第 1 章

Python 的安装和使用

工欲善其事，必先利其器，鉴于很多网络工程师读者都是第一次接触 Python，本书开篇将配图详细介绍 Python 在不同操作系统下的安装和使用方法。Python 在 Windows、Linux 及 MacOS 上都可以使用，目前最新的 MacOS 本身已经内置了 Python，打开命令行终端输入命令 python 即可使用。本章主要介绍 Python 在 Windows 和 Linux （CentOS） 上的安装和使用方法。

Python 的运行模式大致分为两种：一种是使用解释器（Interpreter）的交互模式（Interactive Mode），另一种是运行脚本的脚本模式（Script Mode）。使用解释器和脚本来运行 Python 最大的区别是前者能在你执行一行或一段代码后提供“即时反馈”，让你看到是否得到了想要的结果，或者告诉你代码是否有误，而后者则是将整段代码写入一个扩展名为.py 的文本文件中“打包执行”。脚本模式在实际的网络运维工作中很常见，但是从学习的角度来讲，肯定是能提供“即时反馈”的解释器更利于初学者，因此本章大部分内容将基于解释器的交互模式来讲解，当然，也有某些代码案例必须用脚本模式来进行演示。

1.1 安装 Python

本书所有内容以 Windows 10.0（64 位）和 CentOS 8 分别作为 Windows 和 Linux 两大操作系统的演示平台。

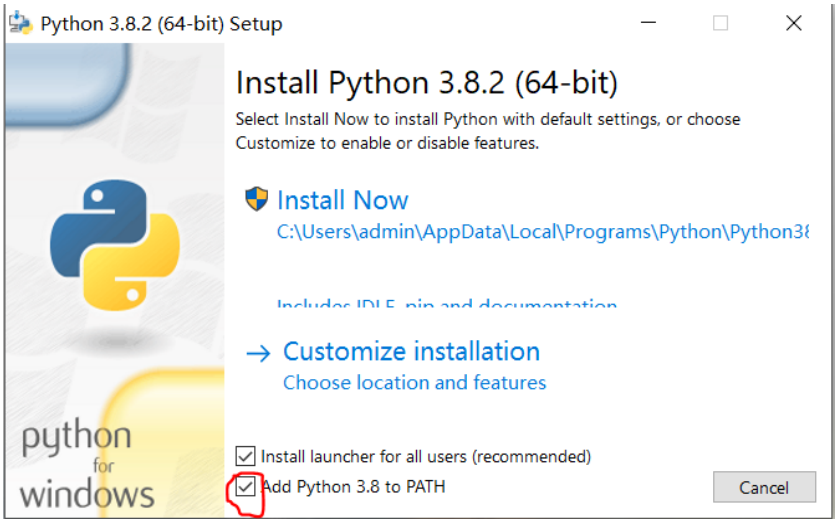
1.1.1 在 Windows 上安装 Python 3.8.2

首先在 Python 官网下载 Windows 版的 Python 3（注意，从 Python 3.5 开始，Python 3 已经不再支持 Windows XP 及更早的 Windows 版本）。截至 2020 年 4 月，最新的版本为 3.8.2，读者可根据自身情况选择 32 位和 64 位版本，安装文件有.zip、.exe 和.web-based 3 种格式可选，这里推荐选择.exe 格式，如下图所示。

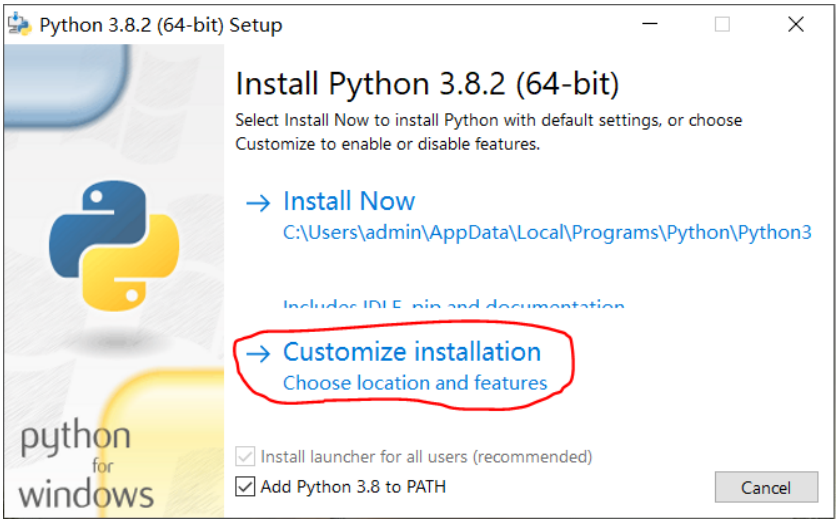
Files

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		f9f3768f757e34b342dbc06b41cbc844	24007411	SIG
XZ compressed source tarball	Source release		e9d6ebc92183a177b8e8a58cad5b8d67	17869888	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	f12203128b5c639dc08e5a43a2812cc7	30023420	SIG
Windows help file	Windows		7506675dcbb9a1569b54e600ae66c9fb	8507261	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	1a98565285491c0ea65450e78afe6f8d	8017771	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	b5df1cbb2bc152cd70c3da9151cb510b	27586384	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	2586cdad1a363d1a8abb5fc102b2d418	1363760	SIG
Windows x86 embeddable zip file	Windows		1b1f0f0c5ee8601f160cfad5b560e3a7	7147713	SIG
Windows x86 executable installer	Windows		6f0ba59c7dbeba7bb0ee21682fe39748	26481424	SIG
Windows x86 web-based installer	Windows		04d97979534f4bd33752c183fc4ce680	1325416	SIG

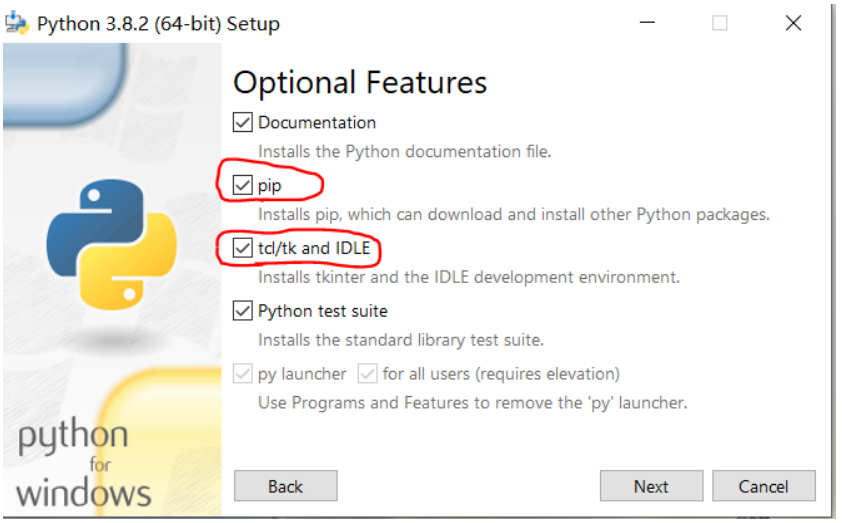
安装过程中有一个很重要的步骤，如下图中的“Add Python 3.8 to PATH”，这里默认是没有勾选的，请务必勾选，它会自动帮你设置好环境变量，也就是说将来在你打开命令行运行 Python 脚本时，你可以在任意盘符和文件夹下直接输入命令 python xxx.py 来运行脚本，而无须输入 Python 执行程序所在的完整路径来运行脚本，例如 C:\Python38\python xxx.py，不要小看这一选项提供的自动环境变量设置，它能帮助 Python 初学者节省很多很多时间！



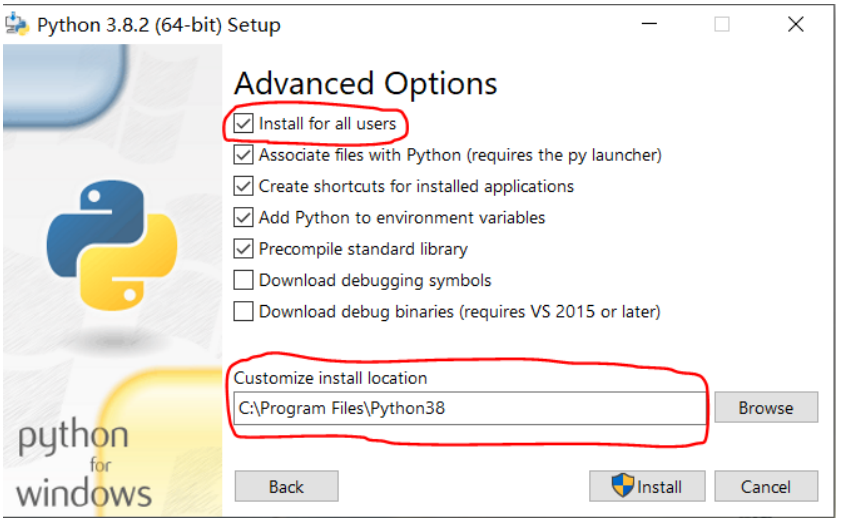
之后选择“Customize installation”进入自定义安装，如下图所示。



在 Optional Features 的选项中确保“pip”和“tcl/tk and IDLE”都被勾选，关于它们的作用后面会提到，其他选项使用默认配置即可，然后点击“Next”，如下图所示。



如下图所示，在 Advanced Options 中，推荐将“Install for all users”勾选，它会将 Python 的安装路径从 C:\Users\admin\AppData\Local\Programs\Python\Python38 换成 C:\Program Files\Python38，方便将来查找和访问。当然读者也可以自定义安装路径，以及根据自身情况决定是否给所有用户都安装 Python 3。



在安装好 Python 3 后，打开命令行输入 py 或者 python，如果可以进入 Python 3.8.2 的解释器，则说明 Python 3 安装成功，如下图所示。

```
命令提示符 - py

C:\Users\admin>
C:\Users\admin>py
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 23:03:10) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

注：如果之前你已经安装过 Python 2，则输入命令 python 会进入 Python 2，两个版本之间的使用互不影响。如果只安装了 Python 3，则只能通过命令 py 来进入 Python 3，如下所示。

```
命令提示符 - python

C:\Users\admin>
C:\Users\admin>py
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 23:03:10) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()

C:\Users\admin>python
Python 2.7.16 (v2.7.16:413a49145e, Mar  4 2019, 01:37:19) [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

1.1.2　在 Linux 上安装 Python 3.8.2

本书将使用 CentOS 8 作为 Linux 版本的演示平台（在 Windows 上运行的 VMware 虚拟机），这里只介绍在 CentOS 命令行终端里使用 Python 的方法，在 GNOME 桌面环境下使用 Python 的方法不在本书的讨论范围内。因为是实验环境，所以直接使用 Root 用户，

免去了 sudo 命令，读者请根据自身情况决定是否使用 sudo。

和 MacOS 一样，最新的 CentOS 8 已经内置了 Python 2 和 Python 3，输入 python2 和 python3 两个命令可以分别进入 Python 2 和 Python 3，如下所示。

```
[root@CentOS-Python ~]# python2
Python 2.7.16 (default, Nov 17 2019, 00:07:27)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
[root@CentOS-Python ~]#
[root@CentOS-Python ~]# python3
Python 3.6.8 (default, Nov 21 2019, 19:31:34)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
[root@CentOS-Python ~]#
```

CentOS 8 内置的 Python 3 的版本为 3.6.8，我们需要将它升级到 3.8.2，方法如下。

首先通过下列命令来下载 Python 3.8.2 的安装包。

```
wget https://www.python.org/ftp/python/3.8.2/Python-3.8.2.tgz
```

运行该命令后的截图如下。

```
[root@CentOS-Python ~]# wget https://www.python.org/ftp/python/3.8.2/Python-3.8.2.tgz
--2020-04-27 22:38:45--  https://www.python.org/ftp/python/3.8.2/Python-3.8.2.tgz
Resolving www.python.org (www.python.org)... 151.101.228.223, 2a04:4e42:1a::223
Connecting to www.python.org (www.python.org)|151.101.228.223|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 24007411 (23M) [application/octet-stream]
Saving to: ‘Python-3.8.2.tgz’

Python-3.8.2.tgz                                    100%[=============================================

2020-04-27 22:46:44 (49.0 KB/s) - ‘Python-3.8.2.tgz’ saved [24007411/24007411]
```

然后输入下列命令来下载安装 Python 3.8 所需要的环境依赖包。

```
yum install gcc openssl-devel bzip2-devel libffi-devel
```

运行命令后的截图如下。

```
[root@CentOS-Python ~]# yum install gcc openssl-devel bzip2-devel libffi-devel
Last metadata expiration check: 0:55:06 ago on Mon 27 Apr 2020 10:25:05 PM EDT.
Package gcc-8.3.1-4.5.el8.x86_64 is already installed.
Package openssl-devel-1:1.1.1c-2.el8_1.1.x86_64 is already installed.
Package bzip2-devel-1.0.6-26.el8.x86_64 is already installed.
Package libffi-devel-3.1-21.el8.x86_64 is already installed.
Dependencies resolved.
Nothing to do.
Complete!
[root@CentOS-Python ~]#
```

接下来用 tar 命令对刚才下载的 Python-3.8.2.tgz 包解压缩，解压缩完成后，当前盘符下会多出一个 Python 3.8.2 的文件夹，用 cd 命令进入该文件夹，如下图所示。

```
[root@CentOS-Python ~]# tar zxf Python-3.8.2.tgz
[root@CentOS-Python ~]# ls
anaconda-ks.cfg  initial-setup-ks.cfg  Python-3.8.2  Python-3.8.2.tgz
[root@CentOS-Python ~]# cd Python-3.8.2/
[root@CentOS-Python Python-3.8.2]#
```

接着依次输入下列命令来完成 Python 3.8.2 的安装。

```
./configure --enable-optimizations
make altinstall
```

运行命令后的截图如下。

```
[root@CentOS-Python Python-3.8.2]# ./configure --enable-optimizations
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
checking for python3.8... python3.8
checking for --enable-universalsdk... no
checking for --with-universal-archs... no
checking MACHDEP... "linux"
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
```

```
[root@CentOS-Python Python-3.8.2]# make altinstall
gcc -pthread -c -Wno-unused-result -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall    -std=c99 -Wextra
 -I./Include/internal  -I. -I./Include    -DPy_BUILD_CORE -o Programs/python.o ./Programs/python.c
gcc -pthread -c -Wno-unused-result -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall    -std=c99 -Wextra
 -I./Include/internal  -I. -I./Include    -DPy_BUILD_CORE -o Parser/acceler.o Parser/acceler.c
gcc -pthread -c -Wno-unused-result -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall    -std=c99 -Wextra
 -I./Include/internal  -I. -I./Include    -DPy_BUILD_CORE -o Parser/grammar1.o Parser/grammar1.c
gcc -pthread -c -Wno-unused-result -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall    -std=c99 -Wextra
 -I./Include/internal  -I. -I./Include    -DPy_BUILD_CORE -o Parser/listnode.o Parser/listnode.c
gcc -pthread -c -Wno-unused-result -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall    -std=c99 -Wextra
 -I./Include/internal  -I. -I./Include    -DPy_BUILD_CORE -o Parser/node.o Parser/node.c
gcc -pthread -c -Wno-unused-result -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall    -std=c99 -Wextra
 -I./Include/internal  -I. -I./Include    -DPy_BUILD_CORE -o Parser/parser.o Parser/parser.c
gcc -pthread -c -Wno-unused-result -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall    -std=c99 -Wextra
 -I./Include/internal  -I. -I./Include    -DPy_BUILD_CORE -o Parser/token.o Parser/token.c
gcc -pthread -c -Wno-unused-result -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall    -std=c99 -Wextra
```

安装完毕后，输入命令 python3.8，如果可以进入 Python 的解释器，则说明 Python 3.8.2 安装成功，如下所示。

```
[root@CentOS-Python ~]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
[root@CentOS-Python ~]#
```

注：安装 Python 3.8.2 并不会覆盖 CentOS 内置的 Python 3.6.8，使用命令 python3 仍然可以进入 3.6.8 版本，必须使用 Python3.8 才能进入 3.8.2 版本，如下所示。

```
[root@CentOS-Python ~]# python3
Python 3.6.8 (default, Nov 21 2019, 19:31:34)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
[root@CentOS-Python ~]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
[root@CentOS-Python ~]#
```

1.2 在 Windows 上使用 Python 3.8.2

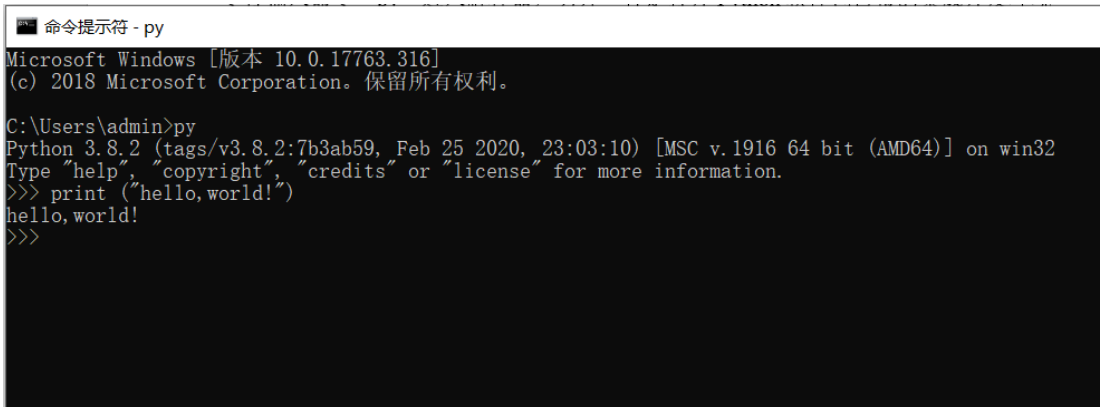
前面提到 Python 运行模式分为使用解释器的交互模式和运行脚本的脚本模式，下面分别举例介绍这两种运行模式在 Windows 中的使用方法。

1.2.1　交互模式

在 Windows 下，有两种方法进入 Python 解释器来使用交互模式，一种是通过命令行输入命令 py 或者 python 进入解释器，另一种是打开 Python 软件包自带的集成开发环境（IDE），也就是 IDLE。两种方法进入的解释器的界面稍有不同，但是功能完全一样。

1. 使用命令行进入 Python 解释器

首先来看第一种方法，打开 Windows 的命令行（CMD），输入命令 py 或者 python 即可进入 Python 解释器，如下图所示。



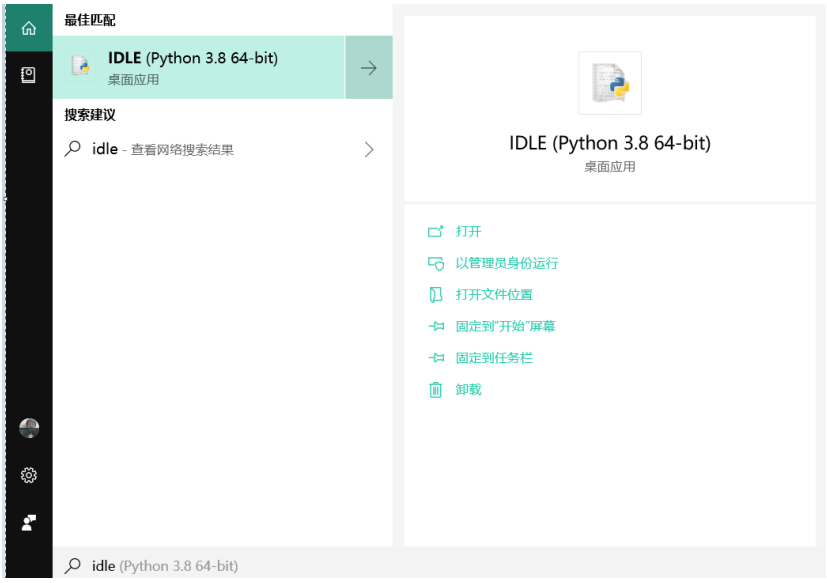
我们在 Python 解释器中输入第一段代码 print ("hello, world!")，解释器随即打印出“hello，world!”的内容。这种“即时反馈”的特性是交互模式下特有的，脚本模式下不具备。

注：在 Python 2 中，print ("hello，world! ")也可以省去括号写成 print "hello, world! "，但是在 Python 3 中，print 后面的内容必须加上括号，否则 Python 会报错提醒你加上括号，如下图所示。

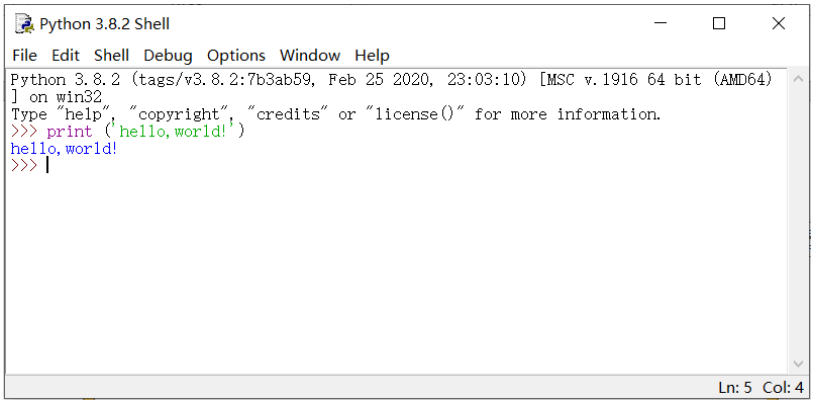
```
命令提示符 - python

C:\Users\admin>python
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 23:03:10) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print "hello, world!"
  File "<stdin>", line 1
    print "hello, world!"
          ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("hello, world!")?
>>>
```

2. 使用 IDLE 进入 Python 解释器

然后介绍使用 IDLE 进入解释器的方法。以 Windows 10 为例，单击左下角的“开始”后搜索“idle”即可找到 IDLE（Python 3.8 64-bit)）这个桌面应用程序，如下图所示。



将 IDLE 打开后会弹出下图的窗口。再次输入代码 print ('hello， world!')，可以看到解释器同样立即打印出“hello，world!”的内容，并且默认支持语法和代码高亮。

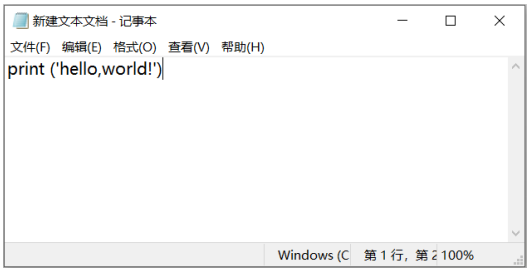


1.2.2　脚本模式

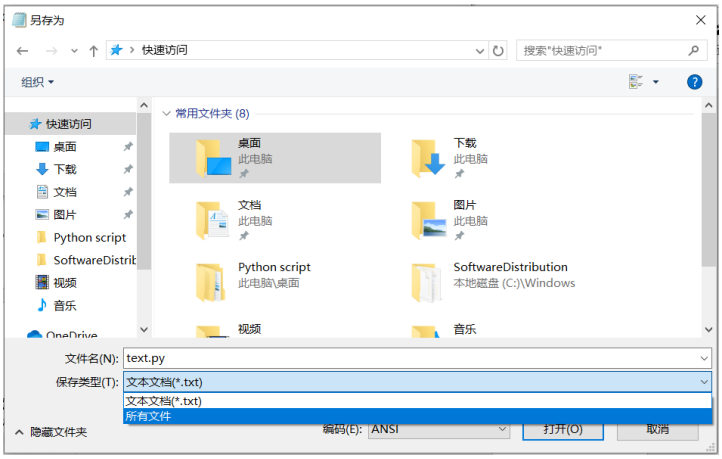
在 Windows 里，有两种方法创建 Python 脚本，一种是将代码写进 Windows 记事本里，另一种是借助第三方编辑器。两种方法分别介绍如下。

1. 使用记事本创建 Python 脚本

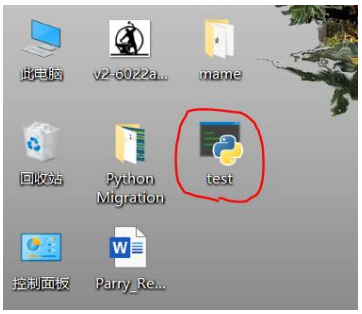
在桌面上新建一个记事本文件，将代码 print ('hello,world!')写入，如下图所示。



然后将其另存为.py 格式存放在桌面上。这里需要将“保存类型”选择为“所有文件”，否则该文件的类型依然为.txt，如下图所示。



回到桌面，可以发现第一个 Python 脚本已经创建成功，如下图所示。



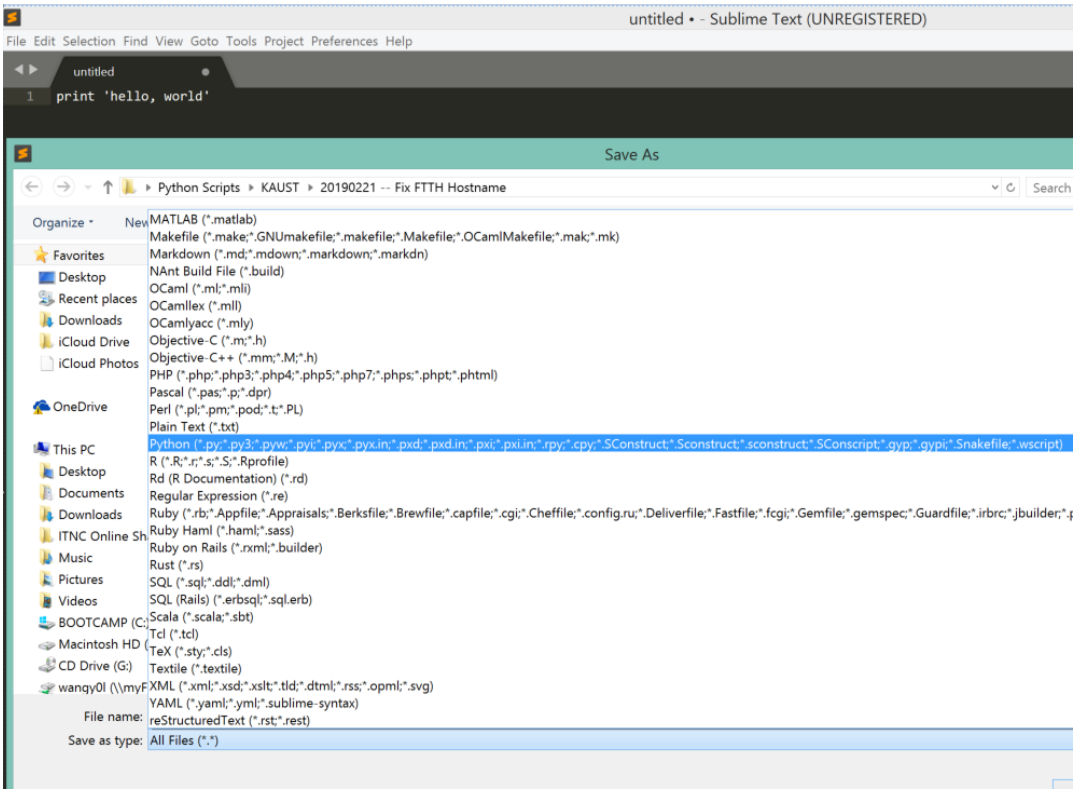
2. 使用第三方编辑器创建 Python 脚本

支持 Python 的第三方编辑器很多，Pycharm、Sublime Text 2/3、Notepad ++、vim（Linux 系统）和 Python 自带的 IDLE 等都是很优秀也很常用的编辑器。本书以 Sublime Text 3 为例简单介绍使用第三方编辑器创建 Python 脚本的方法。

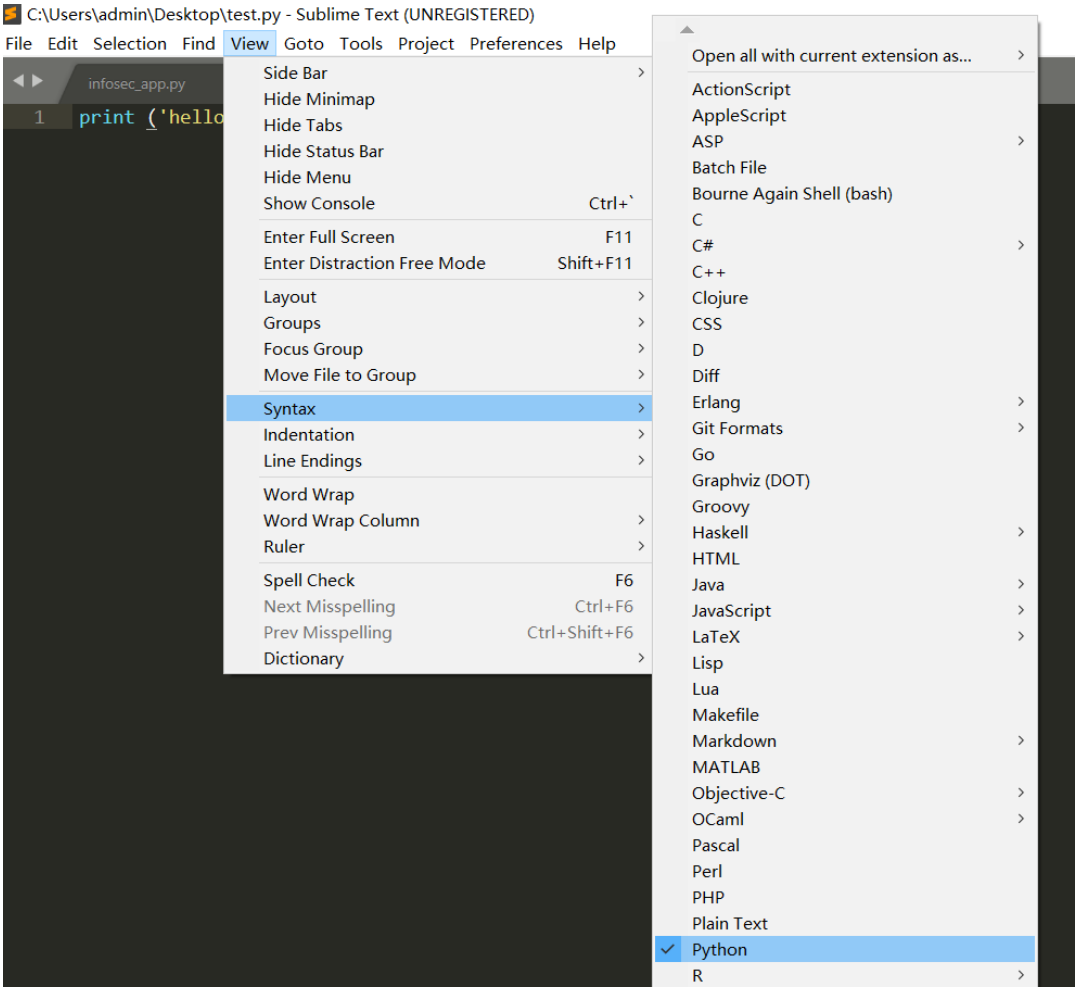
首先在 Sublime Text 官网下载 Sublime Text 3。Sublime Text 为付费软件，但是也可以免费使用，免费版本每使用几次后会弹出一个窗口问你是否愿意购买付费版本，如果你不愿意付费，将窗口关闭即可，基本不会影响使用体验。

Sublime Text 支持近 50 种编程语言，默认句法（Syntax）是 Plain Text。在 Plain Text

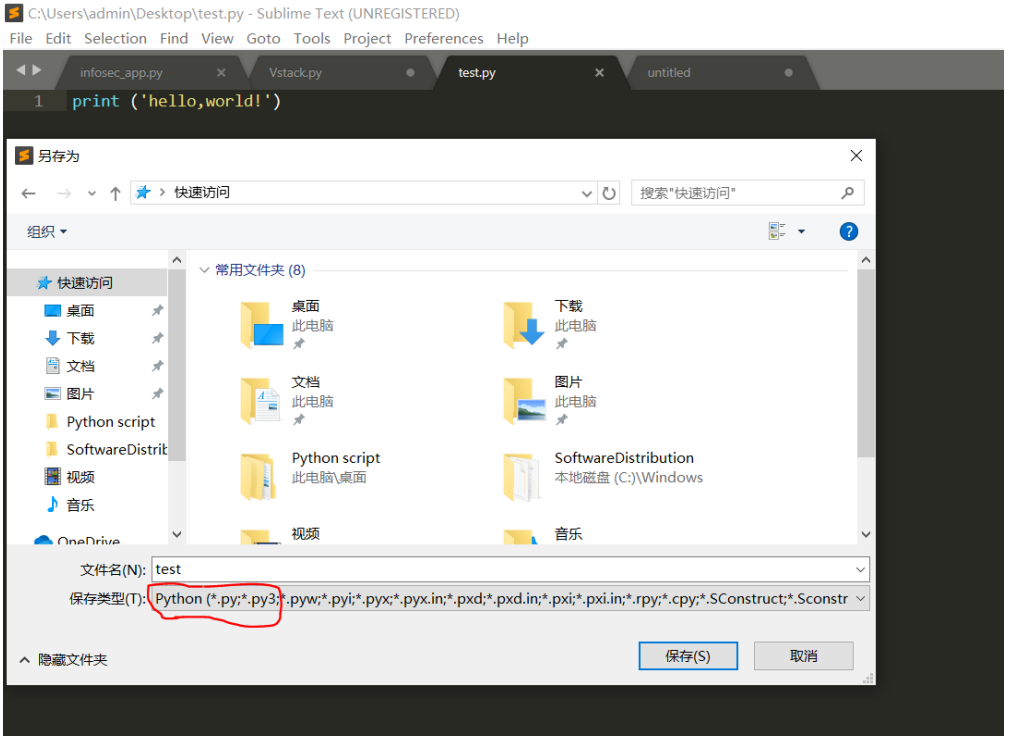
下写出来的 Python 代码的效果和记事本没有区别，依然只有黑白两色，而且保存文件的时候依然需要手动将文件另存为.py 格式，如下图所示。



因此，在进入 Sublime Text 后需要做的第一件事是选择 View→ Syntax → Python 将句法改为 Python，这样才能获得对 Python 最好的支持，包括代码高亮、语法提示、代码自动补完、默认将脚本保存为.py 格式等诸多实用功能，如下图所示。



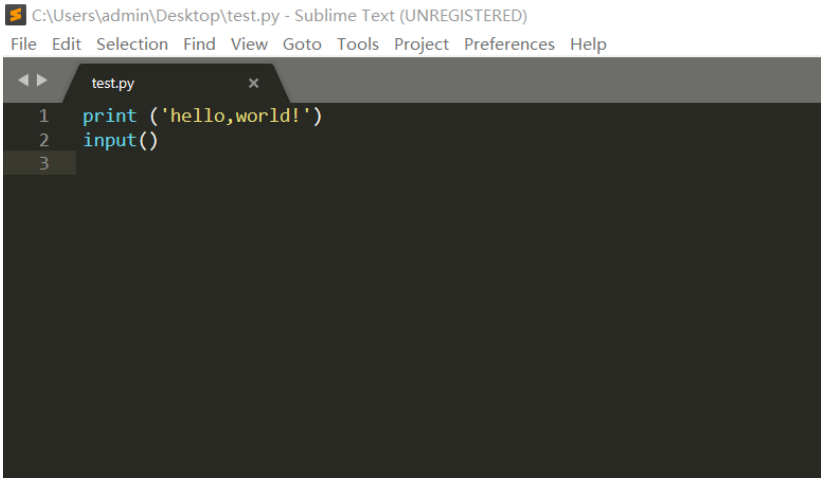
将句法改为 Python 后，代码立刻变为高亮，并且保存代码的时候文件类型已经自动默认为.py 格式，如下图所示。



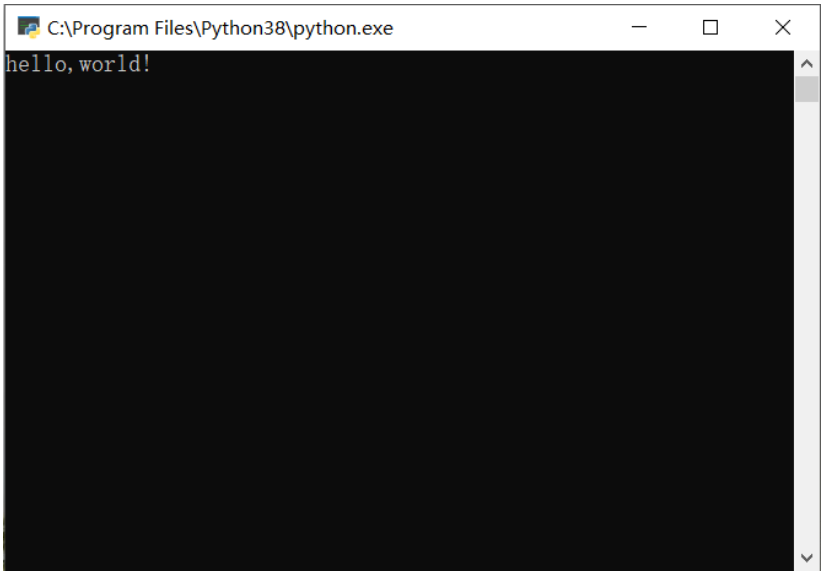
1.2.3　运行 Python 脚本

在 Windows 里，有四种运行脚本的方法。

第一种方法是双击执行.py 文件，这种方法的缺点是在双击运行脚本后，你会看到一个“闪退”的命令行窗口，窗口闪退速度很快，从弹出到消失只有 0.1～0.2s，肉眼刚刚能看到窗口的轮廓，但是无法看清窗口的内容。这是因为 Python 脚本程序执行完后自动退出了，要让窗口停留，需要在代码最后写上一个 input()，如下图所示。

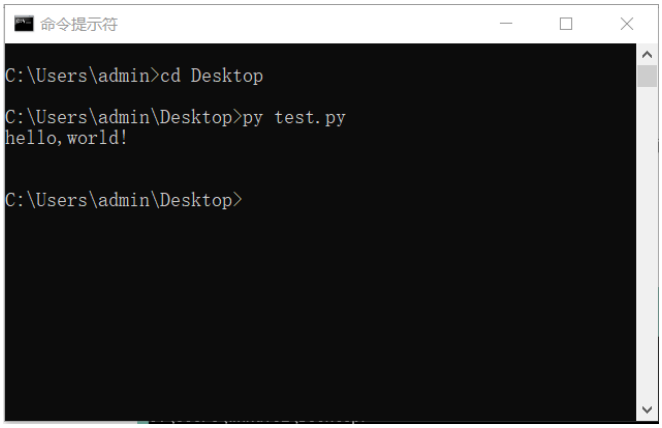


然后用同样的方法将该脚本另存为.py 文件，再次双击运行该脚本。效果如下图所示。

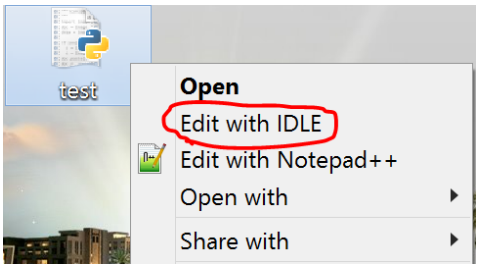


关于 input()会在 2.4.1 节中详细解释，这里只需要知道可以用它来解决双击运行 Python 脚本时窗口闪退的问题即可。

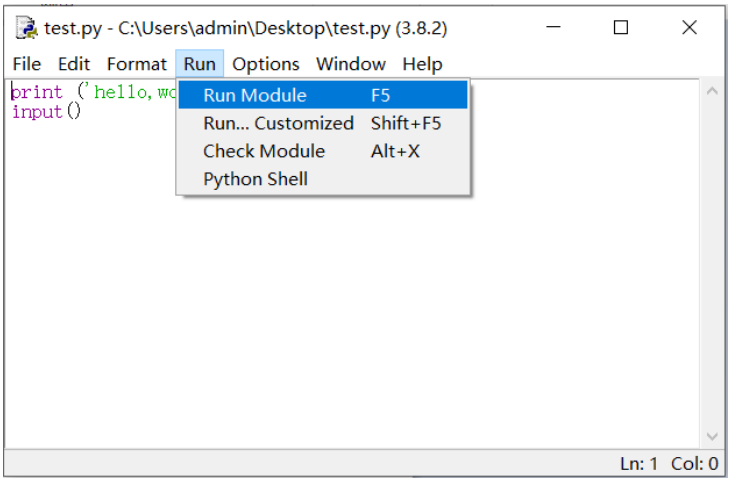
第二种方法是在命令行里移动到脚本文件所在的文件夹下，输入 y xxx.py 或 python xxx.py 命令来运行脚本，如下图所示。



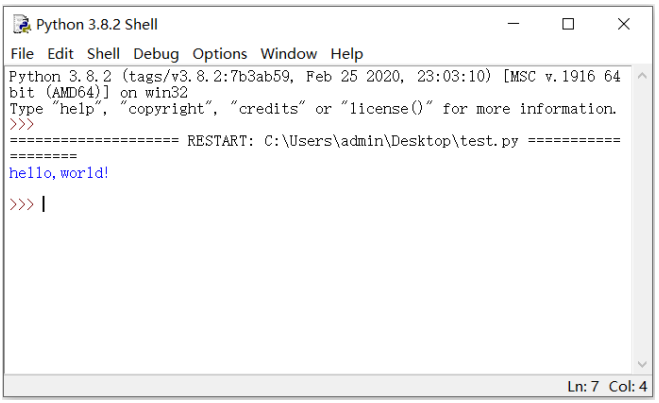
第三种方法是使用 IDLE 来运行脚本，具体步骤为：首先右键单击脚本文件，选择“Edit with IDLE”进入 IDLE，如下图所示。



然后在 IDLE 里单击 Run →Run Module 来运行脚本，如下图所示。

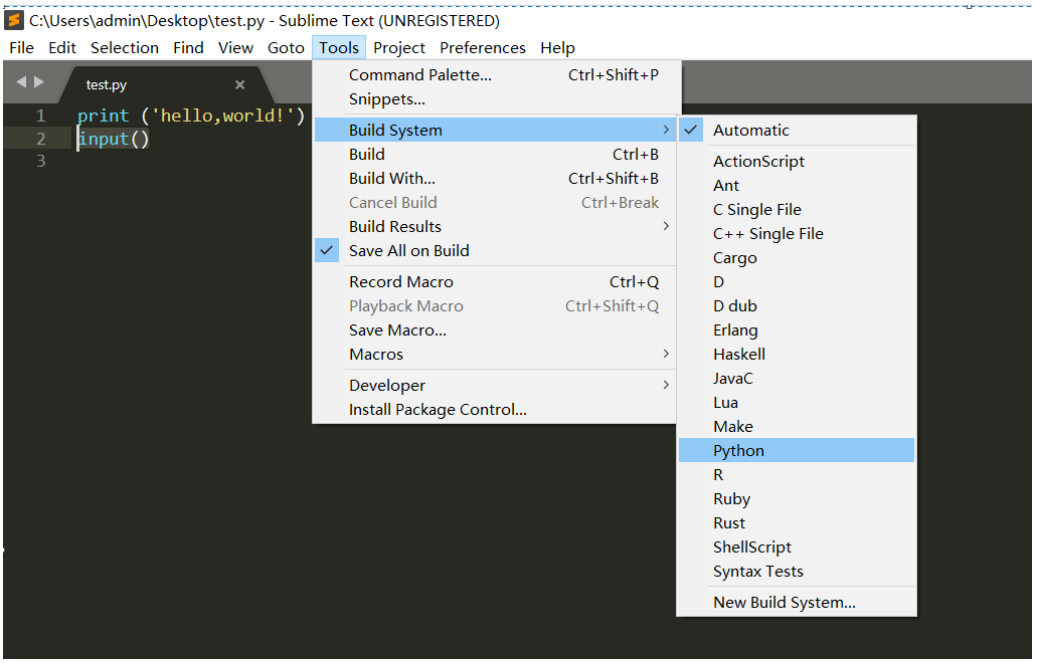


得到的效果如下图所示。

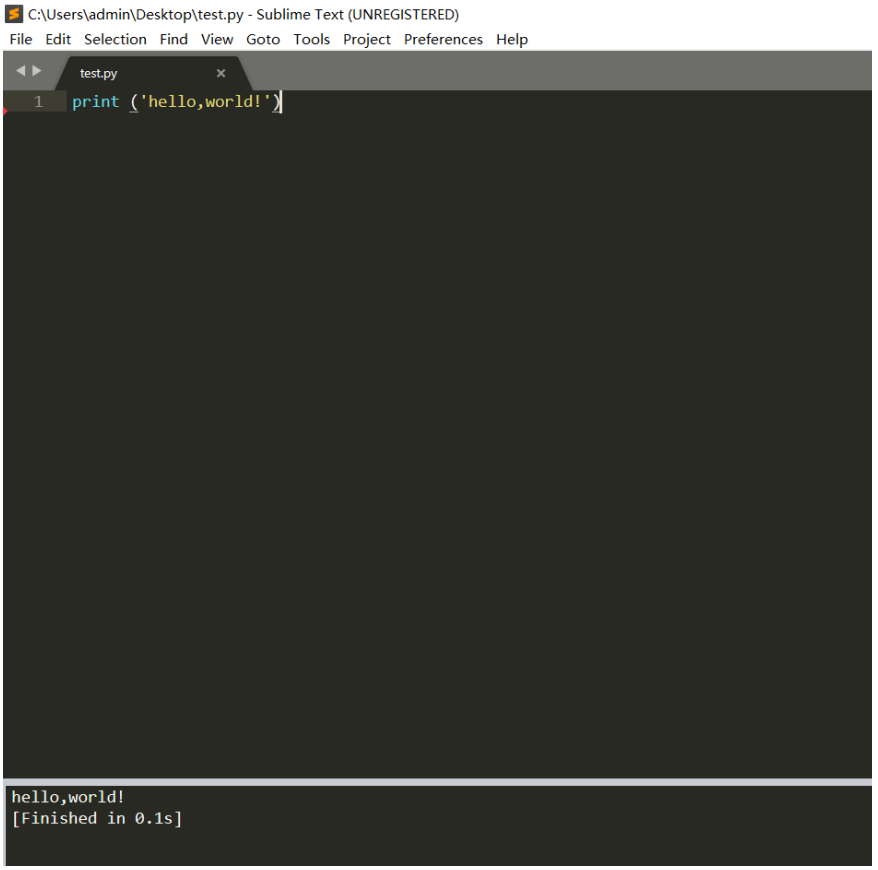


可以发现，在 IDLE 里即使不使用 input()，运行脚本时也不会出现窗口闪退的问题，因此，通常建议使用 IDLE 来运行脚本。

第四种方法是在第三方编辑器里运行脚本。依然以 Sublime Text 3 为例，方法很简单，首先进入 Sublime Text 3，如下图所示，依次选择 Tools → Build System → Python。



写好代码并保存后，打开 Tools → Build 或者使用快捷键 Ctrl + B 就可以在窗口底部看到运行脚本的结果了，效果如下图所示。



1.3　在 Linux 上使用 Python 3.8.2

前面提到本书只介绍在 CentOS 命令行终端里使用 Python 的方法，在 GNOME 桌面环境下使用 Python 的方法不在本书的讨论范围内。下面介绍交互模式和脚本模式在 CentOS 中的使用方法。

1.3.1 交互模式

我们知道，在 CentOS 的命令行终端里输入命令 python3.8 即可进入 Python 3.8.2 的解释器，也就进入了 Python 的交互模式，如下所示。

```
[root@CentOS-Python ~]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

在 Python 解释器里输入第一段代码 print ('hello, world!')，解释器随即打印出了“hello, world!”的内容。这种“即时反馈”的特性是交互模式下特有的，脚本模式下不具备，如下图所示。

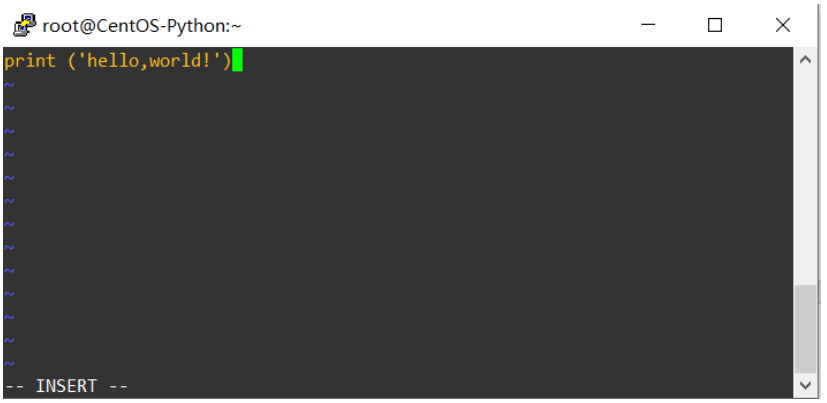
```
root@CentOS-Python:~
[root@CentOS-Python ~]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print ('hello,world!')
hello,world!
>>>
```

1.3.2 脚本模式

在 CentOS 的命令行终端里，我们可以使用文本编辑器来创建脚本，CentOS 有几种常见的文本编辑器，比如 emacs、nano、vi 等。这里介绍用 vi 创建 Python 脚本的方法。

关于 vi 的用法本书将只做简单介绍。另外，vi 有一个加强版本叫作 vim，两者的具体区别不在本书的讨论范围内。读者只需要知道在创建 Python 脚本时，vim 支持语法高亮，而 vi 不支持。

vi 不支持语法高亮，仅显示 PuTTY 默认字体的颜色，如下图所示。

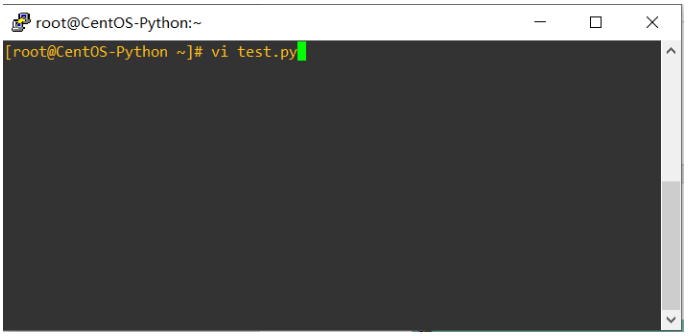


vim 支持语法高亮，显示彩色字体，如下图所示。

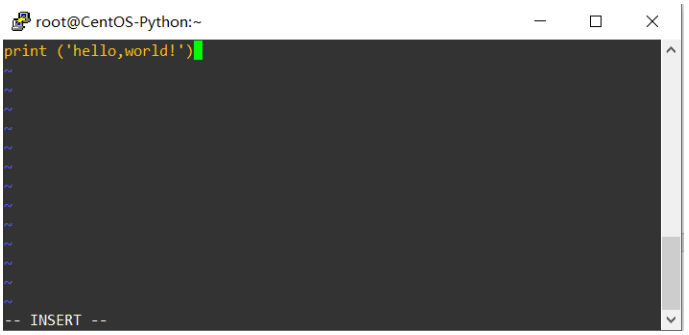
除此之外，两者对 Python 的支持并无本质区别，是否喜欢语法高亮全凭个人喜欢。只需要注意一点：vi 是 CentOS 安装时自带的文本编辑器，vim 则需要通过输入命令“yum install vim”安装后才能使用。

下面用实例介绍使用 vi 创建 Python 脚本的方法。

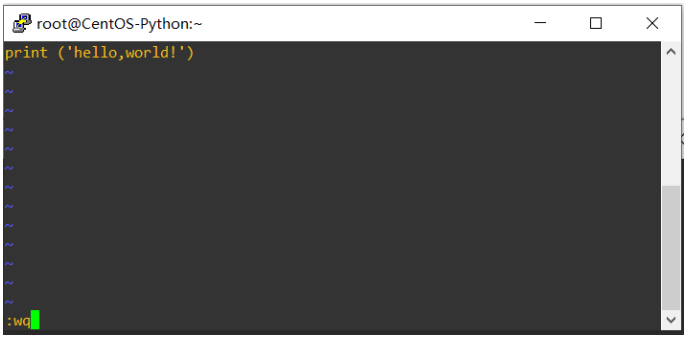
首先输入命令 vi test.py 创建一个名为 test.py 的 Python 脚本，如下图所示。



进入 vi 后，按“i”键进入输入模式（屏幕左下角会显示“- - INSERT - -”），输入第一段代码 print (‘hello, world!’)，如下图所示。



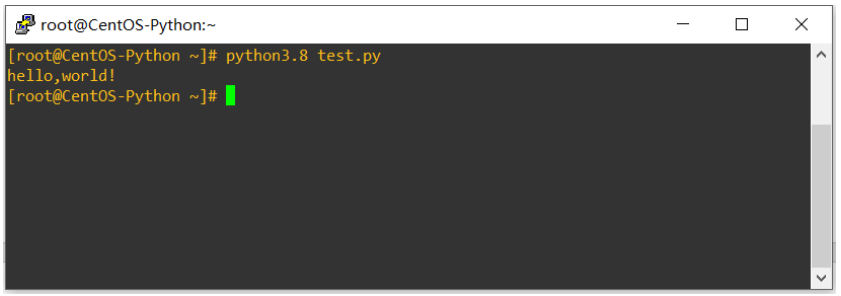
然后按“ESC”键，在屏幕左下角的“- - INSERT - -”消失后，接着输入:wq，按回车键后即可保存文件并退出 vi，如下图所示。



之后回到命令行终端，输入 ls 即可看到刚刚创建成功的 Python 脚本 test.py，如下图所示。

1.3.3　运行 Python 脚本

与 Windows 命令行终端一样，在 CentOS 命令行终端也是通过输入 python xxx.py 来运行 Python 脚本的，不同的是，因为本书以 Python 3.8.2 为例，这里需要把 python 换成 python3.8，写成 python3.8 xxx.py 来运行脚本，如下图所示。



1.3.4　Shebang 符号

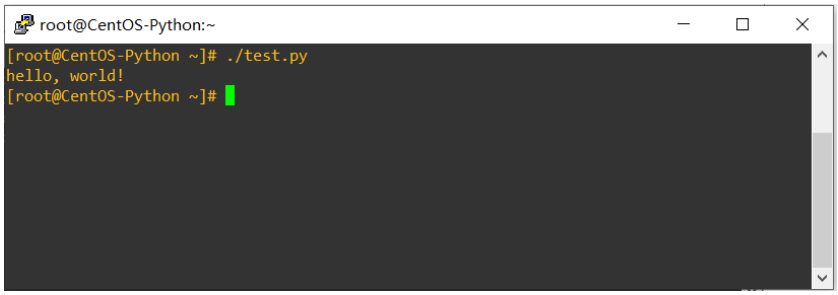
在 Linux 和 Unix 里，符号#!叫作 Shebang，通常可以在 Linux/Unix 系统脚本中第一行的开头看到它。它的作用是指明执行脚本文件的解释程序。写在 Shebang 后面的解释程序

如果是一个可执行文件，则当执行脚本时，Shebang 会把文件名作为参数传递给解释程序去执行。比如 python3.8 test.py 中的 python3.8 是解释程序，test.py 是文件名，使用 Shebang 后，可以省去解释程序，把 python3.8 test.py 写成./test.py 就可以运行 Python 脚本了。另外，Shebang 指定的解释程序必须为可执行程序，否则系统会报错“Permission denied.”。

因此，如果你觉得每次都需要输入命令 python2、python3 或者 python3.8 来运行脚本比较麻烦，则可以在脚本的开头部分使用 Shebang 符号，然后在其后面加上/usr/bin/env python3 来指定 python 3 为解释程序（同理，如果你想使用 python 2 来做解释程序，则可以写成#!/usr/bin/env python2），如下图所示。

保存退出后，用 chmod 命令将 test.py 改为可执行，如下图所示。

然后就可以用./test.py 来运行脚本，省去每次必须都输入命令 python2、python3 或者 python3.8 的麻烦，如下图所示。


root@CentOS-Python:~
[root@CentOS-Python ~]# ./test.py
hello, world!
[root@CentOS-Python ~]#

第 2 章

Python 基本语法

本章主要介绍 Python 的基本语法知识，举例演示的所有代码都将在 CentOS 中执行，如果代码前面带有>>>符号，则为在解释器下运行的代码，不带>>>符号的代码即为脚本模式下运行的代码。

2.1 变量

所谓变量（Variable），顾名思义，指在程序运行过程中，值会发生变化的量。与变量相对应的是常量，也就是在程序运行过程中值不会发生变化的量，不同于 C/C++等语言，Python 并没有严格定义常量这个概念，在 Python 中约定俗成的方法是使用全大写字母的命名方式来指定常量，比如圆周率 PI=3.1415926。

变量是存储在内存中的一个值，在创建一个变量后，也就意味着在内存中预留了一部分空间给它。变量用来指向同样存储在内存中的一个对象，每个对象根据自身情况又可以代表不同的数据类型（Data Type）。我们可以通过**变量赋值**这个操作将变量指向一个对象，比如下面的 a = 10 即一个最简单的变量赋值的示例。

```
>>> a = 10
```

在 Python 中，我们使用等号=来连接变量名和值，进而完成变量赋值的操作。这里将 10 这个整数（也就是内存中的对象）赋值给变量 a，因为 10 本身是**“整数”（Integer）**，

所以变量 a 此时就代表了“整数”这个数据类型。我们可以使用 type()函数来确认 a 的数据类型，发现变量 a 的数据类型此时为 int，也就是 integer 的缩写，代码如下。

