在Nexus 3000/9000交换机中开发、调试和部署NX-SDK Python应用

目录

简介

本文档提供Python应用开发的工作流程，该工作流程使用Nexus 3000和Nexus 9000平台上的Cisco NX-OS和Cisco NX-Software Development Kit(SDK)。

先决条件

要求

本文档没有任何特定的要求。

使用的组件

本文档中的信息基于以下软件和硬件版本：

- 本文档使用NX-SDK v1.0.0和NX-SDK v1.5.0
- 从NX-OS版本7.0(3)I6(1)开始，NX-SDK v1.0.0可用于Nexus 9000平台，从NX-OS版本7.0(3)I7(1)开始，可用于Nexus 3000平台
- 从NX-OS版本7.0(3)I7(3)开始，NX-SDK v1.5.0可同时用于Nexus 9000平台和Nexus 3000平台

本文档中的信息都是基于特定实验室环境中的设备编写的。本文档中使用的所有设备最初均采用原始（默认）配置。如果您使用的是真实网络，请确保您已经了解所有命令的潜在影响。

背景信息

Cisco NX-SDK允许开发可在Nexus 9000和Nexus 3000平台上的Cisco NX-OS中本地运行的自定义应用。NX-SDK使客户能够创建自己的CLI命令和输出、生成自定义系统日志以响应特定事件、流定制遥测等。

NX-SDK有一个C++ API，它使用简化的包装和接口生成器(SWIG)翻译成其他语言。 这允许客户使用他们选择的任何语言的NX-SDK。本文档演示了在Python中实施常见NX-SDK功能，并为客户提供了开发其自己的NX-SDK Python应用的工作流程。

使用NX-SDK开发Python应用

启用NX-SDK

要运行任何NX-SDK应用，必须首先在设备上启用NX-SDK功能：

```
switch(config)# feature nxsdk
```

创建Python文件

可以使用NX-OS Bash外壳创建和编辑Python文件。要使用Bash外壳，必须首先在设备上启用：

```
switch(config)# feature bash-shell
```

输入Bash外壳并使用vi文本编辑器创建和编辑Python文件：

```
switch(config)# run bash
bash-4.2$ vi /isan/bin/nxsdk-app.py
```

> **注意**：最佳实践是在/isan/bin/目录**中创建Python**文件。Python文件需要执行权限才能运行 — 不要将Python文件放在/bootflash目**录或**其任何子目录中。

> **注意**：无需通过NX-OS创建和编辑Python文件。开发人员可以使用其本地环境创建应用，并使用他们选择的文件传输协议将完成的文件传输到设备。但是，开发人员使用NX-OS实用程序调试脚本并排除其故障可能会更加高效。

实施NX-SDK组件

建议开发人员开始从 Cisco DevNet NX-SDK GitHub上的customCliPyApp模板创建其NX-SDK Python应用程序。本文档的这些部分引用了使用此模板中使用其名称的必要组件。

NX-SDK Python应用需要四个主要组件：

1. 必须通过import nx_sdk_py语句将NX-SDK导**入到应用程**序。
2. 启动NX-SDK应用并修改与应用相关的各种选项的函数(通常称为**sdkThread**)。
3. 创建自定义CLI命令以及定义sdkThread函数中的自定义CLI命令**语法**。
4. 名为**pyCmdHandler**的类，其方法为**postCliCb**，该类处理由NX-SDK应用程序添加的自定义CLI命令。

sdk线程函数

sdkThread函数初始化、添加功能并启动NX-SDK应用。该函数不需要将任何参数传递到该函数中。所有Python NX-SDK应用程序都需要从nx_sdk_py库调用三种方法：

1. **nx_sdk_py.NxSdk.getSdkInst(len(sys.argv), sys.argv)**返回SDK实例对象或返回None。如果返回SDK实例对象，则NX-SDK应用已成功注册到NX-OS基础设施。如果返回None，则此注册过程发生错误，并且设备的系统日志中会显示错误日志条目。此处记录了此方法。

   注意：从NX-SDK v1.5.0开始，第三个布尔参数可以传递到**NxSdk.getSdkInst**方法，该方法在True时启用Advanced Exceptions，在False时禁用Advanced Exceptions。此处记录了此方法。

1. **sdk.startEventLoop()**方法，**其**中sdk是nx_sdk_py.NxSdk.getSdkInst()方**法返回的SDK实例对**象。此方法启动NX-SDK应用并允许其与NX-OS基础设施交互。此处记录了此方法。
2. **nx_sdk_py.NxSdk.__swig_destroy_(sdk)方**法，其中**sdk**是前面介绍的**nx_sdk_py.NxSdk.getSdkInst()方法返**回的SDK实例对象。在sdkThread函数末尾放置的此方**法允**许NX-SDK应用的平稳退出。

