Основы asyncio для сетевых инженеров

Natasha Samoylenko

мая 30, 2023

Оглавление

В книге рассматриваются практическое применение asyncio для работы с сетевым оборудованием. Не рассматривается зачем и почему сетевому инженеру вдруг нужно разбираться с asyncio, но возможно будет в более поздних версиях книги. Пока что рекомендую посмотреть серию видео лекций про asyncio от Łukasz Langa (core developer Python, создатель black). Это ответит на многие вопросы зачем и почему и нужно ли это конкретно вам.

Примечание: Если вы делаете задания из курса «Advanced Python для сетевых инженеров», то соответствие разделов такое:

- 17 раздел заданий - 1, 2 разделы книги
- 18 раздел заданий - 3, 4 и 5 разделы книги

1

1. Основы работы с сопрограммами

Модуль asyncio можно разделить на две части: высокоуровневый интерфейс для пользователей, которые пишут программы и низкоуровневый интерфейс для авторов модулей, библиотек и фреймворков на основе asyncio. В книге рассматривается только первая часть и чаще всего, вторая не понадобится.

Примечание: На мой взгляд, asyncio это одна из тех тем, которые сложно понять по одному источнику. Поэтому я очень рекомендую читать/смотреть разные материалы, если вы действительно хотите в этом разобраться. Если нужно лишь общее понимание, может хватить и одного источника информации.

Сравнение асинхронного программирования и потоков

В Python есть три основных способа запускать задачи «параллельно»:

- процессы
- потоки
- асинхронное программирование

Примечание: Полноценно параллельная работа будет только при использовании процессов, по всех остальных случая речь идет о попеременном выполнении (concurrent).

Модули, которые позволяют запустить задачи:

- в процессах: multiprocessing, concurrent.futures
- в потоках: threading, concurrent.futures

- асинхронно: asyncio и масса других вариантов и сторонних фреймворков

Примечание: Для упрощения я буду говорить о запуске функций в потоках/процессах, но технически это может быть любой вызываемый объект. Аналогично дальше я в основном пишу asyncio, но почти все из сказанного про asyncio относится и к другим вариантам асинхронного программирования.

Запуск функций в процессах это единственный вариант из перечисленных выше, который позволит запустить код на разных ядрах. При этом, когда речь идет о количестве процессов это единицы-десятки. Если задача завязана на CPU это единственный способ как-то распараллелить задачу.

Потоки в CPython выполняются не параллельно, а попеременно из-за GIL. Поэтому преимущества при использовании потоков будут только для задач, которые завязаны на ввод-вывод. Например, в книге по основам Python мы рассматривали именно потоки, потому что подключение к оборудованию это задача завязанная на операции ввода-вывода. Количество параллельных потоков может быть десятки-сотни.

Асинхронное программирование это еще один вариант выполнения задач попеременно. В Python есть много способов реализации асинхронности, но в книге рассматривается только один - asyncio. Как подсказывает название, asyncio также предназначен для задач завязанных на операции ввода-вывода (IO). В этом подходе все выполняется в одном потоке, но при этом можно делать тысячи параллельных подключений.

Примечание: Для знакомства с другими вариантами реализации асинхронности очень рекомендую послушать Олега Молчанова.

Часто можно встретить фразу, что асинхронный код проще чем многопоточный. Тут имеется в виду, что в асинхронном коде очень четко видно те места, где будет переключение, а весь остальной код будет выполняться последовательно без прерывания. В то время как многопоточный код может быть прерван в любом месте и часто это приводит к проблемам.

При этом разобраться с тем как работает asyncio совсем не просто. К тому же, в большинстве случаев не получится использовать уже известные модули, например, netmiko, а вместо них надо будет использовать асинхронный аналог.

Самые большие преимущества при использовании asyncio можно получить, например, в веб. Тут подключений действительно может быть тысячи и с asynio не надо будет создавать поток для каждого подключения. Однако для работы с сетевым оборудованием asyncio тоже может быть полезен и во многих случаях, особенно когда подключений много, работать быстрее многопоточного варианта.

При этом конечно надо учитывать, что почти все модули, которые работали в многопоточном варианте, не будут работать с asyncio, так как они будут блокировать работу и не будут отдавать управление циклу событий.

Отличия asyncio от многопоточной работы:

- при работе с потоками, планировщик может прервать работу потока в любой момент, не всегда это «удобный» момент, поэтому надо явно указывать, что в какие-то моменты прерывать поток нельзя
- при работе с asyncio, мы явно указываем в каком месте надо сделать переключение (await)
- при использовании asyncio работа идет в одном потоке
- в потоках будет работать почти любой из распространенных модулей
- для работы с asyncio, как правило, надо будет искать альтернативные модули

Терминология

- цикл событий (event loop) - менеджер управляющий различными задачами и сопрограммами
- сопрограмма (coroutine) в asyncio - специальный тип функций, которые создаются со словом async перед определением.
- future - это объект, который представляет отложенное вычисление.
- задача (task) - отвечает за управление работой сопрограммы, запрашивает статус сопрограммы. Является подклассом Future.

Блокирующий/неблокирующий вызов

Примечание: Технически любой вызов функции блокирующий, но тут обсуждается терминология относящаяся к операциям ввода-вывода.

Блокирующий вызов останавливает работу программы, пока не будет получен результат вызова. Например, `subprocess.run` это блокирующий вызов и если вызвать, к примеру, ping какого-то адреса, то строка с subprocess.run будет блокировать выполнение скрипта пока не отработает ping.

Неблокирующий вызов возвращает какой-то объект сразу и, как правило, предполагается что позже надо будет еще раз вызывать какой-то метод этого объекта, чтобы получить результат вызова. Например, `subprocess.Popen`:

```
import subprocess

def ping_ip_addresses(ip_list):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    reachable = []
    unreachable = []
    result = []
    for ip in ip_list:
        p = subprocess.Popen(
            ["ping", "-c", "3", "-n", ip],
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
        )
        result.append(p)
    for ip, p in zip(ip_list, result):
        returncode = p.wait()
        if returncode == 0:
            reachable.append(ip)
        else:
            unreachable.append(ip)
    return reachable, unreachable
```

Тут вызов `subprocess.Popen` неблокирующий и с помощью этого вызова запущены несколько ping, но затем, чтобы получить результат, надо вызвать блокирующий `p.wait`.

Awaitables

Awaitables (ожидаемые объекты) - это объекты, которые можно использовать в выражении вместе с `await`. Три базовых типа awaitables:

- сопрограммы (coroutines)
- задачи (tasks)
- future

Coroutine (сопрограмма)

Как и с генераторами, различают:

- функцию сопрограмму - функция, которая создается с помощью `async def`
- объект сопрограмму - объект, который возвращается при вызове функции сопрограммы

Функции-сопрограммы возвращают объекты сопрограммы, которые запускаются менеджером (циклом событий). Сопрограмма может периодически прерывать выполнение и отдавать управление менеджеру, но при этом она не теряет состояние.

Task (задача)

Объекты класса Task используются для запуска сопрограмм в цикле событий и для отслеживания их состояния. Как только сопрограмма «обернута» в Task, например, с помощью функции `asyncio.create_task`, сопрограмма автоматически запущена для выполнения.

asyncio.Future

Future - это специальный низкоуровневый объект, который представляет отложенное вычисление асинхронных операций. Чаще всего, при работе с модулем asyncio, нет необходимости создавать Future напрямую, но некоторые функции могут возвращать Future. Task является подклассом Future. Менеджер может следить за future и ожидать их завершения.

Multitasking

- Preemptive multitasking - ОС решает, когда надо сделать переключение
- Cooperative multitasking - переключения указаны в коде

Сопрограммы и задачи

Создание сопрограммы (coroutine):

```
import asyncio

async def main():
    print(f'Start {datetime.now()}')
    await asyncio.sleep(3)
    print(f'End   {datetime.now()}')


In [6]: coro = main()

In [7]: coro
Out[7]: <coroutine object main at 0xb449fdac>
```

Как и с генераторами, различают:

- функцию сопрограмму - функция, которая создается с помощью `async def`
- объект сопрограмму - объект, который возвращается при вызове функции сопрограммы

Создать сопрограмму недостаточно для того чтобы она запускалась параллельно с другими сопрограммами - для управления сопрограммами нужен менеджер - event loop. Также по

умолчанию в сопрограмме код выполняется последовательно и надо явно указывать в каких местах можно переключаться - await.

Запустить сопрограмму можно несколькими способами:

- asyncio.run
- await
- asyncio.create_task

asyncio.run

Функция asyncio.run запускает сопрограмму и возвращает результат:

```
asyncio.run(coro, *, debug=False)
```

Функция asyncio.run всегда создает новый цикл событий и закрывает его в конце. В идеале, функция asyncio.run должна вызываться в программе только один раз и использоваться как основная точка входа. Эту функцию нельзя вызвать, когда в том же потоке запущен другой цикл событий.

Запуск с помощью asyncio.run:

```
In [8]: asyncio.run(coro)
Start 2019-10-30 06:36:03.396389
End   2019-10-30 06:36:06.399606


In [9]: asyncio.run(main())
Start 2019-10-30 06:46:22.162731
End   2019-10-30 06:46:25.166902
```

await

Второй вариант запуска сопрограммы - ожидание ее результата в другой сопрограмме с помощью `await`.

Сопрограмма delay_print выводит указанное сообщение с задержкой:

```
from datetime import datetime

async def delay_print(delay, task_name):
    print(f'>>> start {task_name}')
    await asyncio.sleep(delay)
    print(f'<<< end   {task_name}')
```

Для запуска сопрограммы delay_print, ее результат ожидается в сопрограмме main:

```
async def main():
    print(f'Start {datetime.now()}')
    await delay_print(4, 'task1')
    await delay_print(2, 'task2')
    print(f'End   {datetime.now()}')


In [5]: asyncio.run(main())
Start 2021-03-16 09:54:42.163949
>>> start task1
<<< end   task1
>>> start task2
<<< end   task2
End   2021-03-16 09:54:48.172434
```

Обратите внимание на время выполнения main - в данном случае сопрограммы выполнились последовательно и суммарное время 6 секунд.

asyncio.create_task

Еще один вариант запуска сопрограммы - это создание задачи (task). Обернуть сопрограмму в задачу и запланировать ее выполнение можно с помощью функции asyncio.create_task. Она возвращает объект Task, который можно ожидать с await, как и сопрограммы.

```
asyncio.create_task(coro, *, name=None)
```

Функция asyncio.create_task позволяет запускать сопрограммы одновременно, так как создание задачи означает для цикла, что надо запустить эту сопрограмму при первой возможности.

