Как работать с сетью с помощью сокетов?

Допустим, мы делаем поисковик.

Задача любого поисковика — при вводе текста запроса выдать какие-то релевантные сайты, чтобы поисковик построил индекс.

Поисковик должен обкачать эти сайты, понять, что за текст там написан, построить обратный индекс и дальше по словам пользователя возвращать сайты, которые удовлетворяют поиску.

Алгоритм работы веб-краулера, который занимается обкачкой сайтов, достаточно простой. У него есть список url-адресов, которые необходимо скачать первоначально. Мы берем эти сайты и начинаем с их корня. Получаем HTML-странички этих сайтов и *анализируем* каждую.

Наша задача — вытащить текст с каждой страницы и найти связанный url с каждой страницы. Т.е. мы парсим теги <a>, содержащие ссылки на другие ресурсы. Тем самым мы обогащаем список url-адресов.

После этого повторяем эти же действия. Идем по большому количеству адресов, и так мы можем спуститься до бесконечности.

Точный алгоритм чуть сложнее и больше, но поверхностно он выглядит примерно так:

1. Первоначальный список URL, которые необходимо скачать
2. Скачивание страниц из списка URL
3. Анализ каждой страницы
4. Поиск связанных URL
5. Повторить с п. 2

Пример. Здесь достаточно простой низкоуровневый код, чтобы потом нам было чуть проще рассмотреть ввод-вывод. Здесь код подключается к какому-то ресурсу и выполняет HTTP GET-запрос:

```
import socket


def fetch(url: str):
  sock = socket.socket() # создаем сокет, по умолчанию это TCP-сокет
  sock.connect(('xkcd.com', 80)) # подключаемся по TCP к xkcd.com на 80 порт
  request = f'GET {url} HTTP/1.0\r\nHost: xkcd.com\r\n\r\n' # формируем HTTP-запрос
  sock.send(request.encode('ascii')) # записываем запрос в сокет, он начинает отправляться
  response = b''
  chunk = sock.recv(4096) # читаем данные из сокета
  while chunk: # возможно, не все данные были считаны за один раз, читаем пока можем
    response += chunk
    chunk = sock.recv(4096)
  links = parse_links(response) # вытаскиваем ссылки из страницы (ненастоящая функция)
  q.update(links) # обогащаем наш список ссылок
```

(*этот пример доступен в examples/sockets/blocking.py*)

Разберем, что происходит в примере выше:

1. Создается сущность socket. Это пара IP-порт, которая подключена к другой паре IP-порт, и между ними создан какой-то канал, по которому в обе стороны бегают данные

2. Происходит системный вызов connect()

Вот так он выглядит в CPython. Мы уже видели его, когда говорили о IO-Bound операциях и тредах:

```
static int
internal_connect(PySocketSockObject *s, struct sockaddr *addr, int addrlen,
                 int raise)
{
    int res, err, wait_connect;

    Py_BEGIN_ALLOW_THREADS
    res = connect(s->sock_fd, addr, addrlen);
    Py_END_ALLOW_THREADS

    if (!res) {
        /* connect() succeeded, the socket is connected now */
        return 0;
    }
```

3. Составляем текст нашего запроса. Помним, что HTTP — это текстовый протокол.

4. Делаем системный вызов sock.send, куда отправляем наш запрос.

5. Последние строчки считывают ответ, т.е. http-сервер на домене xkcd.com получил наш запрос и отвечает тоже http-ответом. Мы собираем этот ответ с помощью системного вызова *chunk = sock.recv(4096)*, пока не получим все данные — то есть пока сокет не вернет пустой ответ.

Проблема такого кода: мы читаем ответ до тех пор, пока он не закончится. А это значит, что мы ничего не можем сделать в этот момент. Если мы хотим обкачать 1 000 000 страниц с помощью такого кода, то будем делать это последовательно и очень *долго*.

Решения

Треды и потоки. Это первый способ *решения* проблемы, про который мы уже говорили.

Все перечисленные операции — *connect*, send, receive — *блокирующие*.

connect будет блокировать выполнение *скрипта* до тех пор, пока не осуществит успешное или неуспешное подключение. Сервер может ему ответить connection refused, и тогда он отправит ошибку, чтоб подключиться не удалось.

Более яркий пример — *receive*. У нас есть данные на нашем компьютере, и мы отправляем их на удаленный сервер. Данные не перемещаются мгновенно, поэтому у нас есть время ожидания, пока данные находятся на нашей сетевой карте или на удаленной сетевой карте.

Это означает, что когда наш удаленный сервер отправил какие-то данные, мы сидим и ждем. Наши данные где-то *летят* — условно, через Америку, через океаны — и приходят через период времени. Пока данных нет, код заблокирован. Он ждет, пока они не появятся, чтобы их вычитать. А ведь вместо этого мы могли бы заняться чем-то полезным во время ожидания: распарсить другой сайт, сходить на другой домен.

