Desenvolver, Depurar e Implantar o Aplicativo Python NX-SDK nos Switches Nexus 3000/9000

Contents

Introduction

Este documento fornece um fluxo de trabalho para o desenvolvimento de aplicativos Python com o Cisco NX-Software Development Kit (SDK) com o uso do Cisco NX-OS nas plataformas Nexus 3000 e Nexus 9000.

Prerequisites

Requirements

Não existem requisitos específicos para este documento.

Componentes Utilizados

As informações neste documento são baseadas nestas versões de software e hardware:

- Este documento utiliza NX-SDK v1.0.0 e NX-SDK v1.5.0
- O NX-SDK v1.0.0 pode ser utilizado na plataforma Nexus 9000 a partir do NX-OS versão 7.0(3)I6(1) e da plataforma Nexus 3000 a partir do NX-OS versão 7.0(3)I7(1)
- O NX-SDK v1.5.0 pode ser utilizado na plataforma Nexus 9000 e na plataforma Nexus 3000 a partir do NX-OS versão 7.0(3)I7(3)

The information in this document was created from the devices in a specific lab environment. All of the devices used in this document started with a cleared (default) configuration. If your network is live, make sure that you understand the potential impact of any command.

Informações de Apoio

O Cisco NX-SDK permite o desenvolvimento de aplicativos personalizados que podem ser executados nativamente no Cisco NX-OS nas plataformas Nexus 9000 e Nexus 3000. O NX-SDK oferece a capacidade de os clientes criarem seus próprios comandos e saídas CLI, gerarem syslogs personalizados em resposta a eventos específicos, telemetria de fluxo personalizada e muito mais.

O NX-SDK tem uma API C++, que é traduzida para outros idiomas com o uso do Simplified Wrapper and Interface Generator (SWIG). Isso permite que o cliente utilize o NX-SDK em qualquer idioma de sua escolha. Este documento demonstra a implementação de funções NX-SDK comuns em Python, bem como fornece um fluxo de trabalho para os clientes desenvolverem seus próprios aplicativos NX-SDK Python.

Desenvolver um aplicativo Python com NX-SDK

Habilitar NX-SDK

Para que qualquer aplicativo NX-SDK seja executado, o recurso NX-SDK deve primeiro ser ativado no dispositivo:

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

Criar um arquivo Python

Um arquivo Python pode ser criado e editado com o uso do shell do NX-OS Bash. Para que o shell Bash seja usado, ele deve primeiro ser ativado no dispositivo:

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

Insira o shell Bash e use o editor de texto vi para criar e editar o arquivo Python:

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

> **Note**: A melhor prática é criar arquivos Python no diretório **/isan/bin/**. Os arquivos Python precisam de permissões de execução para serem executados - não coloque os arquivos

Python no diretório **/bootflash** ou em nenhum de seus subdiretórios.

Note: Não é necessário criar e editar arquivos Python por meio do NX-OS. O desenvolvedor pode criar o aplicativo usando seu ambiente local e transferir os arquivos concluídos para o dispositivo usando um protocolo de transferência de arquivos de sua escolha. No entanto, pode ser mais eficiente para o desenvolvedor depurar e solucionar problemas de seu script usando utilitários NX-OS.

Implementar componentes do NX-SDK

Recomenda-se que os desenvolvedores comecem a criar seu aplicativo NX-SDK Python a partir do modelo customCliPyApp no Cisco DevNet NX-SDK GitHub. Estas seções deste documento se referem aos componentes necessários com o uso de seus nomes dentro deste modelo.

Há quatro componentes principais necessários para um aplicativo NX-SDK Python:

1. O NX-SDK deve ser importado para o aplicativo por meio da instrução **import nx_sdk_py**.
2. Uma função (normalmente denominada **sdkThread**) que inicia o aplicativo NX-SDK e modifica várias opções relacionadas ao aplicativo.
3. A criação de comandos CLI personalizados, bem como a definição da sintaxe de comando CLI personalizada dentro da função **sdkThread**.
4. Uma classe chamada **pyCmdHandler** com um método chamado **postCliCb**, que processa comandos CLI personalizados adicionados pelo aplicativo NX-SDK.

função sdkThread

A função sdkThread inicializa, adiciona funcionalidade e inicia o aplicativo NX-SDK. A função não exige que nenhum parâmetro seja passado para ela. Todos os aplicativos Python NX-SDK exigem três métodos da biblioteca nx_sdk_py para serem chamados:

1. **nx_sdk_py.NxSdk.getSdkInst(len(sys.argv), sys.argv)** retorna um objeto de instância SDK ou retorna Nenhum. Se um objeto de instância do SDK for retornado, o aplicativo NX-SDK foi registrado com êxito na infraestrutura do NX-OS. Se Nenhum for retornado, ocorrerão erros no momento desse processo de registro e uma entrada de registro de erro será exibida no syslog do dispositivo. Este método está documentado aqui.