```
>>> type(a)
<class 'int'>
>>>
```

Python 是一门动态类型语言，和 C、Java 等不同，我们无须手动指明变量的数据类型，根据赋值的不同可以随意更改一个变量的数据类型。举例来说，刚才我们把“整数”这个数据类型赋值给变量 a，现在再次赋值一个内容为 test 的**“字符串”（String）**数据类型给变量 a，然后用 type()函数确认，这时 a 的数据类型已经从 int 变为了 str，即字符串，代码如下。

```
>>> a = 'test'
>>> type(a)
<class 'str'>
>>>
```

变量名可以用大小写英文字母、下划线、数字来表示，但是不能包含标点符号、空格及各类其他特殊符号，比如括号、货币符号等。

变量名可以以字母和下划线开头，**但是不能以数字开头**，举例如下。

```
>>> test = 'test'
>>> _a_ = 1
>>> 123c = 10
  File "<stdin>", line 1
    123c = 10
       ^
SyntaxError: invalid syntax
>>>
```

这里 Python 解释器返回了“SyntaxError: invalid syntax”这个无效语法的错误提示，告诉我们 123c 为无效的变量名。这也是使用解释器来学习 Python 的优势，无论代码里出现什么问题，都能得到“即时反馈”。

变量名区分大小写，举例如下。

```
>>> a = 10
>>> print (A)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'A' is not defined
>>> print (a)
10
>>>
```

如果变量名中间出现两个或以上的单词，则只能用下划线将它们连接，不可以使用空格将它们隔开，举例如下。

```
>>> ip_address = '192.168.1.1'
>>> ip address = '192.168.1.1'
  File "<stdin>", line 1
    ip address = '192.168.1.1'
       ^
SyntaxError: invalid syntax
>>>
```

最后，不是所有的英文单词都能用作变量名，Python 中有**保留字（Reserved Word）**的概念，保留字通常是 Python 中常用的关键词，比如用作创建函数的“def”，用作 while 循环和 for 循环的“while”和“for”，等等。可以用下面的方法来查询当前的 Python 版本中有哪些保留字。

```
>>> import keyword
>>> print (keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break',
'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from',
'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass',
'raise', 'return', 'try', 'while', 'with', 'yield']
>>>
```

上面的代码看不懂没关系，后面会讲到它们的用法。这里只需注意输入 print

(keyword.kwlist)后的返回值为一个列表，该列表中的元素（列表和元素的概念后文会讲到）即当前 Python 版本中的保留字，这些保留字均不能用来做变量名，举例如下。

```
>>> and = 1
  File "<stdin>",  line 1
    and = 1
      ^
SyntaxError: invalid syntax
>>> as = 1
  File "<stdin>",  line 1
    as = 1
     ^
SyntaxError: invalid syntax
>>> else = 1
  File "<stdin>",  line 1
    else = 1
       ^
SyntaxError: invalid syntax
>>>
```

2.2　注释

在编程中，尤其是当编写代码内容较多的大型程序时，**注释（Comments）**可以起到备注的作用，在回顾代码时帮助回忆和理解。除此之外，在与团队合作的时候，在代码中使用注释也是极其重要的一项要求，因为你写的代码可能会被他人调用、维护，为了让他人更容易理解你写的代码的目的和用途，那么在代码中使用注释是非常必要的。

在 Python 中，我们使用#来做注释符号，和#写在同一排，并且写在#后面的代码将只做注释使用，不会被当作代码的一部分，也就不会被执行。大多数情况下我们会在脚本模式中用到注释，在交互模式中使用注释的情况很少，举例如下。

```
# coding=utf-8
#生成一个整数列表，该列表为整数 1～10 的平方数的集合
test_list = [i ** 2 for i in range(1,11)]
```

```
print (test_list)
```

这段代码也许你看不懂，但是通过注释你知道它的作用是“生成一个整数列表，该列表为整数 1～10 的平方数的集合”，也就是 1，4，9，16，…，100。我们可以运行该脚本查看输出内容。

需要注意的是，如果使用脚本模式运行 Python，并且代码中出现了中文，则必须在代码的开头加上一段“# coding=utf-8”，因为 Python 默认的编码格式是 ASCII，如果不修改编码格式，则 Python 将无法正确显示中文，如下图所示。

因为写在#后面的代码只做注释使用，并不会被当做代码的一部分执行，因此有时我们还可以巧用#来“遮盖”我们不想执行的代码。比如你可以选择性地在脚本的某些 print()函数前加上#，不想看输出内容时把#加在 print 前面，想看输出内容时把#拿掉，而不至于每次都要在脚本里反复删除、重写该段 print()函数。

2.3 方法和函数

在 Python 中，方法（Method）和函数（Function）大体来说是可以互换的两个词，它们之间有一个细微的区别：函数是独立的功能，无须与对象关联。方法则与对象有关，不需要传递数据或参数就可以使用。举个例子，前面讲到的 type()就是一个函数，代码如下。

```
>>> a = 123
>>> type(a)
```

```
<class 'int'>
>>>
>>> type('xyz')
<class 'str'>
>>>
```

方法则需要与一个对象（变量或数据）关联，比如 upper()是一个方法，它的作用是将字符串内容里小写的英文字母转换为大写的英文字母，代码如下。

```
>>> vendor = 'Cisco'
>>> vendor.upper()
'CISCO'
>>>
```

这里我们创建了一个名为 vendor 的变量，并将字符串内容“Cisco”赋值给它，随后对该变量调用 upper()方法，返回值即所有字母都变为大写的“CISCO”。

在 Python 中，每种数据类型都有自己默认自带的函数、方法和变量，要查看某一数据类型本身具有的函数、方法和变量，可以使用 dir()函数，这里以字符串和整数为例，代码如下。

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_formatter_field_name_split', '_formatter_parser',
'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']

>>> dir(int)
```

```
    ['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__',
'__delattr__', '__div__', '__divmod__', '__doc__', '__float__', '__floordiv__',
'__format__', '__getattribute__', '__getnewargs__', '__hash__', '__hex__',
'__index__', '__init__', '__int__', '__invert__', '__long__', '__lshift__',
'__mod__', '__mul__', '__neg__', '__new__', '__nonzero__', '__oct__', '__or__',
'__pos__', '__pow__', '__radd__', '__rand__', '__rdiv__', '__rdivmod__',
'__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__',
'__rmod__', '__rmul__', '__ror__', '__rpow__', '__rrshift__', '__rshift__',
'__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
'__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'bit_length', 'conjugate', 'denominator', 'imag', 'numerator', 'real']
    >>>
```

以上即使用 dir()函数列出的字符串和整数所自带的函数、方法和变量。注意，其中前后带单下划线或双下划线的变量不会在本书中介绍，比如“_formatter_parser”和“__contains__”，初学 Python 的网络工程师只需要知道它们在 Python 中分别表示私有变量与内置变量，学有余力的网络工程师可以自行阅读其他 Python 书籍深入学习，其他不带下划线的函数与方法并不是每一个都会在网络运维中用到，笔者将在下一章中选取讲解。

2.4 数据类型

前面讲到，我们可以使用变量来指定不同的数据类型，对网络工程师来说，常用的数据类型有字符串（String）、整数（Integer）、列表（List）、字典（Dictionary）、浮点数（Float）、布尔（Boolean）。另外，不是很常用但需要了解的数据类型包括集合（Set）、元组（Tuple）及空值（None），下面一一举例讲解。

2.4.1 字符串

字符串即文本，可以用单引号''、双引号""和三引号''' '''表示，下面分别介绍三者的区别和用法。

1. 单引号和双引号

当表示内容较短的字符串时，单引号和双引号比较常用且两者用法相同，比如'cisco' 和 "juniper"，需要注意的是**单引号和双引号不可以混用。**

```
>>> vendor1 = 'Cisco'
>>> vendor2 = "Juniper"
>>> vendor3 = 'Arista"
  File "<stdin>", line 1
    vendor3 = 'Arista"
                     ^
SyntaxError: EOL while scanning string literal
>>> vendor3 = 'Arista'
```

这里创建了三个变量：vendor1、vendor2 和 vendor3，分别将字符串 Cisco、Juniper 和 Arista 赋值给它们，因为字符串 Arista 混用了单引号和双引号，导致解释器报错，重新给 vendor3 赋值并且只使用单引号后解决了这个问题。这时我们可以用 print 语句（Statements）将三个变量的内容打印出来，如下所示。

```
>>> print (vendor1)
Cisco
>>> print (vendor2)
Juniper
>>> print (vendor3)
Arista
>>>
```

除了使用 print()函数，我们还可以在解释器里直接输入变量名来获取它的值，这是编辑器交互模式下特有的，脚本模式做不到，举例如下。

```
>>> vendor1
'Cisco'
>>> vendor2
'Juniper'
>>> vendor3
'Arista'
>>>
```

需要指出的是，如果变量中存在换行符\n，则 print 会执行换行动作，但如果在解释器里直接输入变量名，则解释器会把换行符\n 当作字符串内容的一部分一起返回，举例如下。

```
>>> banner = "\n\n  Warning: Access restricted to Authorised users only. \n\n"
>>> print banner


  Warning: Access restricted to Authorised users only.


>>> banner
'\n\n  Warning: Access restricted to Authorised users only. \n\n'
>>>
```

看出区别了吗？

在 Python 中，我们还可以通过加号+来拼接（Concatenate）字符串，举例如下。

```
>>> ip = '192.168.1.100'
>>> statement = '交换机的 IP 地址为'
>>>
>>> print (statement + ip)
交换机的 IP 地址为 192.168.1.100
```

注意，在使用加号+将变量拼接合并时，如果其中一个变量为字符串，那么其他所有要与之拼接的变量也都必须为字符串，否则 Python 会报错，举例如下。

```
>>> statement1='网段 192.168.1.0/24 下有'
>>> quantity = 60
>>> statement2='名用户'
>>>
>>> print (statement1 + quantity + statement2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

这里 statement1 和 statement2 两个变量都为字符串，但是 quantity 这个变量为整数，因此 print statement1 + quantity + statement2 会报错 TypeError: cannot concatenate 'str' and 'int' objects，提示不能将字符串和整数拼接合并。解决的办法是使用 str()函数将 quantity 从整数转化为字符串，代码如下。

```
>>> print (statement1 + str(quantity) + statement2)
网段 192.168.1.0/24 下有 60 名用户
>>>
```

2. 三引号

三引号形式的字符串通常用来表示内容较长的文本文字，它最大的好处是如果遇到需要换行的文本，文本内容里将不再需要用到换行符\n。比如路由器和交换机中用来警告非授权用户非法访问设备后果的 MOTD（Message of The Day）之类的旗标（Banner）配置，此类文本内容通常比较长且需要换行，这时用三引号来表示该文本内容是最好的选择，举例如下。

```
>>> motd = ''' ----------------------------------------------------------------
...
... Warning: You are connected to the Cisco systems， incorporated network.
... Unauthorized access and use of this network will be vigorously prosecuted.
...
... ----------------------------------------------------------------'''
>>>
>>> print (motd)
---------------------------------------------------------------------------
Warning: You are connected to the Cisco systems， incorporated network.
Unauthorized access and use of this network will be vigorously prosecuted.
---------------------------------------------------------------------------
>>>
```

3. 与字符串相关的方法与函数

◎ upper()

前面提到过 upper()方法，它的作用是将字符串内容里小写的英文字母转换为大写的英文字母。**upper()的返回值是字符串**，举例如下。

```
>>> vendor = 'Cisco'
>>> vendor.upper()
'CISCO'
>>>
```

◎ lower()

顾名思义，与 upper()相反，lower()方法的作用是将字符串内容里大写的英文字母转换为小写的英文字母。**lower()的返回值是字符串**，举例如下。

```
>>> vendor = 'CISCO'
>>> vendor.lower()
'cisco'
>>>
```

◎ strip()

strip()用来在字符串的开头和结尾移除指定的字符（如字母、数字、空格、换行符\n、标点符号等）。如果没有指定任何参数，则默认移除字符串开头和结尾处的所有空格和换行符\n。有时在字符串内容的开头和结尾会夹杂一些空格，如' 192.168.100.1 '，要拿掉这些多余的空格，可以使用 strip()。**strip()的返回值是字符串**，举例如下。

```
>>> ip='  192.168.100.1  '
>>> ip.strip()
'192.168.100.1'
>>>
```

有时在字符串末尾会有换行符\n（比如使用 open()函数的 readlines()方法来读取文本文

件里的内容后所返回的列表里的元素，后文会讲到），我们也可以使用 strip()来移除这些换行符，举例如下。

```
>>> ip='192.168.100.1\n'
>>> ip.strip()
'192.168.100.1'
>>>
```

◎　count()

count()用来判断一个字符串内容里给定的字母或数字具体有多少个，比如要找出'39419591034989320'这个字符串里面有多少个数字 9，就可以用 count()方法。**count()的返回值是整数**，举例如下。

```
>>> '39419591034989320'.count('9')
5
>>>
```

◎　len()

len()用来判断字符串内容的长度，比如要回答上面提到的'39419591034989320'总共是多少位数，就可以用 len()方法。**len()的返回值是整数，**举例如下。

```
>>> a='39419591034989320'
>>> len(a)
17
>>>
```

◎　split()和 join()

之所以把这两个方法放在一起讲，是因为它俩关系比较接近，在字符串、列表的转换中互成对应的关系，split()将字符串转换成列表，join()将列表转换成字符串。

目前为止，我们还没有讲到列表（List），这里简单讲解一下：在 Python 中，列表是一种有序的集合，用中括号[]表示，该集合里的数据又被叫作元素，比如[1,3,5,7,9]就是一个最简单的列表，其中的整数 1、3、5、7、9 都属于该列表的元素。下面我们把该列表赋

值给变量 list1，用 type()来确认该变量的数据类型，可以发现它的数据类型为 list。

```
>>> list1 = [1,3,5,7,9]
>>> type(list1)
<class 'list'>
```

我们可以使用索引来访问和指定列表中的每个元素，索引的顺序是从数字 0 开始的。列表索引的用法举例如下。

```
>>> list1[0]
1
>>> list1[1]
3
>>> list1[2]
5
>>> list1[3]
7
>>> list1[4]
9
>>>
```

讲完列表后，为了配合下面的案例，这里需要讲一下 input()函数。

- **input()的返回值是字符串。**它的作用是提示用户输入数据与 Python 程序互动，比如你想写一段代码询问用户的年龄多大，让用户自己输入年龄，可以写一段这样的脚本代码。

```
[root@CentOS-Python ~]# cat demo.py

age = input('How old are you? ')
print ('Your age is: ' + age)

[root@CentOS-Python ~]#
```

然后执行该脚本代码。

```
[root@CentOS-Python ~]# python demo.py
```

```
How old are you? 32
Your age is: 32

[root@CentOS-Python ~]#
```

注意这里的 32 是用户自己输入的，虽然它看着像整数，但是它实际的数据类型是字符串。

- 在 Python 2 中，上面提到的 Python 3 的 input()函数实际对应的是 Python 2 的 raw_input()，而 Python 2 中的 input()函数只能用来接收数字（整数或浮点数），并且相对应的返回值也为整数或浮点数。实际上 Python 3 将 Pyhon 2 中的 input()和 raw_input()整合成了 input()（在 Python 3 中已经没有 raw_input()，这点需要注意）。

在了解了列表和 input()函数大致的原理和用法后，再来看网络工程师如何在网络运维中使用 split()和 join()。举例来说，在大中型公司里，IP 地址的划分一般是有规律可循的，比如说某公司有一栋 10 层楼的建筑，一楼的 IP 子网为 192.168.1.0/24，二楼为 192.168.2.0/24，三楼为 192.168.3.0/24，依此类推。现在你需要做个小程序，让用户输入任意一个属于公司内网的 IP 地址，然后让 Python 告诉用户这个 IP 地址属于哪一层楼。思路如下：

因为该公司内网 IP 地址的第一段都为 192，第二段都为 168，第四段不管用户输入任何 IP 地址都不影响我们对楼层的判断。换句话说，我们只能从该 IP 地址的第三段来判断是哪一层楼，但是我们要怎样告诉 Python 去判断哪一个数字属于 IP 地址的第三段呢？这时就可以用 split()将用户输入的 IP 地址（字符串）转化成列表，然后通过列表的索引来指向 IP 地址的第三段，代码如下。

```
>>> floor1='192.168.1.0'
>>> floor1_list = floor1.split('.')
>>>
>>> print floor1_list
['192', '168', '1', '0']
>>>
```

```
>>> floor1_list[2]
'1'
>>>
```

我们先将'192.168.1.0'赋值给 floor1 这个变量，再对该变量调用 split()方法，然后将返回值赋给另一个变量 floor1_list，注意 split()括号里的“.”表示分隔符，该分隔符用来对字符串进行切片，因为 IP 地址的写法都是 4 个数字用 3 个点“.”分开，所以这里分隔符用的是“.”，**因为 split()的返回值是列表，所以我们 print floor1_list 后可以看到，IP 地址的四个数字已经被切片独立开来，分别成为组成 floor1_list 列表的四个元素的其中之一**，之后我们就可以通过 floor1_list[2]这种索引的方式来查询该列表的第三个元素，从而得到 IP 地址的第三段的数字了，也就是这里的数字 1。

在知道怎么通过 split()来获取 IP 地址的第三段的数字后，回到前面的需求：让用户输入任意一个属于公司内网的 IP 地址，然后让 Python 告诉用户这个 IP 地址属于哪一层楼。脚本代码如下。

```
[root@CentOS-Python ~]# cat demo.py

# coding=utf-8
ip = input('请输入要查询的 IP 地址: ')
ip_list = ip.split('.')
print ('该 IP 地址属于' + ip_list[2] + '楼.')

[root@CentOS-Python ~]#
```

我们使用 input()函数提示用户输入想要查询的 IP 地址，然后将得到的值（字符串）赋值给变量 ip，随后对其调用 split()函数，并将返回值（列表）赋给另一个变量 ip_list，接着通过 ip_list[2]做索引，得到该列表的第三个元素，也就是用户输入的 IP 地址的第三段，最后用 print 将查询的结果返回告知用户。

执行以下代码来看效果。

```
[root@CentOS-Python ~]# python demo.py
```

```
请输入要查询的 IP 地址：192.168.3.100
该 IP 地址属于 3 楼

[root@CentOS-Python ~]#
```

讲完 split()后，再来看 join()怎么用。首先来看下面这个列表，它包含了开启一个思科交换机端口的几条最基本的命令。

```
>>> commands = ['configure terminal', 'interface Fa0/1', 'no shutdown']
```

这几条命令缺少了关键的一点：换行符\n（也就是回车键），这时我们可以使用 join()将换行符\n 加在每条命令的末尾，注意 **join()的返回值是字符串。**

```
>>> '\n'.join(commands)
'configure terminal\ninterface Fa0/1\nno shutdown\n'
>>>
```

再举个例子，如果我们要把之前的列表['192', '168', '1', '0']转换回字符串'192.168.1.0'，可以采用如下做法。

```
>>> '.'.join(['192', '168', '1', '0'])
'192.168.1.0'
>>>
```

如果不加这个点“.”会怎样？试试看。

```
>>> ''.join(['192', '168', '1', '0'])
'19216810'
>>>
```

◎ startswith()，endswith()，isdigit()，isalpha()

之所以把上述四个字符串的函数和方法放在一起讲，是因为**它们的返回值都是布尔值（Boolean）。**布尔值只有两种：True 和 False，且首字母必须为大写，true 和 false 都不是有效的布尔值。布尔值通常用来判断条件是否成立，如果成立，则返回 True；如果不成立，则返回 False。

首先来看 startswith()，用来判断字符串的内容是否以给定的字符串开头，举例如下。

```
>>> ip = '172.16.5.12'
>>> ip.startswith('17')
True
>>> ip.startswith('172.')
True
>>> ip.startswith('192')
False
>>>
```

endswith()与 startswith()恰好相反，用来判断字符串的内容是否以给定的字符串结尾，举例如下。

```
>>> ip = '192.168.100.11'
>>> ip.endswith('1')
True
>>> ip.endswith('11')
True
>>> ip.endswith('2')
False
>>>
```

字符串的内容包罗万象，字符串可以为空，可以为中文汉字或英文字母，可以为整数或小数，可以为任何标点符号，也可以为上述任意形式的组合。而 isdigit()就是用来判断字符串的内容是否为整数的，举例如下。

```
>>> year='2019'
>>> year.isdigit()
True
>>> vendor='F5'
>>> vendor.isdigit()
False
>>> PI='3.1415926'
>>> PI.isdigit()
False
>>> IP='1.1.1.1'
```

```
>>> IP.isdigit()
False
>>>
```

isalpha()用来判断字符串的内容是否为英文字母，举例如下。

```
>>> chinese = '中文'
>>> chinese.isalpha()
False
>>> english = 'English'
>>> english.isalpha()
True
>>> family_name = 'Wang'
>>> family_name.isalpha()
True
>>> full_name = 'Parry Wang'
>>> full_name.isalpha()
False
>>> age = '33'
>>> age.isalpha()
False
>>>
```

注意，isalpha()很严格，只要字符串的内容出现了哪怕一个非英文字母，isalpha()就会返回 False，比如'Parry Wang'（包含了空格）、 'Parry_Wang'（包含了下划线）等。

2.4.2　整数和浮点数

在 Python 中，有 5 种数值类型（Numeric Type），分别为整数（Integer）、浮点数（Float）、布尔类型（Boolean）、长整数（Long）和复数（Complex），对网络工程师来说，掌握前面三种就够了，后面两种不是我们需要关心的。

所谓整数，即通常理解的不带小数点的正数或负数，浮点数则是带小数点的正数或负

数。可以通过 type()函数来验证，比如下面的 1 为整数，1.0 则为浮点数。

```
>>> type(1)
<class 'int'>
>>> type(1.0)
<class 'float'>
>>>
```

我们可以把 Python 当成一个计算器，使用+、-、*、//、**等**算术运算符**做加、减、乘、除、求幂等常见的数学计算，举例如下。

```
>>> 256 + 256
512
>>> 1.2 + 3.5
4.7
>>> 1024 - 1000
24
>>> 16 * 16
256
>>> 100/10
10
>>> 12 // 10
1
>>> 12 % 10
2
>>> 8**2
64
>>> 3**3
27
>>>
```

在 Python 中，可以通过运算符**做幂运算，比如 8 的 2 次方可以表示为 8**2，3 的 3 次方表示为 3**3。

在做除法时，可以看到示例中分别使用了/、//和%三个运算符，它们的区别如下。

/ 表示正常的除法运算，注意在 Python 2 中，如果碰到整数除整数，结果出现小数部

分的时候，比如 12 / 10，Python 2 只会返回整数部分，即 1，要想得到小数点后面的部分，必须将除数或被除数通过 float()函数换成浮点数来运算，举例如下。

```
[root@CentOS-Python ~]# python2
Python 2.7.16 (default, Nov 17 2019, 00:07:27)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 12/10
1
>>> 12/float(10)
1.2
>>> float(12)/10
1.2
>>>
```

而在 Python 3 中，12/10 则直接返回浮点数 1.2。

```
[root@CentOS-Python ~]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 12/10
1.2
>>>
```

// 表示向下取整，求商数。

```
>>> 12//10
1
>>>
```

% 则表示求余数。

```
>>> 12%10
2
>>>
```

整数也不单单用来做数学运算，通过加号+或乘号*两种运算符，**还可以与字符串互动**，

适合用来画分割线，举例如下。

```
>>> print ('CCIE ' * 8)
CCIE CCIE CCIE CCIE CCIE CCIE CCIE CCIE
>>>
>>> print ('CCIE ' + 'CCIE')
CCIE CCIE
>>>
>>> print ('*' * 50)
**************************************************
>>>
```

在网络运维中，有时会遇到需要用计数器做统计的时候，比如说某公司有 100 台思科 2960 的交换机，由于长期缺乏系统性的运维管理，交换机的 IOS 版本并不统一，为了统计其中有多少台交换机的 IOS 版本是最新的，需要登录所有的交换机，每发现一台 IOS 版本为最新的交换机就通过计数器加 1，直到结束。由于要完成这个脚本需要涉及 Paramiko、if、for 循环、正则表达式等进阶性的 Python 知识点，所以这里仅演示计数器的用法。

```
>>> counter = 0
>>> counter = counter + 1
>>> counter
1
>>> counter = counter + 1
>>> counter
2
>>> counter += 1
>>> counter
3
>>> counter += 1
>>> counter
4
>>>
```

首先我们创建一个变量 counter，将 0 赋值给它，该变量就是我们最初始的计数器。之

后如果每次发现有交换机的 IOS 版本为最新，我们就在该计数器上加 1，**注意 counter = counter + 1 可以简写为 counter + = 1。**

2.4.3　列表

列表（List）是一种有序的集合，用中括号[]表示，列表中的数据又被叫作元素（Element），每个元素之间都用逗号隔开。列表中的元素的数据类型可以不固定，举例如下。

```
>>> list1 = [2020, 1.23, 'Cisco', True, None, [1, 2, 3]]
>>>
>>> type(list1[0])
<class 'int'>
>>> type(list1[1])
<class 'float'>
>>> type(list1[2])
<class 'str'>
>>> type(list1[3])
<class 'bool'>
>>> type(list1[4])
<class 'NoneType'>
>>> type(list1[5])
<class 'list'>
>>>
```

由上可知，我们创建了一个名为 list1 的变量（注意 list 在 Python 中是保留字，并不能被用作变量名，所以我们用 list1 作为变量名），并将一个含有 6 个元素的列表赋值给它。可以看到这 6 个元素的数据类型都不一样，我们使用 type()函数配合列表的索引来验证每个元素的数据类型，**列表的索引号从 0 开始，对应列表里的第 1 个元素**。可以发现从第 1 个到第 6 个元素的数据类型分别为整数、浮点数、字符串、布尔值、空值，以及列表。

注：一个列表本身也可以以元素的形式存在于另一个列表中，举例来说，上面的列表

list1 的第 6 个元素为列表[1,2,3]，我们可以通过使用两次索引的方法来单独调取列表[1,2,3]中的元素，也就是整数 1、2、3。

```
>>> list1 = [2020, 1.23, 'Cisco', True, None, [1,2,3]]
>>> list1[5][0]
1
>>> list1[5][1]
2
>>> list1[5][2]
3
>>>
```

下面介绍与列表相关的方法和函数

◎ range()

range()函数在 Python 2 和 Python 3 中有较大区别。在 Python 2 中，range()函数用来创建一个整数列表，返回值为列表，举例如下。

```
[root@CentOS-Python ～]# python2
Python 2.7.16 (default, Nov 17 2019, 00:07:27)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>a = range(10)
>>>type(a)
<type 'list'>  #Python 2 中的 range()函数返回值的类型为列表
>>> print a
[0， 1， 2， 3， 4， 5， 6， 7， 8， 9]
>>> range(1, 15)
[1， 2， 3， 4， 5， 6， 7， 8， 9， 10， 11， 12， 13， 14]
>>> range(1, 20, 2)
[1，3， 5， 7， 9， 11， 13， 15， 17， 19]
>>>
```

- Python 2 中的 range()创建的整数列表从 0 开始，因此 range(10)返回的是一个包含整数 0～9 的列表，**并不包含 10。**

- 也可以在 range()中指定起始数和结尾数，返回的整数列表的最后一个元素为指定的结尾数减 1，比如 range(1, 15)将返回一个包含整数 1～14 的列表，14 由结尾数 15 减 1 得来。

- range()还可以通过指定步长来得到我们想要的整数，比如你只想选取 1～19 中所有的单数，那么就可以使用 range (1, 20, 2)来实现（这里的 2 即步长）。

这种返回列表的 range()函数有一个缺点是会占用内存，列表所含元素数量不多时对主机的性能影响不大，但是当使用 range(10000000000000)来建立诸如这样巨大的列表时所占用的内存就非常“恐怖”了。因此在 Python 3 中，range()函数的返回值被改为了 range，这是一种可以被迭代的对象，这样改变的目的就是节省内存。

```
[root@CentOS-Python ～]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = range(10)
>>> type(a)
<class 'range'> #Python 3 中的 range()函数的返回值不再是列表，而是 range 迭代值。
>>> print (a)
range(0, 10)  #不再像 Python 2 那样返回整数列表[0,1,2,3,4,5,6,7,8,9]
>>> range (1,15)
range(1, 15) #同上
>>> range (1,20,2)
range(1, 20, 2) #同上
>>>
```

如果要使 Python 3 的 range()也返回列表，则需要对其使用 list()函数，举例如下。

```
>>> a = list(range(10))
>>> print (a)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1,15))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> list(range(1,20,2))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>>
```

◎　append()

append()用来向列表添加元素，举例如下。

```
>>> interfaces = []
>>> interfaces.append('Gi1/1')
>>> print (interfaces)
['Gi1/1']
>>> interfaces.append('Gi1/2')
>>> print (interfaces)
['Gi1/1',  'Gi1/2']
>>>
```

首先我们建立一个空列表（以[]表示），并把它赋值给 interfaces 变量，然后使用 append()方法将端口 Gi1/1 加入该列表，随后调用 append()将 Gi1/2 加入该列表，现在列表 interfaces 中就有 Gi1/1 和 Gi1/2 两个元素了。

◎　len()

列表的 len()方法和字符串的 len()方法大同小异，前者用来统计列表中有多少个元素，后者用来统计字符串内容的长度，其返回值也依然为整数，举例如下。

```
>>> len(interfaces)
2
>>>
>>> cisco_switch_models = ['2960', '3560', '3750', '3850', '4500', '6500', '7600',
'9300']
>>> len(cisco_switch_models)
8
>>>
```

◎　count()

与字符串一样，列表也有 count()方法，列表的 count()方法用来找出指定的元素在列表中有多少个，返回值为整数，举例如下。

```
>>> vendors = ['Cisco', 'Juniper', 'HPE', 'Aruba', 'Arista', 'Huawei', 'Cisco',
```

```
    'Palo Alto', 'CheckPoint', 'Cisco', 'H3C', 'Fortinet']
    >>> vendors.count('Cisco')
    3
    >>>
```

◎ insert()

列表是有序的集合，前面讲到的 append()方法是将新的元素添加到列表的最后面，如果我们想自己控制新元素在列表中的位置，则要用 insert()方法。举例如下。

```
    >>> ospf_configuration = ['router ospf 100\n', 'network 0.0.0.0 255.255.255.255
area 0\n']
    >>> ospf_configuration.insert(0, 'configure terminal\n')
    >>> print (ospf_configuration)
    ['configure terminal\n',  'router ospf 100\n', 'network 0.0.0.0 255.255.255.255
area 0\n']
    >>>
```

首先我们创建一个名为 ospf_configuration 的变量，将配置 OSPF 的命令写在一个列表中赋值给该变量。随后发现列表漏了 configure terminal 命令，该命令要写在列表的最前面，这时我们可以用 insert(0, 'configure terminal\n')将该命令加在列表的最前面（记住列表的索引号是从 0 开始的）。

如果这时我们还想给该 OSPF 路由器配置一个 router-id，比如把 router-id 这条命令写在 router ospf 100 的后面，可以再次使用 insert()，举例如下。

```
    >>> ospf_configuration.insert(2, 'router-id 1.1.1.1\n')
    >>> print (ospf_configuration)
    ['configure terminal\n', 'router ospf 100\n', 'router-id 1.1.1.1\n', 'network
0.0.0.0 255.255.255.255 area 0\n']
    >>>
```

◎ pop()

pop()用来移除列表中的元素，如果不指定索引号，则 pop()默认将拿掉排在列表末尾的元素；如果指定了索引号，则可以精确移除想要移除的元素，举例如下。

```
>>> cisco_switch_models = ['2960', '3560', '3750', '3850', '4500', '6500', '7600',
'9300']
>>> cisco_switch_models.pop()
'9300'  #排在末尾的 9300 被拿掉
>>> print cisco_switch_models
['2960',  '3560',  '3750',  '3850',  '4500',  '6500',  '7600']
>>> cisco_switch_models.pop(1)
'3560' #使用索引号 1 将排在第 2 位的 3560 拿掉
>>> print (cisco_switch_models)
['2960',  '3750',  '3850',  '4500',  '6500',  '7600']
>>>
```

◎　index()

看了 pop()的用法后，你也许会问：在拥有很多元素的列表中，怎么知道想要移除的元素的索引号是多少？这时就需要用 index()，比如想从 cisco_switch_models 列表中移除元素 4500，可以按照如下操作。

```
>>> cisco_switch_models = ['2960', '3560', '3750', '3850', '4500', '6500', '7600',
'9300']
>>> cisco_switch_models.index('4500')
4
>>> cisco_switch_models.pop(4)
'4500'
>>> print (cisco_switch_models)
['2960',  '3560',  '3750',  '3850',  '6500',  '7600',  '9300']
>>>
```

先通过 index()找出 4500 的索引号为 4，然后配合 pop(4)将它从列表中移除。

2.4.4　字典

在 Python 里，字典（Dictionary）是若干组无序的键值对（Key-Value pair）的集合，用大括号{}表示，每一组键值对都用逗号隔开，举例如下。

```
>>> dict = {'Vendor':'Cisco', 'Model':'WS-C3750E-48PD-S', 'Ports':48,
'IOS':'12.2(55)SE12', 'CPU':36.3}
```

这里我们创建了一个变量名为 dict 的字典，该字典有 5 组键值对，分别如下。

```
'Vendor':'Cisco'
'Model':'WS-C3750E-48PD-S'
'Ports':48
'IOS':'12.2(55)SE12'
'CPU':36.3
```

- 键值对里的键（Key）和值（Value）以冒号:隔开，冒号的左边为键，右边为值。
- 键的数据类型可为字符串、常数、浮点数或者元组，对网络工程师来说，最常用的肯定是字符串，比如'Vendor'、'Model'等。
- 值可为任意的数据类型，比如这里的'Cisco'为字符串，48 为整数，36.3 为浮点数。

与列表不同，字典是无序的，举例如下。

```
>>> a = [1, 2, 3, 'a', 'b', 'c']
>>> print (a)
[1, 2, 3, 'a', 'b', 'c']
>>>
>>> dict = {'Vendor':'Cisco', 'Model':'WS-C3750E-48PD-S', 'Ports':48,
'IOS':'12.2(55)SE12', 'CPU':36.3}
>>> print (dict)
{'IOS': '12.2(55)SE12', 'CPU':36.3, 'Model': 'WS-C3750E-48PD-S', 'Vendor':
'Cisco', 'Ports': 48}
```

```
>>>
```

这里我们创建一个内容为[1，2，3，'a'，'b'，'c']的列表 a， 将它打印出来后，列表的元素的位置没有发生任何变化，因为列表是有序的。但是如果我们将刚才的字典 dict 打印出来，你会发现字典里键值对的顺序已经彻底被打乱了，没有规律可循，正因为字典是无序的，我们自然也不能像列表那样使用索引来查找字典中某个键对应的值。

在字典里，查找某个值的格式为'字典名[键名]'，举例如下。

```
>>> dict = {'Vendor':'Cisco', 'Model':'WS-C3750E-48PD-S', 'Ports':48,
'IOS':'12.2(55)SE12', 'CPU':36.3}
>>> print (dict['Vendor'])
Cisco
>>> print (dict['CPU'])
36.3
>>> print (dict['Ports'])
48
>>>
```

如果要在字典里新添加一组键值对，则格式为'字典名[新键名]' = '新值'，举例如下。

```
>>> dict['Number of devices']=100
>>> print (dict)
{'Vendor': 'Cisco', 'Number of devices': 100, 'IOS': '12.2(55)SE12', 'CPU':
36.3, 'Model': 'WS-C3750E-48PD-S', 'Ports': 48}
>>>
```

如果要更改字典里某个已有键对应的值，则格式为'字典名[键名]' = '新值'，举例如下。

```
>>> dict['Model'] = 'WS-C2960X-24PS-L'
>>> dict['Ports'] = '24'
>>> print (dict)
{'IOS': '12.2(55)SE12', 'Model': 'WS-C2960X-24PS-L', 'Vendor': 'Cisco',
'Ports': '24', 'CPU': 36.3}
>>>
```

如果要删除字典里的某组键值对，则命令为 del '字典名[键名]'，举例如下。

```
>>> del dict['Number of devices']
>>> print (dict)
{'Vendor': 'Cisco', 'IOS': '12.2(55)SE12', 'CPU': 36.3, 'Model':
'WS-C3750E-48PD-S', 'Ports': 48}
>>>
```

下面介绍与字典相关的函数和方法。

◎ len()

len()用来统计字典里有多少组键值对。**len()的返回值是整数，**举例如下。

```
>>> print (dict)
{'Vendor': 'Cisco', 'IOS': '12.2(55)SE12', 'CPU': 36.3, 'Model':
'WS-C3750E-48PD-S', 'Ports': 48}
>>> len(dict)
5
>>>
```

◎ keys()

keys()用来返回一个字典里所有的键。**keys()在 Python 2 中的返回值为列表；**在 Python 3 中的返回值是可迭代的对象，需要使用 list()将它转换为列表，了解即可。举例如下。

```
>>> print (dict)
{'Vendor': 'Cisco', 'IOS': '12.2(55)SE12', 'CPU': 36.3, 'Model':
'WS-C3750E-48PD-S', 'Ports': 48}
>>> print (dict.keys())
['Vendor', 'IOS', 'CPU', 'Model', 'Ports']
>>>
```

◎ values()

values()用来返回一个字典里所有的值。**注意，values()在 Python 2 中的返回值为列表，在 Python 3 中的返回是可迭代的对象**，在有必要的情况下需要使用 list()将它转换为列表，举例如下。

```
[root@CentOS-Python ~]# python2
Python 2.7.16 (default, Nov 17 2019, 00:07:27)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> dict = {'Vendor':'Cisco', 'Model':'WS-C3750E-48PD-S', 'Ports':48,
'IOS':'12.2(55)SE12', 'CPU':36.3}
>>> print (dict.values())
['Cisco', '12.2(55)SE12', 36.3, 'WS-C3750E-48PD-S', 48] #Python 2 里返回列表
>>>
```

```
[root@CentOS-Python ~]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> dict = {'Vendor':'Cisco', 'Model':'WS-C3750E-48PD-S', 'Ports':48,
'IOS':'12.2(55)SE12', 'CPU':36.3}
>>> print (dict.values())
dict_values(['Cisco', 'WS-C3750E-48PD-S', 48, '12.2(55)SE12', 36.3]) #Python 3
里返回可迭代对象
>>>
```

◎ pop()

前面讲到，要删除字典中某组键值可以用命令 del '字典名[键名]'。另外，我们可以使用 pop()来达到同样的目的，与列表的 pop()不同，字典的 pop()不能导入索引号，需要导入的是键名，**而且字典的 pop()的返回值不是列表，而是键名对应的值（比如下面的 48）**，举例如下。

```
>>> print (dict)
{'Vendor': 'Cisco', 'IOS': '12.2(55)SE12', 'CPU': 36.3, 'Model':
'WS-C3750E-48PD-S', 'Ports': 48}
>>> dict.pop('Ports')
48
>>> print (dict)
{'Vendor': 'Cisco', 'IOS': '12.2(55)SE12', 'CPU': 36.3, 'Model':
'WS-C3750E-48PD-S'}
>>>
```

◎ get()

前面讲到，我们可以使用 values()方法得到一个字典里所有的值，除此之外，还可以使用 get()来返回字典里具体键名对应的值，**get()的返回值是所导入的键名对应的值**，举例如下。

```
>>> print (dict)
{'Vendor': 'Cisco', 'IOS': '12.2(55)SE12', 'CPU': 36.3, 'Model':
```

```
'WS-C3750E-48PD-S'}
    >>> dict.get('Vendor')
    'Cisco'
    >>> dict.get('CPU')
    36.3
    >>>
```

2.4.5 布尔类型

布尔类型（Boolean）用来判断条件是否成立，布尔值只有两种：True 和 False，如果条件成立，则返回 True；如果条件不成立，则返回 False。**两种布尔值的首字母（T 和 F）必须为大写，true 和 false 都不是有效的布尔值。**布尔类型在判断语句中常用，Python 的判断语句将在进阶内容中详细讲解。

1. 比较运算符

既然布尔类型用来判断条件是否成立，那就不得不提一下 Python 中的**比较运算符（Comparison Operators）**。比较运算符包括等于号==、不等于号!=、大于号>、小于号<、大于等于号>=、小于等于号<=。**比较运算符**和+、-、*、//、**这些做加减乘除求幂的**算术运算符**（在 2.3.2 节中提到过）**最大的区别是：前者用来判断符号左右两边的变量和数据是否满足运算符本身的条件，并且返回值是布尔值，后者则单纯做加减乘除等运算，返回值是整数或浮点数。**

在编辑器模式下，使用比较运算符后可以马上看到返回的布尔值 True 或者 False，如果是脚本模式，则需要配合 print 命令才能看到，举例如下。

```
    >>> a = 100
    >>> a == 100
    True
    >>> a == 1000
    False
```

```
>>> a != 1000
True
>>> a != 100
False
>>> a > 99
True
>>> a > 101
False
>>> a < 101
True
>>> a < 99
False
>>> a >= 99
True
>>> a >= 101
False
>>> a <= 100
True
>>> a <= 99
False
>>>
```

2. 逻辑运算符

除了比较运算符，使用**逻辑运算符（Logical Operators）**也能返回布尔值。逻辑运算符有 3 种：与（and）、或（or）、非（not）。学过离散数学的读者对与、或、非的逻辑运算不会感到陌生，在逻辑运算中使用的真值表（Truth Table）如下。

真值表（Truth Table）				
P	Q	P 与 Q(and)	P 或 Q(or)	非 P(not)
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True

False	False	False	False	True

逻辑运算符在 Python 中的使用举例如下。

```
>>> A = True
>>> B = True
>>> A and B
True
>>> A or B
True
>>> not A
False
>>>
>>>
>>> A = False
>>> B = True
>>> A and B
False
>>> A or B
True
>>> not A
True
>>>
>>> A = False
>>> B = False
>>> A and B
False
>>> A or B
False
>>> not A
True
>>>
```

2.4.6 集合、元组、空值

作为同样需要网络工程师掌握的 Python 数据类型，**集合**（Set）、**元组**（Tuple）、**空值**

（None）相对来说使用频率不如字符串、整数、浮点数、列表、字典及布尔类型那么高，这里进行简单介绍。

1. 集合

- 集合是一种特殊的列表，里面**没有重复的元素**，因为每个元素在集合中都只有一个，因此集合没有 count()方法。
- 集合可以通过大括号{}（与字典一样，但是集合没有键值对）或者函数 set()创建。

```
>>> interfaces = {‘Fa0/0’, 'Fa0/1', 'Fa0/2'}
>>> type(interfaces)
<class 'set'>
>>> vendors = set(['Cisco', 'Juniper', 'Arista', 'Cisco'])
>>> type(vendors)
<class 'set'>
>>> print (vendors)
{'Cisco',  'Arista',  'Juniper'}
>>>
```

vendors 列表中有两个重复的元素，即'Cisco'，在用 set()函数将它转换成集合后，多余的一个'Cisco'被去掉，只保留了一个。

- 集合是无序的，不能像列表那样使用索引号，也不具备 index()函数。

```
>>> vendors[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
>>>
>>> vendors.index('Cisco')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'set' object has no attribute 'index'
>>>
```

下面介绍与集合有关的方法和函数。

◎　add()

add()用来向一组集合添加新元素，**其返回值依然是集合**，举例如下。

```
>>> vendors.add('Huawei')
>>> vendors
{'Huawei', 'Cisco', 'Arista', 'Juniper'}
>>>
```

◎　remove()

remove()用来删除一组集合中已有的元素，**其返回值依然是集合**，举例如下。

```
>>> vendors.remove('Arista')
>>> vendors
{'Huawei', 'Cisco', 'Juniper'}
>>>
```

2. 元组

- 与集合一样，元组也是一种特殊的列表。它与列表最大的区别是：可以任意对列表中的元素进行增添、删除、修改，而元组则不可以。一旦创建元组，将无法对其做任何形式的更改，所以元组没有 append()、insert()、pop()、add()和 remove()，只保留了 index()和 count()两种方法。
- 元组可以通过小括号()创建，也可以使用函数 tuple()创建。
- 与列表一样，元组是有序的，可以对元素进行索引。

```
>>> vendors = ('Cisco', 'Juniper', 'Arista')
>>> print (vendors)
('Cisco', 'Juniper', 'Arista')
>>> print (vendors[1])
Juniper
>>> vendors[2] = 'Huawei'
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
```

```
    TypeError: 'tuple' object does not support item assignment  #不能对元组做任何形式的修改
    >>>
```

下面介绍与元组有关的方法和函数。

◎ index()

元组的 index()与列表用法相同，都用来查询指定元素的索引号。**index()的返回值为整数**，举例如下。

```
    >>> vendors
    ('Cisco', 'Juniper', 'Arista')
    >>> vendors.index('Cisco')
    0
    >>>
```

◎ count()

元组的 count()与列表用法相同，都用来查询指定元素在元组中的数量。**count()的返回值为整数**，举例如下。

```
    >>> vendors = ('Cisco', 'Juniper', 'Arista', 'Cisco')
    >>> vendors.count('Cisco')
    2
    >>> vendors.count('Juniper')
    1
    >>>
```

3. 空值

空值是比较特殊的数据类型，它没有自带的函数和方法，也无法做任何算术和逻辑运算，但是可以把它赋值给一个变量，举例如下。

```
    >>> type(None)
    <type 'NoneType'>
    >>> None == 100
```

```
False
>>> a = None
>>> print (a)
None
```

空值（None）较常用在判断语句和正则表达式中。对网络工程师来说，日常工作中需要经常使用显示命令（show 或者 display）来对网络设备进行排错或者查询网络信息，通常这类显示命令都会给出很多回显内容，而大多时候我们只需要关注其中的一两项参数即可。如果用 Python 来实现网络运维自动化，则需要使用正则表达式来告诉 Python 应该抓取哪一个“关键词”（即我们想要的参数），而空值则可以用来判断“关键词”抓取得是否成功。关于判断语句和正则表达式的用法将会在第 3 章中详细介绍。

第 3 章

Python 进阶语法

前两章分别介绍了 Python 在 Windows 和 Linux 里的安装和使用方法，详细讲解了网络工程师需要掌握的 Python 数据类型及每种数据类型自带的函数和方法的用法。本章将讲解 Python 中的条件（判断）语句、循环语句、文本文件的读写、自定义函数、模块、正则表达式及异常处理等网络工程师需要掌握的 Python 进阶语法知识。

3.1 条件（判断）语句

在 Python 中，条件语句（Conditional Statements）又叫作判断语句，判断语句由 if、elif 和 else 3 种语句组成，其中 if 为强制语句，可以独立使用，elif 和 else 为可选语句，并且不能独立使用。判断语句配合布尔值，通过判断一条或多条语句的条件是否成立（True 或者 False），从而决定下一步的动作，如果判断条件成立（True），则执行 if 或 elif 语句下的代码；如果判断条件不成立（False），则执行 else 语句下的代码；如果没有 else 语句，则不做任何事情。

布尔值是判断语句不可或缺的部分，在基础语法中讲到的比较运算符、逻辑运算符，以及字符串自带的 startswith()、endswith()、isdigit()、isalpha()等方法，还有下面将会讲到的成员运算符等都会返回布尔值，下面就举例讲解它们各自在 Python 判断语句中的应用场景。

3.1.1　通过比较运算符做判断

在讲布尔类型时，我们已经提到与布尔值 True 和 False 息息相关的各种比较运算符，包括等于号==、不等于号!=、大于号>、小于号<、大于等于号>=和小于等于号<=，因为使用比较运算符后会直接返回布尔值，所以比较运算符在判断语句中会经常被用到，举例如下。

首先我们用脚本模式写一段代码。

```
[root@CentOS-Python ~]# cat lab.py

# coding=utf-8
final_score = input('请输入你的 CCNA 考试分数:')
if int(final_score) > 811:
    print ('恭喜你通过考试。')
elif int(final_score) == 811:
    print ('恭喜你压线通过考试。')
else:
    print ('成绩不及格。')
```

这段代码用来让用户输入自己的 CCNA 考试成绩并做判断，假设及格线为 811 分，如果用户所得分数大于 811 分，则打印“恭喜你通过考试。”，如果分数刚刚等于 811 分，则打印“恭喜你压线通过考试。”，如果低于 811 分，则打印“成绩不及格。”，这段代码需要注意以下几点。

- 写在 if、elif 和 else 下的代码都有做**代码缩进（Indentation）**，也就是 print()函数的前面保留了 4 个空格。不同于 C/C++/Java 等语言，**Python 要求严格的代码缩进**，目的是让代码工整并且具有可读性，方便阅读和修改。缩进不一定必须为 4 个空格，两个空格甚至 8 个空格都是允许的，目前最常见的是 4 个空格的缩进。
- if、elif 和 else 语句的结尾必须接冒号:，这点需要注意。

- 使用 input()函数让用户输入自己的分数，并把它赋值给 final_score 变量。在第 2 章里讲过，input()函数的返回值是字符串，因为要与 811 这个整数做比较，所以需要通过 int()函数先将 final_score 从字符串转换为整数。
- 与 if 和 elif 语句不同，else 后面不需要再给任何判断条件。

运行这段脚本看效果。

```
[root@CentOS-Python ~]# python3.8 lab.py
请输入你的 CCNA 考试分数:1000
恭喜你通过考试。
[root@CentOS-Python ~]# python3.8 lab.py
请输入你的 CCNA 考试分数:811
恭喜你压线通过考试。
[root@CentOS-Python ~]# python3.8 lab.py
请输入你的 CCNA 考试分数:700
成绩不及格。
[root@CentOS-Python ~]#
```

3.1.2 通过字符串方法+逻辑运算符做判断

当使用 input()函数让用户输入内容时，你无法保证用户输入的内容合乎规范。比如你给用户 6 个选项，每个选项分别对应一个动态路由协议的名称（选项 1：RIP；选项 2：IGRP；选项 3：EIGRP；选项 4：OSPF；选项 5：ISIS；选项 6：BGP），提示用户输入路由协议的选项号码来查询该路由协议的类型，然后让 Python 根据用户输入的选项号码告诉用户该路由协议属于链路状态路由协议、距离矢量路由协议，还是路径适量路由协议。

这里你无法保证用户输入的肯定是整数，即使用户输入的是整数，也无法保证输入的是 1～6 的数字。因为 input()函数的返回值是字符串，所以可以首先使用字符串的 isdigit()函数来判断用户输入的内容是否为整数，这是判断条件之一。然后通过 int()将该字符串数字转换成整数，继续判断该整数是否介于 1 和 6 之间（包含 1 和 6），这是判断条件之二。

再将这两个判断条件通过逻辑运算符 and 来判断它俩是否同时成立。如果成立，则返回相应的答案；如果不成立，则提示用户输入的内容不符合规范并终止程序。代码如下。

```
[root@CentOS-Python ~]# cat lab.py

# coding=utf-8

print ('''请根据对应的号码选择一个路由协议：
1. RIP
2. IGRP
3. EIGRP
4. OSPF
5. ISIS
6. BGP ''')

option = input('请输入你的选项(数字 1-6)： ')
if option.isdigit() and 1 <= int(option) <= 6:
  if option == '1' or option == '2' or option == '3':
     print ('该路由协议属于距离矢量路由协议。')
  elif option == '4' or option == '5':
     print ('该路由协议属于链路状态路由协议。')
  else:
     print ('该路由协议属于路径矢量路由协议。')
else:
  print ('选项无效，程序终止。')
```

这里我们用到了**嵌套 if 语句**。在某一个条件成立（判定为 True）后，如果还需要检查其他子条件，就可以用嵌套 if 语句来完成。在嵌套 if 语句中，一组 if、elif 和 else 可以构造在另一组 if、elif 和 else 中，不过需要注意缩进。

运行代码，测试效果如下。

```
[root@CentOS-Python ~]# python3.8 lab.py

请根据对应的号码选择一个路由协议：
1. RIP
```

```
2. IGRP
3. EIGRP
4. OSPF
5. ISIS
6. BGP
请输入你的选项(数字 1-6)：1
该路由协议属于距离矢量路由协议。

[root@CentOS-Python ~]# python lab.py
请根据对应的号码选择一个路由协议：
1. RIP
2. IGRP
3. EIGRP
4. OSPF
5. ISIS
6. BGP
请输入你的选项(数字 1-6)：4
该路由协议属于链路状态路由协议。

[root@CentOS-Python ~]# python lab.py
请根据对应的号码选择一个路由协议：
1. RIP
2. IGRP
3. EIGRP
4. OSPF
5. ISIS
6. BGP
请输入你的选项(数字 1-6)：6
该路由协议属于路径矢量路由协议。

[root@CentOS-Python ~]# python lab.py
请根据对应的号码选择一个路由协议：
1. RIP
2. IGRP
3. EIGRP
4. OSPF
```

```
5. ISIS
6. BGP
请输入你的选项(数字 1-6): abc
选项无效，程序终止。

[root@CentOS-Python ~]# python lab.py
请根据对应的号码选择一个路由协议:
1. RIP
2. IGRP
3. EIGRP
4. OSPF
5. ISIS
6. BGP
请输入你的选项(数字 1-6): 8
选项无效，程序终止。
[root@CentOS-Python ~]#
```

3.1.3　通过成员运算符做判断

成员运算符用于判断是否可以在给定的一组字符串、列表、字典和元组中找到一个给定的值或变量，如果能找到，则返回布尔值 True；如果找不到，则返回布尔值 False。**成员运算符有两种：in 和 not in**，举例如下。

```
>>> netdevops = '网络工程师需要学习 Python 吗? '
>>> 'Python' in netdevops
True
>>> 'Java' in netdevops
False
>>> 'C++' not in netdevops
True
>>>
>>> interfaces = ['Gi1/1', 'Gi1/2', 'Gi1/3', 'Gi1/4', 'Gi1/5']
>>> 'Gi1/1' in interfaces
```

```
True
>>> 'Gi1/10' in interfaces
False
>>> 'Gi1/3' not in interfaces
False
>>>
```

依靠成员运算符，我们还可以将类似上一节的脚本简化。上一节的脚本给出的选项只有 6 种，我们尚且能够使用“if option == '1' or option == '2' or option == '3':”此类的方法来列举所需的选项，但是如果所需选项超过 20 个甚至上百个，那么再使用这种方法岂不是太笨了？这时可以使用第 2 章讲的 range()函数配合 list()函数创造一个整数列表，然后配合成员运算符来判断用户输入的选项号码是否存在于该整数列表中。

按照这种思路，我们将上一节的脚本做如下简化。

```
[root@CentOS-Python ~]# cat lab.py
# coding=utf-8

print ('''请根据对应的号码选择一个路由协议:
1. RIP
2. IGRP
3. EIGRP
4. OSPF
5. ISIS
6. BGP ''')

option = input('请输入你的选项(数字 1-6): ')
if option.isdigit() and int(option) in list(range(1, 7)):
   if int(option) in list(range(1, 4)):
      print ('该路由协议属于距离矢量路由协议。')
   elif int(option) in list(range(4, 6)):
      print ('该路由协议属于链路状态路由协议。')
   else:
      print ('该路由协议属于路径矢量路由协议。')
```

```
else:
    print ('选项无效，程序终止。')

[root@CentOS-Python ~]#
```

之前依靠比较运算符+逻辑运算符做判断的 if option == '1' or option == '2' or option == '3':方法已经被成员运算符+range()函数简化为 if int(option) in list(range(1, 4)):了。同理，判断选项在整数 1～6 之间也已经通过成员运算符配合 range()和 list()函数写成 int(option) inrange(1, 7)了。需要注意的是，由于 input()函数返回的是字符串，因此需要把变量 option 先通过 int()函数转换为整数才能使用成员运算符 in 来判断它是否存在于 range()和 list()函数所创建的整数列表中。

3.2　循环语句

Python 中最常用的循环语句（Looping Statements）**有两种：while 和 for**。除此之外，还有文件迭代器（File Iterator）、列表解析式（List Comprehension）等循环工具，不过对于网络工程师来说，用得最多的还是 while 和 for，因此本节将只讲解这两个循环语句。

3.2.1　while 语句

在 Python 中，while 语句用于循环执行一段程序，它和 if 语句一样，两者都离不开判断语句和缩进。每当写在 while 语句下的程序被执行一次，程序就会自动回到“顶上”（也就是 while 语句的开头部分），根据 while 后的判断语句的返回值来决定是否要再次执行该程序，如果判断语句的返回值为 True，则继续执行该程序，一旦判断语句的返回值为 False，则该 while 循环随即终止，如此反复。如果需要中途强行中止 while 循环，则需要使用 break 语句。下面通过 3 个例子帮助大家理解。

◎　**例 1**

```
>>> a = 1
>>> b = 10
>>> while a < b:
...    print (a)
...    a += 1
...
1
2
3
4
5
6
7
8
9
>>>
```

上面代码中，我们用 while 循环来判断变量 a 是否小于 b，如果判断结果为 True，则打印出变量 a 的值，并且每次让 a 的值加 1，如此反复循环，直到第 10 次执行该 while 循环，a = 10，因为 a < b 不再成立（10<10 不成立），程序随即终止。

◎ **例 2**

