Desarrollar, depurar e implementar aplicaciones de NX-SDK en switches Nexus 3000/9000

Contenido

Introducción

Este documento proporciona un flujo de trabajo para el desarrollo de aplicaciones de Python con el Kit de desarrollo de software (SDK) de Cisco NX con el uso de Cisco NX-OS en las plataformas Nexus 3000 y Nexus 9000.

Prerequisites

Requirements

No hay requisitos específicos para este documento.

Componentes Utilizados

La información que contiene este documento se basa en las siguientes versiones de software y

hardware.

- Este documento utiliza NX-SDK v1.0.0 y NX-SDK v1.5.0
- NX-SDK v1.0.0 se puede utilizar en la plataforma Nexus 9000 a partir de NX-OS versión 7.0(3)I6(1) y la plataforma Nexus 3000 a partir de NX-OS versión 7.0(3)I7(1)
- NX-SDK v1.5.0 se puede utilizar tanto en la plataforma Nexus 9000 como en la plataforma Nexus 3000 a partir de NX-OS versión 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.

Antecedentes

Cisco NX-SDK permite el desarrollo de aplicaciones personalizadas que se pueden ejecutar de forma nativa en Cisco NX-OS en plataformas Nexus 9000 y Nexus 3000. NX-SDK ofrece a los clientes la posibilidad de crear sus propios comandos y resultados CLI, generar registros del sistema personalizados en respuesta a eventos específicos, transmitir telemetría personalizada y mucho más.

NX-SDK cuenta con una API de C++, que se traduce a otros idiomas con el uso de Simplified Wrapper and Interface Generator (SWIG). Esto permite al cliente utilizar NX-SDK en cualquier idioma que elija. Este documento demuestra la implementación de funciones comunes de NX-SDK en Python, así como un flujo de trabajo para que los clientes desarrollen sus propias aplicaciones de NX-SDK Python.

Desarrollar una aplicación Python con NX-SDK

Habilitar NX-SDK

Para que se ejecute cualquier aplicación NX-SDK, primero debe habilitarse la función NX-SDK en el dispositivo:

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

Crear un archivo Python

Se puede crear y editar un archivo Python con el shell NX-OS Bash. Para que se utilice el shell Bash, primero debe estar habilitado en el dispositivo:

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

Ingrese el shell Bash y utilice el editor de texto vi para crear y editar el archivo Python:

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

Nota: La práctica recomendada es crear archivos Python en el **/isan/bin/** directorio. Los archivos Python necesitan permisos de ejecución para ejecutarse - no coloque los archivos Python en el directorio **/bootflash** o en cualquiera de sus subdirectorios.

Nota: No es necesario crear y editar archivos Python a través de NX-OS. El desarrollador puede crear la aplicación utilizando su entorno local y transferir los archivos completados al dispositivo mediante un protocolo de transferencia de archivos de su elección. Sin embargo, puede ser más eficaz para el desarrollador depurar y resolver problemas de su secuencia de comandos mediante las utilidades NX-OS.

Implementar componentes NX-SDK

Se recomienda que los desarrolladores comiencen a crear su aplicación NX-SDK Python a partir de la plantilla CliPyApp personalizada en Cisco DevNet NX-SDK GitHub. Estas secciones de este documento se refieren a componentes necesarios con el uso de sus nombres dentro de esta plantilla.

Hay cuatro componentes principales necesarios para una aplicación NX-SDK Python:

1. NX-SDK se debe importar a la aplicación mediante la instrucción **import nx_sdk_py**.
2. Función (normalmente denominada **sdkThread**) que inicia la aplicación NX-SDK y modifica varias opciones relacionadas con la aplicación.
3. La creación de comandos CLI personalizados, así como la definición de sintaxis de comandos CLI personalizados dentro de la función **sdkThread**.
4. Clase denominada **pyCmdHandler** con un método denominado **postCliCb**, que procesa los comandos CLI personalizados agregados por la aplicación NX-SDK.

sdkThread (función)

La función sdkThread se inicializa, agrega funcionalidad a e inicia la aplicación NX-SDK. La función no requiere que se le pase ningún parámetro. Todas las aplicaciones de NX-SDK de Python requieren tres métodos de la biblioteca nx_sdk_py a los que se debe llamar:

1. **nx_sdk_py.NxSdk.getSdkInst(len(sys.argv), sys.argv)** devuelve un objeto de instancia de SDK o devuelve None. Si se devuelve un objeto de instancia de SDK, la aplicación NX-SDK se ha registrado correctamente en la infraestructura NX-OS. Si se devuelve Ninguno, se producen errores en el momento del proceso de registro y aparece una entrada de registro de errores en el syslog del dispositivo. Este método se documenta aquí.