   Note: A partir do NX-SDK v1.5.0, um terceiro parâmetro booleano pode ser passado para o método **NxSdk.getSdkInst**, que permite exceções avançadas quando verdadeiro e desativa exceções avançadas quando falso. Este método está documentado aqui .

1. método **sdk.startEventLoop()**, em que **sdk** é o objeto de instância do SDK retornado pelo método **nx_sdk_py.NxSdk.getSdkInst()**. Esse método inicia o aplicativo NX-SDK e permite que ele interaja com a infraestrutura do NX-OS. Este método está documentado aqui .
2. **nx_sdk_py.NxSdk.__swig_struct__(sdk)** method, onde **sdk** é o objeto de instância do SDK

retornado pelo método **nx_sdk_py.NxSdk.getSdkInst()** explicado anteriormente. Este método colocado no final da função **sdkThread** permite a saída harmoniosa do aplicativo NX-SDK.

Alguns métodos comumente usados incluem:

- **sdk.getTracer()**, onde **sdk** é o objeto de instância do SDK retornado pelo método **nx_sdk_py.NxSdk.getSdkInst()**. Esse método retorna um objeto NxTrace que pode ser usado para gerar syslogs personalizados, bem como eventos de log e erros para o histórico de eventos do aplicativo. Os syslogs personalizados aparecerão no syslog do dispositivo (visível por meio do comando **show logging logfile**) enquanto os eventos registrados no histórico de eventos do aplicativo são visíveis por meio dos comandos **show <application-name> nxsdk event-history** ou **show <application-name> nxsdk event-history errors**. Este método está documentado aqui. O objeto NxTrace retornado por este método e seus métodos associados são documentados aqui. Por exemplo, você pode exibir o histórico de eventos de um aplicativo chamado **Transceiver_DOM.py** através dos comandos **show Transceiver_DOM.py nxsdk event-history** e **show Transceiver_DOM.py nxsdk event-history errors**.
- **sdk.getCliParser()**, onde **sdk** é o objeto de instância do SDK retornado pelo método **nx_sdk_py.NxSdk.getSdkInst()**. Esse método retorna um objeto NxCliParser, que pode ser usado para executar os comandos CLI que já existem via Python, bem como criar comandos CLI personalizados. Este método está documentado aqui. O objeto NxCliParser retornado por este método e seus métodos associados são documentados aqui .
- **cliP.execShowCmd("cmd", return_type)**, onde **cliP** é o objeto NxCliParser retornado pelo método **sdk.getCliParser()**, **cmd** é o comando show que você deseja executar encapsulado entre aspas, e **return_type** são os dados formato a ser enviado. O formato dos dados pode ser **R_TEXT**, **R_JSON** ou **R_XML**. Este método está documentado aqui .

  Note: Os formatos de dados **R_JSON** e **R_XML** só funcionam se o comando suportar a saída nesses formatos. No NX-OS, você pode verificar se um comando oferece suporte à saída em um formato de dados específico, inserindo a saída no formato de dados solicitado. Se o comando piped retornar uma saída significativa, o formato de dados será suportado. Por exemplo, se você executar **show mac address-table dynamic | json** no NX-OS retorna a saída JSON e o formato de dados R_JSON também é suportado no NX-SDK.

- **cliP.execConfigCmd(cmd_filename)**, onde **cliP** é o objeto NxCliParser retornado pelo método **sdk.getCliParser()**, e **cmd_filename** é o caminho de arquivo absoluto para um arquivo que contém comandos separados por linhas. Este método retorna uma string que indica o sucesso da execução do comando - se "SUCCESS" está na string, então todos os comandos são executados com êxito. Caso contrário, a string contém a exceção que descreve por que os comandos não foram executados. Este método está documentado aqui.