一些常用方法包括：

- **sdk.getTracer()**，其中**sdk**是nx_sdk_py.NxSdk.getSdkInst()方**法返回的SDK实例对**象。此方法返回可用于生成自定义系统日志的NxTrace对象，以及将事件和错误记录到应用程序的事件历史记录。自定义系统日志将显示在设备的系统日志中(通过**show logging logfile**命令可见)，而通过**show <application-name> nxsdk event-history events** 或**show <application-name> nxsdk event-history errors** 命令可以看到记录到应用事件历史记录的事件。此处记录了此方法。此处记录了此方法返回的NxTrace对象及其相关方法。例如，可以通过**show Transever_DOM.py nxsdk event-history events**和**show Transever_DOM.py nxsdk event-history errors**命令查看名为**Transever_DOM.py**的应用的事件历史记录。
- **sdk.getCliParser()**，其中**sdk**是nx_sdk_py.NxSdk.getSdkInst()方**法返回的SDK实例对**象。此方法返回NxCliParser对象，该对象可用于执行已经通过Python存在的CLI命令以及创建自定义CLI命令。此处记录了此方法。此方法返回的NxCliParser对象及其关联方法记录在此处。
- **cliP.execShowCmd("cmd",return_type)**，其中**cliP**是**sdk.getCliParser()方法返回的NxCliParser对象**,**cmd**是要以引号进行封装的show命令，**return_type**是要输出的数据格式。数

据格式可以是R_TEXT、R_JSON或R_XML。此处记录了此方法。

注意：R_JSON和R_XML数据格式仅在命令支持这些格式的输出时有效。在NX-OS中，可以通过将输出管道化为请求的数据格式来验证命令是否支持特定数据格式的输出。如果管道命令返回有意义的输出，则支持该数据格式。例如，如果运行**show mac address-table dynamic | NX**-OS中的json返回JSON输出，NX-SDK中也支持R_JSON数据格式。

- **cliP.execConfigCmd(cmd_filename)**，其中**cliP**是**sdk.getCliParser()方法返回的NxCliParser对象，cmd_filename**是包含以行分隔的命令的文件的绝对文件路径。此方法返回一个字符串，指示命令执行成功 — 如果字符串中包含“SUCCESS”，则成功执行所有命令。否则，字符串包含说明命令执行失败原因的异常。此处记录了此方法。

有些可选方法可以有所帮助：

- **sdk.setAppDesc('description string')**，其中**sdk**是nx_sdk_py.NxSdk.getSdkInst()方**法返回的SDK实例对**象。此方法设置NX-SDK应用的说明。说明显示在NX-OS上下文相关帮助菜单中，该帮助菜单可通过CLI上的问号访问。此处记录了此方法。例如，名为**Transever_DOM.py**的应用程序说明“返回插入了支持DOM的收发器的所有接口”(Returns all interfaces with DOM-capable transfeers intered)显示在NX-OS上下文相关帮助中，如下所示：

```
 N9K-C93180LC-EX# show Tra?
track Tracking information
Transceiver_DOM.py Returns all interfaces with DOM-capable transceivers inserted
```

- **sdk.getAppName()**，其中**sdk**是nx_sdk_py.NxSdk.getSdkinst()方**法返回的SDK实例对**象。此方法返回Python应用的名称。此处记录了此方法。

创建自定义CLI命令

在使用NX-SDK的Python应用中，自定义CLI命令在sdkThread函数中创建和定义。命令有两种类型：**Show**命令和**Config**命令。

1. **Show**命令显示有关设备、其配置或环境的信息。
2. **配置**命令会更改设备的配置，从而修改设备对周围网络的反应方式。

这两种方法允许分别创建show命令和config命令：

- **cliP.newShowCmd("cmd_name", "syntax")**，其中**cliP**是**sdk.getCliParser()方法返回的NxCliParser对象， cmd_name**是自定义NX-SDK应用程序内部命令的唯一名称，语法描述**命令中可以使用哪些关键字和参数。**此方法返回NxCliCmd对象，此处已记录。此处记录了此方法。

注意：此命令是cliP.newCliCmd("cmd_type"、"cmd_name"、"syntax")的子类，其中cmd_type是CONF_CMD或**SHOW_CMD**（取决于所配置的命令类型），**cmd_name**是自定义NX内部命令的唯一名称SDK应用程序和语法描述了命令中可以使用哪些关键字和参数。因此，此命令的API文档可能更有助于参考。

- **cliP.newConfigCmd("cmd_name", "syntax")**，其中**cliP**是**sdk.getCliParser()方法返回的NxCliParser对象， cmd_name**是自定义NX-SDK应用程序内部命令的唯一名称，语法描述**命令中可以使用哪些关键字和参数。**此方法返回NxCliCmd对象，此处记录。此处记录了此方法。

注意：此命令是cliP.newCliCmd("cmd_type"、"cmd_name"、"syntax")的子类，其中cmd_type是CONF_CMD或**SHOW_CMD**（取决于所配置的命令类型），**cmd_name**是自定义CMD内部命令的唯一名称nx-SDK应用程序和语法描述命令中可以使用哪些关键字和参数。因