   Nota: A partir de NX-SDK v1.5.0, un tercer parámetro booleano se puede pasar al método **NxSdk.getSdkInst**, que habilita las excepciones avanzadas cuando es True y deshabilita las excepciones avanzadas cuando es False. Este método se documenta aquí .

1. **sdk.startEventLoop()**, donde **sdk** es el objeto de instancia de SDK devuelto por el método **nx_sdk_py.NxSdk.getSdkInst()**. Este método inicia la aplicación NX-SDK y le permite

interactuar con la infraestructura NX-OS. Este método se documenta aquí .

2. **nx_sdk_py.NxSdk.__swig_Destruc_(sdk)** método, donde **sdk** es el objeto de instancia de SDK devuelto por el **método nx_sdk_py.NxSdk.getSdkInst()** previamente explicado. Este método situado al final de la función **sdkThread** permite la salida correcta de la aplicación NX-SDK.

Algunos de los métodos más utilizados son:

- **sdk.getTracer()**, donde **sdk** es el objeto de instancia de SDK devuelto por el método **nx_sdk_py.NxSdk.getSdkInst()**. Este método devuelve un objeto NxTrace que se puede utilizar para generar registros del sistema personalizados, así como eventos de registro y errores en el historial de eventos de la aplicación. Los syslogs personalizados aparecerán en el syslog del dispositivo (visible a través del comando **show logging logfile**) mientras que los eventos registrados en el historial de eventos de la aplicación son visibles a través de los comandos **show <application-name> nxsdk event-history events** o **show <application-name> nxsdk event-history errors**. Este método se documenta aquí. El objeto NxTrace devuelto por este método y sus métodos asociados se documentan aquí. Por ejemplo, puede ver el historial de eventos de una aplicación denominada **Transceiver_DOM.py** mediante los comandos **show Transceiver_DOM.py nxsdk event-history events** y **show Transceiver_DOM.py nxsdk event-history errors**.
- **sdk.getCliParser()**, donde **sdk** es el objeto de instancia de SDK devuelto por el método **nx_sdk_py.NxSdk.getSdkInst()**. Este método devuelve un objeto NxCliParser, que se puede utilizar para ejecutar los comandos CLI que ya existen a través de Python, así como para crear comandos CLI personalizados. Este método se documenta aquí. El objeto NxCliParser devuelto por este método y sus métodos asociados se documentan aquí .
- **cliP.execShowCmd("cmd", return_type)**, donde **cliP** es el objeto NxCliParser devuelto por el **método sdk.getCliParser()**, **cmd** es el comando show que desea ejecutar encapsulado entre comillas, y el **return_type es el formato de datos que se va a producir.** El formato de datos puede ser **R_TEXT**, **R_JSON** o **R_XML**. Este método se documenta aquí .

  Nota: Los formatos de datos **R_JSON** y **R_XML** sólo funcionan si el comando admite resultados en esos formatos. En NX-OS, puede verificar si un comando admite resultados en un formato de datos determinado canalizando el resultado al formato de datos solicitado. Si el comando piped devuelve un resultado significativo, entonces se soporta ese formato de datos. Por ejemplo, si ejecuta **show mac address-table dynamic | json** en NX-OS devuelve el resultado de JSON y, a continuación, el formato de datos R_JSON también se admite en NX-SDK.

- **cliP.execConfigCmd(cmd_filename)**, donde **cliP** es el objeto NxCliParser devuelto por el método **sdk.getCliParser()**, y **cmd_filename es la ruta de archivo absoluta a un archivo que contiene comandos separados por líneas.** Este método devuelve una cadena que indica el éxito de la ejecución de comandos; si "SUCCESS" está en la cadena, todos los comandos se ejecutan correctamente. De lo contrario, la cadena contiene la excepción que describe por qué los comandos no se ejecutaron. Este método se documenta aquí.

