PYTHON ДЛЯ СЕТЕВЫХ ИНЖЕНЕРОВ

ПАРАЛЛЕЛЬНЫЕ СЕССИИ

ПАРАЛЛЕЛЬНЫЕ СЕССИИ

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

Для параллельного подключения к устройствам в курсе используются модули:

- threading
- multiprocessing
- concurrent.futures

ИЗМЕРЕНИЕ ВРЕМЕНИ ВЫПОЛНЕНИЯ СКРИПТА

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

- утилиту Linux time
- и модуль Python datetime

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

TIME

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

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

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

DATETIME

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

Пример использования:

```
from datetime import datetime
import time

start_time = datetime.now()

#Тут выполняются действия
time.sleep(5)

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

DATETIME

Результат выполнения:

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

ПРОЦЕССЫ И ПОТОКИ В PYTHON (CPYTHON)

- процесс (process) - это, грубо говоря, запущенная программа. Процессу выделяются отдельные ресурсы: память, процессорное время
- поток (thread) - это единица исполнения в процессе. Потоки разделяют ресурсы процесса, к которому они относятся.

ПРОЦЕССЫ И ПОТОКИ В PYTHON (CPYTHON)

Python (а точнее, CPython - реализация, которая используется в курсе) оптимизирован для работы в однопоточном режиме. Это хорошо, если в программе используется только один поток.

И, в то же время, у Python есть определенные нюансы работы в многопоточном режиме. Связаны они с тем, что CPython использует GIL (global interpreter lock).

GIL

GIL не дает нескольким потокам исполнять одновременно код Python. GIL можно представить как некий переходящий флаг, который разрешает потокам выполняться. У кого флаг, тот может выполнять работу.

Флаг передается либо каждые сколько-то инструкций Python, либо, например, когда выполняются какие-то операции ввода-вывода.

Поэтому получается, что разные потоки не будут выполняться параллельно, а программа просто будет между ними переключаться, выполняя их в разное время.

IO BOUND TASK

Но не всё так плохо. Если в программе есть некое "ожидание": пакетов из сети, запроса пользователя, пауза типа sleep - то в такой программе потоки будут выполняться как будто параллельно. А всё потому, что во время таких пауз флаг (GIL) можно передать другому потоку.

Но тут также нужно быть осторожным, так как такой результат может наблюдаться на небольшом количестве сессий, но может ухудшиться с ростом количества сессий.

Потоки отлично подходят для задач, которые связаны с операциями ввода-вывода. Подключение к оборудованию входит в число подобных задач.

ПРОЦЕССЫ

Процессы позволяют выполнять задачи на разных ядрах компьютера. Это важно для задач, которые не завязаны на операции ввода-вывода.

Для каждого процесса создается своя копия ресурсов, выделяется память, у каждого процесса свой GIL. Это же делает процессы более тяжеловесными, по сравнению с потоками.

Кроме того, количество процессов, которые запускаются параллельно, зависит от количества ядер и CPU и обычно исчисляется в десятках, тогда как количество потоков для операций ввода-вывода может исчисляться в сотнях.

МОДУЛЬ THREADING

МОДУЛЬ THREADING

Модуль threading может быть полезен для таких задач:

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

МОДУЛЬ THREADING

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

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

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

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

МОДУЛЬ THREADING

Файл netmiko_function.py:

```
from netmiko import ConnectHandler
import sys
import yaml

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

def connect_ssh(device_dict, command):

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

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

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

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

МОДУЛЬ THREADING

Файл devices.yaml с параметрами подключения к устройствам:

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

МОДУЛЬ THREADING

Время выполнения скрипта (вывод скрипта удален):

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

МОДУЛЬ THREADING

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

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

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

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

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

МОДУЛЬ THREADING

Файл netmiko_threading.py:

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

    for th in threads:
        th.join()

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

МОДУЛЬ THREADING

Время выполнения кода:

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

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

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

МОДУЛЬ THREADING

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

ПОЛУЧЕНИЕ ДАННЫХ ИЗ ПОТОКОВ

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

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

ПОЛУЧЕНИЕ ДАННЫХ ИЗ ПОТОКОВ

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

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

ПОЛУЧЕНИЕ ДАННЫХ ИЗ ПОТОКОВ

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

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

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

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

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

ПОЛУЧЕНИЕ ДАННЫХ ИЗ ПОТОКОВ

Файл netmiko_threading_data.py:

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

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

    for th in threads:
        th.join()

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

    return results

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

ПОЛУЧЕНИЕ ДАННЫХ ИЗ ПОТОКОВ

Обратите внимание, что в функции connect_ssh добавился аргумент queue.

