Nexus 3000/9000スイッチでのNX-SDK Pythonアプリケーションの開発、デバッグ、導入

内容

概要

このドキュメントでは、Nexus 3000およびNexus 9000プラットフォームでCisco NX-OSを使用するCisco NX-Software Development Kit(SDK)を使用したPythonアプリケーション開発のワークフローについて説明します。

前提条件

要件

このドキュメントに特有の要件はありません。

使用するコンポーネント

このドキュメントの情報は、次のソフトウェアとハードウェアのバージョンに基づいています。

- このドキュメントでは、NX-SDK v1.0.0およびNX-SDK v1.5.0を使用します
- NX-SDK v1.0.0は、NX-OSリリース7.0(3)I6(1)以降のNexus 9000プラットフォームと、NX-OSリリース7.0(3)I7(1)以降のNexus 3000プラットフォームで使用できます
- NX-SDK v1.5.0は、NX-OSリリース7.0(3)I7(3)以降のNexus 9000プラットフォームとNexus 3000プラットフォームの両方で使用できます

このドキュメントの情報は、特定のラボ環境にあるデバイスに基づいて作成されました。このドキュメントで使用するすべてのデバイスは、初期（デフォルト）設定の状態から起動しています。対象のネットワークが実稼働中である場合には、どのようなコマンドについても、その潜在的な影響について確実に理解しておく必要があります。

背景説明

Cisco NX-SDKを使用すると、Nexus 9000およびNexus 3000プラットフォーム上のCisco NX-OSでネイティブに実行できるカスタムアプリケーションを開発できます。NX-SDKは、お客様が独自のCLIコマンドと出力を作成し、特定のイベントに応じてカスタマイズされたsyslogを生成し、ストリームに合わせたテレメトリなどを生成する機能を提供します。

NX-SDKにはC++ APIがあり、Simplified Wrapper and Interface Generator(SWIG)を使用して他の言語に翻訳されます。 これにより、お客様は任意の言語でNX-SDKを使用できます。このドキュメントでは、Pythonでの一般的なNX-SDK機能の実装について説明し、お客様が独自のNX-SDK Pythonアプリケーションを開発するためのワークフローを提供します。

NX-SDKを使用したPythonアプリケーションの開発

NX-SDKの有効化

NX-SDKアプリケーションを実行するには、まずデバイスでNX-SDK機能を有効にする必要があります。

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

Pythonファイルの作成

Pythonファイルは、NX-OS Bashシェルを使用して作成および編集できます。Bashシェルを使用するには、まずデバイスで有効にする必要があります。

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

Bashシェルに入り、viテキスト エディタを使用してPythonファイルを作成および編集します。

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

注：ベストプラクティスは、Pythonファイルを/isan/bin/ディレクトリ**に作成することです**

。Pythonファイルを実行するには、実行権限が必要です。Pythonファイルを**/bootflashディレクトリやサブディレクトリに配置しないでください。**

注：NX-OSからPythonファイルを作成および編集する必要はありません。開発者は、自分のローカル環境を使用してアプリケーションを作成し、完成したファイルを自分の選択したファイル転送プロトコルを使用してデバイスに転送することができます。ただし、開発者がNX-OSユーティリティを使用してスクリプトをデバッグおよびトラブルシューティングする方が効率的な場合があります。

NX-SDKコンポーネントの実装

開発者が Cisco DevNet NX-SDK GitHubのcustomCliPyAppテンプレートからNX-SDK Pythonアプリケーションを作成することを推奨しますす。このドキュメントの以降のセクションでは、このテンプレート内の名前を使用して必要コンポーネントを参照します。

NX-SDK Pythonアプリケーションには、次の4つの主要コンポーネントが必要です。

1. NX-SDKは、import nx_sdk_py文を使用してアプリケーションにインポートする必要があります。
2. NX-SDKアプリケーションを起動し、アプリケーションに関連するさまざまなオプションを変更する関数(通常は**sdkThread**という名前)。
3. カスタムCLIコマンドの作成、およびsdkスレッド関数内でのカスタムCLIコマンド構文の**定義**です。
4. **postCliCb**という名前のメソッドを持つ**pyCmdHandler**という名前のクラスで、NX-SDKアプリケーションによって追加されたカスタムCLIコマンドを処理します。

sdkThread関数

sdkThread関数は、NX-SDKアプリケーションを初期化、機能の追加、および起動します。この関数には、パラメータを渡す必要はありません。すべてのPython NX-SDKアプリケーションは、nx_sdk_pyライブラリから3つのメソッドを呼び出す必要があります。

