开发，调试并且实施NX-SDK在连结的Python应用程序3000/9000交换机

目录

简介

本文为Python应用发展提供工作流Cisco NX软件发展工具包(SDK)使用Cisco NX-OS在连结3000和连结9000平台。

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：

- 本文使用NX-SDK v1.0.0和NX-SDK v1.5.0
- NX-SDK v1.0.0在连结9000平台由NX-OS版本7.0(3)I6(1)开始和连结3000平台可以使用由NX-OS版本7.0(3)I7(1)开始
- NX-SDK v1.5.0在连结9000平台和连结3000平台可以使用由NX-OS版本7.0(3)I7(3)开始

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允许在连结9000和连结的Cisco NX-OS能本地运行3000平台定制应用的发展。NX-SDK提供用户的能力能创建他们自己的CLI命令，并且输出，生成定制的Syslog以回应特定事件，流被剪裁的远测术和更多。

NX-SDK有a.c. ++ API，被转换成其他语言与使用简化的封皮和接口生成器(痛饮)。这允许用户使用NX-SDK用他们的选择所有语言。本文展示普通的NX-SDK功能的实施在Python的，以及为用户提供工作流开发他们自己的NX-SDK Python应用程序。

开发与NX-SDK的Python应用程序

Enable (event) NX-SDK

为了所有NX-SDK应用程序能运行，在设备必须首先启用NX-SDK功能：

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

创建Python文件

Python文件可以创建和编辑与使用NX-OS打击shell。为了能将使用的打击shell，在设备必须首先启用它：

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

输入打击shell并且请使用vi文本编辑为了创建和编辑Python文件：

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

> **Note**:最佳实践是创建Python文件在**/isan/bin/**目录里。Python文件需要执行权限为了运行-请勿安置Python文件在**/bootflash**目录或任何其子目录。

> **Note**:没有要求通过NX-OS创建和编辑Python文件。使用他们的选择，文件传输协议开发者可能创建应用程序使用他们的本地环境和调用完成文件到设备。然而，也许是效率更高开发者的能调试和排除他们的脚本故障使用NX-OS工具。

实现NX-SDK组件

是推荐的为了开发员能开始创建他们的从customCliPyApp模板的NX-SDK Python应用程序在 Cisco DevNet NX-SDK GitHub。 本文的这些部分是指与使用的必要的组件他们的在此模板内的名字。

有四个主要组件必要为NX-SDK Python应用程序：

1. 必须导入NX-SDK到应用程序通过**导入nx_sdk_py**语句。
2. 该的功能(典型地已命名**sdkThread**)运行NX-SDK应用程序并且修改多种选项与应用程序有关。
3. 自定义CLI命令的创建以及自定义CLI命令语法的定义在**sdkThread内的**作用。
4. 名为与方法的**pyCmdHandler的**组命名了**postCliCb**，处理NX-SDK应用程序添加的自定义CLI命令。

sdkThread功能

sdkThread功能初始化，添加功能对，并且运行NX-SDK应用程序。功能不要求任何参数通过到它。所有Python NX-SDK应用程序要求从nx_sdk_py库的三个方法称为：

1. **nx_sdk_py.NxSdk.getSdkInst什么都(len(sys.argv)， sys.argv)不返回SDK实例**对象或者返回**。如果SDK实例对象返回，则NX-SDK应用程序成功注册与NX-OS基础设施。如果什么都没有返回，则错误在此注册过程时出现，并且错误日志条目出现于设备的Syslog。此方法描述得这里。**

   Note:从NX-SDK v1.5.0开始，第三个布尔型参数可以通过到**NxSdk.getSdkInst方法**， enable (event)提前例外，当真和功能失效提前例外，当错误。此方法描述得这里。

1. **sdk.startEventLoop ()方法**，其中sdkis SDK实例对象用**nx_sdk_py.NxSdk.getSdkInst ()方法**返回了**。此方法**运行NX-SDK应用程序并且允许它与NX-OS基础设施呼应。此方法描述得这里。
2. **nx_sdk_py.NxSdk. __swig_destroy__(sdk)**方法，其中**sdk是nx_sdk_py.NxSdk.getSdkInst ()方法**返回的SDK实例对象以前**解释**。此方法被放置在NX-SDK应用程序的从容退出的sdkThread functionallows结束时。

