTokenRef>X-AUTH-TOKEN</headerTokenRef>
   </tokenAuth>
</rest-protocol-info>
```

9.1.14 aoc.snd.restconfcapability module

典型场景:

当aoc基于restconf协议下发配置时，设备或aoc无法识别默认转换规则传递的请求报文， 基于上述场景，本模块允许用户修改默认的aoc转换规则。

功能描述:

该模块允许用户更改默认的aoc转换规则。 提供的功能如下：
configCustomYangToRestconf, rpcCustomPreProcess, configCustomRestconfToYang, rpcCustomPostProcess, readCustomPostProcess, rpcCustomYangToRestconf, rpcCustomRestconfToYang, readCustomPostProcess, readCustomPreProcess, readCustomYangToRestconf, readCustomRestconfToYang, configCustomYangToRestconf, errorCustomPreProcess, configCustomPreProcess.

class aoc.snd.restconfcapability.RestconfCapability(logger, input_types, output_types)

基类：aoc.snd.restconfcommoncapability.RestconfCommonCapability

典型场景:

当aoc基于restconf协议下发配置时，设备或aoc无法识别默认转换规则传递的请求报文， 基于上述场景，本模块允许用户修改默认的aoc转换规则。

功能描述:

该模块允许用户更改默认的aoc转换规则。 提供的功能如下:
configCustomYangToRestconf, rpcCustomPreProcess,
configCustomRestconfToYang, rpcCustomPostProcess, readCustomPostProcess,
rpcCustomYangToRestconf, rpcCustomRestconfToYang, readCustomPostProcess,
readCustomPreProcess, readCustomYangToRestconf, readCustomRestconfToYang,
configCustomYangToRestconf, errorCustomPreProcess, configCustomPreProcess.

接口约束:

无。

***abstract* configCustomPostProcess**(aoccontext**,** request)

在restconfdriver生成下发到设备的请求报文后，处理生成的请求报文。

典型场景:

1. restconfdriver根据yang模型执行默认的yang-restconf转换。
2. 但是，在某些特殊情况下，restconf驱动生成的restconf报文无法满足设备的要求。
3. 需要重新修改自定义参数，以满足下发参数到设备的要求。

功能描述:

在restconfdriver生成下发到设备的请求报文后，处理生成的请求报文。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfTransform.ConfigCustomPostProcessInput。
- **返回**
  - restconf的请求信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.RestconfRequestInfo

实际案例

描述样例:

```
def configCustomPostProcess(self, aoccontext, input):
    configRequestInfo = input.configReqeustInfo
    restconfRequestInfo = input.restconfRequestInfo
    url = restconfRequestInfo.url
    httpMethod = restconfRequestInfo.httpMethod
    requestBody = restconfRequestInfo.requestBody
    if configRequestInfo.path == 'testconfigpath':
```



```
        restconfRequestInfo.url = url
        restconfRequestInfo.httpMethod = httpMethod
        restconfRequestInfo.requestBody = requestBody
    return restconfRequestInfo
input = {}
input.setdefault("configReqeustInfo", {"neId":"123","path":"/path1"})
input.setdefault("restconfRequestInfo", {"url":"/test123","httpMethod":"DELETE"})
>>> print configCustomPostProcess (self, aoccontext, input)
"url":"/test123","httpMethod":"DELETE","responseBody":"responseBody_input"}
```

***abstract* configCustomPreProcess**(aoccontext**,** request)

config操作返回响应报文后发送给restconfdriver。

典型场景:

1. restconfdriver根据yang模型执行默认的restconf-yang转换。
2. 但是，在某些特殊情况下，设备返回的restconf报文不能作为restconf驱动的入参。
3. 需要再次修改restconf报文，该参数可以作为restconf-yang转换的入参。

功能描述:

config操作返回响应报文后发送给restconfdriver。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfTransform.ConfigCustomPreProcessInput。
- **返回**
  - restconf的响应信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.RestconfReponseInfo

实际案例

描述样例:

```
def configCustomPreProcess(self, aoccontext, input):
    configRequestInfo = input.configRequestInfo
    restconfResponseInfo = input.restconfResponseInfo
    responseBody = restconfResponseInfo.responseBody
    if configRequestInfo.path == 'testconfigpath':
        restconfResponseInfo.responseBody = responseBody
    return restconfResponseInfo
input = {}
input.setdefault("configReqeustInfo", {"neId":"123","path":"testconfigpath"})
restconfRequestInfo = {"url":"/
test123","httpMethod":"DELETE","responseBody": /"responseBody_input"}
input.setdefault("restconfRequestInfo", restconfRequestInfo)
>>> print configCustomPreProcess (self, aoccontext, input)
{"url":"/test123","httpMethod":"DELETE","responseBody":"responseBody_input"}
```


***abstract* configCustomRestconfToYang**(aoccontext**,** request)

客户可以自定义将restconf转换为yang的逻辑。

典型场景:

如果restconfdriver不支持该功能，可以使用自定义的restconf-to-yang方法解决配置转换问题。

功能描述:

客户可以自定义将restconf转换为yang的逻辑。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfTransform.ConfigCustomRestconfToYangInput。
- **返回**
  - config的响应信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.ConfigReponseInfo

实际案例

描述样例:

```
def configCustomRestconfToYang(self, aoccontext, request):
    configRequestInfo = request.configRequestInfo
    restconfResponseIno = request.restconfReponseInfo
    responseBody = restconfResponseInfo.responseBody
    configReponseInfo = ConfigReponseInfo()
    if 'testconfigpath' == configRequestInfo.path:
        # 自定义转换逻辑
        configReponseInfo.responseBody = responseBody
    return configReponseInfo
input = {}
input.setdefault("restconfRequestInfo", {"neId":"123","path":"testreadpath"})
>>> print configCustomRestconfToYang (self, aoccontext, input)
{"data":"<xml></xml>"}
```

***abstract* configCustomYangToRestconf**(aoccontext**,** request)

客户可以自定义将yang转换为restconf的逻辑。

典型场景:

如果restconfdriver不支持该功能，可以使用自定义的yang-to-restconf转换方法来解决配置操作转换问题。

功能描述:

客户可以自定义将yang转换为restconf的逻辑。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.ConfigRequestInfo。
- **返回**
  - restconf的请求信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.RestconfRequestInfo

实际案例

描述样例:

```
def configCustomYangToRestconf(self, aoccontext, request):
    url = None
    httpMethod = None
    requestBody = None
    restconfRequestInfo = RestconfRequestInfo()
    if request.path == 'testconfigpath':
        restconfRequestInfo.url = url
        restconfRequestInfo.httpMethod = httpMethod
        restconfRequestInfo.requestBody = requestBody
    else:
        return restconfRequestInfo
input = {}
input.setdefault("ConfigRequestInfo", {"neId":"123","path":"testreadpath"})
input.setdefault("restconfRequestInfo", {"url":"/test123","httpMethod":"DELETE"})
>>> print configCustomYangToRestconf (self, aoccontext, input)
{"url":"/test123","httpMethod":"DELETE","responseBody":"responseBody_input"}
```

***abstract* errorCustomPreProcess**(aoccontext**,** request)

设备对返回的错误消息进行预处理。

典型场景:

1. 一般情况下，restconf在获得错误响应后，会解析响应报文中的错误信息。
2. 如果设备返回的报文无法使用默认方法解析，则设备对返回的报文进行预处理。
3. restconfdriver处理可解析的报文数据结构，在页面上显示错误信息。

功能描述:

当设备返回错误消息时，设备对返回的错误消息进行预处理。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfTransform.ErrorCustomPreProcessInput。
- **返回**

  - restconf的响应信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.RestconfReponseInfo

实际案例

描述样例:

```
def errorCustomPreProcess(self, aoccontext, input):
    rpcRequestInfo = input.restconfRequestInfo
    errorMsg= None
    response = RestconfReponseInfo()
    response.responseBody= errorMsg
    return response
input = {}
input.setdefault("restconfRequestInfo", {"neId":"123","path":"testreadpath"})
>>> print errorCustomPreProcess (self, aoccontext, input)
{"data":"<xml></xml>"}
```

***abstract* readCustomPostProcess**(aoccontext, request)

在restconfdriver生成要下发到设备的restconf查询报文后，处理生成的请求报文。

典型场景:

读操作返回响应后，系统会在将响应发送到Restconfdriver之前处理响应。

功能描述:

在restconfdriver生成要下发到设备的restconf查询报文后，处理生成的请求报文。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfTransform.ReadCustomPostProcessInput。
- **返回**
  - restconf的请求信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.RestconfRequestInfo

实际案例

描述样例:

```
def readCustomPostProcess(self, aoccontext, input):
    readReqeustInfo = input.readRequestInfo
    restconfRequestInfo = input.restconfRequestInfo
    url = restconfRequestInfo.url
    httpMethod = restconfRequestInfo.httpMethod
    if readRequestInfo.path == 'testpath':
        restconfRequestInfo.url = url
        restconfRequestInfo.httpMethod = httpMethod
```



```
    return restconfRequestInfo
input = {}
input.setdefault("readRequestInfo", {"neId":"123","path":"/path1"})
input.setdefault("restconfRequestInfo", {"url":"/test123","httpMethod":"DELETE"})
>>> print readCustomPostProcess (self, aoccontext, input)
{"url":"/test123","httpMethod":"DELETE"}
```

***abstract* readCustomPreProcess**(aoccontext**,** request)

根据yang模型执行默认的restconf-yang转换。

典型场景:

读操作返回响应后，系统会在将响应发送到Restconfdriver之前处理响应。

功能描述:

1. restconf驱动程序根据yang模型执行默认的restconf-yang转换。
2. 但在某些特殊情况下，设备返回的restconf报文不能作为restconfdriver转换的入参。
3. 需要再次修改restconf报文，该参数可以作为restconf-yang转换的入参。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfTransform.ReadCustomPreProcessInput。
- **返回**
  - restconf的响应信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.RestconfReponseInfo

实际案例

描述样例:

```
def readCustomPreProcess(self, aoccontext, input):
    readRequestInfo = input.readRequestInfo
    restconfResponse = input.restconfResponse
    responseBody = restconfResponse.responseBody
    if readRequestInfo .path == 'testpath':
        restconfResponse.responseBody = responseBody
    return restconfResponse
input = {}
input.setdefault("readRequestInfo", {"neId":"123","path":"/path1"})
input.setdefault("restconfRequestInfo", {"url":"/test123","httpMethod":"DELETE"})
>>> print readCustomPreProcess (self, aoccontext, input)
{"url":"/test123","httpMethod":"DELETE","responseBody":"responseBody_input"}
```

***abstract* readCustomRestconfToYang**(aoccontext**,** request)

定制查询操作将restconf转换为yang的逻辑。

典型场景:

如果restconfdriver不支持该功能，可以使用自定义的restconf-to-yang方法解决查询操作转换问题。

功能描述:

客户可以自定义将restconf转换为yang进行查询操作的逻辑。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfTransform.ReadCustomRestconfToYangInput。
- **返回**
  - 读取的响应信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.ReadReponseInfo

实际案例

描述样例:

```
def readCustomRestconfToYang(self, aoccontext, input):
    rpcRequestInfo = input.readRequestInfo
    data = None
    if rpcRequestInfo.path == 'testreadpath':
        data = "<xml></xml>"
    response = ReadReponseInfo()
    response.data= data
    return response
input = {}
input.setdefault("readRequestInfo", {"neId":"123","path":"testreadpath"})
>>> print readCustomRestconfToYang (self, aoccontext, input)
{"data":"<xml></xml>"}
```

***abstract* readCustomYangToRestconf**(aoccontext**,** request)

定制查询操作中将yang转换为restconf的逻辑。

典型场景:

如果restconfdriver不支持该功能，则使用自定义的yang-to-restconf转换方法解决查询操作转换问题。

功能描述:

定制查询操作中将yang转换为restconf的逻辑。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.ReadRequestInfo。

- **返回**
  - restconf的请求信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.RestconfRequestInfo

实际案例

描述样例:

```
def readCustomYangToRestconf(self, aoccontext, request):
    restconfRequestInfo = RestconfRequest()
    url = None
    if request.path == "testreadpath":
        restconfRequestInfo.url = url
    return restconfRequestInfo
input = {}
input.setdefault("readRequestInfo", {"neId":"123","path":"testreadpath"})
input.setdefault("restconfRequestInfo", {"url":"/test123","httpMethod":"DELETE"})
>>> print readCustomYangToRestconf (self, aoccontext, input)
{"url":"/test123","httpMethod":"DELETE","responseBody":"responseBody_input"}
```

***abstract* rpcCustomPostProcess**(aoccontext**,** request)

当系统将yang模型数据转换为restconf rpc报文后，修改restconf rpc报文。

典型场景:

1. aoc根据yang模型执行默认的yang-restconf转换。
2. 但是，在某些特殊情况下，restconf驱动生成的restconf rpc报文无法满足设备的要求。
3. aoc生成要下发到设备的restconf rpc报文后，aoc处理生成的请求rpc报文。

功能描述:

当系统将yang模型数据转换为restconf rpc报文后，修改restconf rpc报文。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfTransform.RpcCustomPostProcessInput。
- **返回**
  - Restconf的请求信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.RestconfRequestInfo

实际案例

描述样例:

```
def rpcCustomPostProcess(self, aoccontext, input):
    requestInfo = input.rpcRequestInfo
```