Пример создания задач:

```
async def delay_print(delay, task_name):
    print(f'>>> start {task_name}')
    await asyncio.sleep(delay)
    print(f'<<< end   {task_name}')


async def main():
    print(f'Start {datetime.now()}')
    task1 = asyncio.create_task(delay_print(4, 'task1'))
    task2 = asyncio.create_task(delay_print(2, 'task2'))

    await asyncio.sleep(1)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    print("Все задачи запущены")

    await task1
    await task2
    print(f'End   {datetime.now()}')


In [8]: asyncio.run(main())
Start 2021-03-16 09:58:10.817222
>>> start task1
>>> start task2
Все задачи запущены
<<< end   task2
<<< end   task1
End   2021-03-16 09:58:14.821104
```

При выполнении строк с созданием задач, выполнение сопрограмм уже запланировано и цикл событий их запустит, как только появится возможность. При этом обе сопрограммы уже будут работать и await уже только ждет их результат.

Примеры использования await и create_task

Что именно использовать await или create_task для запуска сопрограмм, зависит от ситуации. Некоторые действия внутри функции должны выполняться последовательно, некоторые могут выполняться параллельно.

Например, если есть две сопрограмы get_command_output и parse_command_output:

```
async def get_command_output(device_ip, show_command):
    print(f">>> Start get_command_output {device_ip}")
    await asyncio.sleep(1)
    print(f"<<< End  get_command_output  {device_ip}")


async def parse_command_output(output):
    print(">>> Start parse_command_output")
    await asyncio.sleep(1)
    print("<<< End parse_command_output")
```

Внутри функции, которая должна получить вывод команды и распарсить его, эти функции надо вызывать последовательно, не параллельно, поэтому используется await:

```
async def get_and_parse_show_command(device, command):
    print(f"### Start get_and_parse_show_command {device}")
    output = await get_command_output(device, command)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    parsed_data = await parse_command_output(output)
    print(f"### End   get_and_parse_show_command {device}")
```

При этом саму функцию get_and_parse_show_command можно запускать параллельно для разных устройств, поэтому используем create_task.

```
async def main():
    print(f'Start {datetime.now()}')
    tasks = [asyncio.create_task(get_and_parse_show_command(ip, "sh ip int br"))
             for ip in ["10.1.1.1", "10.2.2.2", "10.3.3.3"]]
    results = [await t for t in tasks]
    print(f'End {datetime.now()}')

In [15]: asyncio.run(main())
Start 2021-03-16 10:29:24.280408
### Start get_and_parse_show_command 10.1.1.1
>>> Start get_command_output 10.1.1.1
### Start get_and_parse_show_command 10.2.2.2
>>> Start get_command_output 10.2.2.2
### Start get_and_parse_show_command 10.3.3.3
>>> Start get_command_output 10.3.3.3
<<< End  get_command_output  10.1.1.1
>>> Start parse_command_output
<<< End  get_command_output  10.2.2.2
>>> Start parse_command_output
<<< End  get_command_output  10.3.3.3
>>> Start parse_command_output
<<< End parse_command_output
### End   get_and_parse_show_command 10.1.1.1
<<< End parse_command_output
### End   get_and_parse_show_command 10.2.2.2
<<< End parse_command_output
### End   get_and_parse_show_command 10.3.3.3
End 2021-03-16 10:29:26.286659
```

Еще один пример работы с await и create_task, но теперь функции выполняют много действий и после каждого выполняется asyncio.sleep:

```
async def print_number(task_name):
    print(f">>> Start {task_name}")
    for _ in range(10):
        print(42)
        await asyncio.sleep(0.5)
    print(f"<<< End   {task_name}")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
async def print_text(task_name):
    print(f">>> Start {task_name}")
    for _ in range(10):
        print("hello")
        await asyncio.sleep(0.9)
    print(f"<<< End   {task_name}")


async def main():
    print(f'Start {datetime.now()}')
    task1 = asyncio.create_task(print_number('task1'))
    task2 = asyncio.create_task(print_text('task2'))

    await task1
    await task2
    print(f'End   {datetime.now()}')


In [11]: asyncio.run(main())
Start 2021-03-16 15:35:31.668084
>>> Start task1
42
>>> Start task2
hello
42
hello
42
42
hello
42
42
hello
42
42
hello
42
42
hello
42
<<< End   task1
hello
hello
hello
hello
<<< End   task2
End   2021-03-16 15:35:40.684632
```

Также очень важно понимать, что сопрограммы не являются асинхронными сами по себе и если не отдавать в сопрограмме управление, то она будет блокирующей и ничего не будет выполняться, пока она не выполнится до конца. Например, если в примере выше изменить asyncio.sleep на time.sleep в функции print_number, то сначала выведутся print из print_number и только потом print_text:

```
import time

async def print_number(task_name):
    print(f">>> Start {task_name}")
    for _ in range(10):
        print(42)
        time.sleep(0.2)
    print(f"<<< End   {task_name}")


In [16]: asyncio.run(main())
Start 2021-03-16 15:43:31.593333
>>> Start task1
42
42
42
42
42
42
42
42
42
42
<<< End   task1
>>> Start task2
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
<<< End   task2
End   2021-03-16 15:43:42.614749
```

Запуск нескольких awaitables

Тут рассматриваются функции, которые позволяют запускать несколько сопрограмм или задач:

- asyncio.gather
- asyncio.as_completed

Примечание: Кроме функций gather и as_completed, несколько awaitables может запускать и функция wait, но она рассматривается позже, в 18 разделе.

---

asyncio.gather

Функция gather запускает на выполнение awaitable объекты, которые перечислены в последовательности aws:

```
asyncio.gather(*aws, return_exceptions=False)
```

Если какие-то из объектов являются сопрограммами, они автоматически оборачиваются в задачи и планируются на выполнение уже как объекты Task.

В данном примере функция connect_ssh якобы делает подключение к устройству по SSH и отправляет команду. Все реальные действия пока заменены на asyncio.sleep. В зависимости от числа, которое передается как аргумент, выполнение сопрограмм, которые возвращает функция connect_ssh, занимает разное время. Функция send_command_to_devices создает сопрограммы с помощью map и запускает их на выполнение с помощью asyncio.gather:

```
async def connect_ssh(ip, command):
    print(f'Подключаюсь к {ip}')
    await asyncio.sleep(ip)
    print(f'Отправляю команду {command} на устройство {ip}')
    await asyncio.sleep(1)
    return f"{command} {ip}"


async def send_command_to_devices(ip_list, command):
    coroutines = map(connect_ssh, ip_list, repeat(command))
    result = await asyncio.gather(*coroutines)
    return result
```

Если все объекты отработали корректно, asyncio.gather вернет список со значениями, которые вернули объекты. Порядок значений в списке соответствует порядку объектов:

```
In [2]: ip_list = [5, 2, 3, 7]

In [3]: result = asyncio.run(send_command_to_devices(ip_list, 'test'))
Подключаюсь к 5
Подключаюсь к 2
Подключаюсь к 3
Подключаюсь к 7
Отправляю команду test на устройство 2
Отправляю команду test на устройство 3
Отправляю команду test на устройство 5
Отправляю команду test на устройство 7

In [4]: result
Out[4]: ['test 5', 'test 2', 'test 3', 'test 7']
```

Если return_exceptions равно False (по умолчанию), при возникновении исключения, оно появляется в том месте, где ожидается (await) результат asyncio.gather:

```
async def connect_ssh(ip, command):
    print(f'Подключаюсь к {ip}')
    await asyncio.sleep(ip)
    if ip == 3:
        raise OSError(f'Не могу подключиться к {ip}')
    print(f'Отправляю команду {command} на устройство {ip}')
    await asyncio.sleep(1)
    return f"{command} {ip}"


In [11]: result = asyncio.run(send_command_to_devices(ip_list, 'test'))
Подключаюсь к 5
Подключаюсь к 2
Подключаюсь к 3
Подключаюсь к 7
Отправляю команду test на устройство 2
---------------------------------------------------------------------------
OSError                                   Traceback (most recent call last)
<ipython-input-11-4c2a35eaf7cd> in <module>
----> 1 result = asyncio.run(send_command_to_devices(ip_list, 'test'))
...

<ipython-input-1-7f470cb98776> in send_command_to_devices(ip_list, command)
     13 async def send_command_to_devices(ip_list, command):
     14     coroutines = map(connect_ssh, ip_list, repeat(command))
---> 15     result = await asyncio.gather(*coroutines)
     16     return result
```

(continues on next page)

(продолжение с предыдущей страницы)

```
<ipython-input-10-5e26dce87ca7> in connect_ssh(ip, command)
      3     await asyncio.sleep(ip)
      4     if ip == 3:
----> 5         raise OSError(f'Не могу подключиться к {ip}')
      6     print(f'Отправляю команду {command} на устройство {ip}')
      7     await asyncio.sleep(1)

OSError: Не могу подключиться к 3
```

Если return_exceptions равно True, исключение попадает в список как результат:

```
async def connect_ssh(ip, command):
    print(f'Подключаюсь к {ip}')
    await asyncio.sleep(ip)
    if ip == 3:
        raise OSError(f'Не могу подключиться к {ip}')
    print(f'Отправляю команду {command} на устройство {ip}')
    await asyncio.sleep(1)
    return f"{command} {ip}"


async def send_command_to_devices(ip_list, command):
    coroutines = map(connect_ssh, ip_list, repeat(command))
    result = await asyncio.gather(*coroutines, return_exceptions=True)
    return result


In [14]: result = asyncio.run(send_command_to_devices(ip_list, 'test'))
Подключаюсь к 5
Подключаюсь к 2
Подключаюсь к 3
Подключаюсь к 7
Отправляю команду test на устройство 2
Отправляю команду test на устройство 5
Отправляю команду test на устройство 7

In [15]: result
Out[15]: ['test 5', 'test 2', OSError('Не могу подключиться к 3'), 'test 7']

In [16]: result[2]
Out[16]: OSError('Не могу подключиться к 3')

In [17]: isinstance(result[2], Exception)
Out[17]: True
```

asyncio.as_completed

Функция as_completed запускает на выполнение awaitable объекты, которые перечислены в последовательности aws:

```
asyncio.as_completed(aws, *, timeout=None)
```

Возвращает итератор с сопрограмами, в порядке получения результата от сопрограмм. Функция генерирует исключение asyncio.TimeoutError, если за timeout отработали не все сопрограмы.