一些常用的方法包括：

- **sdk.getTracer ()，其中**sdkis SDK实例对象用**nx_sdk_py.NxSdk.getSdkInst ()方法**返回了**。此方法**返回可以用于生成自定义Syslog的NxTrace对象，以及日志事件和错误到应用程序事件历史记录。自定义Syslog将出现于设备的Syslog (可视通过show logging logfile**命令)，当事件**被记录对应用程序事件历史记录通过显示<application-name> **nxsdk事件历史记录事件时**是可视的**或显示<application-name> nxsdk事件历史记录错误命令。此方法**描述得这里。NxTrace以这个方法objectreturned的和其相关的方法描述得这里。例如， youcan视图名为Transceiver_DOM.py throughthe的应用程序的事件历史记录**显示Transceiver_DOM.py** nxsdkevent历史记录事件并且显示Transceiver_DOM.py nxsdk事件**历史记录错误命令。**
- **sdk.getCliParser ()，其中**sdkis SDK实例对象用**nx_sdk_py.NxSdk.getSdkInst ()方法**返回了**。此方法**返回NxCliParser对象，可以用于执行CLI发出命令通过Python已经存在以及创建自定义CLI命令。此方法描述得这里。NxCliParserobject以这个方法返回的和其相关的方法描述得这里。

- **cliP.execShowCmd (“cmd”， return_type)，其中**cliPis sdk.getCliParser **()方法**返回的NxCliParser对象， cmd show命令您在引号希望执行封装，并且return_type**是**将输出的数据格式。**数据格式可以是R_TEXT、R_JSON或者**R_XML**。此**方法描述得这里。

  Note:如果命令支持输出以那些格式， **R_JSON**和**R_XML**数据格式只运作。在NX-OS中，您能验证命令是否通过管道传送输出支持输出以特定的数据格式对被请求的数据格式。如果管道传送的命令返回有意义的输出，则支持该数据格式。例如，如果运行**动态的show mac address-table|**在NX-OS的**json**返回JSON输出，然后NX-SDK支持R_JSON数据格式。

- 对文件的绝对文件路径包含命令由线路分离的**cliP.execConfigCmd(cmd_filename)，**其中cliPis NxCliParser对象用sdk.getCliParser **()方法**返回了和cmd_filenameis。**此方法返回指示命令执行的成功-的字符串，如果“成功”在字符串，然后所有命令顺利地被执行。否则，字符串包含描述的例外命令为什么没有能执行。此方法描述得这里。**

可以是有用的某些可选的方法是：

- **sdk.setAppDesc (‘description string’)，其中**sdkis SDK实例对象用**nx_sdk_py.NxSdk.getSdkInst ()方法**返回了**。此方法**设置NX-SDK应用程序的说明。说明在通过在CLI的一个问号被获取的NX-OS敏感内容Help菜单显示。此方法描述得这里。例如， anapplication命名了Transceiver_DOM.py，所有接口用插入的DOM能够收发器出现于NX-OS上下文相关的帮助回归的**应用程序说明**如下：

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

- **sdk.getAppName ()，其中**sdkis SDK实例对象用**nx_sdk_py.NxSdk.getSdkinst ()方法**返回了**。此方法**返回Python应用程序的名字。此方法描述得这里。

创建自定义CLI命令

在与使用的一个Python应用程序NX-SDK，自定义CLI命令在sdkThread功能内被创建并且被定义。有命令的两种类型：**显示**命令和**设置**命令。

1. **显示**关于设备、其配置，或者其环境的display命令信息。
2. **设置**命令更改设备配置，修改设备如何起反应对周围的网络。

这两个方法允许创建分别显示命令和设置命令：

- **cliP.newShowCmd (“cmd_name”， “语法”)，其中**cliPis NxCliParser对象由sdk.getCliParser **()方法**、cmd_nameis**一个**唯一名字对于internal命令自定义NX-SDK应用程序和syntaxdescribes回到了什么关键字和参数可以用于命令。此方法返回NxCliCmd对象，描述得这里。此方法描述得这里。

  Note:此命令是子类**cliP.newCliCmd (“cmd_type”， “cmd_name”， “语法”)** cmd_type**是**CONF_CMD**或**SHOW_CMD的地方**(根据**被配置的命令的种类)， cmd_name**是**一个唯一名字对于internal命令对自定义NX-SDK应用程序和syntaxdescribes什么关键字和参数可以用于命令。因此，此命令的APIDOCUMENTATION也许是有用供参考。