1. **nx_sdk_py.NxSdk.getSdkInst(len(sys.argv), sys.argv)はSDKインスタンスオブジェクトを返す**か、Noneを返します。SDKインスタンスオブジェクトが返された場合、NX-SDKアプリケーションはNX-OSインフラストラクチャに正常に登録されました。Noneが返された場合、この登録プロセスの時点でエラーが発生し、デバイスのsyslogにエラーログエントリが表示されます。この方法は、ここに記載されています。

   注：NX-SDK v1.5.0から、3番目のブール型パラメータを**NxSdk.getSdkInstメソッドに渡すことができます。**このメソッドは、Trueの場合は拡張例外を有効にし、Falseの場合は拡張例外を無効にします。この方法は、ここに記載されています。

1. **sdk.startEventLoop()メソ**ッド。ここで、sdkは**nx_sdk_py.**NxSdk.getSdkInst()メソッドによって返さ**れるSDKインスタンスオブジェクト**です。このメソッドはNX-SDKアプリケーショ

ンを起動し、NX-OSインフラストラクチャと対話できるようにします。この方法は、ここに記載されています。

2. nx_sdk_py.NxSdk.__swig_destroy__(sdk)メソッド。ここでsdkは、nx_sdk_py.NxSdk.getSdkInst()メソッドによって返されたSDKインスタンスオブジェクトです。sdkThread関数の最後にこのメソッドを配置すると、NX-SDKアプリケーションを正常に終了できます。

一般的に使用される方法には、次のものがあります。

- sdk.getTracer()。ここで、sdkはnx_sdk_py.NxSdk.getSdkInst()メソッドによって返されるSDKインスタンスオブジェクトです。このメソッドは、カスタムsyslogの生成に使用できるNxTraceオブジェクトを返します。また、アプリケーションのイベント履歴にイベントや エラーを記録することもできます。カスタムsyslogは、デバイスのsyslogに表示されます(show logging logfileコマンドで表示されます)。アプリケーションのイベント履歴に記録されるイベントは、show <application-name> nxsdk event-history eventsまたはshow <application-name> nxsdk event-history errorsコマンドで表示されます。この方法は、ここに記載されています。このメソッドによって返されるNxTraceオブジェクトとその関連するメソッドは、ここに記述されています。たとえば、Transceiver_DOM.pyという名前のアプリケーションのイベント履歴を表示するには、show Transceiver_DOM.py nxsdk event-history eventsコマンドとshow Transceiver_DOM.py nxsdk event-history errorsコマンドを使用します。
- sdk.getCliParser()です。このsdkは、nx_sdk_py.NxSdk.getSdkInst()メソッドによって返されたSDKインスタンスオブジェクトです。このメソッドはNxCliParserオブジェクトを返します。このオブジェクトは、Pythonにすでに存在するCLIコマンドを実行したり、カスタムCLIコマンドを作成するために使用できます。この方法は、ここに記載されています。このメソッドによって返されるNxCliParserオブジェクトとそれに関連するメソッドは、ここに記載されています。
- cliP.execShowCmd("cmd", return_type)。ここでcliPはsdk.getCliParser()メソッドによって返されるNxCliParserオブジェクトで、cmdは引用符で囲んで実行するshowコマンド return_type。データ形式は、R_TEXT、R_JSON、またはR_XMLのいずれかです。この方法は、ここに記載されています。

  注：R_JSONおよびR_XMLのデータ形式は、コマンドがこれらの形式の出力をサポートする場合にのみ機能します。NX-OSでは、コマンドが特定のデータ形式の出力をサポートしているかどうかを確認するために、要求されたデータ形式に出力をパイプで渡すことができます。pipedコマンドが意味のある出力を返す場合、そのデータ形式がサポートされます。たとえば、show mac address-table dynamic | NX-OSのjsonはJSON出力を返し、R_JSONデータ形式はNX-SDKでもサポートされます。