Пример использования as_completed:

```
async def delay_print(task_name):
    delay = round(random.random() * 10, 2)
    print(f'>>> start {task_name} sleep {delay}')
    await asyncio.sleep(delay)
    print(f'<<< end   {task_name}')
    return task_name


async def main():
    coroutines = [delay_print(f"task {i}") for i in range(1, 6)]
    for cor in asyncio.as_completed(coroutines):
        cor_result = await cor
        print(f"DONE {cor_result}")
```

Результаты возвращаются в порядке отрабатывания сопрограм, а не в порядке их запуска:

```
In [27]: asyncio.run(main())
>>> start task 2 sleep 8.93
>>> start task 1 sleep 0.03
>>> start task 4 sleep 8.33
>>> start task 5 sleep 3.43
>>> start task 3 sleep 5.09
<<< end   task 1
DONE task 1
<<< end   task 5
DONE task 5
<<< end   task 3
DONE task 3
<<< end   task 4
DONE task 4
<<< end   task 2
DONE task 2
```

Это может быть полезно когда сразу после получения результата, надо запускать следующую операцию. Например, в примере ниже сразу после получения результата сопрограмы,

идет запись результата в файл:

```
import asyncio
import time
from datetime import datetime
import random


async def connect_ssh(ip, command):
    print(f"Подключаюсь к {ip}")
    await asyncio.sleep(1)
    print(f"Отправляю команду {command}")
    await asyncio.sleep(random.random() * 10)
    print(f"Получен результат от {ip}")
    return ip, command


async def write_to_file(filename, data):
    print(f">>> Записываю результат в файл {filename}")
    await asyncio.sleep(1)
    print(f"<<< Результат записан в файл {filename}")


async def main():
    ip_list = ["10.1.1.1", "10.1.1.2", "10.1.1.3", "10.1.1.4"]
    coroutines = [connect_ssh(ip, "sh clock") for ip in ip_list]
    tasks = []
    for coro in asyncio.as_completed(coroutines):
        result = await coro
        tasks.append(asyncio.create_task(write_to_file(f"{result[0]}.txt", result)))
    await asyncio.gather(*tasks)


if __name__ == "__main__":
    asyncio.run(main())
```

Дополнительные материалы

Документация:

- Модуль asyncio

Статьи:

- Overview of Async IO in Python 3.7. Перевод статьи

- Asynchronous Python, Вторая часть - статья о том зачем asyncio нужен, чем отличается от потоков и немного истории
- Относительно простое объяснение зачем нужен asyncio
- Multiprocessing VS Threading VS AsyncIO in Python
- What are the advantages of asyncio over threads?

Подводка к asyncio и другие варианты асинхронного программирования в Python:

- Асинхронность в Python (Олег Молчанов) - Основы асинхронности в Python. О событийных циклах, генераторах, asyncio, async/await
- A Hitchhikers Guide to Asynchronous Programming
- Python Concurrency with asyncio. Chapter 3: A first asyncio application (publication in January 2022)

2. Асинхронные модули

Почти все модули, которые работали в многопоточном варианте, не будут работать с asyncio, так как они будут блокировать работу и не будут отдавать управление циклу событий.

В этом разделе рассматриваются модули:

- asyncssh
- scrapli

asyncssh

Модуль asynssh это реализация SSHv2 клиента и сервера. Модуль написан с использованием asyncio и требует Python 3.6+.

Основы asynssh

Модуль asynssh это реализация SSHv2 клиента и сервера. Модуль написан с использованием asyncio и требует Python 3.6+.

Установка asyncssh (в книге показаны примеры на asyncssh 2.5.0):

```
pip install asyncssh
```

Подключение выполняется таким образом:

```
In [1]: import asyncssh

In [2]: ssh = await asyncssh.connect(
   ...:     "192.168.100.1",
   ...:     username="cisco",
   ...:     password="cisco",
   ...:     encryption_algs="+aes128-cbc,aes256-cbc",
   ...: )
```

Можно попробовать сначала подключиться без указания алгоритмов (без параметра encryption_algs), а если возникнет ошибка, добавить недостающие алгоритмы.

После подключения, для работы с интерактивной сессией при подключении к сетевому оборудовании, удобнее всего использовать метод open_session, так как он возвращает три отдельных объекта stdin, stdout и stderr. В примере ниже stdin называется writer, a stdout reader. Так как оборудование с Cisco IOS не выводит информацию на stderr, далее этот объект не используется. Тут term_type равен Dumb, но при подключении к Linux, например, тут может указываться, например, `xterm-color`.

```
In [3]: writer, reader, stderr = await ssh.open_session(
   ...:     term_type="Dumb", term_size=(200, 24)
   ...: )
```

reader.readuntil, writer.write

Дальше вся работа будет через объекты reader (класс SSHReader) и writer (класс SSHWriter) и методы readuntil и write соответственно. При этом reader.readuntil это сопрограмма, поэтому надо писать await, а writer.write не сопрограмма, поэтому await использовать не нужно:

```
In [4]: await reader.readuntil(">")
Out[4]: '\r\nR1>'

In [5]: writer.write("enable\n")

In [6]: await reader.readuntil("Password")
Out[6]: 'enable\r\nPassword'

In [7]: writer.write("cisco\n")

In [8]: await reader.readuntil("#")
Out[8]: ': \r\nR1#'
```

Аналогично отправка команд и получение вывода:

```
In [9]: writer.write("terminal length 0\n")

In [10]: await reader.readuntil("#")
Out[10]: 'terminal length 0\r\nR1#'

In [11]: writer.write("sh ip int br\n")

In [12]: output = await reader.readuntil("#")

In [13]: output
Out[13]: 'sh ip int br\r\nInterface                  IP-Address      OK? Method Status
↪            Protocol\r\nEthernet0/0              192.168.100.1   YES NVRAM  up
↪            up      \r\nEthernet0/1              192.168.200.1   YES NVRAM  up
↪            up      \r\nEthernet0/2              unassigned      YES NVRAM  up
↪            up      \r\nEthernet0/3              192.168.130.1   YES NVRAM  up
↪            up      \r\nLoopback8                10.8.8.8        YES manual up
↪            up      \r\nLoopback9                10.90.90.1      YES manual up
↪            up      \r\nLoopback22               10.2.2.2        YES NVRAM  up
↪            up      \r\nLoopback33               unassigned      YES unset  up
↪            up      \r\nLoopback55               5.5.5.5         YES NVRAM  up
↪            up      \r\nLoopback100              10.1.1.100      YES manual up
↪            up      \r\nLoopback123              123.1.2.3       YES NVRAM  up
↪            up      \r\nLoopback300              10.30.3.3       YES manual up
↪            up      \r\nR1#'

In [14]: ssh.close()
```

Тот же код в виде функции:

```
from pprint import pprint
import asyncio
import asyncssh


async def send_show(host, username, password, enable_password, command):
    ssh = await asyncssh.connect(
        host=host,
        username=username,
        password=password,
        encryption_algs="+aes128-cbc,aes256-cbc",
    )

    writer, reader, stderr = await ssh.open_session(
        term_type="Dumb", term_size=(200, 24)
    )
    await reader.readuntil(">")
    writer.write("enable\n")
    await reader.readuntil("Password")
    writer.write(f"{enable_password}\n")
    await reader.readuntil([">", "#"])
    writer.write("terminal length 0\n")
    await reader.readuntil("#")

    writer.write(f"{command}\n")
    output = await reader.readuntil("#")
    ssh.close()
    return output


if __name__ == "__main__":
    r1 = {
        'host': '192.168.100.1',
        'username': 'cisco',
        'password': 'cisco',
        'enable_password': 'cisco',
    }
    result = asyncio.run(send_show(**r1, command="sh ip int br"))
    print(result)
```

Пока что это одна функция, которая последовательно выполняет ряд действий на одном устройстве, но каждый await в функции, это точка где идет ожидание ввода-вывода и в этих точках можно переключаться на другие функции. Например, если запустить подключение с помощью этой функции на несколько устройств.

```
async def send_command_to_devices(devices, command):
    coroutines = [send_show(**device, command=command) for device in devices]
    result = await asyncio.gather(*coroutines)
    return result


if __name__ == "__main__":
    devices = [
        {'host': '192.168.100.1',
         'username': 'cisco',
         'password': 'cisco',
         'enable_password': 'cisco'},
        {'host': '192.168.100.2',
         'username': 'cisco',
         'password': 'cisco',
         'enable_password': 'cisco'},
        {'host': '192.168.100.3',
         'username': 'cisco',
         'password': 'cisco',
         'enable_password': 'cisco'},
    ]
    result = asyncio.run(send_command_to_devices(devices, "sh ip int br"))
    pprint(result, width=120)
```

Теперь подключение выполняется на три устройства параллельно, с помощью gather.

asyncio.wait_for

У метода readuntil есть одна проблема - у него нет параметра timeout, в итоге, если указанная строка не найдена, метод зависает, пока соединение не прервется. Исправить это можно с помощью функции asyncio.wait_for:

```
asyncio.wait_for(aw, timeout)
```

Функция wait_for запускает awaitable и ждет его выполнение указанный timeout. Если сопрограмма не выполнилась за timeout, генерируется исключение asyncio.TimeoutError.

С использованием asyncio.wait_for функция send_show будет выглядеть так:

```
async def send_show(host, username, password, enable_password, command):
    print(f"Подключение к {host}")
    ssh = await asyncssh.connect(
        host=host,
        username=username,
        password=password,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        encryption_algs="+aes128-cbc,aes256-cbc",
    )
    writer, reader, stderr = await ssh.open_session(
        term_type="Dumb", term_size=(200, 24)
    )
    try:
        await asyncio.wait_for(reader.readuntil(">"), timeout=3)
        writer.write("enable\n")
        await asyncio.wait_for(reader.readuntil("Password"), timeout=3)
        writer.write(f"{enable_password}\n")
        await asyncio.wait_for(reader.readuntil([">", "#"]), timeout=3)
        writer.write("terminal length 0\n")
        await asyncio.wait_for(reader.readuntil("#"), timeout=3)

        print(f"Отправка команды {command} на {host}")
        writer.write(f"{command}\n")
        output = await asyncio.wait_for(reader.readuntil("#"), timeout=3)
        ssh.close()
        return output
    except asyncio.TimeoutError as error:
        print("TimeoutError при выполнении reader.readuntil")
```

Так как писать asyncio.wait_for для каждого вызова reader.readuntil не очень удобно, можно сделать отдельную сопрограмму, которая выполняет эти операции, а также выводит последний вывод, который был получен с устройства, если readuntil не дождался нужного символа:

```
async def read_until(reader, line, timeout=3):
    try:
        return await asyncio.wait_for(reader.readuntil(line), timeout)
    except asyncio.TimeoutError as error:
        output = ""
        while True:
            try:
                output += await asyncio.wait_for(reader.read(1000), 0.1)
            except asyncio.TimeoutError as error:
                break
        print(
            f"TimeoutError при выполнении reader.readuntil('{line}')\n"
            f"Последний вывод:"
        )
        print(output)
```

Функция read_until сначала пытается выполнить reader.readuntil с указанным разделителем, при этом asyncio.wait_for ждет выполнения reader.readuntil только указанный timeout. Если разделитель line не найден, функция read_until пытается считать доступный

вывод с помощью метода reader.read.