此，此命令的API文档可能更有助于参考。

这两种命令都有两个不同的组件：参数和关键字：

1.**参数**是用于更改命令结果的值。例如，在命令**show ip route 192.168.1.0**中，有一个**route**关键字，后跟一个接受IP地址的参数，该参数指定只显示包含所提供IP地址的路由。

2.关**键字**通过单独存在更改命令的结果。例如，在命令**show mac address-table dynamic**中，有一个**dynamic**关键字，该关键字指定仅显示动态获取的MAC地址。

创建NX-SDK命令时，这两个组件都在其语法中定义。NxCliCmd对象的方法可用于修改两个组件的具体实现。

- **nx_cmd.updateParam("<parameter>", "help_str", type)，其中nx_cmd是cliP.newShowCmd()或cliP.newConfigCmd()方法返回的NxCliCmd对象，<parameter>**是可以由其中括的命令参数的名称角括号(<>),**help_str设置自定义命令的帮助字符串，并显示在通过CLI上的问号访问的NX-OS上下文相关帮助菜单中，type是参数的类型。**此处提供并记录了此方法的其他可选参数。以下是可在type参数中指定的参数的有效类型：

  P_INTEGER — 指定任意整数**P_STRING** — 指定任何字符串**P_INTERFACE** — 指定任何网络接口**P_IP_ADDR. — 指**定任何IP地址**P_MAC_ADDR. — 指**定任何MAC地址**P_VRF** — 指定任何虚拟路由和转发(VRF)实例
- **nx_cmd.updateKeyword（"关键字", "help_str", is_key），其中nx_cmd是cliP.newShowCmd()或cliP.newConfigCmd()方法返回的NxCliCmd对象，关键字**是要修改的命令关键字的名称，**help_str**设置自定义命令的帮助字符串，并显示在通过CLI上的问号访问的NX-OS上下文相关帮助菜单中，而is_key是默认为False的可选布尔值。如果**is_key为**True，则使用此关键字的命令创建的唯一配置不会覆盖命令创建的其他唯一配置。如果**is_key为**False，则使用此关键字的命令创建的配置将覆盖命令创建的其他配置。此处记录了此方法。

要查看常用命令组件的代码示例，请查看本文档的自定义CLI命令示例部分。

在创建自定义CLI命令后，需要从本文档后面部分介绍的**pyCmdHandler**类创建对象，并将其设置为NxCliParser对象的CLI回调处理程序对象。演示如下：

```
 cmd_handler = pyCmdHandler()
cliP.setCmdHandler(cmd_handler)
```

然后，需要将NxCliParser对象添加到NX-OS CLI解析器树中，以便用户可以看到自定义CLI命令。这是使用**cliP.addToParseTree()命令完成的**，其中**cliP是sdk.getCliParser()方法返回的NxCliParser对象。**

sdkThread函数示例

以下是使用前面介绍的函数的典型sdkThread函数的示例。此函数（在典型的自定义NX-SDK Python应用中的其他函数）利用全局变量，这些变量在脚本执行时进行实例化。

```
cliP = ""
sdk = ""
event_hdlr = ""
```

```
tmsg = ""

def sdkThread():
    global cliP, sdk, event_hdlr, tmsg

    sdk = nx_sdk_py.NxSdk.getSdkInst(len(sys.argv), sys.argv)
    if not sdk:
       return

    sdk.setAppDesc("Returns all interfaces with DOM-capable transceivers inserted")

    tmsg = sdk.getTracer()
    tmsg.event("[{}] Started service".format(sdk.getAppName()))

    cliP = sdk.getCliParser()

    nxcmd = cliP.newShowCmd("show_port_bw_util_cmd", "port bw utilization [<port>]")
    nxcmd.updateKeyword("port", "Port Information")
    nxcmd.updateKeyword("bw", "Port Bandwidth Information")
    nxcmd.updateKeyword("utilization", "Port BW utilization in (%)")
    nxcmd.updateParam("<port>", "Optional Filter Port Ex) Ethernet1/1", nx_sdk_py.P_INTERFACE)

    nxcmd1 = cliP.newConfigCmd("port_bw_threshold_cmd", "port bw threshold <threshold>")
    nxcmd1.updateKeyword("threshold", "Port BW Threshold in (%)")

    int_attr = nx_sdk_py.cli_param_type_integer_attr()
    int_attr.min_val = 1;
    int_attr.max_val = 100;
    nxcmd1.updateParam("<threshold>", "Threshold Limit. Default 50%", nx_sdk_py.P_INTEGER,
int_attr, len(int_attr))

    mycmd = pyCmdHandler()
    cliP.setCmdHandler(mycmd)

    cliP.addToParseTree()

    sdk.startEventLoop()

    # If sdk.stopEventLoop() is called or application is removed from VSH...
    tmsg.event("Service Quitting...!")

    nx_sdk_py.NxSdk.__swig_destroy__(sdk)
```

pyCmdHandler类

pyCmdHandler**类**从nx_sdk_py库**中的**NxCmdHandler类继承。每当**从NX-SDK应用程**序发起的CLI命令时，都会调用**pyCmdHandler**类中定义的**postCliCb(self，clicmd)方法。**因此，**postCliCb(self，clicmd)**方法用于定义在sdkThread函数中定义的自定义CLI命令在设备上的行为。