Algunos métodos opcionales que pueden resultar útiles son:

- **sdk.setAppDesc('description string')**, donde **sdk** es el objeto de instancia de SDK devuelto por el método **nx_sdk_py.NxSdk.getSdkInst()**. Este método establece la descripción de la aplicación NX-SDK. La descripción se muestra en el menú de ayuda contextual de NX-OS al que se accede mediante un signo de interrogación en la CLI. Este método se documenta

aquí. Por ejemplo, una aplicación denominada **Transceiver_DOM.py**, una descripción de la aplicación Devuelve todas las interfaces con transceptores compatibles con DOM insertados aparece en la ayuda contextual de NX-OS de la siguiente manera:

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

- **sdk.getAppName()**, donde **sdk** es el objeto de instancia de SDK devuelto por el método **nx_sdk_py.NxSdk.getSdkinst()**. Este método devuelve el nombre de la aplicación Python. Este método se documenta aquí.

Crear comandos CLI personalizados

En una aplicación Python con el uso de NX-SDK, se crean y definen comandos CLI personalizados dentro de la función sdkThread. Hay dos tipos de comandos: **Mostrar** comandos y **Configurar** comandos.

1. **Los** comandos **Show** muestran información sobre el dispositivo, su configuración o su entorno.
2. **Los** comandos de **configuración** cambian la configuración del dispositivo, lo que modifica la forma en que el dispositivo reacciona a la red circundante.

Estos dos métodos permiten la creación de los comandos show y los comandos config respectivamente:

- **cliP.newShowCmd("cmd_name", "sintaxis")**, donde **cliP** es el objeto NxCliParser devuelto por el **método sdk.getCliParser()**, **cmd_name es un nombre único para el comando interno de la aplicación NX-SDK personalizada, y** sintaxisdescribe lo que puede se utilizará en el comando. Este método devuelve un objeto NxCliCmd, que se documenta aquí. Este método se documenta aquí.

  Nota: Este comando es una subclase de cliP.newCliCmd("cmd_type", "cmd_name", "sintaxis") donde cmd_type es CONF_CMD o **SHOW_CMD** (dependiendo del tipo de comando que se configure), cmd_nameSDK es un nombre único para el comando interno de la aplicación NX personalizada y sintaxis describe las palabras clave y los parámetros que se pueden utilizar en el comando. Debido a esto, la documentación de API para este comando podría ser más útil para la referencia.

- **cliP.newConfigCmd("cmd_name", "sintaxis")**, donde **cliP** es el objeto NxCliParser devuelto por el método **sdk.getCliParser()**, **cmd_name** es un nombre único para el comando interno de la aplicación NX-SDK personalizada, y sintaxisdescribe lo que puede se utilizará en el comando. Este método devuelve un objeto NxCliCmd, que se documenta aquí . Este método se documenta aquí .

  Nota: Este comando es una subclase de cliP.newCliCmd("cmd_type", "cmd_name", "sintaxis") donde cmd_type es CONF_CMD o **SHOW_CMD** (depende del tipo de comando configurado), cmd_name es un nombre único para el comando interno del NX-NX-CMD personalizado y sintaxis describe las palabras clave y los parámetros que se pueden utilizar en el comando. Debido a esto, la documentación de API para este comando podría ser más útil para la referencia.

Ambos tipos de comandos tienen dos componentes diferentes: Parámetros y palabras clave:

1. **Los parámetros** son valores utilizados para cambiar los resultados del comando. Por ejemplo, en el comando **show ip route 192.168.1.0**, hay una palabra clave **route** seguida de un parámetro que acepta una dirección IP, que especifica que sólo se deben mostrar las rutas que incluyen la dirección IP proporcionada.

2. **Las palabras clave** cambian los resultados del comando sólo a través de su presencia. Por ejemplo, en el comando **show mac address-table dynamic**, hay una **palabra clave dynamic**, que especifica que sólo se mostrarán las direcciones MAC aprendidas dinámicamente.

Ambos componentes se definen en la sintaxis de un comando NX-SDK cuando se crea. Existen métodos para que el objeto NxCliCmd modifique la implementación específica de ambos componentes.