Теперь функция send_show выглядит так:

```
async def send_show(host, username, password, enable_password, command):
    print(f"Подключение к {host}")
    ssh = await asyncssh.connect(
        host=host,
        username=username,
        password=password,
        encryption_algs="+aes128-cbc,aes256-cbc",
    )
    writer, reader, stderr = await ssh.open_session(
        term_type="Dumb", term_size=(200, 24)
    )
    await read_until(reader, ">")
    writer.write("enable\n")
    await read_until(reader, "Password")
    writer.write(f"{enable_password}\n")
    await read_until(reader, [">", "#"])
    writer.write("terminal length 0\n")
    await read_until(reader, "#")

    print(f"Отправка команды {command} на {host}")
    writer.write(f"{command}\n")
    output = await read_until(reader, "#")
    ssh.close()
    return output
```

И если указанная строка не была найдена в выводе, функция read_until выведет такую информацию на stdout:

```
TimeoutError при выполнении reader.readuntil('>')
Последний вывод:
sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
Ethernet0/0                192.168.100.3   YES NVRAM  up                    up
Ethernet0/1                10.100.23.3     YES NVRAM  up                    up
Ethernet0/2                unassigned      YES NVRAM  administratively down down
Ethernet0/3                unassigned      YES NVRAM  administratively down down
Loopback9                  unassigned      YES unset  up                    up
R3#
```

Соответственно будет видно при выполнении какой команды возникло исключение asyncio.TimeoutError и какой вывод был получен с устройства.

async with

До сих пор подключение выполнялось не в менеджере контекста. Модуль asynssh может выполнять подключение в менеджере контекста, но не в обычном блоке with, а в async with. Для асинхронного менеджера контекста созданы отдельные специальные методы __aenter__, __aexit__, которые равнозначны синхронным вариантам по смыслу, но при этом являются сопрограммами.

Пример подключения с помощью асинхронного менеджера контекста:

```
async def send_show(host, username, password, enable_password, command):
    print(f"Подключение к {host}")
    async with asyncssh.connect(
        host=host,
        username=username,
        password=password,
        encryption_algs="+aes128-cbc,aes256-cbc",
    ) as ssh:
        writer, reader, stderr = await ssh.open_session(
            term_type="Dumb", term_size=(200, 24)
        )
        await read_until(reader, ">")
        writer.write("enable\n")
        await read_until(reader, "Password")
        writer.write(f"{enable_password}\n")
        await read_until(reader, [">", "#"])
        writer.write("terminal length 0\n")
        await read_until(reader, "#")

        print(f"Отправка команды {command} на {host}")
        writer.write(f"{command}\n")
        output = await read_until(reader, "#")
        return output
```

Подключение к нескольким устройствам

```
from pprint import pprint
import asyncio

import yaml
import asyncssh


async def read_until(reader, prompt="#", timeout=3, strict=True):
    try:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        return await asyncio.wait_for(reader.readuntil(prompt), timeout)
    except asyncio.TimeoutError as error:
        output = ""
        while True:
            try:
                output += await asyncio.wait_for(reader.read(1000), 0.1)
            except asyncio.TimeoutError as error:
                break
        message = (
            f"TimeoutError while executing reader.readuntil('{prompt}')\n"
            f"Last output from device:\n{output}"
        )
        if strict:
            raise asyncio.TimeoutError(message)
        else:
            print(message)
            return output


async def send_show(
    host, username, password, enable_password, command, connection_timeout=5
):
    try:
        ssh = await asyncio.wait_for(
            asyncssh.connect(
                host=host,
                username=username,
                password=password,
                encryption_algs="+aes128-cbc,aes256-cbc",
            ),
            timeout=connection_timeout,
        )
    except asyncio.TimeoutError:
        print(f"Connection Timeout on {host}")
    except asyncssh.PermissionDenied:
        print(f"Authentication Error on {host}")
    except asyncssh.Error as error:
        print(f"{error} on {host}")
    else:
        writer, reader, _ = await ssh.open_session(
            term_type="Dumb", term_size=(200, 24)
        )
        await read_until(reader, ">")
        writer.write("enable\n")
        await read_until(reader, "Password")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        writer.write(f"{enable_password}\n")
        await read_until(reader, [">", "#"])
        writer.write("terminal length 0\n")
        await read_until(reader)

        writer.write(f"{command}\n")
        output = await read_until(reader, "#")
        return output


async def send_command_to_devices(devices, command):
    coroutines = [send_show(**device, command=command) for device in devices]
    result = await asyncio.gather(*coroutines)
    return result


if __name__ == "__main__":
    with open("devices.yaml") as f:
        devices = yaml.safe_load(f)
    result = asyncio.run(send_command_to_devices(devices, "sh ip int br"))
    pprint(result, width=120)
```

scrapli

Примечание: Основы scrapli с использованием синхронных вариантов транспорта рассматриваются в книге Python для сетевых инженеров.

scrapli - это модуль, который позволяет подключаться к сетевому оборудованию используя Telnet, SSH или NETCONF.

scrapli поддерживает разные варианты подключения: system, paramiko, ssh2, telnet, asyncssh, asynctelnet. Тут рассматривается только asyncssh и asynctelnet.

Основы scrapli

scrapli поддерживает разные варианты подключения: system, paramiko, ssh2, telnet, asyncssh, asynctelnet. Тут рассматривается только asyncssh и asynctelnet.

Доступные варианты асинхронного транспорта:

- asyncssh
- asynctelnet (реализация telnet на основе asyncio)

Примечание: Рассматривается scrapli версии 2021.1.30.

Для асинхронного транспорта, как и для синхронного, в scrapli есть два варианта подключения: используя общий класс AsyncScrapli, который выбирает нужный driver по параметру platform или конкретный driver, например, AsyncIOSXEDriver. При этом параметры передаются те же самые и конкретному драйверу и Scrapli.

В целом методы и параметры методов такие же как и в синхронном варианте scrapli, отличается транспорт, драйверы и то, что методы являются сопрограммами.

Параметры подключения

Основные параметры подключения:

- host - IP-адрес или имя хоста
- auth_username - имя пользователя
- auth_password - пароль
- auth_secondary - пароль на enable

- auth_strict_key - контролирует проверку SSH ключей сервера, а именно разрешать ли подключаться к серверам ключ которых не сохранен в ssh/known_hosts. False - разрешить подключение (по умолчанию значение True)
- platform - нужно указывать при использовании AsyncScrapli
- transport - для async варианта transport указывать обязательно: asyncssh или asynctelnet
- transport_options - опции для конкретного транспорта

Процесс подключения немного отличается в зависимости от того используется асинхронный менеджер контекста или нет. При подключении без менеджера контекста, сначала надо передать параметры драйверу или AsyncScrapli, а затем вызвать метод open:

```
In [1]: from scrapli import AsyncScrapli

In [2]: r1 = {
   ...:     "host": "192.168.100.1",
   ...:     "auth_username": "cisco",
   ...:     "auth_password": "cisco",
   ...:     "auth_secondary": "cisco",
   ...:     "auth_strict_key": False,
   ...:     "platform": "cisco_iosxe",
   ...:     "transport": "asyncssh",
   ...: }

In [3]: ssh = AsyncScrapli(**r1)

In [5]: await ssh.open()
```

После этого можно отправлять команды:

```
In [6]: await ssh.get_prompt()
Out[6]: 'R1#'

In [7]: await ssh.close()
```

При использовании асинхронного менеджера контекста, open вызывать не надо:

```
In [8]: async with AsyncScrapli(**r1) as ssh:
   ...:     print(await ssh.get_prompt())
R1#
```

Примечание: Отличия в подключении с менеджером контекста и без это особенность классов для работы с асинхронным подключением.

Использование драйвера

Доступные async драйверы:

Оборудование	Драйвер	Параметр platform
Cisco IOS-XE	AsyncIOSXEDriver	cisco_iosxe
Cisco NX-OS	AsyncNXOSDriver	cisco_nxos
Cisco IOS-XR	AsyncIOSXRDriver	cisco_iosxr
Arista EOS	AsyncEOSDriver	arista_eos
Juniper JunOS	AsyncJunosDriver	juniper_junos

Пример подключения с использованием драйвера IOSXEDriver (технически подключение выполняется к Cisco IOS):

```
In [10]: from scrapli.driver.core import AsyncIOSXEDriver

In [11]: r1_driver = {
    ...:     "host": "192.168.100.1",
    ...:     "auth_username": "cisco",
    ...:     "auth_password": "cisco",
    ...:     "auth_secondary": "cisco",
    ...:     "auth_strict_key": False,
    ...:     "transport": "asyncssh",
    ...: }

In [12]: async with AsyncIOSXEDriver(**r1_driver) as ssh:
    ...:     print(await ssh.get_prompt())
R1#
```

Пример базового использования scrapli

В остальном, принципы работы те же, что и с синхронным вариантом.

Пример подключения к одному устройству с помощью asyncssh и AsyncScrapli:

```
import asyncio
from scrapli import AsyncScrapli
from scrapli.exceptions import ScrapliException

r1 = {
    "host": "192.168.100.1",
    "auth_username": "cisco",
    "auth_password": "cisco",
    "auth_secondary": "cisco",
    "auth_strict_key": False,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    "timeout_socket": 5,  # timeout for establishing socket/initial connection in seconds
    "timeout_transport": 10,  # timeout for ssh|telnet transport in seconds
    "platform": "cisco_iosxe",
    "transport": "asyncssh",
}


async def send_show(device, command):
    try:
        async with AsyncScrapli(**device) as conn:
            result = await conn.send_command(command)
            return result.result
    except ScrapliException as error:
        print(error, device["host"])


if __name__ == "__main__":
    output = asyncio.run(send_show(r1, "show ip int br"))
    print(output)
```

Получение структурированного вывода с помощью TextFSM

Пример получения структурированного вывода с помощью TextFSM. Функция возвращает структурированный вывод, если удалось его получить (есть шаблон и вернулось не пустое значение) и обычный вывод команды, если нет:

```
from pprint import pprint
import asyncio
from scrapli import AsyncScrapli
from scrapli.exceptions import ScrapliException

r1 = {
    "host": "192.168.100.1",
    "auth_username": "cisco",
    "auth_password": "cisco",
    "auth_secondary": "cisco",
    "auth_strict_key": False,
    "timeout_socket": 5,  # timeout for establishing socket/initial connection in seconds
    "timeout_transport": 10,  # timeout for ssh|telnet transport in seconds
    "platform": "cisco_iosxe",
    "transport": "asyncssh",
}

async def send_show(device, show_commands):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    cmd_dict = {}
    if type(show_commands) == str:
        show_commands = [show_commands]
    try:
        async with AsyncScrapli(**device) as ssh:
            for cmd in show_commands:
                reply = await ssh.send_command(cmd)
                parsed_data = reply.textfsm_parse_output()
                if parsed_data:
                    cmd_dict[cmd] = parsed_data
                else:
                    cmd_dict[cmd] = reply.result
        return cmd_dict
    except ScrapliException as error:
        print(error, device["host"])


if __name__ == "__main__":
    output = asyncio.run(send_show(r1, ["sh run | i ^interface", "show ip int br"]))
    pprint(output)
```