postCliCb**(self， clicmd)函**数返回布尔值。如果**True**返回，则假定命令已成功执行。**如果**由于任何原因未成功执行命令，则应返回False。

clicmd参数使用在sdkThread函数中创建时为命令定义的唯一名称。例如，如果创建一个新的**show**命令，其唯一名称为**show_xcvr_dom**，则建议在检查clicmd参数的名称是否包含**show_xcvr_dom**后，在**postCliCb(self， clicmd)**函数中以相同名称引用此命令。此处演示：

```
def sdkThread():
    <snip>
    sh_xcvr_dom = cliP.newShowCmd("show_xcvr_dom", "dom")
```

```
    sh_xcvr_dom.updateKeyword("dom", "Show all interfaces with transceivers that are DOM-
capable")
    </snip>

class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        if "show_xcvr_dom" in clicmd.getCmdName():
            get_dom_capable_interfaces()
```

如果创建了使用参数的命令，则您很可能需要在postCliCb(self， clicmd)函数的某**个点使用这些**参数。这可以使用**clicmd.getParamValue("<parameter>")**方法完成，其中**<parameter>**是要获得用尖括号(<>)括起的值的命令参数的名称。 此处记录了此方法。但是，此函数返回的值需要转换为所需的类型。这可以通过以下方法实现：

- **nx_sdk_py.void_to_int将**值转换为整数类型。
- **nx_sdk_py.void_to_string将**值转换为字符串类型。

postCliCb**(self， clicmd)**功能（或任何后续功能）通常也将打印到控制台的show命令输出。这是使用clicmd.printConsole**()方法完**成的。

> **注意：**如果应用程序遇到错误、未处理的异常或突然退出，则**clicmd.printConsole()函**数的输出将不显示。因此，调试Python应用程序时的最佳做法是使用sdk.getTracer()方法返回的NxTrace对象将调试消息记录到系统日志中，或使用打印语句并通过Bash外壳的**/isan/bin/python二进制文件执行应**用。