```
>>> vendors = ['Cisco', 'Huawei', 'Juniper', 'Arista', 'HPE', 'Extreme']
>>> while len(vendors) > 0:
...    vendors.pop()
...    print (vendors)
...
'Extreme'
['Cisco', 'Huawei', 'Juniper', 'Arista', 'HPE']
'HPE'
['Cisco', 'Huawei', 'Juniper', 'Arista']
'Arista'
['Cisco', 'Huawei', 'Juniper']
'Juniper'
['Cisco', 'Huawei']
```

```
'Huawei'
['Cisco']
'Cisco'
[]
>>>
```

上面代码中，我们用 while 循环配合 len()函数来判断列表 vendors 的长度是否大于 0，如果判断结果为 True，则用 pop()方法从列表中删掉一个元素，并且随即打印列表里剩余的元素。最终当列表中所有的元素都被移除时，列表的长度为 0，while len(vendors)>0:的返回值为 False，该 while 循环也就随即终止。

◎　**例 3**

在 3.1.2 节和 3.1.3 节的案例代码中，一旦用户输入的选项不符合规范，程序就会立即中止，用户必须再次手动运行一次脚本重新输入选项，这样显得很笨拙。借助 while 循环，可以不断地重复执行 input()函数，直到用户输入正确的选项号码。优化后的脚本代码如下。

```
# coding=utf-8

print ('''请根据对应的号码选择一个路由协议:
1. RIP
2. IGRP
3. EIGRP
4. OSPF
5. ISIS
6. BGP ''')

while True:
    option = input('请输入你的选项(数字 1-6): ')
    if option.isdigit() and int(option) in list(range(1, 7)):
     if int(option) in list(range(1, 4)):
            print ('该路由协议属于距离矢量路由协议。')
        elif int(option) in list(range(5, 7)):
            print ('该路由协议属于链路状态路由协议。')
        else:
            print ('该路由协议属于路径矢量路由协议。')
```

```
        break
       else:
        print ('选项无效，请再次输入。')

[root@CentOS-Python ~]#
```

这里我们使用了 **while True**。**while True 是一种很常见的 while 循环的用法，因为这里的判定条件的结果已经手动指定了是 True，意味着判定条件将永久成立，也就意味着 while 下面的程序将会被无数次重复执行**，从而引起“无限循环”（Indefinite Loop）的问题。为了避免无限循环，我们必须在程序代码中使用 break 语句来终止 while 循环，注意 break 在上面代码里的位置，带着这个问题去思考，你会更加明白缩进在 Python 中的重要性。

执行代码看效果。

```
[root@CentOS-Python ~]#python lab.py
请根据对应的号码选择一个路由协议：
1. RIP
2. IGRP
3. EIGRP
4. OSPF
5. ISIS
6. BGP
请输入你的选项(数字 1-6)： 7
选项无效，请再次输入。
请输入你的选项(数字 1-6)： a
选项无效，请再次输入。
请输入你的选项(数字 1-6)： ..!!!
选项无效，请再次输入。
请输入你的选项(数字 1-6)： 134ui134lkadjl
选项无效，请再次输入。
请输入你的选项(数字 1-6)： 1
该路由协议属于距离矢量路由协议。

[root@CentOS-Python ~]#
```

3.2.2　for 语句

同为循环语句，for 语句的循环机制和 while 语句完全不同：while 语句需要配合判断语句来决定什么时候开始循环和中止循环，而 for 语句则用来遍历一组可迭代的序列，可迭代的序列包括字符串、列表、元组等。在将这些序列中的元素遍历完后，for 语句的循环也随即终止。for 语句的基本语法格式如下。

```
for item in sequence:
    statements
```

这里的 sequence 为可迭代的序列（如字符串、列表、元组），而 item 可以理解为该序列里的每个元素（item 名称可以任意选取），statements 则是循环体（将要循环的程序部分）。

举例如下。

◎　**例 1**

```
>>> for letter in 'Python':
...  print (letter)
...
P
y
t
h
o
n
>>>
```

我们用 letter 作为 for 语句中的 item 来遍历字符串'Python'，并将该字符串中的元素依次全部打印出来，得到 P、y、t、h、o、n。

◎　**例 2**

```
>>> sum = 0
>>> for number in range(1, 6):
```

```
...     sum = sum + number
...     print (sum)
...
1
3
6
10
15
>>>
```

我们将 0 赋值给变量 sum，然后用 number 作为 for 语句中的 item 来遍历 range(1,6)，返回 1～5 的 5 个整数，将这 5 个整数依次与 sum 累加，每累加一次用 print()函数打印出结果，最后得到 1、3、6、10、15。

◎ **例 3**

```
>>> routing_protocols = ['RIP', 'IGRP', 'EIGRP', 'OSPF', 'ISIS', 'BGP']
>>> link_state_protocols = ['OSPF', 'ISIS']
>>> for protocols in routing_protocols:
...     if protocols not in link_state_protocols:
...         print (protocols + '不属于链路状态路由协议：' )
RIP 不属于链路状态路由协议：
IGRP 不属于链路状态路由协议：
EIGRP 不属于链路状态路由协议：
BGP 不属于链路状态路由协议：
>>>
```

我们分别创建两个列表：routing_protocols 和 link_state_protocols，列表中的元素为相对应的路由协议和链路状态路由协议。首先用 protocols 作为 for 语句中的 item 来遍历第一个列表 routing_protocols，然后使用 if 语句来判断哪些 protocols 不属于第二个列表 link_state_protocols 中的元素，并将它们打印出来。

正如前面讲到的，上述三个例子中写在 for 后面的 letter、number 和 protocols 代表将要遍历的可迭代序列里的每一个元素（即 item 名称），**它们的名称可以由用户随意制定**，比如在例 1 中，我们把 letter 换成 a 也没问题。

```
>>> for a in 'Python':
...  print (a)
...
P
y
t
h
o
n
```

通常建议取便于理解的 item 名称，像 a 这类的 item 名称在做实验或者练习时可以偷懒使用，在实际工作代码中用这种毫无意义的 item 名称肯定是会被人诟病的。

3.3　文本文件的读写

在日常网络运维中，网络工程师免不了要和大量的文本文件打交道，比如用来批量配置网络设备的命令模板文件，存放所有网络设备 IP 地址的文件，以及备份网络设备 show run 输出结果之类的配置备份文件。正因如此，知道如何使用 Python 来访问和管理文本文件是学习网络运维自动化技术的网络工程师必须掌握的一项 Python 知识点。

3.3.1　open()函数及其模式

在 Python 中，我们可以通过 open()函数来访问和管理文本文件，open()函数用来打开一个文本文件并创建一个文件对象（File Object），通过文件对象自带的多种函数和方法，可以对文本文件执行一系列访问和管理操作。在讲解这些函数和方法之前，首先创建一个名为 test.txt 的测试文本文件，该文件包含 5 个网络设备厂商的名字，内容如下。

```
[root@CentOS-Python ~]# cat test.txt
Cisco
Juniper
```

```
Arista
H3C
Huawei
[root@CentOS-Python ~]#
```

然后用 open()函数访问该文件。

```
>>> file = open('test.txt', 'r')
```

我们通过 open()函数的 r 模式（只读）访问 test.txt 文件，并返回一个文件对象，再将该文件对象赋值给 file 变量。**r(reading)是默认访问模式**，除此之外，open()函数还有很多其他文件访问模式。这里只介绍网络工程师最常用的几种模式，如下表所示。

模　式	作　用
r	以只读方式打开文件，r 模式只能打开已存在的文件。如果文件不存在，则会报错
w	打开文件并只用于写入。如果文件已经存在，则原有内容将被删除覆盖；如果文件不存在，则创建新文件
a	以追加方式打开文件。如果文件已经存在，则原有内容不会被删除覆盖，新内容将添加在原有内容后面；如果文件不存在，则创建新文件
r+	以读写方式打开文件，r+模式只能打开已存在的文件。如果文件不存在，则会报错
w+	以读写方式打开文件。如果文件已经存在，则原有内容将被删除覆盖；如果文件不存在，则创建新文件
a+	以读写方式打开文件。如果文件已经存在，则原有内容不会被删除覆盖，新内容将添加在原有内容后面；如果文件不存在，则创建新文件

open()函数的上述 6 种模式必须熟练掌握，关于它们的具体使用将在下一节中举例讲解。

3.3.2　文件读取

在使用 open()函数创建文件对象之后，我们并不能马上读取文件里的内容。如下所示，

在创建了文件对象并将它赋值给 file 变量后，如果用 print()函数将 file 变量打印出来，则只会得到文件名称、open()函数的访问模式及该文件对象在内存中的位置（0x7fa194215660）等信息。

```
>>> file = open('test.txt', 'r')
>>> print (file)
<open file 'test.txt',  mode 'r' at 0x7fa194215660>
>>>
```

要想读取文件里的具体内容，我们还需要用 read()、readline()或者 readlines() 3 种方法中的一种。因为这 3 种方法都和读取有关，因此 open()函数中只允许写入的 w 模式和只允许追加的 a 模式不支持它们，而其他 4 种模式则都没有问题，举例如下。

```
#w 模式不支持 read()，readline()，readlines()
>>> file = open('test.txt', 'w')
>>> print (file.read())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
io.UnsupportedOperation: not readable
>>>

#a 模式也不支持 read()，readline()，readlines()
>>> file = open('test.txt', 'a')
>>> print (file.readline())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
io.UnsupportedOperation: not readable
>>>
```

read()、readline()和 readlines()是学习 open()函数的重点内容，三者的用法和差异很大，其中 readlines()更是重中之重（原因后面会讲到），作为网络工程师必须熟练掌握。下面对这 3 种函数一一进行讲解。

1. read()

read()方法读取文本文件里的全部内容，**返回值为字符串**。

```
>>> file = open('test.txt')
>>> print (file.read())
Cisco
Juniper
Arista
H3C
Huawei
```

```
>>> print (file.read())
>>>
```

我们尝试连续两次打印 test.txt 文件里的内容，第一次打印出的内容没有任何问题，**为什么第二次打印的时候内容为空了呢**？这是因为在使用 read()方法后，**文件指针的位置从文件的开头移动到了末尾**，要想让文件指针回到开头，必须使用 seek()函数，方法如下。

```
>>> file.seek(0)
0
>>> file.tell()
0
>>> print (file.read())
Cisco
Juniper
Arista
H3C
Huawei

>>> file.tell()
32
>>>
```

我们用 seek(0)将文件指针从末尾移回开头，并且用 tell()方法确认文件指针的位置（文件开头的位置为 0），随后使用 read()方法打印文件内容并成功，之后再次使用 tell()方法确认文件指针的位置，可以发现指针现在已经来到文件末尾处（32）。这个 32 是怎么得来的？下面我们拿掉 print()函数，再次通过 read()方法来读取一次文件内容。

```
>>> file.seek(0)
0
>>> file.read()
'Cisco\nJuniper\nArista\nH3C\nHuawei\n'
>>>
```

拿掉 print()函数的目的是能清楚地看到换行符\n，如果这时从左往右数，则会发现 Cisco(5) + \n(1) + Juniper(7) + \n(1) + Arista(6) + \n(1) + H3C(3) + \n(1) + Huawei(6) + \n(1) = 32，这就解释了为什么在文件指针移动到文件末尾后，tell()方法返回的文件指针的位置是 32。文件指针的位置及 seek()和 tell()方法的用法是文本文件访问和管理中很重要但又容易被忽略的知识点，网络工程师务必熟练掌握。

2. readline()

readline()与 read()的区别是它不会像 read()那样把文本文件里的所有内容都一次性读完，而是会一排一排地去读。**readline()的返回值也是字符串。**举例如下。

```
>>> file = open('test.txt')
>>> print (file.readline())
Cisco

>>> print (file.readline())
Juniper

>>> print (file.readline())
Arista

>>> print (file.readline())
H3C

>>> print (file.readline())
Huawei

>>> print (file.readline())

>>>
```

readline()方法每次返回文件的一排内容，顺序由上至下（这里出现的空排部分是因为换行符的缘故）。另外，文件指针会跟随移动直到文件末尾，因此最后一个 print

(file.readline())的返回值为空。

3. readlines()

readlines()与前两者最大的区别是它的返回值不再是字符串，而是列表。可以说readlines()是 read()和 readline()的结合体，首先它同 read()一样把文本文件的所有内容都读完。另外，它会像 readline()那样一排一排去读，并将每排的内容以列表元素的形式返回，举例如下。

```
>>> file = open('test.txt')
>>> print (file.readlines())
['Cisco\n', 'Juniper\n', 'Arista\n', 'H3C\n', 'Huawei\n']
>>> file.seek(0)
>>> devices = file.readlines()
>>> print (devices[0])
Cisco

>>> print (devices[1])
Juniper

>>> print (devices[2])
Arista

>>> print (devices[3])
H3C

>>> print (devices[4])
Huawei


>>>
```

同 read()和 readline()一样，使用一次 readlines()后，文件指针会移动到文件的末尾。为了避免每次重复使用 seek(0)的麻烦，可以将 readlines()返回的列表赋值给一个变量，即 devices，之后便可以使用索引号来一个一个地验证列表里的元素。**注意 readlines()返回的列表里的元素都带换行符\n，这与我们手动创建的普通列表是有区别的。**

同 read()和 readline()相比，笔者认为 readlines()应该是网络运维中使用频率最高的一

种读取文本文件内容的方法，因为它的返回值是列表，通过列表我们可以做很多事情。举个例子，现在有一个名为 ip.txt 的文本文件，该文件保存了一大堆没有规律可循的交换机管理 IP 地址，具体如下。

```
[root@CentOS-Python ~]# cat ip.txt
172.16.100.1
172.16.30.1
172.16.41.1
172.16.10.1
172.16.8.1
172.16.112.1
172.16.39.1
172.16.121.1
172.16.92.1
172.16.73.1
192.168.54.1
192.168.32.1
192.168.2.1
10.3.2.1
10.58.23.3
192.168.230.29
10.235.21.42
192.168.32.32
10.4.3.3
172.16.30.2
172.16.22.30
172.16.111.33
[root@CentOS-Python ~]#
```

现在需要回答 3 个问题：

（1）怎么使用 Python 来确定该文件有多少个 IP 地址？

（2）怎么使用 Python 来找出该文件中的 B 类 IP 地址（172.16 开头的 IP 地址），并将它们打印出来？

（3）怎么使用 Paramiko 来批量 SSH 登录这些交换机修改或查看配置（Paramiko 后面会讲到）？

答案（1）：最好的方法是使用 open()函数的 readlines()来读取该文件的内容，因为 readlines()的返回值是列表，可以方便我们使用 len()函数来判断该列表的长度，从而得到该文件中所包含 IP 地址的数量，举例如下。

```
>>> f = open('ip.txt')
>>> print (len(f.readlines()))
22
>>>
```

仅仅通过两行代码，就得到了结果：22 个 IP 地址，readlines()返回的列表是不是很方便？

答案（2）：最好的方法依然是使用 readlines()来完成，因为 readlines()返回的列表中的元素的数据类型为字符串，可以使用 for 循环来遍历所有的字符串元素，然后配合 if 语句，通过字符串的 startswith()函数判断这些 IP 地址是否以 172.16 开头。如果是，则将它们打印出来，举例如下。

```
>>> f = open('ip.txt')
>>> for ip in f.readlines():
...     if ip.startswith('172.16'):
...             print (ip)
...
172.16.100.1

172.16.200.1

172.16.130.1

172.16.10.1

172.16.8.1

172.16.112.1

172.16.39.1
```

```
172.16.121.1

172.16.92.1

172.16.73.1

172.16.30.2

172.16.22.30

172.16.111.33
>>>
```

这里需要注意一点，因为 readlines()返回的列表中的元素是带换行符\n 的，所以打印出来的每个 B 类 IP 地址之间都空了一排，影响阅读和美观，解决方法也很简单，只需要使用第 2 章讲过的 strip()函数去掉换行符即可，举例如下。

```
>>> f = open('ip.txt')
>>> for ip in f.readlines():
...     if ip.startswith('172.16'):
...             print (ip.strip())
...
172.16.100.1
172.16.200.1
172.16.130.1
172.16.10.1
172.16.8.1
172.16.112.1
172.16.39.1
172.16.121.1
172.16.92.1
172.16.73.1
172.16.30.2
172.16.22.30
```

```
172.16.111.33
>>>
```

答案（3）：同答案（2）一样，因为 readlines()返回的是列表，我们可以通过 for 循环来一一访问该列表中的每个元素，也就是交换机的每个 IP 地址，进而达到使用 Paramiko 来一一登录每个交换机做配置的目的，该脚本的代码如下。

```
import Paramiko

username = input('Username: ')
password = input('Password: ')
f = open('ip.txt')

for ip in f.readlines():
    ssh_client = Paramiko.SSHClient()
    ssh_client.set_missing_host_key_policy(Paramiko.AutoAddPolicy())
    ssh_client.connect(hostname=ip, username=username, password=password)
    print ("Successfully connect to ", ip)
```

- 关于 Paramiko 的用法在第 4 章会讲到，这里看不懂没关系。
- 这里通过使用 for 循环配合 readlines()返回列表的方式来访问 ip.txt 文件中的交换机管理 IP 地址。
- 该 for 循环会尝试一一登录 ip.txt 的所有交换机 IP 地址，每成功登录一个交换机都随即打印信息“"Successfully connect to "，ip”来提醒用户登录成功。

open()函数的 readlines()在网络运维中的用处远不止这 3 点，这里只是给出了几个比较典型的实用例子，再次强调每一位学习 Python 的网络工程师都必须熟练掌握它。

3.3.3　文件写入

在使用 open()函数创建文件对象后，我们可以使用 **write()函数**来对文件写入数据。顾

名思义，既然 write()函数与文件写入相关，**那么只允许只读的 r 模式并不支持它**，而其他 5 种模式则不受限制（包括 r+模式），举例如下。

```
>>> f = open('test.txt', 'r')
>>> f.write()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: write() takes exactly one argument (0 given)
>>>
```

write()函数在 r+、w/w+、a/a+这 5 种模式中的应用讲解如下。

1. r+

在 r+模式下使用 write()函数，新内容会添加在文件的开头部分，而且会覆盖开头部分原来已有的内容，举例如下。

```
#文本修改前的内容
[root@CentOS-Python ~]# cat test.txt
Cisco
Juniper
Arista
H3C
Huawei

[root@CentOS-Python ~]#

#在 r+模式下使用 write()函数修改文本内容
>>> f = open('test.txt', 'r+')
>>> f.write('Avaya')
>>> f.close()

#文本修改后的内容
[root@CentOS-Python ~]# cat test.txt
Avaya
Juniper
```

```
Arista
H3C
Huawei

[root@CentOS-Python ~]#
```

可以看到，文本开头的 Cisco 已经被 Avaya 覆盖了。这里注意使用 write()函数对文本写入新内容后，必须再用 close()方法将文本关闭，这样新写入的内容才能被保存。

2. w/w+

在 w/w+模式下使用 write()函数，新内容会添加在文件的开头部分，已存在的文件的内容将会完全被清空，举例如下。

```
#文本修改前的内容
[root@CentOS-Python ~]# cat test.txt
Avaya
Juniper
Arista
H3C
Huawei

[root@CentOS-Python ~]#

#在 w 模式下使用 write()函数修改文本内容
>>> f = open('test.txt', 'w')
>>> f.write('test')
>>> f.close()

#文本修改后的内容
[root@CentOS-Python ~]# cat test.txt
test

[root@CentOS-Python ~]#

#在 w+模式下使用 write()函数修改文本内容
```

```
>>> f = open('test.txt', 'w+')
>>> f.write('''Cisco
... Juniper
... Arista
... H3C
... Huawei\n''')
>>> f.close()

#文本修改后的内容
[root@CentOS-Python ~]# cat test.txt
Cisco
Juniper
Arista
H3C
Huawei

[root@CentOS-Python ~]#
```

3. a/a+

在 a/a+模式下使用 write()函数，新内容会添加在文件的末尾部分，已存在的文件的内容将不会被清空，举例如下。

```
#文本修改前的内容：
[root@CentOS-Python ~]# cat test.txt
Cisco
Juniper
Arista
H3C
Huawei

[root@CentOS-Python ~]#

#在 a 模式下使用 write()函数修改文本内容
>>> f = open('test.txt', 'a')
>>> f.write('Avaya')
```

```
>>> f.close()

#文本修改后的内容
[root@CentOS-Python ~]# cat test.txt
Cisco
Juniper
Arista
H3C
Huawei
Avaya

[root@CentOS-Python ~]#

#在 a+模式下使用 write()函数修改文本内容
>>> f = open('test.txt', 'a+')
>>> f.write('Aruba')
>>> f.close()

#文本修改后的内容
[root@CentOS-Python ~]# cat test.txt
Cisco
Juniper
Arista
H3C
Huawei
Avaya
Aruba

[root@CentOS-Python ~]#
```

3.3.4　with 语句

每次用 open()函数打开一个文件，该文件都将一直处于打开状态。这一点我们可以用 closed 方法来验证：closed 方法会返回一个布尔值。如果文件处于打开状态，则返回 False；

如果文件已被关闭，则返回 True。

```
>>> f = open('test.txt')
>>> f.closed
False
>>> f.close()
>>> f.closed
True
>>>
```

这种每次都要手动关闭文件的做法略显麻烦，可以使用 with 语句来管理文件对象。用 with 语句打开的文件将被自动关闭，举例如下。

```
>>> with open('test.txt') as f:
...     print f.read()
...
Cisco
Juniper
Arista
H3C
Huawei

>>> f.closed
True
>>>
```

这里没有使用 close()来关闭文件，因为 with 语句已经自动将文件关闭（closed 方法的返回值为 True）。

3.4 自定义函数

函数是已经组织好的可以被重复使用的一组代码块，它的作用是用来提高代码的重复使用率。在 Python 中，有很多内建函数（Built-in Function），比如前面已经讲到的 type()、dir()、print()、int()、str()、list()、open()等，这些函数在安装好 Python 后就能立即使用。

除了上述内建函数，我们也可以通过创建**自定义函数（User-Defined Function）**来完成一些需要重复使用的代码块，提高工作效率。

3.4.1　函数的创建和调用

在 Python 中，我们使用 def 语句来自定义函数。def 语句后面接函数名和括号()，括号里根据情况可带参数也可不带参数。在自定义函数创建好后，要将该函数调用才能得到函数的输出结果（即使该函数不带参数），举例如下。

```
#带参数的自定义函数
>>> def add(x, y):
...     result = x + y
...     print (result)
...
>>> add(1, 2)
3
>>>

#不带参数的自定义函数
>>> def name():
...     print ('Parry)'
...
>>>
>>> name()
Parry
>>>
```

不管自定义函数是否带参数，**函数都不能在被创建前就调用，**比如下面这段用来求一个数的二次方的脚本代码。

```
[root@CentOS-Python ~]# cat test.txt
square(10)

def square(x):
```

```
 squared = x ** 2
 print (squared)

[root@CentOS-Python ~]# python3.8 test.txt
Traceback (most recent call last):
  File "test.txt", line 1, in <module>
    square(10)
NameError: name 'square' is not defined
[root@CentOS-Python ~]#
```

运行该脚本后报错，原因就是在函数还没有被创建的情况下提前调用了该函数，正确写法如下。

```
[root@CentOS-Python ~]# cat test.txt

def square(x):
 squared = x ** 2
 print squared

square(10)

[root@CentOS-Python ~]# python test.txt
100
[root@CentOS-Python ~]#
```

3.4.2 函数值的返回

任何函数都需要返回一个值才有意义。自定义函数可以用 print 和 return 两种语句来返回一个值，**如果函数中没有使用 print 或 return，则该函数只会返回一个空值 None，举例如下。**

```
>>> def add_1(x):
...  x = x+1
>>> print add_1(1)
```

```
None
>>>
```

◎　print 和 return 的区别

print 用来将返回值打印输出在控制端上，以便让用户看到，但是该返回值不会被保存下来。也就是说，如果将来把该函数赋值给一个变量，该变量的值将仍然为空值 None，举例如下。

```
[root@CentOS-Python ~]# cat test.txt
def name():
   print ('Parry')

name()
a = name()
print (a)

[root@CentOS-Python ~]# python3.8 test.txt
Parry
Parry
None
[root@CentOS-Python ~]#
```

注意这里返回了两个 Parry，是因为除了 name()直接调用函数，将函数赋值给一个变量时，比如这里的 a = name()也会触发调用函数的效果。

而 return 则恰恰相反，在调用函数后，return 的返回值不会被打印输出在控制端上，如果想看输出的返回值，则要在调用函数时在函数前面加上 print。**但是返回值会被保存下来，如果把该函数赋值给一个变量，则该变量的值即该函数的返回值，**举例如下。

```
[root@CentOS-Python ~]# cat test.txt
def name():
   return 'Parry'

name()
a = name()
```

```
print (a)

[root@CentOS-Python ～]# python test.txt
Parry
[root@CentOS-Python ～]#
```

3.4.3　嵌套函数

函数支持嵌套，也就是一个函数可以在另一个函数中被调用，举例如下。

```
[root@CentOS-Python ~]# cat test.txt

def square(x):
    result = x ** 2
    return result

def cube(x):
    result = square(x) * x
       return result

cube(3)
print (cube(3))

[root@CentOS-Python ~]# python test.txt
27
```

我们首先创建一个求 2 次方的函数 square(x)（注意这里用的是 return，不是 print，否则返回值将会是 None，不能被其他函数套用）。然后创建一个求 3 次方的函数 cube(x)，在 cube(x)中我们套用了 square(x)，并且也使用了 return 来返回函数的结果。最后分别使用了 cube(3)和 print(cube(3))来展示和证明 3.4.2 节中提到的“在调用函数后，return 的返回值不会被打印输出在控制端上，如果想看输出的返回值，则要在调用函数时在函数前面加上 print()函数”，因为这里只能看到一个输出结果 27。

3.5　模块

在第 1 章已经讲过，Python 的运行模式大致分为两种：一种是使用解释器的交互模式，

另一种是运行脚本的脚本模式。在交互模式下，一旦退出解释器，那么之前定义的变量、函数及其他所有代码都会被清空，一切都需要从头开始。因此，如果想写一段较长、需要重复使用的代码，则最好使用编辑器将代码写进脚本，以便将来重复使用。

不过在网络工程师的日常工作中，随着网络规模越来越大，需求越来越多，相应的代码也将越写越多。为了方便维护，我们可以把其中一些常用的自定义函数分出来写在一个独立的脚本文件中，然后在交互模式或脚本模式下将该脚本文件导入（import）以便重复使用，这种在交互模式下或其他脚本中被导入的脚本被称为**模块（Module）**。在 Python 中，我们使用 **import** 语句来导入模块，**脚本的文件名（不包含扩展名.py）即模块名。**

被用作模块的脚本中可能带自定义函数，也可能没带，下面将分别举例说明。

3.5.1 不带自定义函数的模块

首先创建一个脚本，将其命名为 script1.py，该脚本的代码只有一行，即打印内容“这是脚本 1.”。

```
[root@CentOS-Python ~]# cat script1.py
# coding=utf-8
print ("这是脚本 1.")
[root@CentOS-Python ~]#
```

然后创建第二个脚本，将其命名为 script2.py。在脚本 2 里，我们将使用 import 语句导入脚本 1（import script1），打印内容“这是脚本 2.”，并运行脚本 2。

```
[root@CentOS-Python ~]# cat script2.py
# coding=utf-8
import script1
print "这是脚本 2."

#运行 script2
```

```
[root@CentOS-Python ~]# python3.8 script2.py
这是脚本 1.
这是脚本 2.
[root@CentOS-Python ~]#
```

可以看到，在运行脚本 2 后，我们同时得到了“这是脚本 1.”和“这是脚本 2.”的打印输出内容，其中，“这是脚本 1.”正是脚本 2 通过 import script1 导入脚本 1 后得到的。

3.5.2　带自定义函数的模块

首先修改脚本 1 的代码，创建一个 test()函数，该函数的代码只有一行，即打印内容“这是带函数的脚本 1.”。

```
[root@CentOS-Python ~]# cat script1.py
# coding=utf-8
def test():
    print ("这是带函数的脚本 1.")
[root@CentOS-Python ~]#
```

然后修改脚本 2 的代码，调用脚本 1 的 test()函数，因为是脚本 1 的函数，所以需要在函数名前加入模块名即 script1.test()。

```
[root@CentOS-Python ~]# cat script2.py
# coding=utf-8
import script1
print "这是脚本 2."
script1.test()

#运行 script2
[root@CentOS-Python ~]# python script2.py
这是脚本 2.
这是带函数的脚本 1.
```

3.5.3 Python 内建模块和第三方模块

除了上述两种用户自己创建的模块，Python 还有内建模块及需要通过 pip 下载安装的第三方模块（也可叫作“库”），下面分别讲解。

1. Python 内建模块

Python 有大量的内建模块直接通过 import 导入就可以使用，后面实验部分的案例代码将会重点讲解这些内建模块的使用。这里仅举一例，我们可以使用 **os** 这个内建函数来发送 Ping 包，判断网络目标是否可达，脚本代码及解释如下。

```
import os
hostname = 'www.cisco.com'
response = os.system("ping -c 1 " + hostname)
if response == 0:
    print (hostname + ' is reachable.')
else:
    print (hostname + ' is not reachable.')
```

- os 是很常用的 Python 内建模块，os 是 operating system 的简称。顾名思义，它是用来与运行代码的主机操作系统互动的，后面实验部分的案例代码会讲到关于 os 在网络运维中的许多用法。
- 这里我们使用 os 模块的 system()函数来直接在主机里执行 ping -c 1 http://www.cisco.com 这条命令，也就是向 http://www.cisco.com 发送 1 个 Ping 包。注意，我们做代码演示的主机是 CentOS，CentOS 中指定 Ping 包个数的参数为-c。如果在 Windows 中运行这段脚本，因为 Windows 中指定 Ping 包个数的参数为-n，则需要将代码里的参数-c 换成-n，也就是"ping -n 1 " + hostname！
- **os.system()的返回值为一个整数。**如果返回值为 0，则表示目标可达；如果为非 0，则表示不可达。我们将 os.system()的返回值赋给变量 response，然后写一个简单

的 if...else 判断语句来做判断。如果 response 的值等于 0，则打印目标可达的信息，反之则打印目标不可达的信息。

执行代码看效果。

```
    [root@CentOS-Python ~]# python3.8 test.py
    PING e2867.dsca.akamaiedge.net (104.86.224.155) 56(84) bytes of data.
    64 bytes from a104-86-224-155.deploy.static.akamaitechnologies.com
(104.86.224.155): icmp_seq=1 ttl=128 time=115 ms

    --- e2867.dsca.akamaiedge.net ping statistics ---
    1 packets transmitted, 1 received, 0% packet loss, time 0ms
    rtt min/avg/max/mdev = 115.206/115.206/115.206/0.000 ms
    www.cisco.com is reachable.
    [root@CentOS-Python ~]#
```

除了 os，能实现 ping 命令功能的 Python 内建模块还有很多，比如 subprocess。当在 Python 中使用 os 和 subprocess 模块来执行 ping 命令时，两者都有个“小缺陷”，就是它们都会显示操作系统执行 ping 命令后的回显内容，如果想让 Python“静悄悄”地 ping，则可以使用第三方模块 pythonping，关于 pythonping 模块的用法在实验部分中将会提到。

2. Python 第三方模块

Python 第三方模块需要从 pip 下载安装，pip 随 Python 版本的不同有相对应的 pip2、pip3 和 pip3.8，在 CentOS 8 里已经内置了 pip2 和 pip3，我们在第一章介绍 Python 3.8 在 Windows 和 Linux 的安装时已经安装了 pip3.8，可以在 Windows 命令行及 CentOS 8 下输入 pip3.8 来验证，如下图所示。

```
命令提示符
C:\Users\admin>
C:\Users\admin>pip3.8

Usage:
  pip3.8 <command> [options]

Commands:
  install                     Install packages.
  download                    Download packages.
  uninstall                   Uninstall packages.
  freeze                      Output installed packages in requirements format.
  list                        List installed packages.
  show                        Show information about installed packages.
  check                       Verify installed packages have compatible dependencies.
  config                      Manage local and global configuration.
  search                      Search PyPI for packages.
  wheel                       Build wheels from your requirements.
  hash                        Compute hashes of package archives.
  completion                  A helper command used for command completion.
  debug                       Show information useful for debugging.
  help                        Show help for commands.
```

```
root@CentOS-Python:~
[root@CentOS-Python ~]# pip3.8

Usage:
  pip3.8 <command> [options]

Commands:
  install                     Install packages.
  download                    Download packages.
  uninstall                   Uninstall packages.
  freeze                      Output installed packages in requirements format.
  list                        List installed packages.
  show                        Show information about installed packages.
  check                       Verify installed packages have compatible dependen
cies.
  config                      Manage local and global configuration.
  search                      Search PyPI for packages.
  wheel                       Build wheels from your requirements.
  hash                        Compute hashes of package archives.
  completion                  A helper command used for command completion.
  debug                       Show information useful for debugging.
  help                        Show help for commands.
```

对于网络工程师来说，最常用的 Python 第三方模块无疑是用来 SSH 登录网络设备的 Paramiko 和 Netmiko。首先使用命令 pip3.8 install Paramiko 和 pip3.8 install Netmiko 来分别安装它们（如果你的 Python 不是 3.8.x，而是其他 Python 3 版本，则将 pip3.8 替换成 pip3 即可）。在安装之前，请确认你的 CentOS 8 主机或虚拟机能够连上外网。

```
[root@CentOS-Python ~]#pip3.8 install Paramiko
```

```
[root@CentOS-Python ~]#pip3.8 install Netmiko
```

注：如果你使用的是 Python 2，需要使用 pip2 install Netmiko == 2.4.2 来安装 Netmiko，因为 2020 年 1 月后，所有通过 pip 安装的 Netmiko 都默认只支持 Python 3。

使用 pip 安装好 Paramiko 和 Netmiko 后，打开 Python 测试是否可以使用 import Paramiko 和 import Netmiko 来引用它，如果没报错，则说明安装成功。

```
>>> import Paramiko
>>> import Netmiko
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'Netmiko'
>>>
```

由上面代码可以看到，import Paramiko 后没有任何问题，但是 import Netmiko 却收到解释器报错 ModuleNotFoundError: No module named 'Netmiko'，这是因为为了效果演示，笔者故意只安装了 Paramiko 而没有安装 Netmiko。关于 Paramiko 模块的具体用法后面将会详细讲到，这里仅做演示来说明如何在 Python 中通过 pip 来安装第三方模块。

3.5.4 from ... import ...

对于带自定义函数的模块，除了 import 语句，我们还可以使用 from...import...来导入模块。**它的作用是省去了在调用模块函数时必须加上模块名的麻烦**，该语句具体的格式为 **from[模块名] import [函数名]**，举例如下。

将脚本 2 的代码修改如下，然后运行脚本 2。

```
[root@CentOS-Python ~]# cat script2.py
# coding=utf-8
from script1 import test
print ("这是脚本 2.")
test()

#运行 script2
[root@CentOS-Python ~]# python script2.py
这是脚本 2.
这是带函数的脚本 1.
```

因为我们在脚本 2 中使用了 from script1 import test，所以在调用脚本 1 的 test()函数时不再需要写成 script1.test()，直接用 test()即可。

3.5.5 if __name__ == '__main__':

在 3.5.1 节所举的例子中，脚本 2 在导入了模块脚本 1 后，立即就引用了脚本 1 的代码，即 print ("这是脚本 1.")。有时我们只是希望使用所导入的模块中的部分函数，并不希望一次性全部引入模块中的代码内容，这时就可以用 **if __name__ == '__main__':**这个判断

语句来达到目的。

在基础语法中已经讲过，在 Python 中前后带双下划线的变量叫作内置变量，如 __name__，关于内置变量的内容已经超出了本书的范围。我们可以这么来理解：**如果一个 Python 脚本文件中用到了 if __name__ == '__main__':判断语句，则所有写在其下面的代码将不会在该脚本被其他脚本用作模块导入时被执行。**

我们首先将脚本 1 和脚本 2 的代码分别修改如下，在脚本 1 中将 print ("这是脚本 1.)" 写在 if __name__ == '__main__':的下面。

```
[root@CentOS-Python ~]# cat script1.py
# coding=utf-8
if __name__ == '__main__':
   print ("这是脚本 1.")
[root@CentOS-Python ~]#

[root@CentOS-Python ~]# cat script2.py
# coding=utf-8
import script1
print ("这是脚本 2.")
[root@CentOS-Python ~]#
```

然后运行脚本 2，发现虽然脚本 2 导入了脚本 1，但是输出结果中却不再出现“这是脚本 1.”的输出结果。

```
[root@CentOS-Python ~]# python3.8 script2.py
这是脚本 2.
[root@CentOS-Python ~]#
```

3.6　正则表达式

在网络工程师的日常工作中，少不了要在路由器、交换机、防火墙等设备的命令行中使用各种 show 或者 display 命令来查询配置、设备信息或者进行排错，比如思科和 Arista 设备上最常见的 show run、show log、show interface 和 show ip int brief 等命令。通常这些命令输出的信息和回显内容过多，需要用管道符号|（Pipeline）配合 grep （Juniper 设备）

或者 include/exclude/begin（思科设备）等命令来匹配或筛选我们所需要的信息。举个例子，要查询一台 24 口的思科 2960 交换机当前有多少个端口是 Up 的，可以使用命令 show ip int brief | i up，代码如下。

```
2960#show ip int b | i up
FastEthernet0/2 unassigned YES unset up up
FastEthernet0/10  unassigned YES unset up up
FastEthernet0/11  unassigned YES unset up up
FastEthernet0/12  unassigned YES unset up up
FastEthernet0/15  unassigned YES unset up up
FastEthernet0/23  unassigned YES unset up up
FastEthernet0/24  unassigned YES unset up up
GigabitEthernet0/2  unassigned YES unset up up
Loopback0 unassigned YES NVRAM up up
```

管道符号后面的命令部分（i up）即本节将要讲到的**正则表达式（Regular Expression）**。另外，如果你是 CCIE 或者是拥有数年从业经验、负责过大型企业网或运营商网络运维的资深网络工程师，那么应该不会对 BGP（Border Gateway Protocol，边界网关协议）中出现的自治域路径访问控制列表（as-path access list）感到陌生。as-path access list 是正则表达式在计算机网络中最典型的应用，比如下面的 as-path access list 1 中的^4$就是一个标准的正则表达式。

```
ip as-path access-list 1 permit ^4$
```

3.6.1 什么是正则表达式

根据维基百科的解释：

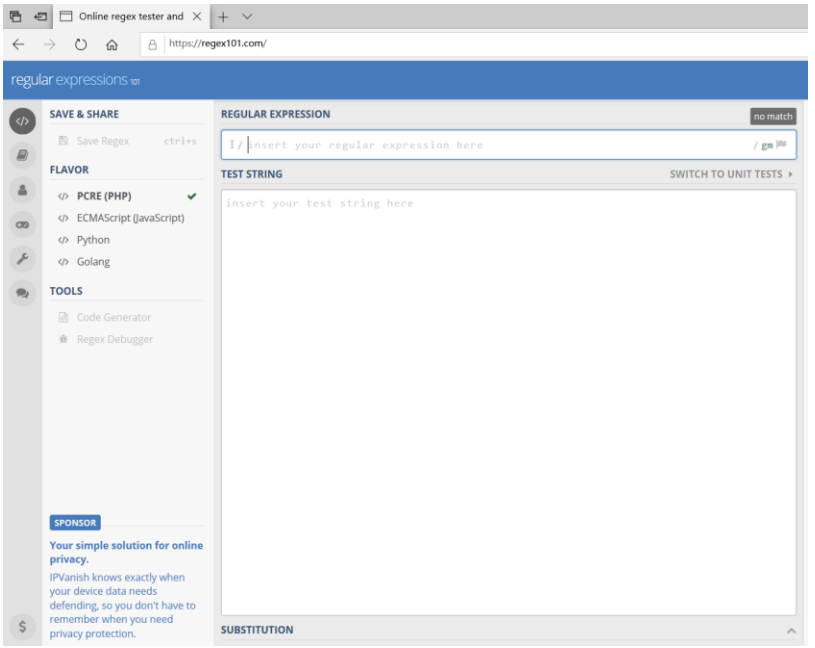
正则表达式（Regular Expression，在代码中常简写为 regex、regexp 或 RE），又称**正规表示式、正规表示法、正规表达式、规则表达式、常规表示法**，是计算机科学的一个概念。正则表达式使用单个字符串来描述、匹配一系列匹配某个句法规则的字符串。在很多

文本编辑器 k，正则表达式通常被用来检索、替换那些匹配某个模式的文本。

在 Python 中，我们使用正则表达式来对文本内容做解析（Parse），即从一组给定的字符串内容中通过给定的字符来搜索和匹配我们需要的“模式”（Pattern，**可以把“模式”理解成我们要搜索和匹配的“关键词”**）。正则表达式本身并不是 Python 的一部分，它拥有自己独特的语法及一个独立的处理引擎，效率上可能不如字符串自带的各种方法，但它的功能远比字符串自带的方法强大得多，因为它同时支持**精确匹配(Exact Match)**和**模糊匹配（Wildcard Match）**。

3.6.2　正则表达式的验证

怎么知道自己的正则表达式是否写正确了呢？方法很简单，可以通过在线正则表达式模拟器来校验自己写的正则表达式是否正确。在线正则表达式模拟器很多，通过搜索引擎很容易找到.在线正则表达式模拟器的使用也很简单，通常只需要提供文本内容（即字符串内容），以及用来匹配模式的正则表达式即可，下图的 regex101 就是笔者常用的在线正则表达式模拟器，后面在举例讲解正则表达式时，笔者会提供在该模拟器上验证的截图来佐证。



3.6.3 正则表达式的规则

正则表达式是一套十分强大但也十分复杂的字符匹配工具，本小节只选取其中部分网络运维中常用并且适合网络工程师学习的知识点，然后配合案例讲解。首先来了解正则表达式的**精确匹配**和**模糊匹配**，其中模糊匹配又包括**匹配符号（Matching Characters）**和**特殊序列（Special Sequence）**。

1. 精确匹配

精确匹配很好理解，即明文给出我们想要匹配的模式。比如上面讲到的在 24 口思科的 2960 交换机里查找 Up 的端口，我们就在管道符号|后面明文给出模式 Up。又比如我们想在下面的交换机日志中找出所有日志类型为%LINK-3-UPDOWN 的日志，那我们按照要求明文给出模式%LINK-3-UPDOWN 即可，即命令 show logging | i %LINK-3- UPDOWN。

```
    2960#show logging | i %LINK-3-UPDOWN
    000458: Feb 17 17:10:31.906: %LINK-3-UPDOWN: Interface FastEthernet0/2, changed
state to down
```

但是如果同时有多种日志类型需要匹配，代码如下。

```
    000459: Feb 17 17:10:35.202: %LINK-3-UPDOWN: Interface FastEthernet0/2, changed
state to up
    000460: Feb 17 17:10:36.209: %LINEPROTO-5-UPDOWN: Line protocol on Interface
FastEthernet0/2, changed state to up
    000461: Feb 17 22:39:26.464: %SSH-5-SSH2_SESSION: SSH2 Session request from
10.1.1.1 (tty = 0) using crypto cipher 'aes128-cbc', hmac 'hmac-sha1' Succeeded
    000462: Feb 17 22:39:27.748: %SSH-5-SSH2_USERAUTH: User 'test' authentication
for SSH2 Session from 10.1.1.1 (tty = 0) using crypto cipher 'aes128-cbc', hmac
'hmac-sha1' Succeeded
```

这时，再用精确匹配就显得很笨拙，因为要通过命令 show logging | i %LINK-3-UPDOWN|%LINEPROTO-5-UPDOWN|%SSH-5-SSH2_SESSION|%SSH-5-SSH2_USERAUTH 把所有感兴趣的日志类型都明文列出来，这里只有 4 种感兴趣的日志类型还比较容易，如果有几十上百种日志类型去匹配，再用精确匹配那工作量就会很大，这时我们需要借助模糊匹配来完成这项任务。

2. 模糊匹配

模糊匹配包括匹配符号和特殊序列，下面分别讲解。

正则表达式中常见的匹配符号如下表所示。

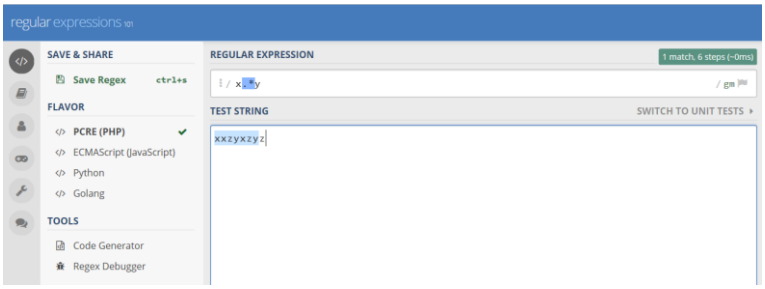
匹配符号	用　法
.	匹配除换行符外的所有字符（一次）
*	用来匹配紧靠该符号左边的符号，匹配次数为 0 次或多次
+	用来匹配紧靠该符号左边的符号，匹配次数为 1 次或多次
?	用来匹配紧靠该符号左边的符号，匹配次数为 0 次或 1 次
{m}	用来匹配紧靠该符号左边的符号，指定匹配次数为 *m* 次。例如字符串'abbbbcccd'，使

	用 ab{2}将匹配到 abb，使用 bc{3}d 将匹配到 bcccd
{m,n}	用来匹配紧靠该符号左边的符号，指定匹配次数为最少 *m* 次，最多 *n* 次。例如，如果字符串为'abbbbcccd'，使用 ab{2,3}将只能匹配到 abb；如果字符串为'abbbcccdabbccd'，使用 ab{2,3}将能同时匹配到 abbb 和 abb；如果字符串内容为'abcd'，使用 ab{2,3}将匹配不到任何东西
{m,}	用来匹配紧靠该符号左边的符号，指定匹配次数最少为 *m* 次，最多无限次
{,n}	用来匹配紧靠该符号左边的符号，指定匹配次数最少为 0 次，最多为 *n* 次
\	转义字符，用来匹配上述“匹配符号”。例如字符串内容中出现了问号？，而又想精确匹配这个问号，那就要用\？来进行匹配。除此之外，\也用来表示一个特殊序列，特殊序列将在下节讲到
[]	表示字符集合，用来精确匹配。例如要精确匹配一个数字，可以使用[0-9]；如果要精确匹配一个小写字母，可以用[a-z]；如果要精确匹配一个大写字母，可以用[A-Z]；如果要匹配一个数字、字母或者下划线，可以用[0-9a-zA-Z_]。另外，在[]中加^表示取非，比如[^0-9]表示匹配一个非数字的字符，[^a-z]表示匹配一个非字母的字符，依此类推
|	表示“或匹配”（两项中匹配其中任意一项），比如要匹配 FastEthernet 和 GigabitEthernet 这两种端口名，可以写作 Fa|Gi
(…)	组合，匹配括号内的任意正则表达式，并表示除组合的开始和结尾，例如(b|cd)ef 表示匹配 bef 或 cdef

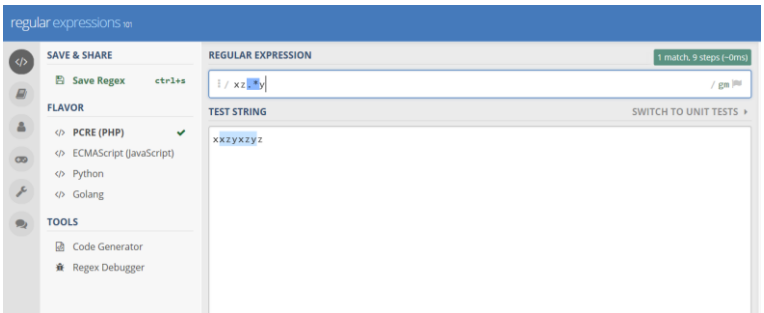
3. 贪婪匹配

*、+、?、{m}、{m，}和{m，n}这 6 种匹配符号默认都是贪婪匹配的，即会尽可能多地去匹配符合条件的内容。

假设给定的一组字符串为'xxzyxzyz'，我们使用正则表达式模式 x.*y 来做匹配（注：**精确匹配和模糊匹配可以混用**）。在匹配到第一个 x 后，开始匹配.*，**因为.和*默认是贪婪匹配的，这里它会一直往后匹配，直到匹配到最后一个 y**，因此匹配结果为 xxzyxzy，可以在 regex101 上做验证，如下图所示。



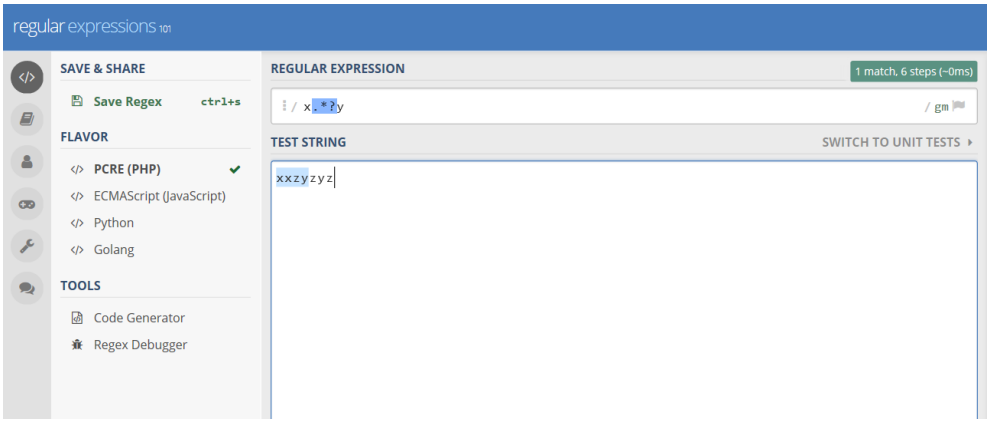
又假设给定的字符串依然为'xxzyxzyz'，我们使用正则表达式模式 xz.*y 来做匹配，在匹配到第一个 xz 后，开始匹配“贪婪”的.*，这里将会一直往后匹配，直到最后一个 y，因此匹配结果为 xzyxzy，在 regex101 上验证，如下图所示。



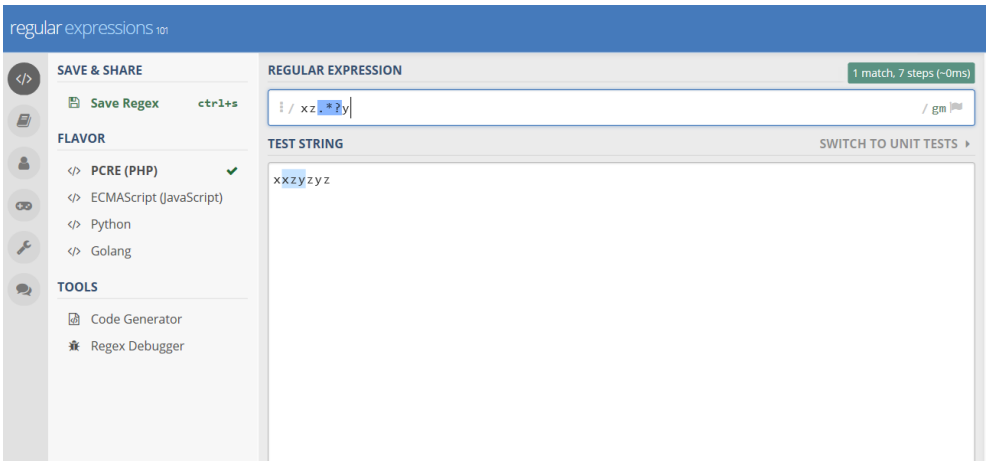
4. 非贪婪匹配

要实现非贪婪匹配很简单，就是在上述 6 种贪婪匹配符号后面加上问号?即可，即*?、+?、??、{m}?、{m，}?和{m，n}?。

假设给定的另一组字符串为'xxzyzyz'（注意不是之前的'xxzyxzyz'），我们使用正则表达式模式 x.*?y 来做匹配。在匹配到第一个 x 后，开始匹配.*?，**因为.*?是非贪婪匹配的，它在匹配到第一个 y 后便随即停止**，因此匹配结果为 xxzy，在 regex101 上验证，如下图所示。



又假设给定的字符串依然为'xxzyzyz'，我们使用正则表达式模式 xz.*?y 来做匹配，在匹配到第一个 xz 后，开始匹配.*?，它在匹配到第一个 y 后便随即停止，因此匹配结果为 xzy，在 regex101 上验证，如下图所示。



◎　正则表达式中常见的特殊序列

特殊序列由转义符号\和一个字符组成，常见的特殊序列及其用法如下表所示。

特殊序列	用　法
\d	匹配任意一个十进制数字，等价于[0-9]

\D	\d 取非，匹配任意一个非十进制数字，等价于[^0-9]
\w	匹配任意一个字母、十进制数字及下划线，等价于[a-zA-Z0-9]
\W	\w 取非，等价于[^a-zA-Z0-9]
\s	匹配任意一个空白字符，包括空格、换行符\n
\S	\s 取非，匹配任意一个非空白字符

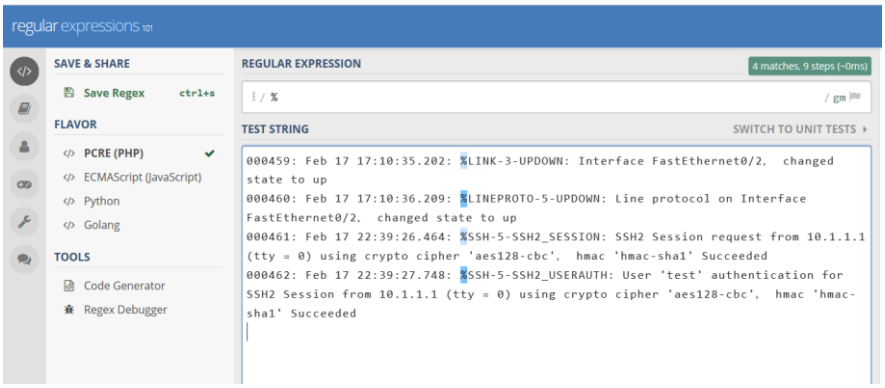
模糊匹配在正则表达式中很常用，前面精确匹配中提到的匹配思科交换机日志类型的例子可以用模糊匹配来处理，比如我们要在下面的日志中同时匹配%LINK-3-UPDOWN、%LINEPROTO-5-UPDOWN、%SSH-5-SSH2_SESSION 和%SSH-5-SSH2_USERAUTH 4 种日志类型，用正则表达式**%\w{3,9}-\d-\w{6,13}**即可完全匹配。