Подключение к нескольким устройствам

Пример подключения к нескольким устройствам:

```
from pprint import pprint
import asyncio

import yaml
from scrapli import AsyncScrapli
from scrapli.exceptions import ScrapliException


async def send_show(device, show_commands):
    cmd_dict = {}
    if type(show_commands) == str:
        show_commands = [show_commands]
    try:
        async with AsyncScrapli(**device) as ssh:
            for cmd in show_commands:
                reply = await ssh.send_command(cmd)
                cmd_dict[cmd] = reply.result
        return cmd_dict
    except ScrapliException as error:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        print(error, device["host"])


async def send_command_to_devices(devices, commands):
    coroutines = [send_show(device, commands) for device in devices]
    result = await asyncio.gather(*coroutines)
    return result


if __name__ == "__main__":
    with open("devices_async.yaml") as f:
        devices = yaml.safe_load(f)
    result = asyncio.run(send_command_to_devices(devices, "sh ip int br"))
    pprint(result, width=120)
```

Файл devices_async.yaml:

```
- host: 192.168.100.1
  auth_username: cisco
  auth_password: cisco
  auth_secondary: cisco
  auth_strict_key: false
  timeout_socket: 5
  timeout_transport: 10
  platform: cisco_iosxe
  transport: asyncssh
- host: 192.168.100.2
  auth_username: cisco
  auth_password: cisco
  auth_secondary: cisco
  auth_strict_key: false
  timeout_socket: 5
  timeout_transport: 10
  platform: cisco_iosxe
  transport: asyncssh
- host: 192.168.100.3
  auth_username: cisco
  auth_password: cisco
  auth_secondary: cisco
  auth_strict_key: false
  timeout_socket: 5
  timeout_transport: 10
  platform: cisco_iosxe
  transport: asyncssh
```

Подключение с транспортом asynctelnet

При подключении asynctelnet надо указать транспорт asynctelnet и порт 23. Кроме того, надо данный момент (scrapli 2021.1.30) при подключении asynctelnet к недоступному адресу таймаут будет через 2 минуты, чтобы уменьшить его, можно использовать async_timeout:

```
import asyncio
from scrapli.driver.core import AsyncIOSXEDriver
from scrapli.exceptions import ScrapliException
from async_timeout import timeout

r1 = {
    "host": "192.168.100.11",
    "auth_username": "cisco",
    "auth_password": "cisco",
    "auth_secondary": "cisco",
    "auth_strict_key": False,
    "transport": "asynctelnet",
    "port": 23,
}


async def send_show(device, command):
    # На данный момент (scrapli 2021.1.30) таймаут при подключении к недоступному
    # хосту будет 2 минуты, поэтому пока что лучше добавлять wait_for или
    # async_timeout вокруг подключения
    try:
        async with timeout(10):
            async with AsyncIOSXEDriver(**device) as ssh:
                result = await ssh.send_command(command)
                return result.result
    except ScrapliException as error:
        print(error, device["host"])
    except asyncio.exceptions.TimeoutError:
        print("asyncio.exceptions.TimeoutError", device["host"])


if __name__ == "__main__":
    output = asyncio.run(send_show(r1, "show ip int br"))
    print(output)
```

Дополнительные материалы

Документация:

- Модуль asyncio
- aiofiles
- aiohttp
- async-timeout

asyncssh:

- asyncssh
- Примеры asyncssh

scrapli:

- scrapli
- основы и подключение к нескольким устройствам
- основы, scrapli-cfg, NETCONF

Полезные ссылки:

- The current state and future of Asyncio
- Другие модули async
- aiojobs
- Structured concurrency in Python with AnyIO
- Designing Libraries for Async and Sync I/O

Примеры кода:

- asyncssh
- scrapli

3. Создание классов с asyncio

В целом асинхронные классы пишутся так же, как синхронные, но есть несколько отличий:

- особенности создания экземпляров, так как `__init__` не может быть сопрограммой
- некоторые специальные методы имеют отдельные версии для асинхронных классов, например, `__aenter__`, `__aexit__`, `__anext__`, `__aiter__`

Методы класса могут быть сопрограммами или обычными функциями.

Создание экземпляра

В синхронном варианте, при создании класса который подключается по SSH, само подключение обычно будет выполняться в методе `__init__` (напрямую или через вызов другого метода). В асинхронных классах так сделать не получится, так как для выполнения подключения в `__init__`, надо сделать init сопрограммой. Так как init должен возвращать None, если сделать init сопрограммой, возникнет ошибка:

```
In [3]: class ConnectSSH:
   ...:     async def __init__(self):
   ...:         await asyncio.sleep(0.1)
   ...:

In [5]: r1 = ConnectSSH()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-ceb9b33fd87d> in <module>
----> 1 r1 = ConnectSSH()

TypeError: __init__() should return None, not 'coroutine'
```

Пожалуй, самый распространенный вариант решения проблемы - создание отдельного метода для подключения (вместе с добавлением асинхронного менеджера контекста).

```
from pprint import pprint
import asyncio
import asyncssh


class ConnectAsyncSSH:
    def __init__(self, host, username, password, enable_password, connection_timeout=5):
        self.host = host
        self.username = username
        self.password = password
        self.enable_password = enable_password
        self.connection_timeout = connection_timeout

    async def connect(self):
        self._ssh = await asyncio.wait_for(
            asyncssh.connect(
                host=self.host,
                username=self.username,
                password=self.password,
                encryption_algs="+aes128-cbc,aes256-cbc",
            ),
            timeout=self.connection_timeout,
        )
        self.writer, self.reader, _ = await self._ssh.open_session(
            term_type="Dumb", term_size=(200, 24)
        )
        await self.reader.readuntil(">")
        self.writer.write("enable\n")
        await self.reader.readuntil("Password")
        self.writer.write(f"{self.enable_password}\n")
        await self.reader.readuntil("#")
        self.writer.write("terminal length 0\n")
        await self.reader.readuntil("#")

    async def get_prompt(self):
        self.writer.write("\n")
        output = await self.reader.readuntil("#")
        return output
```

В этом случае для подключения надо сначала создать экземпляр класса, а потом вызвать метод connect где выполняется само подключение:

```
async def main():
    r1 = {
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        "host": "192.168.100.1",
        "username": "cisco",
        "password": "cisco",
        "enable_password": "cisco",
    }
    ssh = ConnectAsyncSSH(**r1)
    await ssh.connect()
    print(await ssh.get_prompt())


if __name__ == "__main__":
    asyncio.run(main())
```

Как правило, вместе с таким вариантом используется асинхронный менеджер контекста. В этом случае, метод connect вызывается в методе __aenter__ (аналог __enter__). Асинхронный менеджер контекста рассматривается позже.

classmethod

Еще один распространенный вариант - использование classmethod для создания экземпляра. В примере ниже classmethod connect единственный способ создания экземлпяра:

```
from pprint import pprint
import asyncio
import asyncssh


class ConnectAsyncSSH:

    @classmethod
    async def connect(cls, host, username, password, enable_password, connection_
↪timeout=5):
        self = cls()

        self.host = host
        self.username = username
        self.password = password
        self.enable_password = enable_password
        self.connection_timeout = connection_timeout
        self._ssh = await asyncio.wait_for(
            asyncssh.connect(
                host=self.host,
                username=self.username,
                password=self.password,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                encryption_algs="+aes128-cbc,aes256-cbc",
            ),
            timeout=self.connection_timeout,
        )
        self.writer, self.reader, _ = await self._ssh.open_session(
            term_type="Dumb", term_size=(200, 24)
        )
        await self.reader.readuntil(">")
        self.writer.write("enable\n")
        await self.reader.readuntil("Password")
        self.writer.write(f"{self.enable_password}\n")
        await self.reader.readuntil("#")
        self.writer.write("terminal length 0\n")
        await self.reader.readuntil("#")
        return self

    async def get_prompt(self):
        self.writer.write("\n")
        output = await self.reader.readuntil("#")
        return output
```

Создание экземпляра выглядит так:

```
async def main():
    r1 = {
        "host": "192.168.100.1",
        "username": "cisco",
        "password": "cisco",
        "enable_password": "cisco",
    }
    ssh = await ConnectAsyncSSH.connect(**r1)
    print(await ssh.get_prompt())


if __name__ == "__main__":
    asyncio.run(main())
```

Второй вариант - оставить init и обычный метод connect и добавить отдельный classmethod:

```
class ConnectAsyncSSH:
    def __init__(self, host, username, password, enable_password, connection_timeout=5):
        self.host = host
        self.username = username
        self.password = password
        self.enable_password = enable_password
        self.connection_timeout = connection_timeout
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    async def connect(self):
        self._ssh = await asyncio.wait_for(
            asyncssh.connect(
                host=self.host,
                username=self.username,
                password=self.password,
                encryption_algs="+aes128-cbc,aes256-cbc",
            ),
            timeout=self.connection_timeout,
        )
        self.writer, self.reader, _ = await self._ssh.open_session(
            term_type="Dumb", term_size=(200, 24)
        )
        await self.reader.readuntil(">")
        self.writer.write("enable\n")
        await self.reader.readuntil("Password")
        self.writer.write(f"{self.enable_password}\n")
        await self.reader.readuntil("#")
        self.writer.write("terminal length 0\n")
        await self.reader.readuntil("#")

    @classmethod
    async def create_connection(cls, host, username, password, enable_password,␣
↪connection_timeout=5):
        self = cls(host, username, password, enable_password, connection_timeout)
        await self.connect()
        return self

    async def get_prompt(self):
        self.writer.write("\n")
        output = await self.reader.readuntil("#")
        return output
```

Создание экземпяра через classmethod выглядит так, но при этом можно создавать экземпляр и через init + connect:

```
async def main():
    r1 = {
        "host": "192.168.100.1",
        "username": "cisco",
        "password": "cisco",
        "enable_password": "cisco",
    }
    ssh = await ConnectAsyncSSH.create_connection(**r1)
    print(await ssh.get_prompt())
```

Асинхронный менеджер контекста

В целом, смысл работы асинхронного менеджера контекста остается таким же, как и в синхронном. Только так как методы для подключения к устройству являются сопрограммами, их надо правильно ожидать (await) и соответственно специальные методы для создания асинхронного менеджера контекста, тоже должны быть сопрограммами. Поэтому для асинхронного менеджера контекста созданы отдельные специальные методы `__aenter__` и `__aexit__` и соответственно чтобы они вызывались, надо использовать асинхронный менеджер контекста `async with`, а не обычный `with`.

Примечание: `async with` можно использовать только внутри сопрограммы.