pyCmdHandler类示例

以下代码用作上述pyCmdHandler**类**的示例。此代码取自此处提供的ip-movement NX-SDK应用程序中的ip_move.py文件。此应用的目的是跟踪用户定义的IP地址在Nexus设备接口间的移动。为此，代码在设备的ARP缓存中通过**<ip>**参数查找IP地址输入的MAC地址，然后使用设备的MAC地址表验证MAC地址驻留在哪个VLAN中。使用此MAC和VLAN，**show system internal l2fm l2dbg macdb address <mac> vlan <vlan>**命令显示此组合最近与之关联的SNMP接口索引列表。然后，代码使用**show interface snmp-ifindex**命令将最近的SNMP接口索引转换为易于人们理解的接口名称。

```
class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        global cli_parser

        if "show_ip_movement" in clicmd.getCmdName():
            target_ip = nx_sdk_py.void_to_string(clicmd.getParamValue("<ip>"))

            target_mac = get_mac_from_arp(cli_parser, clicmd, target_ip)
            mac_vlan = ""
            if target_mac:
                mac_vlan = get_vlan_from_cam(cli_parser, clicmd, target_mac)
                if mac_vlan:
                    find_mac_movement(cli_parser, clicmd, target_mac, mac_vlan)
                else:
                    print("No entires in MAC address table")
                    clicmd.printConsole("No entries in MAC address table for
{}".format(target_mac))
            else:
                clicmd.printConsole("No entries in ARP table for {}".format(target_ip))
        return True

def get_mac_from_arp(cli_parser, clicmd, target_ip):
    exec_cmd = "show ip arp {}".format(target_ip)
    arp_cmd = cli_parser.execShowCmd(exec_cmd, nx_sdk_py.R_JSON)
```

```
    if arp_cmd:
        try:
            arp_json = json.loads(arp_cmd)
        except ValueError as exc:
            return None
        count = int(arp_json["TABLE_vrf"]["ROW_vrf"]["cnt-total"])
        if count:
            intf = arp_json["TABLE_vrf"]["ROW_vrf"]["TABLE_adj"]["ROW_adj"]
            if intf.get("ip-addr-out") == target_ip:
                target_mac = intf["mac"]
                clicmd.printConsole("{} is currently present in ARP table, MAC address
{}\n".format(target_ip, target_mac))
                return target_mac
            else:
                return None
        else:
            return None
    else:
        return None

def get_vlan_from_cam(cli_parser, clicmd, target_mac):
    exec_cmd = "show mac address-table address {}".format(target_mac)
    mac_cmd = cli_parser.execShowCmd(exec_cmd, nx_sdk_py.R_JSON)
    if mac_cmd:
        try:
            cam_json = json.loads(mac_cmd)
        except ValueError as exc:
            return None
        mac_entry = cam_json["TABLE_mac_address"]["ROW_mac_address"]
        if mac_entry:
            if mac_entry["disp_mac_addr"] == target_mac:
                egress_intf = mac_entry["disp_port"]
                mac_vlan = mac_entry["disp_vlan"]
                clicmd.printConsole("{} is currently present in MAC address table on interface
{}, VLAN {}\n".format(target_mac, egress_intf, mac_vlan))
                return mac_vlan
            else:
                return None
        else:
            return None
    else:
        return None

def find_mac_movement(cli_parser, clicmd, target_mac, mac_vlan):
    exec_cmd = "show system internal l2fm l2dbg macdb address {} vlan {}".format(target_mac,
mac_vlan)
    l2fm_cmd = cli_parser.execShowCmd(exec_cmd)
    if l2fm_cmd:
        event_re = re.compile(r"^\s+(\w{3}) (\w{3}) (\d+) (\d{2}):(\d{2}):(\d{2}) (\d{4})
(0x\S{8}) (\d+)\s+(\S+) (\d+)\s+(\d+)\s+(\d+)")
        unique_interfaces = []
        l2fm_events = l2fm_cmd.splitlines()
        for line in l2fm_events:
            res = re.search(event_re, line)
            if res:
                day_name = res.group(1)
                month = res.group(2)
                day = res.group(3)
                hour = res.group(4)
                minute = res.group(5)
                second = res.group(6)
                year = res.group(7)
                if_index = res.group(8)
                db = res.group(9)
```

```
                event = res.group(10)
                src=res.group(11)
                slot = res.group(12)
                fe = res.group(13)
                if "MAC_NOTIF_AM_MOVE" in event:
                    timestamp = "{} {} {} {}:{}:{} {}".format(day_name, month, day, hour,
minute, second, year)
                    intf_dict = {"if_index": if_index, "timestamp": timestamp}
                    unique_interfaces.append(intf_dict)
        if not unique_interfaces:
            clicmd.printConsole("No entries for {} in L2FM L2DBG\n".format(target_mac))
        if len(unique_interfaces) == 1:
            clicmd.printConsole("{} has not been moving between
interfaces\n".format(target_mac))
        if len(unique_interfaces) > 1:
                clicmd.printConsole("{} has been moving between the following interfaces, from
most recent to least recent:\n".format(target_mac))
                unique_interfaces = get_snmp_intf_index(unique_interfaces)
                clicmd.printConsole("\t{} - {} (Current interface)\n".format(unique_interfaces[-
1]["timestamp"], unique_interfaces[-1]["intf_name"]))
                for intf in unique_interfaces[-2::-1]:
                    clicmd.printConsole("\t{} - {}\n".format(intf["timestamp"],
intf["intf_name"]))
def get_snmp_intf_index(if_index_dict_list): global cli_parser snmp_ifindex =
cli_parser.execShowCmd("show interface snmp-ifindex", nx_sdk_py.R_JSON) snmp_ifindex_json =
json.loads(snmp_ifindex) snmp_ifindex_list =
snmp_ifindex_json["TABLE_interface"]["ROW_interface"] for index_dict in if_index_dict_list:
index = index_dict["if_index"] for ifindex_json in snmp_ifindex_list: if index ==
ifindex_json["snmp-ifindex"]: index_dict["intf_name"] = ifindex_json["interface"] return
if_index_dict_list
```

自定义CLI命令语法示例

本节展示使用**cliP.newShowCmd()或cliP.newConfigCmd()**方法创建自定义CLI命令时使用的一些语法参数示例，其中**cliP是sdk.getCliParser()方法返回的NxCliParser对象。**

> **注意：**在NX-SDK v1.5.0中引入了对带有左括号和右括号(" ("和") ")的语法的支持，该版本包含在NX-OS版本7.0(3)I7(3)中。 假设用户在遵循以下任何示例时使用NX-SDK v1.5.0，包括使用左括号和右括号的语法。

单个关键字

此show命令采用单个关键字mac，并将帮助字符串Shows all programmed MAC addresses on this device to the keyword添加。

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed", "mac")
nx_cmd.updateKeyword("mac", "Shows all misprogrammed MAC addresses on this device")
```

单参数

此show命令采用单个参**数<mac>**。mac一词周围的尖括号表示这是参数。MAC地址的帮助字符串被添加到参数中以检查编程错误。**nx_cmd.updateParam()方法中的nx_sdk_py.P_MAC_ADDR.**参数用于将参数的类型定义为MAC地址，从而防止最终用户输入其他类型（如字符串、整数或IP地址）。

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed_mac", "<mac>")
```

```
nx_cmd.updateParam("<mac>", "MAC address to check for misprogramming", nx_sdk_py.P_MAC_ADDR)
```

可选关键字

此show命令可以选择使用单个关键字**[mac]**。mac一词周围的括号表示此关键字是可选的。帮助字符串显示此设备上所有编程错误的MAC地址已添加到关键字。

```
 nx_cmd = cliP.newShowCmd(  "show_misprogrammed_mac"  ,   "[mac]"  )
nx_cmd.updateKeyword( "mac" , "Shows all misprogrammed MAC addresses on this device" )
```