Alguns métodos opcionais que podem ser úteis são:

- **sdk.setAppDesc('description string')**, onde **sdk** é o objeto de instância do SDK retornado pelo método **nx_sdk_py.NxSdk.getSdkInst()**. Este método define a descrição do aplicativo NX-SDK. A descrição é exibida no menu de ajuda sensível ao contexto do NX-OS acessado por meio de um ponto de interrogação na CLI. Este método está documentado aqui. Por exemplo, um aplicativo chamado **Transceiver_DOM.py**, uma descrição de aplicativo de Devolve todas as interfaces com transceptores compatíveis com DOM inseridos aparece na ajuda sensível ao contexto do NX-OS da seguinte maneira:

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

- **sdk.getAppName()**, onde **sdk** é o objeto de instância do SDK retornado pelo método **nx_sdk_py.NxSdk.getSdkinst()**. Este método retorna o nome do aplicativo Python. Este método está documentado aqui.

Criar comandos CLI personalizados

Em um aplicativo Python com o uso do NX-SDK, comandos CLI personalizados são criados e definidos na função sdkThread. Há dois tipos de comandos: Comandos **show** e **Config**.

1. **Os** comandos **show** exibem informações sobre o dispositivo, sua configuração ou seu ambiente.
2. Os comandos **de configuração** alteram a configuração do dispositivo, que modifica como o dispositivo reage à rede ao redor.

Esses dois métodos permitem a criação de comandos show e de comandos config respectivamente:

- **cliP.newShowCmd("cmd_name", "sintaxe")**, onde **cliP** é o objeto NxCliParser retornado pelo método **sdk.getCliParser()**, **cmd_name** é um nome exclusivo para o comando interno para o aplicativo NX-SDK personalizado, e **tax** descreve quais palavras-chave e parâmetros podem ser usados no comando. Este método retorna um objeto NxCliCmd, que está documentado aqui. Este método está documentado aqui.

  Note: Este comando é uma subclasse de cliP.newCliCmd("cmd_type", "cmd_name", "syntax") onde cmd_type é CONF_CMD ou **SHOW_CMD** (dependendo do tipo de comando sendo configurado), **cmd_name** é um nome exclusivo para o comando internal para o aplicativo NX-SDK personalizado e a **sintaxe** descreve quais palavras-chave e parâmetros podem ser usados no comando. Por causa disso, a documentação da API para esse comando pode ser mais útil para referência.

- **cliP.newConfigCmd("cmd_name", "sintaxe")**, onde **cliP** é o objeto NxCliParser retornado pelo método **sdk.getCliParser()**, **cmd_name** é um nome exclusivo para o comando interno do aplicativo NX-SDK personalizado, e a sintaxe descreve quais palavras-chave e parâmetros podem ser usados no comando. Este método retorna um objeto NxCliCmd, que está documentado aqui . Este método está documentado aqui .

  Note: Este comando é uma subclasse de cliP.newCliCmd("cmd_type", "cmd_name", "syntax") onde cmd_type é CONF_CMD ou **SHOW_CMD** (depende do tipo de comando configurado), **cmd_name** é um nome exclusivo para o comando interno do aplicativo NX-SDK personalizado e **sintaxe** descreve quais palavras-chave e parâmetros podem ser usados no comando. Por causa disso, a documentação da API para esse comando pode ser mais útil para referência.

Ambos os tipos de comandos têm dois componentes diferentes: Parâmetros e palavras-chave:

1. **Parâmetros** são valores usados para alterar os resultados do comando. Por exemplo, no comando **show ip route 192.168.1.0**, há uma palavra-chave **route** seguida por um parâmetro que aceita um endereço IP, que especifica que somente as rotas que incluem o endereço IP fornecido devem ser mostradas.

2. **Palavras-chave** mudam os resultados do comando somente por sua presença. Por exemplo, no comando **show mac address-table dynamic**, há uma palavra-chave **dinâmica**, que especifica que somente endereços MAC aprendidos dinamicamente devem ser exibidos.

Os dois componentes são definidos na sintaxe de um comando NX-SDK quando ele é criado. Existem métodos para o objeto NxCliCmd para modificar a implementação específica de ambos os componentes.

