PYTHON ДЛЯ СЕТЕВЫХ ИНЖЕНЕРОВ

ПОДКЛЮЧЕНИЕ К ОБОРУДОВАНИЮ

ВВОД ПАРОЛЯ

ВВОД ПАРОЛЯ

При подключении к оборудованию вручную, как правило, пароль также вводится вручную.

При автоматизизации подключения, надо решить каким образом будет передаваться пароль:

- запрашивать пароль при старте скрипта и считывать ввод пользователя
  - минус в том, что будет видно какие символы вводит пользователь
- записывать логин и пароль в каком-то файле
  - это не очень безопасно

МОДУЛЬ GETPASS

Модуль getpass позволяет запрашивать пароль, не отображая вводимые символы:

```
In [1]: import getpass

In [2]: password = getpass.getpass()
Password:

In [3]: print(password)
testpass
```

ПЕРЕМЕННЫЕ ОКРУЖЕНИЯ

Еще один вариант хранения пароля (а можно и пользователя) - переменные окружения.

Например, таким образом логин и пароль записываются в переменные:

```
$ export SSH_USER=user
$ export SSH_PASSWORD=userpass
```

А затем, в Python, считываются значения в переменные в скрипте:

```
import os

USERNAME = os.environ.get('SSH_USER')
PASSWORD = os.environ.get('SSH_PASSWORD')
```

МОДУЛЬ REXREST

МОДУЛЬ PEXPECT

Модуль pexpect позволяет автоматизировать интерактивные подключения, такие как:

- telnet
- ssh
- ftp

Для начала, модуль pexpect нужно установить:

```
pip install pexpect
```

МОДУЛЬ REXPECT

Логика работы rexpect такая:

- запускается какая-то программа
- rexpect ожидает определенный вывод (приглашение, запрос пароля и подобное)
- получив вывод, он отправляет команды/данные
- последние два действия повторяются столько, сколько нужно

При этом, сам rexpect не реализует различные утилиты, а использует уже готовые.

МОДУЛЬ REXPECT

В rexpect есть два основных инструмента:

- функция run()
- класс spawn

PEXPECT.RUN()

Функция run() позволяет вызвать какую-то программу и вернуть её вывод.

```
In [1]: import pexpect

In [2]: output = pexpect.run('ls -ls')

In [3]: print(output)
b'total 44\r\n4 -rw-r--r-- 1 vagrant vagrant 3203 Jul 14 07:15 1_pexpect.py\r\n4 -rw-r--r-- 1 vagrant vagra

In [4]: print(output.decode('utf-8'))
total 44
4 -rw-r--r-- 1 vagrant vagrant 3203 Jul 14 07:15 1_pexpect.py
4 -rw-r--r-- 1 vagrant vagrant 3393 Jul 14 07:15 2_telnetlib.py
4 -rw-r--r-- 1 vagrant vagrant 3452 Jul 14 07:15 3_paramiko.py
```

PEXPECT.SPAWN

Класс spawn поддерживает больше возможностей. Он позволяет взаимодействовать с вызванной программой, отправляя данные и ожидая ответ.

Пример использования pexpect.spawn:

```
t = pexpect.spawn('ssh user@10.1.1.1')

t.expect('Password:')
t.sendline('userpass')
t.expect('>')
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ PEXPECT

Пример использования pexpect для подключения к оборудованию и передачи команды show (файл 1_pexpect.py):

```
import pexpect
import getpass
import sys

COMMAND = sys.argv[1]
USER = input("Username: ")
PASSWORD = getpass.getpass()
ENABLE_PASS = getpass.getpass(prompt='Enter enable password: ')

DEVICES_IP = ['192.168.100.1','192.168.100.2','192.168.100.3']
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ PEXPECT

Пример использования pexpect для подключения к оборудованию и передачи команды show (файл 1_pexpect.py):

```
for IP in DEVICES_IP:
    print("Connection to device {}".format( IP ))
    t = pexpect.spawn('ssh {}@{}'.format( USER, IP ))

    t.expect('Password:')
    t.sendline(PASSWORD)

    t.expect('>')
    t.sendline('enable')

    t.expect('Password:')
    t.sendline(ENABLE_PASS)

    t.expect('#')
    t.sendline("terminal length 0")

    t.expect('#')
    t.sendline(COMMAND)

    t.expect('#')
    print(t.before.decode('utf-8'))
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ PEXPECT

Выполнение скрипта выглядит так:

```
$ python 1_pexpect.py "sh ip int br"
Username: cisco
Password:
Enter enable secret:
Connection to device 192.168.100.1
sh ip int br
Interface              IP-Address      OK? Method Status                Protocol
FastEthernet0/0        192.168.100.1   YES NVRAM  up                    up
FastEthernet0/1        unassigned      YES NVRAM  up                    up
FastEthernet0/1.10     10.1.10.1       YES manual up                    up
FastEthernet0/1.20     10.1.20.1       YES manual up                    up
FastEthernet0/1.30     10.1.30.1       YES manual up                    up
FastEthernet0/1.40     10.1.40.1       YES manual up                    up
FastEthernet0/1.50     10.1.50.1       YES manual up                    up
FastEthernet0/1.60     10.1.60.1       YES manual up                    up
FastEthernet0/1.70     10.1.70.1       YES manual up                    up
R1
Connection to device 192.168.100.2
sh ip int br
Interface              IP-Address      OK? Method Status                Protocol
FastEthernet0/0        192.168.100.2   YES NVRAM  up                    up
FastEthernet0/1        unassigned      YES NVRAM  up                    up
FastEthernet0/1.10     10.2.10.1       YES manual up                    up
FastEthernet0/1.20     10.2.20.1       YES manual up                    up
FastEthernet0/1.30     10.2.30.1       YES manual up                    up
FastEthernet0/1.40     10.2.40.1       YES manual up                    up
FastEthernet0/1.50     10.2.50.1       YES manual up                    up
FastEthernet0/1.60     10.2.60.1       YES manual up                    up
FastEthernet0/1.70     10.2.70.1       YES manual up                    up
```

СПЕЦИАЛЬНЫЕ СИМВОЛЫ В SHELL

Pexpect не интерпретирует специальные символы shell, такие как >, |, *.

Для того чтобы, например, команда ls -ls | grep SUMMARY отработала, нужно запустить shell таким образом:

```
In [1]: import pexpect

In [2]: p = pexpect.spawn('/bin/bash -c "ls -ls | grep SUMMARY"')

In [3]: p.expect(pexpect.EOF)
Out[3]: 0

In [4]: print(p.before)
b'4 -rw-r--r-- 1 vagrant vagrant 3203 Jul 14 07:15 1_pexpect.py\r\n'

In [5]: print(p.before.decode('utf-8'))
4 -rw-r--r-- 1 vagrant vagrant 3203 Jul 14 07:15 1_pexpect.py
```

pexpect.EOF

В предыдущем примере встретилось использование pexpect.EOF.

- EOF (end of file) — конец файла

Это специальное значение, которое позволяет отреагировать на завершенние исполнения команды или сессии, которая была запущена в spawn.

PEXPECT.EOF

При вызове команды ls -ls, pexpect не получает интерактивный сеанс. Команда выполняется и всё, на этом завершается её работа.

Поэтому, если запустить её и указать в expect приглашение, возникнет ошибка:

```
In [5]: p = pexpect.spawn('/bin/bash -c "ls -ls | grep SUMMARY"')

In [6]: p.expect('nattaur')
...
EOF: End Of File (EOF). Empty string style platform.
<pexpect.pty_spawn.spawn object at 0x107100b10>
...
```

Но, если передать в expect EOF, ошибки не будет.

ВОЗМОЖНОСТИ PEXPECT.EXPECT

pexpect.expect в качестве шаблона может принимать не только строку.

Что может использоваться как шаблон в pexpect.expect:

- строка
- EOF - этот шаблон позволяет среагировать на исключение EOF
- TIMEOUT - исключение timeout (по умолчанию значение timeout = 30 секунд)
- compiled re

ВОЗМОЖНОСТИ PEXPECT.EXPECT

Еще одна очень полезная возможность pexpect.expect: можно передавать не одно значение, а список.

Например:

```
In [7]: p = pexpect.spawn('/bin/bash -c "ls -ls | grep SUMMARY"')

In [8]: p.expect(['nattaur', pexpect.TIMEOUT, pexpect.EOF])
Out[8]: 2
```

ВОЗМОЖНОСТИ PEXPECT.EXPECT

Несколько важных моментов:

- когда pexpect.expect вызывается со списком, можно указывать разные ожидаемые строки
- кроме строк, можно указывать исключения
- pexpect.expect возвращает номер элемента списка, который сработал
  - в данном случае, номер 2, так как исключение EOF находится в списке под номером два
- засчет такого формата, можно делать ответвления в программе, в зависимости от того с каким элементом было совпадение

МОДУЛЬ TELNETLIB

МОДУЛЬ TELNETLIB

Модуль telnetlib входит в стандартную библиотеку Python. Это реализация клиента telnet.

Файл 2_telnetlib.py:

```
import telnetlib
import time
import getpass
import sys

COMMAND = sys.argv[1].encode('utf-8')
USER = input("Username: ").encode('utf-8')
PASSWORD = getpass.getpass().encode('utf-8')
ENABLE_PASS = getpass.getpass(prompt='Enter enable password: ').encode('utf-8')

DEVICES_IP = ['192.168.100.1','192.168.100.2','192.168.100.3']
```

МОДУЛЬ TELNETLIB

Файл 2_telnetlib.py:

```
for IP in DEVICES_IP:
    print("Connection to device {}".format( IP ))
    t = telnetlib.Telnet(IP)

    t.read_until(b"Username:")
    t.write(USER + b'\n')

    t.read_until(b"Password:")
    t.write(PASSWORD + b'\n')
    t.write(b"enable\n")

    t.read_until(b"Password:")
    t.write(ENABLE_PASS + b'\n')
    t.write(b"terminal length 0\n")
    t.write(COMMAND + b'\n')

    time.sleep(5)

    output = t.read_very_eager().decode('utf-8')
    print(output)
```

МОДУЛЬ TELNETLIB

telnetlib очень похож на pexpect:

- t = telnetlib.Telnet(ip) - класс Telnet представляет соединение к серверу.
  - в данном случае, ему передается только IP-адрес, но можно передать и порт, к которому нужно подключаться
- read_until - похож на метод expect в модуле pexpect. Указывает до какой строки следует считывать вывод
- write - передать строку
- read_very_eager - считать всё, что получается

МОДУЛЬ TELNETLIB

Выполнение скрипта:

```
$ python 2_telnetlib.py "sh ip int br"
Username: cisco
Password:
Enter enable secret:
Connection to device 192.168.100.1

R1#terminal length 0
R1#sh ip int br
Interface              IP-Address      OK? Method Status                Protocol
FastEthernet0/0        192.168.100.1   YES NVRAM  up                    up
FastEthernet0/1        unassigned      YES NVRAM  up                    up
FastEthernet0/1.10     10.1.10.1       YES manual up                    up
FastEthernet0/1.20     10.1.20.1       YES manual up                    up
FastEthernet0/1.30     10.1.30.1       YES manual up                    up
FastEthernet0/1.40     10.1.40.1       YES manual up                    up
FastEthernet0/1.50     10.1.50.1       YES manual up                    up
FastEthernet0/1.60     10.1.60.1       YES manual up                    up
FastEthernet0/1.70     10.1.70.1       YES manual up                    up
R1#
Connection to device 192.168.100.2

R2#terminal length 0
R2#sh ip int br
Interface              IP-Address      OK? Method Status                Protocol
FastEthernet0/0        192.168.100.2   YES NVRAM  up                    up
FastEthernet0/1        unassigned      YES NVRAM  up                    up
FastEthernet0/1.10     10.2.10.1       YES manual up                    up
FastEthernet0/1.20     10.2.20.1       YES manual up                    up
FastEthernet0/1.30     10.2.30.1       YES manual up                    up
```

МОДУЛЬ PARAMIKO

МОДУЛЬ PARAMIKO

Paramiko это реализация протокола SSHv2 на Python. Paramiko предоставляет функциональность клиента и сервера.

Так как Paramiko не входит в стандартную библиотеку модулей Python, его нужно установить:

```
pip install paramiko
```

МОДУЛЬ PARAMIKO

Пример использования Paramiko (файл 3_paramiko.py):

```
import paramiko
import getpass
import sys
import time

COMMAND = sys.argv[1]
USER = input("Username: ")
PASSWORD = getpass.getpass()
ENABLE_PASS = getpass.getpass(prompt='Enter enable password: ')

DEVICES_IP = ['192.168.100.1','192.168.100.2','192.168.100.3']
```

МОДУЛЬ PARAMIKO

Пример использования Paramiko (файл 3_paramiko.py):

```
for IP in DEVICES_IP:
    print("Connection to device {}".format( IP ))
    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

    client.connect(hostname=IP, username=USER, password=PASSWORD,
                   look_for_keys=False, allow_agent=False)
    ssh = client.invoke_shell()

    ssh.send("enable\n")
    ssh.send(ENABLE_PASS + '\n')
    time.sleep(1)

    ssh.send("terminal length 0\n")
    time.sleep(1)
    print(ssh.recv(1000).decode('utf-8'))

    ssh.send(COMMAND + "\n")
    time.sleep(2)
    result = ssh.recv(5000).decode('utf-8')
    print(result)
```

МОДУЛЬ PARAMIKO

- client = paramiko.SSHClient() - этот класс представляет соединение к SSH-серверу. Он выполняет аутентификацию клиента.
- client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
  - set_missing_host_key_policy - устанавливает какую политику использовать, когда выполнятся подключение к серверу, ключ которого неизвестен.
  - paramiko.AutoAddPolicy() - политика, которая автоматически добавляет новое имя хоста и ключ в локальный объект HostKeys.

МОДУЛЬ PARAMIKO

- client.connect - метод, который выполняет подключение к SSH-серверу и аутентифицирует подключение
  - hostname - имя хоста или IP-адрес
  - username - имя пользователя
  - password - пароль
  - look_for_keys - по умолчанию paramiko выполняет аутентификацию по ключам. Чтобы отключить это, надо поставить поставив False
  - allow_agent - paramiko может подключаться к локальному SSH агенту ОС. Это нужно при работе с ключами, а так как, в данном случае, аутентификация выполняется по логину/паролю, это нужно отключить.

МОДУЛЬ PARAMIKO

- ssh = client.invoke_shell() - после выполнения предыдущей команды уже есть подключение к серверу. Метод invoke_shell позволяет установить интерактивную сессию SSH с сервером.

МОДУЛЬ PARAMIKO

- Внутри установленной сессии выполняются команды и получаются данные:
  - ssh.send - отправляет указанную строку в сессию
  - ssh.recv - получает данные из сессии. В скобках указывается максимальное значение в байтах, которое можно получить. Этот метод возвращает считанную строку
- Кроме этого, между отправкой команды и считыванием, кое-где стоит строка time.sleep
  - с помощью неё указывается пауза - сколько времени подождать, прежде чем скрипт продолжит выполняться. Это делается для того, чтобы дождаться выполнения команды на оборудовании

МОДУЛЬ PARAMIKO

Так выглядит результат выполнения скрипта:

```
$ python 3_paramiko.py "sh ip int br"
Username: cisco
Password:
Enter enable secret:
Connection to device 192.168.100.1

R1>enable
Password:
R1#terminal length 0


R1#
sh ip int br
Interface              IP-Address      OK? Method Status                Protocol
FastEthernet0/0        192.168.100.1   YES NVRAM  up                    up
FastEthernet0/1        unassigned      YES NVRAM  up                    up
FastEthernet0/1.10     10.1.10.1       YES manual up                    up
FastEthernet0/1.20     10.1.20.1       YES manual up                    up
FastEthernet0/1.30     10.1.30.1       YES manual up                    up
FastEthernet0/1.40     10.1.40.1       YES manual up                    up
FastEthernet0/1.50     10.1.50.1       YES manual up                    up
FastEthernet0/1.60     10.1.60.1       YES manual up                    up
FastEthernet0/1.70     10.1.70.1       YES manual up                    up
R1#
Connection to device 192.168.100.2

R2>enable
Password:
R2#terminal length 0
```

МОДУЛЬ PARAMIKO

Обратите внимание, что в вывод попал и процесс ввода пароля enable и команда terminal length.

Это связано с тем, что paramiko собирает весь вывод в буфер. И, при вызове метода recv (например, ssh.recv(1000)), paramiko возвращает всё, что есть в буфере. После выполнения recv, буфер пуст.

МОДУЛЬ PARAMIKO

Поэтому, если нужно получить только вывод команды sh ip int br, то надо оставить recv, но не делать print:

```
ssh.send("enable\n")
ssh.send(ENABLE_PASS + '\n')
time.sleep(1)

ssh.send("terminal length 0\n")
time.sleep(1)
#Тут мы вызываем recv, но не выводим содержимое буфера
ssh.recv(1000)

ssh.send(COMMAND + "\n")
time.sleep(3)
result = ssh.recv(5000).decode('utf-8')
print(result)
```

МОДУЛЬ NETMIKO

МОДУЛЬ NETMIKO

Netmiko это модуль, который позволяет упростить использование paramiko для сетевых устройств.

Грубо говоря, netmiko это такая "обертка" для paramiko.

Сначала netmiko нужно установить:

```
pip install netmiko
```

МОДУЛЬ NETMIKO

Пример использования netmiko (файл 4_netmiko.py):

```
from netmiko import ConnectHandler
import getpass
import sys


COMMAND = sys.argv[1]
USER = input("Username: ")
PASSWORD = getpass.getpass()
ENABLE_PASS = getpass.getpass(prompt='Enter enable password: ')

DEVICES_IP = ['192.168.100.1','192.168.100.2','192.168.100.3']
```

МОДУЛЬ NETMIKO

Пример использования netmiko (файл 4_netmiko.py):

```
for IP in DEVICES_IP:
    print("Connection to device {}".format( IP ))
    DEVICE_PARAMS = {'device_type': 'cisco_ios',
                     'ip': IP,
                     'username':USER,
                     'password':PASSWORD,
                     'secret':ENABLE_PASS }

    ssh = ConnectHandler(**DEVICE_PARAMS)
    ssh.enable()

    result = ssh.send_command(COMMAND)
    print(result)
```

МОДУЛЬ NETMIKO

- DEVICE_PARAMS - это словарь, в котором указываются параметры устройства
  - device_type - это предопределенные значения, которые понимает netmiko
    - в данном случае, так как подключение выполняется к устройству с Cisco IOS, используется значение 'cisco_ios'

МОДУЛЬ NETMIKO

- ssh = ConnectHandler(**DEVICE_PARAMS) - устанавливается соединение с устройством, на основе параметров, которые находятся в словаре
- ssh.enable() - переход в режим enable
  - пароль передается автоматически
  - используется значение ключа secret, который указан в словаре DEVICE_PARAMS
- result = ssh.send_command(COMMAND) - отправка команды и получение вывода

В этом примере не передается команда terminal length, так как netmiko по умолчанию, выполняет эту команду.

МОДУЛЬ NETMIKO

Так выглядит результат выполнения скрипта:

```
$ python 4_netmiko.py "sh ip int br"
Username: cisco
Password:
Enter enable password:
Connection to device 192.168.100.1
Interface              IP-Address      OK? Method Status                Protocol
FastEthernet0/0        192.168.100.1   YES NVRAM  up                    up
FastEthernet0/1        unassigned      YES NVRAM  up                    up
FastEthernet0/1.10     10.1.10.1       YES manual up                    up
FastEthernet0/1.20     10.1.20.1       YES manual up                    up
FastEthernet0/1.30     10.1.30.1       YES manual up                    up
FastEthernet0/1.40     10.1.40.1       YES manual up                    up
FastEthernet0/1.50     10.1.50.1       YES manual up                    up
FastEthernet0/1.60     10.1.60.1       YES manual up                    up
FastEthernet0/1.70     10.1.70.1       YES manual up                    up
Connection to device 192.168.100.2
Interface              IP-Address      OK? Method Status                Protocol
FastEthernet0/0        192.168.100.2   YES NVRAM  up                    up
FastEthernet0/1        unassigned      YES NVRAM  up                    up
FastEthernet0/1.10     10.2.10.1       YES manual up                    up
FastEthernet0/1.20     10.2.20.1       YES manual up                    up
FastEthernet0/1.30     10.2.30.1       YES manual up                    up
FastEthernet0/1.40     10.2.40.1       YES manual up                    up
FastEthernet0/1.50     10.2.50.1       YES manual up                    up
FastEthernet0/1.60     10.2.60.1       YES manual up                    up
FastEthernet0/1.70     10.2.70.1       YES manual up                    up
Connection to device 192.168.100.3
Interface              IP-Address      OK? Method Status                Protocol
FastEthernet0/0        192.168.100.3   YES NVRAM  up                    up
```

ПОДДЕРЖИВАЕМЫЕ ТИПЫ УСТРОЙСТВ

Netmiko поддерживает несколько типов устройств:

- Arista vEOS
- Cisco ASA
- Cisco IOS
- Cisco IOS-XR
- Cisco SG300
- HP Comware7
- HP ProCurve
- Juniper Junos
- Linux
- и другие

Актуальный список можно посмотреть в репозитории модуля.

СЛОВАРЬ, ОПРЕДЕЛЯЮЩИЙ ПАРАМЕТРЫ УСТРОЙСТВ

В словаре могут указываться такие параметры:

```
cisco_router = {'device_type': 'cisco_ios', # предопределенный тип устройства
                'ip': '192.168.1.1', # адрес устройства
                'username': 'user', # имя пользователя
                'password': 'userpass', # пароль пользователя
                'secret': 'enablepass', # пароль режима enable
                'port': 20022, # порт SSH, по умолчанию 22
                 }
```

ПОДКЛЮЧЕНИЕ ПО SSH

```
ssh = ConnectHandler(**cisco_router)
```

РЕЖИМ ENABLE

Перейти в режим enable:

```
ssh.enable()
```

Выйти из режима enable:

```
ssh.exit_enable_mode()
```

ОТПРАВКА КОМАНД

В netmiko есть несколько способов отправки команд:

- send_command - отправить одну команду
- send_config_set - отправить список команд
- send_config_from_file - отправить команды из файла (использует внутри метод send_config_set)
- send_command_timing - отправить команду и подождать вывод на основании таймера

SEND_COMMAND

Метод send_command позволяет отправить одну команду на устройство.

Например:

```
result = ssh.send_command("show ip int br")
```

SEND_COMMAND

Метод работает таким образом:

- отправляет команду на устройство и получает вывод до строки с приглашением или до указанной строки
  - приглашение определяется автоматически
  - если на вашем устройстве оно не определилось, можно просто указать строку, до которой считывать вывод
  - ранее так работал метод send_command_expect, но с версии 1.0.0 так работает send_command, а метод send_command_expect оставлен для совместимости
- метод возвращает вывод команды

SEND_COMMAND

- методу можно передавать такие параметры:
  - command_string - команда
  - expect_string - до какой строки считывать вывод
  - ```delay_factor - параметр позволяет увеличить задержку до начала поиска строки
  - max_loops - количество итераций, до того как метод выдаст ошибку (исключение). По умолчанию 500
  - strip_prompt - удалить приглашение из вывода. По умолчанию удаляется
  - strip_command - удалить саму команду из вывода

В большинстве случаев, достаточно будет указать только команду.

SEND_CONFIG_SET

Метод send_config_set позволяет отправить несколько команд конфигурационного режима.