Очередь вполне можно воспринимать как список:

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

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

ПОЛУЧЕНИЕ ДАННЫХ ИЗ ПОТОКОВ

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

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

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

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

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

ПОЛУЧЕНИЕ ДАННЫХ ИЗ ПОТОКОВ

Файл netmiko_threading_data_list.py:

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

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

    for th in threads:
        th.join()

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

    return q

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

МОДУЛЬ MULTIPROCESSING

МОДУЛЬ MULTIPROCESSING

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

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

МОДУЛЬ MULTIPROCESSING

Пример использования модуля multiprocessing (файл netmiko_multiprocessing.py):

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


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

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

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

МОДУЛЬ MULTIPROCESSING

Файл netmiko_multiprocessing.py:

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

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

    for p in processes:
        p.join()

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

    return results

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

МОДУЛЬ MULTIPROCESSING

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

МОДУЛЬ MULTIPROCESSING

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

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

МОДУЛЬ MULTIPROCESSING

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

МОДУЛЬ CONCURRENT.FUTURES

Модуль concurrent.futures предоставляет высокоуровневый интерфейс для работы с процессами и потоками. При этом и для потоков, и для процессов используется одинаковый интерфейс, что позволяет легко переключаться между ними.

Если сравнивать этот модуль с threading или multiprocessing, то у него меньше возможностей. Но зато с concurrent.futures работать проще и интерфейс более понятный.

МОДУЛЬ CONCURRENT.FUTURES

Модуль concurrent.futures позволяет легко решить задачу запуска нескольких потоков/процессов и получения из них данных.

Модуль предоставляет два класса:

- **ThreadPoolExecutor** - для работы с потоками
- **ProcessPoolExecutor** - для работы с процессами

Оба класса используют одинаковый интерфейс, поэтому достаточно разобраться с одним и затем просто переключиться на другой при необходимости.

FUTURE

Модуль использует понятие future. Future - это объект, который представляет отложенное вычисление. Этот объект можно запрашивать о состоянии (завершена работа или нет), можно получать результаты или исключения, которые возникли в процессе работы, по мере возникновения.

При этом нет необходимости создавать их вручную. Эти объекты создаются ThreadPoolExecutor и ProcessPoolExecutor.

МЕТОД MAP

Метод map - это самый простой вариант работы с concurrent.futures.

Пример использования функции map с ThreadPoolExecutor (файл netmiko_threads_map_ver1.py):

```
from concurrent.futures import ThreadPoolExecutor
from pprint import pprint

import yaml
from netmiko import ConnectHandler


def connect_ssh(device_dict, command='sh clock'):
    print('Connection to device: {}'.format(device_dict['ip']))
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
    return {device_dict['ip']: result}


def threads_conn(function, devices, limit=2):
    with ThreadPoolExecutor(max_workers=limit) as executor:
        f_result = executor.map(function, devices)
    return list(f_result)
```

МЕТОД MAP

```
def threads_conn(function, devices, limit=2):
    with ThreadPoolExecutor(max_workers=limit) as executor:
        f_result = executor.map(function, devices)
    return list(f_result)
```

Обратите внимание, что функция занимает всего 4 строки, и для получения данных не надо создавать очередь и передавать ее в функцию connect_ssh.

МЕТОД MAP

- with ThreadPoolExecutor(max_workers=limit) as executor: - класс ThreadPoolExecutor инициируется в блоке with с указанием количества потоков
- f_result = executor.map(function, devices) - метод map похож на функцию map, но тут функция function вызывается в разных потоках. При этом в разных потоках функция будет вызываться с разными аргументами - элементами итерируемого объекта devices.
- метод map возвращает генератор. В этом генераторе содержатся результаты выполнения функций

МЕТОД MAP

Результат выполнения:

```
$ python netmiko_threads_map_ver1.py
Connection to device: 192.168.100.1
Connection to device: 192.168.100.2
Connection to device: 192.168.100.3
[{'192.168.100.1': '*04:43:01.629 UTC Mon Aug 28 2017'},
 {'192.168.100.2': '*04:43:01.648 UTC Mon Aug 28 2017'},
 {'192.168.100.3': '*04:43:07.291 UTC Mon Aug 28 2017'}]
```

МЕТОД MAP

Важная особенность метода map - он возвращает результаты в том же порядке, в котором они указаны в итерируемом объекте.