Пример класса с поддержкой протокола асинхронного менеджера контекста:

```
from pprint import pprint
import asyncio
import asyncssh


class ConnectAsyncSSH:
    def __init__(self, host, username, password, enable_password, connection_timeout=5):
        self.host = host
        self.username = username
        self.password = password
        self.enable_password = enable_password
        self.connection_timeout = connection_timeout

    async def connect(self):
        self._ssh = await asyncio.wait_for(
            asyncssh.connect(
                host=self.host,
                username=self.username,
                password=self.password,
                encryption_algs="+aes128-cbc,aes256-cbc",
            ),
            timeout=self.connection_timeout,
        )
        self.writer, self.reader, _ = await self._ssh.open_session(
            term_type="Dumb", term_size=(200, 24)
        )
        await self.reader.readuntil(">")
        self.writer.write("enable\n")
        await self.reader.readuntil("Password")
        self.writer.write(f"{self.enable_password}\n")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        await self.reader.readuntil("#")
        self.writer.write("terminal length 0\n")
        await self.reader.readuntil("#")

    async def get_prompt(self):
        self.writer.write("\n")
        output = await self.reader.readuntil("#")
        return output

    def close(self):
        self._ssh.close()

    async def __aenter__(self):
        await self.connect()
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        self.close()
```

Пример использования:

```
async def main():
    r1 = {
        "host": "192.168.100.1",
        "username": "cisco",
        "password": "cisco",
        "enable_password": "cisco",
    }
    async with ConnectAsyncSSH(**r1) as ssh:
        print(await ssh.get_prompt())


if __name__ == "__main__":
    asyncio.run(main())
```

Асинхронная итерация

При работе с асинхронным кодом, могут использоваться как обычные итераторы и итерируемые объекты, так и асинхронные. Асинхронные итераторы и итерируемые объекты в целом работают так же, как и синхронные. Для асинхронных итераторов и итерируемых объектов созданы отдельные методы `__aiter__` и `__anext__`, плюс для перебора асинхронных итерируемых объектов есть цикл `async for`.

Примечание: `async for` можно использовать только внутри сопрограммы.

Асинхронный итерируемый объект (asynchronous iterable) - это объект, который можно использовать в `async for`. В асинхронном итерируемом объекте должен быть метод `__aiter__`, который возвращает асинхронный итератор.

Асинхронный итератор (asynchronous iterator) - это объект, у которого есть методы `__anext__` и `__aiter__`. Метод `__anext__` должен возвращать awaitable объект. При завершении итерации генерируется исключение `StopAsyncIteration`.

Асинхронные итераторы можно создавать с помощью классов или асинхронных генераторов. Тут рассматривается вариант через классы, а в следующем разделе рассматриваются асинхронные генераторы.

Пример асинхронного итератора, который на каждой итерации пытается подключиться к одному устройству с помощью scrapli:

```
import asyncio
from datetime import datetime
from scrapli import AsyncScrapli
from scrapli.exceptions import ScrapliException
from async_timeout import timeout
import yaml


class CheckConnection:
    def __init__(self, device_list):
        self.device_list = device_list
        self._current_device = 0

    async def _scan_device(self, device):
        ip = device["host"]
        try:
            async with timeout(5): # для asynctelnet
                async with AsyncScrapli(**device) as conn:
                    prompt = await conn.get_prompt()
                return True, prompt
        except (ScrapliException, asyncio.exceptions.TimeoutError) as error:
            return False, f"{error} {ip}"

    async def __anext__(self):
        if self._current_device >= len(self.device_list):
            raise StopAsyncIteration
        device_params = self.device_list[self._current_device]
        scan_results = await self._scan_device(device_params)
        self._current_device += 1
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        return scan_results

    def __aiter__(self):
        return self
```

Смысл этого итератора в том чтобы проверить получится ли подключиться к оборудованию по SSH с помощью scrapli. Если подключиться получилось, __anext__ возвращает True и приглашение устройства, если нет - False и исключение.

Примечание: Обратите внимание на то, что метод __anext__ сопрограмма, а __aiter__ нет.

Для перебора асинхронного итератора надо использовать async for и соответственно перебор надо делать в сопрограмме:

```
async def ssh_scan(devices):
    check = CheckConnection(devices)
    async for status, msg in check:
        if status:
            print(f"{datetime.now()} SSH. Подключение успешно: {msg}")
        else:
            print(f"{datetime.now()} SSH. Не удалось подключиться: {msg}")


if __name__ == "__main__":
    with open("devices_asyncssh.yaml") as f:
        devices = yaml.safe_load(f)
    asyncio.run(ssh_scan(devices))
```

Содержимое файла devices_asyncssh.yaml (первые два устройства доступны и с правильными параметрами, третье доступно, но указан неправильный пароль, четвертое недоступно):

```
- host: 192.168.100.1
  auth_username: cisco
  auth_password: cisco
  auth_secondary: cisco
  auth_strict_key: false
  timeout_socket: 5
  timeout_transport: 10
  platform: cisco_iosxe
  transport: asyncssh
- host: 192.168.100.2
  auth_username: cisco
  auth_password: cisco
  auth_secondary: cisco
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  auth_strict_key: false
  timeout_socket: 5
  timeout_transport: 10
  platform: cisco_iosxe
  transport: asyncssh
- host: 192.168.100.3
  auth_username: cisco
  auth_password: ciscoe
  auth_secondary: cisco
  auth_strict_key: false
  timeout_socket: 5
  timeout_transport: 10
  platform: cisco_iosxe
  transport: asyncssh
- host: 192.168.100.11
  auth_username: cisco
  auth_password: cisco
  auth_secondary: cisco
  auth_strict_key: false
  timeout_socket: 5
  timeout_transport: 10
  platform: cisco_iosxe
  transport: asyncssh
```

Вызов скрипта:

```
$ python ex05_async_iterator_ssh.py
2021-04-19 11:25:40.775305 SSH. Подключение успешно: R1#
2021-04-19 11:25:41.334456 SSH. Подключение успешно: R2#
2021-04-19 11:25:43.638459 SSH. Не удалось подключиться: all authentication methods␣
↪failed 192.168.100.3
2021-04-19 11:25:48.647160 SSH. Не удалось подключиться: timed out opening connection to␣
↪device 192.168.100.11
```

Итератор CheckConnection проверяет устройства последовательно, но вместе с этим итератором, параллельно можно запускать что-то еще, например, паралелльно проверять подключение telnet (файл ex06_async_iterator_telnet_ssh.py):

```
async def scan(devices, protocol):
    check = CheckConnection(devices)
    async for status, msg in check:
        if status:
            print(f"{datetime.now()} {protocol}. Подключение успешно: {msg}")
        else:
            print(f"{datetime.now()} {protocol}. Не удалось подключиться: {msg}")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
async def main():
    with open("devices_asyncssh.yaml") as f:
        devices_ssh = yaml.safe_load(f)
    with open("devices_asynctelnet.yaml") as f:
        devices_telnet = yaml.safe_load(f)
    await asyncio.gather(scan(devices_ssh, "SSH"), scan(devices_telnet, "Telnet"))


if __name__ == "__main__":
    asyncio.run(main())
```

Вызов скрипта:

```
$ python ex06_async_iterator_telnet_ssh.py
2021-04-19 11:30:14.820195 SSH. Подключение успешно: R1#
2021-04-19 11:30:14.983307 Telnet. Подключение успешно: R1#
2021-04-19 11:30:15.296011 SSH. Подключение успешно: R2#
2021-04-19 11:30:15.449338 Telnet. Подключение успешно: R2#
2021-04-19 11:30:17.599259 SSH. Не удалось подключиться: all authentication methods␣
↪failed 192.168.100.3
2021-04-19 11:30:19.775107 Telnet. Не удалось подключиться: username/login prompt seen␣
↪more than once, assuming auth failed 192.168.100.3
2021-04-19 11:30:22.603411 SSH. Не удалось подключиться:  192.168.100.11
2021-04-19 11:30:24.777987 Telnet. Не удалось подключиться:  192.168.100.11
```

Примеры классов

Класс для подключения по SSH с помощью asyncssh

```
from pprint import pprint
import asyncio
import asyncssh


class ConnectAsyncSSH:
    def __init__(self, host, username, password, enable_password, connection_timeout=5):
        self.host = host
        self.username = username
        self.password = password
        self.enable_password = enable_password
        self.connection_timeout = connection_timeout
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    async def connect(self):
        self.ssh = await asyncio.wait_for(
            asyncssh.connect(
                host=self.host,
                username=self.username,
                password=self.password,
                encryption_algs="+aes128-cbc,aes256-cbc",
            ),
            timeout=self.connection_timeout,
        )
        self.writer, self.reader, _ = await self.ssh.open_session(
            term_type="Dumb", term_size=(200, 24)
        )
        await self._read_until(">")
        self.writer.write("enable\n")
        await self._read_until("Password")
        self.writer.write(f"{self.enable_password}\n")
        await self._read_until([">", "#"])
        self.writer.write("terminal length 0\n")
        await self._read_until()

    async def _read_until(self, prompt="#", timeout=3):
        try:
            return await asyncio.wait_for(self.reader.readuntil(prompt), timeout)
        except asyncio.TimeoutError as error:
            output = ""
            while True:
                try:
                    output += await asyncio.wait_for(self.reader.read(1000), 0.1)
                except asyncio.TimeoutError as error:
                    break
            message = (
                f"TimeoutError while executing self.reader.readuntil('{prompt}')\n"
                f"Last output from device:\n{output}"
            )
            raise asyncio.TimeoutError(message)

    async def send_show_command(self, command):
        self.writer.write(command + "\n")
        output = await self._read_until()
        return output

    async def send_config_commands(self, commands):
        cfg_output = ""
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        if type(commands) == str:
            commands = ["conf t", commands, "end"]
        else:
            commands = ["conf t", *commands, "end"]
        for cmd in commands:
            self.writer.write(cmd + "\n")
            cfg_output += await self._read_until()
        return cfg_output

    async def close(self):
        self.ssh.close()
        await self.ssh.wait_closed()

    async def __aenter__(self):
        await self.connect()
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        await self.close()


async def main():
    r1 = {
        "host": "192.168.100.1",
        "username": "cisco",
        "password": "cisco",
        "enable_password": "cisco",
    }
    config_commands = ["logging buffered 20010", "ip http server"]
    async with ConnectAsyncSSH(**r1) as ssh:
        print(await ssh.send_show_command("sh ip int br"))
        print(await ssh.send_config_commands(config_commands))


if __name__ == "__main__":
    asyncio.run(main())
```

Пример использования класса:

```
from pprint import pprint
import asyncio
import asyncssh

import yaml
from ex11_asyncssh_basic_class import ConnectAsyncSSH
```

(continues on next page)

(продолжение с предыдущей страницы)

```
async def send_show(device, command):
    host = device["host"]
    try:
        async with ConnectAsyncSSH(**device) as ssh:
            output = await ssh.send_show_command("sh ip int br")
            return output
    except asyncio.TimeoutError:
        print(f"Connection Timeout on {host}")
    except asyncssh.PermissionDenied:
        print(f"Authentication Error on {host}")
    except asyncssh.Error as error:
        print(f"{error} on {host}")


async def send_command_to_devices(devices, command):
    coroutines = [send_show(device, command) for device in devices]
    result = await asyncio.gather(*coroutines)
    return result


if __name__ == "__main__":
    with open("devices.yaml") as f:
        devices = yaml.safe_load(f)
    result = asyncio.run(send_command_to_devices(devices, "sh ip int br"))
    pprint(result, width=120)
```