Пример использования:

```
commands = ["router ospf 1",
            "network 10.0.0.0 0.255.255.255 area 0",
            "network 192.168.100.0 0.0.0.255 area 1"]

result = ssh.send_config_set(commands)
```

SEND_CONFIG_SET

Метод работает таким образом:

- заходит в конфигурационный режим,
- затем передает все команды
- и выходит из конфигурационного режима
  - в зависимости от типа устройства, выхода из конфигурационного режима может и не быть. Например, для IOS-XR выхода не будет, так как сначала надо закомитить изменения

SEND_CONFIG_FROM_FILE

Метод send_config_from_file отправляет команды из указанного файла в конфигурационный режим.

Пример использования:

```
result = ssh.send_config_from_file("config_ospf.txt")
```

Метод открывает файл, считывает команды и передает их методу send_config_set.

ДОПОЛНИТЕЛЬНЫЕ МЕТОДЫ

Кроме перечисленных методов для отправки команд, netmiko поддерживает такие методы:

- config_mode - перейти в режим конфигурации
  - ssh.config_mode()
- exit_config_mode - выйти из режима конфигурации
  - ssh.exit_config_mode()

ДОПОЛНИТЕЛЬНЫЕ МЕТОДЫ

- check_config_mode - проверить находится ли netmiko в режиме конфигурации (возвращает True, если в режиме конфигурации и False - если нет)
  - ssh.check_config_mode()
- find_prompt - возвращает текущее приглашение устройства
  - ssh.find_prompt()
- commit - выполнить commit на IOS-XR и Juniper
  - ssh.commit()
- disconnect - завершить соединение SSH

TELNET

С версии 1.0.0 netmiko поддерживает подключения по Telnet. Пока что, только для Cisco IOS устройств.

Внутри, netmiko использует telnetlib, для подключения по Telnet. Но, при этом, предоставляет тот же интерфейс дял работы, что и подключение по SSH.

TELNET

Для того, чтобы подключиться по Telnet, достаточно в словаре, который определяет параметры подключения, указать тип устройства 'cisco_ios_telnet':

```
DEVICE_PARAMS = {'device_type': 'cisco_ios_telnet',
                 'ip': IP,
                 'username':USER,
                 'password':PASSWORD,
                 'secret':ENABLE_PASS }
```

TELNET

В остальном, методы, которые применимы к SSH, применимы и к Telnet. Пример, аналогичный примеру с SSH (файл 4_netmiko_telnet.py):

```
from netmiko import ConnectHandler
import getpass
import sys
import time

COMMAND = sys.argv[1]
USER = input("Username: ")
PASSWORD = getpass.getpass()
ENABLE_PASS = getpass.getpass(prompt='Enter enable password: ')

DEVICES_IP = ['192.168.100.1','192.168.100.2','192.168.100.3']
```

TELNET

Файл 4_netmiko_telnet.py:

```
for IP in DEVICES_IP:
    print("Connection to device {}".format( IP ))
    DEVICE_PARAMS = {'device_type': 'cisco_ios_telnet',
                     'ip': IP,
                     'username':USER,
                     'password':PASSWORD,
                     'secret':ENABLE_PASS,
                     'verbose': True}
    ssh = ConnectHandler(**DEVICE_PARAMS)
    ssh.enable()

    result = ssh.send_command(COMMAND)
    print(result)
```

TELNET

Аналогично работают и методы:

- send_command_timing()
- find_prompt()
- send_config_set()
- send_config_from_file()
- check_enable_mode()
- disconnect()

ПАРАЛЛЕЛЬНЫЕ СЕССИИ

ПАРАЛЛЕЛЬНЫЕ СЕССИИ

Когда нужно опросить много устройств, выполнение подключений поочередно, будет достаточно долгим. Конечно, это будет быстрее, чем подключение вручную. Но, хотелось бы получать отклик как можно быстрее.

Для параллельного подключения к устройствам в Python есть два модуля:

- threading
- multiprocessing

ИЗМЕРЕНИЕ ВРЕМЕНИ ВЫПОЛНЕНИЯ СКРИПТА

Для оценки времени выполнения скрипта есть несколько вариантов. В курсе используются самые простые варианты:

- утилиту Linux time
- и модуль Python datetime

При оценке времени выполнения скрипта, в данном случае, не важна высокая точность. Главное, сравнить время выполнения скрипта в разных вариантах.

TIME

Утилита time в Linux позволяет замерить время выполнения скрипта. Например:

```
$ time python thread_paramiko.py
...
real    0m4.712s
user    0m0.336s
sys     0m0.064s
```

Для использования утилиты time достаточно написать time перед строкой запуска скрипта.

DATETIME

Второй вариант - модуль datetime. Этот модуль позволяет работать с временем и датами в Python.