- **nx_cmd.updateParam("<parámetro>", "help_str", tipo), donde nx_cmd es el objeto NxCliCmd devuelto por los cliP.newShowCmd() o los métodos cliP.newConfigCmd(), <parámetro>** es el nombre del parámetro de comando que se puede modificar encerrado los corchetes angulares (<>), **help_str establece la cadena de ayuda del comando personalizado y se muestra en el menú de ayuda contextual de NX-OS al que se accede a través de un signo de interrogación en la CLI, y type es el tipo del parámetro.** Los parámetros opcionales adicionales para este método están disponibles y documentados aquí. Estos son tipos válidos para los parámetros que se pueden especificar en el argumento type:

  P_INTEGER: especifica cualquier número entero**P_STRING**: especifica cualquier cadena**P_INTERFACE**: especifica cualquier interfaz de red**P_IP_ADDR**: especifica cualquier dirección IP**P_MAC_ADDR**: especifica cualquier dirección MAC**P_VRF**: especifica cualquier instancia de routing y reenvío virtual (VRF)
- **nx_cmd.updateKeyword("palabra clave", "help_str", is_key), donde nx_cmd es el objeto NxCliCmd devuelto por los métodos cliP.newShowCmd() o cliP.newConfigCmd(), la palabra clave** es el nombre de la palabra clave de comando que desea modificar, **help_str establece la cadena de ayuda del comando personalizado y se muestra en el menú de ayuda contextual de NX-OS al que se accede a través de un signo de interrogación en la CLI, y is_key es un valor booleano opcional cuyo valor predeterminado es False.** Si **is_key** es True, la configuración única creada por el comando con el uso de esta palabra clave no sobrescribe otra configuración única creada por el comando. Si **is_key** es False, la configuración creada por el comando con el uso de esta palabra clave sobrescribe la otra configuración creada por el comando. Este método se documenta aquí .

Para ver ejemplos de código de los componentes de comandos más utilizados, vea la sección Ejemplos de Comandos de CLI Personalizados de este documento.

Una vez creados los comandos CLI personalizados, se debe crear un objeto de la clase **pyCmdHandler** descrita más adelante en este documento y configurarlo como el objeto controlador de devolución de llamada CLI para el objeto NxCliParser. Esto se demuestra de la siguiente manera:

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

A continuación, el objeto NxCliParser debe agregarse al árbol de analizador de CLI de NX-OS para que el usuario pueda ver los comandos de CLI personalizados. Esto se realiza con el comando **cliP.addToParseTree()**, donde **cliP** es el objeto NxCliParser devuelto por el método **sdk.getCliParser()**.

Ejemplo de la Función sdkThread

A continuación se muestra un ejemplo de una función sdkThread típica con el uso de las funciones explicadas anteriormente. Esta función (entre otras, en una aplicación personalizada típica de NX-SDK Python) utiliza variables globales, que se crean instancias de la ejecución de secuencias de comandos.

```
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)
```

Clase pyCmdHandler

La clase **pyCmdHandler** se hereda de la clase **NxCmdHandler** dentro de la biblioteca nx_sdk_py. El método **postCliCb(self, climd)** definido en la clase **pyCmdHandler** se llama siempre que se originan comandos CLI de una aplicación NX-SDK. Por lo tanto, el método **postCliCb(self, climd)** es donde se define el comportamiento de los comandos CLI personalizados definidos en la función sdkThread en el dispositivo.

La función **postCliCb(self, clmd)** devuelve un valor booleano. Si se devuelve **True**, se presume que el comando se ha ejecutado correctamente. **False** debe devolverse si el comando no se ejecutó correctamente por ninguna razón.

El parámetro **climd** utiliza el nombre único definido para el comando cuando se creó en la función sdkThread. Por ejemplo, si crea un nuevo comando **show** con un nombre único de **show_xcvr_dom**, se recomienda hacer referencia a este comando con el mismo nombre en la función **postCliCb(self, climd)** después de verificar si el nombre del argumento clmd contiene **show_xcvr_dom**. Se demuestra aquí:

```
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()
```

Si se crea un comando que utiliza parámetros, es muy probable que necesite utilizar esos parámetros en algún momento de la función **postCliCb(self, climd)**. Esto se puede hacer con el método **climd.getParamValue("<parámetro>")**, donde **<parámetro>** es el nombre del parámetro de comando que desea obtener el valor de entre corchetes angulares (**<>**). Este método se documenta aquí. Sin embargo, el valor devuelto por esta función debe convertirse al tipo que necesita. Esto se puede hacer con estos métodos:

- **nx_sdk_py.void_to_int** convierte un valor en un tipo entero.
- **nx_sdk_py.void_to_string** convierte un valor en un tipo de cadena.