- cliP.execConfigCmd(cmd_filename)。ここでcliPはsdk.getCliParser()メソッドによって返されるNxCliParserオブジェクトで、cmd_filenameは行で区切られたコマンドを含むファイルへの絶対ファイルです。このメソッドは、コマンド実行の成功を示す文字列を返します。「SUCCESS」が文字列の中にある場合、すべてのコマンドが正常に実行されます。それ以外の場合は、コマンドの実行に失敗した理由を説明する例外が文字列に含まれます。この方法は、ここに記載されています。

便利なオプションの方法は次のとおりです。

- sdk.setAppDesc('description string')。ここで、sdkは、nx_sdk_py.NxSdk.getSdkInst()メソッドによって返されたSDKインスタンスオブジェクトです。このメソッドは、NX-SDKアプリケーションの説明を設定します。説明は、CLIの疑問符を使用してアクセスしたNX-OSの状況依存ヘルプメニューに表示されます。この方法は、ここに記載されています。たとえば、

Transceiver_DOM.pyという名前のアプリケーションで、Returns all interfaces with DOM対応トランシーバが挿入されている場合、次のようにNX-OSのコンテキスト対応ヘルプに表示されます。

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

- **sdk.getAppName()です。ここで、sdkはnx_sdk_py.NxSdk.getSdkinst()メソッドによって返されるSDKインスタンスオブジェクト**です。このメソッドは、Pythonアプリケーションの名前を返します。この方法は、ここに記載されています。

カスタムCLIコマンドの作成

NX-SDKを使用するPythonアプリケーションでは、カスタムCLIコマンドがsdkThread関数で作成および定義されます。次の2種類のコマンドがあります。**showコマンド**、および**Configコマンド**。

1. **showコマンド**は、デバイス、その設定、またはその環境に関する情報を表示します。
2. **Configコマンド**は、デバイスの設定を変更します。これにより、デバイスが周囲のネットワークにどのように反応するかを変更できます。

次の2つの方法により、それぞれshowコマンドとconfigコマンドを作成できます。

- **cliP.newShowCmd("cmd_name", "syntax")。ここで、cliPはsdk.getCliParser()メソッドによって返されるNxCliParserオブジェクトで、cmd_nameはカスタムNX-SDKアプリケーションの内部コマンドの一意名前と構文を説明しますコマンドで使用できるキーワードとパラメータ**このメソッドは、ここに記載されているNxCliCmdオブジェクトを返します。この方法は、ここに記載されています。

  注：このコマンドは、cliP.newCliCmd("cmd_type", "cmd_name", "syntax")のサブクラスで、cmd_typeはCONF_CMDまたは**SHOW_CMD(設定するコマンドのタイプに応じる)です SDKアプリケーションおよび構文は、コマンドで使用できるキーワードとパラメータを説明します。**このため、このコマンドのAPIドキュメントは、より参考になります。

- **cliP.newConfigCmd("cmd_name", "syntax")。ここで、cliPはsdk.getCliParser()メソッドによって返されるNxCliParserオブジェクトで、cmd_nameはカスタムNX-SDKアプリケーションの内部コマンドの一意名前と構文を説明しますコマンドで使用できるキーワードとパラメータ**このメソッドは、ここに記載されているNxCliCmdオブジェクトを返します。この方法は、ここに記載されています。

  注：このコマンドは、cliP.newCliCmd("cmd_type", "cmd_name", "syntax")のサブクラスで、cmd_typeはCONF_CMDまたは**SHOW_CMD**（設定されているコマンドのタイプに応じて異なる）ですnx-SDKアプリケーションおよび構文**は、コマンドで使用できるキーワードとパラメータを示します。**このため、このコマンドのAPIドキュメントは、より参考になります。

どちらのタイプのコマンドにも、次の2つの異なるコンポーネントがあります。パラメータとキーワード：

1.パラ**メータ**は、コマンドの結果を変更するために使用される値です。たとえば、**show ip route**

192.168.1.0コマンドでは、routeキーワードが続き、IPアドレスを受け入れるパラメータが指定されています。このパラメータは、指定されたIPアドレスを含むルートだけを表示することを指定します。