Пример использования:

```
from datetime import datetime
import time

start_time = datetime.now()

#Тут выполняются действия
time.sleep(5)

print(datetime.now() - start_time)
```

DATETIME

Результат выполнения:

```
$ python test.py
0:00:05.004949
```

МОДУЛЬ THREADING

МОДУЛЬ THREADING

Модуль threading может быть полезен для таких задач:

- фоновое выполнение каких-то задач:
  - например, отправка почты во время ожидания ответа от пользователя
- параллельное выполнение задач связанных с вводом/выводом
  - ожидание ввода от пользователя
  - чтение/запись файлов
- задачи, где присутствуют паузы:
  - например, паузы с помощью sleep

МОДУЛЬ THREADING

Следует учитывать, что в ситуациях, когда требуется повышение производительности, засчет использования нескольких процессоров или ядер, нужно использовать модуль multiprocessing, а не модуль threading.

Рассмотрим пример использования модуля threading вместе с последним примером с netmiko.

Так как для работы с threading, удобнее использовать функции, код изменен:

- код подключения по SSH перенесен в функцию
- параметры устройств перенесены в отдельный файл в формате YAML

МОДУЛЬ THREADING

Файл netmiko_function.py:

```
from netmiko import ConnectHandler
import sys
import yaml

COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))

def connect_ssh(device_dict, command):

    print("Connection to device {}".format( device_dict['ip'] ))

    ssh = ConnectHandler(**device_dict)
    ssh.enable()

    result = ssh.send_command(command)
    print(result)

for router in devices['routers']:
    connect_ssh(router, COMMAND)
```

МОДУЛЬ THREADING

Файл devices.yaml с параметрами подключения к устройствам:

```
routers:
- device_type: cisco_ios
  ip: 192.168.100.1
  username: cisco
  password: cisco
  secret: cisco
- device_type: cisco_ios
  ip: 192.168.100.2
  username: cisco
  password: cisco
  secret: cisco
- device_type: cisco_ios
  ip: 192.168.100.3
  username: cisco
  password: cisco
  secret: cisco
```

МОДУЛЬ THREADING

Время выполнения скрипта (вывод скрипта удален):

```
$ time python netmiko_function.py "sh ip int br"
...
real    0m6.189s
user    0m0.336s
sys     0m0.080s
```

МОДУЛЬ THREADING

Пример использования модуля threading для подключения по SSH с помощью netmiko (файл netmiko_threading.py):

```
from netmiko import ConnectHandler
import sys
import yaml
import threading

COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))

def connect_ssh(device_dict, command):
    ssh = ConnectHandler(**device_dict)
    ssh.enable()
    result = ssh.send_command(command)

    print("Connection to device {}".format( device_dict['ip'] ))
    print(result)
```

МОДУЛЬ THREADING

Файл netmiko_threading.py:

```
def conn_threads(function, devices, command):
    threads = []
    for device in devices:
        th = threading.Thread(target = function, args = (device, command))
        th.start()
        threads.append(th)

    for th in threads:
        th.join()

conn_threads(connect_ssh, devices['routers'], COMMAND)
```

МОДУЛЬ THREADING

Время выполнения кода:

```
$ time python netmiko_function_threading.py "sh ip int br"

...
real    0m2.229s
user    0m0.408s
sys     0m0.068s
```

Время почти в три раза меньше. Но, надо учесть, что такая ситуация не будет повторяться при большом количестве подключений.

МОДУЛЬ THREADING

- threading.Thread - класс, который создает поток. Ему передается функция, которую надо выполнить, и её аргументы
- th.start() - запуск потока
- threads.append(th) - поток добавляется в список
- th.join() - метод ожидает завершения работы потока. Метод join выполняется для каждого потока в списке. Таким образом основная программа завершится только когда завершат работу все потоки
  - по умолчанию, join ждет завершения работы потока бесконечно. Но, можно ограничить время ожидания передав join время в секундах. В таком случае, join завершится после указанного количества секунд.

ПОЛУЧЕНИЕ ДАННЫХ ИЗ ПОТОКОВ

В предыдущем примере, данные выводились на стандартный поток вывода. Для полноценной работы с потоками, необходимо также научиться получать данные из потоков. Чаще всего, для этого используется очередь.

Очередь - это структура данных, которая используется и в работе с сетевым оборудованием. Объект queue.Queue() - это FIFO очередь.

ПОЛУЧЕНИЕ ДАННЫХ ИЗ ПОТОКОВ

В Python есть модуль queue, который позволяет создавать разные типы очередей.

Очередь передается как аргумент в функцию connect_ssh, которая подключается к устройству по SSH. Результат выполнения команды добавляется в очередь.