```
    restconfRequestInfo = input.restconfRequestInfo
    requestBody = None
    httpMethod = None
    if requstInfo.rpcQname == "testqname":
        restconfRequestInfo.requestBody = requestBody
        restconfRequestInfo.httpMethod = httpMethod
    return restconfRequest
input = {}
input.setdefault("rpcRequestInfo", "/path1")
input.setdefault("restconfRequestInfo", "123")
request.setdefault("rpcQname", "testqname")
>>> print rpcCustomPostProcess (self, aoccontext, input)
{"requestBody":"https:///123","httpMethod":"GET"}
```

***abstract* rpcCustomPreProcess**(aoccontext**,** request)

在系统将restconf rpc报文转换为yang模型数据之前，请修改restconf rpc报文。

典型场景:

aoc读取设备数据，并根据yang模型的默认转换规则将restconf rpc报文转换为yang模型数据。

但是，在一些特殊场景下，根据yang模型结构的默认规则，用户自定义的转换实现逻辑， 不能将restconf rpc报文转换为yang数据。

功能描述:

在系统将restconf rpc报文转换为yang模型数据之前，用户可以修改restconf rpc报文。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfTransform.RpcCustomPreProcessInput。
- **返回**
  - Restconf的响应信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.RestconfReponseInfo

实际案例

描述样例:

```
def rpcCustomPreProcess(self, aoccontext, input):
    requestInfo = input.rpcReqeustInfo
    respopnseInfo = input.restResponseInfo
    responseBody = None
    if requestInfo.rpcQname == "testqname":
        responseInfo.responseBody = responseBody
    return responseInfo
input = {}
input.setdefault("rpcReqeustInfo", "/path1")
input.setdefault("restResponseInfo", "123")
```



```
input.setdefault("rpcQname", "testqname")
>>> print rpcCustomPreProcess (self, aoccontext, input)
{"responseBody":"https:///123","httpMethod":"GET"}
```

***abstract* rpcCustomRestconfToYang**(aoccontext**,** request)

在rpc操作中定制将restconf转换为yang的部分逻辑。

典型场景:

如果restconfdriver不支持此功能，可以使用自定义的restconf-to-yang方法解决rpc操作转换问题。

功能描述:

在rpc操作中定制将restconf转换为yang的部分逻辑。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfTransform.RpcCustomRestconfToYangInput。
- **返回**
  - rpc的响应信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.RpcReponseInfo

实际案例

描述样例:

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

***abstract* rpcCustomYangToRestconf**(aoccontext**,** request)

在rpc操作中定制将yang转换为restconf的部分逻辑。

典型场景:

如果restconfdriver不支持此功能，可以使用自定义的yang-to-restconf方法解决rpc操作转换问题。

功能描述:

在rpc操作中定制将yang转换为restconf的部分逻辑。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.RpcRequestInfo。
- **返回**
  - Restconf的请求信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.RestconfRequestInfo

实际案例

描述样例:

```
def rpcCustomYangToRestconf(self, aoccontext, request):
    url = None
    httpMethod = None
    if request.rpcQName == "/path1":
        url = '/sample/url_1'
        httpMethod = POST
    requestInfo = RestconfRequestInfo()
    requestInfo.url = url
    requestInfo.httpMethod = httpMethod
    return requestInfo
request = {}
request.setdefault("rpcQname", "/path1")
>>> print rpcCustomYangToRestconf (self, aoccontext, request)
{"url":"/sample/url_1","httpMethod":"POST"}
```

9.1.15 aoc.snd.restconfcommoncapability module

典型场景:

当aoc根据restconf下发配置时，需要设置设备连接建立参数和配置下发参数。

功能描述:

本模块提供了使用restconf建立关联的方法和设置参数的方法。

class aoc.snd.restconfcommoncapability.RestconfCommonCapability(logger, input_types, output_types)

基类：aoc.base.baseservice.BaseService

典型场景:

当aoc根据restconf下发配置时，需要设置设备连接建立参数和配置下发参数。

功能描述:

本模块提供了使用restconf建立关联的方法和设置参数的方法。

接口约束:

无。

***abstract* getRestconfDriverInfo**(aoccontext**,** request=None)

通过此接口获取restconf驱动程序信息。

典型场景:

您可以使用此方法配置必要的驱动程序信息。

功能描述:

1. 在不同的场景下，restconf驱动默认机制使用的httpmethod可能与实际需求不同。
2. 对于不同的路径，同一操作对应的http方法可能不同，在某些情况下， 客户需要识别某些路径的非原子操作。
3. 如果下发失败，会提示控制器的配置与设备的配置不一致。
4. 上述配置信息可以通过该接口定制。

接口约束:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*dict, optional*) - 自定义参数的扩展使用。
- **返回**
  - Restconf 驱动的信息。
- **返回类型**
  - 在 proto 文件中定义， 路径为 proto.aoc.snd.restconfDriver.proto

实际案例

描述样例:

```
def getRestconfDriverInfo(self, aoccontext, request=None):
    restconf_driver_info = RestconfDriverInfo()
    restconf_driver_entity = restconf_driver_info.restconfDriverEntity.add()
    restconf_driver_entity.key = "configOperationUrlPrefix"
    restconf_driver_entity.value= "/restconf/v1/config"
    path_entity = restconf_driver_info.pathConfigEntity.add()
    path_entity.atomic = True
    path_entity.path ="/restconf/data/huawei-nce-tunnel-trail:tunnel-trails/tunnel-trail"
    path_entity.operation["create"] = "POST"
    path_entity.operation["delete"] = "DELETE"
    return restconf_driver_info
>>> print getRestconfDriverInfo(aoccontext, request)
{"data":"<xml></xml>"}
```

9.1.16 aoc.snd.restconfsnd module

RESTCONF SND

class aoc.snd.restconfsnd.RestconfSND(logger=None, resourceDir='')

基类： **aoc.snd.snd.SND**, **aoc.snd.restconfcapability.RestconfCapability**, **aoc.snd.restcapability.RestCapability**

getCommonDriverInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.snd.SND.getCommonDriverInfo()**。

getConnectInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.commonsnd.CommonSND.getConnectInfo()**。

getDeviceClassification(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.snd.SND.getDeviceClassification()**。

getDiscoverInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.commonsnd.CommonSND.getDiscoverInfo()**。

getPrimaryProtocolInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.snd.SND.getPrimaryProtocolInfo()**。

getProtocolInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.commonsnd.CommonSND.getProtocolInfo()**。

getRestProtocolInfo(aoccontext, request)

引用

具体操作请参见 **aoc.snd.restcapability.RestCapability.getRestProtocolInfo()**。

getRestconfDriverInfo(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.restconfcommoncapability.RestconfCommonCapability.getRestconfDriverInfo()**。

getSysoidInfo(aoccontext, request=None)

引用

For details, see **aoc.snd.commonsnd.CommonSND.getSysoidInfo()**.

9.1.17 aoc.snd.snd module

SND Common snd Operation

class aoc.snd.snd.SND(logger=None, resourceDir='')

基类：aoc.snd.commonsnd.CommonSND

abstract **closeConnection**(aoccontext, request)

生成对接数据，用于下发配置到设备。

典型场景:

1. aoc下发配置到设备后，部分设备会生成其他配置数据。
2. 但是，aoc只存储关联前的配置数据。导致AOC和设备数据不一致。
3. 在导入数据库之前，您可以使用此方法修改配置数据。
4. 确保aoc与设备的数据一致性。

Function Description:

生成对接数据，用于下发配置到设备。

Constraint:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*obj*) - 在 proto 文件中定义， 路径为 proto.aoc.snd.ecsDbProcess: EcsDbConfigInput。
- **返回**
  - 处理数据。
- **返回类型**
  - 在 proto 文件中定义，路径为 proto.aoc.snd.ecsDbProcess.EcsDbConfigOut

实际案例

描述样例:

```
def dbPostProcess(self, aoccontext, request):
    ecsData = request.data
    ecsPath = request.path
    ecsopType = request.opType
    ecsDbConfigOut = EcsDbConfigOut()
    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"
    return ecsDbConfigOut

request = {}
request.setdefault("data",
"<c55 xmlns="http://huawei.com/cli-custom"><l55>hello_world</l55></c55>")
request.setdefault("path", "/huawei-ifm:interfaces/huawei-ifm:interface/")
request.setdefault("opType", "CREATE")
>>> print dbPostProcess (aoccontext, request)
{data", "<c55 xmlns="http://huawei.com/cli-custom"><l55>hello_world</l55></c55>",
"path": "/huawei-ifm:interfaces/huawei-ifm:interface/","opType", "CREATE"}
```

***abstract* getCommonDriverInfo**(aoccontext, request=None)

配置不同型号设备的驱动信息。

Typical Scenarios:

您可以使用此方法配置必要的驱动程序信息。

Function Description:

不同型号的设备，人机交互命令可能不同。因此，需要为不同型号的设备配置驱动程序信息。

Constraint:

无。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **request** (*dict, optional*) - 自定义参数的扩展使用。
- **返回**
  - 普通驱动程序的信息。
- **返回类型**
  - 在 proto 文件中定义，路径为 proto.aoc.snd.ecsdriver.proto

实际案例

描述样例:

```
def getCommonDriverInfo(self, aoccontext, request=None):
    common_driver = CommonDriverInfo()
    common_driver.deleteStrategy = 1
    syncToDel = common_driver.para.add()
    syncToDel.key = "sync-to-del-enable"
    syncToDel.value = "true"
    return common_driver
>>> print getCommonDriverInfo (self, aoccontext, None)
["deleteStrategy":1,"para":[{"sync-to-del-enable":"true"}]]
```

***abstract* getConnectInfo**(aoccontext, request=None)

引用

具体操作请参见 **aoc.snd.commonsnd.CommonSND.getConnectInfo()**。

getDeviceClassification(aoccontext, request=None)

获取设备类型信息。

典型场景:

当设备被控制器管理时，可以通过此接口获取类型。

功能描述:

获取设备类型信息。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，有关详细信息，请参见 **aoc.base.aoccontext**.

  - **request** (*str, optional*) - 目前未使用。
- **返回**
  - 设备分类信息。
- **返回类型**
  - 由proto文件定义，路径为 proto.aoc.snd.deviceClassification.DeviceClassificationInfo

实际案例

如果设备类型为路由器，则示例如下:

```
def getDeviceClassification(self, aoccontext, request=None):
    device_classification_info = DeviceClassificationInfo()
    device_classification_info.protocolType = DeviceClassificationInfo.ROUTER
    return device_classification_info
```

abstract **getDiscoverInfo**(aoccontext**,** request=None)

引用

具体操作请参见 **aoc.snd.commonsnd.CommonSND.getDiscoverInfo()**。

classmethod **getEcsConfigParams**(aoccontext**,** request=None)

引用

详细信息请参考 **aoc.snd.commonsnd.CommonSND.getEcsConfigParams()**。

abstract **getKeepAliveInfo**(aoccontext**,** request=None)

获取设备的NETCONF保活报文定制信息。

典型场景:

1. 如果南向协议为NETCONF，则可以使用此方法设置NETCONF保活报文定制信息。 如果某些设备长时间不向控制器发送NETCONF保活报文，则会断开与控制器的连接。
2. 如果不实现该方法，NETCONF保活信息可以在netconf-driver.xml文件中配置。
3. 默认使用SSH保活。

功能描述:

获取设备的NETCONF保活报文定制信息。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，有关详细信息，请参见 **aoc.base.aoccontext**.
  - **request** (*str, optional*) - 目前未使用。
- **返回**
  - 设备连接定制信息。
- **返回类型**
  - 由proto文件定义，路径为 proto.aoc.snd.keepAliveinfo.KeepAliveInfo

实际案例

在netconf-driver.xml文件中配置NETCONF保活信息。示例如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
   ~ Copyright (c) Huawei Technologies Co., Ltd. 2020-2021. All rights reserved.
   -->
   <netconf-driver xmlns="http://huawei.com/nce/netconf-driver">
      <netconf-keepalive-info>
         <!--keepAliveMsg is a NETCONF keepalive packet, which must be escaped
            using CDATA. It complies with the NETCONF XML packet format.
            The value of message-id must be replaced with ${messageId}-->
         <keepAliveMsg><![CDATA[<?xml version="1.0" encoding="UTF-8"?>
            <rpc message-id="${messageId}"
            xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
            <get-config><source><running/></source><filter type="subtree">
            <system xmlns="http://www.huawei.com/netconf/vrp"
            content-version="1.0" format-version="1.0"> <systemInfo>
            <sysUpTime/></systemInfo></system>
            </filter></get-config></rpc>]]></keepAliveMsg>
         <!--keepAlivePeriod: keepalive interval, in seconds.
            The minimum value is 10.-->
         <keepAlivePeriod>15</keepAlivePeriod>
   </netconf-keepalive-info>
</netconf-driver>
```

***abstract* getPrimaryProtocolInfo**(aoccontext**,** request=None)

获取主协议信息。

典型场景:

在设备连接建立时，使用南向双协议（NETCONF+cli）获取南向主协议信息。

功能描述:

获取南向主协议信息。

接口约束:

仅适合NETCONF是主协议，CLI是辅协议的场景。

说明

获取主协议信息。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，有关详细信息，请参见 **aoc.base.aoccontext**.
  - **request** (*str, optional*) - 目前未使用。
- **返回**
  - 协议信息。
- **返回类型**
  - 由proto文件定义，路径为 proto.aoc.snd.protocolinfo.ProtocolInfo

实际案例

如果主协议为NETCONF，则示例如下:

```
def getProtocolInfo(self, aoccontext, request=None):
   self.logger.info('getPrimaryProtocolInfo start.')
   protocol_info = ProtocolInfo()
```

```
    protocol_info.protocolType = ProtocolInfo.netconf
    return protocol_info
```

***abstract* getProtocolInfo**(aoccontext**,** request=None)

引用

具体操作请参见 **aoc.snd.commonsnd.CommonSND.getProtocolInfo()**。

***abstract* getSysoidInfo**(aoccontext**,** request=None)

引用

For details, see **aoc.snd.commonsnd.CommonSND.getSysoidInfo()**.

***abstract* parseMibNode**(aoccontext**,** request)

Internal function, not exposed to users.