La función **postCliCb(self, clmd)** (o cualquier función subsiguiente) también será normalmente la que se imprima el resultado del comando show en la consola. Esto se realiza con el método **clmd.printConsole()**.

> **Nota:** Si la aplicación encuentra un error, una excepción no controlada o de algún otro modo sale repentinamente, el resultado de la función **clmd.printConsole()** no se mostrará en absoluto. Por esta razón, la mejor práctica al depurar su aplicación de Python es registrar los mensajes de depuración en el syslog con el uso de un objeto NxTrace devuelto por el método **sdk.getTracer()**, o utilizar instrucciones de impresión y ejecutar la aplicación a través del **/isan/bin/python** binario del shell Bash.

Ejemplo de Clase pyCmdHandler

El código siguiente sirve como ejemplo para la clase **pyCmdHandler** descrita anteriormente. Este código se toma del archivo ip_motion.py de la aplicación ip-motion NX-SDK disponible aquí. El

propósito de esta aplicación es realizar un seguimiento del movimiento de una dirección IP definida por el usuario a través de las interfaces de un dispositivo Nexus. Para hacerlo, el código encuentra la dirección MAC de la entrada de dirección IP a través del parámetro **<ip>** dentro de la memoria caché ARP del dispositivo, luego verifica en qué VLAN reside esa dirección MAC usando la tabla de direcciones MAC del dispositivo. Con este MAC y VLAN, el comando **show system internal l2fm l2dbg macdb address <mac> vlan <vlan>** muestra una lista de índices de interfaz SNMP con los que esta combinación se ha asociado recientemente. El código luego utiliza el comando **show interface snmp-ifindex** para traducir los índices de interfaz SNMP recientes a nombres de interfaz legibles 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
```

Ejemplos de Sintaxis de Comandos CLI Personalizados

Esta sección muestra algunos ejemplos del parámetro de sintaxis utilizado cuando se crean comandos CLI personalizados con los métodos **cliP.newShowCmd()** o **cliP.newConfigCmd()**, donde **cliP** es el objeto NxCliParser devuelto por el método **sdk.getCliParser()**.

> **Nota:** La compatibilidad con la sintaxis con los paréntesis de apertura y cierre ("(" y ")") se introduce en NX-SDK v1.5.0, incluida en NX-OS versión 7.0(3)I7(3). Se supone que el usuario utiliza NX-SDK v1.5.0 cuando sigue cualquiera de estos ejemplos dados que incluyen sintaxis utilizando paréntesis de apertura y cierre.

Palabra clave única

Este comando show toma una sola palabra clave mac y agrega una cadena de ayuda de Shows all misprogramado MAC Addresses en este dispositivo a la palabra clave.

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

Parámetro único

Este comando show toma un único parámetro **<mac>**. Los corchetes angulares que rodean la palabra mac significan que se trata de un parámetro. Se agrega al parámetro una cadena auxiliar de dirección MAC para comprobar si hay programación incorrecta. El **parámetro nx_sdk_py.P_MAC_ADDR** del método **nx_cmd.updateParam()** se utiliza para definir el tipo del parámetro como una dirección MAC, que impide que el usuario final introduzca otro tipo, como una cadena, un número entero o una dirección 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)
```

Palabra clave opcional

Este comando show puede opcionalmente tomar una sola palabra clave **[mac]**. Los corchetes de cierre alrededor de la palabra mac significan que esta palabra clave es opcional. Una cadena de ayuda de Muestra todas las direcciones MAC mal programadas en este dispositivo se agrega a la palabra clave.