```
    000459: Feb 17 17:10:35.202: %LINK-3-UPDOWN: Interface FastEthernet0/2, changed state to up
    000460: Feb 17 17:10:36.209: %LINEPROTO-5-UPDOWN: Line protocol on Interface FastEthernet0/2, changed state to up
    000461: Feb 17 22:39:26.464: %SSH-5-SSH2_SESSION: SSH2 Session request from 10.1.1.1 (tty = 0) using crypto cipher 'aes128-cbc', hmac 'hmac-sha1' Succeeded
    000462: Feb 17 22:39:27.748: %SSH-5-SSH2_USERAUTH: User 'test' authentication for SSH2 Session from 10.1.1.1 (tty = 0) using crypto cipher 'aes128-cbc', hmac 'hmac-sha1' Succeeded
```

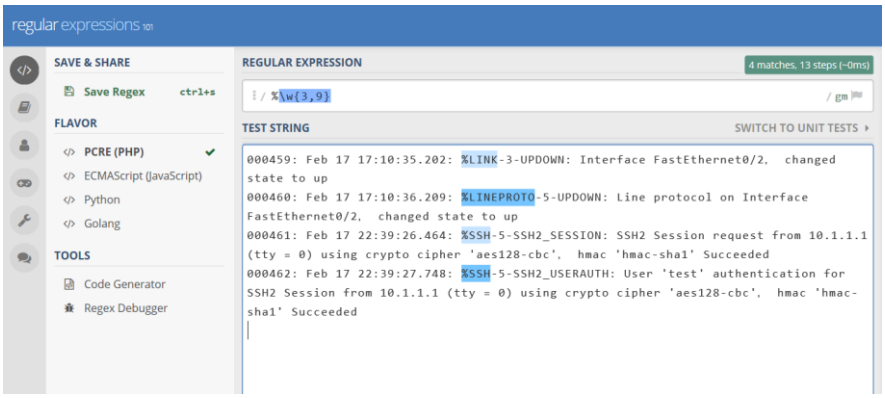
关于该正则表达式%\w{3,9}-\d-\w{4,13}是如何完整匹配上述 4 种日志类型的讲解如下。

首先将%\w{3,9}-\d-\w{4,13}拆分为 6 部分。

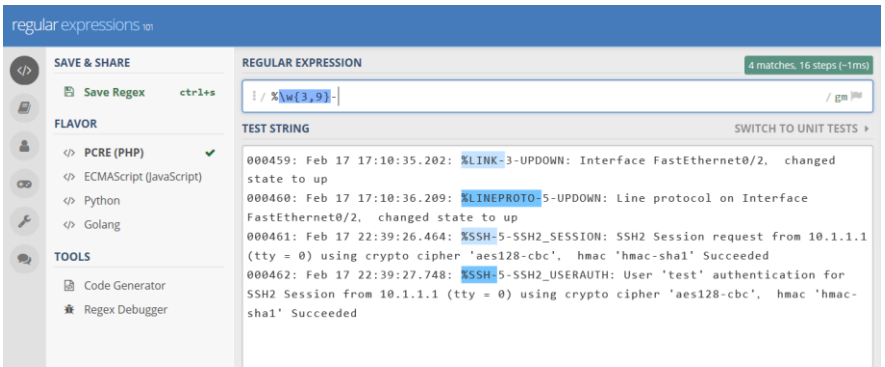
第 1 部分：%用来精确匹配百分号“%”（4 种日志全部以%开头），如下图所示。



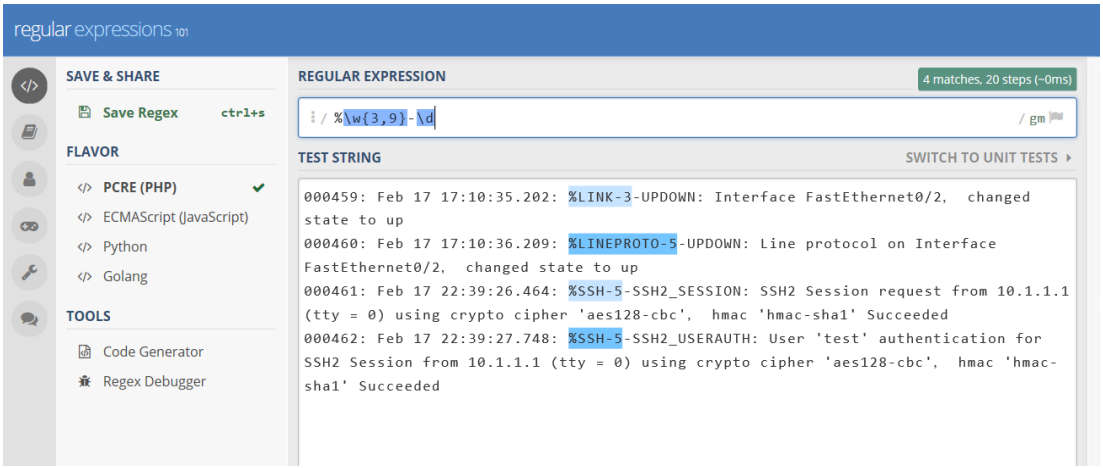
第 2 部分：\w{3,9}用来匹配“SSH”和“LINEPROTO”，如下图所示。



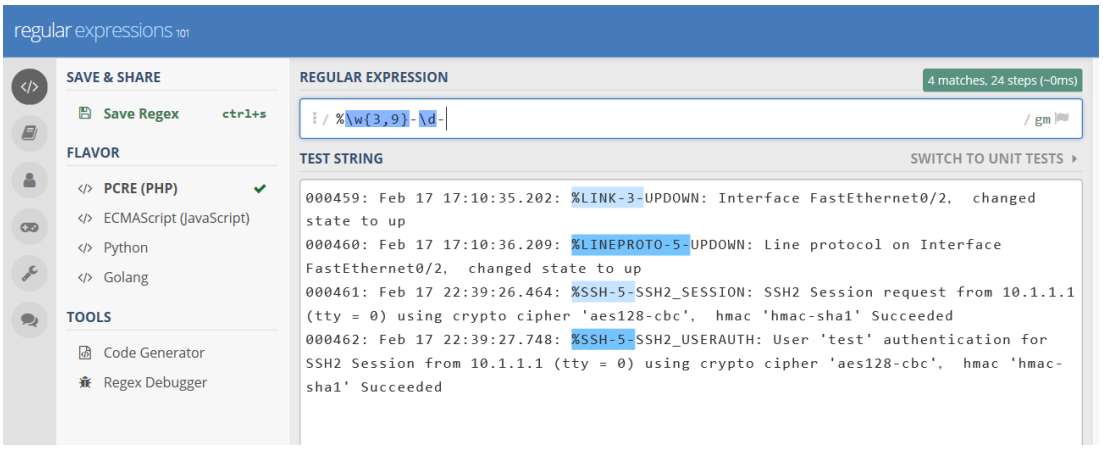
第 3 部分：-用来精确匹配第一个“-”，如下图所示。



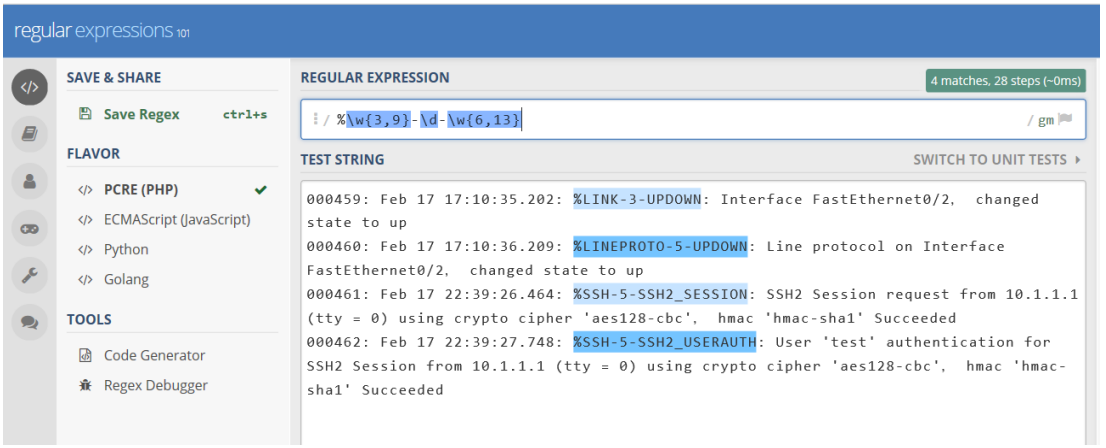
第 4 部分：\d 用来匹配数字“3”或“5”，如下图所示。



第 5 部分：-用来精确匹配第二个“-”，如下图所示。



第 6 部分：由上图可知，至此还剩下“UPDOWN”“SSH2_SESSION”“SSH2_USERAUTH”3 部分需要匹配，因为\w 可以同时匹配数字、字母及下划线，因此，用\w{6,13}即可完整匹配最后这 3 部分，如下图所示。



3.6.4 正则表达式在 Python 中的应用

在 Python 中，我们使用 **import re** 来导入正则表达式这个 Python 内建的模块（无须使用 pip 来安装）。

```
import re
```

首先使用 dir()函数来看下 re 模块中有哪些内建的函数和方法。

```
>>> import re
>>> dir(re)
['DEBUG', 'DOTALL', 'I', 'IGNORECASE', 'L', 'LOCALE', 'M', 'MULTILINE', 'S', 'Scanner', 'T', 'TEMPLATE', 'U', 'UNICODE', 'VERBOSE', 'X', '_MAXCACHE', '__all__', '__builtins__', '__doc__', '__file__', '__name__', '__package__', '__version__', '_alphanum', '_cache', '_cache_repl', '_compile', '_compile_repl', '_expand', '_locale', '_pattern_type', '_pickle', '_subx', 'compile', 'copy_reg', 'error', 'escape', 'findall', 'finditer', 'match', 'purge', 'search', 'split', 'sre_compile', 'sre_parse', 'sub', 'subn', 'sys', 'template']
```

其中，网络工程师较常用的 Python 正则表达式的函数主要有 4 种，分别为 re.match()、re.search()、 re.findall()和 re.sub()，下面对它们分别进行说明。

1. re.match()

re.match()函数用来在字符串的起始位置匹配指定的模式，如果匹配成功，则 **re.match()的返回值为匹配到的对象。如果想查看匹配到的对象的具体值，则还要对该对象调用group()函数**。如果匹配到的模式不在字符串的起始位置，则 re.match()将返回空值 None。

re.match()函数的语法如下。

```
re.match(pattern, string, flags=0)
```

pattern 即我们要匹配的正则表达式模式，string 为要匹配的字符串，flags 为标志位，用来控制正则表达式的匹配方式，如是否区分大小写、是否多行匹配等，flags 为可选项，不是很常用。

举例如下。

```
[root@CentOS-Python ~]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import re
>>> test = 'Test match() function of regular expression.'
>>> a = re.match(r'Test', test)
>>> print (a)
<re.Match object; span=(0, 4), match='Test'>
>>> print (a.group())
Test
>>>
```

我们使用 re.match()函数，从字符串'Test match() function of regular expression.'里精确匹配模式'Test'，因为'Test'位于该段字符串的起始位置，所以匹配成功，并且返回一个匹配到的对象<re.Match object; span=(0, 4), match='Test'>（即用 print(a)看到的内容），为了查看该对象的具体值，我们可以对该对象调用 group()方法，得到具体值'Test'（即用 print(a.group())看到的内容），**该值的数据类型为字符串**，group()函数在 Python 的正则表达式中很常用，务必熟练使用。

如果我们不从字符串的起始位置去匹配，而是去匹配中间或末尾的字符串内容，则 re.match()将匹配不到任何东西，从而返回空值 None。比如我们尝试匹配 function 这个词，因为 function 不在'Test match() function of regular expression.'的开头，所以 re.match()的返回值为 None。

```
>>> import re
>>> test = 'Test match() function of regular expression.'
>>> a = re.match(r'function', test)
>>> print (a)
None
>>>
```

在上面两个例子中，我们分别在模式 Test 和 function 的前面加上了一个 r，**这个 r 代表原始字符串（Raw String）。**在 Python 中，原始字符串主要用来处理特殊字符所产生的歧义，比如前面讲到的转义字符就是一种特殊字符，它会产生很多不必要的歧义。举个例子，假设你要在 Windows 中用 Python 的 open()函数打开一个文件，代码如下。

```
>>> f = open('C:\Program Files\test.txt', 'r')
IOError: [Errno 2] No such file or directory: 'C:\\Program Files\test.txt'
>>>
```

这时，你会发现该文件打不开了，Python 返回了一个“IOError: [Errno 2] No such file or directory: 'C:\\Program Files\test.txt'”错误，这是因为“\t”被当作了不属于文件名的特殊符号。解决的办法也很简单，就是在代表文件所在路径的字符串前面使用原始字符串。

```
>>> f = open(r'C:\Program Files\test.txt', 'r')
>>> print f
<open file 'C:\\Program Files\\test.txt', mode 'r' at 0x0000000002A01150>
>>>
```

在正则表达式中使用原始字符串也是同样的道理，只是还有更多其他原因，对这些原因的解释已经超出了本书的范围。网络工程师只需记住：**在正则表达式中建议使用原始字符串。**

2. re.search()

re.search()函数和 re.match()一样，返回值**为字符串，但是它比 re.match()更灵活，因为它允许在字符串的任意位置匹配指定的模式**。

re.search()函数的语法如下。

```
re.search(pattern, string, flags=0)
```

前面我们用 re.match()在'Test match() function of regular expression.'中尝试匹配 function 这个内容不成功，因为 function 不在该字符串的起始位置。我们改用 re.search()来匹配。

```
>>>import re
>>> test ='Test search() function of regular expression.'
>>> a = re.search(r'function', test)
>>> print (a)
<re.Match object; span=(14, 22), match='function'>
>>> print (a.group())
function
>>>
```

虽然 re.search()可以在字符串的任意位置匹配模式，但是它和 re.match()一样**一次只能匹配一个字符串内容**，比如下面是某台路由器上 show ip int brief 的回显内容，我们希望用正则表达式来匹配在该回显内容中出现的所有 IPv4 地址。

```
Router#show ip int b
Interface IP-Address OK? Method Status Protocol
GigabitEthernet1/1 192.168.121.181 YES NVRAM up up
GigabitEthernet1/2 192.168.110.2  YES NVRAM up up
GigabitEthernet2/1 10.254.254.1 YES NVRAM up up
GigabitEthernet2/2 10.254.254.5 YES NVRAM up up
```

我们尝试用 re.search()来匹配。

```
>>> test = '''Router#show ip int b
Interface IP-Address OK? Method Status Protocol
```

```
GigabitEthernet1/1 192.168.121.181 YES NVRAM up up
GigabitEthernet1/2 192.168.110.2  YES NVRAM up up
GigabitEthernet2/1 10.254.254.1 YES NVRAM up up
GigabitEthernet2/2 10.254.254.5 YES NVRAM up up '''
>>> ip_address = re.search(r'\d{1, 3}\.\d{1, 3}\.\d{1, 3}\.\d{1, 3}', test)
>>> print (ip_address.group())
192.168.121.181
```

我们用正则表达式\d{1，3}\.\d{1，3}\.\d{1，3}\.\d{1，3}作为模式来匹配任意 IPv4 地址，注意我们在分割每段 IP 地址的“.”前面加了转义符号\，如果不加\，写成\d{1，3}.\d{1，3}.\d{1，3}.\d{1，3}，则将会匹配到 GigabitEthernet1/1 中的‘1/1’，原因请读者自行思考。

print (a.group())后可以看到只匹配到了 192.168.121.181 这一个 IPv4 地址，这是 re.search()的短板。如果想匹配到其他所有 IPv4 地址（192.168.110.2，10.254.254.1，10.254.254.5），则必须用下面将要讲到的 re.findall()。

3. re.findall()

如果字符串中有多个能被模式匹配到的关键词，并且我们希望把它们全部匹配出来，则要用到 re.findall()。同 re.match()和 re.search()不一样，**re.findall()的返回值为列表，每个被模式匹配到的字符串内容分别为该列表中的元素之一。**

re.findall()函数的语法如下。

```
re.findall(pattern, string, flags=0)
```

以上面尝试匹配命令 show ip int brief 的回显内容中所有 IPv4 地址的例子为例。

```
>>> test = '''Router#show ip int b
Interface IP-Address OK? Method Status Protocol
GigabitEthernet1/1 192.168.121.181 YES NVRAM up up
GigabitEthernet1/2 192.168.110.2  YES NVRAM up up
GigabitEthernet2/1 10.254.254.1 YES NVRAM up up
GigabitEthernet2/2 10.254.254.5 YES NVRAM up up '''
>>> ip_address = re.findall(r'\d{1, 3}\.\d{1, 3}\.\d{1, 3}\.\d{1, 3}', test)
```

```
>>> type(ip_address)
<type 'list'>
>>> print (ip_address)
['192.168.121.181', '192.168.110.2', '10.254.254.1', '10.254.254.5']
>>>
```

这里成功通过 re.findall()匹配到所有的 4 个 IPv4 地址，每个 IPv4 地址分别为 re.findall()返回的列表中的一个元素。

4. re.sub()

最后要讲的 **re.sub()函数是用来替换字符串里被匹配到的字符串内容的，**类似 Word 里的替换功能。除此之外，它还可以定义最大替换数（Maximum Number of Replacement）来指定 sub()函数所能替换的字符串内容的数量，默认状态下为全部替换。**re.sub()的返回值是字符串**。

re.sub()函数的语法如下。

```
a = re.match(pattern, replacement, string, optional flags)
```

re.sub()函数的语法与前面讲到的 re.match()、re.search()、re.findall() 3 个函数略有不同，re.sub()函数里多了一个 replacement 参数，它表示被替换后的字符串内容。optional flags 可以用来指定所替换的字符串内容的数量，如果只想替换其中 1 个字符串内容，则可以将 optional flags 位设为 1；如果想替换其中的前两个字符串内容，则设为 2；依此类推。如果 optional flags 位空缺，则默认状态下为全部替换。

下面以某台路由器上的 ARP 表的输出内容为例，我们用 re.sub()来将其中所有的 MAC 地址全部替换为 1234.56ab.cdef。

```
>>> test = '''
... Router#show ip arp
... Protocol Address Age (min) Hardware Addr  Type  Interface
... Internet 10.1.21.1    -   b4a9.5aff.c845 ARPA  TenGigabitEthernet2/1
... Internet 10.11.22.1   51  b4a9.5a35.aa84 ARPA  TenGigabitEthernet2/2
```

```
... Internet 10.201.13.17  -   b4a9.5abe.4345 ARPA  TenGigabitEthernet2/3'''
>>> a = re.sub(r'\w{4}\.\w{4}\.\w{4}', '1234.56ab.cdef', test)
>>> print (a)

Router#show ip arp
Protocol  Address     Age (min)  Hardware Addr   Type   Interface
Internet  10.1.21.1       -      1234.56ab.cdef  ARPA   TenGigabitEthernet2/1
Internet  10.11.22.1     51      1234.56ab.cdef  ARPA   TenGigabitEthernet2/2
Internet  10.201.13.17    -      1234.56ab.cdef  ARPA   TenGigabitEthernet2/3
```

因为 optional flags 位空缺，所以默认将 3 个 MAC 地址全部替换成 1234.56ab.cdef。如果将 optional flags 位设为 1，则只会替换第一个 MAC 地址，效果如下。

```
>>> a = re.sub(r'\w{4}\.\w{4}\.\w{4}', '1234.56ab.cdef', test, 1)
>>> print (a)

Router#show ip arp
Protocol  Address     Age (min)  Hardware Addr   Type   Interface
Internet  10.1.21.1       -      1234.56ab.cdef  ARPA   TenGigabitEthernet2/1
Internet  10.11.22.1     51      b4a9.5a35.aa84  ARPA   TenGigabitEthernet2/2
Internet  10.201.13.17    -      b4a9.5abe.4345  ARPA   TenGigabitEthernet2/3
```

如果只希望替换某一个特定的 MAC 要怎么做呢？方法也很简单，这个问题留给读者朋友自行思考和实验。

3.7 异常处理

异常处理（Exception Handling）是 Python 中很常用的知识点。通常在写完代码第一次运行脚本时，我们难免会遇到一些代码错误。Python 中大致有两种代码错误：**语法错误（Syntax Errors）和异常（Exceptions）**。比如下面这种忘了在 if 语句末尾加上冒号:就是一种典型的语法错误，Python 会回复一个“SyntaxError: invalid syntax”的报错信息。

```
>>> if True
  File "<stdin>", line 1, in ?
```

```
    if True
         ^
SyntaxError: invalid syntax
>>>
```

有时一条语句在语法上是正确的，但是执行代码后依然会引发错误，这类错误叫作异常。异常的种类很多，比如把零当作除数的“零除错误”（ZeroDivisonError）、变量还没创建就被调用的“命名错误”（NameError）、数据类型使用有误的“类型错误”（TypeError），以及尝试打开不存在的文件时会遇到的“I/O 错误”（IOError）等都是很常见的异常。这些都是 Python 中常见的内置异常，也就是在没有导入第三方模块的情况下会遇到的异常，举例如下。

```
#零除错误
>>> 100 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>

#命名错误
>>> print name
Traceback (most recent call last):
  File "<stdin>",  line 1,  in ?
NameError: name 'name' is not defined
>>>

#类型错误
>>> a = 10
>>> print ("There are " + a + "books.")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>>

#I/O 错误
```

```
>>> f = open('abc.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'abc.txt'
>>>
```

除了这些常见的 Python 内置异常，从第三方导入的模块也有自己独有的异常，比如后面实验部分将会重点讲的 Paramiko 就有与 SSH 用户名、密码错误相关的“AuthenticationException 异常”，以及网络设备 IP 地址不可达导致的 Socket 模块的“socket.error 异常”。关于这些第三方模块的使用及它们的异常的处理在第 4 章的实验 5 将会重点介绍。

使用异常处理能提高代码的鲁棒性，帮助程序员快速修复代码中出现的错误。在 Python 中，我们使用 try...except...语句来做异常处理，举例如下。

```
>>> for i in range(1, 6):
...     print (i/0)
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>

>>> for i in range(1, 6):
...     try:
...       print (i/0)
...     except ZeroDivisionError:
...       print "Division by 0 is not allowed"
...
Division by 0 is not allowed
Division by 0 is not allowed
Division by 0 is not allowed
Division by 0 is not allowed
Division by 0 is not allowed
>>>
```

我们故意尝试触发零除错误，在没有做异常处理时，如果 Python 解释器发现 range(1，6)返回的第一个数字 1 试图去和 0 相除，会马上返回一个 ZeroDivisonError，**并且程序就此终止。**在使用 try...except...做异常处理时，我们使用 except 去主动捕获零除错误这个异常类型 （except ZeroDivisonError:），并告诉 Python 解释器：在遇到零除错误时，**不要马上终止程序，**而是打印出“Division by 0 is not allowed”这个信息告知用户代码具体出了什么问题，**然后继续执行剩下的代码，直至完成**。正因如此，在遇到零除错误后，程序并没有终止，而是接连打印出了 5 个“Division by 0 is not allowed”（因为 range(1,6)返回了 1、2、3、4、5 共 5 个整数）。

为了更清楚地展示 try...except...带来的当执行代码遇到语法错误和异常时，Python 解释器**“不会马上终止程序”**的好处，我们把上述代码稍作修改。

```
>>> for i in range(1, 6):
...     if i == 1:
...         print (i/0)
...     else:
...         print (i/1)
...
Traceback (most recent call last):
 File "<stdin>", line 3, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>

>>> for i in range (1, 6):
...     try:
...         if i == 1:
...             print (i/0)
...         else:
...             print (i/1)
...     except ZeroDivisionError:
...         print 'Division by 0 is not allowed'
...
Division by 0 is not allowed
2
```

```
3
4
5
>>>
```

我们在原先的 for 循环语句里加入了 if 判断语句，只有当 range(1，6)创建的整数 1～5 的数字为 1 时，才会触发零除错误。当整数列表里的数字不为 1 时，我们把它和 1 相除并打印出结果。在没有使用异常处理的情况下，Python 解释器在返回 ZeroDivisionError: integer division or modulo by zero 后马上终止了程序。在使用异常处理后，在触发零除错误后，程序并没有被中断，而是在打印“Division by 0 is not allowed”后继续执行。

另外，在上面两个例子中，我们都在 except 语句后面加入了 ZeroDivisionError，这种提前捕获异常类型的做法是因为我们知道自己的程序将会触发零除错误，如果这时代码里出现了另外一种异常，则程序还是会被中断，因为 except 并没有捕获到该异常。一般来说，我们很难记住 Python 中所有可能出现的异常类型，如果代码复杂，则无法预测脚本中将会出现什么样的异常类型，当出现这种情况时，我们要怎么做异常处理呢？有两种方法：一是在 except 语句后面不接任何异常类型，直接写成“except:”。二是通过 Exceptions 捕获所有异常。下面分别举例讲解。

```
>>> try:
...     10 / 0
... except:
...     print ("There’s an error.")
...
There’s an error.
>>>

>>> try:
...     10/0
... except Exception as e:
...     print (e)
...
integer division or modulo by zero
>>>
```

当单独使用 except、不捕获任何异常时，可以确保程序在遭遇任何异常时都不会被中断，并返回自定义打印出来的错误信息（比如上面例子中的"There’s an error. "）来代替所有可能出现的异常类型。这么做的优点是省事，但缺点也很明显：我们无法知道代码中具体出现了什么类型的异常，导致代码排错困难。

而使用 Exceptions 时（except Exception as e:），不仅可以捕获除 SystemExit、KeyboardInterrupt、GeneratorExit 外的所有异常，还能让 Python 告诉我们具体的错误原因（比如上面例子中出现的零除错误），方便我们对代码进行排错。如果要捕获 SystemExit、KeyboardInterrupt 和 GeneratorExit 这 3 种异常，则只需要把 Exception 替换成 BaseExceptions 即可。

第 4 章

Python 网络运维实践（GNS3 模拟器）

在第 2 章和第 3 章详细介绍了 Python 的基础语法和进阶语法后，本章和第 5 章将分别以实验和实战的形式讲解 Python 在网络运维中的具体应用。本章共分为 4 个实验，实验难度循序渐进，所有实验都将在 GNS3 模拟器上演示，而第 5 章的 3 个实验将在生产网络的真机网络设备上实战运行。本章和第 5 章的每个 Python 脚本代码都将提供详细的分段讲解，并且提供脚本运行前、脚本运行中、脚本运行后的截图，帮助读者清晰、直观地了解 Python 是如何把繁杂、单调、耗时的传统网络运维工作实现自动化的。

4.1 实验运行环境

◎ 模拟器运行环境

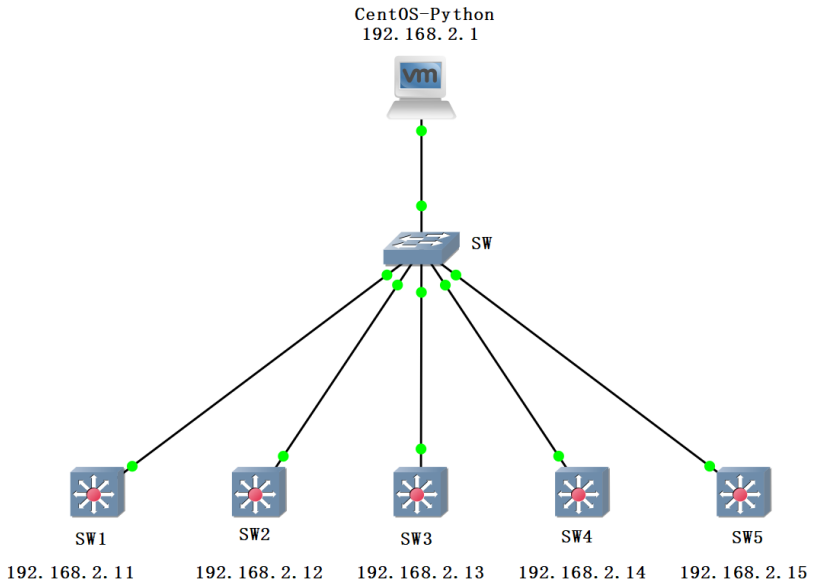
主机操作系统：Windows 10 上运行 CentOS 8（VMware 虚拟机）

网络设备：GNS3 模拟器上运行的思科三层交换机

网络设备 OS 版本：思科 IOS （vios_12-ADVENTERPRISEK9-M）

Python 版本：3.8.2

实验网络拓扑图如下所示。



局域网 IP 地址段：192.168.2.0 /24

运行 Python 的 CentOS 主机：192.168.2.1

SW1：192.168.2.11

SW2：192.168.2.12

SW3：192.168.2.13

SW4：192.168.2.14

SW5：192.168.2.15

所有交换机都已经预配好了 SSH 和 Telnet，用户名为 python，密码为 123，用户的权限为 15 级，SSH 或 Telnet 登录后直接进入特权模式，无须输入 enable 密码。

4.2 Python 中的 Telnet 和 SSH 模块

在 Python 中，支持 Telnet/SSH 远程登录访问网络设备的模块很多，常见的有 Telnetlib、Ciscolib、Paramiko、Netmiko 和 Pexpect。其中，Telnetlib 和 Ciscolib 对应 Telnet 协议，后面 3 个对应 SSH 协议。

因为篇幅有限，Ciscolib 和 Pexpect 不在本书的讨论范围之内，本书将简单介绍 Telnetlib 和 Netmiko，重点介绍 Paramiko，后面的实验脚本也都将基于 Paramiko 完成。关于这 3 种模块的使用方法将通过下面的实验来介绍。

实验目的：通过 Telnetlib、Netmiko 和 Paramiko 模块，分别登录交换机 SW1（192.168.2.11）、SW2（192.168.2.12）、SW3（192.168.2.13），给 SW1 的 loopback 1 端口配置 IP 地址 1.1.1.1/32，给 SW2 的 loopback 1 端口配置 IP 地址 2.2.2.2/32，给 SW3 的 loopback 1 端口配置 IP 地址 3.3.3.3/32。

4.2.1 Telnetlib

在 Python 中，我们使用 Telnetlib 模块来 Telnet 远程登录网络设备，Telnetlib 为 Python 内建模块，不需要 pip 下载安装就能直接使用。鉴于 Telnet 的安全性，通常不建议在生产网络中使用 Telnet，这里只举一例来讲解 Telnetlib 模块的使用方法。

Telnetlib 在 Python 2 和 Python 3 中有非常大的区别，虽然本书内容基于 Python 3.8，但是鉴于部分读者对 Telnet 还有需求，并且可能使用过 Python 2，这里将分别介绍 Telnetlib 在 Python 2 和 Python 3 中的使用方法。

1. Telnetlib 在 Python 2 中的应用

首先手动登录 SW1，确认它此时没有 loopback 1 这个端口，再开启 debug telnet 便于后面运行代码时做验证。

```
SW1#
SW1#show run int loop 1
                        ^
% Invalid input detected at '^' marker.

SW1#debug telnet
Incoming Telnet debugging is on
SW1#
```

然后在运行 Python 的主机上（后面统称主机）创建下面的脚本，将其命名为 telnet.py。

```
[root@CentOS-Python ~]# cat telnet.py
import telnetlib

host = "192.168.2.11"
user = "python"
password = "123"

tn = telnetlib.Telnet(host)
tn.read_until("Username: ")
tn.write(user + "\n")
tn.read_until("Password: ")
tn.write(password + "\n")

tn.write("conf t\n")
tn.write("int loopback 1\n")
tn.write("ip address 1.1.1.1 255.255.255.255\n")
tn.write("end\n")
tn.write("exit\n")

print (tn.read_all())
```

代码分段讲解如下。

（1）首先通过 import 语句导入 telnetlib 模块。

```
import telnetlib
```

（2）然后创建 host、user、password3 个变量，分别对应 SW1 的管理 IP 地址、Telnet 用户名和密码，注意这 3 个变量的数据类型均为字符串。

```
host = "192.168.2.11"
user = "python"
password = "123"
```

（3）调用 telnetlib 的 Telnet()函数，将它赋值给变量 tn，尝试以 Telnet 方式登录 192.168.2.11。

```
tn = telnetlib.Telnet(host)
```

（4）通过 Telnet 登录思科交换机时，终端输出最下面提示的信息始终为“Username:”，如下所示。

```
[root@localhost ~]# telnet 192.168.2.11
Trying 192.168.2.11...
Connected to 192.168.2.11.
Escape character is '^]'.

******************************************************************************
* IOSv - Cisco Systems Confidential                                          *
*                                                                            *
* This software is provided as is without warranty for internal              *
* development and testing purposes only under the terms of the Cisco         *
* Early Field Trial agreement.  Under no circumstances may this software     *
* be used for production purposes or deployed in a production                *
* environment.                                                               *
*                                                                            *
* By using the software, you agree to abide by the terms and conditions      *
* of the Cisco Early Field Trial Agreement as well as the terms and          *
* conditions of the Cisco End User License Agreement at                      *
* http://www.cisco.com/go/eula                                               *
*                                                                            *
* Unauthorized use or distribution of this software is expressly             *
* Prohibited.                                                                *
******************************************************************************
User Access Verification

Username:
```

我们通过 tn.read_until(“Username:”)函数来告诉 Python：如果在终端信息里读到“Username:”字样，则使用 tn.write(user + “\n”)函数来输入 Telnet 用户名并回车。

```
tn.read_until("Username: ")
tn.write(user + "\n")
```

（5）同理，在输入 Username 后，接下来终端将显示“Password:”，如下所示。

```
[root@localhost ~]# telnet 192.168.2.11
Trying 192.168.2.11...
Connected to 192.168.2.11.
Escape character is '^]'.

******************************************************************************
* IOSv - Cisco Systems Confidential                                          *
*                                                                            *
* This software is provided as is without warranty for internal              *
* development and testing purposes only under the terms of the Cisco         *
* Early Field Trial agreement.  Under no circumstances may this software     *
* be used for production purposes or deployed in a production                *
* environment.                                                               *
*                                                                            *
* By using the software, you agree to abide by the terms and conditions      *
* of the Cisco Early Field Trial Agreement as well as the terms and          *
* conditions of the Cisco End User License Agreement at                      *
* http://www.cisco.com/go/eula                                               *
*                                                                            *
* Unauthorized use or distribution of this software is expressly             *
* Prohibited.                                                                *
******************************************************************************

User Access Verification

Username: python
Password:
```

我们通过 tn.read_until("Password: ")函数来告诉 Python：如果在终端信息里读到“Password:”字样，则使用 tn.write(password + “\n”)函数来输入 Telnet 密码并回车。

```
tn.read_until("Password: ")
tn.write(password + "\n")
```

（6）至此，我们已经让 Python 成功通过 Telnet 登录 SW1 了，接下来要做的就是继续用 Telnetlib 的 write()函数在 SW1 上输入各种配置命令来给 loopback 1 端口配置 1.1.1.1/32 这个 IP 地址，这是网络工程师最熟悉的部分，就不浪费篇幅解释了。这里只提一点，最后必须通过 tn.write(“exit\n”)来退出 Telnet，否则位于脚本末尾的 print tn.read_all()将失效，原因在下一步中解释。

```
tn.write("conf t\n")
tn.write("int loopback 1\n")
tn.write("ip address 1.1.1.1 255.255.255.255\n")
tn.write("end\n")
tn.write("exit\n")
```

（7）最后我们通过 Telnetlib 的 read_all()方法将登录 SW1 后执行命令的所有过程记录下来，通过 print tn.read_all()将其打印，这样就能清楚地看到 Python 对 SW1 做了什么。注：read_all()方法只有在退出 Telnet 后才会生效，因此我们必须在其之前通过 tn.write("exit\n")退出 Telnet。

```
print (tn.read_all())
```

运行代码后看结果、做验证。

（1）在主机上输入 python2 telnet.py 用 Python 2 来运行代码，如下所示。

```
[root@CentOS-Python ~]# python2 telnet.py

*************************************************************************
* IOSv is strictly limited to use for evaluation, demonstration and IOS  *
* education. IOSv is provided as-is and is not supported by Cisco's      *
* Technical Advisory Center. Any use or disclosure, in whole or in part, *
* of the IOSv Software or Documentation to any third party for any       *
* purposes is expressly prohibited except as otherwise authorized by     *
* Cisco in writing.                                                      *
*************************************************************************
SW1#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW1(config)#int loopback 1
SW1(config-if)#ip address 1.1.1.1 255.255.255.255
SW1(config-if)#end
SW1#exit

[root@CentOS-Python ~]#
```

（2）同一时间在 SW1 上可以看到 Python 通过 Telnet 登录 SW 的详细 debug 日志（之前我们已经在 SW1 开启了 debug telnet），这里能够看到“*May 2 14:00: 10.023:%LINK-3-UPDOWN: Interface Loopback1, changed state to up”“*May 2 14:00: 10.668: %SYS-5-CONFIG_I: Configured from console by python on vty0 (192.168.2.1)”和“*May 2 14:00:11.060: %LINEPROTO-5-UPDOWN: Line protocol on Interface Loopback1, changed state to up”3 条日志，表明端口 loopback 1 已经被运行 Python 的主机 192.168.2.1 创建，并且端口已经起来了。

```
SW1#
*May  2 14:00:07.396: Telnet2: 1 1 251 1
*May  2 14:00:07.397: TCP2: Telnet sent WILL ECHO (1)
*May  2 14:00:07.397: Telnet2: 2 2 251 3
*May  2 14:00:07.398: TCP2: Telnet sent WILL SUPPRESS-GA (3)
*May  2 14:00:07.398: Telnet2: 80000 80000 253 24
*May  2 14:00:07.399: TCP2: Telnet sent DO TTY-TYPE (24)
*May  2 14:00:07.399: Telnet2: 10000000 10000000 253 31
*May  2 14:00:07.400: TCP2: Telnet sent DO WINDOW-SIZE (31)
*May  2 14:00:07.466: TCP2: Telnet received DONT ECHO (1)
*May  2 14:00:07.467: TCP2: Telnet sent WONT ECHO (1)
*May  2 14:00:07.479: TCP2: Telnet received DONT SUPPRESS-GA (3)
*May  2 14:00:07.479: TCP2: Telnet sent WONT SUPPRESS-GA (3)
SW1#
*May  2 14:00:07.481: TCP2: Telnet received WONT TTY-TYPE (24)
*May  2 14:00:07.482: TCP2: Telnet sent DONT TTY-TYPE (24)
*May  2 14:00:07.484: TCP2: Telnet received WONT WINDOW-SIZE (31)
*May  2 14:00:07.486: TCP2: Telnet sent DONT WINDOW-SIZE (31)
*May  2 14:00:07.663: TCP2: Telnet received DONT ECHO (1)
*May  2 14:00:07.664: TCP2: Telnet received DONT SUPPRESS-GA (3)
*May  2 14:00:07.664: TCP2: Telnet received WONT TTY-TYPE (24)
*May  2 14:00:07.664: TCP2: Telnet received WONT WINDOW-SIZE (31)
SW1#
*May  2 14:00:10.023: %LINK-3-UPDOWN: Interface Loopback1, changed state to up
SW1#
*May  2 14:00:10.668: %SYS-5-CONFIG_I: Configured from console by python on vty0 (192.168.2.1)
*May  2 14:00:11.060: %LINEPROTO-5-UPDOWN: Line protocol on Interface Loopback1, changed state to up
SW1#
```

（3）最后在 SW1 上输入 show run int loop 1 做验证，确认端口 loopback 1 的 IP 地址配置正确，通过 Telnetlib 模块登录交换机修改配置的实验成功。

```
SW1#show run int loop 1
Building configuration...

Current configuration : 63 bytes
!
interface Loopback1
 ip address 1.1.1.1 255.255.255.255
end

SW1#
```

2. Telnetlib 在 Python 3 中的应用

在 Python 2 中，Telnetlib 模块下所有函数的返回值均为字符串，比如前面给出的代码中的 tn.read_until("Username: ")，这里的"Username: "即一个字符串。同理，tn.write(user + "\n")中的 user 虽然是一个变量，但是该变量的类型依然为字符串，后面的 tn.read_until ("Password: ")和 tn.write(password + "\n")，以及 tn.write('conf t\n')等都是同样的原理。

而在 Python 3 中，Telnetlib 模块下所有函数的返回值都变成字节型字符串（Byte

Strings）。因此，在 Python 3 中使用 Telnetlib 需要注意下面几点。

- 在字符串的前面需要加上一个 b。
- 在变量和 Telnetlib 函数后面需要加上.encode('ascii')函数。
- 在 read_all()函数后面需要加上 decode(‘ascii’)函数。

下面是在 Python 3 或 Python 3.8 中使用的 Telnetlib 脚本。

```
import telnetlib

host = "192.168.2.11"
user = "python"
password = "123"

tn = telnetlib.Telnet(host)
tn.read_until(b"Username: ")
tn.write(user.encode('ascii') + b"\n")
tn.read_until(b"Password: ")
tn.write(password.encode('ascii') + b"\n")

tn.write(b"conf t\n")
tn.write(b"int loopback 1\n")
tn.write(b"ip address 1.1.1.1 255.255.255.255\n")
tn.write(b"end\n")
tn.write(b"exit\n")

print (tn.read_all().decode('ascii'))
```

值得一提的是，这段 Python 3 的 Telnetlib 脚本在 Python 2 中依然可以正确使用。但是 Python 2 的那段脚本在 Python 3 中却无法兼容，感兴趣的读者朋友可以自行在 Python 3 或者 Python 3.8 中实验一下，如果不使用 b 和.encode(‘ascii’)函数来做字节型字符串会收到什么类型的错误。

4.2.2　Paramiko 和 Netmiko

Python 中支持 SSH 协议实现远程连接设备的模块主要有 Paramiko 和 Netmiko 两种。Paramiko 是 Python 中一个非常著名的开源 SSHv2 项目，基于 Python 2.7 和 Python 3.4+开发，于 2013 年左右发布，最早的作者是 Jeff Forcier。Paramiko 同时支持 SSH 的服务端和客户端，源码可以在 GitHub 上下载。

Netmiko 是另一个 SSH 开源项目，于 2014 年年底发布，作者是 Kirk Byers。根据作者的介绍，Netmiko 是基于 Paramiko 项目开发的，在 Paramiko 的基础上主要做了支持多厂商设备、简化命令 show 的执行和回显内容的读取、简化网络设备的配置命令等改进，具体的改进细节在后面会提到。

目前，NetDevOps 界的主流意见是 Netmiko 比 Paramiko 好用，事实上也的确如此。但是根据笔者学习和使用 Python 的经验来看，Netmiko 将太多东西简化、优化，反而不利于初学者学习。初学者如果从 Paramiko 上手，则能更直观地理解使用 Python 来 SSH 远程登录、管理设备时需要注意什么问题，更全面地锻炼自己的编程能力，并且在需要时能更顺利地在短时间内上手 Netmiko，让自己技术更全面。反之则基本是不可逆的（可以想象成从一开始就学手动挡汽车和从一开始就学自动挡汽车的两位司机的区别）。

基于上述考虑，本章和第 5 章的所有实验都将以 Paramiko 模块实现，在第 6 章中会讲解一些基于 Netmiko 的 Python 第三方模块在网络运维中的应用，这里只以一个简单的实验来讲解如何使用 Netmiko 登录 SW2，并为它的 loopback1 端口配置 IP 地址 2.2.2.2/32。

1. Netmiko 实验举例

Netmiko 为 Python 第三方模块，需要使用 pip 来下载安装，因为实验基于 Python 3.8，这里用命令 pip3.8 install Netmiko 来下载安装 Netmiko，如下图所示。

```
[root@CentOS-Python ~]# pip3.8 install netmiko
Collecting netmiko
  Downloading https://files.pythonhosted.org/packages/29/85/4c6845989486168a9a3a
d6c1c51e2a7876fc754ae4d89bd890bac4432542/netmiko-3.1.0-py2.py3-none-any.whl (144
kB)
     |████████████████████████████████| 153kB 94kB/s
Collecting scp>=0.13.2 (from netmiko)
  Downloading https://files.pythonhosted.org/packages/4d/7a/3d76dc5ad8deea79642f
50a572e1c057cb27e8b427f83781a2c05ce4e5b6/scp-0.13.2-py2.py3-none-any.whl
Collecting textfsm (from netmiko)
  Downloading https://files.pythonhosted.org/packages/bd/27/0b149b6da3e47cc8daeb
ace6920093114392171a8f5c24f1f2ad9a9e9c4d/textfsm-1.1.0-py2.py3-none-any.whl
Collecting pyserial (from netmiko)
  Downloading https://files.pythonhosted.org/packages/0d/e4/2a744dd9e3be04a0c090
7414e2a01a7c88bb3915cbe3c8cc06e209f59c30/pyserial-3.4-py2.py3-none-any.whl (193k
B)
     |████████████████████████████████| 194kB 90kB/s
Requirement already satisfied: setuptools>=38.4.0 in /usr/local/lib/python3.8/si
te-packages (from netmiko) (41.2.0)
Requirement already satisfied: paramiko>=2.4.3 in /usr/local/lib/python3.8/site-
packages (from netmiko) (2.7.1)
Collecting future (from textfsm->netmiko)
  Downloading https://files.pythonhosted.org/packages/45/0b/38b06fd9b92dc2b68d58
b75f900e97884c45bedd2ff83203d933cf5851c9/future-0.18.2.tar.gz (829kB)
     |████████████████████████████████| 829kB 75kB/s
Requirement already satisfied: six in /usr/local/lib/python3.8/site-packages (from textfsm->netmiko) (1.14.0)
Requirement already satisfied: pynacl>=1.0.1 in /usr/local/lib/python3.8/site-packages (from paramiko>=2.4.3->netmiko) (1.3.0)
Requirement already satisfied: cryptography>=2.5 in /usr/local/lib/python3.8/site-packages (from paramiko>=2.4.3->netmiko) (2.9.2)
Requirement already satisfied: bcrypt>=3.1.3 in /usr/local/lib/python3.8/site-packages (from paramiko>=2.4.3->netmiko) (3.1.7)
Requirement already satisfied: cffi>=1.4.1 in /usr/local/lib/python3.8/site-packages (from pynacl>=1.0.1->paramiko>=2.4.3->netmiko) (1.14.0)
Requirement already satisfied: pycparser in /usr/local/lib/python3.8/site-packages (from cffi>=1.4.1->pynacl>=1.0.1->paramiko>=2.4.3->netmiko) (2.20)
Installing collected packages: scp, future, textfsm, pyserial, netmiko
  Running setup.py install for future ... done
Successfully installed future-0.18.2 netmiko-3.1.0 pyserial-3.4 scp-0.13.2 textfsm-1.1.0
WARNING: You are using pip version 19.2.3, however version 20.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
[root@CentOS-Python ~]#
```

下载完毕后，进入 Python 3.8 解释器，如果 import netmiko 没有报错，则说明 Netmiko 安装成功，如下图所示。

```
[root@CentOS-Python ~]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import netmiko
>>>
```

与 Telnetlib 的实验步骤一样，首先登录 SW2，确认 loopback 1 端口当前不存在，然后启用 debug ip ssh 来监督验证下面创建的 Netmiko 脚本是否以 SSH 协议登录访问 SW2，如下图所示。

```
SW2#show run int loop1
                      ^
% Invalid input detected at '^' marker.

SW2#debug ip ssh
Incoming SSH debugging is on
SW2#
```

接下来在主机上创建一个名为 ssh_Netmiko.py 的 Python 3.8 脚本，内容如下。

```
[root@CentOS-Python ~]# cat ssh_Netmiko.py
```

```
from netmiko import ConnectHandler

SW2 = {
    'device_type': 'cisco_ios',
    'ip': '192.168.2.12',
    'username': 'python',
    'password': '123',
}

connect = ConnectHandler(**SW2)
print ("Sucessfully connected to " + SW2['ip'])
config_commands = ['int loop 1',  'ip address 2.2.2.2 255.255.255.255']
output = connect.send_config_set(config_commands)
print (output)
result = connect.send_command('show run int loop 1')
print (result)
```

代码分段讲解如下。

（1）首先通过 import 语句从 Netmiko 模块导入它的链接库函数 ConnectHandler()。该函数用来实现 SSH 登录网络设备，是 Netmiko 最重要的函数。

```
from Netmiko import ConnectHandler
```

（2）创建一个名为 SW2 的字典，该字典包含'device_type'、'ip'、'username'和'password' 4 个必选的键，其中后面 3 个键的意思很好理解，这里主要说下'device_type'。前面提到支持多厂商的设备是 Netmiko 的优势之一，截至 2019 年 1 月，Netmiko 支持 Arista、Cisco、HP、Juniper、Alcatel、Huawei 和 Extreme 等绝大多数主流厂商的设备。除此之外，Netmiko 同样支持拥有多种不同 OS 类型的厂商的设备，比如针对思科的设备，Netmiko 能同时支持 Cisco ASA、Cisco IOS、Cisco IOS-XE、Cisco IOS-XR、Cisco NX-OS 和 Cisco SG300 共 6 种不同 OS 类型的设备。由于不同厂商的设备登录 SSH 后命令行界面和特性不尽相同，因此我们必须通过'device_type'来指定需要登录的设备的类型。因为实验里我们用到的是思科 IOS 设备，因此'deivce_type'的键值为'cisco_ios'。

```
SW2 = {
```

```
        'device_type': 'cisco_ios',
        'ip': '192.168.2.12',
        'username': 'python',
        'password': '123',
    }
```

（3）调用 ConnectHandler()函数，用已经创建的字典 SW2 进行 SSH 连接，将它赋值给 connect 变量，注意 SW2 前面的**作标识用，不可以省去。

```
connect = ConnectHandler(**SW2)
```

（4）如果 SSH 登录设备成功，则提示用户并告知所登录的交换机的 IP 地址。

```
print ("Sucessfully connected to " + SW2['ip'])
```

（5）创建一个名为 config_commands 的列表，其元素为需要在交换机上依次执行的命令。

```
config_commands = ['int loop 1',  'ip address 2.2.2.2 255.255.255.255']
```

（6）然后以刚刚创建的 config_commands 列表为参数，调用 ConnectHandler()的 send_config_set()函数来使用上述命令对 SW2 做配置，并将配置过程打印出来。

```
output = connect.send_config_set(config_commands)
print (output)
```

（7）最后调用 ConnectHandler()的 send_command()函数，对交换机输入命令“show run int loop 1”并将回显内容打印出来。需要注意的是，send_command()一次只能向设备输入一条命令，而 send_config_set()则可向设备输入多条命令。

```
result = connect.send_command('show run int loop 1')
print (result)
```

运行代码后看结果、做验证。

（1）在主机上输入 python3.8 ssh_Netmiko.py 运行代码，可以看到除了我们在代码里写的 int loop 1 和 ip address 2.2.2.2 255.255.255.255 两个命令，**Netmiko 额外替我们输入了 3 个命令，一个是 config term，一个是 end，还有一个是 write memory**。最后我们能看到

show run int loop 1 命令的回显内容，证实 loopback1 端口配置成功，如下所示。

```
[root@CentOS-Python ~]# python3.8 ssh_netmiko.py
Sucessfully connected to 192.168.2.12
config term
Enter configuration commands, one per line.  End with CNTL/Z.
SW2(config)#int loop 1
SW2(config-if)#ip address 2.2.2.2 255.255.255.255
SW2(config-if)#end
SW2#

Building configuration...

Current configuration : 63 bytes
!
interface Loopback1
 ip address 2.2.2.2 255.255.255.255
end

[root@CentOS-Python ~]#
```

（2）同一时间在 SW2 上可以看到通过 Netmiko 来 SSH 登录 SW2 的详细日志（之前我们已经在 SW2 上开启了 debug ip ssh），如下所示。

```
SW2#
*May  3 05:25:13.262: SSH0: starting SSH control process
*May  3 05:25:13.262: SSH0: sent protocol version id SSH-2.0-Cisco-1.25
*May  3 05:25:13.275: SSH0: protocol version id is - SSH-2.0-paramiko_2.7.1
*May  3 05:25:13.275: SSH2 0: kexinit sent: hostkey algo = ssh-rsa
*May  3 05:25:13.276: SSH2 0: kexinit sent: encryption algo = aes128-ctr,aes192-ctr,aes256-ctr,aes128-cbc,3des-cbc,aes192
-cbc,aes256-cbc
*May  3 05:25:13.292: SSH2 0: kexinit sent: mac algo = hmac-sha1,hmac-sha1-96
*May  3 05:25:13.293: SSH2 0: send:packet of  length 368 (length also includes padlen of 5)
*May  3 05:25:13.297: SSH2 0: SSH2_MSG_KEXINIT sent
*May  3 05:25:13.298: SSH2 0: ssh_receive: 880 bytes received
*May  3 05:25:13.298: SSH2 0: input: total packet length of 880 bytes
*May  3 05:25:13.299: SSH2 0: partial packet length(block size)8 bytes,needed 872 bytes,
               maclen 0
```

（3）最后我们在 SW2 上输入 show run int loop 1 做验证，确认 loopback 1 端口的 IP 地址配置正确，通过 Netmiko 模块登录交换机修改配置的实验成功，如下所示。

```
SW2#show run int loop1
Building configuration...

Current configuration : 63 bytes
!
interface Loopback1
 ip address 2.2.2.2 255.255.255.255
end

SW2#
```

注：在使用 Netmiko 的 ConnectHandler 的脚本中，不能将脚本命名为 netmiko.py，否

则会遇到“ImportError: cannot import name 'ConnectHandler' from partially initialized module 'Netmiko' (most likely due to a circular import) (/root/Netmiko.py)”这个错误。Paramiko 也一样，脚本名字不能为 paramiko.py，如下所示。

```
[root@CentOS-Python ~]# python3.8 netmiko.py
Traceback (most recent call last):
  File "netmiko.py", line 1, in <module>
    from netmiko import ConnectHandler
  File "/root/netmiko.py", line 1, in <module>
    from netmiko import ConnectHandler
ImportError: cannot import name 'ConnectHandler' from partially initialized modu
le 'netmiko' (most likely due to a circular import) (/root/netmiko.py)
[root@CentOS-Python ~]#
```

2. Paramiko 实验举例

与 Netmiko 不同，Paramiko 不会在做配置的时候替我们自动加上 config term、end 和 write memory 等命令，也不会在执行各种 show 命令后自动保存该命令的回显内容，一切都需要我们手动搞定。另外，Python 不像人类，后者在手动输入每条命令后会间隔一定时间，再输入下一条命令。Python 是一次性执行所有脚本里的命令，中间没有间隔时间。当你要一次性输入很多条命令时，便经常会发生 SSH 终端跟不上速度，导致某些命令缺失没有被输入的问题（用传统的“复制、粘贴”方法给网络设备做配置的人应该遇到过这个问题）。同样，在用 print()函数输入回显内容或者用 open.write()将回显内容写入文档中保存时，也会因为缺乏间隔时间而导致 Python“截屏”不完整，从而导致回显内容不完整。Netmiko 自动帮我们解决了这个问题，也就是说，不管上面所举的 Netmiko 例子中 config_commands 列表中的元素（命令）有多少个，都不会出现因为间隔时间不足而导致配置命令缺失的问题。而在 Paramiko 中，我们必须导入 time 模块，使用该模块的 sleep()方法来解决这个问题，关于 time 模块和 sleep()方法的使用将在下面的实验中讲到。

下面我们用 Paramiko 来完成最后一个实验：用 Paramiko 来登录 SW3 并为它的 loopback1 端口配置 IP 地址 3.3.3.3/32。

与 Netmiko 一样，Paramiko 也是第三方模块，需要使用 pip 来下载安装，方法如下所示。

```
[root@CentOS-Python ~]# pip3.8 install paramiko
Requirement already satisfied: paramiko in /usr/local/lib/python3.8/site-package
s (2.7.1)
Requirement already satisfied: bcrypt>=3.1.3 in /usr/local/lib/python3.8/site-pa
ckages (from paramiko) (3.1.7)
Requirement already satisfied: pynacl>=1.0.1 in /usr/local/lib/python3.8/site-pa
ckages (from paramiko) (1.3.0)
Requirement already satisfied: cryptography>=2.5 in /usr/local/lib/python3.8/sit
e-packages (from paramiko) (2.9.2)
Requirement already satisfied: cffi>=1.1 in /usr/local/lib/python3.8/site-packag
es (from bcrypt>=3.1.3->paramiko) (1.14.0)
Requirement already satisfied: six>=1.4.1 in /usr/local/lib/python3.8/site-packa
ges (from bcrypt>=3.1.3->paramiko) (1.14.0)
Requirement already satisfied: pycparser in /usr/local/lib/python3.8/site-packag
es (from cffi>=1.1->bcrypt>=3.1.3->paramiko) (2.20)
WARNING: You are using pip version 19.2.3, however version 20.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
[root@CentOS-Python ~]#
```

下载完毕后，进入 Python 3.8 解释器，如果 import paramiko 没有报错，则说明 Paramiko 安装成功，如下所示。

```
[root@CentOS-Python ~]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import paramiko
>>>
```

与前面两个实验一样，首先登录 SW3，确认 loopback 1 端口目前并不存在，然后启用 debug ip ssh 来监督验证 Paramiko 脚本是否以 SSH 协议登录访问 SW3，如下所示。

```
SW3#show run int loop1
                       ^
% Invalid input detected at '^' marker.

SW3#debug ip ssh
Incoming SSH debugging is on
SW3#
```

接下来在主机上创建一个名为 ssh_paramiko.py 的 Python 3.8 脚本，内容如下。

```
import paramiko
import time

ip = "192.168.2.13"
```

```
username = "python"
password = "123"

ssh_client = paramiko.SSHClient()
ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
ssh_client.connect(hostname=ip, username=username, password=password)

print ("Successfully connected to ",  ip)
command = ssh_client.invoke_shell()
command.send("configure terminal\n")
command.send("int loop 1\n")
command.send("ip address 3.3.3.3 255.255.255.255\n")
command.send("end\n")
command.send("wr mem\n")

time.sleep(2)
output = command.recv(65535)
print (output.decode("ascii"))

ssh_client.close
```

代码分段讲解如下。

（1）首先通过 import 语句导入 paramiko 和 time 两个模块。

```
import paramiko
import time
```

（2）创建 3 个变量：ip、username 和 password，分别对应我们要登录的交换机（SW3）的管理 IP 地址、SSH 用户名和密码。

```
ip = "192.168.2.13"
username = "python"
password = "123"
```

（3）调用 Paramiko 的 SSHClient()方法，将其赋值给变量 ssh_client。顾名思义，CentOS 主机做 SSH 客户端，而 SSH 服务端则是我们要登录的 SW3（192.168.2.13）。

```
ssh_client = paramiko.SSHClient()
```

（4）默认情况下，Paramiko 会拒绝任何未知的 SSH 公匙，这里我们需要使用 ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy()) 来让 Paramiko 接受来自 SSH 服务端（也就是 SW3）提供的公匙，这是任何时候使用 Paramiko 都要用到的标准配置。

```
ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
```

（5）在做完 Paramiko 关于 SSH 公匙相关的配置后，调用 Paramiko.SSHClient()的 connect()函数进行 SSH 登录，该函数包含 3 个必选的参数 hostname、username 和 password，分别对应我们创建的 ip、username 和 password 3 个变量，也就是远程登录的设备的主机名/IP 地址、SSH 用户名和密码。如果 SSH 登录设备成功，则提示用户并告知所登录的交换机的管理 IP 地址。

```
ssh_client.connect (hostname=ip, username=username, password=password)
print ("Successfully connected to ", ip)
```

（6）SSH 连接成功后，需要调用 Paramiko.SSHClient()的 invoke_shell()方法来唤醒 shell，也就是思科设备 IOS 命令行，并将它赋值给变量 command。

```
command = ssh_client.invoke_shell()
```

（7）之后便可以调用 invoke_shell()的 command()函数来向 SW3“发号施令”了。这里注意需要手动在代码中输入 configure terminal、end 和 wr mem 3 个命令，这一点与 Netmiko 不同。

```
command.send("configure terminal\n")
command.send("int loop 1\n")
command.send("ip address 3.3.3.3 255.255.255.255\n")
command.send("end\n")
command.send("wr mem\n")
```

（8）前面讲到，Python 是一次性执行所有脚本里的命令，中间没有间隔时间，这样会导致某些命令遗漏和回显内容不完整的问题。在用 Paramiko 的 recv()函数将回显结果保存

之前，我们需要调用 time 模块下的 sleep()函数手动让 Python 休眠 2s，这样回显内容才能被完整打印出来（sleep()中参数的单位为 s）。这里的 command.recv(65535)中的 65535 代表截取 65535 个字符的回显内容，这也是 Paramiko 一次能截取的最大回显内容数。另外，与 Telnetlib 类似，在 Python 3 中，Paramiko 截取的回显内容格式为字节型字符串，需要用 decode("ascii")将其解析为 ASCII 编码，否则打印出来的 output 的内容格式会很难看（下面验证部分会给出不带 decode("ascii")的回显内容供参考）。

```
time.sleep(2)
output = command.recv(65535)
print (output.decode("ascii"))
```

（9）配置完毕后，使用 close 方法退出 SSH。

```
ssh_client.close
```

运行代码后看结果、做验证。

（1）在主机上输入 python ssh_paramiko.py 来运行代码，可以看到脚本提示登录 192.168.2.13（SW3）成功，然后给 loopback1 端口配置 IP 地址 3.3.3.3，最后保存配置。代码运行如下图所示。

```
[root@CentOS-Python ~]# python3.8 ssh_paramiko.py
Successfully connected to  192.168.2.13

*************************************************************************
* IOSv is strictly limited to use for evaluation, demonstration and IOS  *
* education. IOSv is provided as-is and is not supported by Cisco's      *
* Technical Advisory Center. Any use or disclosure, in whole or in part, *
* of the IOSv Software or Documentation to any third party for any       *
* purposes is expressly prohibited except as otherwise authorized by     *
* Cisco in writing.                                                      *
*************************************************************************SW3#configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
SW3(config)#int loop 1
SW3(config-if)#ip address 3.3.3.3 255.255.255.255
SW3(config-if)#end
SW3#wr mem
Building configuration...

