Netmiko 4 新功能使用手册

> 📌 原文链接：https://zhuanlan.zhihu.com/...
> 🕘 收藏时间：2025 年 05 月 11 日

今年（2022 年）3 月 23 日，在 Netmiko 3 的最后一版 Netmiko 3.4.0 发布整整 11 个月后，“Netmiko 之父”Kirk Byers 正式公布并发行了 Netmiko 4 的初版即 Netmiko 4.0.0，和很多模块的重大更新类似，Netmiko 4.0.0 发布后存在众多 bug，尤其是华为、H3C 等很多国产设备受影响十分严重，很多用户通过 pip install 安装 Netmiko 的时候不得不手动指定版本，用回稳定的 3.4.0。不过 Kirk 行动迅速，在 4 月 27 号，6 月 29 号以及上星期 8 月 9 号相继发布了 4.1.0、4.1.1 和 4.1.2 版本，不仅优化了 Netmiko 4 的性能，解决了 4.0.0 众多的 bug，也提供了对更多设备的支持。

1. read_timeout 和 read_timeout_override

Netmiko 的作者 Kirk Byers 曾坦诚，Netmiko 4 诞生之前的所有版本中最困扰他的就是截屏（screen-scraping）问题。所谓截屏问题是指在用户向设备输入一条 show 或者 display 命令后，Netmiko 无法判断回显内容是否已经完整返回，其实不仅是 Netmiko，任何需要用到截屏功能的模块都会遇到类似的问题，比如 Paramiko 需要我们用 time.sleep()手动指定休眠时间，完全把问题抛回给用户自己解决。

Netmiko 是 Paramiko 的衍生版本，它帮我们省去了手动指定休眠时间的这一步骤。在 Netmiko4 之前，Netmiko 会用 delay_factor 参数（在 send_command()中使用，只对在 send_command()里使用的命令有效）或 global_delay_factor 参数（在设定登录设备参数的字典中使用，对在该设备上输入的所有命令都有效）配合 expect_string 参数来处理这个问题。比如说通过 send_command()在思科交换机里输入 show ip int brief 命令，我们可以通过 expect_string=r'#'告诉 Netmiko 去回显内容中抓取最后一个字符，即思科设备的命令提示符#（如果没有指定 expect_string 并且 device_type 为 cisco_ios，则 expect_string 的默认值为#），如下图所示：

Python

```python
device = {
    "device_type":"hp_comware",
    "host":"11.0.0.2",
    "username":"netops",
    "password":"Admin@1234"
    "fast_cli":True; #device_type是cisco_ios时，fast_cli=True，其他设备类型时，fast_cli=False
    "global_delay_factor":0.1;#延时因子，默认为1,fast_cli=True时，global_delay_factor=0.1
    }
```

Python

```python
with ConnectHandler(**device) as net_connect:
    output = net_connect.send_command("show ip interface brief",
                                      expect_string=r"#", #device_type是cisco_ios时，expect_string=r"#"
                                      delay_factor=0.1) #默认值为1，fast_cli=True时，delay_factor=0.1
    print(output)
```

在向设备输入一条命令后，Netmiko 默认**最多等待 100 秒**来从回显内容中抓取 expect_string 指定的字符，如果 fast_cli=True，则 Netmio **最多只等待 10 秒钟，这 10 秒怎么来的？也就是 100（默认等待时间）乘以 0.1（fast_cli=True 时，delay_factor 和 global_delay_factor 的值）得来的。**如果抓取到了 expect_string 指定的回显字符，则 Netmiko 会立即返回回显内容（**注意是立即返回，不是非要等到第 10 秒或第 100 秒才返回**），如果过了 10 秒或 100 秒都还没抓取到#，则 Netmiko 会返回一个异常。

一般来说 `show ip interface brief` 的回显内容肯定是能在 10 秒钟内顺利返回的，但是 `show run` 则不一定，`show tech-support` 就更不可能了，这个时候我们必须手动修改 fast_cli，delay_factor 或者 global_delay_factor 几个参数来调整 Netmiko 等待回显内容的时间。

为了简化这个步骤，Netmiko 4 中特意在 `send_command()` 中引入了 `read_timeout` 参数，`read_timeout` 参数可以让我们直接指定 Netmiko 最多等待多少秒来从回显内容中抓取 expect_string 指定的字符，比如说在生产网络里一台思科的 6800 三层交换机上输入 show tech-support 后要等 45-50 秒以上才能返回完整的回显内容，则我们可以把 read_timeout 参数设为 60，如下图所示：

```python
with ConnectHandler(**device) as net_connect:
    output = net_connect.send_command("show tech-support",
                                      read_timeout=60) # 让netmiko最多等待60秒。
```