- **cliP.newConfigCmd (“cmd_name”， “语法”)，其中**cliPis NxCliParser对象由sdk.getCliParser **()方法**、cmd_nameis**一个**唯一名字对于internal命令自定义NX-SDK应用程序和syntaxdescribes回到了什么关键字和参数可以用于命令。此方法返回NxCliCmd对象，描述得这

里。此方法描述得这里。

Note:此命令是子类cliP.newCliCmd (“cmd_type”， “cmd_name”， “语法”) cmd_type是CONF_CMD或SHOW_CMD的地方(取决于配置)命令的种类， cmd_name是一个唯一名字对于internal命令对自定义NX-SDK应用程序和syntaxdescribes什么关键字和参数可以用于命令。因此，此命令的APIDOCUMENTATION也许是有用供参考。

命令的两个类型有两个不同的组件：参数和关键字：

1. 参数是用于的值更改命令的结果。例如，在show ip route命令192.168.1.0，有接受一个IP地址，指定的参数跟随的路由关键字包括应该显示仅的路由提供的IP地址。

2. 关键字通过单独他们的存在更改命令的结果。例如，在show mac address-table dynamic命令，有一个动态关键字，指定仅动态学习的MAC地址将显示。

当被创建时，两个组件在NX-SDK命令的语法被定义。NxCliCmd对象的方法存在修改两个组件的特定实施。

- nx_cmd.updateParam (“<parameter>”， “help_str”，类型)，其中nx_cmd是cliP.newShowCmd返回的NxCliCmd对象()或cliP.newConfigCmd ()方法， <parameter>是在通过在CLI的一个问号被获取的NX-OS敏感内容Help菜单可以被尖括号命令参数的名字(<>)修改围绕， help_strsets帮助字符串custom命令和显示和类型是参数的种类。此方法的另外的可选参数是可用和描述这里。在类型参数可以指定的参数的Theseare有效类型：

  P_INTEGER -指定任何整数P_STRING -指定所有字符串P_INTERFACE -指定所有网络接口P_IP_ADDR -指定所有IP地址P_MAC_ADDR -指定所有MAC地址P_VRF -指定所有虚拟路由和转发(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，关键字是您希望修改命令关键字，help_strsets的名字帮助字符串custom命令和显示按默认为错误的通过在CLI的一个问号被获取的NX-OS敏感内容Help菜单和is_key isan可选的布尔值。如果is_key isTrue，命令创建的独特配置用使用此关键字然后不重写是由命令创建的其他独特配置。如果is_key isFalse，命令创建的配置用使用此关键字然后重写命令创建的另一种配置。此方法描述得这里。

为了查看常用命令组件代码示例，查看本文的自定义CLI命令示例部分。

在自定义CLI命令被创建了后，从在本文以后描述的pyCmdHandler组的一个对象需要被创建和设置作为NxCliParser对象的CLI回拨处理程序对象。这被展示如下：

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

然后， NxCliParser对象需要被添加到NX-OS CLI分析程序树，以便自定义CLI命令是可视的对用户。这实行同cliP.addToParseTree ()命令， cliPis NxCliParser对象用sdk.getCliParser ()方法返回。

sdkThread功能示例

这是一个典型的sdkThread功能的示例与以前解释的使用的功能。此功能(除了别的以外在一个典型的自定义NX-SDK Python应用程序内)使用全局变量，是例示的在脚本执行。

```
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组

pyCmdHandler组从在nx_sdk_py库内的**NxCmdHandler**组被继承。在**pyCmdHandler**组内被定义的**postCliCb (自己， clicmd)**方法称为，每当起源于NX-SDK应用程序的CLI命令。因此， **postCliCb (自己， clicmd)**方法是您定义了的地方在sdkThread功能内被定义的自定义CLI命令如何在设备正常运行。

postCliCb (自己， clicmd)功能返回布尔值。如果**真**返回，则被假定命令成功执行。如果命令因故，没有成功执行**错误**应该返回。

clicmd参数使用为命令被定义的唯一名字，当在sdkThread功能被创建了。例如，如果用**show_xcvr_dom的**一个唯一名字创建一新**show命令**，然后它由在**postCliCb (自己， clicmd)**功能的同一个名字推荐是指此命令，在您检查发现后clicmd参数的名字是否包含**show_xcvr_dom**。被展示

得这里：