[root@CentOS-Python ~]#
```

上面提到，如果我们不在 print (output)后面加上 decode("ascii")会怎样呢？如下图所示。

```
[root@CentOS-Python ~]# python3.8 ssh_paramiko.py
Successfully connected to  192.168.2.13
b"\r\n*******************************************************************
\r\n* IOSv is strictly limited to use for evaluation, demonstration and IOS  *\r
\n* education. IOSv is provided as-is and is not supported by Cisco's      *\r\n
* Technical Advisory Center. Any use or disclosure, in whole or in part, *\r\n*
of the IOSv Software or Documentation to any third party for any       *\r\n* pu
rposes is expressly prohibited except as otherwise authorized by     *\r\n* Cisc
o in writing.                                                       *\r\n********
*******************************************************************\r\nSW3#config
ure terminal\r\nEnter configuration commands, one per line.  End with CNTL/Z.\r\
nSW3(config)#int l"
```

是不是很乱？根本不知道脚本在做什么。

注：如果在 CentOS 中运行代码时遇到 SSH 用户名和密码都正确，但是一直出现“Authentication failed”验证失败的异常提示（如下所示），很可能是主机之前用 ssh-keygen 命令生成过 RSA 密匙对用来做 SSH 免密码登录（比如在使用 Gitlab 时可以用到）。一旦在 CentOS 上生成了本地 RSA 密匙对，Paramiko 就会一直尝试使用该密匙对来登录设备，原因是默认情况下 Paramiko.SSHClient().connect()中的 look_for_keys 可选参数默认为 True。通常情况下，我们登录交换机等网络设备时不是 SSH 免密登录，没有用到 RSA 密匙对，但是 Paramiko 又一直尝试使用该密匙对来登录设备，导致 Authentication failed 验证失败。

```
[root@localhost ~]# python ssh_paramiko.py
Traceback (most recent call last):
  File "ssh_paramiko.py", line 10, in <module>
    ssh_client.connect(hostname=ip,username=username,password=password,look_for_keys=True)
  File "/usr/lib/python2.7/site-packages/paramiko/client.py", line 437, in connect
    passphrase,
  File "/usr/lib/python2.7/site-packages/paramiko/client.py", line 749, in _auth
    raise saved_exception
paramiko.ssh_exception.AuthenticationException: Authentication failed.
[root@localhost ~]#
```

在交换机上开启 debug ip ssh 后，能看到用户名 python 的公匙（publickey）缺失，导致验证失败的日志记录，如下所示。

```
*Mar 25 06:52:18.152: SSH2 0: Using method = publickey
*Mar 25 06:52:18.153: S
S3#SH2 0: Publickey for 'python' not found
*Mar 25 06:52:18.153: SSH2 0: Pubkey Authentication failed for user 'python'
*Mar 25 06:52:18.153: SSH0: password authentication failed for python
*Mar 25 06:52:20.153: SSH2 0: Authentications that can continue = publickey,keyboard-interactive,password
*Mar 25 06:52:20.153: SSH2 0: send:packet of  length 64 (length also includes padlen of 14)
*Mar 25 06:52:20.153: SSH2 0: computed MAC for sequence no.#6 type 51
*Mar 25 06:52:20.169: SSH2 0: ssh_receive: 52 bytes received
*Mar 25 06:52:20.169: SSH2 0: input: total packet length of 32 bytes
*Mar 25 06:52:20.169: SSH2 0: partial packet length(block size)16 bytes,needed 16 bytes,
                maclen 20
*Mar 25 06:52:20.170: SSH2 0: MAC compared for #6 :ok
*Mar 25 06:52:20.170: SSH2 0: input: padlength 10 bytes
S3#
*Mar 25 06:52:20.171: SSH2 0: send:packet of  length 80 (length also includes padlen of 16)
*Mar 25 06:52:20.171: SSH2 0: computed MAC for sequence no.#7 type 1
*Mar 25 06:52:20.274: SSH0: Session disconnected - error 0x07
S3#
```

解决办法也很简单，将 look_for_keys 参数修改为 False 即可。

```
    ssh_client.connect(hostname=ip,  username=username, password=password,
look_for_keys=False)
```

（2）同一时间在 SW3 上可以看到我们通过 Python 来 SSH 登录 SW3 的详细日志（之前我们已经在 SW3 开启了 debug ip ssh），如下图所示。

```
SW3#debug ip ssh
Incoming SSH debugging is on
SW3#
*May  3 05:48:26.936: SSH0: starting SSH control process
*May  3 05:48:26.937: SSH0: sent protocol version id SSH-2.0-Cisco-1.25
*May  3 05:48:26.973: SSH0: protocol version id is - SSH-2.0-paramiko_2.7.1
*May  3 05:48:26.975: SSH2 0: kexinit sent: hostkey algo = ssh-rsa
*May  3 05:48:26.976: SSH2 0: kexinit sent: encryption algo = aes128-ctr,aes192-ctr,aes256-ctr,aes128-cbc,3des-cbc,aes192
-cbc,aes256-cbc
*May  3 05:48:26.977: SSH2 0: kexinit sent: mac algo = hmac-sha1,hmac-sha1-96
*May  3 05:48:26.979: SSH2 0: send:packet of  length 368 (length also includes padlen of 5)
*May  3 05:48:26.989: SSH2 0: SSH2_MSG_KEXINIT sent
*May  3 05:48:26.991: SSH2 0: ssh_receive: 880 bytes received
*May  3 05:48:26.992: SSH2 0: input: total packet length of 880 bytes
*May  3 05:48:26.993: SSH2 0: partial packet length(block size)8 bytes,needed 872 bytes,
               maclen 0
*May  3 05:48:26.993: SSH2 0: input: padlength 10 bytes
```

（3）最后我们在 SW3 上输入 show run int loop 1 做验证，确认 loopback 1 端口的 IP 地址配置正确，通过 Paramiko 模块登录交换机修改配置的实验成功，如下图所示。

```
SW3#show run int loop1
Building configuration...

Current configuration : 63 bytes
!
interface Loopback1
 ip address 3.3.3.3 255.255.255.255
end

SW3#
```

在举例介绍了 Telnetlib、Netmiko 和 Paramiko 的基本使用方法后，下面将正式进入实验部分，每个实验都将以网络运维中常见的需求为实验背景，讲解怎样用 Python 来实现这些需求的自动化（所有实验都将使用 Paramiko 实现）。

4.3　实验 1　SSH 用户名和密码手动输入

在前面针对 Telnetlib、Netmiko 和 Paramiko 模块的基础讲解中，我们都将 SSH 登录的用户名和密码明文写在了脚本里，这种做法在实验练习中可以使用，但是在生产环境中肯定是不够安全的。在生产环境中正确的做法是使用 input()函数和 getpass 模块来分别提示用户手动输入 SSH 用户名和密码，这是本实验将重点讲解的部分。

4.3.1　实验目的

- 使用 input()函数和 getpass 模块实现交互式的 SSH 用户名和密码输入。
- 通过 for 循环同时给 5 台交换机 SW1～SW5 配置 VLAN 10～VLAN 20。

4.3.2　实验准备

- 运行代码前，首先检查 5 台交换机的配置，确认它们都没有 VLAN 10～VLAN 20。

运行代码前，SW1 的配置如下所示。

```
SW1#show vlan b

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Gi0/0, Gi0/1, Gi0/2, Gi0/3
                                                Gi1/0, Gi1/1, Gi1/2, Gi1/3
                                                Gi2/0, Gi2/1, Gi2/2, Gi2/3
                                                Gi3/0, Gi3/1, Gi3/2, Gi3/3
1002 fddi-default                     act/unsup
1003 token-ring-default               act/unsup
1004 fddinet-default                  act/unsup
1005 trnet-default                    act/unsup
SW1#
```

运行代码前，SW2 的配置如下所示。

```
SW2#show vlan b

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Gi0/0, Gi0/1, Gi0/2, Gi0/3
                                                Gi1/0, Gi1/1, Gi1/2, Gi1/3
                                                Gi2/0, Gi2/1, Gi2/2, Gi2/3
                                                Gi3/0, Gi3/1, Gi3/2, Gi3/3
1002 fddi-default                     act/unsup
1003 token-ring-default               act/unsup
1004 fddinet-default                  act/unsup
1005 trnet-default                    act/unsup
SW2#
```

运行代码前，SW3 的配置如下所示。

```
SW3#show vlan b

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Gi0/0, Gi0/1, Gi0/2, Gi0/3
                                                Gi1/0, Gi1/1, Gi1/2, Gi1/3
                                                Gi2/0, Gi2/1, Gi2/2, Gi2/3
                                                Gi3/0, Gi3/1, Gi3/2, Gi3/3
1002 fddi-default                     act/unsup
1003 token-ring-default               act/unsup
1004 fddinet-default                  act/unsup
1005 trnet-default                    act/unsup
SW3#
```

运行代码前，SW4 的配置如下所示。

```
SW4#show vlan b

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Gi0/0, Gi0/1, Gi0/2, Gi0/3
                                                Gi1/0, Gi1/1, Gi1/2, Gi1/3
                                                Gi2/0, Gi2/1, Gi2/2, Gi2/3
                                                Gi3/0, Gi3/1, Gi3/2, Gi3/3
1002 fddi-default                     act/unsup
1003 token-ring-default               act/unsup
1004 fddinet-default                  act/unsup
1005 trnet-default                    act/unsup
SW4#
```

运行代码前，SW5 的配置如下所示。

```
SW5#show vlan b

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Gi0/0, Gi0/1, Gi0/2, Gi0/3
                                                Gi1/0, Gi1/1, Gi1/2, Gi1/3
                                                Gi2/0, Gi2/1, Gi2/2, Gi2/3
                                                Gi3/0, Gi3/1, Gi3/2, Gi3/3
1002 fddi-default                     act/unsup
1003 token-ring-default               act/unsup
1004 fddinet-default                  act/unsup
1005 trnet-default                    act/unsup
SW5#
```

（2）在主机上创建实验 1 的 Python 脚本，将其命名为 lab1.py。

```
[root@CentOS-Python ~]# vi lab1.py
```

4.3.3　实验代码

将下列代码写入脚本 lab1.py。

```
import paramiko
import time
import getpass

username = input('Username: ')
```

```
password = getpass.getpass('Password: ')

for i in range(11,16):
    ip = "192.168.2." + str(i)
    ssh_client = paramiko.SSHClient()
    ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh_client.connect(hostname=ip,username=username,password=password)
    print ("Successfully connect to ", ip)
    command = ssh_client.invoke_shell()
    command.send("configure terminal\n")
    for n in range (10,21):
        print ("Creating VLAN " + str(n))
        command.send("vlan " + str(n) +  "\n")
        command.send("name Python_VLAN " + str(n) +  "\n")
        time.sleep(1)

    command.send("end\n")
    command.send("wr mem\n")
    time.sleep(2)
    output = command.recv(65535)
    print (output.decode('ascii'))

ssh_client.close
```

4.3.4 代码分段讲解

（1）首先导入 paramiko、time 和 getpass 3 种模块。前两种模块的用法之前已经讲过，这里讲下 getpass 模块。getpass 是 Python 的内建模块，无须通过 pip 下载安装即可使用。它和 input()函数一样，都是 Python 的交互式功能，用来提示用户输入密码，区别是如果用 input()输入密码，用户输入的密码是明文可见的，如果你身边坐了其他人，密码就这么暴露了。而通过 getpass 输入的密码则是隐藏不可见的，安全性很高，所以强烈建议使用 getpass 来输入密码，使用 input()来输入用户名。注：getpass 在 Windows 中有 bug，输出

的密码依然明文可见，但是不影响脚本的运行。

```
import Paramiko
import time
import getpass

username = input('Username: ')
password = getpass.getpass('Password: ')
```

（2）因为这里 5 个交换机 SW1～SW5 的 IP 地址是连续的（192.168.2.11-15），这样我们可以配合 for i in range(11，16)做一个简单的 for 循环来遍历 11～15 的迭代值（记住在 Python 3 中，range()函数的返回值不再是列表，而是一组迭代值），然后以此配合下一行代码 ip = "192.168.2." + str(i)来实现循环（批量）登录交换机 SW1～SW5。注意：这里的 i 是整数，整数不能和字符串做拼接，所以要用 str(i)先将 i 转化成字符串。

```
for i in range(11,16):
    ip = "192.168.2." + str(i)
    ssh_client = Paramiko.SSHClient()
    ssh_client.set_missing_host_key_policy(Paramiko.AutoAddPolicy())
    ssh_client.connect(hostname=ip, username=username, password=password)
    print ("Successfully connect to ", ip)
    command = ssh_client.invoke_shell()
    command.send("configure terminal\n")
```

（3）同样的道理，我们要创建 VLAN 10 ～ VLAN 20，这些 VLAN Id 是连续的，所以这里又可以配合一个简单的 for 循环 for n in range (10，21)来达到循环配置 VLAN10～20 的目的，这里用到的是嵌套 for 循环，需要注意缩进。每创建一个 VLAN 之前，都打印内容“print ("Creating VLAN " + str(n))”来提示用户当前正在创建的 VLAN。每个 VLAN 的命名格式都是 Python_VLAN XX，比如 VLAN 10 的名字是 Python_VLAN 10，VLAN 11 的名字是 Python_VLAN 11，依此类推。每创建一个 VLAN 之间都需要 1s 的间隔。

```
    for n in range (10, 21):
    print ("Creating VLAN " + str(n))
    command.send("vlan " + str(n) + "\n")
```

```
        command.send("name Python_VLAN " + str(n) +  "\n")
        time.sleep(1)
```

（4）最后保存配置，间隔 2s 后打印出回显内容并关闭 SSH。

```
        command.send("end\n")
        command.send("wr mem\n")
        time.sleep(2)
        output = command.recv(65535)
        print (output)

    ssh_client.close
```

4.3.5 验证

（1）因打印出的回显内容过长，这里只截取自动登录 SW1、SW2 和 SW3 做配置的部分代码。可以看到：当运行脚本后系统提示输入用户名和密码时，我们输入的用户名是可见的，而密码是不可见的，原因就是输入用户名时我们使用的是 input()，输入密码时我们用的是 getpass.getpass()，如下所示。

```
[root@CentOS-Python ~]# python3.8 lab1.py
Username: python
Password:
Successfully connect to  192.168.2.11
Creating VLAN 10
Creating VLAN 11
Creating VLAN 12
Creating VLAN 13
Creating VLAN 14
Creating VLAN 15
Creating VLAN 16
Creating VLAN 17
Creating VLAN 18
Creating VLAN 19
Creating VLAN 20

**************************************************************************
* IOSv is strictly limited to use for evaluation, demonstration and IOS  *
* education. IOSv is provided as-is and is not supported by Cisco's      *
* Technical Advisory Center. Any use or disclosure, in whole or in part, *
* of the IOSv Software or Documentation to any third party for any       *
* purposes is expressly prohibited except as otherwise authorized by     *
* Cisco in writing.                                                      *
**************************************************************************SW1#configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
SW1(config)#vlan 10
SW1(config-vlan)#name Python_VLAN 10
SW1(config-vlan)#vlan 11
SW1(config-vlan)#name Python_VLAN 11
SW1(config-vlan)#vlan 12
SW1(config-vlan)#name Python_VLAN 12
SW1(config-vlan)#vlan 13
SW1(config-vlan)#name Python_VLAN 13
SW1(config-vlan)#vlan 14
SW1(config-vlan)#name Python_VLAN 14
SW1(config-vlan)#vlan 15
SW1(config-vlan)#name Python_VLAN 15
SW1(config-vlan)#vlan 16
SW1(config-vlan)#name Python_VLAN 16
SW1(config-vlan)#vlan 17
SW1(config-vlan)#name Python_VLAN 17
SW1(config-vlan)#
Successfully connect to  192.168.2.12
Creating VLAN 10
Creating VLAN 11
Creating VLAN 12
Creating VLAN 13
Creating VLAN 14
Creating VLAN 15
Creating VLAN 16
```

```
Creating VLAN 17
Creating VLAN 18
Creating VLAN 19
Creating VLAN 20

**************************************************************************
* IOSv is strictly limited to use for evaluation, demonstration and IOS  *
* education. IOSv is provided as-is and is not supported by Cisco's      *
* Technical Advisory Center. Any use or disclosure, in whole or in part, *
* of the IOSv Software or Documentation to any third party for any       *
* purposes is expressly prohibited except as otherwise authorized by     *
* Cisco in writing.                                                      *
**************************************************************************
SW2#configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
SW2(config)#vlan 10
SW2(config-vlan)#name Python_VLAN 10
SW2(config-vlan)#vlan 11
SW2(config-vlan)#name Python_VLAN 11
SW2(config-vlan)#vlan 12
SW2(config-vlan)#name Python_VLAN 12
SW2(config-vlan)#vlan 13
SW2(config-vlan)#name Python_VLAN 13
SW2(config-vlan)#vlan 14

Successfully connect to  192.168.2.13
Creating VLAN 10
Creating VLAN 11
Creating VLAN 12
Creating VLAN 13
Creating VLAN 14
Creating VLAN 15
Creating VLAN 16
Creating VLAN 17
Creating VLAN 18
Creating VLAN 19
Creating VLAN 20

**************************************************************************
* IOSv is strictly limited to use for evaluation, demonstration and IOS  *
* education. IOSv is provided as-is and is not supported by Cisco's      *
* Technical Advisory Center. Any use or disclosure, in whole or in part, *
* of the IOSv Software or Documentation to any third party for any       *
* purposes is expressly prohibited except as otherwise authorized by     *
* Cisco in writing.                                                      *
**************************************************************************
SW3#configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
SW3(config)#vlan 10
SW3(config-vlan)#name Python_VLAN 10
```

（2）依次登录 5 个交换机验证配置。

运行代码后，SW1 的配置如下所示。

```
SW1#show vlan b

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Gi0/0, Gi0/1, Gi0/2, Gi0/3
                                                Gi1/0, Gi1/1, Gi1/2, Gi1/3
                                                Gi2/0, Gi2/1, Gi2/2, Gi2/3
                                                Gi3/0, Gi3/1, Gi3/2, Gi3/3
10   Python_VLAN 10                   active
11   Python_VLAN 11                   active
12   Python_VLAN 12                   active
13   Python_VLAN 13                   active
14   Python_VLAN 14                   active
15   Python_VLAN 15                   active
16   Python_VLAN 16                   active
17   Python_VLAN 17                   active
18   Python_VLAN 18                   active
19   Python_VLAN 19                   active
20   Python_VLAN 20                   active
1002 fddi-default                     act/unsup
1003 token-ring-default               act/unsup
1004 fddinet-default                  act/unsup
1005 trnet-default                    act/unsup
SW1#
```

运行代码后，SW2 的配置如下所示。

```
SW2#show vlan b

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Gi0/0, Gi0/1, Gi0/2, Gi0/3
                                                Gi1/0, Gi1/1, Gi1/2, Gi1/3
                                                Gi2/0, Gi2/1, Gi2/2, Gi2/3
                                                Gi3/0, Gi3/1, Gi3/2, Gi3/3
10   Python_VLAN 10                   active
11   Python_VLAN 11                   active
12   Python_VLAN 12                   active
13   Python_VLAN 13                   active
14   Python_VLAN 14                   active
15   Python_VLAN 15                   active
16   Python_VLAN 16                   active
17   Python_VLAN 17                   active
18   Python_VLAN 18                   active
19   Python_VLAN 19                   active
20   Python_VLAN 20                   active
1002 fddi-default                     act/unsup
1003 token-ring-default               act/unsup
1004 fddinet-default                  act/unsup
1005 trnet-default                    act/unsup
SW2#
```

运行代码后，SW3 的配置如下所示。

```
SW3#show vlan b

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Gi0/0, Gi0/1, Gi0/2, Gi0/3
                                                Gi1/0, Gi1/1, Gi1/2, Gi1/3
                                                Gi2/0, Gi2/1, Gi2/2, Gi2/3
                                                Gi3/0, Gi3/1, Gi3/2, Gi3/3
10   Python_VLAN 10                   active
11   Python_VLAN 11                   active
12   Python_VLAN 12                   active
13   Python_VLAN 13                   active
14   Python_VLAN 14                   active
15   Python_VLAN 15                   active
16   Python_VLAN 16                   active
17   Python_VLAN 17                   active
18   Python_VLAN 18                   active
19   Python_VLAN 19                   active
20   Python_VLAN 20                   active
1002 fddi-default                     act/unsup
1003 token-ring-default               act/unsup
1004 fddinet-default                  act/unsup
1005 trnet-default                    act/unsup
SW3#
```

运行代码后，SW4 的配置如下所示。

```
SW4#show vlan b

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Gi0/0, Gi0/1, Gi0/2, Gi0/3
                                                Gi1/0, Gi1/1, Gi1/2, Gi1/3
                                                Gi2/0, Gi2/1, Gi2/2, Gi2/3
                                                Gi3/0, Gi3/1, Gi3/2, Gi3/3
10   Python_VLAN 10                   active
11   Python_VLAN 11                   active
12   Python_VLAN 12                   active
13   Python_VLAN 13                   active
14   Python_VLAN 14                   active
15   Python_VLAN 15                   active
16   Python_VLAN 16                   active
17   Python_VLAN 17                   active
18   Python_VLAN 18                   active
19   Python_VLAN 19                   active
20   Python_VLAN 20                   active
1002 fddi-default                     act/unsup
1003 token-ring-default               act/unsup
1004 fddinet-default                  act/unsup
1005 trnet-default                    act/unsup
SW4#
```

运行代码后，SW5 的配置如下所示。

```
SW5#show vlan b

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Gi0/0, Gi0/1, Gi0/2, Gi0/3
                                                Gi1/0, Gi1/1, Gi1/2, Gi1/3
                                                Gi2/0, Gi2/1, Gi2/2, Gi2/3
                                                Gi3/0, Gi3/1, Gi3/2, Gi3/3
10   Python_VLAN 10                   active
11   Python_VLAN 11                   active
12   Python_VLAN 12                   active
13   Python_VLAN 13                   active
14   Python_VLAN 14                   active
15   Python_VLAN 15                   active
16   Python_VLAN 16                   active
17   Python_VLAN 17                   active
18   Python_VLAN 18                   active
19   Python_VLAN 19                   active
20   Python_VLAN 20                   active
1002 fddi-default                     act/unsup
1003 token-ring-default               act/unsup
1004 fddinet-default                  act/unsup
1005 trnet-default                    act/unsup
SW5#
```

4.4　实验 2　批量登录所有交换机

在生产环境中，交换机的管理 IP 地址基本不可能像实验 1 中那样 11～15 是连续的，有些交换机的管理 IP 地址甚至在不同的网段。在这种情况下，我们就不能简单地用 for 循环来登录交换机了。我们要额外建立一个文本文件，把我们需要登录的交换机的管理 IP 地址全部写进去，然后用 for 循环配合 open()函数来读取该文档中的管理 IP 地址，从而达到批量登录交换机的目的。

4.4.1　实验目的

- 通过 Python 脚本批量登录所有交换机，并在每个交换机上都开启 EIGRP。

4.4.2 实验准备

（1）把 SW5 的管理地址从 192.168.2.15 改成 192.168.2.55。

```
S5#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
S5(config)#int vlan 1
S5(config-if)#ip add 192.168.2.55 255.255.255.0
S5(config-if)#end
S5#
*Mar 26 07:03:04.886: %SYS-5-CONFIG_I: Configured from console by console
S5#
```

（2）在 CentOS 上创建一个名为 ip_list 的 TXT 文件，把所有交换机的管理 IP 地址都放进去，注意该文件和等下我们要创建的脚本位于同一个文件夹下，如下所示。

```
[root@CentOS-Python ~]# cat ip_list.txt
192.168.2.11
192.168.2.12
192.168.2.13
192.168.2.14
192.168.2.55
[root@CentOS-Python ~]#
```

（3）运行代码前，首先检查 5 台交换机的配置，确认它们都没有开启 EIGRP。

运行代码前，SW1 的配置如下所示。

```
SW1#show run | s router eigrp
SW1#
```

运行代码前，SW2 的配置如下所示。

```
SW2#show run | s router eigrp
SW2#
```

运行代码前，SW3 的配置如下所示。

```
SW3#show run | s router eigrp
SW3#
```

运行代码前，SW4 的配置如下所示。

```
SW4#show run | s router eigrp
SW4#
```

运行代码前，SW5 的配置如下所示。

```
SW5#show run | s router eigrp
SW5#
```

（4）在主机上创建实验 2 的脚本，将其命名为 lab2.py，如下所示。

```
[root@CentOS-Python ~]# vi lab2.py
```

4.4.3　实验代码

将下列代码写入脚本 lab2.py。

```
import paramiko
import time
from getpass import getpass

username = input('Username: ')
password = getpass('password: ')

f = open("ip_list.txt", "r")
for line in f.readlines():
    ip = line.strip()
    ssh client = paramiko.SSHClient()
    ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh_client.connect(hostname=ip, username=username, password=password)
    print ("Successfully connect to ", ip)
    remote_connection = ssh_client.invoke_shell()
    remote_connection.send("conf t\n")
    remote_connection.send("router eigrp 1\n")
    remote_connection.send("end\n")
    remote_connection.send("wr mem\n")
```

```
        time.sleep(1)
        output = remote_connection.recv(65535)
        print (output.decode("ascii"))

    f.close()
    ssh_client.close
```

4.4.4 代码分段讲解

（1）和实验 1 稍有不同，我们在导入 getpass 模块时用的是 from getpass import getpass，因此下面我们可以把 getpass.getpass('password: ')简写成 getpass('password: ')。

```
import paramiko
import time
from getpass import getpass

username = input('Username: ')
password = getpass('password: ')
```

（2）用 open()函数打开之前我们创建好的包含 5 个交换机的管理 IP 地址的文档（ip_list.txt），通过 for 循环来依次遍历 readlines()方法返回的列表中的每个元素（即每个交换机的管理 IP 地址），即可达到批量依次登录 SW1～SW5 的目的，即使这 5 个交换机的管理 IP 地址不是连续的。

```
f = open("ip_list.txt", "r")
for line in f.readlines():
    ip = line.strip()
    ssh_client = paramiko.SSHClient()
    ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh_client.connect(hostname=ip, username=username, password=password)
    print ("Successfully connect to ", ip)
```

（3）登录每台交换机后配置 EIGRP，将回显内容打印出来。

```
    remote_connection = ssh_client.invoke_shell()
    remote_connection.send("conf t\n")
    remote_connection.send("router eigrp 1\n")
    remote_connection.send("end\n")
    remote_connection.send("wr mem\n")
    time.sleep(1)
    output = remote_connection.recv(65535)
    print (output.decode("ascii"))
```

（4）文件有开有关，脚本结束前用 close()关掉 ip_list.txt 文档，并且关闭 SSH 链接。

```
f.close()
ssh_client.close
```

4.4.5　验证

（1）因打印出的回显内容过长，这里只截取自动登录 SW1、SW2 做配置的部分代码，可以看见代码中自动登录了每个交换机开启 EIGRP 并保存配置，随后退出，如下所示。

```
[root@CentOS-Python ~]# python3.8 lab2.py
Username: python
password:
Successfully connect to  192.168.2.11

**************************************************************************
* IOSv is strictly limited to use for evaluation, demonstration and IOS  *
* education. IOSv is provided as-is and is not supported by Cisco's      *
* Technical Advisory Center. Any use or disclosure, in whole or in part, *
* of the IOSv Software or Documentation to any third party for any       *
* purposes is expressly prohibited except as otherwise authorized by     *
* Cisco in writing.                                                      *
**************************************************************************
SW1#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW1(config)#router eigrp 1
SW1(config-router)#end
SW1#wr mem
Building configuration...

Successfully connect to  192.168.2.12

**************************************************************************
* IOSv is strictly limited to use for evaluation, demonstration and IOS  *
* education. IOSv is provided as-is and is not supported by Cisco's      *
* Technical Advisory Center. Any use or disclosure, in whole or in part, *
* of the IOSv Software or Documentation to any third party for any       *
* purposes is expressly prohibited except as otherwise authorized by     *
* Cisco in writing.                                                      *
**************************************************************************
SW2#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW2(config)#router eigrp 1
SW2(config-router)#end
SW2#wr mem
Building configuration...
```

（2）依次登录 5 个交换机验证配置。

运行代码后，SW1 的配置如下所示。

```
SW1#show run | s router eigrp
router eigrp 1
SW1#
```

运行代码后，SW2 的配置如下所示。

```
SW2#show run | s router eigrp
router eigrp 1
SW2#
```

运行代码后，SW3 的配置如下所示。

```
SW3#show run | s router eigrp
router eigrp 1
SW3#
```

运行代码后，SW4 的配置如下所示。

```
SW4#show run | s router eigrp
router eigrp 1
SW4#
```

运行代码后，SW5 的配置如下所示。

```
SW5#show run | s router eigrp
router eigrp 1
SW5#
```

4.5　实验 3　异常处理的应用

在网络设备数量超过千台甚至上万台的大型企业网中，难免会遇到某些设备的管理 IP 地址不通、SSH 连接失败的情况，设备数量越多，这种情况发生的概率越大。这个时候如果你想用 Python 批量配置所有的设备，就一定要注意这种情况，很可能你的脚本运行了还不到一半就因为中间某一个连接不通的设备而停止了。比如说你有 5 000 台交换机需要统一更改本地用户名和密码，前 500 台交换机的连通性都没问题，第 501 台交换机因为某个网络问题导致管理 IP 地址不可达，SSH 连不上，此时 Python 会返回一个“socket.error: [Errno 10060] A connection attempt failed because the connected party did not properly respond after a period of time，or established connection failed because connected host has failed to respond”的错误，然后脚本就此停住！脚本不会再对剩下的 4 500 台交换机做配置，也就意味着“挂机”失败！如下所示。

```
Traceback (most recent call last):
  File "Verify configuration violation.py", line 23, in <module>
    ssh_client.connect(hostname=ip,username=username,password=password,look_for_
keys=False)
  File "C:\Python27\lib\site-packages\paramiko\client.py", line 338, in connect
    retry_on_signal(lambda: sock.connect(addr))
  File "C:\Python27\lib\site-packages\paramiko\util.py", line 279, in retry_on_s
ignal
    return function()
  File "C:\Python27\lib\site-packages\paramiko\client.py", line 338, in <lambda>

    retry_on_signal(lambda: sock.connect(addr))
  File "C:\Python27\lib\socket.py", line 228, in meth
    return getattr(self._sock,name)(*args)
socket.error: [Errno 10060] A connection attempt failed because the connected pa
rty did not properly respond after a period of time, or established connection f
ailed because connected host has failed to respond
```

知乎 @弈心

同样的问题也会发生在当你输入了错误的交换机用户名和密码之后，或者某些交换机和其他大部分交换机用户名和密码不一致的时候（因为我们只能输入一次用户名和密码，用户名和密码不一致会导致个别交换机无法登录的情况发生），也许你会问大型企业网不都是统一配置 AAA 配合 TACACS 或者 RADIUS 做用户访问管理吗？怎么还会出现登录账号和密码不一致的问题？这个现象就发生在笔者目前所任职的沙特阿卜杜拉国王科技大学，学校里的 TACACS 服务器（思科 ACS）已经服役 9 年，当前的问题是每天早晨 8 点左右该 ACS 会“失效”，必须手动重启 ACS 所在的服务器才能解决问题，在 ACS 无法正常工作期间，我们只能通过网络设备的本地账号和密码登录，鉴于此，我们已经部署了思科的 ISE 来替代 ACS 做 TACACS 服务器，但由于学校网络过于庞大，迁徙过程漫长，就导致了部分设备已经迁徙，使用上了 ISE 配置的账号和密码；而另一部分还没有迁徙的设备在 ACS 出问题的时候只能用本地的账号和密码，这就出现了两套账号和密码的情况，后果就是使用 Paramiko 来 SSH 登录网络设备的 Python 会返回“Paramiko.ssh_exception.AuthenticationException: Authentication failed”的错误，如下所示，导致脚本戛然而止，无法继续运行。

```
Traceback (most recent call last):
  File "Verify configuration violation.py", line 25, in <module>
    ssh_client.connect(hostname=ip,username=username,password=password,look_for_
keys=False)
  File "C:\Python27\lib\site-packages\paramiko\client.py", line 424, in connect
    passphrase,
  File "C:\Python27\lib\site-packages\paramiko\client.py", line 714, in _auth
    raise saved_exception
paramiko.ssh_exception.AuthenticationException: Authentication failed.
```

知乎 @弈心

解决上述两个问题的方法也许你也想到了，就是使用我们在第 3 章中讲到的异常处理。下面我们就用实验来演示异常处理在上述两种情况中的应用。

4.5.1　实验目的

- 用回实验 1 和实验 2 中用到的网络拓扑，将交换机 SW3（192.168.2.13）用户名 python 的密码从 123 改为 456，并将 SW4（192.168.2.14）的 Gi0/0 端口断掉。
- 创建一个带有 try、except 异常处理语句的脚本来批量在交换机 SW1～SW5 上执行 show clock 命令，让脚本在 SW3、SW4 分别因为用户名密码不匹配，以及连通性出现故障的情况下，依然可以不受干扰，进而完成剩下的配置。

4.5.2　实验准备

（1）首先将 SW3 用户名 python 的密码从 123 改为 456，如下所示。

```
SW3#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW3(config)#username python password 456
SW3(config)#end
```

（2）然后将 SW4 的端口 Gi0/0 关掉，如下所示。

```
SW4#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW4(config)#int gi0/0
SW4(config-if)#shutdown
SW4(config-if)#end
SW4#
*May  4 08:31:15.664: %SYS-5-CONFIG_I: Configured from console by console
SW4#
```

（3）在主机上创建一个名为 ip_list.txt 的文本文件，内含 SW1～SW5 的管理 IP 地址，注意在实验 2 中我们已经将 SW5 的管理 IP 地址从 192.168.2.15 改成 192.168.2.55，如下所示。

```
[root@CentOS-Python ~]# cat ip_list.txt
192.168.2.11
192.168.2.12
192.168.2.13
192.168.2.14
192.168.2.55
[root@CentOS-Python ~]#
```

（4）延续实验 2 的思路，我们在主机上创建一个名为 cmd.txt 的文本文件，写入我们要在 SW1～SW5 上执行的命令：show clock，如下所示。

```
[root@CentOS-Python ~]#
[root@CentOS-Python ~]# cat cmd.txt
show clock
[root@CentOS-Python ~]#
```

（5）最后创建实验 3 的脚本文件 lab3.py，如下所示。

```
[root@CentOS-Python ~]#
[root@CentOS-Python ~]# vi lab3.py
```

4.5.3 实验代码

将下列代码写入脚本 lab3.py。

```
import paramiko
import time
```

```
import getpass
import sys
import socket

username = input('Username: ')
password = getpass.getpass('password: ')
ip_file = sys.argv[1]
cmd_file = sys.argv[2]

switch_with_authentication_issue = []
switch_not_reachable = []

iplist = open(ip_file, 'r')
for line in iplist.readlines():
    try:
        ip = line.strip()
        ssh_client = paramiko.SSHClient()
        ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        ssh_client.connect(hostname=ip, username=username, password=password)
        print ("You have successfully connect to ", ip)
        command = ssh_client.invoke_shell()
        cmdlist = open(cmd_file, 'r')
        cmdlist.seek(0)
        for line in cmdlist.readlines():
            command.send(line + "\n")
        time.sleep(2)
        cmdlist.close()
        output = command.recv(65535)
        print (output.decode('ascii'))
    except paramiko.ssh_exception.AuthenticationException:
        print ("User authentication failed for " + ip + ".")
        switch_with_authentication_issue.append(ip)
    except socket.error:
        print (ip +  " is not reachable.")
        switch_not_reachable.append(ip)
```

```
iplist.close()
ssh_client.close

print ('\nUser authentication failed for below switches: ')
for i in switch_with_authentication_issue:
    print (i)

print ('\nBelow switches are not reachable: ')
for i in switch_not_reachable:
    print (i)
```

4.5.4 代码分段讲解

（1）为了使用异常处理来应对网络设备不可达引起的 socket.error，必须导入 Python 内建模块 socket。

```
import paramiko
import time
import getpass
import sys
import socket

username = input('Username: ')
password = getpass.getpass('password: ')
ip_file = sys.argv[1]
cmd_file = sys.argv[2]
```

（2）创建两个空列表，分别命名为 switch_with_authentication_issue 和 switch_not_reachable，它们的作用是在脚本最后配合 for 循环来统计哪些设备是因为认证问题无法登录的，哪些设备是因为设备本身不可达而无法登录的。

```
switch_with_authentication_issue = []
switch_not_reachable = []
```

（3）在 for 循环下使用 try　except 异常处理语句。当 SSH 登录交换机时，如果用户名和密码不正确，Python 会报错“Paramiko.ssh_exception.AuthenticationException”，因此我们用“excpet　Paramiko.ssh_exception.AuthenticationException:”来应对该异常，一旦有交换机出现用户名和密码不正确的情况，打印出“User authentication failed for [交换机 IP]”来取代前面的“Paramiko.ssh_exception.AuthenticationException”错误信息，然后将出现该异常的交换机的管理 IP 地址用列表的 append()方法放入 switch_with_authentication_issue 列表中。同理，用 except socket.error:来应对交换机不可达时返回的错误“socket.error: [Errno 10060] A connection attempt failed because the connected party did not properly respond after a period of time，or established connection failed because connected host has failed to respond”，并打印出“[交换机 IP] is not reachable”来取代上述错误信息，然后将出现该错误的交换机的管理 IP 地址用列表的 append()方法放入 switch_not_reachable 列表中。

```
iplist = open(ip_file, 'r')
for line in iplist.readlines():
    try:
        ip = line.strip()
        ssh_client = paramiko.SSHClient()
        ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        ssh_client.connect(hostname=ip, username=username, password=password)
        print ("You have successfully connect to ", ip)
        command = ssh_client.invoke_shell()
        cmdlist = open(cmd_file, 'r')
        cmdlist.seek(0)
        for line in cmdlist.readlines():
            command.send(line + "\n")
        time.sleep(2)
        cmdlist.close()
        output = command.recv(65535)
        print (output.decode('ascii'))
    except paramiko.ssh_exception.AuthenticationException:
        print ("User authentication failed for " + ip + ".")
        switch_with_authentication_issue.append(ip)
    except socket.error:
```

```
        print (ip +  " is not reachable.")
        switch_not_reachable.append(ip)

iplist.close()
ssh_client.close
```

（4）最后使用 for 循环，打印出 switch_with_authentication_issue 和 switch_not_reachable 两个列表中的元素，这样就能清楚看到有哪些交换机的用户名和密码验证失败，哪些交换机的管理 IP 地址不可达。

```
print ('\nUser authentication failed for below switches: ')
for i in switch_with_authentication_issue:
    print (i)

print ('\nBelow switches are not reachable: ')
for i in switch_not_reachable:
    print (i)
```

4.5.5 验证

如下所示，重点部分已经用线标注。

```
[root@localhost ~]# python lab4.py ip_list.txt cmd.txt
Username: python
password:
You have successfully connect to  192.168.2.11

*************************************************************************
* IOSv - Cisco Systems Confidential                                     *
*                                                                       *
* This software is provided as is without warranty for internal         *
* development and testing purposes only under the terms of the Cisco    *
* Early Field Trial agreement.  Under no circumstances may this software *
* be used for production purposes or deployed in a production           *
* environment.                                                          *
*                                                                       *
* By using the software, you agree to abide by the terms and conditions *
* of the Cisco Early Field Trial Agreement as well as the terms and     *
* conditions of the Cisco End User License Agreement at                 *
* http://www.cisco.com/go/eula                                          *
*                                                                       *
* Unauthorized use or distribution of this software is expressly        *
* Prohibited.                                                           *
*************************************************************************
```

```
S1#show clock
*06:07:20.072 PDT Sat Mar 30 2019
S1#
S1#
You have successfully connect to  192.168.2.12

**************************************************************************
* IOSv - Cisco Systems Confidential                                      *
*                                                                        *
* This software is provided as is without warranty for internal          *
* development and testing purposes only under the terms of the Cisco     *
* Early Field Trial agreement.  Under no circumstances may this software *
* be used for production purposes or deployed in a production            *
* environment.                                                           *
*                                                                        *
* By using the software, you agree to abide by the terms and conditions  *
* of the Cisco Early Field Trial Agreement as well as the terms and      *
* conditions of the Cisco End User License Agreement at                  *
* http://www.cisco.com/go/eula                                           *
*                                                                        *
* Unauthorized use or distribution of this software is expressly         *
* Prohibited.                                                            *
**************************************************************************
S2#show clock
*13:06:52.531 UTC Sat Mar 30 2019
S2#
S2#
User authentication failed for 192.168.2.13.
192.168.2.14 is not reachable.
You have successfully connect to  192.168.2.55

**************************************************************************
* IOSv - Cisco Systems Confidential                                      *
*                                                                        *
* This software is provided as is without warranty for internal          *
* development and testing purposes only under the terms of the Cisco     *
* Early Field Trial agreement.  Under no circumstances may this software *
* be used for production purposes or deployed in a production            *
* environment.                                                           *
*                                                                        *
* By using the software, you agree to abide by the terms and conditions  *
* of the Cisco Early Field Trial Agreement as well as the terms and      *
* conditions of the Cisco End User License Agreement at                  *
* http://www.cisco.com/go/eula                                           *
*                                                                        *
* Unauthorized use or distribution of this software is expressly         *
* Prohibited.                                                            *
**************************************************************************
S5#show clock
*13:08:41.053 UTC Sat Mar 30 2019
S5#
S5#

User authentication failed for below switches:
192.168.2.13

Below switches are not reachable:
192.168.2.14
[root@localhost ~]#
```

（1）本应出现的“Paramiko.ssh_exception.AuthenticationException”错误已经被“User authentication failed for 192.168.2.13”取代，并且 Python 脚本并未就此停止运行，而是继续尝试登录下一个交换机 SW4，也就是 192.168.2.14。

（2）本应出现的“socket.error: [Errno 10060] A connection attempt failed because the connected party did not properly respond after a period of time，or established connection failed because connected host has failed to respond”错误已经被“192.168.2.14 is not reachable”取代，并且 Python 脚本并未就此停止运行，而是继续尝试登录下一个交换机 SW5，也就是 192.168.2.55。

（3）在脚本的最后，可以看到哪些交换机出现了用户名和密码认证失败的情况，哪些交换机出现了管理 IP 地址不可达的情况。

4.6　实验 4　Python 实现网络设备的配置备份

将网络设备的配置做备份是网络运维中必不可少的一项工作，根据公司的规模和要求的不同，管理层可能会要求对网络设备的配置做月备、周备甚至日备。传统的备份思科交换机配置的办法是手动 SSH 远程登录设备，然后输入命令 term len 0 和 show run，将回显内容手动复制/粘贴到一个 TXT 或者 Word 文本文件中，效率十分低下，在有成百上千台设备需要备份的网络中尤为明显。实验 4 将演示如何用 Python 来解决这个困扰了传统网络工程师很多年的网络运维痛点。

4.6.1　实验目的

在 CentOS 8 主机上开启 FTP Server 服务（鉴于 TFTP 在 CentOS 8 上安装使用的复杂程度，这里用 FTP 做替代），创建 Python 脚本，将 SW1～SW5 的 running configuration 备份保存到 TFTP 服务器上。

4.6.2 实验准备

（1）将 SW5 的管理 IP 地址改回 192.168.2.15，将 SW3 的用户名 python 的密码从 456 改回 123，将 SW4 的 Gi0/0 端口重新开启，如下所示。

```
SW5#
SW5#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW5(config)#int vlan 1
SW5(config-if)#ip add 192.168.2.15 255.255.255.0
SW5(config-if)#end
SW5#wr mem
```

```
SW3#
SW3#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW3(config)#username python password 123
SW3(config)#end
SW3#wr mem
Building configuration...
```

```
SW4#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW4(config)#int gi0/0
SW4(config-if)#no shut
SW4(config-if)#end
SW4#
*May  8 10:05:55.505: %SYS-5-CONFIG_I: Configured from console by console
SW4#
*May  8 10:05:56.060: %LINK-3-UPDOWN: Interface GigabitEthernet0/0, changed state to up
*May  8 10:05:57.064: %LINEPROTO-5-UPDOWN: Line protocol on Interface GigabitEthernet0/0, changed state to up
SW4#
```

（2）将 CentOS 上的 ip_list.txt 里 SW5 的管理 IP 地址也改回为 192.168.2.15，如下所示。

```
[root@CentOS-Python ~]# cat ip_list.txt
192.168.2.11
192.168.2.12
192.168.2.13
192.168.2.14
192.168.2.15
[root@CentOS-Python ~]#
```

（3）在 CentOS 8 主机上输入下列命令下载安装 vsftpd （FTP 服务），安装前需要确认主机能否连通外网。

```
dnf install vsftpd -y
```

（4）安装完成后，在 CentOS 8 主机上输入下面两个命令分别让 CentOS 在当前和开机时启动 vsftpd 服务，如下所示。

```
[root@CentOS-Python ~]# dnf install vsftpd -y
Last metadata expiration check: 0:06:21 ago on Fri 08 May 2020 04:22:58 AM EDT.
Dependencies resolved.
================================================================================
 Package                                        Architecture
================================================================================
Installing:
 vsftpd                                         x86_64

Transaction Summary
================================================================================
Install  1 Package

Total download size: 180 k
Installed size: 359 k
Downloading Packages:
vsftpd-3.0.3-28.el8.x86_64.rpm
--------------------------------------------------------------------------------
Total
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :
  Installing       : vsftpd-3.0.3-28.el8.x86_64
  Running scriptlet: vsftpd-3.0.3-28.el8.x86_64
  Verifying        : vsftpd-3.0.3-28.el8.x86_64

Installed:
  vsftpd-3.0.3-28.el8.x86_64

Complete!
[root@CentOS-Python ~]#
```

```
systemctl start vsftpd
systemctl enable vsftpd
```

```
[root@CentOS-Python ~]# systemctl start vsftpd
[root@CentOS-Python ~]# systemctl enable vsftpd
Created symlink /etc/systemd/system/multi-user.target.wants/vsftpd.service → /usr/lib/s
ystemd/system/vsftpd.service.
[root@CentOS-Python ~]#
```

（5）输入下面的命令确认 vsfptd 已经被启动运行。

```
systemctl status vsftpd
```

```
[root@CentOS-Python ~]# systemctl status vsftpd
● vsftpd.service - Vsftpd ftp daemon
   Loaded: loaded (/usr/lib/systemd/system/vsftpd.service; enabled; vendor preset: disabled)
   Active: active (running) since Fri 2020-05-08 04:35:25 EDT; 2min 58s ago
 Main PID: 33940 (vsftpd)
    Tasks: 1 (limit: 11343)
   Memory: 552.0K
   CGroup: /system.slice/vsftpd.service
           └─33940 /usr/sbin/vsftpd /etc/vsftpd/vsftpd.conf

May 08 04:35:25 CentOS-Python systemd[1]: Starting Vsftpd ftp daemon...
May 08 04:35:25 CentOS-Python systemd[1]: Started Vsftpd ftp daemon.
[root@CentOS-Python ~]#
```

（6）输入下列命令关闭 CentOS 8 的防火墙（仅实验演示使用，生产环境里建议修改防火墙策略）并验证，如下所示。

```
systemctl stop firewalld
systemctl status firewalld
```

```
[root@CentOS-Python ~]# systemctl stop firewalld
[root@CentOS-Python ~]# systemctl status firewalld
● firewalld.service - firewalld - dynamic firewall daemon
   Loaded: loaded (/usr/lib/systemd/system/firewalld.service; enabled; vendor preset: enabled)
   Active: inactive (dead) since Fri 2020-05-08 05:12:23 EDT; 11s ago
     Docs: man:firewalld(1)
  Process: 1110 ExecStart=/usr/sbin/firewalld --nofork --nopid $FIREWALLD_ARGS (code=exited, status=0/SUCCESS)
 Main PID: 1110 (code=exited, status=0/SUCCESS)

May 08 04:22:20 CentOS-Python systemd[1]: Starting firewalld - dynamic firewall daemon...
May 08 04:22:20 CentOS-Python systemd[1]: Started firewalld - dynamic firewall daemon.
May 08 04:22:20 CentOS-Python firewalld[1110]: WARNING: AllowZoneDrifting is enabled. This is considered an insec
May 08 05:12:22 CentOS-Python systemd[1]: Stopping firewalld - dynamic firewall daemon...
May 08 05:12:23 CentOS-Python systemd[1]: Stopped firewalld - dynamic firewall daemon.
[root@CentOS-Python ~]#
```

（7）在 CentOS 上输入下列命令分别创建新的用户名 python 并根据提示设置密码 python，这个用户名 python 将稍后作为 FTP 用户使用名，而交换机的 running configuration 也将被保存在/home/python 文件夹下，如下所示。

```
useradd -create-home python
passwd python
```

```
[root@CentOS-Python ~]# useradd --create-home python
[root@CentOS-Python ~]# passwd python
Changing password for user python.
New password:
BAD PASSWORD: The password is shorter than 8 characters
Retype new password:
passwd: all authentication tokens updated successfully.
[root@CentOS-Python ~]#
```

（8）在主机上创建实验 4 的脚本，将其命名为 lab4.py。

```
[root@CentOS-Python ~]#
[root@CentOS-Python ~]# vi lab4.py
```

4.6.3　实验代码

将下列代码写入脚本 lab4.py。

```
import paramiko
import time
import getpass

username = input('Username: ')
password = getpass.getpass('password: ')

f = open("ip_list.txt")

for line in f.readlines():
    ip_address = line.strip()
    ssh_client = paramiko.SSHClient()
    ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh_client.connect(hostname=ip_address,username=username,password=password)
    print ("Successfully connect to ", ip_address)
    command = ssh_client.invoke_shell()
    command.send("configure terminal\n")
    command.send("ip ftp username python\n")
    command.send("ip ftp password python\n")
```

```
        command.send("file prompt quiet\n")
        command.send("end\n")
        command.send("copy running-config ftp://192.168.2.1\n")
        time.sleep(5)
        output = command.recv(65535)
        print (output.decode('ascii'))

    f.close()
    ssh_client.close
```

4.6.4 代码分段讲解

（1）实验 4 的代码难度不大，下面这段代码的作用在实验 2 中已经有详细解释，这里不再赘述。

```
import paramiko
import time
import getpass

username = input('Username: ')
password = getpass.getpass('password: ')

f = open("ip_list.txt")

for line in f.readlines():
    ip_address = line.strip()
    ssh_client = paramiko.SSHClient()
    ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh_client.connect(hostname=ip_address,username=username,password=password)
    print ("Successfully connect to ", ip_address)
    command = ssh_client.invoke_shell()
```

（2）首先我们在每个交换机中通过命令 ip ftp username python 和 ip ftp password python

创建 FTP 用户名和密码，该用户名和密码同我们在 CentOS 主机上创建的一样。

```
    command.send("configure terminal\n")
    command.send("ip ftp username python\n")
    command.send("ip ftp password python\n")
```

（3）开启“file prompt quiet”，然后将交换机的配置文件备份到 CentOS 主机。

```
command.send("file prompt quiet\n")
    command.send("end\n")
    command.send("copy running-config ftp://192.168.2.1\n")
```

命令 file prompt 用来修改交换机文件操作的提醒方式有 alert、noisy 和 quiet 3 种模式，默认是 alert。该模式和 noisy 都会在用户进行文件操作时提示用户确认目标主机地址及目标文件名等参数，比如我们在交换机里输入命令 copy running-config ftp://192.168.2.1 将交换机的配置文件通过 FTP 备份到 CentOS 主机 192.168.2.1 后，如果使用 file prompt alert 或者 file prompt noisy，则交换机都会提醒你对目标主机地址及目标文件名进行确认，举例如下。

```
SW1#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW1(config)#file pr
SW1(config)#file prompt ?
  alert  Prompt only for destructive file operations
  noisy  Confirm all file operation parameters
  quiet  Seldom prompt for file operations
  <cr>

SW1(config)#file prompt alert
SW1(config)#end
SW1#
SW1#
SW1#
SW1#
*May  8 10:40:30.010: %SYS-5-CONFIG_I: Configured from console by console
SW1#copy running-config ftp://192.168.2.1
Address or name of remote host [192.168.2.1]?
Destination filename [sw1-confg]?
Writing sw1-confg !
3822 bytes copied in 25.809 secs (148 bytes/sec)
SW1#
```

系统已经自动设置好目标文件名，其格式为“交换机的 hostname-config”，如果你对这个系统默认设置好的目标文件名没有问题，那么 alert 和 noisy 这两种 file prompt 模式非

但对我们没有任何帮助，反而还会影响脚本的运行，因此我们使用 quiet 模式。关于 quiet 模式的效果举例如下。

```
SW1#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW1(config)#file prompt quiet
SW1(config)#do copy running-config ftp://192.168.2.1
Writing sw1-confg !
3840 bytes copied in 25.533 secs (150 bytes/sec)
SW1(config)#
```

（4）后面部分的代码都是实验 1 和 2 中讲过的，不再赘述。

```
        time.sleep(3)
        output = command.recv(65535)
        print (output.decode('ascii'))

    f.close()
```

4.6.5 验证

（1）执行代码前，在 CentOS 主机上确认/home/python 文件夹下没有任何文件，如下所示。

```
[root@CentOS-Python ~]# cd /home/python/
[root@CentOS-Python python]# ls
[root@CentOS-Python python]#
```

（2）回到脚本 lab4.py 所在的文件夹，执行脚本 lab4.py 后，输入交换机的 SSH 用户名和密码然后看效果，这里只截取脚本在 SW1 和 SW2 上运行后的回显内容，如下所示。

```
[root@CentOS-Python python]# cd
[root@CentOS-Python ~]# python3.8 lab4.py
Username: python
password:
Successfully connect to  192.168.2.11

**************************************************************************
* IOSv is strictly limited to use for evaluation, demonstration and IOS  *
* education. IOSv is provided as-is and is not supported by Cisco's      *
* Technical Advisory Center. Any use or disclosure, in whole or in part, *
* of the IOSv Software or Documentation to any third party for any       *
* purposes is expressly prohibited except as otherwise authorized by     *
* Cisco in writing.                                                      *
**************************************************************************
SW1#configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
SW1(config)#ip ftp username python
SW1(config)#ip ftp password python
SW1(config)#file prompt quiet
SW1(config)#end
SW1#copy running-config ftp://192.168.2.1

Successfully connect to  192.168.2.12
```

```
**************************************************************************
* IOSv is strictly limited to use for evaluation, demonstration and IOS  *
* education. IOSv is provided as-is and is not supported by Cisco's      *
* Technical Advisory Center. Any use or disclosure, in whole or in part, *
* of the IOSv Software or Documentation to any third party for any       *
* purposes is expressly prohibited except as otherwise authorized by     *
* Cisco in writing.                                                      *
**************************************************************************
SW2#configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
SW2(config)#ip ftp username python
SW2(config)#ip ftp password python
SW2(config)#file prompt quiet
SW2(config)#end
SW2#copy running-config ftp://192.168.2.1

Successfully connect to  192.168.2.13

**************************************************************************
* IOSv is strictly limited to use for evaluation, demonstration and IOS  *
* education. IOSv is provided as-is and is not supported by Cisco's      *
* Technical Advisory Center. Any use or disclosure, in whole or in part, *
* of the IOSv Software or Documentation to any third party for any       *
* purposes is expressly prohibited except as otherwise authorized by     *
```

（3）脚本运行完毕后，回到/home/python，此时可以看到 SW1～SW5 交换机的 running

config 都被成功备份到该文件夹下，如下所示。