2.キーワードは、コマンドの結果をキーワードだけで変更します。たとえば、show mac address-table dynamicコマンドには、動的に学習されたMACアドレスのみを表示することを指定するdynamicキーワードがあります。

どちらのコンポーネントも、作成時にNX-SDKコマンドの構文で定義されます。NxCliCmdオブジェクトには、両方のコンポーネントの特定の実装を変更するためのメソッドがあります。

- nx_cmd.updateParam("<parameter>", "help_str", type)。nx_cmdはcliP.newShowCmd()またはcliP.newConfigCmd()メソッドによって返されるNxCliCmdオブジェクトです。<parameter>角括弧(<>), help_strは、カスタムコマンドのヘルプ文字列を設定し、CLIの疑問符を使用してアクセスするNX-OSの状況依存ヘルプメニューに表示されます。typeはパラメータのタイプです。このメソッドの追加のオプションパラメータについては、こちらをご覧ください。
  type引数で指定できるパラメータには、次の有効な型があります。

  P_INTEGER：任意の整数を指定しますP_STRING：任意の文字列を指定しますP_INTERFACE：任意のネットワークインターフェイスを指定しますP_IP_ADDR：任意のIPアドレスを指定しますP_MAC_ADDR：任意のMACアドレスを指定しますP_VRF：任意のVirtual Routing and Forwarding(VRF)インスタンスを指定します
- nx_cmd.updateKeyword("keyword", "help_str", is_key)。nx_cmdはcliP.newShowCmd()またはcliP.newConfigCmd()メソッドによって返されるNxCliCmdオブジェクトで、キーワードは変更するコマンドの名前ですhelp_strは、カスタムコマンドのヘルプ文字列を設定し、CLIの疑問符を使用してアクセスするNX-OSコンテキスト対応ヘルプメニューに表示されます。is_keyは、デフォルトのFalseに設定されるオプションのブール値です。is_keyがTrueの場合、このキーワードを使用してコマンドによって作成された一意の設定は、コマンドによって作成された他の一意の設定を上書きしません。is_keyがFalseの場合、このキーワードを使用してコマンドによって作成された設定は、コマンドによって作成された他の設定を上書きします。この方法は、ここに記載されています。

よく使用されるコマンドコンポーネントのコード例を表示するには、このドキュメントの「カスタムCLIコマンドの例」セクションを参照してください。

カスタムCLIコマンドを作成した後、このドキュメントで後述するpyCmdHandlerクラスのオブジェクトを作成し、NxCliParserオブジェクトのCLIコールバックハンドラオブジェクトとして設定する必要があります。これは次のように示されています。

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

次に、カスタムCLIコマンドがユーザに表示されるように、NxCliParserオブジェクトをNX-OS CLIパーサツリーに追加する必要があります。これはcliP.addToParseTree()コマンドで行われます。ここで、cliPはsdk.getCliParser()メソッドによって返されるNxCliParserオブジェクトです。

sdkThread関数の例

以下は、前述の関数を使用した一般的なsdkThread関数の例です。この関数（一般的なカスタムNX-SDK Pythonアプリケーション内の他の関数など）は、グローバル変数を使用します。グロー

バル変数は、スクリプトの実行時にインスタンス化されます。

```
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クラス

pyCmdHandlerクラスは、nx_sdk_pyライブラリ内のNxCmdHandlerクラスから継承されます。pyCmdHandlerクラス内で定義されたpostCliCb(self, clicmd)メソッドは、NX-SDKアプリケーションから発信されるCLIコマンドのたびに呼び出されます。したがって、postCliCb(self, clicmd)メソッドは、sdkThread関数で定義されたカスタムCLIコマンドがデバイスでどのように動作するかを定義します。

postCliCb(self, clicmd)関数はブール値を返します。Trueが返された場合は、コマンドが正常に実行されたと見なされます。何らかの理由でコマンドが正常に実行されなかった場合は、Falseを返します。

clicmdパラメータは、sdkThread関数で作成されたときにコマンドに定義された一意の名前を使用します。たとえば、show_xcvr_domという一意の名前を持つ新しいshowコマンドを作成する場合は、clicmd引数の名前にshow_xcvr_domが含まれているかどうかを確認した後、postCliCb(self, clicmd)関数)でをで同同同じ名前でこのコマンドをコマンドを参照することをお勧勧めします。次に示します。