可以看到read_timeout最大的好处是大大节省了新手的学习成本（**read_timeout默认值为10秒**），不用再去学习什么fast_cli, delay_factor和global_delay_factor这些参数，并且也免去了额外做乘法运算的麻烦。和global_delay_factor类似，我们也可以在设定登录设备参数的字典中使用`read_timeout_override`来**全局修改**read_timeout的值，如下图所示：

```python
device02 = {
    "device_type":"cisco_ios",
    "host":"11.0.0.2",
    "username":"netops",
    "password":"Admin@1234",
    "read_timeout_override":90; #全局有效
}
```

另外，出于兼容性考虑，还可以在字典里将delay_factor_compact设为True，这样Netmiko 4会按照Netmiko 3的模式继续使用fast_cli, delay_factor和global_delay_factor来计算等待时间，如下图所示。

```python
device02 = {
    "device_type":"cisco_ios",
    "host":"11.0.0.2",
    "username":"netops",
    "password":"Admin@1234",
    "read_timeout_override":90, #全局有效
    "delay_factor_compact":True # 默认为False，不兼容Netmiko之前的版本
}
```

2.send_multiline()和send_multiline_timing()

在《网络工程师的 Python 之路 -- Netmiko 终极指南》实验 8 里曾经提到过如何处理设备交互命令的场景，比如在思科交换机上输入 del flash0:/test.txt 这个删除 flash: 下文件的命令后，系统会询问你是否 confirm，如下图所示。

Shell

```
1 Switch#del flash0:/test.txt
2 Delete filename [test.txt]?
3 Delete flash0:/test.txt? [confirm]n
4 Delete of flash0:/test.txt aborted!
5 Switch#
```

或者使用 extended ping 模式后，系统让你输入一系列的和 ping 相关的参数，如下图所示。

Shell

```
 1 Switch#ping
 2 Protocol [ip]:
 3 Target IP address: 192.168.12.2
 4 Repeat count [5]:
 5 Datagram size [100]:
 6 Timeout in seconds [2]:
 7 Extended commands [n]:
 8 Sweep range of sizes [n]:
 9 Type escape sequence to abort.
10 Sending 5, 100-byte ICMP Echos to 192.168.12.2, timeout is 2 seconds:
11 .....
12 Success rate is 0 percent (0/5)
13 Switch#
```

在《网络工程师的 Python 之路 -- Netmiko 终极指南》实验 8 中我们是通过 Netmiko 3 中的 send_command()配合 expect_string 来应对这个问题。但是这种做法非常复杂且要写大量代码完成，比如说应对第一个 del flash0:/test.txt 的场景时，因为该交互场景只需要我们输入一个参数（是否 confirm），所以代码相对还比较简洁，如下图所示。

Shell

```
with ConnectHandler(**Switch01) as connect:
    print("已经成功登录交换机"+Switch01['host'])

    output = connect.send_command(command_string="delete flash0:/text.t
xt",
                  expect_string=r"Delete flash:/text.txt?",
                  strip_prompt=False,
                  strip_command=False)
    output += connect.send_command(command_string="y",
                  expect_string=r"#",
                  strip_prompt=False,
                  strip_command=False)
print(output)
```

但是遇到 extended ping 模式这种需要用户输入多个参数的交互场景时，代码量就非常恐怖了，如下图所示。

Python

```
with ConnectHandler(**device) as net_connect:
    cmd = "ping"
    target_ip = "8.8.8.8"
    count = "30"
    output = net_connect.send_command_timing(cmd, strip_prompt=False, s
trip_command=False)
    output += net_connect.send_command_timing("\n", strip_prompt=Fals
e, strip_command=False)
    output += net_connect.send_command_timing(target_ip, strip_prompt=F
alse, strip_command=False)
    output += net_connect.send_command_timing(count, strip_prompt=Fals
e, strip_command=False)
    output += net_connect.send_command_timing("\n", strip_prompt=Fals
e, strip_command=False)
    output += net_connect.send_command_timing("\n", strip_prompt=Fals
e, strip_command=False)
    output += net_connect.send_command_timing("\n", strip_prompt=Fals
e, strip_command=False)
print(output)
```

究其原因就是send_command()函数是基于内容的（pattern-based），它必须要等到用户告诉它等到什么回显内容后才会执行后面的代码。同样应对 extended ping 的交互命令场景时，如果我们用基于时间（time-based）的 send_command_timing()函数来处理的话，代码量会相对小很多，如下图所示。