Для демонстрации этой особенности в функции connect_ssh добавлены сообщения с выводом информации о том, когда функция начала работать и когда закончила.

МЕТОД MAP

Файл netmiko_threads_map_ver2.py:

```
from concurrent.futures import ThreadPoolExecutor
from pprint import pprint
from datetime import datetime
import time

import yaml
from netmiko import ConnectHandler


start_msg = '===> {} Connection to device: {}'
received_msg = '<=== {} Received result from device: {}'
```

МЕТОД MAP

```
def connect_ssh(device_dict, command='sh clock'):
    print(start_msg.format(datetime.now().time(), device_dict['ip']))
    if device_dict['ip'] == '192.168.100.1':
        time.sleep(10)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        print(received_msg.format(datetime.now().time(), device_dict['ip']))
    return {device_dict['ip']: result}


def threads_conn(function, devices, limit=2):
    with ThreadPoolExecutor(max_workers=limit) as executor:
        f_result = executor.map(function, devices)
    return list(f_result)


if __name__ == '__main__':
    devices = yaml.load(open('devices.yaml'))
    all_done = threads_conn(connect_ssh, devices['routers'])
    pprint(all_done)
```

МЕТОД MAP

Результат выполнения:

```
$ python netmiko_threads_map_ver2.py
===> 04:50:50.175076 Connection to device: 192.168.100.1
===> 04:50:50.175553 Connection to device: 192.168.100.2
<=== 04:50:55.582707 Received result from device: 192.168.100.2
===> 04:50:55.689248 Connection to device: 192.168.100.3
<=== 04:51:01.135640 Received result from device: 192.168.100.3
<=== 04:51:05.568037 Received result from device: 192.168.100.1
[{'192.168.100.1': '*04:51:05.395 UTC Mon Aug 28 2017'},
 {'192.168.100.2': '*04:50:55.411 UTC Mon Aug 28 2017'},
 {'192.168.100.3': '*04:51:00.964 UTC Mon Aug 28 2017'}]
```

МЕТОД MAP

Обратите внимание на фактический порядок выполнения задач: 192.168.100.2, 192.168.100.3, 192.168.100.1. Но в итоговом списке все равно соблюдается порядок на основе списка devices['routers'].

МЕТОД MAP

Еще один момент, который тут хорошо заметен, это то, что как только одна задача выполнилась, сразу берется следующая. То есть, ограничение в два потока влияет на количество потоков, которые выполняются одновременно.

МЕТОД MAP

Осталось изменить функцию таким образом, чтобы ей можно было передавать команду как аргумент.

Для этого мы воспользуемся функцией repeat из модуля itertools. Функция repeat тут нужна для того, чтобы команда передавалась при каждом вызове функции connect_ssh.

МЕТОД MAP

Файл netmiko_threads_map_final.py

```
from concurrent.futures import ThreadPoolExecutor
from pprint import pprint
from datetime import datetime
import time
from itertools import repeat

import yaml
from netmiko import ConnectHandler


start_msg = '===> {} Connection to device: {}'
received_msg = '<=== {} Received result from device: {}'
```

МЕТОД MAP

Файл netmiko_threads_map_final.py

```
def connect_ssh(device_dict, command):
    print(start_msg.format(datetime.now().time(), device_dict['ip']))
    if device_dict['ip'] == '192.168.100.1':
        time.sleep(10)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        print(received_msg.format(datetime.now().time(), device_dict['ip']))
    return {device_dict['ip']: result}



def threads_conn(function, devices, limit=2, command=''):
    with ThreadPoolExecutor(max_workers=limit) as executor:
        f_result = executor.map(function, devices, repeat(command))
    return list(f_result)


if __name__ == '__main__':
    devices = yaml.load(open('devices.yaml'))
    all_done = threads_conn(connect_ssh,
                            devices['routers'],
                            command='sh clock')
    pprint(all_done)
```