```
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()
```

パラメータを利用するコマンドが作成された場合、ほとんどの場合、postCliCb(self, clicmd)関数の中の任意の時点でこれらのパラメータを使用する必要があります。これを行うには、clicmd.getParamValue("<parameter>")メソッドを使用します。ここで、<parameter>は、角カッコ(<>)で囲まれた値を取得するコマンドパラメータの名前です。 この方法は、ここに記載されています。ただし、この関数から返される値は、必要な型に変換する必要があります。これは、次の方法で実行できます。

- nx_sdk_py.void_to_intは、値を整数型に変換します。
- nx_sdk_py.void_to_stringは、値を文字列型に変換します。

postCliCb(self, clicmd)関数(または後続の関数)は、通常、showコマンドの出力がコンソールに出力される場所でもあります。これはclicmd.printConsole()メソッドで行います。

> **注**：アプリケーションでエラーが発生した場合、処理されない例外が発生した場合、または突然終了した場合、clicmd.printConsole()関数からの出力はまったく表示されません。このため、Pythonアプリケーションをデバッグする際のベストプラクティスは、sdk.getTracer()メソッドで返されたNxTraceオブジェクトを使用してデバッグメッセージをsyslogに記録するか、print文を使用してBashシェルの/isan/bin/pythonバイナリでアプリケーションをを実行します。

pyCmdHandlerクラスの例

次のコードは、上記のpyCmdHandlerクラスの例として機能します。このコードは、ここで入手可能なip-movement NX-SDKアプリケーションのip_move.pyファイルから取得したものです。このアプリケーションの目的は、Nexusデバイスのインターフェイス間でのユーザ定義IPアドレスの移動を追跡することです。このため、コードはデバイスのARPキャッシュ内の<ip>パラメータを介して入力されたIPアドレスのMACアドレスを見つけ、そのMACアドレスがデバイスのMACアドレステーブルを使用して存在するVLANを確認します。このMACおよびVLANを使用して、show system internal l2fm l2dbg macdb address <mac> vlan <vlan>コマンドを実行すると、この組み合わせが最近関連付けられたSNMPインターフェイスインデックスのリストが表示されます。次に、show interface snmp-ifindexコマンドを使用して、最近のSNMPインターフェイスインデックスを人間が読みやすいインターフェイス名に変換します。

```
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
```

カスタムCLIコマンド構文の例

このセクションでは、**cliP.newShowCmd()メソッドまたはcliP.newConfigCmd()メソッドを使用してカスタムCLIコマンドを作成する際に使用する構文パラメータの例を示します。cliP**は CliParser()メソッドでは ..

> **注：**カッコ(" ("および") ")を開く構文のサポートは、NX-OSリリース7.0(3)I7(3)に含まれるNX-SDK v1.5.0で導入されました。 ユーザがNX-SDK v1.5.0を使用していることを前提としています。この例では、開きカッコと閉じカッコを使用した構文が含まれています。

シングルキーワード

このshowコマンドは、単一のキーワードmacを取得し、「Show all misprogrammed MAC addresses on this device」というヘルパーストリングをキーワードに追加します。

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

単一パラメータ

このshowコマンドは、単一のパラメータ**<mac>を使用します**。macという単語の周りの角カッコは、これがパラメータであることを示します。パラメータに、プログラミングの誤りをチェックするためのMACアドレスのヘルパ文字列が追加されます。**nx_cmd.updateParam()メソッドのnx_sdk_py.P_MAC_ADDRパラメータ**は、パラメータのタイプをMACアドレスとして定義するために使用します。これにより、文字列、整数、IPアドレスなどの別のタイプのエンドユーザ入力が防止されます。

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

オプションキーワード

このshowコマンドでは、オプションで1つのキーワード**[mac]を使用できます**。macという単語の前後の角カッコは、このキーワードがオプションであることを示します。「Show all misprogrammed MAC addresses on this device」というヘルパーストリングがキーワードに追加されています。

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

オプションパラメータ

このshowコマンドでは、オプションで1つのパラメータ**[<mac>]を使用できます**。「< mac >」の前後の角カッコは、このパラメータがオプションであることを示します。macという単語の周りの角カッコは、これがパラメータであることを示します。パラメータに、プログラミングの誤りをチェックするためのMACアドレスのヘルパ文字列が追加されます。**nx_cmd.updateParam()メソッドのnx_sdk_py.P_MAC_ADDRパラメータ**は、パラメータのタイプをMACアドレスとして定義するために使用します。これにより、文字列、整数、IPアドレスなどの別のタイプのエンドユーザ入力が防止されます。

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

単一のキーワードとパラメータ

このshowコマンドは、単一のキーワードmacの直後にパラメータ**<mac-address>を指定します**。「mac-address」という単語の周囲の角カッコは、これがパラメータであることを示します。Check MAC address for misprogrammingのヘルパーストリングがキーワードに追加されます。パラメータに、プログラミングの誤りをチェックするためのMACアドレスのヘルパ文字列が追加さ

れます。nx_cmd.updateParam()メソッドのnx_sdk_py.P_MAC_ADDRパラメータは、パラメータのタイプをMACアドレスとして定義するために使用されます。これにより、文字列、整数、IPアドレスなどの別のタイプのエンドユーザ入力が防止されます。