10 aoc.sys

10.1 aoc.sys package

10.1.1 aoc.sys.alarm_mgr module

典型场景:

发送自定义警报。

功能描述:

提供告警管理能力。

class aoc.sys.alarm_mgr.AlarmMgr(logger)

基类: object

提供报警管理功能。

典型场景:

发送自定义警报。

功能描述:

提供报警管理功能。

接口约束:

无。

- **参数**
  - **logger** (*Log*) - 当前包组的日志打印。

logger

告警信息对象的日志打印。

Type：Logger

write_alarm(alarms)

向控制器报告警报。

典型场景:

触发Snmp告警，向北向发送告警数据。

功能描述:

向控制器报告警报。

接口约束:

无。

- **参数**
  - **alarms** (*AocAlarmEntity*) - AocAlarmEntity的列表，详细信息请参考：class：proto.aoc.event.AocAlarm.AocAlarmEntity。

实际案例

```
>>> from aoc.event.event_model_pb2.AocAlarm_pb2 import AocAlarmEntity
>>> alarm_list = list()
>>> alarm_data = AocAlarmEntity()
>>> alarm_data.alarmId = '0x0de42013'
>>> alarm_data.alarmName = 'hwSpsUcssLinkDownTrap'
>>> alarm_list.append(alarm_data)
>>> alarm_mgr = AlarmMgr(self.logger)
>>> alarm_mgr.write_alarm(alarm_list)
```

10.1.2 aoc.sys.cliproxy module

经典场景:

服务通过Python侧的cli查询设备配置或运行数据。

功能描述:

1. 业务通过python侧的接口，通过cli从设备获取设备配置数据或运行数据。
2. 具体处理如下:
   a. 服务设置sendCli的入参包括neid、cli命令和超时时间。
   b. Java调用CLI协议向设备下发cli命令。
   c. 设备通过Python代理将命令行查询结果返回给服务。

aoc.sys.cliproxy.sendCli(neid, clicommand, timeout=60)

通过cli从设备获取设备配置数据或运行数据。

经典场景:

服务通过Python侧的cli查询设备配置或运行数据。

功能描述:

1. 业务通过python侧的接口，通过cli从设备获取设备配置数据或运行数据。
2. 具体处理如下:
   a. 服务设置sendCli的入参包括neid、cli命令和超时时间。
   b. Java调用CLI协议向设备下发cli命令。
   c. 设备通过Python代理将命令行查询结果返回给服务。

接口约束:

无。

- **参数**
  - **neid** (*str*) - 设备的ID。
  - **clicommand** (*str*) - 向设备下发cli命令。
  - **timeout** (*int,optional*) - 超时时间。默认值为60秒。
- **返回**
  - 设备返回的命令。
- **返回类型**
  - str

实际案例

描述样例:

```
>>> from aoc.sys import cliproxy
>>> neid = "8a394835-cb84-38f3-44d5-36a7f2074a77"
>>> response_data = cliproxy.sendCli(neid,"display this",120)
>>> return response_data
{"# sysname CE6851_125.6.13.8 # undo ftp server source all-interfac ..."}
```

aoc.sys.cliproxy.send_cli(neid, clicommand, aoccontext=None, timeout=60)

通过cli从设备获取设备配置数据或运行数据。

经典场景:

服务通过Python侧的cli查询设备配置或运行数据。

功能描述:

1. 业务通过python侧的接口，通过cli从设备获取设备配置数据或运行数据。
2. 具体处理如下:
   a. 服务设置sendCli的入参包括neid、cli命令和超时时间。
   b. Java调用CLI协议向设备下发cli命令。
   c. 设备通过Python代理将命令行查询结果返回给服务。

接口约束:

无。

- **参数**
  - **neid** (*str*) - 设备的ID。
  - **clicommand** (*str*) - 向设备下发cli命令。
  - **timeout** (*int,optional*) - 超时时间。默认值为60秒。
  - **aoccontext** (*obj,optional*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
- **返回**
  - 设备返回的命令。
- **返回类型**
  - str

实际案例

描述样例:

```
>>> from aoc.sys import cliproxy
>>> neid = "8a394835-cb84-38f3-44d5-36a7f2074a77"
>>> response_data = cliproxy.send_cli(neid,"display this",aoccontext,120)
>>> return response_data
{"# sysname CE6851_125.6.13.8 # undo ftp server source all-interfac ..."}
```

10.1.3 aoc.sys.dataconsistency module

典型场景:

1. 设备同步操作。
2. 业务同步操作。

功能描述:

1. 获取设备的checksync状态。
2. 同步接口。
3. 指定指定路径的同步接口。
4. 触发设备一致性检查。
5. 业务一致性检查。

class aoc.sys.dataconsistency.CheckMode(value)

基类： enum.Enum

标记是否进行深度校验。

典型场景:

用于服务差异发现操作。

功能描述:

标记是否进行深度校验。

接口约束:

无。

dbCheck *= 1*

检查期望的南向网元配置与控制器保存的南向网元运行配置是否不一致。

deepCheck *= 2*

检查期望的南向网元配置是否与南向网元运行配置不一致。

aoc.sys.dataconsistency.device_check_sync(device_ids)

获取设备的checksync状态。

典型场景:

在下发配置前，请使用checksync命令检查控制器数据与设备数据是否一致。 如果不一致，请在下发配置前消除不一致。

功能描述:

配置下发前可以校验设备的checksync状态。

接口约束:

设备上线。

- **参数**
  - **device_ids** (*list*) - 设备ID列表。
- **返回**
  - checksync结果。
- **返回类型**
  - 定义在 proto.aoc.sys.dataconsistency.DeviceCheckSyncOutput.DeviceCheckSyncStatus
- **引发**
  - **InvalidArguments** - 输入不合法。
  - **AOCException** - 设备checksync操作失败。

实际案例

```
>>> device_array = []
>>> device_array.append("4796ace2-1cb4-11eb-b471-ae8e72fade46")
>>> response = dataconsistency.device_check_sync(device_array)
>>> print(response)
{'4796ace2-1cb4-11eb-b471-ae8e72fade46': true}
```

aoc.sys.dataconsistency.device_check_sync_fully(device_ids, time_out_value=300)

检查控制器保存的设备配置是否与南向设备一致。

典型场景:

触发设备一致性检查功能，检查控制器保存的设备配置是否与南向设备一致。

功能描述:

触发设备一致性检查功能，检查控制器保存的设备配置是否与南向设备一致。

接口约束:

设备上线。

- **参数**
  - **device_ids** (*list*) - 设备ID列表。
  - **time_out_value** (*int, optional*) - 超时时间。默认值为300秒。
- **返回**
  - 设备一致性状态。
- **返回类型**
  - 定义在 proto.aoc.sys.daataconsistency.GetDeviceSyncStatusOutput
- **引发**
  - **InvalidArguments** - 输入不合法。
  - **TimeoutException** - 设备一致性检查操作超时。
  - **AOCException** - 设备一致性检查操作失败。

实际案例

```
>>> device_array = []
>>> device_array.append("4796ace2-1cb4-11eb-b471-ae8e72fade46")
>>> response =dataconsistency.device_check_sync_fully(device_array)
>>> print(response)
{'4796ace2-1cb4-11eb-b471-ae8e72fade46': {'status':'Discovered','isSync':true}}
```

aoc.sys.dataconsistency.device_sync_from(device_ids, time_out_value=300)

调用将设备数据同步到控制器。

典型场景:

在配置下发前，checksync检测控制器和转发器之间的数据不一致，并调用同步函数消除控制器和转发器之间的数据不一致。

功能描述:

对指定的设备id进行一致性同步操作。

接口约束:

1. 设备上线。
2. 数据一致性同步范围受 **aoc.snd.commonsnd.CommonSND.getFeatures()** 中定义的采集范围影响。

说明

同步是一个高阶操作。转发器配置会修改控制器配置。

- **参数**
  - **device_ids** (*list*) - 设备ID列表。
  - **time_out_value** (*int, optional*) - 超时时间。默认值为300秒。
- **返回**
  - 设备一致性状态。
- **返回类型**
  - 定义在 proto.aoc.sys.dataconsistency.GetDeviceSyncStatusOutput**引发**

- **InvalidArguments** - 输入不合法。
- **TimeoutException** - 设备同步操作超时。
- **AOCException** - 设备同步操作失败。

实际案例

```
>>> device_array = []
>>> device_array.append("4c785c7c-012b-11eb-8a2d-fa163ebe2e21")
>>> response = dataconsistency.device_sync_from(device_array)
>>> print(response)
{'4796ace2-1cb4-11eb-b471-ae8e72fade46': {'status':'Reverted','isSync':true}}
```

aoc.sys.dataconsistency.device_sync_from_bypath(details, time_out_value=300)

支持指定路径的同步。

典型场景:

当需要同步指定路径时使用。

功能描述:

支持指定路径从设备到控制器的数据同步。

接口约束:

输入路径必须在 :func:aoc.snd.commonsnd.CommonSND.getFeatures定义的采集范围中。

警告

说明

同步是一个高阶操作。转发器配置会修改控制器配置。

- **参数**
  - **details** (*dict*) - 设备和路径列表。 key(str): 设备ID； value(list): 同步特性路径列表。
  - **time_out_value** (*int, optional*) - 超时时间。默认值为300秒。
- **返回**
  - 设备一致性状态。
- **返回类型**
  - 定义在 proto.aoc.sys.dataconsistency.GetDeviceSyncStatusOutput
- **引发**
  - **InvalidArguments** - 输入不合法。
  - **TimeoutException** - 设备同步操作超时。
  - **AOCException** - 设备同步操作失败。

实际案例

```
>>> partialInfo = {}
>>> paths = []
>>> paths.append("/huawei-l3vpn:l3vpn/l3vpncomm/l3vpnInstances");
```



```
>>> partialInfo['e61608da-1b41-11eb-a064-8ea4b0f114c4'] = paths
>>> response = dataconsistency.device_sync_from_bypath(partialInfo)
>>> print(response)
{'4796ace2-1cb4-11eb-b471-ae8e72fade46': {'status':'Reverted','isSync':true}}
```

aoc.sys.dataconsistency.service_check_sync(service_paths, isdeep, time_out_value=300)

检查业务期望的南向网元配置是否一致。

典型场景:

在验证阶段，如果下发的配置冲突，可以使用该接口检查差异，如预期南向网元配置与 控制器上南向网元运行配置的差异或预期南向网元配置与实际运行南向网元配置的差异所示。

功能描述:

检查业务期望的南向网元配置是否与控制器上南向网元的运行配置或南向网元的实际运行 配置不一致。

接口约束:

None.

- **参数**
  - **service_paths** (*list*) - 待检查的业务路径。
  - **isdeep** (*enum*) - 是否进行深度验证。 详情请参考 aoc.sys.dataconsistency.CheckMode
  - **time_out_value** (*int, optional*) - 超时时间。默认值为300秒。
- **返回**
  - 服务一致性检查结果。
- **返回类型**
  - 定义在 proto.aoc.sys.daataconsistency.GetDeviceSyncStatusOutput
- **引发**
  - **InvalidArguments** - 输入不合法。
  - **TimeoutException** - 业务一致性检查操作超时。
  - **AOCException** - 业务一致性检查操作失败。

实际案例

```
>>> servicepaths = []
>>> servicepaths.append("/huawei-ac-applications:applications/aaamini:aaamini/test123")
>>> response =
dataconsistency.service_check_sync(servicepaths,dataconsistency.CheckMode.deepCheck)
>>> print(response)
{'/huawei-ac-applications:applications/aaamini:aaamini/test123':
{'status':'SUCCESS_WITHOUT_DIFF', 'isSync':true}}
```

10.1.4 aoc.sys.datastore module

典型场景:

此模块提供了操作数据存储数据的方法。

功能描述:

读写数据存储。

aoc.sys.datastore.sdk

从open api向aoc发送请求的sdk。

Type: object

aoc.sys.datastore.logger

当前包组的日志打印。

Type: Log

aoc.sys.datastore.ne_id_pattern

用于匹配设备ID的正则表达式。

Type: str

aoc.sys.datastore.read_datastore_cdb(aoccontext, path, query_para=None, dst='ecs')

通过事务ID和YANG子树路径读取Datastore中CDB中的数据。

典型场景:

恢复业务配置时，查询此事务中已编辑的业务数据。

功能描述:

使用全局事务ID，查询RDB中的CDB数据。

接口约束:

path 是必填字段。

- **参数**
  - **aoccontext** (*AocContext*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **path** (*str*) - YANG路径，必填字段。
  - **query_para** (*dict, optional*) - 查询条件。
  - **dst** (*str*) - 目的节点。
- **返回**
  - 数据存储的cdb数据。
- **返回类型**
  - 它由proto文件定义，路径为 proto.aoc.sys.datastore.ReadResult

实际案例

```
>>> form aoc.sys import datastore
>>> print datastore.read_datastore_cdb(aoccontext, '/huawei-ac-nes:inventory-cfg/nes/ne/
8d394835-cb84-38f3-a4d5-36a7f2074b50/huawei-aaa:aaa/lam/users/user')
```

```
huawei
>>> print datastore.read_datastore_cdb(aoccontext, '/huawei-ac-applications:applications/
aaamini:aaamini')
huawei aoc
>>> para = {'offset': '5', 'limit': '5', 'depth': '2'}
>>> print(datastore.read_datastore_cdb(aoccontext, '/huawei-ac-applications:applications/
aaamini:aaamini', para))
<users xmlns="http://www.huawei.com/netconf/vrp/huawei-aaa">
  <user>
    <userName>root</userName>
  </user>
</users>
```

aoc.sys.datastore.read_datastore_rdb(aoccontext, path, query_para=None, dst='ecs')

通过YANG子树路径读取Datastore中RDB中的数据。与读取网元的RDB的区别在于此接口未指定网元。

典型场景:

在配置下发前，checksync检测控制器和转发器之间的数据不一致，并调用同步函数消除控制器和转发器之间的数据不一致。

功能描述:

根据YANG子树路径查询CDB中的数据。

接口约束:

path 是必填字段。

- **参数**
  - **aoccontext** - (AocContext): 上下文信息，详细信息请参考 **aoc.base.aoccontext**.
  - **path** (*str*) - YANG路径，必填字段。
  - **query_para** (*dict*) - 查询条件。
  - **dst** (*str*) - 目的节点。
- **返回**
  - 数据存储的rdb数据。
- **返回类型**
  - 它由proto文件定义，路径为 proto.aoc.sys.datastore.ReadResult