```
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()
```

如果使用参数的命令被创建，则您很可能将需要使用那些参数在某种程度上**postCliCb (自己， clicmd)**功能。这可以执行与**clicmd.getParamValue ("<parameter>")方法**， <parameter>是命令参数的名字您希望由尖括号(<>)获得值的围绕。此方法描述得这里。然而，值由此功能返回了需要被转换成您需要的类型。这可以执行与这些方法：

- **nx_sdk_py.void_to_int变换**值成整数类型。
- **nx_sdk_py.void_to_string变换**值成体串的类型。

postCliCb (自己， clicmd)功能(或任何随后的功能)也典型地将是show命令输出被打印到控制台的地方。这执行与**clicmd.printConsole ()方法**。

> **Note**:如果应用程序突然遇到错误、未处理的例外情况或者退出，则**clicmd.printConsole ()功能的**输出根本不会显示。**为此，最佳实践，当您调试您的Python应用程序时是对任一个日志调试消息对与sdk.getTracer ()方法返回的使用的Syslog NxTrace对象**或者使用打印语句并且通过打击shell的/isan/bin/python**二进制**执行应用程序。

pyCmdHandler组示例

以下代码服务例如如上所述的**pyCmdHandler**组。此代码从在可用IP移动NX-SDK的应用程序的ip_move.py文件被采取这里。此应用程序的目的将跟踪一个用户定义的IP地址的移动在连结设备的接口的间。要执行此，代码查找通过在设备的ARP高速缓存内的**<ip>参数**被输入的IP地址的MAC地址，然后验证哪个VLAN MAC地址位于使用设备的MAC地址表。使用此MAC和VLAN， show system**内部l2fm l2dbg macdb地址<mac> VLAN <vlan>命令**显示此组合最近产生关联SNMP接口索引的列表。代码然后使用show interface SNMP **IfIndex命令**翻译最近SNMP接口索引成人易读接口名字。

```
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

此部分陈列使用的语法参数的一些示例，当您用**cliP.newShowCmd ()时或**cliP.newConfigCmd **()方法**创建自定义CLI命令，夹子**是**sdk.getCliParser **()方法**返回的NxCliParser对象**。**

> **Note**:语法的技术支持与空缺数目和关闭的括号("(”并且")”)在NX-SDK v1.5.0被引入，包括在NX-OS版本7.0(3)I7(3)。假设，用户使用NX-SDK v1.5.0，当他们按照包括使用空缺数目和结束括号的语法这些提供的示例中的任一个时。

单个关键字

此show命令采取单个关键字mac并且添加Shows辅助字符串在此设备的所有编程错误MAC地址到关键字。

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

单个参数

此show命令采取单个参数**<mac>**。在词mac附近的放入的尖括号表示这是参数。检查的MAC地址辅助字符串错误程序设计被添加到参数。在**nx_cmd.updateParam ()方法的nx_sdk_py。P_MAC_ADDR**参数用于定义参数的种类作为MAC地址，防止另一种类型终端用户输入，例如字符串、整数或者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命令可能可选地采取单个关键字**[mac]**。在词mac附近的放入的托架表示此关键字是可选的。shows辅助字符串在此设备的所有编程错误MAC地址被添加到关键字。

```
 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命令可能可选地采取单个参数**[<mac>]**。在词< mac附近的>放入的托架表示此参数是可选的。在词mac附近的放入的尖括号表示这是参数。检查的MAC地址辅助字符串错误程序设计被添加到参数。在**nx_cmd.updateParam ()方法的nx_sdk_py。P_MAC_ADDR**参数用于定义参数的种类作为MAC地址，防止另一种类型终端用户输入，例如字符串、整数或者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命令采取参数**<mac-address>**立即跟随的单个关键字mac。在词MAC地址附近的放入的尖括号表示这是参数。检查MAC地址辅助字符串错误程序设计的被添加到关键字。检查的MAC地址辅助字符串错误程序设计被添加到参数。在**nx_cmd.updateParam ()方法的nx_sdk_py。P_MAC_ADDR**参数用于为了定义参数的种类作为MAC地址，防止另一种类型终端用户输入，例如字符串、整数或者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命令能采取两个关键字之一，其中之二有跟随他们的两个不同的参数。第一关键字mac有**<mac-address>**参数，并且第二关键字ip有**<ip-address>**参数。在词MAC地址和IP地址附近的放入的尖括号表示他们是参数。检查MAC地址辅助字符串错误程序设计的被添加到mac关键字。检查的MAC地址辅助字符串错误程序设计被添加到**<mac-address>**参数。在**nx_cmd.updateParam ()方法的nx_sdk_py。P_MAC_ADDR**参数用于定义**<mac-address>参数的**种类作为MAC地址，防止另一种类型终端用户输入，例如字符串、整数或者IP地址。**检查IP地址辅助字符串misprograming的被添加到ip关键字。检查的IP地址辅助字符串错误程序设计被添加到<ip-address>参数**。在nx_cmd.updateParam ()方法的nx_sdk_py。P_IP_ADDR参数用于定义**<ip-address>参数的**种类作为IP地址，防止另一种类型终端用户输入，例如字符串、整数或者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)
```