Класс итератор для проверки доступности устройств ping'ом

```
import asyncio
from datetime import datetime
from scrapli import AsyncScrapli
from scrapli.exceptions import ScrapliException
from async_timeout import timeout
import yaml

from ex06_async_iterator_telnet_ssh import CheckConnection, scan


class CheckConnectionPing(CheckConnection):
    def __init__(self, device_list):
        self.device_list = device_list
        self._current_device = 0
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    async def _scan_device(self, device):
        reply = await asyncio.create_subprocess_shell(
            f"ping -c 3 -n {device}",
            stdout=asyncio.subprocess.PIPE,
            stderr=asyncio.subprocess.PIPE,
        )

        stdout, stderr = await reply.communicate()
        output = (stdout + stderr).decode("utf-8")

        if reply.returncode == 0:
            return True, output
        else:
            return False, output


async def scan(devices, protocol):
    protocol_class_map = {
        "ssh": CheckConnection,
        "telnet": CheckConnection,
        "icmp": CheckConnectionPing,
    }
    ConnectionClass = protocol_class_map.get(protocol.lower())
    if ConnectionClass:
        check = ConnectionClass(devices)
        async for status, msg in check:
            if status:
                print(f"{datetime.now()} {protocol}. Подключение успешно: {msg}")
            else:
                print(f"{datetime.now()} {protocol}. Не удалось подключиться: {msg}")
    else:
        raise ValueError(f"Для протокола {protocol} нет соответствующего класса")


async def main():
    with open("devices_asyncssh.yaml") as f:
        devices_ssh = yaml.safe_load(f)
    with open("devices_asynctelnet.yaml") as f:
        devices_telnet = yaml.safe_load(f)
    ip_list = ["192.168.100.1", "8.8.8.8", "10.1.1.1"]
    await asyncio.gather(
        scan(devices_ssh, "SSH"), scan(devices_telnet, "Telnet"), scan(ip_list, "ICMP")
    )
```

(continues on next page)

(продолжение с предыдущей страницы)

```
if __name__ == "__main__":
    asyncio.run(main())
```

Дополнительные материалы

Документация

- Asynchronous Iterators
- Asynchronous Context Managers
- async for
- async with

4. Использование модуля asyncio

В этом разделе рассматриваются асинхронные версии привычного функционала Python:

- list/dict/set comprehensions
- декораторы для сопрограмм
- асинхронные генераторы
- asyncio subprocess

Примечание: Синхронная версия этих тем рассматривается в базовой книге (list comprehensions и subprocess) и в advanced книге (декораторы и генераторы).

И некоторые особенности работы:

- работа с синхронными модулями при использовании asyncio
- использование semaphore для ограничения количества параллельных подключений

list/dict/set comprehensions

В list/dict/set comprehensions можно использовать await и, при переборе асинхронного итератора - `async for`.

Пример использования list comprehensions в сопрограмме:

```
from pprint import pprint
import asyncio
import yaml
from scrapli import AsyncScrapli
from scrapli.exceptions import ScrapliException
```

(continues on next page)

(продолжение с предыдущей страницы)

```
async def send_show(device, command):
    print(f">>> connect to {device['host']}")
    try:
        async with AsyncScrapli(**device) as conn:
            result = await conn.send_command(command)
            return result.result
    except ScrapliException as error:
        print(error, device["host"])


async def send_command_to_devices(devices, commands):
    tasks = [asyncio.create_task(send_show(device, commands)) for device in devices]
    result = [await task for task in tasks]
    return result


if __name__ == "__main__":
    with open("devices_async.yaml") as f:
        devices = yaml.safe_load(f)
    result = asyncio.run(send_command_to_devices(devices, "sh clock"))
    pprint(result, width=120)
```

Тут с помощью обычного list comprehensions (без async/await) создается список задач:

```
tasks = [asyncio.create_task(send_show(device, commands)) for device in devices]
```

А в этом используется await чтобы получить результаты каждой задачи:

```
result = [await task for task in tasks]
```

Также в list comprehensions может использоваться `async for`, при переборе асинхронного итератора.

```
import asyncio
↪
↪                                                    from pprint import pprint
from datetime import datetime
from scrapli import AsyncScrapli
from scrapli.exceptions import ScrapliException
import yaml


class CheckConnection:
    def __init__(self, device_list):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        self.device_list = device_list
        self._current_device = 0

    async def _scan_device(self, device):
        ip = device["host"]
        try:
            async with AsyncScrapli(**device) as conn:
                prompt = await conn.get_prompt()
            return True, prompt
        except (ScrapliException, asyncio.exceptions.TimeoutError) as error:
            return False, f"{error} {ip}"

    async def __anext__(self):
        if self._current_device >= len(self.device_list):
            raise StopAsyncIteration
        device_params = self.device_list[self._current_device]
        scan_results = await self._scan_device(device_params)
        self._current_device += 1
        return scan_results

    def __aiter__(self):
        return self


async def ssh_scan(devices):
    check = CheckConnection(devices)
    results = [out async for out in check]
    return results


if __name__ == "__main__":
    with open("devices_asyncssh.yaml") as f:
        devices = yaml.safe_load(f)
    result = asyncio.run(ssh_scan(devices))
    pprint(result)
    for status, msg in result:
        if status:
            print(f"{datetime.now()} SSH. Подключение успешно: {msg}")
        else:
            print(f"{datetime.now()} SSH. Не удалось подключиться: {msg}")
```

В примере выше в list comp используется именно `async for` потому что выполняется перебор асинхронного итератора:

```
results = [out async for out in check]
```

Декораторы для сопрограмм

Декораторы для сопрограмм в целом создаются так же, как и декораторы для обычных функций. Основное отличие в том, что если декоратор должен подменять функцию (это делается в большинстве декораторов функций), надо чтобы декоратор подменял сопрограмму сопрограммой.

Пример декоратора timecode, который замеряет время выполнения сопрограммы:

```
import asyncio
from datetime import datetime
from functools import wraps
import yaml
from scrapli import AsyncScrapli
from scrapli.exceptions import ScrapliException


def timecode(function):
    @wraps(function)
    async def wrapper(*args, **kwargs):
        start_time = datetime.now()
        result = await function(*args, **kwargs)
        print(">>> Функция выполнялась:", datetime.now() - start_time)
        return result
    return wrapper


@timecode
async def send_show(device, command):
    try:
        async with AsyncScrapli(**device) as conn:
            result = await conn.send_command(command)
            await asyncio.sleep(2)
            return result.result
    except ScrapliException as error:
        print(error, device["host"])


if __name__ == "__main__":
    with open("devices_async.yaml") as f:
        devices = yaml.safe_load(f)
        r1 = devices[0]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    result = asyncio.run(send_show(r1, "sh clock"))
    print(result)
```

Для того чтобы замерить время выполнения send_show, надо дождаться выполнения сопрограммы - сделать await, а await можно писать только внутри сопрограммы, соответственно функция wrapper должна быть сопрограммой.

Аналогичный обычный декоратор:

```
def timecode(function):
    @wraps(function)
    def wrapper(*args, **kwargs):
        start_time = datetime.now()
        result = function(*args, **kwargs)
        print('>>> Функция выполнялась:', datetime.now() - start_time)
        return result
    return wrapper
```

Примечание: Измерение времени выполнения сопрограммы неоднозначное занятие, так как оно зависит не только от самой сопрограммы, но и от того что еще работает в цикле событий.

Встроенные декораторы

Многие встроенные декораторы работают с функциями/методами, которые являются сопрограммами: functools.wraps, classmethod, staticmethod.

Примечание: Сопрограммы можно декорировать и property, но по смыслу property обычно применяется к чему-то, что выполняется быстро, поэтому стоит, как минимум, подумать о том надо ли делать property метод, который выполняет операции ввода-вывода.

Пример использования classmethod:

```
from pprint import pprint
import asyncio
import asyncssh


class ConnectAsyncSSH:

    @classmethod
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    async def connect(cls, host, username, password, enable_password, connection_
↪timeout=5):
        self = cls()

        self.host = host
        self.username = username
        self.password = password
        self.enable_password = enable_password
        self.connection_timeout = connection_timeout
        self._ssh = await asyncio.wait_for(
            asyncssh.connect(
                host=self.host,
                username=self.username,
                password=self.password,
                encryption_algs="+aes128-cbc,aes256-cbc",
            ),
            timeout=self.connection_timeout,
        )
        self.writer, self.reader, _ = await self._ssh.open_session(
            term_type="Dumb", term_size=(200, 24)
        )
        await self.reader.readuntil(">")
        self.writer.write("enable\n")
        await self.reader.readuntil("Password")
        self.writer.write(f"{self.enable_password}\n")
        await self.reader.readuntil("#")
        self.writer.write("terminal length 0\n")
        await self.reader.readuntil("#")
        return self

    async def get_prompt(self):
        self.writer.write("\n")
        output = await self.reader.readuntil("#")
        return output
```

Пример использования staticmethod:

```
import asyncio

class PingIP:
    def __init__(self, ip_list):
        self.ip_list = ip_list

    @staticmethod
    async def _ping(ip):
        reply = await asyncio.create_subprocess_shell(
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            f"ping -c 3 -n {ip}",
            stdout=asyncio.subprocess.PIPE,
            stderr=asyncio.subprocess.PIPE,
        )
        stdout, stderr = await reply.communicate()
        ip_is_reachable = reply.returncode == 0
        return ip, ip_is_reachable

    async def scan(self):
        ping_ok = []
        ping_not_ok = []
        coroutines = [self._ping(ip) for ip in self.ip_list]
        result = await asyncio.gather(*coroutines)
        for ip, status in result:
            if status:
                ping_ok.append(ip)
            else:
                ping_not_ok.append(ip)
        return ping_ok, ping_not_ok


if __name__ == "__main__":
    ip_list = ["192.168.100.1", "192.168.100.2", "192.168.100.3", "192.168.100.11"]
    scanner = PingIP(ip_list)
    results = asyncio.run(scanner.scan())
    print(results)
```

Асинхронные генераторы

```
import csv
import asyncio
import aiofiles


async def open_csv(filename):
    async with aiofiles.open(filename) as f:
        headers = await f.readline()
        headers = list(csv.reader([headers]))[0]
        async for line in f:
            print("open_csv line")
            yield dict(list(csv.DictReader([line], fieldnames=headers))[0])
```

(continues on next page)

(продолжение с предыдущей страницы)

```
async def filter_prefix_next_hop(async_iterable, nexthop):
    async for line in async_iterable:
        if line["nexthop"] == nexthop:
            yield line


async def parse_data(filename):
    data = open_csv(filename)
    nhop_45 = filter_prefix_next_hop(data, "200.219.145.45")
    async for line in nhop_45:
        print("parse_data line")
        print(line)


async def main():
    result = await parse_data("rib.table.lg.ba.ptt.br-BGP.csv")


if __name__ == "__main__":
    asyncio.run(main())
```

asyncio subprocess

```
import asyncio


async def ping(ip):
    reply = await asyncio.create_subprocess_shell(
        f"ping -c 3 -n {ip}",
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE,
    )

    stdout, stderr = await reply.communicate()

    ip_is_reachable = reply.returncode == 0
    return ip_is_reachable


async def ping_ip_list(ip_list):
    coroutines = [ping(ip) for ip in ip_list]
    result = await asyncio.gather(*coroutines)
    return result
```

(continues on next page)

(продолжение с предыдущей страницы)

```
if __name__ == "__main__":
    ip_list = ["192.168.100.1", "192.168.100.2", "192.168.100.3", "192.168.100.11"]
    results = asyncio.run(ping_ip_list(ip_list))
    print(results)
```

Запуск синхронного кода в потоках

Сопрограмма `asyncio.to_thread` позволяет запустить блокирующую операцию в потоке.