МЕТОД MAP

Результат выполнения:

```
$ python netmiko_threads_map_final.py
===> 05:01:08.314962 Connection to device: 192.168.100.1
===> 05:01:08.315114 Connection to device: 192.168.100.2
<=== 05:01:13.693083 Received result from device: 192.168.100.2
===> 05:01:13.799002 Connection to device: 192.168.100.3
<=== 05:01:19.363250 Received result from device: 192.168.100.3
<=== 05:01:23.685859 Received result from device: 192.168.100.1
[{'192.168.100.1': '*05:01:23.513 UTC Mon Aug 28 2017'},
 {'192.168.100.2': '*05:01:13.522 UTC Mon Aug 28 2017'},
 {'192.168.100.3': '*05:01:19.189 UTC Mon Aug 28 2017'}]
```

ИСПОЛЬЗОВАНИЕ PROCESSPOOLEXECUTOR С MAP

Для того чтобы предыдущий пример использовал процессы вместо потоков, достаточно сменить ThreadPoolExecutor на ProcessPoolExecutor:

```
from concurrent.futures import ProcessPoolExecutor
from pprint import pprint
from datetime import datetime
import time
from itertools import repeat

import yaml
from netmiko import ConnectHandler


start_msg = '===> {} Connection to device: {}'
received_msg = '<=== {} Received result from device: {}'
```

ИСПОЛЬЗОВАНИЕ PROCESSPOOLEXECUTOR С MAP

```
def connect_ssh(device_dict, command):
    print(start_msg.format(datetime.now().time(), device_dict['ip']))
    if device_dict['ip'] == '192.168.100.1':
        time.sleep(10)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        print(received_msg.format(datetime.now().time(), device_dict['ip']))
    return {device_dict['ip']: result}



def threads_conn(function, devices, limit=2, command=''):
    with ProcessPoolExecutor(max_workers=limit) as executor:
        f_result = executor.map(function, devices, repeat(command))
    return list(f_result)


if __name__ == '__main__':
    devices = yaml.load(open('devices.yaml'))
    all_done = threads_conn(connect_ssh,
                            devices['routers'],
                            command='sh clock')
    pprint(all_done)
```

ИСПОЛЬЗОВАНИЕ PROCESSPOOLEXECUTOR С MAP

Результат выполнения:

```
$ python netmiko_processes_map_final.py
===> 05:26:42.974505 Connection to device: 192.168.100.1
===> 05:26:42.975733 Connection to device: 192.168.100.2
<=== 05:26:48.389420 Received result from device: 192.168.100.2
===> 05:26:48.495598 Connection to device: 192.168.100.3
<=== 05:26:54.104585 Received result from device: 192.168.100.3
<=== 05:26:58.367981 Received result from device: 192.168.100.1
[{'192.168.100.1': '*05:26:58.195 UTC Mon Aug 28 2017'},
 {'192.168.100.2': '*05:26:48.218 UTC Mon Aug 28 2017'},
 {'192.168.100.3': '*05:26:53.932 UTC Mon Aug 28 2017'}]
```

МЕТОД SUBMIT И РАБОТА С FUTURES

МЕТОД SUBMIT И РАБОТА С FUTURES

При использовании метода map объект future использовался внутри, но в итоге мы получали уже готовый результат функции.

Метод submit позволяет запускать future, а функция as_completed, которая ожидает как аргумент итерируемый объект с futures и возвращает future по мере завершения. В этом случае порядок не будет соблюдаться, как с map.

МЕТОД SUBMIT И РАБОТА С FUTURES

Файл netmiko_threads_submit.py:

```
from concurrent.futures import ThreadPoolExecutor, as_completed
from pprint import pprint
from datetime import datetime
import time
from itertools import repeat

import yaml
from netmiko import ConnectHandler


start_msg = '===> {} Connection to device: {}'
received_msg = '<=== {} Received result from device: {}'
```

МЕТОД SUBMIT И РАБОТА С FUTURES

```
def connect_ssh(device_dict, command):
    print(start_msg.format(datetime.now().time(), device_dict['ip']))
    if device_dict['ip'] == '192.168.100.1':
        time.sleep(10)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        print(received_msg.format(datetime.now().time(), device_dict['ip']))
    return {device_dict['ip']: result}
```