多个关键字和参数与可选关键字

此show命令能采取两个关键字之一，其中之二有跟随他们的两个不同的参数。第一关键字mac有**<mac-address>**参数，并且第二关键字ip有**<ip-address>**参数。在词MAC地址和IP地址附近的放入的尖括号表示他们是参数。检查MAC地址辅助字符串错误程序设计的被添加到mac关键字。检查的MAC地址辅助字符串错误程序设计被添加到**<mac-address>**参数。在**nx_cmd.updateParam ()方法的nx_sdk_py。P_MAC_ADDR**参数用于定义**<mac-address>参数的**种类作为MAC地址，防止另一种类型终端用户输入，例如字符串、整数或者IP地址。**检查IP地址辅助字符串misprograming的被添加到ip关键字。检查的IP地址辅助字符串错误程序设计被添加到<ip-address>参数**。在nx_cmd.updateParam ()方法的nx_sdk_py。P_IP_ADDR参数用于定义**<ip-address>参数的**种类作为IP地址，防止另一种类型终端用户输入，例如字符串、整数或者IP地址。**此show命令威力可选地采取关键字[clear]。辅助字符串清除被发现的地址是编程错误被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")
```

多个关键字和参数与可选参数

此show命令能采取两个关键字之一，其中之二有跟随他们的两个不同的参数。第一关键字mac有**<mac-address>**参数，并且第二关键字ip有**<ip-address>**参数。在词MAC地址和IP地址附近的放入的尖括号表示他们是参数。检查MAC地址辅助字符串misprogrammingis的被添加到mac关键字。检查的MAC地址辅助字符串错误程序设计被添加到**<mac-address>**参数。在**nx_cmd.updateParam ()方法的nx_sdk_py。P_MAC_ADDR**参数用于定义**<mac-address>参数的**种类作为MAC地址，防止另一种类型终端用户输入，例如字符串、整数或者IP地址。**检查IP地址辅助字符串misprograming的被添加到ip关键字。检查的IP地址辅助字符串错误程序设计被添加到<ip-address>参数**。在nx_cmd.updateParam ()方法的nx_sdk_py。P_IP_ADDR参数用于定义**<ip-address>参数的**种类作为IP地址，防止另一种类型终端用户输入，例如字符串、整数或者IP地址。**此show命令威力可选地采取参数[<module>]。**在指定的模块的仅**辅助**字符串清楚的地址被添加到此可选参数。

```
 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)
```

调试与NX-SDK的Python应用程序

一旦NX-SDK Python应用程序被创建了，经常将需要调试。NX-SDK通知您，万一有所有语法错误用您的代码，但是，因为Python NX-SDK库使用痛饮转换C++库为Python库，在编码执行时遇到的任何例外导致应用程序核心转储类似于此：

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

由于此错误信息的模棱两可的本质，最佳实践调试Python应用程序是记录调试消息对与**sdk.getTracer ()方法**返回的使用的Syslog NxTrace对象。**这被展示如下：**

```
#! /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")
```

如果记录消息的调试对Syslog不是选项，一个备选方法将使用打印语句和执行应用程序通过打击shell的**/isan/bin/python**二进制。然而，这些打印语句的输出只将是可视的，当如此执行-运行应用程序通过VSH shell不生成任何输出。使用打印语句示例显示得这里：