```
[root@CentOS-Python ~]#
[root@CentOS-Python ~]#
[root@CentOS-Python ~]# cd /home/python
[root@CentOS-Python python]# ls
sw1-confg  sw2-confg  sw3-confg  sw4-confg  sw5-confg
```

（4）用 cat 打开其中任意一个 config 文件，验证其内容，如下所示。

```
[root@CentOS-Python python]# cat sw1-confg

!
! Last configuration change at 10:56:55 UTC Fri May 8 2020 by python
!
version 15.2
service timestamps debug datetime msec
service timestamps log datetime msec
no service password-encryption
service compress-config
!
hostname SW1
!
boot-start-marker
boot-end-marker
```

```
!
!
!
username python privilege 15 password 0 123
no aaa new-model
!
!
!
!
!
!
!
!
no ip domain-lookup
ip domain-name python.com
ip cef
no ipv6 cef
!
!
file prompt quiet
!
spanning-tree mode pvst
spanning-tree extend system-id
!
vlan internal allocation policy ascending
!
!
!
!
!
!
!
!
```

第 5 章

Python 网络运维实践（真机）

本章将以笔者在工作中遇到的 3 个实际案例演示 Python 在工作实战中的应用，本章给出的 Python 脚本和代码将在生产网络里的真机网络设备上实战运行。

◎ 真机运行环境

主机操作系统：Windows 10 上运行 CentOS 8（VMware 虚拟机）

网络设备：思科 2960、2960S、2960X、3750、3850 若干

网络设备 OS 版本：思科 IOS、IOS-XE

Python 版本：3.8.2

实验网络拓扑：与模拟器运行环境类似，不同点是运行 Python 的主机和网络设备处在不同网段中，并且网络设备的管理 IP 地址并不连续。

5.1 实验 1 大规模批量修改交换机 Qos 的配置

在第 4 章的实验 2 中提到了，要使用 Python 来批量连接管理 IP 地址不连续的网络设备，可以把设备的管理 IP 地址预先写入一个文本文件，然后在代码中使用 for 循环配合 open()函数和 readlines()函数逐行读取该文本文件里的管理 IP 地址，达到循环批量登录多台网络设备的目的。

在成功登录交换机后，我们可以配合 command.send()来对网络设备“发号施令”，但在前面的例子中我们都是将要输入的命令预先写在脚本里，比如 command.send("conf t\n")、command.send("router eigrp 1\n")和 command.send("end\n")等。这种将配置命令预先写在脚本里的方法便于初学者理解和学习，在只有几台设备的实验环境中常用。但是在有成千上万台网络设备需要管理的生产环境中，这种方法显得很笨拙，缺乏灵活性。举例来说，假设生产环境中有不同型号、不同操作系统、不同命令格式的设备各 1 000 台，比如思科的 3750 和 3850 交换机，前者运行的是 IOS，后者运行的是 IOS-XE。

最近你接到任务，需要分别给这两种交换机批量修改 QoS 的配置，因为两者的命令格式差异巨大（一个是 MLS QoS，一个是 MQC QoS），必须反复修改 command.send()部分的代码，如果只是简单数条命令还好办，一旦遇到大规模的配置，那么这种方法的效率会很低。

解决这个问题的思路是分别创建两个文本文件，一个用来存放配置 3750 交换机要用的命令集，另一个用来存放配置 3850 交换机要用到的命令集，然后在 Python 脚本里同样通过 for 循环加 open()函数来读取两个文件里的内容，达到分别给所有 3750 和 3850 交换机做 QoS 配置的目的，这样做的好处是无须修改 command.send()部分的代码，因为所有的命令行已经在文本文件里预先设置好了。

但是新的问题又来了，每次配备不同型号的设备，都必须手动修改 open()函数所打开的配置文本文件及 IP 地址文件，比如在给 3750 交换机做配置时，需要 open('command_3750.txt')和 open('ip_3750.txt')，给 3850 交换机做配置时，又需要 open('command_3850.txt')和 open('ip_3850.txt')，这样一来二去修改配置脚本的做法大大缺乏灵活性。如果只有两种不同型号、不同命令格式的设备还能应付，那么当生产环境中同时使用 3750（IOS）、3850（IOS-XE）、Nexus 3k/5k/7k/9k（NX-OS）、CRS3/ASR9K（IOS-XR），甚至其他厂商的设备，而又要对所有这些设备同时修改某个共有的配置。比如网络新添加了某台 TACACS 服务器，要统一给所有设备修改它们的 AAA 配置；又或者网络新添加了某台 NMS 系统，要统一给所有设备修改 SNMP 配置。因为不同 OS 的设备的配置命令完全不同，这时就能体会

到痛苦了。此时我们可以用下面实验中的 sys.argv 来解决这个问题。

5.1.1 实验背景

本实验将在真机下完成。

- 假设现在手边有 3 台管理 IP 地址在 192.168.100.x /24 网段的 3750 交换机和 3 台管理 IP 地址在 172.16.100.x/24 网段的 3850 交换机，它们的 hostname 和管理 IP 地址分别如下。

3750_1: 192.168.100.11

3750_2: 192.168.100.22

3750_3: 192.168.100.33

3850_1: 172.16.100.11

3850_2: 172.16.100.22

3850_3: 172.16.100.33

5.1.2 实验目的

- 修改所有 3750 和 3850 交换机的 QoS 配置，更改它们出队列（output queue）的队列参数集 2（queue-set 2）的缓存（buffers）配置，给队列 1、2、3 和 4 分别分配 15%、25%、40%和 20%的缓存（默认状况下是 25%、25%、25%和 25%）。

5.1.3　实验准备

（1）首先创建两个名为 command_3750.txt 和 ip_3750.txt 的文本文件，分别用来保存我们将要配置 3750 交换机的 QoS 命令，以及所有 3750 交换机的管理 IP 地址。

```
    [root@CentOS-Python ~]# cat command_3750.txt
    configure terminal
mls qos queue-set output 1 buffers 15 25 40 20
    end
    wr mem

[root@CentOS-Python ~]# cat ip_3750.txt
192.168.100.11
192.168.100.22
192.168.100.33
```

（2）同理，创建两个名为 command_3850.txt 和 ip_3850.txt 的文本文件，分别用来保存我们将要配置 3850 交换机的 QoS 命令，以及所有 3850 交换机的管理 IP 地址。

```
    [root@CentOS-Python ~]# cat command_3850.txt
configure terminal
class-map match-any cos7
match cos 7
class-map match-any cos1
match cos 1
exit
policy-map queue-buffer
class cos7
bandwidth percent 10
queue-buffers ratio 15
class cos1
bandwidth percent 30
queue-buffers ratio 25
exit
exit
```

```
interface gi1/0/1
service-policy output queue-buffer
end
wr mem

[root@CentOS-Python ~]# cat ip_3850.txt
172.16.100.11
172.16.100.22
172.16.100.33
```

（3）在主机上创建实验 1 的脚本，将其命名为 lab1.py。

```
[root@CentOS-Python ~]#
[root@CentOS-Python ~]# vi lab1.py_
```

5.1.4 实验代码

将下列代码写入脚本 lab1.py。

```
import paramiko
import time
import getpass
import sys

username = input('username: ')
password = getpass.getpass('password: ')
ip_file = sys.argv[1]
cmd_file = sys.argv[2]

iplist = open(ip_file, 'r')
for line in iplist.readlines():
    ip = line.strip()
    ssh_client = paramiko.SSHClient()
    ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh_client.connect(hostname=ip, username=username, password=password)
    print "You have successfully connect to ", ip
```

```
        command = ssh_client.invoke_shell()
        cmdlist = open(cmd_file,  'r')
        cmdlist.seek(0)
        for line in cmdlist.readlines():
            command.send(line + "\n")
            time.sleep(1)
        cmdlist.close()
        output = command.recv(65535)
        print (output.decode("ascii"))

    iplist.close()
    ssh_client.close
```

5.1.5　代码分段讲解

（1）因为要用到 sys.argv，所以我们导入了 sys 模块。sys 模块是 Python 中十分常用的内建模块。其余部分的代码都是在前面实验中讲过的，不再赘述。

```
import paramiko
import time
import getpass
import sys

username = input('username: ')
password = getpass.getpass('password: ')
```

（2）创建两个变量 ip_file 和 cmd_file，分别对应 sys.argv[1]和 sys.argv[2]。

```
ip_file = sys.argv[1]
cmd_file = sys.argv[2]
```

argv 是 argument variable 参数变量的简写形式，这个变量的返回值是一个列表，该列表中的元素即我们在主机命令行里运行 Python 脚本时输入的命令。sys.argv[0] 一般是被调用的.py 脚本的文件名，从 sys.argv[1]开始就是为这个脚本添加的参数。举个例子，我们

现在返回主机，输入下面这条命令。

```
[root@CentOS-Python ~]# python3.8 lab1.py ip_3750.txt cmd_3750.txt
```

那么，这时的 sys.argv 即含有 lab1.py、ip_3750.txt、cmd_3750.txt 3 个元素的列表。这时 sys.argv[0] = lab1.py，sys.argv[1] = ip_3750.txt，sys.argv[2] = cmd_3750.txt。相应地，代码里的 ip_file = sys.argv[1] 此时等同于 ip_file = ip_3750.txt，cmd_file = sys.argv[2]此时等同于 cmd_file = cmd_3750.txt。同理，如果这时我们在主机上执行如下命令。

```
[root@CentOS-Python ~]# python3.8 lab1.py ip_3850.txt cmd_3850.txt
```

则此时 ip_file = ip_3850.txt，cmd_file = cmd_3850.txt。由此可见，配合 sys.argv，我们可以很灵活地选用脚本需要调用的参数（文本文件），而无须反反复复地修改脚本代码。

（3）需要注意的是，在剩下的代码中，我们没有在脚本里预先写好具体的 QoS 配置命令，取而代之的是通过 cmd_file = sys.argv[2]配合 cmdlist = open(cmd_file， 'r')和 for line in cmdlist.readlines()来读取独立于脚本之外的配置命令文件，可以随意在命令行里选择我们想要的配置命令文件，也就是本实验中的 cmd_3750.txt 和 cmd_3850.txt。

```
iplist = open(ip_file, 'r')
for line in iplist.readlines():
    ip = line.strip()
    ssh_client = paramiko.SSHClient()
    ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh_client.connect(hostname=ip, username=username, password=password)
    print "You have successfully connect to ", ip
    command = ssh_client.invoke_shell()
    cmdlist = open(cmd_file, 'r')
    cmdlist.seek(0)
    for line in cmdlist.readlines():
        command.send(line + "\n")
        time.sleep(1)
    cmdlist.close()
    output = command.recv(65535)
    print (output.decode("ascii"))
```

```
iplist.close()
ssh_client.close
```

5.1.6　验证

```
[root@CentOS-Python ~]# python3.8 lab1.py ip_3750.txt cmd_3750.txt
Username: python
password:
You have successfully connect to 192.168.100.11
3750_1#conf t
3750_1(config)#mls qos queue-set output 1 buffers 15 25 40 20
3750_1(config)#end
3750_1#wr mem
Building configuration...
[OK]

You have successfully connect to 192.168.100.22
3750_2#conf t
3750_2(config)#mls qos queue-set output 1 buffers 15 25 40 20
3750_2(config)#end
3750_2#wr mem
Building configuration...
[OK]

You have successfully connect to 192.168.100.33
3750_3#conf t
3750_3(config)#mls qos queue-set output 1 buffers 15 25 40 20
3750_3(config)#end
3750_3#wr mem
Building configuration...
[OK]

[root@CentOS-Python ~]# python lab1.py ip_3850.txt cmd_3850.txt
```

```
Username: python
password:

You have successfully connect to 172.16.100.11
3850_1#configure terminal
Enter configuration commands, one per line. End with CNTL/Z.
3850_1(config)#class-map match-any cos7
3850_1(config-cmap)#match cos 7
3850_1(config-cmap)#class-map match-any cos1
3850_1(config-cmap)#match cos 1
3850_1(config-cmap)#exit
3850_1(config)#policy-map queue-buffer
3850_1(config-pmap)#class cos7
3850_1(config-pmap-c)#bandwidth percent 10
3850_1(config-pmap-c)#queue-buffers ratio 15
3850_1(config-pmap-c)#class cos1
3850_1(config-pmap-c)#bandwidth percent 30
3850_1(config-pmap-c)#queue-buffers ratio 25
3850_1(config-pmap-c)#exit
3850_1(config-pmap)#exit
3850_1(config)#interface gi1/0/1
3850_1(config-if)#service-policy output queue-buffer
3850_1(config-if)#end
3850_1#wr mem
Building configuration...
Compressed configuration from 62654 bytes to 19670 bytes[OK]

You have successfully connect to 172.16.100.22
3850_2#configure terminal
Enter configuration commands, one per line. End with CNTL/Z.
3850_2(config)#class-map match-any cos7
3850_2(config-cmap)#match cos 7
3850_2(config-cmap)#class-map match-any cos1
3850_2(config-cmap)#match cos 1
3850_2(config-cmap)#exit
3850_2(config)#policy-map queue-buffer
```

```
3850_2(config-pmap)#class cos7
3850_2(config-pmap-c)#bandwidth percent 10
3850_2(config-pmap-c)#queue-buffers ratio 15
3850_2(config-pmap-c)#class cos1
3850_2(config-pmap-c)#bandwidth percent 30
3850_2(config-pmap-c)#queue-buffers ratio 25
3850_2(config-pmap-c)#exit
3850_2(config-pmap)#exit
3850_2(config)#interface gi1/0/1
3850_2(config-if)#service-policy output queue-buffer
3850_2(config-if)#end
3850_2#wr mem
Building configuration...
Compressed configuration from 62654 bytes to 19670 bytes[OK]

You have successfully connect to 172.16.100.33
3850_3#configure terminal
Enter configuration commands, one per line. End with CNTL/Z.
3850_3(config)#class-map match-any cos7
3850_3(config-cmap)#match cos 7
3850_3(config-cmap)#class-map match-any cos1
3850_3(config-cmap)#match cos 1
3850_3(config-cmap)#exit
3850_3(config)#policy-map queue-buffer
3850_3(config-pmap)#class cos7
3850_3(config-pmap-c)#bandwidth percent 10
3850_3(config-pmap-c)#queue-buffers ratio 15
3850_3(config-pmap-c)#class cos1
3850_3(config-pmap-c)#bandwidth percent 30
3850_3(config-pmap-c)#queue-buffers ratio 25
3850_3(config-pmap-c)#exit
3850_3(config-pmap)#exit
3850_3(config)#interface gi1/0/1
3850_3(config-if)#service-policy output queue-buffer
3850_3(config-if)#end
3850_3#wr mem
```

```
Building configuration...
Compressed configuration from 62654 bytes to 19670 bytes[OK]
```

5.2 实验 2 pythonping 的使用方法

在第 3 章中，我们曾经提到过在 Python 中用来执行 ping 命令的模块有很多种，os、subprocess 及 pyping 都可以用来 ping 指定的 IP 地址或者 URL。三者的区别是 os 和 subprocess 在执行 ping 命令时脚本会将系统执行 ping 时的回显内容显示出来，有时这些回显内容并不是必要的，比如下面是用 subprocess 模块在 CentOS 主机上执行“ping -c 3 www.cisco.com”命令的脚本及执行脚本后的回显内容。

```
[root@CentOS-Python ~]# cat ping.py

import subprocess

target = 'www.cisco.com'
ping_result = subprocess.call(['ping','-c','3',target])
if ping_result == 0:
   print (target + ' is reachable.')
else:
   print (target + ' is not reachable.')


[root@CentOS-Python ~]# python3.8 ping.py
PING www.cisco.com (27.151.12.183) 56(84) bytes of data.
64 bytes from 27.151.12.183 (27.151.12.183): icmp_seq=1 ttl=55 time=19.8 ms
64 bytes from 27.151.12.183 (27.151.12.183): icmp_seq=2 ttl=55 time=19.9 ms
64 bytes from 27.151.12.183 (27.151.12.183): icmp_seq=3 ttl=55 time=19.9 ms

--- www.cisco.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 5ms
rtt min/avg/max/mdev = 19.765/19.875/19.935/0.077 ms
www.cisco.com is reachable.
```

```
[root@CentOS-Python ~]#
```

可以明显看到，上述回显内容过多，在用脚本一次性 ping 成百上千个 IP 地址或者 URL 时非常影响美观和阅读，因为我们真正关心的其实是最后一句用 Python 打印出来的通知用户目标 IP 地址或者 URL 是否可达的内容“www.cisco.com is reachable”，而用 pyping 来执行 ping 命令则不会有回显内容过多的问题。很遗憾的是，pyping 只支持 Python 2，并且截至 2020 年 5 月，作者似乎也没有继续更新 pyping 来支持 Python 3 的意愿（虽然 pyping 依然能通过 pip3 下载 Python 3 的版本，但是使用时会报错“ModuleNotFoundError: No module named 'core'”）。

在 Python 3.8 里，我们可以使用 **pythonping** 作为 pyping 的替代品。本实验将详细介绍 pythonping 的使用方法。

5.2.1　实验背景

本实验将在真机下完成。

- 某公司有 48 口的思科 3750 交换机共 1 000 台，分别分布在 5 个子网掩码为/24 的 B 类网络子网下：172.16.0.x /24，172.16.1.x /24，172.16.2.x /24，172.16.3.x /24，172.16.4.x /24。

5.2.2　实验目的

- 在不借助任何第三方 NMS 软件或网络安全工具帮助的前提下，使用 Python 脚本依次 ping 所有交换机的管理 IP 地址，来确定当前（需要记录下运行脚本时的时间，要求精确到年月日和时分秒）有哪些交换机可达，并且统计当前每个交换机有多少用户端的物理端口是 Up 的（级联端口不算），以及这 1 000 台交换机所有 Up 的用户端物理端口的总数，并统计网络里的端口使用率（也就是物理端口的

Up 率）。

5.2.3 实验思路

- 根据实验目的，我们可以写两个脚本，脚本 1 通过导入 pythonping 模块来扫描这 5 个网段下所有交换机的管理 IP 地址，看哪些管理 IP 地址是可达的。因为子网掩码是 255.255.255.0，意味着每个网段下的管理 IP 地址的前三位都是固定不变的，只有最后一位会在 1～254 中变化。我们可以在第一个脚本中使用 for 循环来 ping .1 到.254，然后将所有该网段下可达的交换机管理 IP 地址都写入并保存在一个名为 reachable_ip.txt 的文本文件中。
- 因为我们这里有 5 个连续的/24 的网段需要扫描（从 172.16.0.x 到 172.16.4.x），我们又可以在脚本 1 中再写一个 for 循环来连续 ping 这 5 个网段，然后把上一个 ping .1 到.254 的 for 循环嵌入这一个 for 循环里，这样就能让 Python 一次性把 5 个/24 网段下总共 1270 个可用 IP 地址（254×5 = 1270）全部 ping 一遍。
- 在用脚本 1 生成 reachable_ip.txt 文件后，我们可以再写一个脚本 2 来读取该文本文件中所有可达的交换机的管理 IP 地址，依次登录所有这些可达的交换机，输入命令“show ip int brief | i up”查看有哪些端口是 Up 的，再配合正则表达式（re 模块），在回显内容中匹配我们所要的用户端物理端口号（Gix/x/x），统计它们的总数，即可得到当前一个交换机有多少个物理端口是 Up 的。（注：因为“show ip int brief | i up”的回显内容里也会出现 10GB 的级联端口 Tex/x/x 及虚拟端口，比如 vlan 或者 loopback 端口，所以这里强调的是用正则表达式来匹配用户端物理端口 Gix/x/x）。

5.2.4　实验准备 - 脚本 1

（1）pyping 为第三方模块，使用前需要通过 pip 下载安装，安装完成后进入 Python 3.8 编辑器，如果 import pythonping 没有报错，则说明安装成功，如下所示。

```
[root@CentOS-Python lab5]# pip3.8 install pythonping
Collecting pythonping
  Using cached https://files.pythonhosted.org/packages/6a/5a/f1eb3f521dcf1d2927b9be15e58a7
Installing collected packages: pythonping
  Running setup.py install for pythonping ... done
Successfully installed pythonping-1.0.9
WARNING: You are using pip version 19.2.3, however version 20.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
[root@CentOS-Python lab5]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pythonping
>>>
```

（2）在主机上创建一个新的文件夹，取名为 lab2，在该文件夹下创建实验 2 的脚本 1 文件 lab2_1.py，如下所示。

```
[root@CentOS-Python ~]# mkdir lab2
[root@CentOS-Python ~]# cd lab2
[root@CentOS-Python lab2]# vi lab2_1.py
```

5.2.5　实验代码 - 脚本 1

将下列代码写入脚本 lab2_1.py。

```
from pythonping import ping
import os

if os.path.exists('reachable_ip.txt'):
    os.remove('reachable_ip.txt')
```

```
third_octet = range(5)
last_octet = range(1, 255)

for ip3 in third_octet:
    for ip4 in last_octet:
        ip = '172.16.' + str(ip3) + '.' + str(ip4)
        ping_result = ping(ip)
        f = open('reachable_ip.txt', 'a')
        if 'Reply' in str(ping_result):
            print (ip + ' is reachable. ')
            f.write(ip + "\n")
        else:
            print (ip + ' is not reachable. ')

f.close()
```

5.2.6　脚本 1 代码分段讲解

（1）我们导入 pythonping 和 os 两个模块。在 pythonping 中，最核心的函数显然是 ping()，因为 pythonping 的模块名长度偏长，这里用 from…import…将 ping()函数导入后，后面调用时就能省去使用 pythonping.ping()完整函数路径的麻烦，体会到直接使用 ping()的便利。至于 os 模块的用法下面会讲到。

```
from pythonping import ping
import os
```

（2）每次我们运行脚本 1，都不希望保留上一次运行脚本时生成的 reachable_ip.txt 文件，因为在有成千上万台设备的大型网络里，每时每刻可达交换机的数量都有可能在改变。这时可以用 os 模块下的 os.path.exists()方法来判断该文件是否存在，如果存在，则用 os.remove()方法将该文件删除。这样可以保证每次运行脚本 1 时，reachabe_ip.txt 文件中只会包含本次运行脚本后所有可达的交换机管理 IP 地址。

```
if os.path.exists('reachable_ip.txt'):
    os.remove('reachable_ip.txt')
```

（3）实验思路中讲到，5 个/24 网段的管理 IP 地址是有规律可循的，头两位都不变，为 172.16，第三位为 0～4，第四位为 1～254，因此可以通过 range(5)和 range(1，255)分别创建两个整数列表来囊括管理 IP 地址的第三位和第四位，为后面的两个 for 循环做准备。

```
third_octet = range(5)
last_octet = range(1, 255)
```

（4）通过两个 for 循环做嵌套，依次从 172.16.0.1、172.16.0.2、172.16.0.3……一直遍历到 172.16.4.254 为止，然后配合 ping()函数来依次 ping 所有这些管理 IP 地址（前面已经提到 ping()函数是通过 from pythonping import ping 从 pythonping 模块导入的）。

```
for ip3 in third_octet:
```

```
    for ip4 in last_octet:
        ip = '172.16.' + str(ip3) + '.' + str(ip4)
        ping_result = ping(ip)
```

（5）在实验 2 的脚本里，我们将所有可达的管理 IP 地址以追加模式（a）写入 reachable_ip.txt 文件。因为 pythonping 模块在运行过程中不显示任何回显内容，就无法知道脚本运行的进度，所以通过 print (ip + ' is reachable. ')和 print (ip + ' is not reachable. ') 分别打印出目标 IP 地址是否可达的信息。

```
        f = open('reachable_ip.txt', 'a')
        if 'Reply' in str(ping_result):
            print (ip + ' is reachable. ')
            f.write(ip + "\n")
        else:
            print (ip + ' is not reachable. ')
```

这里重点解释下 if 'Reply' in str(ping_result):的用法和原理。

在使用 os、subprocess 和 pyping 等模块做 ping 测试时，如果目标 IP 地址可达，则它们会返回整数 0；如果不可达，则返回非 0 的整数。而 pythonping 不同，在 pythonping 中，ping()函数默认对目标 IP 地址 ping 4 次，当目标 IP 地址可达时，ping()函数返回的是“Reply from x.x.x.x, x bytes in xx.xx ms”；如果不可达，则返回的是“Request timed out”，如下所示。

```
[root@CentOS-Python lab5]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from pythonping import ping
>>> ping('www.cisco.com')
Reply from 117.34.19.157, 9 bytes in 36.67ms
Reply from 117.34.19.157, 9 bytes in 36.09ms
Reply from 117.34.19.157, 9 bytes in 36.71ms
Reply from 117.34.19.157, 9 bytes in 41.84ms

Round Trip Times min/avg/max is 36.09/37.83/41.84 ms
>>> ping('10.1.1.1')
Request timed out
Request timed out
Request timed out
Request timed out

Round Trip Times min/avg/max is 2000/2000.0/2000 ms
>>>
```

也许你会问：既然 pythonping 的 ping()函数返回的不再是 0 或非 0 的整数，那么我们怎么将上面代表目标 IP 地址可达的“Reply from x.x.x.x, x bytes in xx.xx ms”和代表目标 IP 地址不可达的“Request timed out”通过 if 语句将可达的目标 IP 地址打印出来并写入 reachable_ip.txt 文件呢？也许你猜到用成员运算符 in 来判断返回值中是否有 Reply 和 Request 这两个字符串。如果有 Reply，则说明目标可达；如果没有，则说明目标不可达，但前提是 ping()函数返回值的类型必须是字符串。而实际情况是 ping()函数返回值的类型是一个叫作 pythonping.executor.ResponseList 的特殊类型，如下所示。

```
>>>
>>> from pythonping import ping
>>> ping_result = ping('www.cisco.com')
>>> type(ping_result)
<class 'pythonping.executor.ResponseList'>
>>>
```

因此，必须通过 str()函数将它转换成字符串后才能使用成员运算符 in 来做判断（即这里的 str(ping_result)），**否则 if 'Reply' in str(ping_result):会永远返回布尔值 False，表示目标不可达**。

（6）最后在退出程序前不要忘记关闭已经打开的 reachable_ip.txt 文件。

```
f.close()
```

5.2.7 脚本 1 验证

（1）运行脚本 1 前，确认/root/lab2 文件夹下只有 lab2_1.py 这一个文件。

```
[root@CentOS-Python lab2]# ls
lab2_1.py
[root@CentOS-Python lab2]#
```

（2）运行脚本 1，因为 1270 个管理 IP 地址实在太多，不方便截图演示，所示这里将 last_octet = range(1，255)改为 last_octet = range(1，4)，只 ping 每个网段下前 3 个管理 IP 地址，也就是总共 15 个管理 IP 地址。运行脚本后的回显内容如下所示。

```
[root@CentOS-Python lab2]# python3.8 lab2_1.py
172.16.0.1 is reachable.
172.16.0.2 is not reachable.
172.16.0.3 is not reachable.
172.16.1.1 is not reachable.
172.16.1.2 is not reachable.
172.16.1.3 is not reachable.
172.16.2.1 is not reachable.
172.16.2.2 is not reachable.
172.16.2.3 is not reachable.
172.16.3.1 is not reachable.
172.16.3.2 is not reachable.
172.16.3.3 is not reachable.
```

（3）再次查看/root/lab2 文件夹，可以看到这时多出来了一个 reachable_ip.txt 的文本文件，该文件正是脚本自动生成的，用来保存所有可达交换机的管理 IP 地址。可以看到只有 172.16.0.1 这一个可达的管理 IP 地址被写入 reachable_ip.txt，至此证明脚本 1 运行成功，如下所示。

```
[root@CentOS-Python lab2]# ls
lab2_1.py  reachable_ip.txt
[root@CentOS-Python lab2]# cat reachable_ip.txt
172.16.0.1
[root@CentOS-Python lab2]# _
```

5.2.8　实验准备 - 脚本 2

讲解脚本 2 之前，先来看下在一个 48 口的思科 3750 交换机里输入命令 show ip int brief | i up 后能得到什么样的回显内容，如下图所示。

```
[redacted]#show ip int b | i up
Vlan3999                [redacted]      YES NVRAM  up                    up
GigabitEthernet1/0/1    unassigned      YES unset  up                    up
GigabitEthernet1/0/3    unassigned      YES unset  up                    up
GigabitEthernet1/0/5    unassigned      YES unset  up                    up
GigabitEthernet1/0/6    unassigned      YES unset  up                    up
GigabitEthernet1/0/7    unassigned      YES unset  up                    up
GigabitEthernet1/0/9    unassigned      YES unset  up                    up
GigabitEthernet1/0/10   unassigned      YES unset  up                    up
GigabitEthernet1/0/11   unassigned      YES unset  up                    up
GigabitEthernet1/0/12   unassigned      YES unset  up                    up
GigabitEthernet1/0/15   unassigned      YES unset  up                    up
GigabitEthernet1/0/16   unassigned      YES unset  up                    up
GigabitEthernet1/0/17   unassigned      YES unset  up                    up
GigabitEthernet1/0/18   unassigned      YES unset  up                    up
GigabitEthernet1/0/20   unassigned      YES unset  up                    up
GigabitEthernet1/0/21   unassigned      YES unset  up                    up
GigabitEthernet1/0/22   unassigned      YES unset  up                    up
GigabitEthernet1/0/29   unassigned      YES unset  up                    up
GigabitEthernet1/0/31   unassigned      YES unset  up                    up

GigabitEthernet1/0/33   unassigned      YES unset  up                    up
GigabitEthernet1/0/34   unassigned      YES unset  up                    up
GigabitEthernet1/0/36   unassigned      YES unset  up                    up
GigabitEthernet1/0/37   unassigned      YES unset  up                    up
GigabitEthernet1/0/38   unassigned      YES unset  up                    up
GigabitEthernet1/0/39   unassigned      YES unset  up                    up
GigabitEthernet1/0/42   unassigned      YES unset  up                    up
GigabitEthernet1/0/45   unassigned      YES unset  up                    up
Te1/0/1                 unassigned      YES unset  up                    up
Te1/0/2                 unassigned      YES unset  up                    up
```

上图是笔者在某个生产网络中的交换机里得到的回显内容。可以看到，除了 GigabitEthernet 用户端物理端口，还有 VLAN 虚拟端口和两个万兆的级联端口 Te1/0/1 和 Te1/0/2。在实验目的中已经明确说明不考虑虚拟端口和级联端口，只统计共有多少用户端物理端口是 Up 的。

创建实验 2 的第二个脚本，将它取名为 lab2_2.py，如下所示。

```
[root@CentOS-Python lab2]#
[root@CentOS-Python lab2]# vi lab2_2.py
```

5.2.9 实验代码 - 脚本 2

将下列代码写入脚本 lab2_2.py。

```
import paramiko
import time
import re
from datetime import datetime
import socket
import getpass

username = input('Enter your SSH username: ')
password = getpass.getpass('Enter your SSH password: ')
now = datetime.now()
date = "%s-%s-%s" % (now.month, now.day, now.year)
time_now = "%s:%s:%s" % (now.hour, now.minute, now.second)

switch_with_tacacs_issue = []
switch_not_reachable = []
total_number_of_up_port = 0

iplist = open('reachable_ip.txt')
number_of_switch = len(iplist.readlines())
total number of ports = number of switch * 48

iplist.seek(0)
for line in iplist.readlines():
    try:
        ip = line.strip()
        ssh_client = paramiko.SSHClient()
        ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        ssh_client.connect(hostname=ip, username=username, password=password)
        print ("\nYou have successfully connect to ", ip)
        command = ssh_client.invoke_shell()
        command.send('term len 0\n')
```

```
            command.send('show ip int b | i up\n')
            time.sleep(1)
            output = command.recv(65535)
            #print (output)
            search_up_port = re.findall(r'GigabitEthernet',  output)
            number_of_up_port = len(search_up_port)
            print (ip + " has " + str(number_of_up_port) + " ports up.")
            total_number_of_up_port += number_of_up_port
        except Paramiko.ssh_exception.AuthenticationException:
            print ("TACACS is not working for " + ip + ".")
            switch_with_tacacs_issue.append(ip)
        except socket.error:
            print (ip +  " is not reachable.")
            switch_not_reachable.append(ip)
    iplist.close()

    print ("\n")
    print ("There are totally " + str(total_number_of_ports) + " ports available in
the network.")
    print (str(total_number_of_up_port) + " ports are currently up.")
    print ("Port up rate is %.2f%%" % (total_number_of_up_port /
float(total_number_of_ports) * 100))
    print ('\nTACACS is not working for below switches: ')
    for i in switch_with_tacacs_issue:
        print (i)
    print ('\nBelow switches are not reachable: ')
    for i in switch_not_reachable:
        print (i)
    f = open(date + ".txt",  "a+")
    f.write('As of ' + date + " " + time_now)
    f.write("\n\nThere are totally " + str(total_number_of_ports) + " ports available
in the network.")
    f.write("\n" + str(total_number_of_up_port) + " ports are currently up.")
    f.write("\nPort up rate is %.2f%%" % (total_number_of_up_port /
float(total_number_of_ports) * 100))
    f.write("\n*************************************************************\n\n")
```

```
f.close()
```

5.2.10 脚本 2 代码分段讲解

（1）这里导入了 datetime 模块，这个 Python 内置的模块可以用来显示运行脚本时的系统日期和时间，因为实验目的里提到需要记录下运行脚本时的时间（精确到年月日和时分秒）。其余模块的用法在前面已经都讲过，这里不再赘述。

```
import paramiko
import time
import re
from datetime import datetime
import socket
import getpass
```

（2）使用 input()和 getpass.getpass()来分别提示用户输入 SSH 登录交换机的用户名和密码。记录当前时间可以调用 datetime.now()方法，我们将它赋值给变量 now。datetime.now()方法下面又包含了.year()（年）、.month()（月）、.day()（日）、.hour()（时）、.minute()（分）、.second()（秒）几个子方法，这里将“月-日-年”赋值给变量 date，将“时：分：秒”赋值给变量 time_now。

```
username = input('Enter your SSH username: ')
password = getpass.getpass('Enter your SSH password: ')
now = datetime.now()
date = "%s-%s-%s" % (now.month, now.day, now.year)
time_now = "%s:%s:%s" % (now.hour, now.minute, now.second)
```

（3）创建 switch_with_tacacs_issue 和 switch_not_reachable 两个空列表来统计有哪些交换机的 TACACS 失效导致用户验证失败，哪些交换机的管理 IP 地址不可达。另外创建一个 total_number_of_up_port 变量，将其初始值设为 0，后面再用累加的方法来统计所有可达的 3750 交换机上状态为 Up 的端口的总数。

```
switch_with_tacacs_issue = []
```

```
switch_not_reachable = []
total_number_of_up_port = 0
```

（4）用 open()函数打开脚本 1 创建的 reachable_ip.txt 文件，用 readlines()将其内容以列表形式返回，再配合 len()函数得到可达交换机的数量，因为每个交换机都有 48 个端口，所以通过交换机数量×48 可以得到端口总数（无论端口状态是否为 Up）。

```
iplist = open('reachable_ip.txt')
number_of_switch = len(iplist.readlines())
total_number_of_ports = number_of_switch * 48
```

（5）因为已经用 open()函数打开过一次 reachable_ip.txt 文件，所以要用 seek(0)回到文件的起始位置。这里需要注意用来应对交换机登录失败的问题，异常处理语句 try 要写在 for 循环的下面，另外剩下的代码是使用 Paramiko 配合 for 循环登录交换机并进入命令行的最基础的知识点，这里不再赘述。

```
iplist.seek(0)
for line in iplist.readlines():
    try:
        ip = line.strip()
        ssh_client = Paramiko.SSHClient()
        ssh_client.set_missing_host_key_policy(Paramiko.AutoAddPolicy())
        ssh_client.connect(hostname=ip, username=username, password=password)
        print ("\nYou have successfully connect to ",  ip)
        command = ssh_client.invoke_shell()
```

（6）因为是 48 口的交换机，show ip int brief | i up 的回显内容会比较长，无法一次性完整地显示，所以这里首先要用命令 term len 0 完整地显示所有的回显内容。“sleep”1s 后再用 recv(65535)将所有的回显内容保存在变量 output 中，如果你想在脚本执行过程中查看完整的回显内容，则可以选择 print (output)；如果不想看，则在前面加上注释符号#。

```
        command.send('term len 0\n')
        command.send('show ip int b | i up\n')
        time.sleep(1)
        output = command.recv(65535)
        #print (output)
```

（7）因为我们只想统计有多少用户端的物理端口（GigabitEthernet）是 Up 的，所以可以用正则表达式的 findall()方法去精确匹配 GigabitEthernet，将 findall()返回的列表赋值给变量 search_up_port，然后通过 len(search_up_port)即可得到 Up 的物理端口的数量，并将该数量赋值给变量 number_of_port。随后打印出每个交换机有多少个用户端物理端口是 Up 的。因为前面已经定义了变量 total_number_of_up_port，并将整数 0 赋值给它，所以可以通过 total_number_of_up_port += number_of_up_port 的方法将每个交换机 Up 的物理端口数量累加起来，最后得到整个网络下 Up 的物理端口的总数。后面的异常处理语句 except 和关闭文件的部分不再赘述。

```
            search_up_port = re.findall(r'GigabitEthernet',  output)
            number_of_up_port = len(search_up_port)
            print (ip + " has " + str(number_of_up_port) + " ports up.")
            total_number_of_up_port += number_of_up_port
        except Paramiko.ssh_exception.AuthenticationException:
            print ("TACACS is not working for " + ip + ".")
            switch_with_tacacs_issue.append(ip)
        except socket.error:
            print (ip +  " is not reachable.")
            switch_not_reachable.append(ip)
    iplist.close()
```

（8）最后除了将统计信息各种打印出来，我们还将另外创建一个文件，通过 f = open(date + ".txt"，"a")将运行脚本时的日期用作该脚本的名字，将统计信息写入，方便以后调阅查看。注意写入的内容里有 f.write('As of ' + date + " " + time_now)，这样可以清晰直观地看到我们是在哪一天的几时几分几秒运行的脚本。为什么要用日期作为文件名？这样做的好处是一旦运行脚本时的日期不同，脚本就会自动创建一个新的文件，比如 2018 年 6 月 16 号运行了一次脚本，Python 创建了一个名为 6-16-2018.txt 的文件，如果第二天再运行一次脚本，Python 又会创建一个名为 6-17-2018.txt 的文件。如果在同一天里数次运行脚本，则多次运行的结果会以追加的形式写入同一个.txt 文件，不会创建新文件。这么做可以让我们配合 Windows 的 Task Scheduler 或者 Linux 的 Crontab 来定期自动执行脚本，每天自动生成当天的端口使用量的统计情况，方便公司管理层随时观察网络里交换机的端

口使用情况。

```
    print ("\n")
    print ("There are totally " + str(total_number_of_ports) + " ports available in
the network.")
    print (str(total_number_of_up_port) + " ports are currently up.")
    print ("Port up rate is %.2f%%" % (total_number_of_up_port /
float(total_number_of_ports) * 100))
    print ('\nTACACS is not working for below switches: ')
    for i in switch_with_tacacs_issue:
        print (i)
    print ('\nBelow switches are not reachable: ')
    for i in switch_not_reachable:
        print (i)
    f = open(date + ".txt",  "a+")
    f.write('As of ' + date + " " + time_now)
    f.write("\n\nThere are totally " + str(total_number_of_ports) + " ports available
in the network.")
    f.write("\n" + str(total_number_of_up_port) + " ports are currently up.")
    f.write("\nPort up rate is %.2f%%" % (total_number_of_up_port /
float(total_number_of_ports) * 100))
    f.write("\n*********************************************************\n\n")
    f.close()
```

5.2.11　脚本 2 验证

（1）移动到/root/lab2，运行脚本 lab2_2.py，让脚本读取 reachable_ip.txt 文件自动登录所有可达的交换机（出于演示的目的，这里将之前除 172.16.0.1 外所有不可达的交换机全部开启，并将它们的管理 IP 地址写入 reachable_ip.txt），并查看每个交换机当前有多少端口是 Up 的，在最后给出统计数据，如下所示。

```
[root@localhost lab2]# python lab2_2.py
Enter your SSH username: parry
Enter your SSH password:

You have successfully connect to  172.16.0.1
172.16.0.1 has 27 ports up.

You have successfully connect to  172.16.0.2
172.16.0.2 has 41 ports up.

You have successfully connect to  172.16.0.3
172.16.0.3 has 41 ports up.

You have successfully connect to  172.16.1.1
172.16.1.1 has 11 ports up.

You have successfully connect to  172.16.1.2
172.16.1.2 has 15 ports up.

You have successfully connect to  172.16.1.3
172.16.1.3 has 12 ports up.

You have successfully connect to  172.16.2.1
172.16.2.1 has 7 ports up.

You have successfully connect to  172.16.2.3
172.16.2.3 has 8 ports up.

You have successfully connect to  172.16.3.1
172.16.3.1 has 48 ports up.

You have successfully connect to  172.16.3.2
172.16.3.2 has 51 ports up.

You have successfully connect to  172.16.3.3
172.16.3.3 has 44 ports up.

You have successfully connect to  172.16.4.1
172.16.4.1 has 118 ports up.

You have successfully connect to  172.16.4.2
172.16.4.2 has 65 ports up.

You have successfully connect to  172.16.4.3
172.16.4.3 has 120 ports up.


There are totally 672 ports available in the network.
608 ports are currently up.
Port up rate is 90.48%
```

（2）再次查看/root/lab2 文件夹，可以看到这时多出来一个名为 4-22-2019.txt 的文本文件，该文件是脚本 2 自动生成的，用来保存每次运行脚本后的统计信息。文件名即运行该脚本的日期。在 Linux 中配合 Crontab，在 Windows Server 中配合任务计划程序（Task Scheduler）定期每天运行该脚本，即可达到每天自动化监控交换机端口使用率的目的，如下所示。

```
[root@localhost lab2]# ls -l
total 16
-rw-r--r-- 1 root root  195 Apr 22 08:40 4-22-2019.txt
-rw-r--r-- 1 root root  513 Apr  6 17:38 lab2_1.py
-rw-r--r-- 1 root root 2504 Apr 15 12:56 lab2_2.py
-rw-r--r-- 1 root root  154 Apr 22 08:39 reachable_ip.txt
[root@localhost lab2]# cat 4-22-2019.txt
As of 4-22-2019 8:39:44

There are totally 672 ports available in the network.
608 ports are currently up.
Port up rate is 90.48%
*************************************************************

[root@localhost lab2]#
```

5.3　实验 3　利用 Python 脚本检查交换机的配置

前面几个实验（包括第 4 章）已经由浅入深地讲解了 Python 在网络运维中的实际应用技巧，并给出了相应的范例。这里将继续讨论如何使用 Python 来解决在大型生产网络中常见的一个网络运维自动化的需求。

5.3.1　实验背景

本实验将在真机上完成。

- 某公司有 1 000 台思科 2960、100 台思科 2960S、300 台思科 2960X 交换机，均为 24 口，它们的型号和 IOS 版本分别如下表所示。

类　别	型　号	IOS 版本
2960	WS-C2960-24PC-L	c2960-lanbasek9-mz.122-55.SE5
2960S	WS-C2960S-F24PS-L	c2960s-universalk9-mz.150-2.SE5
2960X	WS-C2960X-24PS-L	c2960x-universalk9-mz.152-2.E5

最近公司决定将所有上述交换机的 IOS 升级，修补漏洞，消除 Bug。升级后的 IOS 版本分别如下。

2960：c2960-lanbasek9-mz.122-55.SE12

2960S：c2960s-universalk9-mz.150-2.SE11

2960X：c2960x-universalk9-mz.152-2.E8

5.3.2　实验目的

- 所有交换机的新 IOS 版本已经手动完成上传，并且 boot system 的路径也改为了新的 IOS 版本，为确保 IOS 版本升级顺利完成，需要创建 Python 脚本检查所有交换机的配置是否正确，避免升级 IOS 版本的过程中出现人为的错误。

5.3.3　实验思路

升级交换机 IOS 版本的验证部分大致分为以下两个步骤。

1. 重启交换机前的验证步骤

将新版 IOS 上传到 flash 后到重启交换机前可能会遇到以下三种人为错误。

（1）IOS 版本和交换机型号货不对板，比如把 2960 的 IOS 版本上传给了 2960X。

（2）boot system 忘记修改或者修改错误。

（3）重启交换机前，忘记 write memory 保存配置。

因此，在重启所有交换机前，需要检验交换机的三处配置。

（1）show inventory | i PID: WS。

（2）show flash: | i c2960。

上述两步的目的是查看交换机型号和闪存下面的 IOS 版本，避免前者和后者货不对板的情况发生。

（3）show boot | i BOOT path。

这一步的目的检查 boot system 配置是否修改正确。

2. 重启交换机后的验证步骤

重启交换机后的验证步骤较为简单，只需要用 show ver | b SW Version 验证交换机的 IOS 版本是否已经成功升级到新版本即可。

综上所述，我们可以分别写两个脚本来对应上面两个验证步骤。脚本 1 用作交换机重启前的验证，脚本 2 用作交换机重启后的验证。

5.3.4　实验准备 - 脚本 1

- 在主机上创建一个新的文件夹，取名为 lab3，在该文件夹下创建实验 3 的脚本 1 文件 lab3_1.py，如下所示。

```
[root@CentOS-Python ~]#
[root@CentOS-Python ~]# mkdir lab3
[root@CentOS-Python ~]# cd lab3
[root@CentOS-Python lab3]# vi lab3_1.py
```

- 在主机上创建一个 ip_list.txt 文件，该文件用来保存我们要登录验证 IOS 是否升级成功的所有交换机的管理 IP 地址，如下所示。

```
[root@CentOS-Python lab3]# cat ip_list.txt
172.16.206.39
172.16.206.40
172.16.206.41
172.16.206.42
172.16.206.43
172.16.206.119
172.16.206.128
172.16.206.129
172.16.206.145
172.16.206.146
[root@CentOS-Python lab3]#
```

5.3.5 实验代码 - 脚本 1

将下列代码写入脚本 lab3_1.py。

```
import paramiko
import time
import getpass
import sys
import re
import socket

username = input("Username: ")
password = getpass.getpass("Password: ")
iplist = open('ip_list.txt', 'r+')

switch_upgraded = []
switch_not_upgraded = []
```

```
    switch_with_tacacs_issue = []
    switch_not_reachable = []

    for line in iplist.readlines():
        try:
            ip_address = line.strip()
            ssh_client = paramiko.SSHClient()
            ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
            ssh_client.connect(hostname=ip_address, username=username, password=password)
            print ("Successfully connect to ", ip_address)
            command = ssh_client.invoke_shell(width=300)
            command.send("show inventory | i PID: WS\n")
            time.sleep(0.5)
            command.send("show flash: | i c2960\n")
            time.sleep(0.5)
            command.send("show boot | i BOOT path\n")
            time.sleep(0.5)
            output = command.recv(65535)
            command.send("wr mem\n")
            switch_model = re.search(r'WS-C2960\w?-\w{4, 5}-L', output)
            ios_version = re.search(r'c2960\w?-\w{8, 10}\d?-mz.\d{3}-\d{1, 2}.\w{2,
4}(.bin)?', output)
            boot_system = re.search(r'flash:.+mz.\d{3}-\d{1, 2}\.\w{2, 4}\.bin',
output)
            if switch_model.group() == "WS-C2960-24PC-L" and ios_version.group() ==
"c2960-lanbasek9-mz.122-55.SE12.bin" and boot_system.group() ==
'flash:c2960-lanbasek9-mz.122-55.SE12.bin' or boot_system.group() ==
'flash:/c2960-lanbasek9-mz.122-55.SE12.bin':
                switch_upgraded.append(ip_address)
            elif switch_model.group() == "WS-C2960S-F24PS-L" and ios_version.group()
== "c2960s-universalk9-mz.150-2.SE11.bin" and boot_system.group() ==
'flash:c2960s-universalk9-mz.150-2.SE11.bin' or boot_system.group() ==
'flash:/c2960s-universalk9-mz.150-2.SE11.bin':
```

```
                switch_upgraded.append(ip_address)
            elif switch_model.group() == "WS-C2960X-24PS-L" and ios_version.group()
== "c2960x-universalk9-mz.152-2.E8.bin" and boot_system.group() ==
'flash:c2960x-universalk9-mz.152-2.E8.bin' or boot_system.group() ==
'flash:/c2960x-universalk9-mz.152-2.E8.bin':
                switch_upgraded.append(ip_address)
            else:
                switch_not_upgraded.append(ip_address)
        except paramiko.ssh_exception.AuthenticationException:
            print ("TACACS is not working for " + ip_address + ".")
            switch_with_tacacs_issue.append(ip_address)
        except socket.error:
            print (ip_address +  " is not reachable.")
            switch_not_reachable.append(ip_address)

    iplist.close()
    ssh_client.close

    print ('\nTACACS is not working for below switches: ')
    for i in switch_with_tacacs_issue:
        print (i)

    print ('\nBelow switches are not reachable: ')
    for i in switch_not_reachable:
        print (i)

    print ('\nBelow switches IOS version are up-to-date: ')
    for i in switch_upgraded:
        print (i)

    print ('\nBelow switches IOS version are not updated yet: ')
    for i in switch_not_upgraded:
        print (i)
```

5.3.6　脚本 1 代码分段讲解

（1）前面导入模块，创建 4 个空列表，用来统计有多少交换机已经成功升级 IOS 版本，有多少交换机没有升级 IOS 版本，有哪些交换机因为 TACACS 或者管理 IP 地址不可达而无法登录，以及配合 for loop 和 Paramiko 依次循环 SSH 登录所有交换机，这些部分都是老生常谈的话题，这里不再赘述。

```
import paramiko
import time
import getpass
import sys
import re
import socket

username = input("Username: ")
password = getpass.getpass("Password: ")
iplist = open('ip_list.txt', 'r+')

switch_upgraded = []
switch_not_upgraded = []
switch_with_tacacs_issue = []
switch_not_reachable = []

for line in iplist.readlines():
    try:
        ip_address = line.strip()
        ssh_client = paramiko.SSHClient()
        ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        ssh_client.connect(hostname=ip_address, username=username, password=password)
        print ("Successfully connect to ",  ip_address)
```

（2）因为后面要用 show flash: | i c2960 查看闪存下面的 IOS 文件，而 IOS 文件名通常很长，

如果不调整宽度，则会导致后面截取的 output 不完整，从而影响正则表达式对关键词的匹配。所以可以用 command = ssh_client.invoke_shell(width=300)来调整 Paramiko 回显内容的宽度（默认为 100）。

```
    command = ssh_client.invoke_shell(width=300)
```

（3）command.send("show inventory | i PID: WS\n")、command.send("show flash: | i c2960\n")和 command.send("show boot | i BOOT path\n")这 3 个命令的作用前面已经讲过，这里给大家看下在生产网络下的 2960、 2960S 和 2960X 交换机中执行这 3 条命令得到的输出结果（均为 IOS 上传完毕，交换机重启前的输出结果），如下所示。

2960

```
AS-2960-EC3-G-3630#show inventory | i PID: WS
PID: WS-C2960-24PC-L    , VID: V02   , [redacted]
AS-2960-EC3-G-3630#
AS-2960-EC3-G-3630#
AS-2960-EC3-G-3630#show flash: | i c2960
     2  -rwx      9827106   Mar 4 2018 09:32:04 +03:00  c2960-lanbasek9-mz.122-55.SE12.bin
     4  drwx          512   Mar 1 1993 03:08:37 +03:00  c2960-lanbase-mz.122-44.SE2
AS-2960-EC3-G-3630#
AS-2960-EC3-G-3630#
AS-2960-EC3-G-3630#show boot | i BOOT path
BOOT path-list      : flash:c2960-lanbasek9-mz.122-55.SE12.bin
AS-2960-EC3-G-3630#
```

知乎 @弈心

2960S

```
AS-2960-EC3-G-3748#show inventory | i PID: WS
PID: WS-C2960S-F24PS-L , VID: V01  , [redacted]
AS-2960-EC3-G-3748#
AS-2960-EC3-G-3748#
AS-2960-EC3-G-3748#show flash: | i c2960
     4  -rwx     14572032  May 10 2018 14:48:49 +03:00  c2960s-universalk9-mz.150-2.SE11.bin
     7  drwx          512   Mar 1 1993 03:16:11 +03:00  c2960s-universalk9-mz.150-2.SE5
AS-2960-EC3-G-3748#
AS-2960-EC3-G-3748#
AS-2960-EC3-G-3748#show boot | i BOOT path
BOOT path-list      : flash:c2960s-universalk9-mz.150-2.SE11.bin
```

知乎 @弈心

2960X

```
AS-2960-EC3-G-3622#show inventory | i PID: WS
PID: WS-C2960X-24PS-L  , VID: V04  , [redacted]
AS-2960-EC3-G-3622#
AS-2960-EC3-G-3622#
AS-2960-EC3-G-3622#show flash: | i c2960
     5  -rwx     21287936   Apr 9 2018 15:40:23 +03:00  c2960x-universalk9-mz.152-2.E8.bin
AS-2960-EC3-G-3622#
AS-2960-EC3-G-3622#
AS-2960-EC3-G-3622#show boot | i BOOT path
BOOT path-list      : flash:/c2960x-universalk9-mz.152-2.E8.bin
AS-2960-EC3-G-3622#
```

知乎 @弈心

从上面的 show inventory | i PID: WS 的回显内容可以看到，3 种 29XX 交换机的型号分别为 WS-C2960-24PC-L、WS-C2960S-F24PS-L 和 WS-C2960X-24PS-L，这里可以用正则表达式 switch_model = re.search(r'WS-C2960\w?-\w{4，5}-L'， output)来匹配。

再来看 show flash: | i c2960 的输出结果，我们得到了对应 3 种交换机型号的 IOS 的.bin 文件名分别为 c2960-lanbasek9-mz.122-55.SE12.bin、c2960s-universalk9-mz.150-2. SE11.bin 和 c2960x-universalk9-mz.152-2.E8.bin，可以用正则表达式 ios_version = re.search(r'c2960\w?-\w{8，10}\d?-mz.\d{3}-\d{1，2}.\w{2，4}(.bin)?'，output)来匹配。

同理，最后的 show boot | i BOOT path 分别得到了 flash:c2960-lanbasek9-mz.122-55.SE12.bin、flash:c2960s-universalk9-mz.150-2.SE11.bin 和 flash:/c2960x-universalk9-mz.152-2.E8.bin 3 种不同的 boot system 引导路径，用正则表达式 boot_system = re.search(r'flash:/?c2960\w?-\w{9，11}-mz.\d{3}-\d{1，2}.\w{2，4}.bin'， output)来匹配。如果你足够细心，则会发现 boot system 路径可以写成 flash:，也可以写成 flash:/，有些网络工程师喜欢加/号，有些网络工程师不喜欢加，所以在正则表达式中我们用 flash:/?将两种情况都匹配到了。

```
            command.send("show inventory | i PID: WS\n")
            time.sleep(0.5)
            command.send("show flash: | i c2960\n")
            time.sleep(0.5)
            command.send("show boot | i BOOT path\n")
            time.sleep(0.5)
            output = command.recv(65535)
            command.send("wr mem\n")
            switch_model = re.search(r'WS-C2960\w?-\w{4, 5}-L',  output)
            ios_version = re.search(r'c2960\w?-\w{8, 10}\d?-mz.\d{3}-\d{1, 2}.\w{2,
4}(.bin)?',  output)
            boot_system = re.search(r'flash:.+mz.\d{3}-\d{1, 2}\.\w{2, 4}\.bin',
output)
```

（4）用正则表达式匹配所有可能出现的输出结果后，接下来就可以用 if 语句配合 and 和 or 两个布尔逻辑运算来做判断，先看第一条匹配 2960 交换机的 if 语句。

```
            if switch_model.group() == "WS-C2960-24PC-L" and ios_version.group() ==
"c2960-lanbasek9-mz.122-55.SE12.bin" and boot_system.group() ==
'flash:c2960-lanbasek9-mz.122-55.SE12.bin' or boot_system.group() ==
'flash:/c2960-lanbasek9-mz.122-55.SE12.bin':
                switch_upgraded.append(ip_address)
```

整个语句可以这样翻译：如果交换机型号为 WS-2960-24PC-L 且交换机的 IOS 版本为 c2960-lanbasek9-mz.122-55.SE12.bin，并且 boot system 的引导路径为 flash:c2960-lanbasek9-mz.122-55.SE12.bin（不加/号）或者 flash:/c2960-lanbasek9-mz.122-55.SE12.bin（加/号），则将该交换机的管理 IP 地址加入 switch_upgraded 列表。

同理，再来看第二条匹配 2960S 交换机的 elif 语句。

```
            elif switch_model.group() == "WS-C2960X-24PS-L" and ios_version.group()
== "c2960x-universalk9-mz.152-2.E8.bin" and boot_system.group() ==
'flash:c2960x-universalk9-mz.152-2.E8.bin' or boot_system.group() ==
'flash:/c2960x-universalk9-mz.152-2.E8.bin':
                switch_upgraded.append(ip_address)
```

如果交换机型号为 WS-C2960S-F24PS-L 且交换机的 IOS 版本为 c2960s-universalk9-mz.150-2.SE11.bin，并且 flash:c2960s-universalk9-mz.150-2.SE11.bin（不加/号）或者 flash:/c2960s-universalk9-mz.150-2.SE11.bin（加/号），则将该交换机的 IP 地址加入 switch_upgraded 列表。

第三条匹配 29640X 的同理，这里略过不讲。最后如果任何交换机不满足以上 3 条匹配条件，则将它的管理 IP 地址加入 switch_not_upgraded 列表。