МЕТОД SUBMIT И РАБОТА С FUTURES

Теперь функция threads_conn выглядит немного по-другому:

```
def threads_conn(function, devices, limit=2, command=''):
    all_results = []
    with ThreadPoolExecutor(max_workers=limit) as executor:
        future_ssh = [executor.submit(function, device, command)
                      for device in devices]
        for f in as_completed(future_ssh):
            all_results.append(f.result())
    return all_results


if __name__ == '__main__':
    devices = yaml.load(open('devices.yaml'))
    all_done = threads_conn(connect_ssh,
                            devices['routers'],
                            command='sh clock')
    pprint(all_done)
```

МЕТОД SUBMIT И РАБОТА С FUTURES

Остальной код не изменился, поэтому разобраться надо только с функцией threads_conn:

```
def threads_conn(function, devices, limit=2, command=''):
    all_results = []
    with ThreadPoolExecutor(max_workers=limit) as executor:
        future_ssh = [executor.submit(function, device, command)
                      for device in devices]
        for f in as_completed(future_ssh):
            all_results.append(f.result())
    return all_results
```

МЕТОД SUBMIT И РАБОТА С FUTURES

В блоке with два цикла:

- future_ssh - это список объектов future, который создается с помощью list comprehensions
- для создания future используется функция submit
  - ей как аргументы передаются: имя функции, которую надо выполнить, и ее аргументы
- следующий цикл проходится по списку future с помощью функции as_completed. Эта функция возвращает future только когда они завершили работу или были отменены. При этом future возвращаются по мере завершения работы

МЕТОД SUBMIT И РАБОТА С FUTURES

Результат выполнения:

```
$ python netmiko_threads_submit.py
===> 06:02:14.582011 Connection to device: 192.168.100.1
===> 06:02:14.582155 Connection to device: 192.168.100.2
<=== 06:02:20.155865 Received result from device: 192.168.100.2
===> 06:02:20.262584 Connection to device: 192.168.100.3
<=== 06:02:25.864270 Received result from device: 192.168.100.3
<=== 06:02:29.962225 Received result from device: 192.168.100.1
[{'192.168.100.2': '*06:02:19.983 UTC Mon Aug 28 2017'},
 {'192.168.100.3': '*06:02:25.691 UTC Mon Aug 28 2017'},
 {'192.168.100.1': '*06:02:29.789 UTC Mon Aug 28 2017'}]
```

Обратите внимание, что порядок не сохраняется и зависит от того, какие функции раньше завершили работу.

ОБРАБОТКА ИСКЛЮЧЕНИЙ

ОБРАБОТКА ИСКЛЮЧЕНИЙ

Если при выполнении функции возникло исключение, оно будет сгенерировано при получении результата

Например, в файле devices.yaml пароль для устройства 192.168.100.2 изменен на неправильный:

```
$ python netmiko_threads_submit.py
===> 06:29:40.871851 Connection to device: 192.168.100.1
===> 06:29:40.872888 Connection to device: 192.168.100.2
===> 06:29:43.571296 Connection to device: 192.168.100.3
<=== 06:29:48.921702 Received result from device: 192.168.100.3
<=== 06:29:56.269284 Received result from device: 192.168.100.1
Traceback (most recent call last):
  File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/netmiko/base_connection.py", line 491,
    self.remote_conn_pre.connect(**ssh_connect_params)
  File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/paramiko/client.py", line 394, in conne
    look_for_keys, gss_auth, gss_kex, gss_deleg_creds, gss_host)
  File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/paramiko/client.py", line 649, in _auth
    raise saved_exception
  File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/paramiko/client.py", line 636, in _auth
    self._transport.auth_password(username, password)
  File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/paramiko/transport.py", line 1329, in a
    return self.auth_handler.wait_for_response(my_event)
  File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/paramiko/auth_handler.py", line 217, in
    raise e
paramiko.ssh_exception.AuthenticationException: Authentication failed.
```