ПОЛУЧЕНИЕ ДАННЫХ ИЗ ПОТОКОВ

Пример использования потоков с получением данных (файл netmiko_threading_data.py):

```
# -*- coding: utf-8 -*-
from netmiko import ConnectHandler
import sys
import yaml
import threading
from queue import Queue

COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))

def connect_ssh(device_dict, command, queue):
    ssh = ConnectHandler(**device_dict)
    ssh.enable()
    result = ssh.send_command(command)
    print("Connection to device {}".format( device_dict['ip'] ))

    #Добавляем словарь в очередь
    queue.put({ device_dict['ip']: result })
```

ПОЛУЧЕНИЕ ДАННЫХ ИЗ ПОТОКОВ

Файл netmiko_threading_data.py:

```
def conn_threads(function, devices, command):
    threads = []
    q = Queue()

    for device in devices:
        # Передаем очередь как аргумент, функции
        th = threading.Thread(target = function, args = (device, command, q))
        th.start()
        threads.append(th)

    for th in threads:
        th.join()

    results = []
    # Берем результаты из очереди и добавляем их в список results
    for t in threads:
        results.append(q.get())

    return results

print(conn_threads(connect_ssh, devices['routers'], COMMAND))
```

ПОЛУЧЕНИЕ ДАННЫХ ИЗ ПОТОКОВ

Обратите внимание, что в функции connect_ssh добавился аргумент queue.

Очередь вполне можно воспринимать как список:

- метод queue.put() равнозначен list.append()
- метод queue.get() равнозначен list.pop(0)

Для работы с потоками и модулем threading, лучше использовать очередь. Но, конкретно в данном примере, можно было бы использовать и список.

ПОЛУЧЕНИЕ ДАННЫХ ИЗ ПОТОКОВ

Пример со списком, скорее всего, будет проще понять. Поэтому ниже аналогичный код, но с использованием обычного списка, вместо очереди (файл netmiko_threading_data_list.py):

```
# -*- coding: utf-8 -*-
from netmiko import ConnectHandler
import sys
import yaml
import threading

COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))

def connect_ssh(device_dict, command, queue):
    ssh = ConnectHandler(**device_dict)
    ssh.enable()
    result = ssh.send_command(command)
    print("Connection to device {}".format( device_dict['ip'] ))

    #Добавляем словарь в список
    queue.append({ device_dict['ip']: result })
```

ПОЛУЧЕНИЕ ДАННЫХ ИЗ ПОТОКОВ

Файл netmiko_threading_data_list.py:

```
def conn_threads(function, devices, command):
    threads = []
    q = []

    for device in devices:
        # Передаем список как аргумент, функции
        th = threading.Thread(target = function, args = (device, command, q))
        th.start()
        threads.append(th)

    for th in threads:
        th.join()

    # Эта часть нам не нужна, так как, при использовании списка,
    # мы просто можем вернуть его
    #results = []
    #for t in threads:
    #    results.append(q.get())

    return q

print(conn_threads(connect_ssh, devices['routers'], COMMAND))
```

МОДУЛЬ MULTIPROCESSING

МОДУЛЬ MULTIPROCESSING

Модуль multiprocessing использует интерфейс подобный модулю threading. Поэтому перенести код с использования потоков на использование процессов, обычно, достаточно легко.

Каждому процессу выделяются свои ресурсы. Кроме того, у каждого процесса свой GIL, а значит, нет тех проблем, которые были с потоками и код может выполняться параллельно и задействовать ядра/процессоры компьютера.

МОДУЛЬ MULTIPROCESSING

Пример использования модуля multiprocessing (файл netmiko_multiprocessing.py):

```
import multiprocessing
from netmiko import ConnectHandler
import sys
import yaml


COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))

def connect_ssh(device_dict, command, queue):
    ssh = ConnectHandler(**device_dict)
    ssh.enable()
    result = ssh.send_command(command)

    print("Connection to device {}".format( device_dict['ip'] ))
    queue.put({device_dict['ip']: result})
```

МОДУЛЬ MULTIPROCESSING

Файл netmiko_multiprocessing.py:

```
def conn_processes(function, devices, command):
    processes = []
    queue = multiprocessing.Queue()

    for device in devices:
        p = multiprocessing.Process(target = function, args = (device, command, queue))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()

    results = []
    for p in processes:
        results.append(queue.get())

    return results

print(( conn_processes(connect_ssh, devices['routers'], COMMAND) ))
```

МОДУЛЬ MULTIPROCESSING

Обратите внимание, что этот пример аналогичен последнему примеру, который использовался с модулем threading. Единственное отличие в том, что в модуле multiprocessing есть своя реализация очереди, поэтому нет необходимости использовать модуль queue.

МОДУЛЬ MULTIPROCESSING

Если проверить время исполнения этого скрипта, аналогичного для модуля threading и последовательного подключения, то получаем такую картину:

```
последовательное: 5.833s
threading:        2.225s
multiprocessing:  2.365s
```

МОДУЛЬ MULTIPROCESSING

Время выполнения для модуля multiprocessing немного больше. Но это связано с тем, что на создание процессов уходит больше времени, чем на создание потоков. Если бы скрипт был сложнее и выполнялось больше задач, или было бы больше подключений, тогда бы multiprocessing начал бы существенно выигрывать у модуля threading.