可选参数

此show命令可以选择采用单**个参数[<mac>]**。字< mac >旁的括号表示此参数是可选的。mac一词周围的尖括号表示这是参数。MAC地址的帮助字符串被添加到参数中以检查编程错误。**nx_cmd.updateParam()**方**法中的nx_sdk_py.P_MAC_ADDR.**参数用于将参数的类型定义为MAC地址，从而防止最终用户输入其他类型（如字符串、整数或IP地址）。

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed_mac", "[<mac>]")
nx_cmd.updateParam("<mac>", "MAC address to check for misprogramming", nx_sdk_py.P_MAC_ADDR)
```

单个关键字和参数

此show命令采用一个关键字mac，紧跟参数**<mac-address>**。mac-address一词周围的括号表示这是参数。关键字中添加了Check MAC address for misprogramming的帮助字符串。MAC地址的帮助字符串被添加到参数中以检查编程错误。**nx_cmd.updateParam()**方**法中的nx_sdk_py.P_MAC_ADDR.**参数用于将参数的类型定义为MAC地址，从而防止最终用户输入其他类型（如字符串、整数或IP地址）。

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed", "mac <mac-address>")
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming",
nx_sdk_py.P_MAC_ADDR)
```

多个关键字和参数

此show命令可以采用两个关键字之一，这两个关键字后面都有两个不同的参数。第一个关键字mac的参数为**<mac-address>**，第二个关键字ip的参数为**<ip-address>**。mac-address和ip-address两词的括号括起尖括号表示它们是参数。Check MAC地址的帮助字符串是否存在编程错误被添加到mac关键字。MAC地址的帮助字符串被添加到<mac-address>参**数中以检查**错误。**nx_cmd.updateParam()**方法**中的nx_sdk_py.P_MAC_ADDR.**参数用于将**<mac-address>**参数的类型定义为MAC地址，这会阻止最终用户输入其他类型（如字符串、整数或IP地址）。将检查IP地址的帮助字符串错误编程添加到ip关键字。IP地址的帮助字符串被添加到<ip-address>参**数中，以检查**错误。**nx_cmd.updateParam()**方**法中的nx_sdk_py.P_IP_ADDR.**参数用于将**<ip-address>**参数的类型定义为IP地址，这会阻止最终用户输入其他类型（如字符串、整数或IP地址）。

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed", "(mac <mac-address> | ip <ip-address>)")
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming",
nx_sdk_py.P_MAC_ADDR)
nx_cmd.updateKeyword("ip", "Check IP address for misprogramming")
nx_cmd.updateParam("<ip-address>", "IP address to check for misprogramming",
nx_sdk_py.P_IP_ADDR)
```

具有可选关键字的多个关键字和参数

此show命令可以采用两个关键字之一，这两个关键字后面都有两个不同的参数。第一关键字mac的参数为**<mac-address>**，第二关键字ip的参数为**<ip-address>**。mac-address和ip-address两词的括号括起尖括号表示它们是参数。Check MAC地址的帮助字符串是否存在编程错误被添加到mac关键字。MAC地址的帮助字符串被添加到<mac-address>参**数中以检查**错误。**nx_cmd.updateParam()**方法**中的nx_sdk_py.P_MAC_ADDR.**参数用于将**<mac-address>**参数的类型定义为MAC地址，这会阻止最终用户输入其他类型（如字符串、整数或IP地址）。将检查IP地址的帮助字符串错误编程添加到ip关键字。IP地址的帮助字符串被添加到<ip-address>参**数中，以检查**错误。**nx_cmd.updateParam()**方**法中的nx_sdk_py.P_IP_ADDR.**参数用于将**<ip-address>**参数的类型定义为IP地址，这会阻止最终用户输入其他类型（如字符串、整数或IP地址）。此show命令可能会选择使用关键字[clear]。帮助字符串清除检测到被错误编程的地址会添加到此可选关键字。

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed", "(mac <mac-address> | ip <ip-address>) [clear]")
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming",
nx_sdk_py.P_MAC_ADDR)
nx_cmd.updateKeyword("ip", "Check IP address for misprogramming")
nx_cmd.updateParam("<ip-address>", "IP address to check for misprogramming",
nx_sdk_py.P_IP_ADDR)
nx_cmd.updateKeyword("clear", "Clears addresses detected to be misprogrammed")
```

具有可选参数的多个关键字和参数

此show命令可以采用两个关键字之一，这两个关键字后面都有两个不同的参数。第一关键字mac的参数为**<mac-address>**，第二关键字ip的参数为**<ip-address>**。mac-address和ip-address两词的括号括起尖括号表示它们是参数。Check MAC address for misprograming的帮助字符串被添加到mac关键字。MAC地址的帮助字符串被添加到<mac-address>参**数中以检查**错误。**nx_cmd.updateParam()**方法**中的nx_sdk_py.P_MAC_ADDR.**参数用于将**<mac-address>**参数的类型定义为MAC地址，这会阻止最终用户输入其他类型（如字符串、整数或IP地址）。将检查IP地址的帮助字符串错误编程添加到ip关键字。IP地址的帮助字符串被添加到<ip-address>参**数中，以检查**错误。**nx_cmd.updateParam()**方**法中的nx_sdk_py.P_IP_ADDR.**参数用于将**<ip-address>**参数的类型定义为IP地址，这会阻止最终用户输入其他类型（如字符串、整数或IP地址）。此show命令可能会选择采**用参数[<module>]**。将帮助字符串仅清除指定模块上的地址添加到此可选参数。