实际案例

```
>>> form aoc.sys import datastore
>>> print datastore.read_datastore_rdb(aoccontext, '/huawei-ac-nes:inventory-cfg/nes/ne/
8d394835-cb84-38f3-a4d5-36a7f2074b50/huawei-aaa:aaa/lam/users/user')
huawei
>>> print datastore.read_datastore_rdb(aoccontext, '/huawei-ac-applications:applications/
aaamini:aaamini')
huawei aoc
>>> para = {'offset': '5', 'limit': '5', 'depth': '2'}
>>> print(datastore.read_datastore_cdb(aoccontext, '/huawei-ac-applications:applications/
aaamini:aaamini', para))
<users xmlns="http://www.huawei.com/netconf/vrp/huawei-aaa">
  <user>
```



```
    <userName>root</userName>
  </user>
</users>
```

aoc.sys.datastore.write_datastore(aoc_context, content, path, source, dst='ecs')

通过事务ID和YANG子树路径将数据写入Datastore的CDB。

典型场景:

恢复业务数据时，将编辑后的业务数据写入CDB。

功能描述:

使用全局事务ID，将数据写入Datastore的CDB。

- **参数**
  - **aoccontext** - (AocContext): 上下文信息，详细信息请参考 **aoc.base.aoccontext**.
  - **content** (*str*) - 要写入数据存储区的内容。
  - **path** (*str*) - YANG模型子树的路径。
  - **source** (*str*) - 数据源。
  - **dst** (*str*) - 目的节点。

接口约束:

1. aoc_context中的事务ID为必填字段。 #.内容和路径是必填字段。
2. path 是必填字段。

- **引发**
  - **AOCException** - 设备离线。

实际案例

```
>>> from aoc.sys import transaction
>>> from aoc.sys import datastore
>>> ne_id = '8d394835-cb84-38f3-a4d5-16a7f2074b40'
>>> trans_id = transaction.create_transaction().transId
>>> aoc_context.transactionId = trans_id
>>> 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(aoc_context, content, path, '')
>>> self.logger.info('write_datastore response=%s' % response)
>>> transaction.commit_transaction(trans_id)
```

10.1.5 aoc.sys.datastore_migrate module

典型场景:

snd包不兼容升级的场景，如果您需要将datastore的数据从旧包所在的域迁移到新包所在的域中， 您可以使用此API提供的接口来进行datastore数据迁移。

功能描述:

该模块提供datastore数据迁移所需的操作接口。

aoc.sys.datastore_migrate.read_datastore_sources(aoccontext, node_path, query_param)

典型场景:

此API用于SND包不兼容升级场景，可以调用它查找节点的数据源信息。

功能描述:

查询节点数据源。

接口约束:

此接口用于snd包不兼容升级场景。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **node_path** (*str*) - yang文件中定义的节点对应的xpath路径。例：/module1:container1/container2/list1/container3。
  - **query_param** (*dict*) - 扩展参数，包含: ne_id(str): 设备ID。 old_domain_id(str): 旧的snd包所在的域。
- **返回**
  - 节点的数据源集合，例如: [‘source1’,’source2’]。
- **返回类型**
  - list
- **引发**
  - **MigrateException** - 设备ID不存在或者旧snd包的domain不存在或者路径不合法。

示例

```
>>> path = '/huawei-l3vpn:l3vpn'
>>> query_param = {'ne_id': '8d394835-cb84-38f3-a4d5-36a7f2074b50',
'old_domain_id': 'migrate_old'}
>>> print(read_datastore_sources(aoccontext, path, query_param))
['source1','source2']
```

aoc.sys.datastore_migrate.read_migrate_node_data(aoccontext, node_path, query_param, source)

典型场景:

此API用于SND包不兼容升级场景，可以调用它查找节点某个数据源下的数据。

功能描述:

查询某个数据源引用的节点数据。

接口约束:

此接口用于snd包不兼容升级场景。

- **参数**

  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **node_path** (*str*) - yang文件中定义的节点对应的xpath路径。例：/module1:container1/container2/list1/container3。
  - **query_param** (*dict*) - 扩展参数，包含: ne_id(str): 设备ID。 old_domain_id(str): 旧的snd包所在的域。
  - **source** (*str*) - 当前节点的数据源。
- **返回**
  - key是节点路径，value是节点信息。
- **返回类型**
  - dict
- **引发**
  - **MigrateException** - 设备ID不存在或者旧snd包的domain不存在或者路径不合法。

示例

```
>>> path = '/huawei-l3vpn:l3vpn'
>>> query_param = {'ne_id': '8d394835-cb84-38f3-a4d5-36a7f2074b50','old_domain_id':
'migrate_old'}
>>> source = 'source1'
>>> print(read_migrate_node_data(aoccontext, path, query_param, source))
{'/huawei-l3vpn:l3vpn': '{"l3vpn":{"l3vpnName":"aaa"}'}
```

aoc.sys.datastore_migrate.write_migrate_node_data(aoccontext, write_param, source, node_new_data)

典型场景:

此API用于SND包不兼容升级场景，可以调用它带源写入新的节点数据到新的snd包所在的域中。

功能描述:

带源写入新的节点数据到新的snd包所在的域中。

接口约束:

此接口用于snd包不兼容升级场景。

- **参数**
  - **aoccontext** (*aoccontext*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **write_param** (*dict*) - 扩展参数，包含: ne_id(str): 设备ID。 new_domain_id(str): 新的snd包所在的域。 tx_id(str): 事务ID。
  - **source** (*str*) - 节点的数据源。
  - **node_new_data** (*dict*) - 新的数据。
- **返回**
  - 如果成功则返回”Migrate datastore success” ，否则会抛MigrateException异常。
- **返回类型**
  - str

- **引发**
  - **MigrateException** - 事务ID非法，或者设备ID或新的域不存在，或者新的数据不合法。

示例

```
>>> write_param = {'ne_id': '8d394835-cb84-38f3-a4d5-36a7f2074b50',
'new_domain_id': 'migrate_new','tx_id': '2222'}
>>> source = 'source1'
>>> new_data = {'/huawei-l3vpn:l3vpn': '{"l3vpn":{"l3vpnName":"aaa"}'}
>>> print(write_migrate_node_data(aoccontext, query_param, source, new_data))
Migrate datastore success
```

10.1.6 aoc.sys.devicemgr module

典型场景:

当开发SSP包时，需要通过条件获取设备基础信息。

功能描述:

该模块提供查询设备信息的接口,主要方法: query_basic_data_from_db, query_neid, query_status, query_data_from_device, device_rpc, is_snd, get_snd_id.

aoc.sys.devicemgr.QUERY_DEVICE_KEY

查询设备信息的方法名称。内部变量，不暴露给用户。

Type: str

aoc.sys.devicemgr.QUERY_NE_MANAGERS_KEY

查询neManagers信息的方法名。内部变量，不公开给用户。

Type: str

aoc.sys.devicemgr.DEVICE_INFO_TABLE

Redis设备信息表名称。内部变量，不公开给用户。

Type: str

aoc.sys.devicemgr.GET

Redis读操作代码。内部变量，不公开给用户。

Type: str

aoc.sys.devicemgr.SMEMBERS

Redis读操作代码。内部变量，不公开给用户。

Type: str

aoc.sys.devicemgr.EXPIRE_TEN_SECONDS

Redis过期时间。内部变量，不公开给用户。

Type: int

aoc.sys.devicemgr.sdk

外部注入基准。

Type: obj

aoc.sys.devicemgr.logger

日志参考。

Type: obj

aoc.sys.devicemgr.device_rpc(neid, rpcdata, path, timeout=30000, aoccontext=None)

根据yang模型生成南向报文，等待返回结果。

典型场景:

设备上的rpc操作，这里的rpc是指设备上实现的基于yang的rpc能力，可以通过netconf下发rpc报文。

功能描述:

1. device_rpc接口实现实时rpc功能。
2. 具体处理如下:
   a. 根据入参neid和入参path指定的yang模型生成南向报文。
   b. 下发南向报文，阻塞报文，等待返回结果。

接口约束:

构建与设备yang模型一致的路径。

- **参数**
  - **neid** (*str*) - 设备的ID。
  - **rpcdata** (*str*) - YANG模型中定义的RPC对应XML数据。
  - **path** (*str*) - 要从设备读取的数据对应的yang subtree路径。
  - **timeout** (*int, optional*) - 超时时间。默认值为30000毫秒。
- **返回**
  - 设备返回的Netconf报文。
- **返回类型**
  - str

实际案例

描述样例

```
>>> from aoc import devicemgr
>>> neid = '868e778e-153c-3afe-a02c-89678e31e3e4'
>>> path = '/huawei-ac-ne-snmp:snmp'
>>> rpcdata = '''<activate-software-image xmlns="urn:huawei:yang:huawei-aoc-rpc-test">''' +
'''<image-name>acmefw-2.3</image-name></activate-software-image>'''
```



```
>>> output = result = devicemgr.device_rpc(neid, rpcdata, path)
>>> print output
<activate-software-image xmlns="urn:huawei:yang:huawei-aoc-rpc-test">
<status>The image acmefw-2.3 is being installed.</status>
</activate-software-image>
```

aoc.sys.devicemgr.get_snd_id(neid)

根据输入参数neid查询设备对应的snd包的id。

典型场景:

查询设备对应的snd包id。

功能描述:

get_snd_id接口用于查询设备对应的snd报文的id.。 处理如下: 根据入参neid查询设备对应的snd包id

接口约束:

无。

- **参数**
  - **neid** (*str*) - 设备的ID。
- **返回**
  - snd包的ID。
- **返回类型**
  - str

实际案例

描述样例

```
>>> from aoc import devicemgr
>>> neid = '868e778e-153c-3afe-a02c-89678e31e3e4'
>>> sndid = get_snd_id(neid)
>>> print sndid
"123"
```

aoc.sys.devicemgr.is_snd(neid, sndid)

根据入参neid和sndid检查设备是否绑定了snd包。

典型场景:

检查设备是否绑定了snd包。

功能描述:

is_snd接口用于查询设备是否绑定了snd报文。 处理如下: 根据入参neid和sndid检查设备是否绑定了snd包。

接口描述:

无。

- **参数**

  - **neid** (*str*) - 设备的ID。
  - **sndid** (*str*) - snd包的ID。
- **返回**
  - 如果ID为neid的设备由ID为sndid的插件包管理，则为True，否则为False。
- **返回类型**
  - bool

实际案例

描述样例

```
>>> from aoc import devicemgr
>>> neid = '868e778e-153c-3afe-a02c-89678e31e3e4'
>>> sndid = '76365e38-9d57-3ecb-0ea0-e924d784ed39'
>>> isSnd = is_snd(neid, sndid)
>>> print output
True
```

aoc.sys.devicemgr.query_basic_data_from_db(neid='', nename='', nemanageip='')

查询设备基本信息。

典型场景:

当开发SSP包时，你需要根据设备ID，设备名称，设备IP地址来查询实时设备基础信息

功能描述:

根据设备ID（neid），设备名称（nename），设备IP地址（nemanageip）查询实时设备基础信息

接口约束:

三个参数均为可选，但需要保证至少填入一个参数。 满足查询条件的设备基础信息将被展示。

- **参数**
  - **neid** (*str*) - 设备 ID.
  - **nename** (*str*) - 设备的操作名称。
  - **nemanageip** (*str*) - 设备的管理IP地址。
- **返回**
  - 设备基本信息对象。
- **返回类型**
  - 由proto文件定义，路径为 proto.aoc.sys.querydevicesoutput.QueryDevicesPagedResult
- **引发**
  - **InnerException** - 设备不存在。
  - **InvalidArguments** - 输入参数错误。

实际案例

```
>>> from aoc import devicemgr
>>> neid = '868e778e-153c-3afe-a02c-89678e31e3e4'
```



```
>>> nename = 'HUAWEI_ROUTER'
>>> nemanageip = '{ip:port}'
>>> print devicemgr.query_basic_data_from_db(neid, nename, nemanageip)
neId: "868e778e-153c-3afe-a02c-89678e31e3e4",
hardWare: "CX600-X1-M4",
softWare: "V800R012C10SPC680B680",
manageIp: "192.168.3.11",
type: "ROUTER",
manufacturer: "HUAWEI",
manageMac: "38:BA:16:58:1E:03",
deviceModel: "CX600-X1-M4",
esn: "391091484237311",
location: "Beijing China",
get_channel_index: "b9b51c1d-38cd-35ed-9acd-2cb477762707",
edit_channel_index: "a377dae5-6f88-3b15-a04d-56b9f769fdbb"
```

aoc.sys.devicemgr.query_data_from_device(neid, storetype, path, properties=None, timeout=30000)

实时查询设备数据。

典型场景:

当用户编译ssp、gnd或snd包时，需要实时从设备获取一些数据。

功能描述:

1. 接口实现实时查询设备数据的功能。
2. 处理如下:
   a. 根据入参neid和入参path指定的YANG模型生成南向报文。
   b. 下发南向报文，等待设备返回的结果。

接口约束:

构建与设备yang模型一致的路径。

- **参数**
  - **neid** (*str*) - 设备的ID。
  - **storetype** (*enum*) - action: 用于查询状态数据，配置netconf get-config报文。'CONFIGURATION'或者''
  - **path** (*str*) - 要从设备读取的数据对应的yang subtree路径。
  - **properties** (*dict, optional*) - 附加字段，支持subtree过滤。
  - **timeout** (*int, optional*) - 超时时间。默认值为30000毫秒。
- **返回**
  - 设备返回的Netconf报文。
- **返回类型**
  - str