```
            elif switch_model.group() == "WS-C2960X-24PS-L" and ios_version.group()
== "c2960x-universalk9-mz.152-2.E8.bin" and boot_system.group() ==
'flash:c2960x-universalk9-mz.152-2.E8.bin' or boot_system.group() ==
'flash:/c2960x-universalk9-mz.152-2.E8.bin':
                switch_upgraded.append(ip_address)
            else:
                switch_not_upgraded.append(ip_address)
```

（5）最后将 switch_upgraded 和 switch_not_upgraded 两个列表的所有元素打印出来，这样就能清楚地看到哪些交换机已经成功升级，哪些还没有升级，以及它们的地址。其余的代码内容则略过不讲。

```
        except Paramiko.ssh_exception.AuthenticationException:
            print ("TACACS is not working for " + ip_address + ".")
            switch_with_tacacs_issue.append(ip_address)
        except socket.error:
            print (ip_address +  " is not reachable.")
            switch_not_reachable.append(ip_address)

    iplist.close()
    ssh_client.close

    print ('\nTACACS is not working for below switches: ')
    for i in switch_with_tacacs_issue:
        print (i)

    print ('\nBelow switches are not reachable: ')
    for i in switch_not_reachable:
        print (i)

    print ('\nBelow switches IOS version are up-to-date: ')
    for i in switch_upgraded:
        print (i)

    print ('\nBelow switches IOS version are not updated yet: ')
    for i in switch_not_upgraded:
        print (i)
```

5.3.7 脚本 1 验证

移动到/root/lab3，执行脚本 1（lab3_1.py），脚本自动登录 ip_list.txt 中所有交换机的管理 IP 地址，然后执行交换机重启前的验证步骤，可以看到 172.16.206.119 和 172.16.206.146 两个交换机重启前的 IOS 升级配置有误，如下所示。

```
[root@localhost lab3]# python lab3_1.py
Username: parry
Password:
Successfully connect to  172.16.206.39
Successfully connect to  172.16.206.40
Successfully connect to  172.16.206.41
Successfully connect to  172.16.206.42
Successfully connect to  172.16.206.43
Successfully connect to  172.16.206.119
Successfully connect to  172.16.206.128
Successfully connect to  172.16.206.129
Successfully connect to  172.16.206.145
Successfully connect to  172.16.206.146

TACACS is not working for below switches:

Below switches are not reachable:

Below switches IOS version are up-to-date:
```

```
172.16.206.39
172.16.206.40
172.16.206.41
172.16.206.42
172.16.206.43
172.16.206.128
172.16.206.129
172.16.206.145

Below switches IOS version are not updated yet:
172.16.206.119
172.16.206.146
```

5.3.8 实验准备 - 脚本 2

我们来看脚本 2，也就是交换机重启后，验证其 IOS 版本是否成功升级的脚本。

首先移动到 lab3 文件夹下，创建 lab3_2.py 脚本。

```
[root@CentOS-Python ~]# cd lab3
[root@CentOS-Python lab3]# vi lab3_2.py
```

然后在维护窗口时段重启所有交换机，相信读到这里的所有读者都有能力独自写一个脚本来批量重启交换机了。

5.3.9 实验代码 - 脚本 2

将下列代码写入脚本 lab3_2.py。

```
import paramiko
import time
import getpass
import sys
import re
import socket

username = input("Username: ")
password = getpass.getpass("Password: ")
iplist = open('ip_list.txt', 'r+')

switch_upgraded = []
switch_not_upgraded = []
switch_with_tacacs_issue = []
switch_not_reachable = []

for line in iplist.readlines():
    try:
        ip_address = line.strip()
        ssh_client = paramiko.SSHClient()
        ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        ssh_client.connect(hostname=ip_address,username=username,password=password)
        print ("Successfully connect to ", ip_address)
        command = ssh_client.invoke_shell(width=300)
        command.send("show ver | b SW Version\n")
        time.sleep(0.5)
        output = command.recv(65535)
        print (output)
```

```
        ios_version = re.search(r'\d{2}.\d\(\d{1, 2}\)\w{2, 4}',  output)
        if ios_version.group() == '12.2(55)SE12':
            switch_upgraded.append(ip_address)
        elif ios_version.group() == '15.2(2)E8':
            switch_upgraded.append(ip_address)
        elif ios_version.group() == '15.0(2)SE11':
            switch_upgraded.append(ip_address)
        else:
            switch_not_upgraded.append(ip_address)
    except paramiko.ssh_exception.AuthenticationException:
        print ("TACACS is not working for " + ip_address + ".")
        switch_with_tacacs_issue.append(ip_address)
    except socket.error:
        print (ip_address +  " is not reachable." )
        switch_not_reachable.append(ip_address)

iplist.close()
ssh_client.close

print ('\nTACACS is not working for below switches: ')
for i in switch_with_tacacs_issue:
    print (i)

print ('\nBelow switches are not reachable: ')
for i in switch_not_reachable:
    print (i)

print ('\nBelow switches IOS version are up-to-date: ')
for i in switch_upgraded:
    print (i)

print ('\nBelow switches IOS version are not updated yet: ')
for i in switch_not_upgraded:
    print (i)
```

5.3.10 脚本 2 代码分段讲解

（1）这段代码和脚本 1 基本相同，我们同样创建了 switch_upgraded 和 switch_not_upgraded 两个空列表，用来统计有哪些交换机在重启后 IOS 版本升级成功，有哪些交换机在重启后 IOS 版本升级不成功。

```
import paramiko
import time
import getpass
import sys
import re
import socket

username = input("Username: ")
password = getpass.getpass("Password: ")
iplist = open('ip_list.txt', 'r+')

switch_upgraded = []
switch_not_upgraded = []
switch_with_tacacs_issue = []
switch_not_reachable = []

for line in iplist.readlines():
    try:
        ip_address = line.strip()
        ssh_client = paramiko.SSHClient()
        ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        ssh_client.connect(hostname=ip_address, username=username, password=password)
        print ("Successfully connect to ", ip_address)
        command = ssh_client.invoke_shell(width=300)
        command.send("show ver | b SW Version\n")
```

```
        time.sleep(0.5)
        output = command.recv(65535)
        print (output)
```

（2）交换机重启后直接 show ver | b SW Version 查看当前 IOS 版本，再用正则表达式 ios_version = re.search(r'\d{2}.\d\(\d{1，2}\)\w{2，4}'，output)来匹配，只要能匹配到 12.2(55)SE12、15.2(2)E8、15.0(2)SE11 这 3 种 IOS 中的任意一个，就将交换机的管理 IP 地址添加到 switch_upgraded 列表，反之如果上述 3 种 IOS 都匹配不到，则将交换机的管理 IP 地址添加到 switch_not_reachable 列表，最后用 for 循环将 switch_upgraded 和 switch_not_reachable 中的元素一一打印出来，输出内容的格式和脚本 1 并无二致。

```
        ios_version = re.search(r'\d{2}.\d\(\d{1, 2}\)\w{2, 4}',  output)
        if ios_version.group() == '12.2(55)SE12':
            switch_upgraded.append(ip_address)
        elif ios_version.group() == '15.2(2)E8':
            switch_upgraded.append(ip_address)
        elif ios_version.group() == '15.0(2)SE11':
            switch_upgraded.append(ip_address)
        else:
            switch_not_upgraded.append(ip_address)
    except Paramiko.ssh_exception.AuthenticationException:
        print ("TACACS is not working for " + ip_address + ".")
        switch_with_tacacs_issue.append(ip_address)
    except socket.error:
        print (ip_address +  " is not reachable." )
        switch not reachable.append(ip address)

iplist.close()
ssh_client.close

print ('\nTACACS is not working for below switches: ')
for i in switch_with_tacacs_issue:
    print (i)
```

```
print ('\nBelow switches are not reachable: ')
for i in switch_not_reachable:
    print (i)

print ('\nBelow switches IOS version are up-to-date: ')
for i in switch_upgraded:
    print (i)

print ('\nBelow switches IOS version are not updated yet: ')
for i in switch_not_upgraded:
    print (i)
```

5.3.11 脚本 2 验证

移动至/root/lab3，执行脚本 lab3_2.py，可以看到在重启后，所有交换机的 IOS 都已升级成功，包括在运行脚本 lab3_1.py 时出问题的 172.16.206.119 和 172.16.206.146 两台交换机，如下所示。

```
[root@localhost lab3]# python lab3_2.py
Username: parry
Password:
Successfully connect to  172.16.206.39
Successfully connect to  172.16.206.40
Successfully connect to  172.16.206.41
Successfully connect to  172.16.206.42
Successfully connect to  172.16.206.43
Successfully connect to  172.16.206.119
Successfully connect to  172.16.206.128
Successfully connect to  172.16.206.129
Successfully connect to  172.16.206.145
Successfully connect to  172.16.206.146

TACACS is not working for below switches:

Below switches are not reachable:

Below switches IOS version are up-to-date:
172.16.206.39
172.16.206.40
172.16.206.41
172.16.206.42
172.16.206.43
172.16.206.119
172.16.206.128
172.16.206.129
172.16.206.145
172.16.206.146

Below switches IOS version are not updated yet:
```

第 6 章

Python 第三方库详解

在读完前面 5 章内容并动手实操后，相信读者已经对 Python 的基础知识和 Python 在网络运维中的应用有了一定程度的了解。在此基础上，本章将扩展讨论 Python 在网络运维自动化领域中十分重要的两个话题：“不支持 API 的传统设备”和“默认使用同步、单线程执行脚本的 Python 运行效率低下”的问题，并将举例给出解决这些问题的方案及其原理。针对“不支持 API 的传统设备”的话题将在 6.1～6.5 节中讨论，而在 6.6 和 6.7 两节中将讨论如何应对“Python 默认同步、单线程效率低下”的问题。

1. 不支持 API 的传统设备

众所周知，NETCONF（RFC 6241）这个基于 XML、用来替代 CLI 和 SNMP 的网络配置和网络管理协议最早诞生于 2006 年 12 月，由 IETF 以 RFC 4741 发布。在进行一番修订后，IETF 又于 2011 年 6 月以 RFC 6241 将其再次发布。在此之前的一些“上了年纪”的网络设备，比如思科经典的基于 IOS 操作系统的 Catalyst 系列交换机（2960、3560、3750、4500、6500）是不支持 NETCONF 的，更别说对正在学习 NetDevOps 的网络工程师来说耳熟能详的 JSON、XML、YAML 这些数据类型，以及 YANG 这个数据模型语言了。简而言之，这些古董级的设备是没有 API 的。API 这类“时髦”的东西是在思科为了顺应 SDN 时代潮流，进而推出新的 IOS-XE 操作系统后才逐渐融进思科设备的，比较有代表性的使用 IOS-XE 的设备有思科的 3850、9200、9300 等新一代 Catalyst 系列交换机。

由于思科在 21 世纪初期在数据通信行业“霸主”的地位，加上这些古老的基于 IOS

的设备确实经典耐用、长盛不衰，如今我们仍然能在很多企业和公司的现网里看到它们的存在（就像今天仍然有一批 Windows XP 和 Windows 7 的忠实用户在坚守阵地一样）。在 2016 年 5 月于美国拉斯维加斯举办的第 30 届 Interop 上（Interop 是由英国 Informa 会展公司每年举办一次的 IT 技术峰会），大会嘉宾，*Network Programmability and Automation: Skills for the Next-Generation Network Engineer* 一书的作者之一，网络运维自动化的先驱 Jason Edelman 在他的演讲中提到过目前市场上支持 NETCONF、API 的网络设备只占市场份额的 15%～20%，剩下的 80%以上都还是只支持 SSH 访问命令行的传统设备，如下图所示。

那么问题来了：既然 IOS 设备当前还占据着极大的市场份额，那么是否意味着 JSON、XML、YAML 等真的在这些古董级的设备里毫无用武之地呢？答案是否定的，本章的 6.1～6.5 节将分别介绍 TextFSM、ntc-template、Napalm、pyntc 这些 Python 第三方库和模板是如何将 JSON 和这些老古董们巧妙结合在一起的，以及它到底能帮我们在这些没有 API 的设备上做什么。在此之前，我们需要知道一些关于 JSON 的基础知识。

2. Python 默认同步、单线程效率低下的问题

对网络工程师来说，我们通常必须借助 Telnetlib、Paramiko 或 Netmiko 这些第三方开源模块才能通过 Telnet 或 SSH 来登录、操作、管理各种网络设备。因为 Python 默认的运行方式是同步、单线程的，也就意味着在不对脚本内容做额外调整的前提下，运行 Python 脚本的主机只能一台一台地登录设备执行代码。假设一个脚本登录一台交换机执行配置平均耗时 5s，那么在拥有 1 000 台交换机的大型企业网里就要耗时 5 000s 才能执行完脚本，效率太低。在 6.6 和 6.7 节中将分别介绍如何通过使用 netdev 和 Netmiko 实现异步和多线程来解决这个痛点，提升 Python 的工作效率。

6.1　JSON

JSON 诞生于 1999 年 12 月，是 JavaScript Programming Language（Standard ECMA-262 3rd Edition）的一个子集合，是一种轻量级的数据交换格式。虽然 JSON 基于 JavaScript

开发，但它是一种“语言无关”（Language Independent）的文本格式，并且采用 C 语言家族，比如 C、C++、C#、Java、Python 和 Perl 等语言的用法习惯，成为一门理想的数据交换语言（Data-Interchange Language）。

6.1.1 JSON 基础知识

JSON 的数据结构具有易读的特点，它由键值对（Collection）和对象（Object）组成，在结构上非常类似 Python 的字典（Dictionary），如下是一个典型的 JSON 数据格式。

```
{
"intf":"Gigabitethernet0/0",
"status":"up"
}
```

与 Python 的字典一样，JSON 的键值对也由冒号分开，冒号左边的"intf"和"status"即键值对的键（Name），冒号右边的"Gigabitethernet0/0"和"up"即键值对的值（Value），每组键值对之间都用逗号隔开。

JSON 与字典不一样的地方如下。

（1）JSON 里键的数据类型必须为字符串，而在字典里字符串、常数、浮点数或者元组等都能作为键的数据类型。

（2）JSON 里键的字符串内容必须使用双引号""括起来，不像字典里既可以用单引号，又可以用双引号来表示字符串。

JSON 里键值对的值又分为两种形式，一种形式是简单的值，包括字符串、整数等，比如上面的"Gigabitethernet0/0"和"up"就是一种简单的值。另一种形式被称为对象（Object），对象内容由大括号{}表示，对象中的键值对之间用逗号分开，**它们是无序的**，举例如下。

```
{"Vendor":"Cisco", "Model":"2960"}
```

当有多组对象存在时，我们将其称为 **JSON 阵列**（JSON Array），阵列以中括号[]表示，**阵列中的元素（即各个对象）是有序的（可以把它理解为列表）**，举例如下。

```
{
    "devices":[
        {"Vendor":"Cisco", "Model":"2960"},
        {"Vendor":"Cisco", "Model":"3560"},
        {"Vendor":"Cisco", "Model":"4500"}
        ]
}
```

6.1.2　JSON 在 Python 中的使用

Python 中已经内置了 JSON 模块，需要使用时只需 import json 即可，如下所示。

```
[root@CentOS-Python ~]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import json
>>>
```

JSON 模块主要有两种函数：json.dumps()和 json.loads()。前者是 JSON 的编码器（Encoder），用来将 Python 中的对象转换成 JSON 格式的字符串，如下所示。

```
>>> import json
>>>
>>> a = json.dumps('parry')
>>> type(a)
<class 'str'>
>>> type(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
<class 'str'>
>>>
>>> type({"c": 0, "b": 0, "a": 0})
<class 'dict'>
>>> type(json.dumps([1,2,3]))
<class 'str'>
```

由此可以看到，我们用 json.dumps()将 Python 中 3 种类型的对象：字符串（'parry'）、字典（{"c": 0, "b": 0, "a": 0}）、列表（json.dumps([1,2,3])）转换成了 JSON 格式的字符串，并用 type()函数进行了验证。

而 json.loads()的用法则是将 JSON 格式的字符串转换成 Python 的对象，如下所示。

```
>>> json_list = '[1,2,3]'
>>> type(json_list)
<class 'str'>
>>> python_list = json.loads(json_list)
>>> print (python_list)
[1, 2, 3]
>>> type(python_list)
<class 'list'>
>>>
>>> json_dictionary = '{"vendor":"Cisco", "model":"2960"}'
>>> python_dictionary = json.loads(json_dictionary)
>>> print (python_dictionary)
{'vendor': 'Cisco', 'model': '2960'}
>>> type(python_dictionary)
<class 'dict'>
>>> _
```

我们将两个 JSON 格式的字符串：'[1,2,3]'和 '{"vendor":"Cisco", "model":"2960"}'，用 json.loads()转换成了它们各自对应的 Python 对象：列表[1,2,3]和字典{"vendor":"Cisco", "model":"2960"},。需要注意的是，在创建对应 Python 字典类型的 JSON 字符串时，如果对键使用了单引号，则 Python 会返回“SyntaxError: invalid sytanx”，提示语法错误，如下所示。

```
>>>
>>> json_dictionary = '{'vendor':'Cisco', 'model':'2960'}'
  File "<stdin>", line 1
    json_dictionary = '{'vendor':'Cisco', 'model':'2960'}'
                        ^
SyntaxError: invalid syntax
>>> exit()
```

原因就是：**JSON 里键的字符串内容必须使用双引号""括起来**，不像字典里既可以用单引号，又可以用双引号来表示字符串，这点请务必注意。

6.2　正则表达式的痛点

作为本书的重点内容，我们知道，在 Python 中可以使用正则表达式来对字符串格式的文本内容做解析（Parse），从而匹配到我们感兴趣的文本内容，**但是这么做有一定限制**，比如我们需要用正则表达式从下面一台 2960 交换机 show ip int brief 命令的回显内容中找出哪些**物理端口**是 Up 的。

```
SW1#show ip int brief
Interface              IP-Address      OK? Method Status                Protocol
GigabitEthernet0/0     unassigned      YES unset  up                    up
GigabitEthernet0/1     unassigned      YES unset  down                  down
GigabitEthernet0/2     unassigned      YES unset  down                  down
GigabitEthernet0/3     unassigned      YES unset  down                  down
GigabitEthernet1/0     unassigned      YES unset  down                  down
GigabitEthernet1/1     unassigned      YES unset  down                  down
GigabitEthernet1/2     unassigned      YES unset  down                  down
GigabitEthernet1/3     unassigned      YES unset  down                  down
GigabitEthernet2/0     unassigned      YES unset  down                  down
GigabitEthernet2/1     unassigned      YES unset  down                  down
GigabitEthernet2/2     unassigned      YES unset  down                  down
GigabitEthernet2/3     unassigned      YES unset  down                  down
GigabitEthernet3/0     unassigned      YES unset  down                  down
GigabitEthernet3/1     unassigned      YES unset  down                  down
GigabitEthernet3/2     unassigned      YES unset  down                  down
GigabitEthernet3/3     unassigned      YES unset  down                  down
Vlan1                  192.168.2.11    YES NVRAM  up                    up
```

如果用常规思维的正则表达式 GigabitEhternet\d\/\d 来匹配，则会将除 VLAN1 外的所有物理端口都匹配上，显然这种做法是错误的，因为当前只有 GigabitEthernet0/0 这一个物理端口的状态是 Up 的，如下图所示。

REGULAR EXPRESSION

16 matches, 304 steps (~0ms)

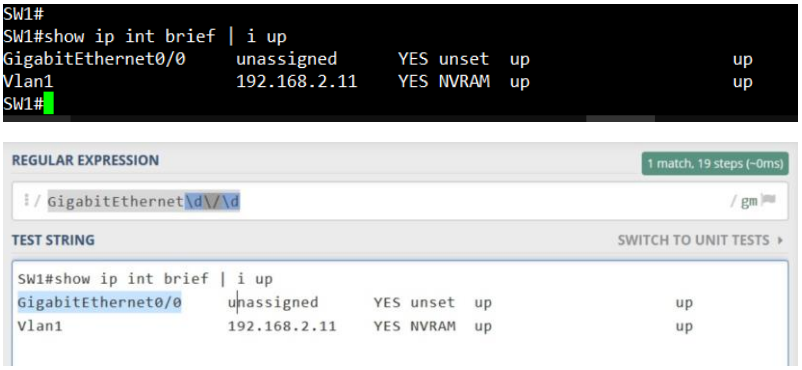
```
/ GigabitEthernet\d\/\d                                    / gm
```

TEST STRING

SWITCH TO UNIT TESTS

```
SW1#show ip int brief
Interface              IP-Address      OK? Method Status                Protocol
GigabitEthernet0/0     unassigned      YES unset  up                    up
GigabitEthernet0/1     unassigned      YES unset  down                  down
GigabitEthernet0/2     unassigned      YES unset  down                  down
GigabitEthernet0/3     unassigned      YES unset  down                  down
GigabitEthernet1/0     unassigned      YES unset  down                  down
GigabitEthernet1/1     unassigned      YES unset  down                  down
GigabitEthernet1/2     unassigned      YES unset  down                  down
GigabitEthernet1/3     unassigned      YES unset  down                  down
GigabitEthernet2/0     unassigned      YES unset  down                  down
GigabitEthernet2/1     unassigned      YES unset  down                  down
GigabitEthernet2/2     unassigned      YES unset  down                  down
GigabitEthernet2/3     unassigned      YES unset  down                  down
GigabitEthernet3/0     unassigned      YES unset  down                  down
GigabitEthernet3/1     unassigned      YES unset  down                  down
GigabitEthernet3/2     unassigned      YES unset  down                  down
GigabitEthernet3/3     unassigned      YES unset  down                  down
Vlan1                  192.168.2.11    YES NVRAM  up                    up
```

读者朋友也许此时想到了另一种方法，那就是在交换机的 show ip int brief 后面加上| i up 提前做好过滤，再用正则表达式 GigabitEhternet\d\/\d 来匹配，比如下图这样。



这种方法确实可以匹配出我们想要的结果，但是如果这时把要求变一变：找出当前交换机下所有 Up 的端口（注意，现在不再只是找出所有 Up 的物理端口，虚拟端口 Vlan1 也要算进去），并且同时给出它们的端口号及 IP 地址。

这个要求意味着不仅要用正则表达式匹配到上面文本里的 GigabitEthernet0/0、Vlan1、unassigned、192.168.2.11 四项内容，还要保证正则表达式没有匹配到其他诸如 YES、unset、NVRAM、up 等文本内容，怎么样？这个难度是不是立即上升了几个等级？

为了解决类似这样的正则表达式的痛点，我们必须搬出“救兵”**TextFSM** 了。

6.3　TextFSM 和 ntc-templates

TextFSM 最早是由 Google 开发的一个开源 Python 模块，它能使用自定义的变量和规则设计出一个模板（Template），然后用该模板来处理文本内容，将这些**无规律的文本内容**按照自己打造的模板来将它们整合成想要的**有序的数据格式**。

举个例子，在 6.2 节的例子中，如果有办法把 show ip int brief 的回显内容转换成 JSON，将它们以 JSON 阵列的数据格式列出来，是不是会很方便配合 for 循环来匹配出我们想要

的东西？比如看到下面这样的 JSON 阵列后，读者朋友们是不是想到点什么呢？

```
[
  {
    "intf": "GigabitEthernet0/0",
    "ipaddr": "unassigned",
    "status": "up",
    "proto": "up"
  },
  {
    "intf": "GigabitEthernet0/1",
    "ipaddr": "unassigned",
    "status": "down",
    "proto": "down"
  },
  {
    "intf": "GigabitEthernet0/2",
    "ipaddr": "unassigned",
    "status": "down",
    "proto": "down"
  },
  {
    "intf": "GigabitEthernet0/3",
    "ipaddr": "unassigned",
    "status": "down",
    "proto": "down"
  },
]
```

可以看到，JSON 阵列格式在 Python 中实际上是一个数据类型为列表的对象（以 [开头，以]结尾），既然是列表，那我们就能很方便地使用 for 语句遍历列表中的每个元素（这里所有的元素均为字典），并配合 if 语句，将端口状态为“up”（“status”键对应的值）的端口号（“intf”键对应的值）和 IP 地址（“ipaddr”键对应的值）一一打印出来即可（具体的代码将在 6.4 节中给出）。

下面介绍如何在 Python 中使用 TextFSM 创建我们需要的模板。

6.3.1　TextFSM 的安装

作为 Python 的第三方模块，TextFSM 有两种安装方法。

第一种方法是使用 pip 安装，如下所示。

```
[root@CentOS-Python templates]# pip3.8 install textfsm
Requirement already satisfied: textfsm in /usr/local/lib/python3.8/site-packages (1.1.0)
Requirement already satisfied: six in /usr/local/lib/python3.8/site-packages (from textfsm) (1.14.0)
Requirement already satisfied: future in /usr/local/lib/python3.8/site-packages (from textfsm) (0.18.2)
WARNING: You are using pip version 19.2.3, however version 20.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
[root@CentOS-Python templates]# _
```

pip 安装完毕后进入 Python 并 import textfsm，如果 Python 没有报错，则说明安装成功，如下所示。

```
[root@CentOS-Python templates]#
[root@CentOS-Python templates]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import textfsm
>>> _
```

第二种方法是使用 git clone 命令从 GitHub 下载 textfsm 的源码，如果你的 CentOS 8 主机没有安装 Git，系统会提醒你一并安装，具体如下所示。

```
Git clone https://github.com/google/textfsm.git
```

```
[root@CentOS-Python python]# git clone https://github.com/google/textfsm.git
Cloning into 'textfsm'...
remote: Enumerating objects: 150, done.
remote: Counting objects: 100% (150/150), done.
remote: Compressing objects: 100% (112/112), done.
remote: Total 346 (delta 90), reused 84 (delta 38), pack-reused 196
Receiving objects: 100% (346/346), 200.28 KiB | 15.00 KiB/s, done.
Resolving deltas: 100% (192/192), done.
```

源码下载后，会看到当前目录下多出一个 textfsm 文件夹。进入该文件夹后，会看到

如下所示的 setup.py 文件。

```
[root@CentOS-Python python]# ls
ip_list.txt  naplm_test.py  ntc-template.py  __pycache__  textfsm
[root@CentOS-Python python]# cd textfsm
[root@CentOS-Python textfsm]# ls
COPYING  examples  LICENSE.txt  MANIFEST.in  README.md  setup.cfg  setup.py  testdata  tests  textfsm
```

然后输入下面命令执行该 py 文件进行安装，如下所示。

```
python3.8 setup.py install
```

```
[root@CentOS-Python textfsm]# python3.8 setup.py install
running install
running bdist_egg
running egg_info
writing textfsm.egg-info/PKG-INFO
writing dependency_links to textfsm.egg-info/dependency_links.txt
writing entry points to textfsm.egg-info/entry_points.txt
writing requirements to textfsm.egg-info/requires.txt
writing top-level names to textfsm.egg-info/top_level.txt
reading manifest file 'textfsm.egg-info/SOURCES.txt'
reading manifest template 'MANIFEST.in'
writing manifest file 'textfsm.egg-info/SOURCES.txt'
installing library code to build/bdist.linux-x86_64/egg
running install_lib
running build_py
creating build/bdist.linux-x86_64/egg
creating build/bdist.linux-x86_64/egg/textfsm
```

6.3.2 TextFSM 模板的创建和应用

TextFSM 的语法本身并不难，但是使用者必须熟练掌握正则表达式。下面我们以思科 Nexus 7000 交换机上 show vlan 命令的回显内容为例，如下所示，来看如何用 TextFSM 创建模板，以及如何使用模板将该 show vlan 命令的回显内容整理成我们想要的数据格式。

```
N7K# show vlan

VLAN Name                             Status    Ports
---- ---------------------------------- ------------------------------
1    default                          active    Eth1/1, Eth1/2, Eth1/3
                                              Eth1/5, Eth1/6, Eth1/7
2    VLAN0002                         active    Po100, Eth1/49, Eth1/50
```

```
3    VLAN0003                    active    Po100, Eth1/49, Eth1/50
4    VLAN0004                    active    Po100, Eth1/49, Eth1/50
5    VLAN0005                    active    Po100, Eth1/49, Eth1/50
6    VLAN0006                    active    Po100, Eth1/49, Eth1/50
7    VLAN0007                    active    Po100, Eth1/49, Eth1/50
8    VLAN0008                    active    Po100, Eth1/49, Eth1/50
```

我们创建一个 TextFSM 模板，模板内容如下。

```
Value VLAN_ID (\d+)
Value NAME (\w+)
Value STATUS (\w+)

Start
  ^${VLAN_ID}\s+${NAME}\s+${STATUS}\s+ -> Record
```

（1）在 TextFSM 中，我们使用 Value 语句来定义变量，这里定义了 3 个变量，分别为 VLAN_ID、NAME 和 STATUS。

```
Value VLAN_ID (\d+)
Value NAME (\w+)
Value STATUS (\w+)
```

（2）每个变量后面都有它自己对应的正则表达式模式（Pattern），这些模式写在括号()中。比如变量 VLAN_ID 顾名思义是要去匹配 VLAN 的 Id 号的，所以它后面的正则表达式模式写为（\d+）。在 3.6 节中讲过，\d 这个特殊序列用来匹配数字，后面的+用来做贪婪匹配。同理，变量 NAME 是用来匹配 VLAN 的名称的，因为这里 VLAN 的名称掺杂了字母和数字，比如 VLAN0002，所以它的正则表达式模式写为（\w+）。\w 这个特殊序列用来匹配字母或数字，配合后面的+用来做贪婪匹配。同理，变量 STATUS（\w+）用来匹配 VLAN 状态，VLAN 状态会有 active 和 inactive 之分。

（3）在定义好变量后，我们使用 Start 语句来定义匹配规则，匹配规则由正则表达式的模式及变量名组成。

```
Start
  ^${VLAN_ID}\s+${NAME}\s+${STATUS}\s+ -> Record
```

（4）**Start 语句后面必须以正则表达式^开头**。^是正则表达式中的一种特殊字符，用于匹配输入字符串的开始位置，**注意紧随其后的$不是正则表达式里的$，它的作用并不是用来匹配输入字符串的结尾位置，而是用来调用我们之前设置好的 VLAN_ID 并匹配该变量**。注意，在 TextFSM 中调用变量时可以用大括号{}，写成${VLAN_ID}，也可以不用，写成$VLAN_ID，但是 TextFSM 官方推荐使用大括号。VLAN_ID 对应的正则表达式恰巧是\d+，这样就匹配到了 1、2、3、4、5、6、7、8 这些 VLAN_ID，而后面的\s+$则表示匹配 1、2、3、4、5、6、7、8 后面的空白字符（\s 这个特殊序列用来匹配空白字符）。

```
^${VLAN_ID}\s+
```

（5）同理，我们调用变量 NAME，它对应的正则表达式模式为\w+，该特殊序列用来匹配“show vlan”命令回显内容中的 default、VLAN0002、VLAN0003、…、VLAN0008 等内容，而后面的\s+则用来匹配之后所有的空白字符。

```
${NAME}\s+
```

（6）变量{STATUS}也一样，它对应的\w+用来匹配 active 和 inactive 这两种 VLAN 状态（例子中给出的 show vlan 的回显内容中没有 inactive，但是不影响理解）以及后面的空白字符。最后用-> Record 来结束 TextFSM 的匹配规则。

```
${STATUS}\s+ -> Record
```

在了解了 TextFSM 的语法基础后，接下来看怎么在 Python 中使用 TextFSM。首先将上面的 TextFSM 模板文件以文件名 show_vlan.template 保存，如下所示。

```
[root@CentOS-Python python]# cat show_vlan.template
Value VLAN_ID (\d+)
Value NAME (\w+)
Value STATUS (\w+)

Start
  ^&{VLAN_ID}\s+${NAME}\s+${STATUS}\s+ -> Record
[root@CentOS-Python python]#
```

然后在相同的文件夹下创建一个名叫 textfsm_demo.py 的 Python 脚本。

```
[root@CentOS-Python python]#
[root@CentOS-Python python]# vi textfsm_demo.py
```

将下面的代码写入该脚本。

```
from textfsm import TextFSM

output = '''
N7K# show vlan

VLAN Name                            Status    Ports
---- -------------------------------------------------------------------------
1    default                         active    Eth1/1, Eth1/2, Eth1/3
                                               Eth1/5, Eth1/6, Eth1/7
2    VLAN0002                        active    Po100, Eth1/49, Eth1/50
3    VLAN0003                        active    Po100, Eth1/49, Eth1/50
4    VLAN0004                        active    Po100, Eth1/49, Eth1/50
5    VLAN0005                        active    Po100, Eth1/49, Eth1/50
6    VLAN0006                        active    Po100, Eth1/49, Eth1/50
7    VLAN0007                        active    Po100, Eth1/49, Eth1/50
8    VLAN0008                        active    Po100, Eth1/49, Eth1/50
'''

f = open('show_vlan.template')
template = TextFSM(f)

print (template.ParseText(output))
```

代码分段讲解如下。

（1）首先我们用 from textfsm import TextFSM 引入 TextFSM 模块的 TextFSM 函数，该函数为 TextFSM 模块下最核心的类。

```
from textfsm import TextFSM
```

（2）将 show vlan 的回显内容以三引号字符串的形式赋值给变量 output。

```
output = '''
N7K# show vlan
```

```
VLAN Name                             Status    Ports
----------------------------------------------------------------------------
1    default                          active    Eth1/1, Eth1/2, Eth1/3
                                              Eth1/5, Eth1/6, Eth1/7
2    VLAN0002                         active    Po100, Eth1/49, Eth1/50
3    VLAN0003                         active    Po100, Eth1/49, Eth1/50
4    VLAN0004                         active    Po100, Eth1/49, Eth1/50
5    VLAN0005                         active    Po100, Eth1/49, Eth1/50
6    VLAN0006                         active    Po100, Eth1/49, Eth1/50
7    VLAN0007                         active    Po100, Eth1/49, Eth1/50
8    VLAN0008                         active    Po100, Eth1/49, Eth1/50
'''
```

（3）打开之前创建好的模板文件 show_vlan.template，调用 TextFSM()函数将它赋值给变量 template，最后调用 template 下的 ParseText()函数对文本内容做解析，ParseText()函数中的参数 output 即 show vlan 命令的回显内容，最后用 print()函数将被模板解析后的回显内容打印出来，看看是什么样的内容。

```
f = open('show_vlan.template')
template = TextFSM(f)

print (template.ParseText(output))
```

一切就绪后，执行脚本看效果，如下所示。

```
[root@CentOS-Python python]# python3.8 textfsm_demo.py
[['1', 'default', 'active'], ['2', 'VLAN0002', 'active'], ['3', 'VLAN0003', 'active'], ['4', 'VLAN0004', 'active'], ['5', 'VLAN0
005', 'active'], ['6', 'VLAN0006', 'active'], ['7', 'VLAN0007', 'active'], ['8', 'VLAN0008', 'active']]
```

由此可以看到，之前无序的纯字符串文本内容在被我们创建的 TextFSM 模板解析后，已经被有序的嵌套列表替代，方便我们配合 for 循环做很多事情。

6.3.3 ntc-templates

用 TextFSM 制作的模板很好用，但是缺点也很明显：每个 TextFSM 模板都只能对应一条 show 或者 display 命令的回显文本内容，而目前每家知名厂商的网络设备都有上百种

show 或 display 命令，并且每家厂商的回显内容和格式都完全不同，有些厂商还有多种不同的操作系统，比如思科就有 IOS、IOS-XE、IOS-XR、NX-OS、ASA、 WLC 等多种 OS 版本，这些版本又有各自特有的 show 命令，难道我们必须自己动手造轮子，每种操作系统的每条命令都要靠自己手动写一个对应的模板吗？不用担心，已经有前人帮我们造好了轮子，这就是 **ntc-templates**。

ntc-templates 是由 Network To Code 团队用 TextFSM 花费了无数心血开发出来的一套模板集，支持 Cisco IOS、Cisco ASA、Cisco NX-OS、Cisco IOS-XR、Arista、Avaya、Brocade、Checkpoint、Fortinet、Dell、Huawei、Palo Alto 等绝大多数主流厂商的设备，将在这些设备里输入 show 和 display 命令后得到的各式各样的回显文本内容整合成 JSON、XML、YAML 等格式。举例来说，针对思科的 IOS 设备，ntc-templates 提供了 show ip int brief、show cdp neighbor、show access-list 等常用的 show 命令对应的 TextFSM 模板（还有数百种思科和其他厂商的 TextFSM 模板等待读者自行去挖掘），如下所示。

```
-rw-r--r--. 1 root root   481 May  8 10:02 cisco_ios_dir.textfsm
-rw-r--r--. 1 root root  1470 May  8 10:02 cisco_ios_show_access-list.textfsm
-rw-r--r--. 1 root root   953 May  8 10:02 cisco_ios_show_adjacency_detail.textfsm
-rw-r--r--. 1 root root   310 May  8 10:02 cisco_ios_show_aliases.textfsm
-rw-r--r--. 1 root root   180 May  8 10:02 cisco_ios_show_archive.textfsm
-rw-r--r--. 1 root root   403 May  8 10:02 cisco_ios_show_authentication_sessions.textfsm
-rw-r--r--. 1 root root  1096 May  8 10:02 cisco_ios_show_boot.textfsm
-rw-r--r--. 1 root root   264 May  8 10:02 cisco_ios_show_capability_feature_routing.textfsm
-rw-r--r--. 1 root root   937 May  8 10:02 cisco_ios_show_cdp_neighbors_detail.textfsm
-rw-r--r--. 1 root root   546 May  8 10:02 cisco_ios_show_cdp_neighbors.textfsm
-rw-r--r--. 1 root root   317 May  8 10:02 cisco_ios_show_clock.textfsm
-rw-r--r--. 1 root root  2327 May  8 10:02 cisco_ios_show_controller_t1.textfsm
-rw-r--r--. 1 root root   491 May  8 10:02 cisco_ios_show_dmvpn.textfsm
-rw-r--r--. 1 root root  1367 May  8 10:02 cisco_ios_show_dot1x_all.textfsm
-rw-r--r--. 1 root root   413 May  8 10:02 cisco_ios_show_environment_power_all.textfsm
-rw-r--r--. 1 root root  1603 May  8 10:02 cisco_ios_show_environment_temperature.textfsm
-rw-r--r--. 1 root root  1085 May  8 10:02 cisco_ios_show_etherchannel_summary.textfsm
-rw-r--r--. 1 root root   454 May  8 10:02 cisco_ios_show_hosts_summary.textfsm
-rw-r--r--. 1 root root   401 May  8 10:02 cisco_ios_show_interfaces_description.textfsm
-rw-r--r--. 1 root root   484 May  8 10:02 cisco_ios_show_interfaces_status.textfsm
-rw-r--r--. 1 root root  1181 May  8 10:02 cisco_ios_show_interfaces_switchport.textfsm
-rw-r--r--. 1 root root  2082 May  8 10:02 cisco_ios_show_interfaces.textfsm
-rw-r--r--. 1 root root   312 May  8 10:02 cisco_ios_show_interface_transceiver.textfsm
-rw-r--r--. 1 root root   570 May  8 10:02 cisco_ios_show_inventory.textfsm
-rw-r--r--. 1 root root  2000 May  8 10:02 cisco_ios_show_ip_access-lists.textfsm
-rw-r--r--. 1 root root   525 May  8 10:02 cisco_ios_show_ip_arp.textfsm
-rw-r--r--. 1 root root  1272 May  8 10:02 cisco_ios_show_ip_bgp_neighbors.textfsm
-rw-r--r--. 1 root root   502 May  8 10:02 cisco_ios_show_ip_bgp_summary.textfsm
-rw-r--r--. 1 root root  1954 May  8 10:02 cisco_ios_show_ip_bgp.textfsm
-rw-r--r--. 1 root root   711 May  8 10:02 cisco_ios_show_ip_device_tracking_all.textfsm
-rw-r--r--. 1 root root   663 May  8 10:02 cisco_ios_show_ip_eigrp_neighbors.textfsm
-rw-r--r--. 1 root root  2407 May  8 10:02 cisco_ios_show_ip_eigrp_topology.textfsm
-rwxr-xr-x. 1 root root   459 May  8 10:02 cisco_ios_show_ip_flow_toptalkers.textfsm
-rw-r--r--. 1 root root   288 May  8 10:02 cisco_ios_show_ip_interface_brief.textfsm
-rw-r--r--. 1 root root  1778 May  8 10:02 cisco_ios_show_ip_interface.textfsm
```

我们可以通过第 4 章讲的 Netmiko 模块来调用 TextFSM 和 ntc-tempalte。接下来就以实验的形式演示如何使用 ntc-template。

（1）首先确认主机的 Python 里安装了 Netmiko 模块，如下所示。

```
[root@CentOS-Python /]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import netmiko
>>>
```

（2）在根目录下创建一个名为 ntc-template 的文件夹，如下所示。

```
[root@CentOS-Python /]#
[root@CentOS-Python /]#
[root@CentOS-Python /]# mkdir /ntc-template
```

（3）移动到该文件夹下，如下所示。

```
[root@CentOS-Python ~]# cd /ntc-template
[root@CentOS-Python ntc-template]#
```

（4）然后用下面的 git clone 命令下载 ntc-template.git 文件，如下所示。

```
git clone https://github.com/networktocode/ntc-templates.git
```

```
[root@CentOS-Python ntc-template]# git clone https://github.com/networktocode/ntc-templates.git
Cloning into 'ntc-templates'...
remote: Enumerating objects: 48, done.
remote: Counting objects: 100% (48/48), done.
remote: Compressing objects: 100% (45/45), done.
remote: Total 7554 (delta 21), reused 18 (delta 3), pack-reused 7506
Receiving objects: 100% (7554/7554), 1.97 MiB | 6.00 KiB/s, done.
Resolving deltas: 100% (4142/4142), done.
[root@CentOS-Python ntc-template]#
```

知乎 @弈心

（5）安装完成后，可以用 ls 命令看到在/ntc-template 下面多出了另一个 ntc-templates 文件夹，再依次输入命令 cd ntc-templates/templates/和 ls，就能看到全部的 ntc-templates 模板集了，如下所示。

```
[root@CentOS-Python ntc-template]# ls
ntc-templates
[root@CentOS-Python ntc-template]# cd ntc-templates/
[root@CentOS-Python ntc-templates]# ls
CHANGELOG  development_scripts.py  lib  LICENSE  MANIFEST.in  pyproject.toml  README.md  setup.py  templates  tests  tox.ini
[root@CentOS-Python ntc-templates]# cd templates/
[root@CentOS-Python templates]# ls
alcatel_aos_show_vlan.textfsm                          cisco_ios_show_adjacency_detail.textfsm                cisco_nxos_show_route-map.textfsm
alcatel_sros_oam_mac-ping.textfsm                      cisco_ios_show_aliases.textfsm                         cisco_nxos_show_vdc.textfsm
alcatel_sros_show_router_bgp_routes_vpn-ipv4.textfsm   cisco_ios_show_archive.textfsm                         cisco_nxos_show_version.textfsm
alcatel_sros_show_service_id_base.textfsm              cisco_ios_show_authentication_sessions.textfsm         cisco_nxos_show_vlan.textfsm
arista_eos_bash_df_-h.textfsm                          cisco_ios_show_boot.textfsm                            cisco_nxos_show_vpc.textfsm
arista_eos_dir_flash.textfsm                           cisco_ios_show_capability_feature_routing.textfsm      cisco_nxos_show_vrf_interface.textfsm
arista_eos_show_boot-config.textfsm                    cisco_ios_show_cdp_neighbors_detail.textfsm            cisco_nxos_show_vrf.textfsm
arista_eos_show_clock.textfsm                          cisco_ios_show_cdp_neighbors.textfsm                   cisco_s300_show_version.textfsm
arista_eos_show_environment_cooling.textfsm            cisco_ios_show_clock.textfsm                           cisco_wlc_ssh_show_802.11a_cleanair_config.textfsm
arista_eos_show_environment_temperature.textfsm        cisco_ios_show_controller_t1.textfsm                   cisco_wlc_ssh_show_802.11a.textfsm
arista_eos_show_hostname.textfsm                       cisco_ios_show_dmvpn.textfsm                           cisco_wlc_ssh_show_advanced_802.11a_channel.textfsm
arista_eos_show_interfaces_status.textfsm              cisco_ios_show_dot1x_all.textfsm                       cisco_wlc_ssh_show_ap_config_general.textfsm
arista_eos_show_interfaces.textfsm                     cisco_ios_show_environment_power_all.textfsm           cisco_wlc_ssh_show_ap_summary.textfsm
arista_eos_show_interfaces_transceiver_detail.textfsm  cisco_ios_show_environment_temperature.textfsm        cisco_wlc_ssh_show_band-select.textfsm
arista_eos_show_interfaces_transceiver.textfsm         cisco_ios_show_etherchannel_summary.textfsm            cisco_wlc_ssh_show_cdp_neighbors_detail.textfsm
arista_eos_show_inventory.textfsm                      cisco_ios_show_hosts_summary.textfsm                   cisco_wlc_ssh_show_client_detail.textfsm
arista_eos_show_ip_access-lists.textfsm                cisco_ios_show_interfaces_description.textfsm          cisco_wlc_ssh_show_exclusionlist.textfsm
arista_eos_show_ip_arp.textfsm                         cisco_ios_show_interfaces_status.textfsm               cisco_wlc_ssh_show_inter[illegible]
arista_eos_show_ip_bgp_summary.textfsm                 cisco_ios_show_interfaces_switchport.textfsm           cisco_wlc_ssh_show_inven[illegible]
arista_eos_show_ip_bgp.textfsm                         cisco_ios_show_interfaces.textfsm                      cisco_wlc_ssh_show_mobility_summary.textfsm
arista_eos_show_ip_helper-address.textfsm              cisco_ios_show_interface_transceiver.textfsm           cisco_wlc_ssh_show_rf-profile_summary.textfsm
arista_eos_show_ip_interface_brief.textfsm             cisco_ios_show_inventory.textfsm                       cisco_wlc_ssh_show_sysinfo.textfsm
```

知乎 @弈心

（6）因为在 Netmiko 里是用变量 NET_TEXTFSM 来调用 ntc-templates 模板集的，所以还要用 export 来设置相应的环境变量，将 ntc-templates 模板集的完整路径/ntc-template/ntc-templates/templates 赋值给变量 NET_TEXTFSM，然后用 echo 命令检验，如下所示。

```
export NET_TEXTFSM='/ntc-template/ntc-templates/templates'
echo $NET_TEXTFSM
```

```
[root@CentOS-Python python]# export NET_TEXTFSM='/ntc-template/ntc-templates/templates'
[root@CentOS-Python python]# echo $NET_TEXTFSM
/ntc-template/ntc-templates/templates
[root@CentOS-Python python]#
```

注：export 命令只在当前生效，下次 CentOS 重启后用 export 命令设置的环境变量就会失效，可以使用 cd～命令回到当前用户的家目录，然后用 vi 编辑家目录下的.bashrc 文件，如下所示。

```
[root@CentOS-Python templates]# cd ~
[root@CentOS-Python ~]# vi .bashrc
```

将 export NET_TEXTFSM='/ntc-template/ntc-templates/templates' 写在.bashrc 的最下面并保存，这样下次 CentOS 重启后环境变量设置仍然有效，如下所示。

```
# .bashrc

# User specific aliases and functions

alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

export NET_TEXTFSM='/ntc-template/ntc-templates/templates'
```

（7）一切准备就绪后，创建一个 Python 3 的脚本，脚本内容如下。

```
from Netmiko import ConnectHandler
import json

SW1 = {
    'device_type': 'cisco_ios',
    'ip': '192.168.2.11',
    'username': 'python',
    'password': '123',
}

connect = ConnectHandler(**SW1)
print ("Sucessfully connected to " + SW1['ip'])
interfaces = connect.send_command('show ip int brief', use_textfsm=True)
print (json.dumps(interfaces, indent=2))
```

由上面代码可以看到，我们导入 Netmiko，以便配合使用 TextFSM 和 ntc-template（导

入 Netmiko 后就不用再导入 textfsm 模块了）。在 6.1 节中讲到，JSON 为 Python 的内置脚本，可以直接 import 导入，关于 Netmiko 其他基础用法请参考 4.2.2 节，这里不再赘述。

注意，我们在 Netmiko 的 send_command()函数中使用了 use_textfsm 参数，并将其设为 True，表示调用 TextFSM，而 TextFSM 调用的 ntc-template 模板这个步骤我们已经在前面第 6 步通过改变环境变量做好了，这里不用操心。最后用 JSON 的 dumps()函数将解析后的 show ip int brief 的回显文本内容打印出来，注意后面设置缩进的 indent = 2 不能省掉，如果写成 print (json.dumps(result))，则 Python 会报错。

（8）运行脚本看效果，完美地得到我们预期想要的将 show ip int brief 的回显内容转换为 JSON 阵列格式的效果，如下所示。

```
[root@CentOS-Python python]# python3.8 test.py
Sucessfully connected to 192.168.2.11
[
  {
    "intf": "GigabitEthernet0/0",
    "ipaddr": "unassigned",
    "status": "up",
    "proto": "up"
  },
  {
    "intf": "GigabitEthernet0/1",
    "ipaddr": "unassigned",
    "status": "down",
    "proto": "down"
  },
  {
    "intf": "GigabitEthernet0/2",
    "ipaddr": "unassigned",
    "status": "down",
    "proto": "down"
  },
  {
    "intf": "GigabitEthernet0/3",
    "ipaddr": "unassigned",
    "status": "down",
    "proto": "down"
  },
```

（9）将之前的 Python 脚本修改如下。

```
from Netmiko import ConnectHandler
```

```
import json

SW1 = {
    'device_type': 'cisco_ios',
    'ip': '192.168.2.11',
    'username': 'python',
    'password': '123',
}

connect = ConnectHandler(**SW1)
print ("Sucessfully connected to " + SW1['ip'])
interfaces = connect.send_command('show ip int brief', use_textfsm=True)
for interface in interfaces:
    if interface["status"] == 'up':
        print (f'{interface["intf"]} is up!  IP address: {interface["ipaddr"]}')
```

前面讲过的代码内容不再重复。需要注意的是，**在 Netmiko 调用 TextFSM 后，send_command()的返回值不再是字符串，而是列表。**所以这里的变量 interfaces 的数据类型为列表，列表里的每个元素又如上一步 JSON 阵列一样为 JSON 对象，因此可以配合 for 循环和 if 语句将所有端口状态为 Up 的端口及对应的 IP 地址都答应出来。

（10）最后再运行一次脚本，效果如下所示。

```
[root@CentOS-Python python]# python3.8 test.py
Sucessfully connected to 192.168.2.11
GigabitEthernet0/0 is up!  IP address: unassigned
Vlan1 is up!  IP address: 192.168.2.11
```

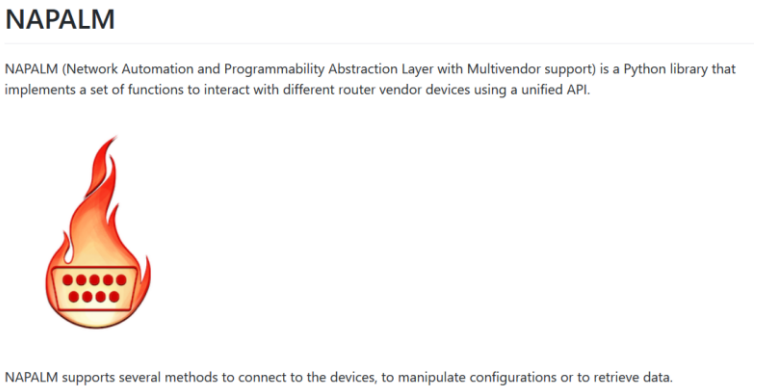
可以看到，在 ntc-templates 的协助下，我们成功地将 show ip int brief 的回显内容转换成了 JSON 阵列格式，并使用 for 循环和 if 语句找出了交换机 192.168.2.11 下所有 Up 的端口，以及对应的 IP 地址信息，完美地解决了正则表达式的痛点。

6.4　NAPALM

前面举例讲解了 JSON 的基础知识及用法，以及如何在 Netmiko 中使用 TextFSM 和 ntc-templates 来对旧型设备的各种 show 和 display 命令的回显内容进行解析，接下来介绍另一个十分强大的第三方开源模块：NAPALM，看看 NAPALM 是如何帮助我们对旧型 IOS 设备做网络运维时提供各种便利的。

6.4.1　什么是 NAPALM

NAPALM 全称为 Network Automation and Programmability Abstraction Layer with Multivendor support（在英语中，Napalm 是凝固汽油弹的意思，因此 NAPALM 的图标是一团火焰包围着一个串行接口，如下图所示）。顾名思义，NAPALM 是一种为多厂商的网络设备提供统一 API 的 Python 库，其源码可以在 GitHub 上下载。



截至 2020 年 5 月，NAPALM 支持 Cisco、Arista、Juniper 三家主流网络设备厂商的 5 种操作系统。

- Cisco IOS
- Cisco IOS-XR
- Cisco NX-OS

- Arista EOS
- Juniper JunOS

6.4.2 NAPALM 的优点

NAPALM 简单易用，首先它依赖 Netmiko，但是又不需要导入 Netmiko 便可独立使用（只需要保证运行脚本的主机安装了 Netmiko 即可），因此在脚本里节省了很多行代码，**让整个脚本更加简捷易懂**。

我们知道 TextFSM 可以配合 ntc-templates，将在网络设备中输入各种 show 和 display 命令后得到的无序的纯字符串类型的回显内容整合成有序的数据结构（列表类型），方便我们使用 for 和 if 等语句对这些文本内容做解析，从而解决 6.2 节提到的问题。而 NAPALM 提供的各种 API 也可以帮助我们在网络设备上得到我们感兴趣的设备信息和参数等内容，**并且 NAPALM 返回的数据类型也是列表**。目前 NAPALM 提供的 API 如下图所示，基本覆盖了网络运维中需要经常关注的各种网络信息和参数。

	EOS	IOS	IOSXR	JUNOS	NXOS	NXOS_SSH
get_arp_table	✅	✅	❌	❌	❌	✅
get_bgp_config	✅	✅	✅	✅	❌	❌
get_bgp_neighbors	✅	✅	✅	✅	✅	✅
get_bgp_neighbors_detail	✅	✅	✅	✅	❌	❌
get_config	✅	✅	✅	✅	✅	✅
get_environment	✅	✅	✅	✅	✅	✅
get_facts	✅	✅	✅	✅	✅	✅
get_firewall_policies	❌	❌	❌	❌	❌	❌
get_interfaces	✅	✅	✅	✅	✅	✅
get_interfaces_counters	✅	✅	✅	✅	❌	❌
get_interfaces_ip	✅	✅	✅	✅	✅	✅
get_ipv6_neighbors_table	❌	✅	❌	✅	❌	❌
get_lldp_neighbors	✅	✅	✅	✅	✅	✅
get_lldp_neighbors_detail	✅	✅	✅	✅	✅	✅
get_mac_address_table	✅	✅	✅	✅	✅	✅
get_network_instances	✅	✅	❌	✅	✅	❌
get_ntp_peers	❌	✅	✅	✅	✅	✅
get_ntp_servers	✅	✅	✅	✅	✅	✅
get_ntp_stats	✅	✅	✅	✅	✅	❌
get_optics	✅	✅	❌	✅	❌	❌
get_probes_config	❌	✅	✅	✅	❌	❌
get_probes_results	❌	❌	✅	✅	❌	❌
get_route_to	✅	✅	✅	✅	❌	✅
get_snmp_information	✅	✅	✅	✅	✅	✅
get_users	✅	✅	✅	✅	✅	✅
is_alive	✅	✅	✅	✅	✅	✅
ping	✅	✅	❌	✅	✅	✅
traceroute	✅	✅	✅	✅	✅	✅

NAPALM 的 API 分为 Getter 类和 Configuration 类，上面所列获取设备参数的 API 即 Getter 类，而 Configuration 类则支持对设备的配置做替换（Config.replace）、合并（Config.merge）、比对（Compare Config）、原子更换（Atomic Changes）、回滚（Rollback）等操作，功能比 TextFSM 和 ntc-template 更强大。

6.4.3 NAPALM 的缺点

前面讲到 ntc-template 支持 Cisco IOS、Cisco ASA、Cisco NX-OS、Cisco IOS-XR、Arista、Avaya、Brocade、Checkpoint、Fortinet、Dell、Huawei、Palo Alto 等绝大多数主流厂商的设备。而相较于 ntc-template，NAPLAM 目前支持的厂商数还较少。

由于 NAPALM 是被其他开发者按照他们自己的习惯和喜好提前造好的轮子，因此 NAPALM 的 API 返回的数据格式是统一、固定的，很难满足有个性化需求的使用者，比如在 6.3.3 小节的最后，我们用 TextFSM 配合 ntc-template 找出了一个 24 口的思科 2960 交换机中当前有哪些接口是 Up 的，以及这些端口对应的 IP 地址，如下所示。

```
[root@CentOS-Python python]# python3.8 test.py
Sucessfully connected to 192.168.2.11
GigabitEthernet0/0 is up!  IP address: unassigned
Vlan1 is up!  IP address: 192.168.2.11
```

从运行脚本后打印出的内容可以看到，这里 GigabitEthernet0/0 是 Up 的，其 IP 地址为 unassigned（二层端口）。另外，Vlan1 接口也是 Up 的，其 IP 地址是 192.168.2.11。但是 NAPALM 的 get_interfaces_ip()只能给出交换机里配置了 IP 地址的端口，那些端口状态为 Up，但是 IP 地址为 unassigned 的二层端口是不会被解析进去的，满足不了我们的需求。对同一个交换机使用 NAPALM 的 get_interfaces_ip()后的输出结果如下所示（缺失了 GigabieEthernet0/0）。