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed", "(mac <mac-address> | ip <ip-address>)
[<module>]")
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming",
nx_sdk_py.P_MAC_ADDR)
nx_cmd.updateKeyword("ip", "Check IP address for misprogramming")
nx_cmd.updateParam("<ip-address>", "IP address to check for misprogramming",
nx_sdk_py.P_IP_ADDR)
nx_cmd.updateParam("<module>", "Clears addresses detected to be misprogrammed",
nx_sdk_py.P_INTEGER)
```

使用NX-SDK调试Python应用

创建NX-SDK Python应用后，通常需要对其进行调试。NX-SDK会通知您，以防代码中出现任何语法错误，但由于Python NX-SDK库使用SWIG将C++库转换为Python库，因此在代码执行时遇到的任何异常都会导致应用核心转储，如下所示：

```
 terminate called after throwing an instance of 'Swig::DirectorMethodException'
what(): SWIG director method error. Error detected when calling 'NxCmdHandler.postCliCb'
Aborted (core dumped)
```

由于此错误消息的性质不明确，调试Python应用程序的最佳做法是使用sdk.getTracer()方法返回的NxTrace对象将调试消息记**录到系统**日志。演示如下：

```
#! /isan/bin/python

tracer = 0

def evt_thread():
    <snip>
    tracer = sdk.getTracer()
    tracer.event("[NXSDK-APP][INFO] Started service")
<snip>
class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        global tracer
        tracer.event("[NXSDK-APP][DEBUG] Received command: {}".format(clicmd))
        if "show_test_command" in clicmd.getCmdName():
            tracer.event("[NXSDK-APP][DEBUG] `show_test_command` recognized")
```

如果将调试消息记录到系统日志不是选项，则另一种方法是使用打印语句并通过Bash外壳的**/isan/bin/python二进制文件执行应**用。但是，只有以这种方式执行时，这些打印语句的输出才可见 — 通过VSH外壳运行应用程序不会产生任何输出。以下是使用打印语句的示例：

```
#! /isan/bin/python

tracer = 0

def evt_thread():
    <snip>
    print("[NXSDK-APP][INFO] Started service")
<snip>
class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        print("[NXSDK-APP][DEBUG] Received command: {}".format(clicmd))
        if "show_test_command" in clicmd.getCmdName():
            print("[NXSDK-APP][DEBUG] `show_test_command` recognized")
```

使用NX-SDK部署Python应用

在Bash外壳中对Python应用程序进行了完全测试并准备好部署后，应通过VSH将该应用程序安装到生产环境中。这样，当设备重新加载或在双管理引擎场景中发生系统切换时，应用程序就可以持续。要通过VSH部署应用，需要使用NX-SDK和ENXOS SDK构建环境创建RPM包。Cisco DevNet提供Docker映像，可轻松创建RPM软件包。

注意：要在特定操作系统上安装Docker，请参阅Docker的安装文档。

在支持Docker的主机上，使用**docker pull dockercisco/nxsdk:<tag>** 命令提取您选择的映像版本，其中**<tag>**是您选择的映像版本的标记。您可以在此处查看可用映像版本及其相应标签。此处使用v1标**记进**行演示：

```
 docker pull dockercisco/nxsdk:v1
```

从此映像启动名为nxsdk的容器，并将其附加到该容器。如果您选择的标记不同，请用v1替换您的标记：

```
docker run -it --name nxsdk dockercisco/nxsdk:v1 /bin/bash
```

更新到NX-SDK的最新版本并导航到NX-SDK目录，然后从git中提取最新文件：

```
cd /NX-SDK/
git pull
```

如果需要使用NX-SDK的较旧版本，可以使用git clone -b v<version> https://github.com/CiscoDevNet/NX-SDK.git命令，其中<version>是您需要的NX-SDK版本。NX-SDK v1.0.0将在此演示：

```
cd /
rm -rf /NX-SDK
git clone -b v1.0.0 https://github.com/CiscoDevNet/NX-SDK.git
```

接下来，将Python应用传输到Docker容器。有几种不同的方法。

- 退出Docker容器（该容器停止容器并要求您再次启动它），将Python应用程序传输到Docker主机，然后使用docker cp命令将应用程序从主机复制到容器。此处演示的假设是，Python应用已传输到/app/python_app.py上的Docker主机。