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

Parámetro opcional

Este comando show puede opcionalmente tomar un único parámetro **[<mac>]**. Los corchetes de

cierre alrededor de la palabra **<** mac **>** significan que este parámetro es opcional. Los corchetes angulares que rodean la palabra mac significan que se trata de un parámetro. Se agrega al parámetro una cadena auxiliar de dirección MAC para comprobar si hay programación incorrecta. El **parámetro nx_sdk_py.P_MAC_ADDR** del método **nx_cmd.updateParam()** se utiliza para definir el tipo del parámetro como una dirección MAC, que impide que el usuario final introduzca otro tipo, como una cadena, un número entero o una dirección 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)
```

Palabra clave y parámetro únicos

Este comando show toma una sola palabra clave mac seguida inmediatamente por el parámetro **<mac-address>**. Los corchetes angulares que rodean la palabra mac-address significan que se trata de un parámetro. Se agrega a la palabra clave una cadena auxiliar de verificación de dirección MAC para la programación incorrecta. Se agrega al parámetro una cadena auxiliar de dirección MAC para comprobar si hay programación incorrecta. El **parámetro nx_sdk_py.P_MAC_ADDR** del método **nx_cmd.updateParam()** se utiliza para definir el tipo del parámetro como dirección MAC, que impide la entrada de otro tipo por parte del usuario final, como una cadena, un número entero o una dirección 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)
```

Palabras clave y parámetros múltiples

Este comando show puede tomar una de dos palabras clave, ambas con dos parámetros diferentes a continuación. La primera palabra clave mac tiene un parámetro de **<mac-address>** , y la segunda palabra clave ip tiene un parámetro de **<ip-address>**. Los corchetes angulares de las palabras mac-address e ip-address significan que son parámetros. Se agrega una cadena auxiliar de verificación de dirección MAC para la mala programación a la palabra clave mac. Se agrega una cadena auxiliar de dirección MAC para comprobar si hay programación incorrecta al parámetro **<mac-address>**. El **parámetro nx_sdk_py.P_MAC_ADDR** del método **nx_cmd.updateParam()** se utiliza para definir el tipo del parámetro **<mac-address>** como dirección MAC, que impide la entrada de otro tipo por parte del usuario final, como una cadena, un número entero o una dirección IP. Se agrega una cadena de ayuda de Verificar la dirección IP para la mala programación a la palabra clave ip. Se agrega una cadena auxiliar de dirección IP para comprobar si hay programación incorrecta al parámetro **<ip-address>**. El **parámetro nx_sdk_py.P_IP_ADDR** del método **nx_cmd.updateParam()** se utiliza para definir el tipo del parámetro **<ip-address>** como dirección IP, que impide la entrada de otro tipo por parte del usuario final, como una cadena, un número entero o una dirección 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)
```

Varias palabras clave y parámetros con palabra clave opcional

Este comando show puede tomar una de dos palabras clave, ambas con dos parámetros diferentes a continuación. La primera palabra clave mac tiene un parámetro de **<mac-address>**, y la segunda palabra clave ip tiene un parámetro de **<ip-address>**. Los corchetes angulares de las palabras mac-address e ip-address significan que son parámetros. Se agrega una cadena auxiliar de verificación de dirección MAC para la mala programación a la palabra clave mac. Se agrega una cadena auxiliar de dirección MAC para comprobar si hay programación incorrecta al parámetro **<mac-address>**. El **parámetro nx_sdk_py.P_MAC_ADDR** del método **nx_cmd.updateParam()** se utiliza para definir el tipo del parámetro **<mac-address>** como dirección MAC, que impide la entrada de otro tipo por parte del usuario final, como una cadena, un número entero o una dirección IP. Se agrega una cadena de ayuda de Verificar la dirección IP para la mala programación a la palabra clave ip. Se agrega una cadena auxiliar de dirección IP para comprobar si hay programación incorrecta al parámetro **<ip-address>**. El **parámetro nx_sdk_py.P_IP_ADDR** del método **nx_cmd.updateParam()** se utiliza para definir el tipo del parámetro **<ip-address>** como dirección IP, que impide la entrada de otro tipo por parte del usuario final, como una cadena, un número entero o una dirección IP. Este comando show opcionalmente podría tomar una palabra clave [clear]. Una cadena de ayuda Borra las direcciones detectadas como mal programadas se agrega a esta palabra clave 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")
```