实际案例

描述样例

```
>>> from aoc import devicemgr
>>> neid = '868e778e-153c-3afe-a02c-89678e31e3e4'
>>> path = '/huawei-ac-ne-snmp:snmp'
```



```
>>> output = devicemgr.query_data_from_device(neid, 'CONFIGURATION', path)
>>> print output
<snmp xmlns="urn:huawei:yang:huawei-ac-ne-snmp">
   <mibViews>
      <mibView>
         <viewName>118</viewName>
         <subtree>iso</subtree>
         <type>excluded</type>
      </mibView>
   </mibViews>
</snmp>
```

aoc.sys.devicemgr.query_data_from_device_nes(params, properties, timeout=30000, aoccontext=None)

实时查询设备数据。

典型场景:

查询设备配置数据时，不需要在接口的入参中设置南向协议信息。

返回值根据设备阳模型显示设备配置数据。

功能描述:

1. 接口实现实时查询设备数据的功能。
2. 处理如下:
   a. 根据入参neid和入参path指定的YANG模型生成南向报文。
   b. 下发南向报文，等待设备返回的结果。

接口约束:

构建与设备yang模型一致的路径。

- **参数**
  - **params** (*map<string, string>*) - neid: AOC中设备的ID， storetype: 值可以是 ‘OPERATIONAL’ 或 ‘CONFIGURATION’， path: yang 路径。
  - **properties** (*map<string, string>*) - 默认值为 {}.
  - **timeout** (*int, optional*) - 超时时间。默认值为30000毫秒。
  - **aoccontext** (*obj, optional*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
- **返回**
  - 设备返回的Netconf报文。
- **返回类型**
  - str

实际案例

描述样例

```
>>> from aoc import devicemgr
>>> params = {}
>>> params.neid = '868e778e-153c-3afe-a02c-89678e31e3e4'
>>> params.path = '/huawei-ac-ne-snmp:snmp'
>>> output = devicemgr.query_data_from_device(params, properties, 30000, aoccontext)
>>> print output
```



```
<snmp xmlns="urn:huawei:yang:huawei-ac-ne-snmp">
  <mibViews>
    <mibView>
      <viewName>118</viewName>
      <subtree>iso</subtree>
      <type>excluded</type>
    </mibView>
  </mibViews>
</snmp>
```

aoc.sys.devicemgr.query_neid(nename)

根据操作名称查询设备ID。

典型场景:

当开发SSP包时，需要根据设备名称查询对应设备ID

功能描述:

1. 该接口用于根据设备名称查询设备id
2. 流程如下：
   a. 根据入参operateName查询设备id
   b. 改接口支持缓存加速以提高查询效率

说明

不建议使用该接口。推荐使用 query_neids_by_operatename.

- **参数**
  - **nename** (*str*) - 设备操作名称 (nename)
- **返回**
  - 设备 id.
- **返回类型**
  - str
- **引发**
  - **InvalidArguments** - 输入参数nename无效。
  - **InnerException** - 设备不存在或根据nename找到多个neid。

实际案例

```
>>> from aoc import devicemgr
>>> print devicemgr.query_neid('HUAWEI_ROUTER')
868e778e-153c-3afe-a02c-89678e31e3e4
```

aoc.sys.devicemgr.query_neids_by_operatename(operatename)

根据操作名称查询设备ID。

典型场景:

当开发SSP包时，需要根据设备id来查询设备名称

功能描述:

1. 该接口用于通过设备名称查询设备id
2. 流程如下：
   a. 根据设备id来查询设备名称
   b. 改接口支持缓存加速以提高查询效率

- **参数**
  - **operatename** (*str*) - 设备操作名称。
- **返回**
  - 设备ID列表。
- **返回类型**
  - list
- **引发**
  - **InvalidArguments** - 入参操作名称无效。
  - **InnerException** - 设备不存在。

实际案例

```
>>> from aoc import devicemgr
>>> print query_neids_by_operatename('HUAWEI_ROUTER')
["868e778e-153c-3afe-a02c-89678e31e3e4", "868e778e-153c-3afe-a02c-89678e31e3e5"]
```

aoc.sys.devicemgr.query_status(neid='', nename='', nemanageip='')

查询设备连接状态。

典型场景:

当开发SSP包时，需要根据设备id，设备名称，设备ip地址来查询设备实时状态。

功能描述:

根据设备id（ neid ），设备名称（ nename ），设备ip地址（ nemanageip ）来查询设备连接状态。

说明

不建议使用该接口。推荐使用 query_status_by_neid.

接口约束:

三个参数均为可选，但需要保证至少填入一个参数。 满足查询条件的设备基础信息将被展示。

- **参数**
  - **neid** (*str*) - 设备 ID.
  - **nename** (*str*) - 设备的操作名称。
  - **nemanageip** (*str*) - 设备的管理IP地址。
- **返回**
  - 连接状态。
- **返回类型**
  - str

- **引发**
  - **InnerException** - 设备不存在。
  - **InvalidArguments** - 输入参数错误。

实际案例

```
>>> from aoc import devicemgr
>>> print query_status('HUAWEI_ROUTER')
OperateUp
```

aoc.sys.devicemgr.query_status_by_neid(neid)

查询设备连接状态。

典型场景:

当开发SSP包时，需要根据设备id来查询设备实时状态。

功能描述:

根据设备id（neid）来查询设备连接状态。

- **参数**
  - **neid** (*str*) - 设备 ID.
- **返回**
  - 连接状态
- **返回类型**
  - str
- **引发**
  - **InnerException** - 设备不存在。
  - **InvalidArguments** - 输入参数错误。

实际案例

```
>>> from aoc import devicemgr
>>> print query_status_by_neid('868e778e-153c-3afe-a02c-89678e31e3e4')
OperateUp
```

10.1.7 aoc.sys.devicenotify module

典型场景:

提供设备信息变更通知Python侧的接口。

功能描述:

classes: notify

class aoc.sys.devicenotify.DeviceNotify(logger=None, resource_dir='')

基类：aoc.base.baseservice.BaseService

典型场景:

该模块为敏捷控制器提供设备信息变更通知Python侧的接口。

功能描述:

classes: notify

abstract **notify**(aoccontext**,** device_event)

接收设备信息变更通知并执行业务。

典型场景:

在编译代码时，实现该方法，检测设备信息变化，处理业务。

功能描述:

接收设备信息变更通知并执行业务。

接口约束:

对于耗时的业务处理逻辑，需要启动新的线程。

- **参数**
  - **aoccontext** - 上下文信息，有关详细信息，请参见 **aoc.base.aoccontext**.
  - **device_event** - 设备信息变更通知的正文，具体操作请参见 proto.aoc.sys.deviceevent.DeviceEventNode.

实际案例

以下是接收设备上线通知，然后收集设备侧接口信息的示例:

```
def notify(self, aoccontext, device_event):
    deviceEventNode = input.deviceEventNode
    # When the device is online, trigger port collection
    if deviceEventNode.eventType == 'ONLINE':
        neid = '868e778e-153c-3afe-a02c-89678e31e3e4'
        path = 'ietf-interfaces:interfaces-state'
        response = devicemgr.query_data_from_device(neid,'CONFIGURATION', path)
```

10.1.8 aoc.sys.event_mgr module

典型场景:

本模块提供发送告警事件的方法。

功能描述:

提供事件管理能力。

class aoc.sys.event_mgr.EventMgr

基类: object

提供事件管理能力。

Typical Scenarios:

ssp包定制处理snd包中的数据变化。

功能描述:

提供事件管理能力。

接口约束:

无。

***static* send_aoc_event**(event, aoccontext=None)

向控制器的网元层发送事件。

典型场景:

两个插件包之间的事件传输，联动处理。

功能描述:

向控制器发送事件。

接口约束:

无。

- **参数**
  - **event** (*AocEventEntity*) - 事件信息，详细信息请参考 proto.aoc.event.AocEventHandler.AocEventEntity。
  - **aoccontext** (*AocContext*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。

实际案例

```
def notify(self, aoc_context, event_handler):
    entity_list = event_handler.eventEntity
    if entity_list:
        for entity in entity_list:
            EventMgr.send_aoc_event(entity, aoc_context)
```

***static* send_aoc_ncs_event**(event)

向控制器的网络层发送事件。

典型场景:

两个插件包之间的事件传输，联动处理。

功能描述:

向控制器发送事件。

接口约束:

无。

- **参数**
  - **event** (*AocEventEntity*) - 事件信息，详细信息请参考 proto.aoc.event.AocEventHandler.AocEventEntity。

实际案例

```
def notify(self, aoc_context, event_handler):
    entity_list = event_handler.eventEntity
    if entity_list:
        for entity in entity_list:
            EventMgr.send_aoc_ncs_event(entity)
```


10.1.9 aoc.sys.lock_mgr module

典型场景:

在制作映射逻辑的过程中，用户可以在处理时调用此函数锁定路径，以避免其他服务同时更改此路径。

功能描述:

该模块为用户提供锁，避免路径被其他服务更改。

aoc.sys.lock_mgr.partial_lock(aoc_context, path, timeout)

用户锁定路径以避免路径被其他业务更改。

典型场景:

用户锁定路径以避免路径被其他业务更改。现在可以用于SSP、GND、SND。

功能描述:

用户锁定路径以避免路径被其他业务更改。

接口约束:

无。

- **参数**
  - **path** (*str*) - 需要上锁的路径
  - **aoc_context** (*dict*) - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
  - **timeout** (*int*) - 如果路径被其他人锁定，则此函数将等待。如果到达此timeout（单位为秒），则函数将失败。
- **引发**
  - **LockException** - timeout（单位为秒）超时时间到达。
  - **IllegalParameterException** - aoc_context中输入的事务Id为空或aoc_context中输入的事务Id未退出。

实际案例

举例：锁定”/huawei-ac-applications:applications/aaamini:aaamini/test”，如下:

```
try:
lock_mgr.partial_lock(aoccontext,
'/huawei-ac-applications:applications/aaamini:aaamini/test', 60)
except LockException as ex:
self.logger.exception('lock exception. %s' % ex)
except IllegalParameterException as ex:
self.logger.exception('lock IllegalParameterException. %s' % ex)
```

10.1.10 aoc.sys.ne_resmgr_service module

典型场景:

1. 在配置下发过程中，用户可以创建资源池，并申请资源，如VLAN资源。
2. 网元层资源管理API

功能描述:

资源管理API用于创建、申请资源，如VLAN资源。配置下发后，创建资源池，申请资源。

class aoc.sys.ne_resmgr_service.NeResMgrService(logger)

基类：object

资源管理API用于创建、申请资源。

典型场景:

1. 在配置下发过程中，用户可以创建资源池，并申请资源，如VLAN资源。
2. 网络层资源管理API

功能描述:

资源管理API用于创建、申请资源，如VLAN资源。配置下发后，创建资源池，申请资源。

接口约束:

该api只能被gnd业务使用。

- **参数**
  - **logger**（*obj*）- 日志打印。

logger

当前对象的日志打印。

Type：obj

service_name

服务名称。

Type：str

serviceinject

服务注入对象。

Type：obj

allocate(aoc_context, allocate_info, key=None, belong_service=None)

在创建的资源池中申请资源。

典型场景:

配置下发时，需要到资源池中去申请对应的资源。

功能描述:

在创建的资源池中申请资源。

接口约束:

资源池已存在。

- **参数**
  - **aoccontext**（*obj*）- 上下文信息，详细信息请参考 **aoc.base.aoccontext**。

  - **key** (*str, optional*) - 如果属于服务实例，则为服务key。
  - **allocate_info** (*dict*) - 分配资源所需参数的json。
  - **belong_service** (*bool, optional*) - 服务实例是否属于。
- **返回**
  - 已申请的资源。
- **返回类型**
  - Resources object
- **引发**
  - **AOCException** - 资源池不存在或资源池已满或输入参数无效。

示例

描述样例:

```
allocate_info = {}
allocate_info.setdefault("neId", "d3cfc6b8-96ad-11eb-8e29-5a9cdbdc7769")
allocate_info.setdefault("resourceKey", "vlan")
allocate_info.setdefault("resourceType", "vlan")
allocate_info.setdefault(
"transId", "G20210408002017786_81a99bb-acb4-4c6e-b904-8e57c026fadad")
>>> print self.allocate(aoc_context, allocate_info, "vlan", True)
['2']
```

create_pool(aoc_context, create_pool_input)

创建资源池，返回提示

典型场景:

在配置下发之前，您需要先创建资源池。

功能描述:

创建资源池，并返回资源池是否创建成功。

接口约束:

增加不允许重复创建的限制。

- **参数**
  - **aoccontext** (*obj*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **create_pool_input** (*dict*) - 创建池的json必需参数。
- **引发**
  - **AOCException** - 资源池已存在或输入参数无效或创建资源池失败。
- **返回**
  - 如果成功创建池，则为True，否则为False。
- **返回类型**
  - bool

示例

描述样例:

```
create_pool_input = {}
create_pool_input.setdefault("neId", "d3cfc6b8-96ad-11eb-8e29-5a9cdbdc7769")
```



```
create_pool_input.setdefault("resourceKey", "vlan")
create_pool_input.setdefault("resourceType", "vlan")
create_pool_input.setdefault("xpath", "/huawei-vlan:vlan/vlans/vlan")
create_pool_input.setdefault("connector", aoc_context.transactionId)
create_pool_input.setdefault("resMin", "1")
create_pool_input.setdefault("resMax", "20")
create_pool_input.setdefault("serviceKey", "vlan")
>>> print self.create_pool(aoc_context, create_pool_input)
True
```

10.1.11 aoc.sys.resourcemgr_service module

典型场景:

1. 在配置下发过程中，用户可以创建资源池。申请、占用、释放资源，销毁资源池，如VLAN资源。
2. 网络层资源管理api

功能描述:

1. 资源管理API用于创建、申请、删除和查询下发资源，如vlan资源。
2. 配置下发过程中，查询资源池、创建资源池、申请资源、占用资源。

class aoc.sys.resourcemgr_service.ResourceMgrService(logger)

基类：object

典型场景:

1. 在配置下发过程中，用户可以创建资源池。申请、占用、释放资源，销毁资源池，如VLAN资源。
2. 网络层资源管理api

功能描述:

1. 资源管理API用于创建、申请、删除和查询下发资源，如vlan资源。
2. 配置下发过程中，查询资源池、创建资源池、申请资源、占用资源。

接口约束:

该api只能由ssp业务使用。

- **参数**
  - **logger**（*object*）- 日志打印。

logger

当前对象的日志打印。

Type：obj

service_name

服务名称。

Type：str

serviceinject

服务注入对象。

Type：obj

allocate(aoc_context, allocate_info, key=None, belong_service=None)

在创建的资源池中申请资源。

典型场景:

配置下发时，需要到资源池中去申请对应的资源。

功能描述:

在创建的资源池中申请资源。

接口约束:

资源池已存在。

- **参数**
  - **aoc_context** - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **key** (*str, optional*) - 如果属于服务实例，则为服务key。
  - **allocate_info** (*dict*) - 分配资源所需参数的json。
  - **belong_service** (*bool, optional*) - 是否属于服务实例。
- **返回**
  - 已申请的资源。
- **返回类型**
  - Resources object
- **引发**
  - **AOCException** - 资源池不存在或资源池已满或输入参数无效。

示例

描述样例:

```
>>> allocate_info = {}
>>> allocate_info.setdefault("url", "http://aoc.serviceinject.test")
>>> allocate_info.setdefault("begin", 1)
>>> allocate_info.setdefault("end", 4096)
>>> allocate_info.setdefault("consumer", "testVlan")
>>> allocate_info.setdefault("gid", aoc_context.transactionId)
>>> resource_service = ResoueceMgrService(self.logger)
>>> print resource_service.allocate(aoc_context, "vlan", allocate_info, True)
['2']
```

create_pool(aoc_context, create_pool_input)

创建资源池，返回提示

典型场景:

在配置下发之前，您需要先创建资源池。

功能描述:

创建资源池，并返回资源池是否创建成功。

接口约束:

增加不允许重复创建的限制。

- **参数**
  - **aoc_context** - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **create_pool_input** (*dict*) - 创建池必需的json参数。
- **引发**
  - **AOCException** - 资源池已存在或输入参数无效或创建资源池失败。
- **返回**
  - 如果成功创建池，则为True，否则为False。
- **返回类型**
  - bool

示例

描述样例:

```
>>> create_pool_input = {}
>>> create_pool_input.setdefault("url", "http://aoc.serviceinject.test")
>>> create_pool_input.setdefault("begin", 1)
>>> create_pool_input.setdefault("end", 4096)
>>> create_pool_input.setdefault("consumer", "testVlan")
>>> create_pool_input.setdefault("gid", aoc_context.transactionId)
>>> create_pool_input.setdefault("segment", "false")
>>> resource_service = ResoueceMgrService(self.logger)
>>> print resource_service.create_pool(aoc_context, create_pool_input)
True
```

destroy_pool(aoc_context**,** destroy_pool_input)

删除已创建的资源池。

典型场景:

删除配置之后，需要删除已创建的资源池

功能描述:

删除已创建的资源池。

接口约束:

资源池存在。

- **参数**
  - **aoc_context** - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **destroy_pool_input** (*dict*) - 销毁资源池必需的json参数。
  - **gid** (*str, optional*) - 事务Id
- **引发**
  - **AOCException** - 资源池不存在。
- **返回**
  - 如果成功销毁池，则为True，否则为False。
- **返回类型**
  - bool

示例

描述样例:

```
>>> destroy_pool_input = {}
>>> destroy_pool_input.setdefault("url", "http://aoc.serviceinject.test")
>>> destroy_pool_input.setdefault("begin", 1)
>>> destroy_pool_input.setdefault("end", 4096)
>>> destroy_pool_input.setdefault("consumer", "testVlan")
>>> destroy_pool_input.setdefault("gid", aoc_context.transactionId)
>>> resource_service = ResoueceMgrService(self.logger)
>>> tid = aoc_context.transactionId
>>> print resource_service.destroy_pool(aoc_context, destroy_pool_input, tid)
True
```

get_available_resource_ranges(aoc_context**,** get_available_resource_ranges_input)

获取可用资源范围。

典型场景:

当资源池已存在时，获取可用资源的范围。

功能描述:

获取可用资源范围。

接口约束:

资源池存在。

- **参数**
  - **aoc_context** - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **get_available_resource_ranges_input** (*dict*) - 获取可用资源范围必需的json参数。
- **返回**
  - 最大可用资源数。
- **返回类型**
  - int
- **引发**
  - **AOCException** - 资源池不存在或资源池已满。

示例

描述样例:

```
>>> input = {}
>>> input.setdefault("url", "http://aoc.serviceinject.test")
>>> input.setdefault("begin", 1)
>>> input.setdefault("end", 4096)
>>> input.setdefault("consumer", "testVlan")
>>> input.setdefault("gid", aoc_context.transactionId)
>>> resource_service = ResoueceMgrService(self.logger)
>>> result = resource_service.get_available_resource_ranges(input, aoc_context)
>>> print result
13
```

get_pool_info(aoc_context**,** get_pool_info_input)

获取资源池信息。

典型场景:

当资源池已存在时，查询资源池信息。

功能描述:

获取资源池信息。

接口约束:

资源池已存在。

- **参数**
  - **aoc_context** - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **get_pool_info_input** (*dict*) - 查询资源池必需的json参数。
  - **gid** (*str, optional*) - 事务Id.
- **引发**
  - **AOCException** - 资源池不存在。
- **返回**
  - 资源池信息。
- **返回类型**
  - Resources object

示例

描述样例:

```
>>> input = {}
>>> input.setdefault("url", "http://aoc.serviceinject.test")
>>> input.setdefault("begin", 1)
>>> input.setdefault("end", 4096)
>>> input.setdefault("consumer", "testVlan")
>>> input.setdefault("gid", aoc_context.transactionId)
>>> resource_service = ResoueceMgrService(self.logger)
>>> tid = aoc_context.transactionId
>>> result = resource_service.get_pool_info(aoc_context, input, tid)
>>> print result
{'capacity': 4094, 'left': 0, 'used': 0}
```

is_available(aoc_context**,** is_available_input)

检查资源池是否可用。

典型场景:

配置下发时，检查资源池是否可用。

功能描述:

检查资源池是否可用。

接口约束:

资源池已存在。

- **参数**
  - **aoccontext** - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。

  - **is_available_input** (*dict*) - 检查资源池是否可用必需的json参数。
- **引发**
  - **AOCException** - 资源池不存在。
- **返回**
  - 如果资源可用，则为True，否则为False。
- **返回类型**
  - bool

示例

描述样例:

```
>>> is_available_input = {}
>>> is_available_input.setdefault("url", "http://aoc.serviceinject.test")
>>> is_available_input.setdefault("begin", 1)
>>> is_available_input.setdefault("end", 4096)
>>> is_available_input.setdefault("consumer", "testVlan")
>>> is_available_input.setdefault("gid", aoc_context.transactionId)
>>> resource_service = ResoueceMgrService(self.logger)
>>> print resource_service.is_available(aoc_context, is_available_input)
True
```

is_pool_full(aoc_context**,** is_pool_full_input)

检查资源池是否已满。

典型场景:

申请资源时，请检查当前资源池是否已满。

功能描述:

检查资源池是否已满。

接口约束:

资源池已存在。

- **参数**
  - **aoccontext** - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **is_pool_full_input** (*dict*) - 检查资源池是否已满必需的json参数。
- **返回**
  - 如果池已满，则为True，否则为False。
- **返回类型**
  - bool
- **引发**
  - **AOCException** - 资源池不存在。

示例

描述样例:

```
>>> is_pool_full_input = {}
>>> is_pool_full_input.setdefault("url", "http://aoc.serviceinject.test")
>>> is_pool_full_input.setdefault("begin", 1)
```



```
>>> is_pool_full_input.setdefault("end", 4096)
>>> is_pool_full_input.setdefault("consumer", "testVlan")
>>> is_pool_full_input.setdefault("gid", aoc_context.transactionId)
>>> resource_service = ResoueceMgrService(self.logger)
>>> print resource_service.is_pool_full(aoc_context, is_pool_full_input)
True
```

occupy_resources(aoc_context**,** occupy_resources_input**,** belong_service=False)

配置下发时，占用对应的资源。

典型场景:

在配置下发时，需要占用指定的资源。

功能描述:

配置下发时，占用对应的资源。

接口约束:

资源池已存在。

- **参数**
  - **aoccontext** - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **occupy_resources_input** (*dict*) - 占用资源必需的json参数。
  - **belong_service** (*bool, optional*) - 是否属于服务实例。
- **返回**
  - 如果资源占用成功，则为True，否则为False。
- **返回类型**
  - bool
- **引发**
  - **AOCException** - 资源池不存在或资源池已满。

示例

描述样例:

```
>>> occupy_resources_input = {}
>>> occupy_resources_input.setdefault("consumer", "testVlan")
>>> occupy_resources_input.setdefault("gid", aoc_context.transactionId)
>>> resources = {}
>>> resources.setdefault("url", "http://aoc.serviceinject.test")
>>> resources.setdefault("value":11)
>>> occupy_resources_input.setdefault("resources",resources)
>>> occupy_resources_input.setdefault("serviceKey","vlan")
>>> resource_service = ResoueceMgrService(self.logger)
>>> print resource_service.occupy_resources(aoc_context, occupy_resources_input, True)
True
```

release(aoc_context**,** resource_input)

根据入参释放资源。

典型场景:

删除下发的配置后，需要释放对应的已使用的资源。

功能描述:

根据入参释放资源。

接口约束:

指定的资源已存在并被使用。

- **参数**
  - **aoccontext** - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **resource_input** (*dict*) - 根据入参释放资源必需的json参数。
- **返回**
  - 如果资源释放成功，则为True，否则为False。
- **返回类型**
  - bool
- **引发**
  - **AOCException** - 没有资源正在使用。

示例

描述样例:

```
>>> occupy_resources_input = {}
>>> occupy_resources_input.setdefault(""restype", "true")
>>> occupy_resources_input.setdefault("consumer", "testVlan")
>>> occupy_resources_input.setdefault("gid", aoc_context.transactionId)
>>> resource_service = ResoueceMgrService(self.logger)
>>> result = resource_service.release(occupy_resources_input, aoc_context)
>>> print result
True
```

release_by_consumer_restype(aoc_context**,** release_by_consumer_restype_input)

根据指定的消费者和retype释放资源。

典型场景:

删除下发的配置后，需要释放对应的已使用的资源。

功能描述:

根据指定的消费者和retype释放资源。

接口约束:

指定的资源已存在并被使用。

- **参数**
  - **aoccontext** - 上下文信息，详细信息请参考 **aoc.base.aoccontext**。
  - **release_by_consumer_restype_input** (*dict*) - 根据指定的消费者和retype释放资源必需的json参数。
- **引发**
  - **AOCException** - 资源池不存在。
- **返回**
  - 如果资源释放成功，则为True，否则为False。

- **返回类型**
  - bool

示例

描述样例:

```
>>> input = {}
>>> input.setdefault("restype", "true")
>>> input.setdefault("consumer", "testVlan")
>>> input.setdefault("gid", aoc_context.transactionId)ssp error
>>> resource_service = ResoueceMgrService(self.logger)
>>> result = resource_service.release_by_consumer_restype(input, aoc_context)
>>> print result
True
```

10.1.12 aoc.sys.restfulUtils module

典型场景:

使用restful或restconf协议连接设备，可以使用工具类直接调用snd包中设备的北向接口。

功能描述:

该工具类提供通过restful或restconf协议直接调用设备北向接口的功能。

class aoc.sys.restfulUtils.Options(time_out)

基类: object

Parameter is a data structure used in constructing rest configuration parameters.

class aoc.sys.restfulUtils.Parameter(param_map, header_map, raw_data)

基类: object

Parameter is a data structure used in construct request parameters.

class aoc.sys.restfulUtils.RestConfClient

基类: object

典型场景:

使用RESTCONF协议连接设备。您可以使用工具类直接调用SND包中设备的北向接口。

功能描述:

该工具类提供通过RESTCONF协议直接调用设备北向接口的功能。

接口约束:

无。

***static* delete**(rest_server, url, parameters, options)

直接访问设备的北向删除接口。

典型场景:

通过restconf协议管理设备后，可以直接调用插件包中设备的北向删除接口。

功能描述:

该方法提供直接访问设备北向删除接口的功能。

接口约束:

无。

- **参数**
  - **rest_server** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Server**。
  - **url** (*str*) - Request URL.
  - **parameters** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Parameter**。
  - **options** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Options**。
- **返回**
  - 调用接口返回的包或参数信息。
- **返回类型**
  - str
- **引发**
  - **UserException** - 服务异常。

实际案例

描述样例:

```
>>> rest_server = Server("d4523983-f608-41bf-8aa2-c48bf7910a9a")
>>> header_map = {"Content-Type":"application/json"}
>>> raw_data = '{"test":"data"}'
>>> parameters = Parameter(None,header_map,raw_data)
>>> options = Options(60000)
>>> url = "/restconf/data/huawei-nce-tunnel-trail:tunnel-trails/tunnel-trail"
>>> response = delete(rest_server,url,parameters,options)
>>> print response
{responseContent:"xxx","status":"success"}
```

***static* get**(rest_server, url, parameters, options)

直接访问设备的北向get接口。

典型场景:

通过restconf协议管理设备后，可以直接调用插件包中设备的北向get接口。

功能描述:

提供直接访问设备北向get接口的功能。

接口约束:

无。

- **参数**

  - **rest_server** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Server**。
  - **url** (*str*) - Request URL.
  - **parameters** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Parameter**。
  - **options** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Options**。
- **返回**
  - 调用接口返回的包或参数信息。
- **返回类型**
  - str
- **引发**
  - **UserException** - 服务异常。