```
 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)
```

複数のキーワードとパラメータ

このshowコマンドは、2つのキーワードのいずれかを使用できます。両方のキーワードの後に2つの異なるパラメータがあります。第1キーワードmacはパラメータ<mac-address>を有し、第2キーワードipはパラメータ<ip-address>を有する。「mac-address」および「ip-address」という単語の前後の角カッコ([()])は、これらがパラメータであることを示します。Check MAC address for misprogrammingのヘルパーストリングがmacキーワードに追加されます。<mac-address>パラメータに、プログラミングの誤りをチェックするためのMACアドレスのヘルパー文字列を追加します。nx_cmd.updateParam()メソッドのnx_sdk_py.P_MAC_ADDRパラメータは、MACアドレスとして<mac-address>パラメータのタイプを定義するために使用されます。これにより、エンドユーザは文字列、整数、IPアドレスなどの別のタイプを入力できません。Check IP address for misprogrammingのヘルパーストリングがipキーワードに追加されます。<ip-address>パラメータに、プログラミングの誤りをチェックするためのIPアドレスのヘルパー文字列を追加します。nx_cmd.updateParam()メソッドのnx_sdk_py.P_IP_ADDR.パラメータは、IPアドレスとして<ip-address>パラメータの型を定義するために使用されます。これにより、エンドユーザは文字列、整数、IPアドレスなどの別の型を入力できません。

```
 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)
```

オプションのキーワードを含む複数のキーワードとパラメータ

このshowコマンドは、2つのキーワードのいずれかを使用できます。両方のキーワードの後に2つの異なるパラメータがあります。第1キーワードmacは<mac-address>のパラメータを有し、第2キーワードipは<ip-address>のパラメータを有する。「mac-address」および「ip-address」という単語の前後の角カッコ([()])は、これらがパラメータであることを示します。Check MAC address for misprogrammingのヘルパーストリングがmacキーワードに追加されます。<mac-address>パラメータに、プログラミングの誤りをチェックするためのMACアドレスのヘルパー文字列を追加します。nx_cmd.updateParam()メソッドのnx_sdk_py.P_MAC_ADDRパラメータは、MACアドレスとして<mac-address>パラメータのタイプを定義するために使用されます。これにより、エンドユーザは文字列、整数、IPアドレスなどの別のタイプを入力できません。Check IP address for misprogrammingのヘルパーストリングがipキーワードに追加されます。<ip-address>パラメータに、プログラミングの誤りをチェックするためのIPアドレスのヘルパー文字列を追加します。nx_cmd.updateParam()メソッドのnx_sdk_py.P_IP_ADDR.パラメータは、IPアドレスとして<ip-address>パラメータの型を定義するために使用されます。これにより、エンドユーザは文字列、整数、IPアドレスなどの別の型を入力できません。このshowコマンドでは、オプションでキーワード[clear]を使用できます。このオプションのキーワードに、誤ってプログラムされたことが検出されたアドレスをクリアするヘルパーストリングが追加されます。