Varias palabras clave y parámetros con parámetro opcional

Este comando show puede tomar una de dos palabras clave, ambas con dos parámetros diferentes a continuación. La primera palabra clave mac tiene un parámetro de **<mac-address>**, y la segunda palabra clave ip tiene un parámetro de **<ip-address>**. Los corchetes angulares de las palabras mac-address e ip-address significan que son parámetros. Se agrega una cadena auxiliar de verificación de dirección MAC para errores de programación a la palabra clave mac. Se agrega una cadena auxiliar de dirección MAC para comprobar si hay programación incorrecta al parámetro **<mac-address>**. El **parámetro nx_sdk_py.P_MAC_ADDR** del método **nx_cmd.updateParam()** se utiliza para definir el tipo del parámetro **<mac-address>** como dirección MAC, que impide la entrada de otro tipo por parte del usuario final, como una cadena, un número entero o una dirección IP. Se agrega una cadena de ayuda de Verificar la dirección IP para la mala programación a la palabra clave ip. Se agrega una cadena auxiliar de dirección IP para comprobar si hay programación incorrecta al parámetro **<ip-address>**. El **parámetro nx_sdk_py.P_IP_ADDR** del método **nx_cmd.updateParam()** se utiliza para definir el tipo del parámetro **<ip-address>** como dirección IP, que impide la entrada de otro tipo por parte del usuario final, como una cadena, un número entero o una dirección IP. Este comando show puede tomar opcionalmente un parámetro **[<module>]**. Una cadena de ayuda Sólo se agregan direcciones despejadas en el módulo especificado a este parámetro 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 una aplicación Python con NX-SDK

Una vez creada una aplicación NX-SDK Python, a menudo tendrá que depurarse. NX-SDK le informa en caso de que haya errores de sintaxis en el código, pero como la biblioteca Python NX-SDK utiliza SWIG para traducir bibliotecas de C++ a bibliotecas de Python, cualquier excepción encontrada en el momento de la ejecución del código da como resultado un vaciado de memoria de la aplicación similar 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)
```

Debido a la naturaleza ambigua de este mensaje de error, la mejor práctica para depurar aplicaciones Python es registrar mensajes de depuración en el syslog con el uso de un objeto NxTrace devuelto por el método **sdk.getTracer()**. Esto se demuestra de la siguiente manera:

```
#! /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")
```

Si el registro de los mensajes de depuración en el syslog no es una opción, un método alternativo es utilizar sentencias de impresión y ejecutar la aplicación a través del binario **/isan/bin/python** del shell Bash. Sin embargo, el resultado de estas instrucciones de impresión sólo será visible cuando se ejecute de esta manera; ejecutar la aplicación a través del shell de VSH no produce ningún resultado. A continuación se muestra un ejemplo de uso de instrucciones de impresión:

```
#! /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")
```

Implementar una aplicación Python con NX-SDK

Una vez que una aplicación de Python se ha probado completamente en el shell de Bash y está lista para su implementación, la aplicación debe instalarse en producción a través de VSH. Esto permite que la aplicación persista cuando el dispositivo se recarga o cuando se produce el switchover del sistema en un escenario de supervisor dual. Para implementar una aplicación a través de VSH, necesita crear un paquete RPM con el uso de un entorno de compilación NX-SDK y ENXOS SDK. Cisco DevNet proporciona una imagen de Docker que permite una fácil creación de paquetes RPM.

> **Nota:** Para obtener ayuda para instalar Docker en su sistema operativo específico, consulte la documentación de instalación de Docker.

En un host compatible con Docker, tire de la versión de imagen que elija con el comando **docker pull dockercisco/nxsdk:<tag>** donde **<tag>** es la etiqueta de la versión de imagen que elija. Puede ver las versiones de imagen disponibles y sus etiquetas correspondientes aquí. Esto se demuestra con la etiqueta **v1** aquí:

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

Inicie un contenedor denominado **nxsdk** desde esta imagen y adáptelo. Si la etiqueta de su elección es diferente, sustituya **v1** por su etiqueta:

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

Actualice a la última versión de NX-SDK y navegue hasta el directorio **NX-SDK** y, a continuación, extraiga los archivos más recientes de git:

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

Si necesita utilizar una versión anterior de NX-SDK, puede clonar la rama de NX-SDK con la etiqueta de versión respectiva con el comando **git clone -b v<version>** https://github.com/CiscoDevNet/NX-SDK.git, donde **<version>** es la versión de NX-SDK que necesita. Esto se demuestra aquí con NX-SDK v1.0.0:

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

A continuación, transfiera la aplicación Python al contenedor Docker. Hay algunas maneras diferentes de hacerlo.

- Salga del contenedor Docker (que detiene el contenedor y requiere que lo inicie una vez más), transfiera la aplicación Python al host Docker y luego use el comando **docker cp** para copiar la aplicación del host al contenedor. Esto se demuestra aquí, bajo la suposición de que la aplicación Python ha sido transferida al host Docker en **/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 el contenido de la aplicación Python en el portapapeles del sistema y, a continuación, pegue el contenido en un archivo creado en el contenedor Docker con el uso de vim.

A continuación, utilice el script **rpm_gen.py** ubicado en **/NX-SDK/scripts/** para crear un paquete RPM desde la aplicación Python. Este script tiene un argumento obligatorio y dos switches necesarios:

- El nombre de archivo de la aplicación Python. Por ejemplo, una aplicación Python en un archivo denominado **python_app.py** daría como resultado un argumento de **python_app.py**. Este nombre de archivo se utilizará posteriormente como nombre de aplicación para NX-SDK y NX-OS también lo utilizará para hacer referencia a los comandos creados por esta aplicación.

  Nota: El nombre de archivo no necesita contener ninguna extensión de archivo, como **.py**. En este ejemplo, si el nombre de archivo era **python_app** en lugar de **python_app.py**, el paquete RPM se generaría sin problema.

- El switch -s toma un argumento para la ruta de archivo absoluta que lleva a dónde se encuentra el nombre de archivo mencionado anteriormente. Por ejemplo, si **python_app.py** se encuentra en **/root/**, el argumento correcto sería **-s /root/**.
- El switch -u indica que el nombre de archivo de origen es el mismo que el nombre del archivo ejecutable.

El uso del script **rpm_gen.py** se muestra aquí.