Неблокирующий ввод-вывод. Наша цель — сделать так, чтобы мы не блокировались во время сетевой операции или любой операции ввода-вывода. Вводом-выводом считается чтение с диска, работа с сетью и многое другое.

Есть так называемая проблема C10k: как один сервер может одновременно работать с 10 000 соединений?

Чтобы один сервер мог эффективно обслуживать 10 000 соединений, используются *неблокирующий ввод-вывод* и асинхронное программирование.

Пример неблокирующего сокета:

```
sock = socket.socket()
sock.setblocking(False)
try:
  sock.connect(('xkcd.com', 80))
except BlockingIOError:
  pass
```

Неблокирующий ввод-вывод делается флагом sock.setblocking(False), т.е. мы просто говорим сокету не быть блокирующим. Но все не так просто.

Действие sock.setblocking(False) означает, что если мы попытаемся выполнить какое-то действие, которое приведет к блокировке, у нас выбросится исключение.

Например, мы пытаемся что-то прочитать из сокета, а данных на сетевой карте нет. Тогда метод socket.recv выкинет *исключение BlockingIOError*. В случае *блокирующего* сокета код заблокируется, а не выкинет исключение.

Плохой вариант неблокирующего сокета

```
import socket


def send_request():
    request = f'GET xkcd.com HTTP/1.0\r\nHost: xkcd.com\r\n\r\n'.encode('ascii')
    sock = socket.socket()  # создаем TCP сокет, файловая система выделяет файловый дескриптор
    sock.setblocking(False)
    try:
        sock.connect(('xkcd.com', 80))  # на этом этапе производится асинхронный TCP-handshake
    except BlockingIOError:
        # выкинется ошибка о том, что соединение еще устанавливается
        pass


    while True:
        try:
            sock.send(request)  # пишем данные в сокет
            break
        except BlockingIOError:
            # сокет может быть уже заполнен
            pass
        except OSError:
            # соединение в сокете еще не установлено - выполняется TCP-handshake
            pass

    data = b''
    while True:
        try:
            # читаем данные из сокета, если данных нет, то выкинется ошибка
            data += sock.recv(4096)
            break
        except BlockingIOError:
            # данные могут еще не прийти, сокет будет заблокирован
            pass


    print(data.decode('ascii')) # нам приходят байты и их нужно декодировать


send_request()
```

(*этот код можно найти в examples/sockets/non-blocking*)

У нас есть запрос, который мы пытаемся отправить в наш неблокирующий сокет.

Сокет на запись может быть недоступен. Например, буфер в нашей сетевой карте полностью заполнен. Или мы еще не успели установить соединение, не выполнился TCP-handshake. Идея в том, что сокет может быть недоступен на запись в данный момент времени так же, как может быть недоступен на чтение. Он заблокируется, о чем сообщит ОС, выкинув OSError или его наследника BlockingIOException.

Мы хотим уметь делать какие-то дополнительные действия в моменты, когда у нас потенциально произойдет блокировка. Т.е. в целом мы могли бы что-то делать внутри этого исключения.

Но отправить данные тоже нужно, поэтому мы все равно зависим от этого сокета. Мы будем на самом деле пытаться отправить данные до тех пор, пока у нас не получится. И в целом — используя такой наивный подход, мы не можем сделать ничего лучше того, что написано в примере.

Т.е. мы сделали сокет неблокирующим, и он не заблокирует код. Но вместо этого мы получим ошибку, которую никак не можем обработать, потому что данные отправить все равно нужно. Так что мы будем так и крутиться в цикле до тех пор, пока не отправим данные.

Проблема такого пути: хотя сокет неблокирующий, мы все равно должны ждать завершения цикла.

На самом деле стало еще хуже, чем было. Когда сокет был *блокирующий*, мы сделали sock.send, и операционная система взяла на себя обязанность отправить эти данные. Сами мы уснули: ждем сигнала от ОС, что все завершилось успешно.

Но в этом примере с неблокирующим сокетом ОС сразу отвечает: «Я сейчас заблокируюсь и ничего отправлять не буду». И мы начинаем крутиться while True, пока в итоге не окажется, что наш процесс потребляет 100% CPU. Он будет просто греть ядро, потому что наш код — бесконечный цикл, который пытается выполнить какую-то операцию, которая сразу завершается с ошибкой.

Мы можем сделать множество сокетов на кучу разных доменов и пытаться поочередно сделать разные send. Они будут давать исключения, а мы будем как-то их ретрашть. Можно ретрашть их все одновременно: пройтись по ним всем циклом, сделать send. Увидим, что у кого-то получилось, а у кого-то нет, то есть надо делать try-except над каждым из них.

Резюме: хотя сокет в *примере* неблокирующий, но *сам пример* оказывается блокирующим — он потребляет 100% CPU и бесконечно пытается послать данные или считать их из сокета.

Как же сделать правильно?
Ответ — в *следующей* карточке.

подробнее о том, как написать асинхронный краулер, вы можете прочесть в нашей статье на Хабре