```
root@2dcbe841742a:~# exit
[root@localhost ~]# docker cp /app/python_app.py nxsdk:/root/
[root@localhost ~]# docker start nxsdk
nxsdk
[root@localhost ~]# docker attach nxsdk
root@2dcbe841742a:/# ls /root/
python_app.py
```

- 将Python应用程序的内容复制到系统剪贴板，然后使用vim将内容粘贴到在Docker容器中创建的文件中。

接下来，使用/NX-SDK/scripts/中的rpm_gen.py脚本，以便从Python应用程序创建RPM包。此脚本有一个必需参数和两个必需的开关：

- Python应用的文件名。例如，在名为python_app.py的文件中使用Python应用程序将产生python_app.py参数。此文件名稍后将用作NX-SDK的应用程序名称，NX-OS也将使用此文件名来引用此应用程序创建的命令。

  注意：文件名不需要包含任何文件扩展名，如.py。在本示例中，如果文件名是python_app而不是python_app.py，则生成RPM包时不会出现问题。

- -s开关为指向上述文件名所在位置的绝对文件路径使用参数。例如，如果python_app.py位于/root/中，则正确的参数是 — s /root/。
- -u开关表示源文件名与可执行文件名相同。

此处演示了rpm_gen.py脚本的用法。

```
root@7bfd1714dd2f:~# python /NX-SDK/scripts/rpm_gen.py test_python_app -s /root/ -u
##################################################################################################
####
```

```
Generating rpm package...
<snip>
RPM package has been built
##############################################################################################
####

SPEC file: /NX-SDK/rpm/SPECS/test_python_app.spec
RPM file : /NX-SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm
```

RPM包的文件路径在rpm_gen.py脚本输出的**最后一行**中指示。必须将此文件从Docker容器复制到主机上，以便可以将其传输到您要在其上运行应用的Nexus设备。退出Docker容器后，可以使用docker cp <container>:<container_filepath> <host_filepath>命令轻松完成，其中<container>是NX-SDK Docker容器（本例中为nxsdk）的名称，<container_filepath>是容器内RPM包的完整文件路径(在本例中,**/NX-SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm**)，以及**<host_filepath>是Docker主机上将RPM包传输到的完整文件路径（在本例中为/root/）。** 此命令在下面演示：

```
root@7bfd1714dd2f:/# exit
[root@localhost ~]# docker cp nxsdk:/NX-SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm /root/
[root@localhost ~]# ls /root/
anaconda-ks.cfg             test_python_app-1.0-1.0.0.x86_64.rpm
```

使用首选的文件传输方法将此RPM包传输到Nexus设备。一旦RPM软件包在设备上，必须安装并激活它，与SMU类似。假设RPM软件包已传输到设备的bootflash，则演示如下。

```
N9K-C93180LC-EX# install add bootflash:test_python_app-1.0-1.0.0.x86_64.rpm
[####################] 100%
Install operation 27 completed successfully at Tue May  8 06:40:13 2018
N9K-C93180LC-EX# install activate test_python_app-1.0-1.0.0.x86_64
[####################] 100%
Install operation 28 completed successfully at Tue May  8 06:40:20 2018
```

> **注意：**使用install add命令安装RPM包时，**请包**括存储设备和包的确切文件名。安装后激活RPM软件包时，请勿包含存储设备和文件名 — 使用软件包本身的名称。您可以使用show install inactive命令验**证软件包**名称。

激活RPM包后，可以使用**nxsdk service <application-name>** 配置命令使用NX-SDK启动应用程序，其中**<application-name>**是之前使用rpm_gen.py脚本时定义的Python文件名（以及随后的应用程序）的名称。演示如下：

```
N9K-C93180LC-EX# conf
Enter configuration commands, one per line. End with CNTL/Z.
N9K-C93180LC-EX(config)# nxsdk service-name test_python_app
% This could take some time. "show nxsdk internal service" to check if your App is Started &
Runnning
```

您可以使用show nxsdk internal service命令验证应用程序是否已启动并**已开始运**行：

```
N9K-C93180LC-EX# show nxsdk internal service

NXSDK Started/Temp unavailabe/Max services : 1/0/32
NXSDK Default App Path         : /isan/bin/nxsdk
NXSDK Supported Versions       : 1.0

Service-name              Base App        Started(PID)      Version    RPM Package
------------------------- --------------- ----------------- ---------- ------------------------
test_python_app           nxsdk_app4      VSH(23195)        1.0        test_python_app-1.0-
1.0.0.x86_64
```

您还可以验证此应用程序创建的自定义CLI命令在NX-OS中是否可访问：

```
N9K-C93180LC-EX# show test?
  test_python_app    Nexus Sdk Application
```

相关信息

- NX-SDK GitHub
- Cisco Nexus 9000系列NX-OS可编程性指南，版本7.x
- Cisco Nexus 3000系列NX-OS可编程性指南，版本7.x
- Cisco Nexus 3500系列NX-OS可编程性指南，版本7.x
- Cisco Nexus 9000系列交换机的网络可编程性和自动化白皮书
- 通过Cisco Open NX-OS实现可编程性和自动化(PDF)
- 技术支持和文档 - Cisco Systems