- **nx_cmd.updateParam("<parâmetro>", "help_str", type), onde nx_cmd é o objeto NxCliCmd retornado pelo cliP.newShowCmd() ou pelos métodos cliP.newConfigCmd(), <parâmetro>** é o nome do comando parâmetro que pode ser modificado entre colchetes angulares (**<>**), **help_str** define a sequência de ajuda do comando personalizado e é exibido no menu de ajuda sensível ao contexto do NX-OS acessado através de um ponto de interrogação na CLI, e **type** é o tipo do parâmetro. Parâmetros opcionais adicionais para este método estão disponíveis e documentados aqui. Estes são tipos válidos para parâmetros que podem ser especificados no argumento de tipo:

  P_INTEGER - especifica qualquer número inteiro**P_STRING** - especifica qualquer string**P_INTERFACE** - especifica qualquer interface de rede**P_IP_ADDR** - especifica qualquer endereço IP**P_MAC_ADDR** - especifica qualquer endereço MAC**P_VRF** - especifica qualquer instância de Virtual Routing and Forwarding (VRF)
- **nx_cmd.updateKeyword("keyword", "help_str", is_key), onde nx_cmd é o objeto NxCliCmd retornado pelo cliP.newShowCmd() ou pelos métodos cliP.newConfigCmd(), keyword** é o nome a palavra-chave do comando que você deseja modificar, **help_str** define a sequência de ajuda do comando personalizado e é exibida no menu de ajuda sensível ao contexto do NX-OS acessado por um ponto de interrogação na CLI, e **is_key** é um valor booleano opcional que assume como padrão False. Se **is_key** é True, a configuração exclusiva criada pelo comando com o uso desta palavra-chave não substitui outra configuração exclusiva criada pelo comando. Se **is_key** for False, a configuração criada pelo comando com o uso desta palavra-chave substituirá a outra configuração criada pelo comando. Este método está documentado aqui .

Para ver exemplos de código de componentes de comando comumente usados, consulte a seção Exemplos de comandos CLI personalizados deste documento.

Após a criação de comandos CLI personalizados, um objeto da classe **pyCmdHandler** descrita mais adiante neste documento precisa ser criado e definido como o objeto de manipulador de chamada de retorno CLI para o objeto NxCliParser. Isto é demonstrado do seguinte modo:

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

Em seguida, o objeto NxCliParser precisa ser adicionado à árvore de analisador CLI do NX-OS para que os comandos CLI personalizados sejam visíveis para o usuário. Isso é feito com o comando **cliP.addToParseTree()**, em que **cliP** é o objeto NxCliParser retornado pelo método **sdk.getCliParser()**.

exemplo de função sdkThread

Aqui está um exemplo de uma função sdkThread típica com o uso das funções explicadas anteriormente. Esta função (entre outras dentro de um aplicativo típico personalizado NX-SDK Python) utiliza variáveis globais, que são instanciadas na execução do script.

```
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)
```

Classe do pyCmdHandler

A classe **pyCmdHandler** é herdada da classe **NxCmdHandler** na biblioteca nx_sdk_py. O método **postCliCb(self, click)** definido na classe **pyCmdHandler** é chamado sempre que os comandos CLI originados de um aplicativo NX-SDK. Assim, o método **postCliCb(self, click)** é onde você define como os comandos CLI personalizados definidos na função sdkThread se comportam no dispositivo.

A função **postCliCb(self, click)** retorna um valor booleano. Se **True** for retornado, presume-se que

o comando foi executado com êxito. **False** deve ser retornado se o comando não foi executado com êxito por qualquer motivo.

O parâmetro **click** usa o nome exclusivo que foi definido para o comando quando ele foi criado na função sdkThread. Por exemplo, se você criar um novo comando **show** com um nome exclusivo de **show_xcvr_dom**, então é recomendável consultar esse comando pelo mesmo nome na função **postCliCb(self, click)** depois de verificar se o nome do argumento click contém **show_xcvr_dom**. É demonstrado aqui:

```
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()
```

Se um comando que utiliza parâmetros for criado, você provavelmente precisará usar esses parâmetros em algum ponto da função **postCliCb(self, click)**. Isso pode ser feito com o método **click.getParamValue("<parâmetro>")**, em que **<parâmetro>** é o nome do parâmetro de comando que você deseja obter o valor entre colchetes angulares (**<>**). Este método está documentado aqui. No entanto, o valor retornado por esta função precisa ser convertido para o tipo necessário. Isso pode ser feito com estes métodos:

- **nx_sdk_py.void_to_int** converte um valor em um tipo inteiro.
- **nx_sdk_py.void_to_string** converte um valor em um tipo de string.