Эта сопрограмма появилась в Python 3.9, до этого использовалась `loop.run_in_executor`. При этом to_thread это по сути обертка вокруг loop.run_in_executor с использованием ThreadPoolExecutor по умолчанию, с максимальным количеством потоков по умолчанию:

```
async def to_thread(func, /, *args, **kwargs):
    """Asynchronously run function *func* in a separate thread.
    Any *args and **kwargs supplied for this function are directly passed
    to *func*. Also, the current :class:`contextvars.Context` is propogated,
    allowing context variables from the main thread to be accessed in the
    separate thread.
    Return a coroutine that can be awaited to get the eventual result of *func*.
    """
    loop = events.get_running_loop()
    ctx = contextvars.copy_context()
    func_call = functools.partial(ctx.run, func, *args, **kwargs)
    return await loop.run_in_executor(None, func_call)
```

Примечание: Начиная с версии Python 3.8 значение по умолчанию для max_workers высчитывается так `min(32, os.cpu_count() + 4)`.

Semaphore

```
import asyncio


async def connect(ip, semaphore):
    async with semaphore:
        print(f"Подключаюсь к {ip}")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        await asyncio.sleep(1)
        print(f"Ответ от {ip}")


async def main():
    sem = asyncio.Semaphore(20)
    coroutines = [connect(i, sem) for i in range(500)]
    await asyncio.gather(*coroutines)


asyncio.run(main())
```

```
from pprint import pprint
import asyncio

import yaml
from scrapli import AsyncScrapli
from scrapli.exceptions import ScrapliException


async def send_show(device, show_commands):
    print(f'>>> Подключаюсь к {device["host"]}')
    cmd_dict = {}
    if type(show_commands) == str:
        show_commands = [show_commands]
    try:
        async with AsyncScrapli(**device) as ssh:
            for cmd in show_commands:
                reply = await ssh.send_command(cmd)
                cmd_dict[cmd] = reply.result
            print(f'<<< Получен результат от {device["host"]}')
        return cmd_dict
    except ScrapliException as error:
        print(error, device["host"])


async def connect_ssh_with_semaphore(semaphore, function, *args, **kwargs):
    async with semaphore:
        return await function(*args, **kwargs)


async def send_command_to_devices(devices, commands, max_workers=50):
    semaphore = asyncio.Semaphore(max_workers)
    coroutines = [
        connect_ssh_with_semaphore(semaphore, send_show, device, commands)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        for device in devices
    ]
    result = await asyncio.gather(*coroutines)
    return result


if __name__ == "__main__":
    with open("devices_async.yaml") as f:
        devices = yaml.safe_load(f)
    result = asyncio.run(send_command_to_devices(devices, "sh ip int br"))
    pprint(result, width=120)
```

5. Работа с циклом событий

Цикл событий (Event loop)

Функция asyncio.get_event_loop возвращает цикл событий. Если цикл событий еще не был создан, создает новый и возвращает его:

```
loop = asyncio.get_event_loop()
```

Отмена задач

6. Дополнительная информация

В этом разделе собрана информация, которая не вошла в основные разделы книги, но которая, тем не менее, может быть полезна.

Тестовый HTTP сервер с FastAPI

netdev

netdev - это модуль, который позволяет упростить использование asyncssh для сетевых устройств. Netdev использует asyncssh, но при этом создает интерфейс и методы, которые нужны для работы с сетевым оборудованием. Интерфейс работы похож на netmiko.

Основы netdev

netdev - это модуль, который позволяет упростить использование asyncssh для сетевых устройств. Netdev использует asyncssh, но при этом создает интерфейс и методы, которые нужны для работы с сетевым оборудованием. Интерфейс работы похож на netmiko.

Установка netdev:

```
pip install netdev
```

Примечание: Текущая версия netdev 0.9.3 не поддерживает последние версии asyncssh.

Поддерживаемые платформы

Поддерживаемые платформы:

- 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

Параметры подключения

Основные параметры подключения:

- host - IP-адрес или имя хоста
- username - имя пользователя
- password - пароль
- secret (для Cisco IOS) - пароль на режим enable
- device_type - тип устройства

Процесс подключения немного отличается в зависимости от того используется асинхронный менеджер контекста или нет. При подключении без менеджера контекста, сначала надо передать параметры `netdev.create`, а затем вызвать метод `connect`:

```
In [1]: import netdev

In [2]: r1 = {
   ...:     'device_type': 'cisco_ios',
   ...:     'host': '192.168.100.1',
   ...:     'username': 'cisco',
   ...:     'password': 'cisco',
   ...:     'secret': 'cisco',
   ...: }

In [3]: ssh = netdev.create(**r1)

In [4]: await ssh.connect()
```

После этого можно отправлять команды:

```
In [5]: ssh.base_prompt
Out[5]: 'R1'

In [6]: await ssh.check_enable_mode()
Out[6]: True

In [7]: await ssh.disconnect()
```

При использовании асинхронного менеджера контекста, open вызывать не надо:

```
In [9]: async with netdev.create(**r1) as ssh:
   ...:     print(await ssh.check_enable_mode())
True
```

Отправка команд

В netdev есть два основных метода для отправки команд:

- send_command - отправить одну show команду
- send_config_set - отправить список команд в конфигурационном режиме (если команда одна, надо отправлять список с одной строкой)

Метод send_command

Метод send_command позволяет отправить одну команду на устройство.

```
In [10]: await ssh.send_command("sh ip int br")
Out[10]: 'Interface                  IP-Address      OK? Method Status
↪Protocol\nEthernet0/0               192.168.100.1   YES NVRAM  up
↪up      \nEthernet0/1               192.168.200.1   YES NVRAM  up
↪up      \nEthernet0/2               unassigned      YES NVRAM  up
↪up      \nEthernet0/3               192.168.130.1   YES NVRAM  up
↪up      \nLoopback11                11.1.1.1        YES manual up
↪up      \nLoopback99                unassigned      YES unset  up
↪up      \nLoopback100               10.1.1.100      YES manual up
↪up      \nLoopback200               10.2.2.2        YES manual up
↪up      \nTunnel0                   10.255.1.1      YES manual up
↪down    \nTunnel1                   unassigned      YES unset  up
↪down    \nTunnel9                   unassigned      YES unset  up
↪down    '
```

Параметры команды:

```
ssh.send_command(
    command_string,
    pattern='',
    re_flags=0,
    strip_command=True,
    strip_prompt=True,
)
```

Параметры strip_command и strip_prompt работают так же как в netmiko и при значении False (по умолчанию True), добавляют в вывод команду и приглашение:

```
In [11]: await ssh.send_command("sh clock")
Out[11]: '*05:54:20.671 UTC Thu Apr 8 2021'

In [12]: await ssh.send_command("sh clock", strip_command=False, strip_prompt=False)
Out[12]: 'sh clock\n*05:54:31.886 UTC Thu Apr 8 2021\nR1#'
```

Параметр pattern позволяет указывать какой строки ждать в выводе (нужно для команд, которые запрашивают подтверждение или ввод информации):

```
In [13]: await ssh.send_command("copy run start", pattern="Destination filename")
Out[13]: 'Destination filename [startup-config]? '

In [14]: await ssh.send_command("\n")
Out[14]: 'Building configuration...\n[OK]'
```

Метод send_config_set

Метод send_config_set позволяет отправить несколько команд конфигурационного режима.

Пример использования:

```
In [16]: await ssh.send_config_set(["interface lo99", "ip address 10.99.9.9 255.25.255.255
↪"])
Out[16]: 'conf t\nEnter configuration commands, one per line.  End with CNTL/Z.\
↪nR1(config)#interface lo99\nR1(config-if)#ip address 10.99.9.9 255.25.255.255\nBad mask␣
↪0xFF19FFFF for address 10.99.9.9\nR1(config-if)#end\nR1#'
```

Для отправки одной команды, ее надо передать как список с одной строкой:

```
In [17]: await ssh.send_config_set(["interface lo99"])
Out[17]: 'conf t\nEnter configuration commands, one per line.  End with CNTL/Z.\
↪nR1(config)#interface lo99\nR1(config-if)#end\nR1#'
```

Пример базового использования netdev

```
import asyncio
import netdev


async def send_show(device, command):
    async with netdev.create(**device) as ssh:
        result = await ssh.send_command(command)
        return result


if __name__ == "__main__":
    r1 = {
        "device_type": "cisco_ios",
        "host": "192.168.100.1",
        "username": "cisco",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        "password": "cisco",
        "secret": "cisco",
    }
    output = asyncio.run(send_show(r1, "show ip int br"))
    print(output)
```

Подключение к нескольким устройствам

```
from pprint import pprint
import asyncio

import netdev
import yaml


async def send_show(device, commands):
    result = {}
    if type(commands) == str:
        commands = [commands]
    try:
        async with netdev.create(**device) as ssh:
            for cmd in commands:
                output = await ssh.send_command(cmd)
                result[cmd] = output
            return result
    except netdev.exceptions.TimeoutError as error:
        print(error)
    except netdev.exceptions.DisconnectError as error:
        print(error)


async def send_command_to_devices(devices, commands):
    coroutines = [send_show(device, commands) for device in devices]
    result = await asyncio.gather(*coroutines)
    return result


if __name__ == "__main__":
    with open("devices_netdev.yaml") as f:
        devices = yaml.safe_load(f)
    result = asyncio.run(send_command_to_devices(devices, "sh ip int br"))
    pprint(result, width=120)
```

Дополнительные материалы

Документация:

- netdev

Примеры кода:

- netdev

7

Скачать PDF/Epub

- Epub
- PDF