```
#! /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")
```

实施与NX-SDK的Python应用程序

一旦Python应用程序在打击shell充分地测试了并且准备好配置，应该安装应用程序到生产通过VSH。这允许应用程序仍然存在，当设备重新加载或，当系统切换在双重Supervisor方案发生。为了通过VSH实施应用程序，您需要用使用NX-SDK和ENXOS SDK修造环境创建RPM程序包。Cisco DevNet提供允许容易的RPM程序包创建的一个码头工人镜像。

Note:关于协助为了在您特定操作系统上安装码头工人，请参见码头工人的安装文档。

在一台码头工人能够主机上，请拉您的与**码头工人下拉式dockercisco/nxsdk的**选择的镜像版本**：<tag>**命令，其中**<tag>**是您的选择的镜像版本的标记。您能查看可用的镜像版本和他们的对应的标签这里。这展示与这里**v1**标记：

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

启动名为**nxsdk的**容器从此镜像并且附有它。如果您的选择标记是不同的，用您的标记请替代**v1** ：

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

对NX-SDK新版本的更新和连接对**NX-SDK**目录，然后拉从git的最新的文件：

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

如果需要使用NX-SDK早版本，您能克隆与使用的NX-SDK分组各自版本标记与**git克隆- b v<version>** https://github.com/CiscoDevNet/NX-SDK.git命令， **<version>**是NX-SDK的版本您需要。这展示这里与NX-SDK v1.0.0 ：

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

其次，请调用您的Python应用程序到码头工人容器。有一些个不同的方式执行此。

- 退出终止容器并且要求您更加开始它)的码头工人容器(调用Python应用程序到码头工人主机，然后请使用**码头工人cp**命令为了从主机复制应用程序到容器。以为Python应用程序调用了到在**/app/python_app.py的**码头工人主机这被展示得这里。

```
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
```

- 复制Python应用程序的内容到您的系统剪贴板，然后粘贴内容到在码头工人容器创建的文件用使用精力。

其次，请使用位于/NX-SDK/scripts/的**rpm_gen.py脚本为了**创建从Python应用程序的一个RPM程序包。**此脚本有一个必需参数，并且两要求了交换机：**

- Python应用程序的文件名。例如，在名为**python_app.py的**文件的一个Python应用程序将导致python_app.py的参数**。此**文件名以后将使用作为应用程序名称NX-SDK和由NX-OS也用于为了是指此应用程序创建的命令。

  Note:文件名不需要包含任何文件扩展，例如**.py**。在本例中，如果文件名是**python_app**而不是**python_app.py**， RPM程序包将生成，不用问题。

- -那导致的s交换机采取绝对文件路径的一个参数找出的地方上述文件名。例如，如果

python_app.py位于/root/，然后正确的参数是- s /root/。

- - u交换机表明源文件名是相同的象可执行的文件名。

rpm_gen.py脚本的使用方法被展示得这里。

```
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
```

对RPM程序包的文件路径在rpm_gen.py脚本输出的最终线路指示。必须复制此文件在主机上的码头工人容器，以便可以调用到连结设备您希望运行应用程序。在您退出码头工人容器后，可以容易地执行与dockercp <container> ：<container_filepath> <host_filepath>命令， <container>是NX-SDK码头工人容器的名字(在这种情况下， nxsdk)， <container_filepath>是RPM程序包的充分的文件路径在容器里面(在这种情况下， /NX-SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm)，并且<host_filepath>是在RPM程序包将调用到的我们的码头工人主机的充分的文件路径(在这种情况下， /root/)。此命令被展示得这里：

```
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
```

调用此RPM程序包到有使用的连结设备文件传输您的首选的方法。一旦RPM程序包在设备，必须类似安装和激活于SMU。以为RPM程序包调用了到设备的Bootflash，这被展示得如下。

```
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
```

> Note:当您安装RPM程序包用install add命令时，请包括程序包的存储设备和确切的文件名。当您在安装以后时激活RPM程序包，请勿包括存储设备和文件名-请使用程序包的名字。您能验证数据包名称用show install非激活命令。

一旦激活RPM程序包，您能运行与NX-SDK的应用程序用nxsdk服务<application-name>配置命令，<application-name>是被定义Python文件名(并且，随后，应用程序的)的名字，当使用了rpm_gen.py脚本前。这被展示如下：

```
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
```

您能验证应用程序启用和开始运行以显示nxsdk内部服务订单：

```
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
```

您能也验证此应用程序创建的自定义CLI命令是可访问的在NX-OS ：

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

相关信息

- NX-SDK GitHub
- Cisco连结9000系列NX-OS可编程序性指南，版本7.x
- Cisco连结3000系列NX-OS可编程序性指南，版本7.x
- Cisco连结3500系列NX-OS可编程序性指南，版本7.x
- 网络可编程序性和自动化与Cisco连结9000系列交换机白皮书
- 可编程序性和自动化与Cisco开放NX-OS (PDF)
- **技术支持和文档 - Cisco Systems**