A função **postCliCb(self, click)** (ou quaisquer funções subsequentes) também normalmente será onde a saída do comando show é impressa no console. Isso é feito com o método **click.printConsole()**.

> **Note**: Se o aplicativo encontrar um erro, uma exceção não tratada ou, de outra forma, sair repentinamente, a saída da função **click.printConsole()** não será exibida. Por esse motivo, a melhor prática ao depurar seu aplicativo Python é registrar mensagens de depuração no syslog com o uso de um objeto NxTrace retornado pelo método **sdk.getTracer()**, ou usar instruções de impressão e executar o aplicativo através do binário Bash shell **/isan/bin/python**.

Exemplo de classe do pyCmdHandler

O código a seguir serve como exemplo para a classe **pyCmdHandler** descrita acima. Este código é extraído do arquivo ip_move.py no aplicativo NX-SDK de movimento de ip disponível aqui. A finalidade deste aplicativo é rastrear a movimentação de um endereço IP definido pelo usuário através das interfaces de um dispositivo Nexus. Para fazer isso, o código localiza o endereço MAC da entrada do endereço IP através do parâmetro **<ip>** dentro do cache ARP do dispositivo e, em seguida, verifica qual VLAN esse endereço MAC reside ao usar a tabela de endereços MAC do dispositivo. Usando esse MAC e VLAN, o comando **show system internal l2fm l2dbg macdb address <mac> vlan <vlan>** exibe uma lista de índices de interface SNMP aos quais essa combinação foi associada recentemente. O código usa o comando **show interface snmp-ifindex** para converter índices recentes de interface SNMP em nomes de interface legíveis por humanos.

```
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
```

Exemplos de sintaxe de comando CLI personalizados

Esta seção mostra alguns exemplos do parâmetro de sintaxe usado quando você cria comandos CLI personalizados com os métodos **cliP.newShowCmd()** ou **cliP.newConfigCmd()**, em que **cliP** é o objeto NxCliParser retornado pelo **método sdk.getCliParser()** .

Note: O suporte para sintaxe com parênteses de abertura e fechamento ("(" e ")") é apresentado no NX-SDK v1.5.0, incluído no NX-OS Versão 7.0(3)I7(3). Supõe-se que o usuário utilize NX-SDK v1.5.0 quando segue qualquer um desses exemplos fornecidos que incluem sintaxe utilizando parênteses de abertura e fechamento.

Palavra-chave única

Esse comando show usa uma única palavra-chave mac e adiciona uma sequência de caracteres auxiliar de Mostra todos os endereços MAC mal programados neste dispositivo à palavra-chave.

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

Parâmetro único

Este comando show usa um único parâmetro **<mac>**. Os colchetes angulares que envolvem a palavra mac significam que este é um parâmetro. Uma cadeia de caracteres auxiliar do endereço MAC para verificar se há má programação é adicionada ao parâmetro. O parâmetro **nx_sdk_py.P_MAC_ADDR** no método **nx_cmd.updateParam()** é usado para definir o tipo do parâmetro como um endereço MAC, que impede a entrada de outro tipo pelo usuário final, como uma string, um inteiro ou um endereço IP.

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

Palavra-chave opcional

Esse comando show pode, opcionalmente, ter uma única palavra-chave **[mac]**. Os colchetes envolventes em torno da palavra mac significam que essa palavra-chave é opcional. Uma sequência de ajuda de Mostra todos os endereços MAC mal programados neste dispositivo são adicionados à palavra-chave.

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

Parâmetro opcional

Esse comando show pode, opcionalmente, pegar um único parâmetro **[<mac>]**. Os colchetes envolventes em torno da palavra < mac > significam que esse parâmetro é opcional. Os colchetes angulares que envolvem a palavra mac significam que este é um parâmetro. Uma cadeia de caracteres auxiliar do endereço MAC para verificar se há má programação é adicionada ao parâmetro. O parâmetro **nx_sdk_py.P_MAC_ADDR** no método **nx_cmd.updateParam()** é usado para definir o tipo do parâmetro como um endereço MAC, que impede a entrada de outro tipo pelo usuário final, como uma string, um inteiro ou um endereço IP.