```
 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")
```

オプションのパラメータを持つ複数のキーワードとパラメータ

このshowコマンドは、2つのキーワードのいずれかを使用できます。両方のキーワードの後に2つの異なるパラメータがあります。第1キーワードmacは<mac-address>のパラメータを有し、第2キーワードipは<ip-address>のパラメータを有する。「mac-address」および「ip-address」という単語の前後の角カッコ([()])は、これらがパラメータであることを示します。Check MAC address for misprogrammingのヘルパーストリングがmacキーワードに追加されます。<mac-address>パラメータに、プログラミングの誤りをチェックするためのMACアドレスのヘルパー文字列を追加します。nx_cmd.updateParam()メソッドのnx_sdk_py.P_MAC_ADDRパラメータは、MACアドレスとして<mac-address>パラメータのタイプを定義するために使用されます。これにより、エンドユーザは文字列、整数、IPアドレスなどの別のタイプを入力できません。Check IP address for misprogrammingのヘルパーストリングがipキーワードに追加されます。<ip-address>パラメータに、プログラミングの誤りをチェックするためのIPアドレスのヘルパー文字列を追加します。nx_cmd.updateParam()メソッドのnx_sdk_py.P_IP_ADDR.パラメータは、IPアドレスとして<ip-address>パラメータの型を定義するために使用されます。これにより、エンドユーザは文字列、整数、IPアドレスなどの別の型を入力できません。このshowコマンドでは、オプションでパラメータ[<module>]を使用できます。ヘルパー文字列Only clear addresses on specified moduleがこのオプションパラメータに追加されます。

```
 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)
```

NX-SDKを使用したPythonアプリケーションのデバッグ

NX-SDK Pythonアプリケーションを作成したら、しばしばデバッグが必要になります。NX-SDKは、コードに構文エラーがある場合に通知しますが、Python NX-SDKライブラリはSWIGを使用してC++ライブラリをPythonライブラリに変換するため、コード実行時に発生した例外は次のようなアプリケーションコアダンプになります。

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

このエラーメッセージの曖昧な性質により、Pythonアプリケーションをデバッグするベストプラクティスは、sdk.getTracer()メソッドによって返されるNxTraceオブジェクトを使用してデバッグメッセージをsyslogに記録することになります。これは次のように示されています。

```
#! /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")
```

デバッグメッセージをsyslogに記録するオプションがない場合は、print文を使用して、Bashシェルの/isan/bin/pythonバイナリを介してアプリケーションを実行する方法も使用します。ただし、これらのprint文からの出力は、この方法で実行した場合にのみ表示されます。VSHシェルを介してアプリケーションを実行しても出力は生成されません。print文の使用例を次に示します。

```
#! /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")
```

NX-SDKを使用したPythonアプリケーションの導入

PythonアプリケーションがBashシェルで完全にテストされ、展開の準備ができたら、アプリケーションはVSHを使用して実稼働にインストールする必要があります。これにより、デバイスのリロード時、またはデュアルスーパーバイザシナリオでシステムのスイッチオーバーが発生した場合に、アプリケーションが保持されます。VSHを介してアプリケーションを展開するには、NX-SDKおよびENXOS SDKビルド環境を使用してRPMパッケージを作成する必要があります。Cisco DevNetは、RPMパッケージを簡単に作成できるDockerイメージを提供します。

> **注：**特定のオペレーティングシステムにDockerをインストールする方法については、Dockerのインストールマニュアルを参照してください。

Docker対応のホストでは、**docker pull dockercisco/nxsdk:<tag>コマンドを使用して任意のイメージバージョンをプルします。**ここで**<tag>は選択したイメージバージョンのタグです。**利用可能なイメージバージョンと対応するタグはこちらで確認できます。これは、次のv1タグで**示され**ています。

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

このイメージから**nxsdk**という名前のコンテナを起動し、それにアタッチします。選択したタグ

が異なる場合は、タグをv1に置き換えます。

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

最新バージョンのNX-SDKに更新し、NX-SDKディレクトリに移動し、gitから最新ファイルを取得します。

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

古いバージョンのNX-SDKを使用する必要がある場合は、git clone -b v<version> https://github.com/CiscoDevNet/NX-SDK.gitコマンドを使用して各バージョンタグを使用してNX-SDKブランチをクローニングできます。<version>は、必要なNX-SDKのバージョンです。これは、NX-SDK v1.0.0で示されています。

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

次に、PythonアプリケーションをDockerコンテナに転送します。これを行う方法はいくつかあります。

- Dockerコンテナを終了し（コンテナを停止し、もう一度起動する必要があります）、PythonアプリケーションをDockerホストに転送し、docker cpコマンドを使用してホストからコンテナにアプリケーションをコピーします。これは、Pythonアプリケーションが/app/python_app.pyのDockerホストに転送されたことを前提に、ここで説明します。