ОБРАБОТКА ИСКЛЮЧЕНИЙ

Так как исключение возникает при получении результата, легко добавить обработку исключений (файл netmiko_threads_submit_exception.py):

```
from concurrent.futures import ThreadPoolExecutor, as_completed
from pprint import pprint
from datetime import datetime
import time
from itertools import repeat

import yaml
from netmiko import ConnectHandler
from netmiko.ssh_exception import NetMikoAuthenticationException


start_msg = '===> {} Connection to device: {}'
received_msg = '<=== {} Received result from device: {}'


def connect_ssh(device_dict, command):
    print(start_msg.format(datetime.now().time(), device_dict['ip']))
    if device_dict['ip'] == '192.168.100.1':
        time.sleep(10)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        print(received_msg.format(datetime.now().time(), device_dict['ip']))
    return {device_dict['ip']: result}
```

ОБРАБОТКА ИСКЛЮЧЕНИЙ

```
def threads_conn(function, devices, limit=2, command=''):
    all_results = {}
    with ThreadPoolExecutor(max_workers=limit) as executor:
        future_ssh = [executor.submit(function, device, command)
                      for device in devices]
        for f in as_completed(future_ssh):
            try:
                result = f.result()
            except NetMikoAuthenticationException as e:
                print(e)
            else:
                all_results.update(result)
    return all_results


if __name__ == '__main__':
    devices = yaml.load(open('devices.yaml'))
    all_done = threads_conn(connect_ssh,
                            devices['routers'],
                            command='sh clock')
    pprint(all_done)
```

ОБРАБОТКА ИСКЛЮЧЕНИЙ

Результат выполнения:

```
$ python netmiko_threads_submit_exception.py
===> 06:45:56.327892 Connection to device: 192.168.100.1
===> 06:45:56.328190 Connection to device: 192.168.100.2
===> 06:45:58.964806 Connection to device: 192.168.100.3
Authentication failure: unable to connect cisco_ios 192.168.100.2:22
Authentication failed.
<=== 06:46:04.325812 Received result from device: 192.168.100.3
<=== 06:46:11.731541 Received result from device: 192.168.100.1
{'192.168.100.1': '*06:46:11.556 UTC Mon Aug 28 2017',
 '192.168.100.3': '*06:46:04.154 UTC Mon Aug 28 2017'}
```

PROCESSPOOLEXECUTOR

Так как все работает аналогичным образом и для процессов, тут приведет последний вариант (файл netmiko_processes_submit_exception.py):

```
from concurrent.futures import ProcessPoolExecutor, as_completed
from pprint import pprint
from datetime import datetime
import time
from itertools import repeat

import yaml
from netmiko import ConnectHandler
from netmiko.ssh_exception import NetMikoAuthenticationException


start_msg = '===> {} Connection to device: {}'
received_msg = '<=== {} Received result from device: {}'


def connect_ssh(device_dict, command):
    print(start_msg.format(datetime.now().time(), device_dict['ip']))
    if device_dict['ip'] == '192.168.100.1':
        time.sleep(10)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        print(received_msg.format(datetime.now().time(), device_dict['ip']))
    return {device_dict['ip']: result}
```

PROCESSPOOLEXECUTOR

```
def processes_conn(function, devices, limit=2, command=''):
    all_results = {}
    with ProcessPoolExecutor(max_workers=limit) as executor:
        future_ssh = [executor.submit(function, device, command)
                      for device in devices]
        for f in as_completed(future_ssh):
            try:
                result = f.result()
            except NetMikoAuthenticationException as e:
                print(e)
            else:
                all_results.update(result)
    return all_results


if __name__ == '__main__':
    devices = yaml.load(open('devices.yaml'))
    all_done = processes_conn(connect_ssh,
                              devices['routers'],
                              command='sh clock')
    pprint(all_done)
```

PROCESSPOOLEXECUTOR

Результат выполнения:

```
$ python netmiko_processes_submit_exception.py
===> 06:40:43.828249 Connection to device: 192.168.100.1
===> 06:40:43.828664 Connection to device: 192.168.100.2
Authentication failure: unable to connect cisco_ios 192.168.100.2:22
Authentication failed.
===> 06:40:46.292613 Connection to device: 192.168.100.3
<=== 06:40:51.890816 Received result from device: 192.168.100.3
<=== 06:40:59.231330 Received result from device: 192.168.100.1
{'192.168.100.1': '*06:40:59.056 UTC Mon Aug 28 2017',
 '192.168.100.3': '*06:40:51.719 UTC Mon Aug 28 2017'}
```