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

Palavra-chave e parâmetro únicos

Esse comando show usa uma única palavra-chave mac imediatamente seguida pelo parâmetro **<mac-address>**. Os colchetes angulares que envolvem a palavra mac-address significam que este é um parâmetro. Uma sequência de ajuda de Verificar endereço MAC para má programação é adicionada à palavra-chave. Uma cadeia de caracteres auxiliar do endereço MAC para verificar se há má programação é adicionada ao parâmetro. O parâmetro **nx_sdk_py.P_MAC_ADDR** no método **nx_cmd.updateParam()** é usado para definir o tipo do parâmetro como um endereço MAC, que impede a entrada de outro tipo pelo usuário final, como uma string, um inteiro ou um endereço IP.

```
 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)
```

Várias palavras-chave e parâmetros

Esse comando show pode ter uma de duas palavras-chave, ambas com dois parâmetros diferentes seguindo-as. A primeira palavra-chave mac tem um parâmetro de **<mac-address>**, e a segunda palavra-chave ip tem um parâmetro de **<ip-address>**. Os colchetes angulares que envolvem as palavras mac-address e ip-address significam que são parâmetros. Uma sequência de ajuda de Verificar endereço MAC para má programação é adicionada à palavra-chave mac. Uma string auxiliar de endereço MAC para verificar se há má programação é adicionada ao parâmetro **<mac-address>**. O parâmetro **nx_sdk_py.P_MAC_ADDR** no método **nx_cmd.updateParam()** é usado para definir o tipo do **<mac-address>** como um endereço MAC, o que impede a entrada do usuário final de outro tipo, como string, inteiro ou endereço IP. Uma sequência de ajuda de Check IP address for misprograming é adicionada à palavra-chave ip. Uma string auxiliar de endereço IP para verificar se há programação incorreta é adicionada ao parâmetro **<ip-address>**. O parâmetro **nx_sdk_py.P_IP_ADDR** no método **nx_cmd.updateParam()** é usado para definir o tipo do **<ip-address>** como um endereço IP, o que impede a entrada do usuário final de outro tipo, como string, inteiro ou endereço IP.

```
 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)
```

Várias palavras-chave e parâmetros com palavra-chave opcional

Esse comando show pode ter uma de duas palavras-chave, ambas com dois parâmetros diferentes seguindo-as. A primeira palavra-chave mac tem um parâmetro de **<mac-address>**, e a segunda palavra-chave ip tem um parâmetro de **<ip-address>**. Os colchetes angulares que envolvem as palavras mac-address e ip-address significam que são parâmetros. Uma sequência de ajuda de Verificar endereço MAC para má programação é adicionada à palavra-chave mac. Uma string auxiliar de endereço MAC para verificar se há má programação é adicionada ao parâmetro **<mac-address>**. O parâmetro **nx_sdk_py.P_MAC_ADDR** no método **nx_cmd.updateParam()** é usado para definir o tipo do **<mac-address>** como um endereço MAC, o que impede a entrada do usuário final de outro tipo, como string, inteiro ou endereço IP. Uma sequência de ajuda de Check IP address for misprograming é adicionada à palavra-chave ip. Uma string auxiliar de endereço IP para verificar se há programação incorreta é adicionada ao parâmetro **<ip-address>**. O parâmetro **nx_sdk_py.P_IP_ADDR** no método **nx_cmd.updateParam()** é usado para definir o tipo do **<ip-address>** como um endereço IP, o que impede a entrada do

usuário final de outro tipo, como string, inteiro ou endereço IP. Esse comando show pode, opcionalmente, ter uma palavra-chave [clear]. Uma sequência de caracteres auxiliar Limpa os endereços detectados como mal programados é adicionada a esta palavra-chave opcional.

```
 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")
```