```
 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
```

- Pythonアプリケーションの内容をシステムのクリップボードにコピーし、vimを使用してDockerコンテナに作成したファイルに内容を貼り付けます。

次に、/NX-SDK/scripts/にあるrpm_gen.pyスクリプトを使用して、PythonアプリケーションからRPMパッケージを作成します。このスクリプトには、必要な引数が1つと、必要なスイッチが2つあります。

- Pythonアプリケーションのファイル名。たとえば、python_app.pyという名前のファイル内のPythonアプリケーションは、python_app.pyという引数を受け取ります。このファイル名は、後でNX-SDKのアプリケーション名として使用され、このアプリケーションによって作成されたコマンドを参照するためにNX-OSでも使用されます。

  注：ファイル名には.pyのようなファイル拡張子を含める必要はありません。この例では、ファイル名がpython_app.pyではなくpython_appである場合、RPMパッケージは問題なく生成されます。

- -sスイッチは、前述のファイル名の場所に至る絶対ファイルパスの引数を取ります。たとえば、python_app.pyが/root/に位置する場合、正しい引数は-s /root/です。

• -uスイッチは、ソースファイル名が実行可能ファイル名と同じであることを示します。

ここでは、rpm_gen.pyスクリプトの使用について説明します。

```
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パッケージへのファイルパスは、rpm_gen.pyスクリプト出力の最終行に示されています。このファイルは、アプリケーションを実行するNexusデバイスに転送できるように、Dockerコンテナからホストにコピーする必要があります。Dockerコンテナを終了した後は、docker cp <container>:<container_filepath> <host_filepath>コマンドを使用して簡単に実行できます。ここで、<container>はNX-SDK Dockerコンテナの名前（この場合はnxsdk）で、<container_filepath>はコンテナ内内のRPMのパッケージのの全体です - SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm)および<host_filepath>は、RPMパッケージが転送されるDockerホスト上の完全なファイルパスです(この場合は/root/)。 次に、このコマンドを示します。

```
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
```

このRPMパッケージをNexusデバイスに転送するには、お好みのファイル転送方法を使用します。RPMパッケージがデバイスにインストールされたら、SMUと同様にインストールしてアクティブにする必要があります。これは、RPMパッケージがデバイスのブートフラッシュに転送されたことを前提として、次のように示されています。

```
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：install addコマンドを使用してRPMパッケージをインストールすると、ストレージデバイスとパッケージの正確なファイル名を含めます。インストール後にRPMパッケージをアクティブ化する場合は、ストレージデバイスとファイル名を含めないでください。パッケージ自体の名前を使用してください。show install inactiveコマンドを使用して、パッケージ名を確認できます。

RPMパッケージがアクティブになると、nxsdk service <application-name>設定コマンドを使用してNX-SDKでアプリケーションを起動できます。ここで、<application-name>はrpm_gen.pyスクリプトが使用された際に定義されたPythonのファイル名（そしてアプリケーション）です。これは次のように示されています。

```
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
```

アプリケーションが起動していて、**show nxsdk internal service**コマンドを使用して実行を開始したことを確認できます。

```
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
```

また、このアプリケーションで作成されたカスタムCLIコマンドがNX-OSでアクセスできることを確認できます。

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

関連情報

- NX-SDK GitHub
- Cisco Nexus 9000シリーズNX-OSプログラマビリティガイド、リリース7.x
- Cisco Nexus 3000シリーズNX-OSプログラマビリティガイド、リリース7.x
- Cisco Nexus 3500シリーズNX-OSプログラマビリティガイド、リリース7.x
- 『Network Programmability and Automation with Cisco Nexus 9000 Series Switches White Paper』
- Cisco Open NX-OSによるプログラマビリティと自動化(PDF)
- テクニカル サポートとドキュメント – Cisco Systems