```
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
```

La trayectoria de archivo al paquete RPM se indica en la línea final de la salida del script **rpm_gen.py**. Este archivo debe copiarse del contenedor Docker en el host para que pueda transferirse al dispositivo Nexus en el que desea ejecutar la aplicación. Después de salir del contenedor de Docker, se puede hacer fácilmente con el comando docker cp <container>:<container_filepath> <host_filepath>, donde <container> es el nombre del contenedor de Docker de NX-SDK (en este caso, nxsdk), <container_filepath> es la ruta de archivo completa del paquete RPM dentro del contenedor (en este caso, **/NX-SDK /rpm/RPMS/test_python_app-1.0-1.0.x86_64.rpm**), y **<host_filepath>** es la ruta de archivo completa en nuestro host Docker donde se va a transferir el paquete RPM a (en este caso, **/root/**). Este comando se muestra aquí:

```
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
```

Transfiera este paquete RPM al dispositivo Nexus con el uso de su método preferido de transferencia de archivos. Una vez que el paquete RPM se encuentra en el dispositivo, se debe instalar y activar de forma similar a una SMU. Esto se demuestra de la siguiente manera, bajo la suposición de que el paquete RPM fue transferido a la memoria flash de inicialización del 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
```

> **Nota:** Cuando instale el paquete RPM con el comando **install add**, incluya el dispositivo de almacenamiento y el nombre de archivo exacto del paquete. Cuando active el paquete RPM después de la instalación, no incluya el dispositivo de almacenamiento ni el nombre de archivo - utilice el nombre del paquete en sí. Puede verificar el nombre del paquete con el comando **show install idle**.

Una vez que se activa el paquete RPM, puede iniciar la aplicación con NX-SDK con el comando de configuración **nxsdk service <application-name>**, donde **<application-name>** es el nombre del nombre de archivo de Python (y, posteriormente, la aplicación) que se definió cuando se utilizó anteriormente la secuencia de comandos rpm_gen.py. Esto se demuestra de la siguiente manera:

```
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
```

Puede verificar que la aplicación esté activa y que haya comenzado a ejecutarse con el 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
```

También puede verificar que los comandos CLI personalizados creados por esta aplicación estén accesibles en NX-OS:

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

Información Relacionada

- GitHub de NX-SDK

- Guía de programabilidad de Cisco Nexus serie 9000 NX-OS, versión 7.x
- Guía de programabilidad de Cisco Nexus serie 3000 NX-OS, versión 7.x
- Guía de programabilidad de Cisco Nexus serie 3500 NX-OS, versión 7.x
- Informe técnico sobre la capacidad de programación y automatización de la red con los switches Nexus de Cisco serie 9000
- Capacidad de programación y automatización con Cisco Open NX-OS (PDF en inglés)
- Soporte Técnico y Documentación - Cisco Systems