Várias palavras-chave e parâmetros com parâmetro opcional

Esse comando show pode ter uma de duas palavras-chave, ambas com dois parâmetros diferentes seguindo-as. A primeira palavra-chave mac tem um parâmetro de **<mac-address>**, e a segunda palavra-chave ip tem um parâmetro de **<ip-address>**. Os colchetes angulares que envolvem as palavras mac-address e ip-address significam que são parâmetros. Uma sequência de ajuda de Verificar endereço MAC para programação incorreta é adicionada à palavra-chave mac. Uma string auxiliar de endereço MAC para verificar se há má programação é adicionada ao parâmetro **<mac-address>**. O parâmetro **nx_sdk_py.P_MAC_ADDR** no método **nx_cmd.updateParam()** é usado para definir o tipo do **<mac-address>** como um endereço MAC, o que impede a entrada do usuário final de outro tipo, como string, inteiro ou endereço IP. Uma sequência de ajuda de Check IP address for misprograming é adicionada à palavra-chave ip. Uma string auxiliar de endereço IP para verificar se há programação incorreta é adicionada ao parâmetro **<ip-address>**. O parâmetro **nx_sdk_py.P_IP_ADDR** no método **nx_cmd.updateParam()** é usado para definir o tipo do **<ip-address>** como um endereço IP, o que impede a entrada do usuário final de outro tipo, como string, inteiro ou endereço IP. Este comando show pode, opcionalmente, pegar um parâmetro **[<module>]**. Uma sequência de caracteres auxiliar Somente endereços claros no módulo especificado são adicionados a este parâmetro opcional.

```
 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)
```

Depurar um aplicativo Python com NX-SDK

Depois que um aplicativo Python NX-SDK for criado, ele frequentemente precisará ser depurado. O NX-SDK informa você caso haja algum erro de sintaxe em seu código, mas como a biblioteca do Python NX-SDK utiliza o SWIG para converter bibliotecas C++ em bibliotecas Python, qualquer exceção encontrada no momento da execução do código resulta em um dump central do aplicativo semelhante a este:

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

Devido à natureza ambígua desta mensagem de erro, a melhor prática para depurar aplicativos Python é registrar mensagens de depuração no syslog com o uso de um objeto NxTrace retornado pelo método **sdk.getTracer()**. Isto é demonstrado do seguinte modo:

```
#! /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")
```

Se o registro de mensagens de depuração no syslog não for uma opção, um método alternativo é usar instruções de impressão e executar o aplicativo por meio do binário Bash shell **/isan/bin/python**. No entanto, a saída dessas instruções de impressão só será visível quando executadas dessa maneira - executar o aplicativo através do shell VSH não produz nenhuma saída. Um exemplo de utilização de instruções de impressão é mostrado aqui:

```
#! /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")
```

Implante um aplicativo Python com NX-SDK

Depois que um aplicativo Python tiver sido totalmente testado no shell Bash e estiver pronto para implantação, o aplicativo deverá ser instalado na produção através da VSH. Isso permite que o aplicativo persista quando o dispositivo é recarregado ou quando a comutação do sistema ocorre em um cenário de supervisor duplo. Para implantar um aplicativo através da VSH, é necessário criar um pacote RPM com o uso de um ambiente de compilação NX-SDK e ENXOS SDK. O Cisco DevNet fornece uma imagem do Docker que permite a fácil criação de pacotes RPM.

> **Note**: Para obter assistência para instalar o Docker em seu sistema operacional específico, consulte a documentação de instalação do Docker.

Em um host compatível com Docker, puxe a versão da imagem de sua escolha com o comando **docker pull dockercisco/nxsdk:<tag>**, onde **<tag>** é a marca da versão da imagem de sua escolha. Você pode exibir as versões de imagem disponíveis e suas marcas correspondentes aqui. Isso é demonstrado com a marca **v1** aqui:

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

Inicie um contêiner chamado **nxsdk** dessa imagem e anexe-o. Se a marca de sua escolha for diferente, substitua **v1** pela sua marca:

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

Atualize para a versão mais recente do NX-SDK e navegue até o diretório **NX-SDK** e, em seguida, puxe os arquivos mais recentes do git:

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

Se precisar usar uma versão mais antiga do NX-SDK, você pode clonar a filial NX-SDK com o uso da respectiva marca de versão com o **comando git clone -b v<version>** https://github.com/CiscoDevNet/NX-SDK.git, onde **<version>** é a versão do NX-SDK de que você precisa. Isso é demonstrado aqui com o NX-SDK v1.0.0:

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

Em seguida, transfira o aplicativo Python para o contêiner Docker. Há algumas maneiras diferentes de fazer isso.