```
[root@CentOS-Python python]# python3 naplm_test.py
{
  "Vlan1": {
    "ipv4": {
      "192.168.2.11": {
        "prefix_length": 24
      }
    }
  }
}
[root@CentOS-Python python]# _
```

也许你会说 ntc-templates 里的各种模板也是提前造好的轮子，返回的数据格式也是统一固定的。是的，所以当我们对回显内容的解析有个性化需求时，推荐使用 TextFSM 来编

写自己需要的模板，关于 TextFSM 的使用可以参考 6.3.2 小节。

6.4.4　NAPALM 的安装

首先确认你的系统中安装了 Netmiko（可以使用命令 pip freeze | grep Netmiko，也可以进入 Python 解释器使用 import Netmiko 来确认），然后通过 pip 下载安装 NAPALM，如下所示。

```
pip install napalm
```

```
[root@CentOS-Python python]# pip3.8 freeze | grep netmiko
netmiko==3.1.0
[root@CentOS-Python python]# pip3.8 install napalm
Requirement already satisfied: napalm in /usr/local/lib/python3.8/site-packages (3.0.0)
Requirement already satisfied: junos-eznc>=2.2.1 in /usr/local/lib/python3.8/site-packages (from napalm) (2.4.1)
Requirement already satisfied: netaddr in /usr/local/lib/python3.8/site-packages (from napalm) (0.7.19)
Requirement already satisfied: pyeapi>=0.8.2 in /usr/local/lib/python3.8/site-packages (from napalm) (0.8.3)
Requirement already satisfied: netmiko>=3.1.0 in /usr/local/lib/python3.8/site-packages (from napalm) (3.1.0)
Requirement already satisfied: textfsm in /usr/local/lib/python3.8/site-packages (from napalm) (1.1.0)
Requirement already satisfied: cffi>=1.11.3 in /usr/local/lib/python3.8/site-packages (from napalm) (1.14.0)
Requirement already satisfied: scp in /usr/local/lib/python3.8/site-packages (from napalm) (0.13.2)
Requirement already satisfied: future in /usr/local/lib/python3.8/site-packages (from napalm) (0.18.2)
Requirement already satisfied: ciscoconfparse in /usr/local/lib/python3.8/site-packages (from napalm) (1.5.4)
Requirement already satisfied: paramiko>=2.6.0 in /usr/local/lib/python3.8/site-packages (from napalm) (2.7.1)
Requirement already satisfied: pyYAML in /usr/local/lib/python3.8/site-packages (from napalm) (5.3.1)
Requirement already satisfied: requests>=2.7.0 in /usr/local/lib/python3.8/site-packages (from napalm) (2.23.0)
Requirement already satisfied: setuptools>=38.4.0 in /usr/local/lib/python3.8/site-packages (from napalm) (41.2.0)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.8/site-packages (from napalm) (2.11.2)
Requirement already satisfied: lxml>=4.3.0 in /usr/local/lib/python3.8/site-packages (from napalm) (4.5.0)
Requirement already satisfied: six in /usr/local/lib/python3.8/site-packages (from junos-eznc>=2.2.1->napalm) (1.14.0)
Requirement already satisfied: ncclient>=0.6.3 in /usr/local/lib/python3.8/site-packages (from junos-eznc>=2.2.1->napalm) (0.6.7)
Requirement already satisfied: pyserial in /usr/local/lib/python3.8/site-packages (from junos-eznc>=2.2.1->napalm) (3.4)
Requirement already satisfied: ntc-templates in /usr/local/lib/python3.8/site-packages (from junos-eznc>=2.2.1->napalm) (1.4.1)
Requirement already satisfied: transitions in /usr/local/lib/python3.8/site-packages (from junos-eznc>=2.2.1->napalm) (0.8.1)
Requirement already satisfied: pyparsing in /usr/local/lib/python3.8/site-packages (from junos-eznc>=2.2.1->napalm) (2.4.7)
Requirement already satisfied: yamlordereddictloader in /usr/local/lib/python3.8/site-packages (from junos-eznc>=2.2.1->napalm) (0.4.0)
Requirement already satisfied: pycparser in /usr/local/lib/python3.8/site-packages (from cffi>=1.11.3->napalm) (2.20)
Requirement already satisfied: passlib in /usr/local/lib/python3.8/site-packages (from ciscoconfparse->napalm) (1.7.2)
Requirement already satisfied: dnspython in /usr/local/lib/python3.8/site-packages (from ciscoconfparse->napalm) (1.16.0)
Requirement already satisfied: colorama in /usr/local/lib/python3.8/site-packages (from ciscoconfparse->napalm) (0.4.3)
Requirement already satisfied: bcrypt>=3.1.3 in /usr/local/lib/python3.8/site-packages (from paramiko>=2.6.0->napalm) (3.1.7)
Requirement already satisfied: pynacl>=1.0.1 in /usr/local/lib/python3.8/site-packages (from paramiko>=2.6.0->napalm) (1.3.0)
Requirement already satisfied: cryptography>=2.5 in /usr/local/lib/python3.8/site-packages (from paramiko>=2.6.0->napalm) (2.9.2)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.8/site-packages (from requests>=2.7.0->napalm) (1.25.9)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.8/site-packages (from requests>=2.7.0->napalm) (2.9)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.8/site-packages (from requests>=2.7.0->napalm) (3.0.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.8/site-packages (from requests>=2.7.0->napalm) (2020.4.5.1)
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.8/site-packages (from jinja2->napalm) (1.1.1)
Requirement already satisfied: terminal in /usr/local/lib/python3.8/site-packages (from ntc-templates->junos-eznc>=2.2.1->napalm) (0.4.0)
WARNING: You are using pip version 19.2.3, however version 20.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
[root@CentOS-Python python]# _
```

进入 Python，如果 import napalm 没有报错，则说明安装成功，如下所示。

```
[root@CentOS-Python python]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import napalm
>>> _
```

6.4.5 NAPALM 的应用

我们以实验的形式举两个例子来分别演示如何使用 NAPLAM 的 Getter 类和 Configuration 类的 API。

首先创建一个名为 napalm1.py 的 Python 脚本，如下所示。

```
[root@CentOS-Python python]# vi napalm1.py_
```

然后在脚本里写入如下代码。

```
from napalm import get_network_driver

driver = get_network_driver('ios')
SW1 = driver('192.168.2.11','python','123')
SW1.open()

output = SW1.get_arp_table()
print (output)
```

由上面代码可以看到，我们用作实验的交换机是一台 IP 地址为 192.168.2.11、用户名为 python、密码为 123 的思科 2960 交换机。因为 NAPALM 支持多厂商设备不同的操作系统，这里首先需要导入 NAPALM 的 get_network_driver 类，用来指定该交换机对应的操作系统 IOS。然后调用 NAPALM 的 open()方法，即完成了 SSH 远程登录交换机的操作。

接着对交换机调用 NAPALM 的 get_arp_table()方法，将它赋值给变量 output，最后将输出结果打印出来。

运行代码，效果如下所示。

```
[root@CentOS-Python python]# python3.8 napalm1.py
[{'interface': 'Vlan1', 'mac': '00:0C:29:9E:A6:8A', 'ip': '192.168.2.1', 'age':
51.0}, {'interface': 'Vlan1', 'mac': '0C:BA:9D:CE:80:01', 'ip': '192.168.2.11',
'age': 0.0}]
[root@CentOS-Python python]# _
```

可以看到我们**仅仅使用 6 行代码**，就将交换机 show ip arp 命令的回显内容转换成了一个有序的列表形式的数据结构，**NAPALM 是不是非常简单易用**？

如果你觉得输出结果不易读，那么我们也可以配合 JSON 模块在脚本里添加两行代码，将输出内容转换成 JSON 阵列，代码如下。

```
from napalm import get_network_driver
import json

driver = get_network_driver('ios')
SW1 = driver('192.168.2.11','python','123')
SW1.open()

output = SW1.get_arp_table()
print (json.dumps(output, indent=2))
```

再次运行脚本看效果，如下所示。

```
[root@CentOS-Python python]# python3.8 napalm1.py
[
  {
    "interface": "Vlan1",
    "mac": "00:0C:29:9E:A6:8A",
    "ip": "192.168.2.1",
    "age": 54.0
  },
  {
    "interface": "Vlan1",
    "mac": "0C:BA:9D:CE:80:01",
    "ip": "192.168.2.11",
    "age": 0.0
  }
]
[root@CentOS-Python python]# _
```

有关 NAPALM 的 Getter 类 API 就介绍到这里，其他比较常用的 Getter 类 API 包括 get_facts()、get_config()、get_interaces()、get_bgp_config()、get_bgp_interfaces()等，就留给读者们自行去尝试和使用吧。

接下来演示如何使用 NAPALM 的 Configuration 类 API。前面提到，Configuration 类 API 包括替换（Config.replace）、合并（Config.merge）、比对（Compare Config）、原子更换（Atomic Changes）、回滚（Rollback）5 项操作，这里举例说明最常用的合并和比对的用法。

Configuration 类的合并（Config.merge）实际就是给设备做配置，它的方法是首先创建一个扩展名为.cfg 的配置文件，将配置命令写入该配置文件；然后在脚本里使用 NAPALM 的 load_merge_candidate()函数读取配置文件中的命令并上传到目标设备；接着通过 commit_config()将这些命令在设备上配置执行。load_merge_candidate()函数是基于 SCP 协议向目标设备传送配置命令的，因此在使用之前，需要先在设备上开启 scp server，否则 NAPALM 会返回“napalm.base.exceptions.CommandErrorException: SCP file transfers are not enabled. Configure 'ip scp server enable' on the device.”的错误，如下所示。

```
[root@CentOS-Python python]# python3 napalm2.py
Traceback (most recent call last):
  File "napalm2.py", line 7, in <module>
    iosvl2.load_merge_candidate(filename='napalm_config.cfg')
  File "/usr/local/lib/python3.6/site-packages/napalm/ios/ios.py", line 319, in load_merge_candidate
    file_system=self.dest_file_system,
  File "/usr/local/lib/python3.6/site-packages/napalm/ios/ios.py", line 285, in _load_candidate_wrapper
    file_system=file_system,
  File "/usr/local/lib/python3.6/site-packages/napalm/ios/ios.py", line 624, in _scp_file
    TransferClass=FileTransfer,
  File "/usr/local/lib/python3.6/site-packages/napalm/ios/ios.py", line 689, in _xfer_file
    raise CommandErrorException(msg)
napalm.base.exceptions.CommandErrorException: SCP file transfers are not enabled. Configure 'ip scp server enable' on the device.
```

```
SW1#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW1(config)#ip scp server enable
```

在了解原理后，首先创建一个名为 napalm_config.cfg 的配置文件，将用来对交换机 VTY 的第 5～15 条线做基本配置的命令集写进该配置文件并保存，如下所示。

```
[root@CentOS-Python python]# cat napalm_config.cfg
line vty 5 15
transport input ssh
transport output ssh
login local
```

然后创建一个名为 napalm2.py 的脚本，如下所示。

```
[root@CentOS-Python python]# vi napalm2.py_
```

将下面的代码写入该脚本。

```
from napalm import get_network_driver

driver = get_network_driver('ios')
SW1 = driver('192.168.2.11', 'python', '123')
SW1.open()

SW1.load_merge_candidate(filename='napalm_config.cfg')
SW1.commit_config()
```

由上面代码可知，我们用 load_merge_candidate(filename='napalm_config.cfg')加载之前创建好的配置文件 napalm_config.cfg，然后配合 NAPALM 的 commit_config()方法执行该配置文件的配置命令，即完成了对交换机的配置任务。

在执行代码前，我们先检查一遍交换机 192.168.2.11 的配置，确认当前交换机没有 line vty 5 15 的配置，如下所示。

```
SW1#show run | s line vty
line vty 0 4
 login local
 transport input ssh
 transport output ssh
```

然后执行脚本（因为代码里没有使用任何 print()函数，所以执行代码后没有任何回显内容），如下所示。

```
[root@CentOS-Python python]# python3 napalm2.py
```

接着回到交换机检查配置，如下所示。

```
SW1#show run | s line vty
line vty 0 4
 login local
 transport input ssh
 transport output ssh
line vty 5 15
 login local
 transport input ssh
 transport output ssh
SW1#
SW1#show start | s line vty
line vty 0 4
 login local
 transport input ssh
 transport output ssh
line vty 5 15
 login local
 transport input ssh
 transport output ssh
SW1#
```

可以看到执行脚本后，NAPALM 已经完成了 VTY 5～15 线的配置，并且 show start | s line vty 的回显内容证实了在完成配置的同时，NAPALM 帮我们保存了该配置。

下面来看如何使用 Configuration 类的比对。首先我们手动将 line vty 5 15 下面的 login local、transport input ssh、transport output ssh 3 条命令都从交换机里移除，移除之后再 show run | s line vty 可以看到之前的 3 条命令已分别被 no login、transport input none 和 transport output none 替代，如下所示。

```
SW1(config)#line vty 5 15
SW1(config-line)#no login local
SW1(config-line)#no transport input
SW1(config-line)#no transport output
SW1(config-line)#end
SW1#
*May 15 10:22:05.047: %SYS-5-CONFIG_I: Configured from console by console
SW1#show run | s line vty
line vty 0 4
 login local
 transport input ssh
 transport output ssh
line vty 5 15
 no login
 transport input none
 transport output none
SW1#
```

然后创建一个名为 napalm3.py 的脚本，如下所示。

```
[root@CentOS-Python python]# vi napalm3.py
```

将下面的代码写入该脚本。

```
from napalm import get_network_driver

driver = get_network_driver('ios')
SW1 = driver('192.168.2.11', 'python', '123')
SW1.open()

SW1.load_merge_candidate(filename='napalm_config.cfg')

differences = SW1.compare_config()
if len(differences) > 0:
    print(differences)
    SW1.commit_config()
else:
    print('No changes needed.')
    SW1.discard_config()
```

由代码可知，我们照例用 load_merge_candidate(filename='napalm_config.cfg')加载之前创建好的配置文件 napalm_config.cfg，然后调用 NAPALM 的 compare_config()方法。顾名思义，compare_config()是 NAPALM 用来将 napalm_conig.cfg 文件里的配置命令和交换机当前的配置做比对用的，它的返回值（即比对的结果）的数据类型为字符串。我们将该比对结果赋值给变量 difference，然后用 if 语句配合 len()函数对比对结果 difference 做分析，如果 len()返回的整数大于 0，则说明 difference 的内容为非空，也说明配置文件的命令和交换机当前的配置有区别，我们将比对结果打印出来，再调用 commit_config()将漏掉的命令补上。反之，如果 len()返回的整数为 0，则说明配置文件里的命令和交换机当前的配置一致，打印内容“No changes needed.”提醒用户不需要对交换机配置做任何更改，并调用 NAPALM 的 discard_config()方法来放弃之前通过 load_merge_candidate(filename='napalm_config.cfg')从配置文件 napalm_config.cfg 里加载好的配置命令。

执行脚本看效果，如下所示。

```
[root@CentOS-Python python]# python3 napalm3.py
+transport input ssh
+transport output ssh
+login local
[root@CentOS-Python python]# _
```

注意到 3 条命令前的 3 个加号+了吗？它代表比对内容的变量 difference 的具体内容，表示 compare_config()方法在将配置文件里的配置命令和交换机现有的配置做比对后，发现交换机还缺少这 3 条命令，然后通过 SW1.commit_config()将这 3 条命令在交换机里补上。我们登录交换机来进行验证，如下所示。

```
SW1#show run | s line vty
line vty 0 4
 login local
 transport input ssh
 transport output ssh
line vty 5 15
 login local
 transport input ssh
 transport output ssh
SW1#
```

可以发现，NAPALM 将交换机缺少的 login local、transport input ssh 和 transport output ssh 这 3 条命令又补回来了。

6.5 pyntc

除了 NAPALM，pyntc 也是一个非常优秀的用来管理网络设备配置、升级网络设备 OS、重启网络设备的第三方开源模块。也许从名字上你已经猜出来了，是的，同 ntc-templates 一样，pyntc 也是由 Network to Code 团队开发制作的，其源码存放在 GitHub 上。

6.5.1　pyntc 和 NAPALM 的比较

同 NAPALM 一样，pyntc 既依赖 Netmiko（如果对象设备使用的是思科 IOS 操作系统），也可以在脚本里独立使用（只需要保证运行脚本的主机安装了 Netmiko 即可）。这个特性使得 pyntc 拥有同 NAPALM 一样的一大优点：让网络运维的脚本代码简捷、易懂、便于维护。

与为不同厂商设备提供统一 API 的 NAPALM 不一样，pyntc 是一种多厂商、多 API 的模块，截至 2020 年 5 月，Pyntc 支持包括 Cisco、Arista、Juniper 在内的 3 家主流设备厂商的 4 种操作系统。

- Cisco IOS platforms - uses SSH (Netmiko)
- Cisco NX-OS - uses pynxos (NX-API)
- Arista EOS - uses pyeapi (eAPI)
- Juniper Junos - uses PyEz (NETCONF)

可以看到，针对不同的操作系统，pyntc 提供的 API 是不一样的。针对使用思科 IOS 的传统设备，因为它们不支持 API，所以 pyntc 借助的是 Netmiko 提供的 SSH 功能来对这些设备进行访问和管理。针对使用 NX-OS 操作系统的思科 Nexus 系列设备，pyntc 使用的是 NX-OS 自带的 NX-API。针对另外两个厂商：Arista 和 Juniper，pyntc 则分别依赖它们自带的 eAPI 和 NETCONF 来对设备进行访问和管理。

6.5.2　pyntc 的安装

作为 Python 的第三方模块，pyntc 有两种安装方法。

第一种方法是使用 pip 直接安装，如下所示。

```
[root@CentOS-Python ~]# pip3.8 install pyntc
Collecting pyntc
  Using cached https://files.pythonhosted.org/packages/69/11/5845a8062f379259ef8
748ad1e0cc39a6abd4cdb422c5233b4455601d96a/pyntc-0.0.9-py2.py3-none-any.whl
Requirement already satisfied: requests>=2.7.0 in /usr/local/lib/python3.8/site-
packages (from pyntc) (2.23.0)
Collecting pynxos>=0.0.3 (from pyntc)
  Using cached https://files.pythonhosted.org/packages/f6/37/2f48df4db4fd130d448
c23eb8db5af48ae39363f2c7afa70cb4f4c6cc786/pynxos-0.0.5.tar.gz
Requirement already satisfied: paramiko in /usr/local/lib/python3.8/site-package
s (from pyntc) (2.7.1)
Requirement already satisfied: future in /usr/local/lib/python3.8/site-packages
(from pyntc) (0.18.2)
Requirement already satisfied: netmiko in /usr/local/lib/python3.8/site-packages
 (from pyntc) (3.1.0)
Collecting coverage (from pyntc)
  Downloading https://files.pythonhosted.org/packages/52/38/53a28fcbe77f56f92d45
f9961feca85caebe7424b63a10b07aa8607a4f8d/coverage-5.1-cp38-cp38-manylinux1_x86_6
4.whl (229kB)
     |[redacted]| 235kB 232kB/s
Collecting bigsuds (from pyntc)
  Using cached https://files.pythonhosted.org/packages/0b/42/4884a0856c9c969910a
1585915502ad25163330af75f6247d92cfa3c404e/bigsuds-1.0.6.tar.gz
Collecting mock>=1.3 (from pyntc)
```

pip 安装完毕后进入 Python 并 import pyntc，如果 Python 没有报错，则说明安装成功，如下所示。

```
[root@CentOS-Python ~]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyntc
>>> _
```

第二种方法是使用 git clone 命令从 GitHub 下载 TextFSM 的源码并安装，如果你的 CentOS 8 主机没有安装 Git，则系统会提醒你一并安装。

```
git clone https://github.com/networktocode/pyntc.git
cd pyntc
python setup.py install
```

6.5.3 pyntc 的应用

我们以实验的形式来分别演示如何使用 pyntc 对使用思科 IOS 操作系统的目标设备完

成以下几项操作。

- 获取目标设备的基本信息。
- 对目标设备进行配置。
- 获取目标设备的 running config。
- 对目标设备的 running config 进行备份。
- 重启目标设备。

1. 获取目标设备的基本信息

首先创建一个名为 pyntc1.py 的脚本，将下列代码写入该脚本。

```
    import json
    from pyntc import ntc_device as NTC

    SW1 = NTC(host='192.168.2.11', username='python', password='123', device_
type='cisco_ios_ssh')
    SW1.open()

    print (json.dumps(SW1.facts, indent=4))
    SW1.close()
```

（1）ntc_device 是 pyntc 最重要的类，pyntc 通过它来 SSH 访问目标 IOS 设备，因为它的名字有点偏长，我们使用 from pyntc import ntc_device as NTC 将其命名为 NTC 并调用它。

（2）前面讲到，同 NAPALM 一样，pyntc 也依赖 Netmiko，因此访问设备的基本配置和方法大体是一样的。需要注意的是，因为我们要访问的目标设备是一台思科 IOS 设备，所以 NTC()里的 device_type 参数我们使用“cisco_ios_ssh”。

（3）同 NAPALM 一样，在访问目标设备的基本参数配置好后，再调用 open()函数即完成 SSH 远程登录交换机的操作。

（4）pyntc 中的 facts 方法用来读取目标设备的厂商、设备型号、OS 版本、序列号、主机名、uptime、端口列表、VLAN 等基本信息和配置，为了使输出的内容更具可读性，我们调用 json.dumps()来将输出内容转换成 JSON 的格式。

（5）最后使用 close()方法退出交换机，关闭 SSH 进程。

运行脚本看效果，如下所示。

```
[root@CentOS-Python python]# python3.8 pyntc1.py
{
    "model": "",
    "os_version": "",
    "serial_number": "",
    "hostname": "SW1",
    "vendor": "cisco",
    "uptime": 4260,
    "uptime_string": "00:01:11:00",
    "fqdn": "N/A",
    "interfaces": [
        "GigabitEthernet0/0",
        "GigabitEthernet0/1",
        "GigabitEthernet0/2",
        "GigabitEthernet0/3",
        "GigabitEthernet1/0",
        "GigabitEthernet1/1",
        "GigabitEthernet1/2",
```

```
        "GigabitEthernet1/3",
        "GigabitEthernet2/0",
        "GigabitEthernet2/1",
        "GigabitEthernet2/2",
        "GigabitEthernet2/3",
        "GigabitEthernet3/0",
        "GigabitEthernet3/1",
        "GigabitEthernet3/2",
        "GigabitEthernet3/3",
        "Vlan1"
    ],
    "vlans": [],
    "cisco_ios_ssh": {
        "config_register": "0x101"
    }
}
[root@CentOS-Python python]#
[root@CentOS-Python python]#
```

注：因为笔者使用的是 GNS3 模拟器上的虚拟交换机（vios_l2-adventerprisek9-m），所以 pyntc 读取不到“model”“os_version”“serial_number”几项信息，这个影响不大，

如果目标设备为真机，则不会有这个问题。

2. 对目标设备进行配置

在 pyntc 中，我们可以使用 config()和 config_list()对目标设备进行配置，两者的区别是前者一次只能对设备执行一条命令，而后者顾名思义可将多条命令作为元素放入一个列表中，pyntc 依次调用列表中的命令对设备进行配置。

创建一个名为 pyntc2.py 的脚本，将下列代码写入该脚本中。

```
    from pyntc import ntc_device as NTC

    SW1 = NTC(host='192.168.2.11', username='python', password='123',
device_type='cisco_ios_ssh')
    SW1.open()

    SW1.config('hostname pyntc_SW1')
    SW1.config_list(['router ospf 1', 'network 0.0.0.0 255.255.255.255 area 0'])
    SW1.close()
```

由代码可知，我们使用 config(‘hostame pyntc_SW1’)将交换机的主机名由 SW1 改为 pyntc_SW1，用 config_list([‘router ospf 1’, ‘network 0.0.0.0 255.255.255.255 area0’])在交换机上开启 OSPF。

运行脚本前，先手动登录交换机，确定目前的主机名为 SW1，并且还没有配置 OSPF，如下所示。

```
SW1#
SW1#show run | s router ospf
SW1#
```

运行脚本后，因为没有使用 print()函数，所以不会有任何回显内容，如下所示。

```
[root@CentOS-Python python]# python3.8 pyntc2.py
[root@CentOS-Python python]# _
```

回到交换机，如下所示，发现其主机名已经被 pyntc 配置为 pyntc_SW1，并且 OSPF 也已 开启。

```
pyntc_SW1#show run | s router ospf
router ospf 1
 network 0.0.0.0 255.255.255.255 area 0
pyntc_SW1#
```

3. 获取目标设备的 running config

通过 pyntc 的 running_config 方法，我们可以很方便地获取设备的 running config。该方法的返回值的数据类型为字符串，可以用 print()函数打印出来。这里创建一个名为 pyntc3.py 的脚本，将下列代码写入该脚本。

```
    from pyntc import ntc_device as NTC

    SW1 = NTC(host='192.168.2.11', username='python', password='123',
device_type='cisco_ios_ssh')
    SW1.open()

    run = SW1.running_config
    print (run)
```

```
SW1.close()
```

代码讲解部分略过，我们直接来看脚本运行后的效果（由于回显内容的长度过长，这里只截取其中一部分），如下所示。

```
[root@CentOS-Python python]# python3.8 pyntc3.py
Building configuration...

Current configuration : 3958 bytes
!
! Last configuration change at 07:53:36 UTC Thu May 21 2020 by python
!
version 15.2
service timestamps debug datetime msec
service timestamps log datetime msec
no service password-encryption
service compress-config
!
hostname pyntc_SW1
!
boot-start-marker
boot-end-marker
!
!
!
username python privilege 15 password 0 123
no aaa new-model
!
!
!
!
!
!
!
!
no ip domain-lookup
ip domain-name python.com
ip cef
no ipv6 cef
!
!
file prompt quiet
!
```

4. 对目标设备的 running config 进行备份

除了获取设备的 running config，我们也可以通过 pyntc 对设备的 running config 做备份。在 4.6 节中，我们介绍了如何将思科交换机的配置备份在 TFTP 服务器里，过程比较

烦琐，而且脚本的代码也比较多（总共 23 行代码）。这里笔者将演示如何通过 pyntc 的 backup_running_conig()方法，仅使用 5 行代码就将一台交换机的配置备份在本地主机上。

首先将下列代码写入名为 pyntc4.py 的脚本。

```
from pyntc import ntc_device as NTC

SW1 = NTC(host='192.168.2.11', username='python', password='123',
device_type='cisco_ios_ssh')
SW1.open()

SW1.backup_running_config('SW1_config.cfg')
SW1.close()
```

由代码可知，我们调用 pyntc 的 backup_running_config()对交换机的 running config 做备份，将它保存在一个名为 SW1_config.cfg 的配置文件里，该配置文件将在运行脚本的 CentOS 主机的当前目录下自动生成。

运行脚本前，我们首先在主机上通过命令 ls SW_config.cfg 确认当前目录下没有 SW1_config.cfg 文件，如下所示。

```
[root@CentOS-Python python]# ls SW1_config.cfg
ls: cannot access 'SW1_config.cfg': No such file or directory
[root@CentOS-Python python]# _
```

执行脚本看效果，如下所示。

```
[root@CentOS-Python python]# python3.8 pyntc4.py
[root@CentOS-Python python]# ls SW1_config.cfg
SW1_config.cfg
[root@CentOS-Python python]# cat SW1_config.cfg

Building configuration...

Current configuration : 3958 bytes
!
! Last configuration change at 07:53:36 UTC Thu May 21 2020 by python
!
version 15.2
service timestamps debug datetime msec
service timestamps log datetime msec
no service password-encryption
service compress-config
```

```
!
hostname pyntc_SW1
!
boot-start-marker
boot-end-marker
!
!
!
username python privilege 15 password 0 123
no aaa new-model
!
!
!
!
!
!
!
!
no ip domain-lookup
ip domain-name python.com
ip cef
no ipv6 cef
```

执行脚本后，可以看到当前目录多出了 SW1_config.cfg 配置文件，查看该文件内容，确认的确是交换机 SW1 的 running config，这样一台思科交换机配置的备份仅通过 5 行代码就完成了。

5. 重启目标设备

最后来看如何使用 pyntc 重启目标设备。首先将下列代码写入名为 pyntc5.py 的脚本。

```
from pyntc import ntc_device as NTC

SW1 = NTC(host='192.168.2.11', username='python', password='123',
device_type='cisco_ios_ssh')
SW1.open()

SW1.save()
SW1.reboot(confirm=True)
```

由代码可知，在重启设备前，首先要保证当前的配置已被保存，这里使用 pyntc 的 save()方法，它的功能等同于思科的 copy run start 命令，随后调用 pyntc 的 reboot()方法，并将参数 confirm 设为 True，表示确认要对设备重启。

执行脚本看效果，如下所示。

```
[root@CentOS-Python python]# python3 pyntc5.py
[root@CentOS-Python python]#
```

```
pyntc_SW1#
-Traceback= 1DDC418z 8DC255z 90582Ez 905550z 90535Dz 9014E5z 90211Bz 9020AFz 909578z 9083F7z 9078D7z 908A48z 8D73E3z 886E
D1z 8BADD3z 8B9F87z - Process "SSH Process", CPU hog, PC 0x0090963B

*May 21 09:39:30.891: %GRUB-5-CONFIG_WRITING: GRUB configuration is being updated on disk. Please wait...
pyntc_SW1#
*May 21 09:39:32.037: %GRUB-5-CONFIG_WRITTEN: GRUB configuration was written to disk successfully.
*May 21 09:39:34.379: %SYS-3-CPUHOG: Task is running for (2000)msecs, more than (2000)msecs (0/0),process = SSH Process.
pyntc_SW1#
*May 21 09:39:39.954: %SYS-5-RELOAD: Reload requested by python on vty0 (192.168.2.1). Reload Reason: Reload command.
pyntc_SW1#
 *May 21 09:39:43.085
Reload requested
```

执行脚本后，可以看到交换机上跳出了日志提示（这里我们通过 console 访问交换机），证明脚本对交换机执行了 copy run start 命令，保存了当前的配置，随后对交换机执行 reload 命令，重启设备，如下所示。

```
pyntc_SW1#show ver | i uptime
pyntc_SW1 uptime is 3 minutes
pyntc_SW1#show run | s router ospf
router ospf 1
 network 0.0.0.0 255.255.255.255 area 0
pyntc_SW1#
```

最后当交换机重新回到命令行时，发现之前配置的交换机的 hostname 及 OSPF 都被保存了下来。

6.6 netdev（异步并行）

netdev 是俄罗斯网络运维开发工程师 Sergey Yakovlev 开发的一套用来实现异步登录和配置网络设备的第三方开源模块。在讲解它的用法前，首先需要知道什么是**“同步”（Synchronous）**，什么是**“异步”（Asynchronous）**，以及为什么使用异步能够提升日常网络运维中的工作效率。

6.6.1　同步与异步

所谓同步，可以理解为每当系统执行完一段代码或者函数后，系统都将一直等待该段代码或函数的返回值或消息，直到系统接收返回值或消息后才继续执行下一段代码或函数，**在等待返回值或消息期间，程序处于阻塞状态，系统将不做任何事情**。

本书前面所有涉及管理多个设备的实验中，我们都是将设备的 IP 地址预先写入一个名为 ip_list.txt 的文本文件，然后在脚本里使用 open()函数将其打开，调用 readlines()函数并配合 for 循环读取每个设备的 IP 地址，再通过 Paramiko 或者 Netmiko 一台设备接一台设备地完成 SSH 登录。像这样 Python 一次只能登录一台设备，只有完成一台设备的配置后才能登录下一台设备继续配置的方式就是一种典型的“同步”。

而异步则恰恰相反，系统在执行完一段代码或者函数后，不用阻塞性地等待返回值或消息，而是继续执行下一段代码或函数，**在同一时间段里执行多个任务（而不是傻傻地等着一件事情做完并且结果出来后才去做下一件事情），将多个任务并行，从而提高程序的执行效率**。如果你读过数学家华罗庚的《统筹方法》，那么对其中所举的例子一定不会感到陌生：同样是沏茶的步骤，因为烧水需要一段时间，你不用等水煮沸了以后才来洗茶杯、倒茶叶（类似“同步”），而是在等待烧水的过程中就把茶杯洗好、把茶叶倒好，等水烧开了就能直接泡茶喝了，这里烧水、洗茶杯、倒茶叶 3 个任务是在同一时间段内并行完成的，这就是一种典型的“异步”。

同步和异步有一个相同点：它们都是单线程下的概念，关于单线程和多线程的比较会在 6.7 节中讨论。

6.6.2　异步在 Python 中的应用

自从 Python 在 3.4.x 版本起开始支持异步后，关于异步的 Python 语法几经更改，在

Python 3.4、Python 3.5、Python 3.7 中的实现方式有很大不同，本书后面的例子都将基于 Python 3.8.2 来讲解异步的使用。

要了解异步在 Python 中的应用，必须知道什么是**协程**（**Coroutine**）、什么是**任务**（**Task**）、什么是**可等待对象**（**Awaitable Object**）。

我们可以把协程理解为线程的优化，把协程看成一种微线程。它是一种比线程更节省资源、效率更高的系统调度机制。异步就是基于协程实现的。在 Python 中，实现协程的模块主要有 asyncio、gevent 和 tornado，使用较多的是 asyncio。首先来看下面的例子。

```
#coding=utf-8
import asyncio
import time

async def main():
    print('hello')
    await asyncio.sleep(1)
    print('world')

print (f"程序于 {time.strftime('%X')} 开始执行")
asyncio.run(main())
print (f"程序于 {time.strftime('%X')} 执行结束")
```

- 在 Python 中，我们通过在 def 语句前加上 async 语句将一个函数定义为协程函数，在上面的例子中，main()函数被定义为**协程函数**。
- 这里的 await asyncio.sleep(1)表示**临时中断**当前的函数 1s。如果程序中还有其他函数，则继续执行下一个函数，直到下一个函数执行完毕，再返回来执行 main()函数，因为除了一个 main()函数，就没有其他函数了，所以在 print('hello')后，main()函数休眠了 1s，然后继续 print(‘world’)。
- 协程函数不是普通的函数，不能直接用 main()来调用，需要使用 asyncio.run(main())才能执行该协程函数。
- 我们配合 time 模块的 strftime()函数来记录程序开始前的时间和程序结束后的时

间，可以看到总共耗时确实是 1s，如下所示。

```
[root@CentOS-Python python]# python3.8 1.py
程序于 06:11:02 开始执行
hello
world
程序于 06:11:03 执行结束
[root@CentOS-Python python]# _
```

需要注意的是，不要把 await asyncio.sleep(1)和 time.sleep(1)弄混，后者是在同步中使用的休眠操作，前者是在异步中使用的，因为只有一个 main()函数需要执行，所以暂时感受不到这两者的区别，不用着急，继续看下面的两个例子。

```
#coding=utf-8
import asyncio
import time

async def say_after(what, delay):
    print(what)
    await asyncio.sleep(delay)

async def main():
    print (f"程序于 {time.strftime('%X')} 执行结束")
    await say after('hello',1)
    await say_after('world',2)
    print (f"程序于 {time.strftime('%X')} 执行结束")

asyncio.run(main())
```

- 我们在协程函数 main()的基础上加入了另一个函数 say_after()。同样地，我们用 async 将它定义为协程函数。
- 我们在 main()函数中两次调用 say_after()函数，因为 say_after()函数是一个协程函数，因此在调用它时，前面必须加上 await。
- 当 main()函数第一次调用 say_after()函数时，首先打印出 hello，然后休眠 1s；第二次调用 say_after()函数时，打印出 world，然后再休眠 2s。两次调用总共花费

了 3s 来运行完整个程序，如下所示。

```
[root@CentOS-Python python]# python3.8 2.py
程序于08:41:16开始执行
hello
world
程序于 08:41:19 执行结束
[root@CentOS-Python python]# _
```

这时你会说，第一次花了 1 秒，第二次花了 2s，总共 3s 时间，这没节省时间啊，两次调用的 say_after()函数并没有被**并行**啊，这和同步有什么区别？别急，继续往下看。

```
#coding=utf-8
import asyncio
import time

async def say_after(what, delay):
    await asyncio.sleep(delay)
    print(what)

async def main():
task1 = asyncio.create_task(say_after('hello',1))
    task2 = asyncio.create_task(say_after('world',2))
    print (f"程序于 {time.strftime('%X')} 开始执行")
    await task1
    await task2
    print (f"程序于 {time.strftime('%X')} 执行结束")

asyncio.run(main())
```

- 要实现异步并行，需要将协程函数打包成一个**任务**，这里使用 asyncio 的 create_task()函数将 say_after()函数打包了两次，并分别赋值给 task1 和 task2 两个变量。然后使用 await 来调用 task1 和 task2 两个任务。
- 运行脚本后可以看到，因为 task1 和 task2 是并行执行的，所以程序总共只耗时 2s 即告完成（06:57:04 到 06:57:06）！如下所示。

```
[root@CentOS-Python python]# python3.8 3.py
程序于 06:57:04 开始执行
hello
world
程序于 06:57:06 执行结束
[root@CentOS-Python python]# _
```

最后来说什么是**可等待对象**。可等待对象的定义很简单：如果一个对象可以在 await 语句中使用，那么它就是**可等待**对象。可等待对象主要有 3 种类型：协程、任务和 Future。协程和任务前面已经讲过，Future 不在本书的讨论范围内，感兴趣的读者可以自己参阅其他材料深入学习。

6.6.3　netdev 的安装和应用

也许你会问，既然 Python 3.4 后加入了 asyncio 内置模块，那么为什么我们不能用 Paramiko、Netmiko、NAPALM 和 pyntc 来配合它实现异步呢？很遗憾的是，**由于异步在 Python 中引入较晚，上述所有这些模块都不支持前面讲到的可等待对象，也就无法支持异步并行**。

2019 年 4 月，受 Netmiko 项目的启发，俄罗斯网络运维开发工程师 Sergey Yakovlev 在 Netmiko 的基础上开发了一个叫作 netdev 的开源模块。该模块依赖 Netmiko，并且需要至少 Python 3.5 以上才能运行，最大的特点是支持对网络设备进行异步登录和操作。截至 2020 年 5 月，netdev 支持 7 家厂商的 12 种操作系统。

- Cisco IOS
- Cisco IOS XE
- Cisco IOS XR
- Cisco ASA

- Cisco NX-OS
- HP Comware (like V1910 too)
- Fujitsu Blade Switches
- Mikrotik RouterOS
- Arista EOS
- Juniper JunOS
- Aruba AOS 6.X
- Aruba AOS 8.X
- Terminal

下面介绍如何安装和使用 netdev。netdev 可以通过 pip 直接下载安装（因为 netdev 依赖 Netmiko，下载 netdev 前请确认你的 Python 主机是否已经安装了 Netmiko），如下所示。

```
[root@CentOS-Python python]# pip3.8 install netdev
Requirement already satisfied: netdev in /usr/local/lib/python3.8/site-packages (0.9.3)
Requirement already satisfied: PyYAML<6.0,>=5.1 in /usr/local/lib/python3.8/site-packages (from netdev
) (5.3.1)
Requirement already satisfied: asyncssh<2.0,>=1.15 in /usr/local/lib/python3.8/site-packages (from net
dev) (1.18.0)
Requirement already satisfied: cryptography>=2.7 in /usr/local/lib/python3.8/site-packages (from async
ssh<2.0,>=1.15->netdev) (2.9.2)
Requirement already satisfied: cffi!=1.11.3,>=1.8 in /usr/local/lib/python3.8/site-packages (from cryp
tography>=2.7->asyncssh<2.0,>=1.15->netdev) (1.14.0)
Requirement already satisfied: six>=1.4.1 in /usr/local/lib/python3.8/site-packages (from cryptography
>=2.7->asyncssh<2.0,>=1.15->netdev) (1.14.0)
Requirement already satisfied: pycparser in /usr/local/lib/python3.8/site-packages (from cffi!=1.11.3,
>=1.8->cryptography>=2.7->asyncssh<2.0,>=1.15->netdev) (2.20)
WARNING: You are using pip version 19.2.3, however version 20.1.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
[root@CentOS-Python python]# _
```

下载完成后进入 Python 编辑器，如果 import netdev 没有报错，则说明安装成功，如下所示。

```
[root@CentOS-Python python]# python3.8
Python 3.8.2 (default, Apr 27 2020, 23:06:10)
[GCC 8.3.1 20190507 (Red Hat 8.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import netdev
>>> _
```

在使用 netdev 进行异步操作之前，先来做一个同步和异步的对比试验。首先用传统的同步方式，通过 Netmiko 对 5 台交换机（192.168.2.11～192.168.2.15）的“line vty 5 15”配置“login local”，并统计从脚本开始运行到脚本执行完成所耗费的时间，然后使用 netdev，通过异步方式对同样的 5 台交换机做同样的配置，并计时，最后将两次的耗时相比较，看看孰优孰劣。

传统的同步方式的脚本如下（交换机 192.168.2.11～192.168.2.15 的 IP 地址被保存在一个名叫 ip_list.txt 的文件里）。

```
from Netmiko import ConnectHandler
import time

f = open('ip_list.txt')
start_time = time.time()

for ips in f.readlines():
    ip = ips.strip()
     SW = {
         'device_type': 'cisco_ios',
         'ip': ip,
         'username': 'python',
         'password': '123',
     }
     connect = ConnectHandler(**SW)
     print ("Sucessfully connected to " + SW['ip'])
     config_commands = ['line vty 5 15','login local','exit']
     output = connect.send_config_set(config_commands)
     print (output)

print ('Time elapsed: %.2f'%(time.time()-start_time))
```

执行脚本看效果，如下所示。

```
[root@CentOS-Python python]# python3.8 sync_netmiko.py
程序于 07:29:55 开始执行

Sucessfully connected to 192.168.2.11
config term
Enter configuration commands, one per line.  End with CNTL/Z.
pyntc_SW1(config)#line vty 5 15
pyntc_SW1(config-line)#login local
pyntc_SW1(config-line)#exit
pyntc_SW1(config)#end
pyntc_SW1#
Sucessfully connected to 192.168.2.12
config term
Enter configuration commands, one per line.  End with CNTL/Z.
SW2(config)#line vty 5 15
SW2(config-line)#login local
SW2(config-line)#exit
SW2(config)#end
SW2#
Sucessfully connected to 192.168.2.13
config term
Enter configuration commands, one per line.  End with CNTL/Z.
SW3(config)#line vty 5 15
SW3(config-line)#login local
SW3(config-line)#exit
SW3(config)#end
SW3#
Sucessfully connected to 192.168.2.14
config term
Enter configuration commands, one per line.  End with CNTL/Z.
SW4(config)#line vty 5 15
SW4(config-line)#login local
SW4(config-line)#exit
SW4(config)#end
SW4#
Sucessfully connected to 192.168.2.15
config term
Enter configuration commands, one per line.  End with CNTL/Z.
SW5(config)#line vty 5 15
SW5(config-line)#login local
SW5(config-line)#exit
SW5(config)#end
SW5#

程序于 07:30:40 执行结束
[root@CentOS-Python python]# _
```

可以看到，通过同步方式一台一台地登录 5 台交换机完成配置总共耗时 45s（07:29:55 – 07:30:40）。

再来看 netdev 的表现，代码如下。

```
import asyncio
import netdev
import time
```

```
async def task(dev):
    async with netdev.create(**dev) as ios:
        commands = ["line vty 5 15", "login local","exit"]
        out = await ios.send_config_set(commands)
        print(out)

async def run():
    devices = []
    f = open('ip_list.txt')
    for ips in f.readlines():
        ip = ips.strip()
        dev = {  'username' : 'python',
                 'password' : '123',
                 'device_type': 'cisco_ios',
                 'host': ip
              }
        devices.append(dev)
    tasks = [task(dev) for dev in devices]
    await asyncio.wait(tasks)

start_time = time.time()
print (f"程序于 {time.strftime('%X')} 开始执行\n")
asyncio.run(run())
print (f"\n程序于 {time.strftime('%X')} 执行结束")
```

执行代码看效果，如下所示。

```
[root@CentOS-Python python]# python3.8 netdev2.py
程序于 07:24:55 开始执行

conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW4(config)#line vty 5 15
SW4(config-line)#login local
SW4(config-line)#exit
SW4(config)#end
SW4#
conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW3(config)#line vty 5 15
SW3(config-line)#login local
```

```
SW3(config-line)#exit
SW3(config)#end
SW3#
conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW2(config)#line vty 5 15
SW2(config-line)#login local
SW2(config-line)#exit
SW2(config)#end
SW2#
conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW5(config)#line vty 5 15
SW5(config-line)#login local
SW5(config-line)#exit
SW5(config)#end
SW5#
conf t
Enter configuration commands, one per line.  End with CNTL/Z.
pyntc_SW1(config)#line vty 5 15
pyntc_SW1(config-line)#login local
pyntc_SW1(config-line)#exit
pyntc_SW1(config)#end
pyntc_SW1#

程序于 07:25:00 执行结束
[root@CentOS-Python python]# _
```

可以看到，通过 netdev 提供的异步方式，我们**仅耗时 5s（07:24:55 – 07:25:00）便执行完脚本**，对 5 台交换机完成了同样的配置，同步和异步的工作效率差距之大可见一斑！

另外，除了 netdev，pexpect 也支持异步并行，有兴趣的读者可以自行参阅其他资料了解。

6.7　Netmiko（多线程）

除了上一节讲到的单线程异步并行，我们也可以使用多线程（Multithreading）来提升 Python 脚本的工作效率。第 4 章已经介绍了 Netmiko 的来历及安装和使用方法，本节主要介绍如何使用 Netmiko 配合 Python 的内置模块 threading 来实现多线程执行 Python 脚本。

6.7.1 单线程与多线程

在 6.6.1 小节，我们引用了数学家华罗庚的《统筹方法》中所举的例子来说明单线程中同步和异步的区别。其实我们也可以引用同样的例子来说明单线程和多线程的区别。在《统筹方法》中讲到的沏茶的这个例子中，如果只有一个人来完成烧水、洗茶杯、倒茶叶 3 项任务，此时只有一个劳动力，我们就可以把它看成是单线程的。**假设我们能找来 3 个人分别负责烧水、洗茶杯、倒茶叶，并且保证 3 个人同时开工干活，那么我们就可以把它看成是多线程的，每一个劳动力代表一个线程。**

在计算机的世界中也是一样的，一个程序可以启用多个线程同时完成多个任务，如果一个线程阻塞，其他线程并不受影响。现在的 CPU 都是多核的，单线程只能用到其中的一核，这其实是对硬件资源的一种浪费（当然不可否认的是，随着时代的进步，现在的 CPU 已经足够强大，即使只用单核也能同时应付多个任务，这也是后来 Python 支持异步的原因）。如果使用多线程来运行 Python 脚本，则不仅能极大地提高脚本的运行速度，增加工作效率，并且还能充分利用主机的硬件资源。下面来看如何在 Python 中使用多线程。

6.7.2 多线程在 Python 中的应用

Python 3 已经内置了_thread 和 threading 两个模块来实现多线程。相较于_thread，threading 提供的方法更多而且更常用，因此我们将举例讲解 threading 模块的用法。首先来看下面这段代码。

```
import threading
import time

def say_after(what, delay):
    print (what)
```

```
    time.sleep(delay)
    print (what)

t = threading.Thread(target = say_after, args = ('hello',3))
t.start()
```

- 我们导入 Python 内置模块 threading 来实现多线程。之后定义一个 say_after(what, delay)函数，该函数包含 what 和 delay 两个参数，分别用来表示打印的内容及 time.sleep()休眠的时间。
- 随后使用 threading 的 Thread()函数为 say_after(what, delay)函数创建一个线程并将它赋值给变量 t，注意 Thread()里的 target 参数对应的是函数名称（即 say_after），args 对应的是该 say_after 函数的参数，等同于 what = ‘hello’和 delay = 3。
- 最后调用 threading 中的 start()来启动刚刚创建的线程。

运行代码看效果，如下所示。

```
[root@CentOS-Python python]# python3.8 threading1.py
hello
hello
[root@CentOS-Python python]# _
```

在打印出第一个 hello 后，程序因为 time.sleep(3)休眠了 3s，3s 之后随即打印出了第二个 hello。因为这时我们只运行了 say_after(what, delay)这一个函数，并且只运行了一次，因此即使我们现在启用了多线程，也感受不到它和单线程的区别。接下来将代码修改如下。

```
#coding=utf-8
import threading
import time

def say_after(what, delay):
    print (what)
    time.sleep(delay)
    print (what)

t = threading.Thread(target = say_after, args = ('hello',3))
```

```
print (f"程序于 {time.strftime('%X')} 开始执行")
t.start()
print (f"程序于 {time.strftime('%X')} 执行结束")
```

- 这一次调用 time.strftime()来尝试记录程序执行前和执行后的时间，看看有什么“意想不到”的结果。

运行代码看效果，如下所示。

```
[root@CentOS-Python python]# python3.8 threading2.py
程序于 02:23:44 开始执行
hello
程序于 02:23:44 执行结束
hello
[root@CentOS-Python python]# _
```

这里你肯定会问为什么程序在 02:23:44 开始执行，又在同一时间结束？难道不是该休眠 3s 吗？为什么明明 print (f"程序于 {time.strftime('%X')} 开始执行")和 print (f"程序于 {time.strftime('%X')} 执行结束")分别写在 t.start()的上面和下面，但是不等第二个 hello 被打印出来，print (f"程序于 {time.strftime('%X')} 执行结束")就被执行了？

这是因为除了 threading.Thread()为 say_after()函数创建的**用户线程**，print(f"程序于 {time.strftime('%X')}开始执行")和 print (f"程序于 {time.strftime('%X')} 执行结束")两个 print()函数也共同占用了公用的**内核线程**。也就是说，该脚本现在实际上调用了两个线程：一个是用户线程，一个是内核线程，也就构成了一个多线程的环境。**因为分属不同的线程，say_after()函数和函数之外的两个 print 语句是同时运行的，互不干涉，**因此 print (f"程序于 {time.strftime('%X')}执行结束")不会像在单线程中那样等到 t.start()执行完才被执行，而是在 print (f"程序于 {time.strftime('%X')} 开始执行")被执行后就马上跟着被执行。这也就解释了为什么你会看到原本需要休眠 3s 的脚本会在 02:23:44 同时开始和结束。

如果想要正确捕捉 say_after(what, delay)函数开始和结束的时间，需要额外使用 threading 模块的 join()方法，来看下面的代码。

```
#coding=utf-8
import threading
```

```
import time

def say_after(what, delay):
    print (what)
    time.sleep(delay)
    print (what)

t = threading.Thread(target = say_after, args = ('hello',3))

print (f"程序于 {time.strftime('%X')} 开始执行")
t.start()
t.join()
print (f"程序于 {time.strftime('%X')} 执行结束")
```

- 我们只修改了代码的一处地方，即在 t.start()下面添加了一个 t.join()。**join()方法的作用是强制阻塞调用它的线程，直到该线程运行完毕或者终止（类似单线程同步）。**因为调用 join()方法的变量 t 正是用 threading.Thread()为 say_after(what, delay)函数创建的用户线程，因此使用内核线程的 print (f"程序于 {time.strftime('%X')}执行结束")必须等待该用户线程执行完毕后才能继续执行，因此脚本在执行时的效果会让你觉得整体还是以单线程同步的方式运行的。

运行代码看效果，如下所示。

```
[root@CentOS-Python python]# python3.8 threading3.py
程序于 03:49:19 开始执行
hello
hello
程序于 03:49:22 执行结束
```

可以看到，因为调用了.join()方法，在内核线程上运行的 print (f"程序于 {time.strftime('%X')}执行结束")必须等待在用户线程上运行的 say_after(what, delay)函数执行完毕后，才能继续被执行，因此程序前后执行总共花费 3s，类似单线程同步的效果。

最后再举一例，来看下如何创建多个用户线程并运行，代码如下。

```
#coding=utf-8
import threading
```

```
import time

def say_after(what, delay):
    print (what)
    time.sleep(delay)
    print (what)

print (f"程序于 {time.strftime('%X')} 开始执行\n")
threads = []

for i in range(1,6):
    t = threading.Thread(target=say_after, name="线程" + str(i), args=('hello',3))
    print(t.name + '开始执行。')
    t.start()
    threads.append(t)

for i in threads:
    i.join()
print (f"\n 程序于 {time.strftime('%X')} 执行结束")
```

- 我们使用 for 循环配合 range(1,6)创建了 5 个线程，并且将它们以多线程的形式执行，也就是把 say_after(what, delay)函数以多线程的形式执行了 5 次。每个线程作为元素加入 threads 空列表，然后使用 for 语句遍历现在已经有 5 个线程的 threads 列表，对其中的每个线程都调用 join()方法，确保直到它们都执行结束，才执行内核线程的 print (f"程序于 {time.strftime('%X')} 执行结束")。

运行代码看效果，如下所示。

```
[root@CentOS-Python python]# python3.8 threading4.py
程序于 05:13:35 开始执行

线程1开始执行。
hello
线程2开始执行。
hello
线程3开始执行。
hello
线程4开始执行。
hello
线程5开始执行。
hello
hello
```

```
hello
hello
hello
hello

程序于 05:13:38 执行结束
[root@CentOS-Python python]# _
```

可以看到，我们成功地使用多线程将程序执行，如果以单线程来执行 5 次 say_after(what,delay)函数，则需要花费 3×5=15s 才能跑完整个脚本，而在多线程形式下，整个程序只花费 3s 就运行完毕。

6.7.3　多线程在 Netmiko 中的应用

在掌握了 threading 模块的基本用法后，我们来看如何将它与 Netmiko 结合，实现通过 Netmiko 对网络做多线程登录和操作。

在 6.6.3 节中，我们使用传统的单线程同步方式，通过 Netmiko 对 5 台交换机（192.168.2.11～192.168.2.15）的 line vty 5 15 配置了 login local，整个脚本从开始执行到结束，前后总共耗时 45.02s。这里我们用 Netmiko 以多线程的方式，对这 5 台交换机做同样的配置并计时，来看下使用多线程能节省多少时间。

脚本代码如下。

```
    #coding=utf-8
    import threading
    from queue import Queue
    import time
    from Netmiko import ConnectHandler

    f = open('ip_list.txt')
    threads = []

    def ssh_session(ip, output_q):
        commands = ["line vty 5 15", "login local","exit"]
        SW = {'device_type': 'cisco_ios', 'ip': ip, 'username': 'python', 'password':
'123'}
        ssh_session = ConnectHandler(**SW)
        output = ssh_session.send_config_set(commands)
        print (output)

    print (f"程序于 {time.strftime('%X')} 开始执行\n")
    for ips in f.readlines():
         t = threading.Thread(target=ssh_session, args=(ips.strip(), Queue()))
         t.start()
         threads.append(t)

    for i in threads:
        i.join()
    print (f"\n 程序于 {time.strftime('%X')} 执行结束")
```

- 在使用 Netmiko 实现多线程时，需要导入内置队列模块 queue。在 Python 中，队列（queue）是线程间最常用的交换数据的形式，这里引用 queue 模块中的 Queue 类，也就是先进先出（First Input First Output，FIFO）队列，并将它作为出队列参数配置给 ssh_session(ip, output_q)函数。有关 queue 模块的具体介绍不在本书讨论范围内，只需要知道这是使用 Netmiko 实现多线程时必备的步骤即可。
- 其余代码讲解从略。

运行脚本看效果，如下所示。

```
[root@CentOS-Python python]# python3.8 threading6.py
程序于 08:11:41 开始执行

config term
Enter configuration commands, one per line.  End with CNTL/Z.
SW5(config)#line vty 5 15
SW5(config-line)#login local
SW5(config-line)#exit
SW5(config)#end
SW5#
config term
Enter configuration commands, one per line.  End with CNTL/Z.
SW4(config)#line vty 5 15
SW4(config-line)#login local
SW4(config-line)#exit
SW4(config)#end
```

```
SW4#
config term
Enter configuration commands, one per line.  End with CNTL/Z.
SW3(config)#line vty 5 15
SW3(config-line)#login local
SW3(config-line)#exit
SW3(config)#end
SW3#
config term
Enter configuration commands, one per line.  End with CNTL/Z.
pyntc_SW1(config)#line vty 5 15
pyntc_SW1(config-line)#login local
pyntc_SW1(config-line)#exit
pyntc_SW1(config)#end
pyntc_SW1#
config term
Enter configuration commands, one per line.  End with CNTL/Z.
SW2(config)#line vty 5 15
SW2(config-line)#login local
SW2(config-line)#exit
SW2(config)#end
SW2#

程序于 08:11:51 执行结束
[root@CentOS-Python python]# _
```

可以看到，使用 Netmiko 多线程总共仅耗时 10s（08:11:41 – 08:11:51）便完成了对 5 台交换机的配置。比默认使用单线程同步的传统方式快了 35s，不过这个速度依然慢于 6.6 节中提到的单线程异步方式。在该节实验中，我们使用 netdev 单线程总共仅耗时 5s 便对同样的 5 台交换机完成了同样的配置。