实际案例

描述样例:

```
>>> rest_server = Server("d4523983-f608-41bf-8aa2-c48bf7910a9a")
>>> header_map = {"Content-Type":"application/json"}
>>> raw_data = '{"test":"data"}'
>>> parameters = Parameter(None,header_map,raw_data)
>>> options = Options(60000)
>>> url = "/restconf/data/huawei-nce-tunnel-trail:tunnel-trails/tunnel-trail"
>>> response = get(rest_server,url,parameters,options)
>>> print response
{responseContent:"xxx","status":"success"}
```

***static* patch**(rest_server, url, parameters, options)

直接访问设备的北向补丁接口。

典型场景:

通过restconf协议管理设备后，可以直接调用插件包中设备的北向补丁接口。

功能描述:

该方法提供直接访问设备北向补丁接口的功能。

接口约束:

无。

- **参数**
  - **rest_server** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Server**。
  - **url** (*str*) - Request URL.
  - **parameters** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Parameter**。
  - **options** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Options**。
- **返回**

  - 调用接口返回的包或参数信息。
- **返回类型**
  - str
- **引发**
  - **UserException** - 服务异常。

实际案例

描述样例:

```
>>> rest_server = Server("d4523983-f608-41bf-8aa2-c48bf7910a9a")
>>> header_map = {"Content-Type":"application/json"}
>>> raw_data = '{"test":"data"}'
>>> parameters = Parameter(None,header_map,raw_data)
>>> options = Options(60000)
>>> url = "/restconf/data/huawei-nce-tunnel-trail:tunnel-trails/tunnel-trail"
>>> response = patch(rest_server,url,parameters,options)
>>> print response
{responseContent:"xxx","status":"success"}
```

***static* post**(rest_server, url, parameters, options)

直接访问设备的北向post接口。

典型场景:

通过restconf协议管理设备后，可以直接调用插件包中设备的北向post接口。

功能描述:

该方法提供直接访问设备北向post接口的功能。

接口约束:

无。

- **参数**
  - **rest_server** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Server**。
  - **url** (*str*) - Request URL.
  - **parameters** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Parameter**。
  - **options** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Options**。
- **返回**
  - 调用接口返回的包或参数信息。
- **返回类型**
  - str
- **引发**
  - **UserException** - 服务异常。

实际案例

描述样例:

```
>>> rest_server = Server("d4523983-f608-41bf-8aa2-c48bf7910a9a")
>>> header_map = {"Content-Type":"application/json"}
>>> raw_data = '{"test":"data"}'
>>> parameters = Parameter(None,header_map,raw_data)
>>> options = Options(60000)
>>> url = "/restconf/data/huawei-nce-tunnel-trail:tunnel-trails/tunnel-trail"
>>> response = post(rest_server,url,parameters,options)
>>> print response
{responseContent:"xxx","status":"success"}
```

***static* put**(rest_server, url, parameters, options)

直接访问设备的北向put接口。

典型场景:

通过restconf协议管理设备后，可以直接调用插件包中设备的北向put接口。

功能描述:

该方法提供直接访问设备北向put接口的功能。

接口约束:

无。

- **参数**
  - **rest_server** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Server**。
  - **url** (*str*) - Request URL.
  - **parameters** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Parameter**。
  - **options** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Options**。
- **返回**
  - 调用接口返回的包或参数信息。
- **返回类型**
  - str
- **引发**
  - **UserException** - 服务异常。

实际案例

描述样例:

```
>>> rest_server = Server("d4523983-f608-41bf-8aa2-c48bf7910a9a")
>>> header_map = {"Content-Type":"application/json"}
>>> raw_data = '{"test":"data"}'
>>> parameters = Parameter(None,header_map,raw_data)
>>> options = Options(60000)
>>> url = "/restconf/data/huawei-nce-tunnel-trail:tunnel-trails/tunnel-trail"
>>> response = put(rest_server,url,parameters,options)
>>> print response
{responseContent:"xxx","status":"success"}
```


class aoc.sys.restfulUtils.RestFulClient

基类： object

通过restful协议直接调用设备北向接口。

典型场景:

使用restful协议连接设备，可以使用工具类直接调用snd包中设备的北向接口。

功能描述:

该工具类提供通过restful协议直接调用设备北向接口的功能。

接口约束:

无。

***static* delete**(rest_server, url, parameters, options)

直接访问设备的北向删除接口。

典型场景:

使用restful协议管理设备后，可以直接调用插件包中设备的北向删除接口。

功能描述:

该方法提供直接访问设备北向删除接口的功能。

接口约束:

无。

- **参数**
  - **rest_server** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Server**。
  - **url** (*str*) - Request URL.
  - **parameters** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Parameter**。
  - **options** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Options**。
- **返回**
  - 调用接口返回的包或参数信息。
- **返回类型**
  - str
- **引发**
  - **UserException** - 服务异常。

实际案例

描述样例:

```
>>> rest_server = Server("d4523983-f608-41bf-8aa2-c48bf7910a9a")
>>> header_map = {"Content-Type":"application/json"}
>>> raw_data = '{"test":"data"}'
>>> parameters = Parameter(None,header_map,raw_data)
>>> options = Options(60000)
```

```
>>> url = "/restconf/data/huawei-nce-tunnel-trail:tunnel-trails/tunnel-trail"
>>> response = delete(rest_server,url,parameters,options)
>>> print response
{responseContent:"xxx","status":"success"}
```

***static* get**(rest_server, url, parameters, options)

直接访问设备的北向get接口。

典型场景:

使用restful协议管理设备后，可以直接调用插件包中设备的北向get接口。

功能描述:

该方法提供直接访问设备北向get接口的功能。

接口约束:

无。

- **参数**
  - **rest_server** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Server**。
  - **url** (*str*) - Request url.
  - **parameters** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Parameter**。
  - **options** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Options**。
- **返回**
  - 调用接口返回的包或参数信息。
- **返回类型**
  - str
  - **引发**
  - **UserException** - 服务异常。

实际案例

描述样例:

```
>>> rest_server = Server("d4523983-f608-41bf-8aa2-c48bf7910a9a")
>>> header_map = {"Content-Type":"application/json"}
>>> raw_data = '{"test":"data"}'
>>> parameters = Parameter(None,header_map,raw_data)
>>> options = Options(60000)
>>> url = "/restconf/data/huawei-nce-tunnel-trail:tunnel-trails/tunnel-trail"
>>> response = get(rest_server,url,parameters,options)
>>> print response
{responseContent:"xxx","status":"success"}
```

***static* patch**(rest_server, url, parameters, options)

直接访问设备的北向补丁接口。

典型场景:

使用restful协议管理设备后，可以直接调用插件包中设备的北向补丁接口。

功能描述:

该方法提供直接访问设备北向补丁接口的功能。

接口约束:

无。

- **参数**
  - **rest_server** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Server**。
  - **url** (*str*) - Request URL.
  - **parameters** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Parameter**。
  - **options** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Options**。
- **返回**
  - 调用接口返回的包或参数信息。
- **返回类型**
  - str
- **引发**
  - **UserException** - 服务异常。

实际案例

描述样例:

```
>>> rest_server = Server("d4523983-f608-41bf-8aa2-c48bf7910a9a")
>>> header_map = {"Content-Type":"application/json"}
>>> raw_data = '{"test":"data"}'
>>> parameters = Parameter(None,header_map,raw_data)
>>> options = Options(60000)
>>> url = "/restconf/data/huawei-nce-tunnel-trail:tunnel-trails/tunnel-trail"
>>> response = patch(rest_server,url,parameters,options)
>>> print response
{responseContent:"xxx","status":"success"}
```

***static* post**(rest_server, url, parameters, options)

直接访问设备的北向post接口。

典型场景:

使用restful协议管理设备后，可以直接调用插件包中设备的北向post接口。

功能描述:

该方法提供直接访问设备北向post接口的功能。

接口约束:

无。

- **参数**
  - **rest_server** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Server**。

  - **url** (*str*) - Request URL.
  - **parameters** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Parameter**。
  - **options** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Options**。
- **返回**
  - 调用接口返回的包或参数信息。
- **返回类型**
  - str
- **引发**
  - **UserException** - 服务异常。

实际案例

描述样例:

```
>>> rest_server = Server("d4523983-f608-41bf-8aa2-c48bf7910a9a")
>>> header_map = {"Content-Type":"application/json"}
>>> raw_data = '{"test":"data"}'
>>> parameters = Parameter(None,header_map,raw_data)
>>> options = Options(60000)
>>> url = "/restconf/data/huawei-nce-tunnel-trail:tunnel-trails/tunnel-trail"
>>> response = post(rest_server,url,parameters,options)
>>> print response
{responseContent:"xxx","status":"success"}
```

***static* put**(rest_server, url, parameters, options)

直接访问设备的北向put接口。

典型场景:

使用restful协议管理设备后，可以直接调用插件包中设备的北向put接口。

功能描述:

该方法提供直接访问设备北向put接口的功能。

接口约束:

无。

- **参数**
  - **rest_server** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Server**。
  - **url** (*str*) - Request URL.
  - **parameters** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Parameter**。
  - **options** (*obj*) - 上下文信息，详细信息请参考 **aoc.sys.restfulUtils.Options**。
- **返回**
  - 调用接口返回的包或参数信息。
- **返回类型**

  - str
- **引发**
  - **UserException** - 服务异常。

实际案例

描述样例:

```
>>> rest_server = Server("d4523983-f608-41bf-8aa2-c48bf7910a9a")
>>> header_map = {"Content-Type":"application/json"}
>>> raw_data = '{"test":"data"}'
>>> parameters = Parameter(None,header_map,raw_data)
>>> options = Options(60000)
>>> url = "/restconf/data/huawei-nce-tunnel-trail:tunnel-trails/tunnel-trail"
>>> response = put(rest_server,url,parameters,options)
>>> print response
{responseContent:"xxx","status":"success"}
```

class aoc.sys.restfulUtils.Server(ne_id)

基类： object

Server is a data structure used in constructing server details

10.1.13 aoc.sys.service module

典型场景:

本模块定义了业务的基本操作和维护。

功能描述:

本模块定义了业务的基本操作和维护。

aoc.sys.service.redeploy(service_path)

同步重新部署。

典型场景:

如果事务正在进行，您想更新另一个业务实例，可以调用此函数。此函数为同步操作。

功能描述:

同步重新部署挂载的业务实例。

接口约束:

无。

说明

要重新部署的参数。

- **参数**
  - **service_path** (*str*) - 此路径指示业务的实例

示例

```
>>>service_path = “/huawei-ac-applications:applications/huawei-ncs-eas:eas/eas_1”
>>>redeploy(service_path)
```

aoc.sys.service.redeploy_async(service_path)

异步重新部署。

典型场景:

如果事务正在进行，您想更新另一个业务实例，可以调用此函数。此函数为异步操作。

功能描述:

以异步方式重新部署挂载的业务实例。

接口约束:

无。

说明

要重新异步部署的参数。

- **参数**
  - **service_path** (*str*) - 此路径指示业务的实例

示例

```
>>>service_path = “/huawei-ac-applications:applications/huawei-ncs-eas:eas/eas_1”
>>>redeploy_async(service_path)
```

aoc.sys.service.undeploy(service_path)

同步取消部署。

典型场景:

如果事务正在进行中，您希望删除另一个业务实例，您可以调用此函数。此函数为同步操作。

功能描述:

以同步方式卸载已挂载的业务实例。

接口约束:

无。

说明

要去部署的参数。

- **参数**
  - **service_path** (*str*) - 此路径指示业务的实例

示例

```
>>>service_path = “/huawei-ac-applications:applications/huawei-ncs-eas:eas/eas_1”
>>>undeploy(service_path)
```


aoc.sys.service.undeploy_async(service_path)

异步取消部署。

典型场景:

如果事务正在进行中，您希望删除另一个业务实例，可以调用此函数。此函数为异步操作。

功能描述:

以异步方式卸载已挂载的业务实例。

接口约束:

无。

说明

要异步去部署的参数。

- **参数**
  - **service_path** (*str*) - 此路径指示业务的实例。

示例

```
>>>service_path = “/huawei-ac-applications:applications/huawei-ncs-eas:eas/eas_1”
>>>undeploy_async(service_path)
```

10.1.14 aoc.sys.serviceinjectservice module

典型场景:

在制作SSP映射逻辑的过程中，用户可能会使用一些系统中没有内置的接口或能力，如外部资源管理，需要通过服务注入将外部接口或能力放入系统中。 此模块被用于SSP脚本用这些外部能力。

功能描述:

该模块为用户提供调用外部服务的接口。

10.1.15 aoc.sys.snmp module

典型场景:

1. 告警同步。
2. 通过snmp获取设备信息。

功能描述:

提供通过snmp与设备交互的能力。

class aoc.sys.snmp.Snmp

基类：object

snmp进程模块。

典型场景:

当用户希望通过snmp协议获取告警数据时，请使用此API。

功能描述:

本模块提供获取告警信息的方法。

接口约束:

无。

static **get**(device, oids, aoccontext=None)

获取设备侧告警数据。

典型场景:

1. 告警同步。
2. 通过snmp获取设备信息。

功能描述:

获取设备侧告警数据。

接口约束:

无。

- **参数**
  - **device** (*str*) - 设备ID。
  - **oids** (*list*) - 设备的oids。
  - **aoccontext** (*AocContext*) - 上下文信息，详细信息请参考 **aoc.base.aoccontext**.
- **返回**
  - oids的成员。
- **返回类型**
  - 它由proto文件定义，路径为 proto.aoc.sys.AocSyncSnmp.AocSyncSnmpOutput

实际案例

```
>>> print(Snmp.get('868e778e-153c-3afe-a02c-89678e31e3e4', ['1.3.6.1.2.1.2.2.1.1']))
{'1.3.6.1.2.1.2.2.1.1': {'deviceId':'868e778e-153c-3afe-a02c-89678e31e3e4',
'trapOidEntity':[{'oid':'1.3.6.1.2.1.2.2.1.1', 'variable': ''DTA}]}}
```