Python

```python
with ConnectHandler(**device) as conn:
    data = ""
    commands = [
        "ping",
        "\n",
        "8.8.8.8",
        "\n",
        "\n",
        "\n",
        "\n",
        "\n",
        "\n"
    ]
    for cmd in commands:
        data += conn.send_command_timing(
            cmd,
            strip_command=False,
            strip_prompt=False
        )
    print(data)
```

而在 Netmiko 4 中加入的 `send_multiline()` 和 `send_multiline_timing()` 则将类似的需求变得更简单，其中前者为 pattern-based，后者为 time-based。

首先来看怎么用 send_multiline()应对第一个 del flash0:/test.txt 的场景，代码如下图所示。

Python

```python
with ConnectHandler(**SW1) as conn:
    cmd_list = [
        ["del flash0:/test.txt",r"Delete flash0:/test.txt?"],
        ["n",r"confirm"]
    ]
output = conn.send_multiline(cmd_list)
print(output)
```

可以看到，我们在 cmd_list 这个列表里额外添加了两组子列表，每组子列表的元素为我们输入的命令，以及执行该命令后我们想要 Netmiko 在回显内容中抓取到的字符（类似 send_command()的 expect_string）。比如说第一组子列表里，我们输入命令 del flash0:/test.txt，希望抓取到的回显内容为"Delete flash0:/test.txt?"]，第二组子列表里我们输入命令 n，希望抓取到的回显内容改为 confirm，以此类推。

如果用 time-based 的 send_multiline_timing()来做的话，上述代码还能更简洁，如下图所示。

Python

```python
with ConnectHandler(**SW1) as conn:
    cmd_list = [
        "del flash0:/test.txt",
        "n"
    ]
output = conn.send_multiline_timing(cmd_list)
print(output)
```

而在 extended ping 场景中，如果用 send_multiline_timing()来做的话，代码如下图所示。

Python

```python
with ConnectHandler(**device) as net_connect:
    target_ip = "8.8.8.8"
    count = "30"
    cmd_list = [
        "ping",
        "\n",
        target_ip,
        count,
        "\n",
        "\n",
        "\n",
        "\n",
    ]
output = net_connect.send_multiline_timing(cmd_list)
print(output)
```

很显然在处理多交互命令的场景时，在 Netmiko 4 中加入的 send_multline()和 send_multiline_timing()将 Netmiko 3 时代的 send_command()和 send_command_timing()的代码大大简化了。另外我们也注意到 send_multiline_timing()的

代码比 send_multiline()更简单易懂，不过相较于 send_multiline()，使用 send_multiline_timing()的话有一个劣势，那就是每输入一条命令后，Netmiko 会默认固定等待 2 秒钟才会执行下一条命令（因为 send_multiline_timing()是 time-based 的），**而 pattern-based 的 send_multiline()则会在读取到指定的回显内容后立即执行后面的代码。鱼和熊掌不可兼得，一个代码简单脚本但运行速度慢，一个代码稍微复杂但脚本运行速度快**，如何取舍完全看用户自己的决定。

3.ConnLogOnly

使用 Netmiko 3 或之前的版本时，用户需要写很多 try/except 异常处理来应对各种各样会导致脚本停止工作的错误或异常，比如最常见的因为 **SSH 用户名 / 密码验证不通过导致的 Netmiko Authentication Exception** 和**设备链接超时无响应导致的 Netmiko Timeout Exception**，类似这样的异常处理在设备数量众多大型网络里基本是标配（设备数量越多，发生问题的概率越大），如下图所示。

Python

```
try:
    conn = ConnectHandler(**device)
except NetmikoAuthenticationException:
    return
except NetmikoTimeoutException:
    return
```

在 Netmiko 4 中，我们可以用 ConnLogOnly 替代 ConnectHandler 来统一处理这个问题，代码如下图所示。

Python

```
from netmiko import ConnLogOnly

conn = ConnLogOnly(**device)
if conn is None:
    print("登陆设备失败！")
```

使用 ConnLogOnly 时

- 如果其返回值为 None，则 Netmiko 会直接判定登陆设备失败

- 如果登陆成功，则和 ConnectHandler 一样返回一个 Netmiko 连接对象（Netmiko Connection Object）。

如果要查看具体登陆失败的原因的话，可以在运行脚本后 Netmiko 生成的 netmiko.log 文件中查看，netmiko.log 也是 Netmiko 4 新引入，Netmiko 3 之前没有的功能，netmiko.log 文件和脚本文件在同一文件夹下，如下图所示。