- Saia do contêiner do Docker (que para o contêiner e exige que você o inicie mais uma vez), transfira o aplicativo Python para o host do Docker e use o comando **docker cp** para copiar o aplicativo do host para o contêiner. Isto é demonstrado aqui, partindo do pressuposto de que a aplicação Python foi transferida para o anfitrião Docker em **/app/python_app.py**.

```
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
```

- Copie o conteúdo do aplicativo Python na área de transferência do sistema e cole o conteúdo em um arquivo criado no contêiner Docker com o uso do vim.

Em seguida, use o script **rpm_gen.py** localizado em **/NX-SDK/scripts/** para criar um pacote RPM do aplicativo Python. Este script tem um argumento necessário e dois switches necessários:

- O nome do arquivo do aplicativo Python. Por exemplo, um aplicativo Python em um arquivo chamado **python_app.py** resultaria em um argumento de **python_app.py**. Esse nome de arquivo será usado posteriormente como o nome do aplicativo para NX-SDK e também pelo NX-OS para se referir aos comandos criados por esse aplicativo.

Note: O nome do arquivo não precisa conter nenhuma extensão de arquivo, como **.py**. Neste exemplo, se o nome do arquivo fosse **python_app** em vez de **python_app.py**, o pacote RPM seria gerado sem um problema.

- O switch -s usa um argumento para o caminho de arquivo absoluto que leva ao local onde o nome de arquivo acima está localizado. Por exemplo, se **python_app.py** estiver localizado em **/root/**, o argumento correto será **-s /root/**.
- O switch -u indica que o nome do arquivo de origem é o mesmo do nome do arquivo executável.

O uso do script **rpm_gen.py** é demonstrado aqui.

```
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
```

O caminho do arquivo para o pacote RPM é indicado na linha final da saída do script **rpm_gen.py**. Esse arquivo deve ser copiado do contêiner do Docker para o host para que possa ser transferido para o dispositivo Nexus no qual você deseja executar o aplicativo. Depois de sair do contêiner do Docker, isso pode ser feito facilmente com o comando docker cp <container>:<container_filepath> <host_filepath>, onde **<container>** é o nome do contêiner do Docker NX-SDK (neste caso, nxsdk), **<container_filepath>** é o caminho de arquivo completo do pacote RPM dentro do contêiner (neste caso, **/NX-SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm**) e **<host_filepath>** é o caminho de arquivo completo em nosso host Docker para o qual o pacote RPM deve ser transferido (neste caso, **/root/**). Este comando é demonstrado aqui:

```
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
```

Transfira este pacote RPM para o dispositivo Nexus com o uso do método preferido de transferência de arquivos. Quando o pacote RPM estiver no dispositivo, ele deverá ser instalado e ativado de forma semelhante a um SMU. Isto é demonstrado da seguinte forma, partindo do pressuposto de que o pacote RPM foi transferido para o flash de inicialização do dispositivo.

```
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: Ao instalar o pacote RPM com o comando **install add**, inclua o dispositivo de armazenamento e o nome exato do pacote. Quando você ativar o pacote RPM após a instalação, não inclua o dispositivo de armazenamento e o nome do arquivo - use o nome do pacote em si. Você pode verificar o nome do pacote com o comando **show install inative**.

Quando o pacote RPM for ativado, você poderá iniciar o aplicativo com NX-SDK com o comando de configuração **nxsdk service <application-name>**, em que **<application-name>** é o nome do arquivo Python (e, subsequentemente, o aplicativo) que foi definido quando o script rpm_gen.py foi usado anteriormente. Isto é demonstrado do seguinte modo:

```
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
```

Você pode verificar se o aplicativo está ativo e começou a ser executado com o comando **show nxsdk internal service**:

```
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
```

Você também pode verificar se os comandos CLI personalizados criados por este aplicativo estão acessíveis no NX-OS:

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

Informações Relacionadas

- GitHub NX-SDK
- Guia de programabilidade do Cisco Nexus 9000 Series NX-OS, versão 7.x
- Guia de programabilidade do Cisco Nexus 3000 Series NX-OS, versão 7.x
- Guia de programabilidade do Cisco Nexus 3500 Series NX-OS, versão 7.x
- Documentação sobre programabilidade e automação de rede com os switches Cisco Nexus 9000 Series
- Programabilidade e automação com o Cisco Open NX-OS (PDF)
- Suporte Técnico e Documentação - Cisco Systems