10.1.16 aoc.sys.sys_model_pb2 package

10.1.17 aoc.sys.transaction module

典型场景:

用户在SSP的rpc代码中发送配置，并可以调用该模块创建事务、编辑配置、提交事务、回滚事务。

功能描述:

提供用户事务配置接口。

aoc.sys.transaction.sdk

sdk，用于将请求从open api发送到aoc。

Type: object

aoc.sys.transaction.logger

当前包组的日志打印

Type: str, optional

aoc.sys.transaction.ne_id_pattern

用于匹配设备ID的正则表达式。

Type: str

class aoc.sys.transaction.CommitParam(only_service=False, force=False, no_network=False, service_parallel=False)

基类：object

典型场景:

使用参数提交。

功能描述:

commitParam是commit函数中使用的参数的定义

接口约束:

无。

- **参数**
  - **only_service**（*bool*）- 是否分解成网元级数据
  - **force**（*bool*）- 系统不检查:
  - **控制器与设备之间的数据是否同步。**（*1.*）-
  - **当前配置中是否存在数据源冲突。**（*2.*）-
  - **no_network**（*bool*）- 系统提交配置并保存到数据库，但不下发到设备。
  - **service_parallel**（*bool*）- 系统将使用并行提交配置。

only_service

是否分解成网元级数据

Type：bool

force

系统不检查:

Type：bool

1. 控制器与设备之间的数据是否同步。
2. 当前配置中是否存在数据源冲突。

no_network

系统提交配置并保存到数据库，但不下发到设备。

Type：bool

service_parallel

系统将使用并行提交配置。

Type：bool

is_force()

典型场景:

判断是否强制提交模式。

功能描述:

判断是否强制提交模式。

接口约束:

无。

- **返回**
  - force的布尔值
- **返回类型**
  - force

示例

```
>>> print is_force()
False
```

is_no_network()

典型场景:

判断是否为不下发模式。

功能描述:

判断是否为不下发模式。

接口约束:

无。

- **返回**
  - no_network的布尔值
- **返回类型**
  - no_network

示例

```
>>> print is_network()
False
```

is_only_service()

典型场景:

判断是否为不分解业务模式。

功能描述:

判断是否为不分解业务模式。

接口约束:

无。

- **返回**
- only_service的布尔值
- **返回类型**
- only_service

示例

```
>>> print is_only_service()
False
```

is_service_parallell()

典型场景:

判断是否并发模式。

功能描述:

判断是否并发模式。

接口约束:

无。

- **返回**
  - service_parallell的布尔值
- **返回类型**
  - service_parallell

示例

```
>>> print is_service_parallell()
False
```

aoc.sys.transaction.commit_transaction(*transid*, *time_out=10*, *commitparam=<aoc.sys.transaction.CommitParam object>*)

系统提供接口供用户提交两阶段交易。

典型场景:

用户在SSP的rpc代码中下发配置，可以调用此接口提交事务。

功能描述:

用户在SSP的rpc代码中下发配置，可以调用此接口提交事务。

接口约束:

提交失败需要释放此事务。

- **参数**
  - **transid** - 事务ID
  - **time_out** - 提交超时时间
  - **commitparam** - commit函数中使用的参数
- **返回**

```
message NcsServiceConfigCommitTransOutput {
    NcsServiceBaseResult result;
}
message NcsServiceBaseResult {
    bool result; # result
    string errorCode; # error code
    string errorMsg; # error msg
}
```

- **引发**
  - **AOCException** - 检查预配置失败或网元数据与系统现有配置重叠或两个服务修改了相同的网元数据， 数据不一致或事务过期或netconf协议断开

示例

```
>>> commit_param = CommitParam()
>>> result = commit_transaction('',10,commit_param)
>>> print(result)
{'result':{'result':true}}
```

aoc.sys.transaction.create_transaction(time_out=10)

系统提供创建两阶段事务的接口。

典型场景:

用户在SSP的rpc代码中下发配置，并可以调用此接口创建事务。

功能描述:

提供创建事务接口。

接口约束:

需要释放此事务。

- **参数**
  - **time_out** (*int*) - 创建事务超时时间
- **返回**
  - 事务ID
- **返回类型**
  - str
- **引发**
  - **AOCException** - 数据不存在或应用的事务数超过默认最大值200

示例

```
>>> result = create_transaction(10)
>>> print(result)
8d394835-cb84-38f3-a4d5-36a7f2074b78
```


aoc.sys.transaction.edit_transaction(transid, data, path, action)

系统提供接口供用户创建两阶段事务编辑。

典型场景:

用户在SSP的rpc代码中发送配置，可以调用此接口根据事务编辑配置。

功能描述:

用户在SSP的rpc代码中发送配置，可以调用此接口根据事务编辑配置。

接口约束:

事务已创建，事务用完后需要释放此事务。

- **参数**
  - **transid** (*str*) - 事务ID
  - **data** (*str, optional*) - 配置数据
  - **path** (*str*) - 业务路径
  - **action** (*str*) - 操作类型： create,delete,remove,merge,replace
- **返回**

```
Structure defined by protobuf
message NcsServiceConfigEditOutput {
    NcsServiceBaseResult result;
}
message NcsServiceBaseResult {
    bool result; # result
    string errorCode; # error code
    string errorMsg; # error msg
}
```

- **引发**
  - **AOCException** - 两个业务修改相同网元数据或事务过期

示例

```
>>> result = edit_transaction('8d394835-cb84-38f3-a4d5-36a7f2074b78',
'<aaa>111</aaa>', 'huawei-aaa:aaa/111', 'replace')
>>> print(result)
{'result':{'result':true}}
```

aoc.sys.transaction.rollback_transaction(transid)

系统提供回滚两阶段事务的接口。

典型场景:

用户在SSP的rpc代码中下发配置，可以调用此回滚取消事务。

功能描述:

用户在SSP的rpc代码中下发配置，可以调用此接口回滚事务。

接口约束:

无。

说明

如果事务撤消失败，则不需要处理。异常场景将由老化机制保证。

- **参数**
  - **transid** (*str*) - 事务ID
- **返回**

```
Structure defined by protobuf
    message NcsServiceConfigResetTransOutput {
    NcsServiceBaseResult result;
}
message NcsServiceBaseResult {
    bool result; # result
    string errorCode; # error code
    string errorMsg; # error msg
}
```

示例

```
>>> result = rollback_transaction('8d394835-cb84-38f3-a4d5-36a7f2074b78', False)
>>> print(result)
{'result':{'result':true}}
```

10.2 aoc.sys.innject package

10.2.1 aoc.sys.inject.serviceinject module

典型场景:

在制作SSP映射逻辑的过程中，用户可能会使用一些系统中没有内置的接口或能力，如外部资源管理，需要通过服务注入将外部接口或能力放入系统中。 SSP脚本使用此模块的类和方法调用这些外部能力。

功能描述:

该模块为用户提供调用外部服务的接口。

class aoc.sys.inject.serviceinject.ServiceInject(logger=None)

基类： object

服务注入接口。

典型场景:

在制作SSP映射逻辑的过程中，用户可能会使用一些系统中没有内置的接口或能力，如外部资源管理，需要通过服务注入将外部接口或能力放入系统中。 SSP脚本使用此模块的类和方法调用这些外部能力。

功能描述:

该类提供服务注入接口，供用户调用。

接口约束:

服务提供者已将服务注入系统。

说明

1. **通过调用服务提供者 self.serviceinject.action(self.service_name, input)**在action方法中，service_name是服务提供者的钩子键，输入需要根据服务提供者构造。
2. **通过self.serviceinject.run(self.service_name, key, input,aoccontext)**调用服务提供者的run方法，key是用户定义的关键字，用于区分其他调用。
3. **通过ncs_context.setdefault（’vlan1’，结果）设置上下文，该上下文将在下次执行时打开。**可编程框架将通过request.context引入最后提交的上下文。

- **参数**
  - **logger** (*str, optional*) - 当前包组的日志打印

logger

当前包组的日志打印

Type：str, optional

client_util

rest发送客户端

Type：obj

action(service_name, service_data, aoc_context=None)

系统为用户调用的服务提供了action接口，action接口没有自动回滚的能力。

典型场景:

用户作为服务的用户，实现机制的action方法申请资源。

功能描述:

用户作为服务的用户，实现机制的action方法申请资源。

接口约束:

无。

说明

通过self.serviceinject.action（self.service_name，输入）Action方法调用服务提供者，service_name是服务提供者的钩子键，输入需要根据服务提供者构造

- **参数**
  - **service_name** - 服务提供商包中的钩子密钥
  - **service_data** - 由服务消费者组装的字典，由服务提供者包操作函数输入参数
- **返回**
  - 通过服务提供者包运行函数返回字典汇编

示例

```
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))
```



```
>>> input = {}
>>> input.setdefault("operation", "query")  # Query whether the resource pool exists
>>> input.setdefault("request", self.gen_request())
>>> result = Serviceinject().action('ippool', input)
```

```
run(service_name, key, service_data, aoc_context, dst='ecs')
```

系统为用户提供注入运行接口的服务，运行接口具有自动回滚的能力。

典型场景:

作为服务的用户，用户实现机制的run方法，如申请资源。

功能描述:

作为服务的用户，用户实现机制的run方法。

接口约束:

服务提供者已将服务注入系统。

说明

通过self.serviceinject.run(self.service_name, key, input,aoccontext)调用服务提供者的run方法， key是用户定义的关键字，用于区分其他调用。

- **参数**
  - **service_name** - 服务提供商包中的钩子密钥
  - **key** - 由服务使用者定义的字符串
  - **service_data** - 由服务消费者组装的字典，由服务提供者包运行函数输入参数
  - **aoc_context** - 上下文信息, 有关详细信息，请参见：类：**aoc.base.aoccontext**.
- **返回**
  - 通过服务提供者包运行函数返回字典汇编

示例

```
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))
>>> input = {}
>>> input.setdefault("operation", "apply")  # apply
>>> input.setdefault("request", self.gen_request(gid))
>>> result = Serviceinject().run('ippool', 'vlan1', input, aoccontext)
>>> ip_addr = result.get("result")
10.10.10.10
```

10.2.2 aoc.sys.inject.serviceprovider module

典型场景:

用户可以实现该模块的类接口，并在aoc系统中提供自定义服务。

功能描述:

提供用户业务注入接口。

class aoc.sys.inject.serviceprovider.ServiceProvider(logger=None, resource_dir='')

基类：aoc.base.baseservice.BaseService

服务注入提供者的接口定义。

典型场景:

在制作SSP映射逻辑的过程中，用户可能会使用一些系统中没有内置的接口或能力，如外部资源管理，需要通过服务注入将外部接口或能力放入系统中。 SSP脚本使用此模块的类和方法调用这些外部能力。

功能描述:

此类提供服务注入提供程序的接口定义，由用户重写。

接口约束:

1. 服务注入生产者应该实现run/revert/action函数
2. **服务注入生产者需要为用户提供：**
   a. 输入构造方法
   b. 返回解析方法
   c. 框架存储运行函数的输出，当需要回滚时，将运行的输出作为恢复函数的输入传递。 所以服务提供者需要确保run的输出可以作为恢复的输入，并产生反向回滚的效果。

说明

1. run/revert/action方法的输入和输出是dict类型，不能是其他类型。
2. 运行是有状态的。当预提交失败时，开放可编程框架将调用恢复方法
3. run的输入结构需要提供给用户，run的输出可以作为恢复的输入。
4. action必须是可重入的，当使用相同的输入重复调用操作时，不能报告错误。
5. pkg.json中的钩子类型必须是service提供者，钩子密钥也必须提供给用户。

- **参数**
  - **logger** (*str, optional*) - 当前包组的日志打印
  - **resourceDir** (*str*) - 资源目录

global_val

全局变量

logger

当前包组的日志打印

Type： str, optional

resourceDir

资源目录

Type： str

action(action_input)

系统提供给用户的业务注入action接口。

典型场景:

作为服务提供者，用户实现机制的action方法。

功能描述:

作为服务提供者，用户实现机制的action方法。

接口约束:

操作必须是可重入的，当使用相同的输入重复调用操作时，不能报告错误。 pkg.json中的钩子类型必须是service提供者，钩子密钥也必须提供给用户。

注解

说明

操作必须是可重入的，当使用相同的输入重复调用操作时，不能报告错误。

- **参数**
  - **action_input** (*str*) - action入参为json格式
- **返回**
  - dict返回数据

示例

```
def action(self, action_input):
    self.log.info('action %s' % action_input)
    if action_input.get("operation")  == "create":
        request = action_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
```

revert(revert_input)

系统向用户提供业务注入恢复接口。

典型场景:

当事务失败时，开放可编程框架将调用恢复方法。

功能描述:

作为服务提供者，用户实现机制的恢复方法。

接口约束:

run的输入结构需要提供给用户，run的输出可以作为恢复的输入。 pkg.json文件中的钩子类型必须是service提供者，钩子密钥也必须提供给用户。

- **参数**
- **revert_input** - 恢复输入参数，需要向用户提供运行的输入结构，运行的输出可以作为恢复的输入

- **返回**
- dict返回数据

示例

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

run(run_input)

系统为用户提供服务注入运行接口。

典型场景:

服务用户用户、调用服务运行接口、申请资源等。

功能描述:

作为服务提供者，用户实现机制的run方法。

接口约束:

pkg.json中的钩子类型必须是service提供者，钩子密钥也必须提供给用户。

注解

运行是有状态的。当事务失败时，开放可编程框架将调用恢复方法。 run的输入结构需要提供给用户，run的输出可以作为恢复的输入。

- **参数**
  - **run_input** - 运行输入参数，json格式数据，服务用户根据提供程序的格式设置json。
- **返回**
  - dict返回数据

示例

```
ip_pool = set()
def run(self, run_input):
    self.log.info('run %s' % run_input)
    output = {}
    if run_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
```

