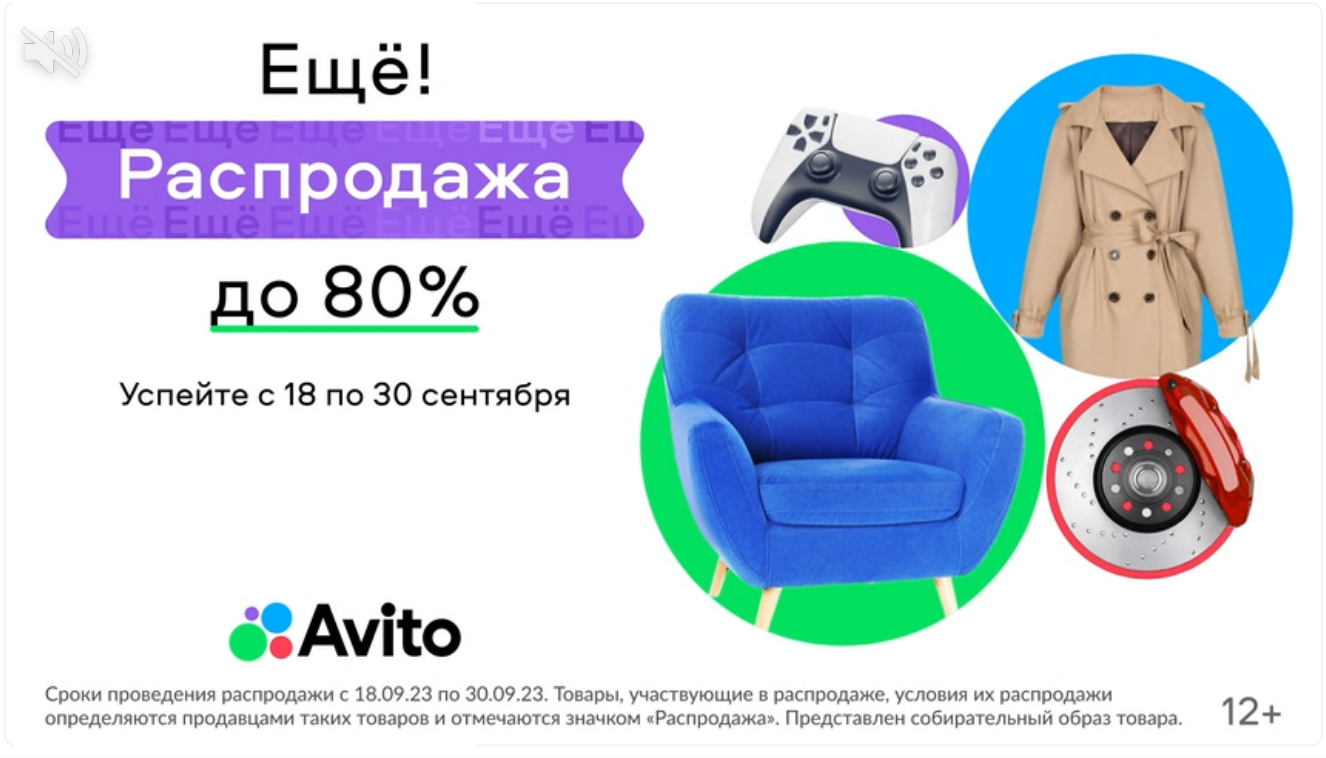
ХОЧУ ПОМОЧЬ ПРОЕКТУ

Сообщить об ошибке.

Модуль socketserver, встроенный фреймворк Python



avito.ru РЕКЛАМА • 16+

Больше информации на сайте рекламодателя

Подробнее

Фреймворк для создания сетевых серверов

Модуль socketserver упрощает задачу написания сетевых серверов.

Модуль определяет четыре основных класса конкретных серверов:

socketserver.TCPServer(server_address, RequestHandlerClass, bind_and_activate=True):

Класс socketserver.TCPServer() использует протокол Internet TCP, который обеспечивает непрерывные потоки данных между клиентом и сервером. Является подклассом socketserver.BaseServer()

Если аргумент bind_and_activate=True, то конструктор автоматически пытается вызвать методы BaseServer.server_bind() и BaseServer.server_activate().

Остальные параметры передаются базовому классу socketserver.BaseServer().

socketserver.UDPServer(server_address, RequestHandlerClass, bind_and_activate=True):

Класс socketserver.UDPServer() использует дейтаграммы, которые представляют собой дискретные пакеты информации, которые могут поступать не по порядку или быть потеряны во время передачи.

Класс использует такие же параметры, как у socketserver.TCPServer().

socketserver.UnixStreamServer(server_address, RequestHandlerClass, bind_and_activate=True)
socketserver.UnixDatagramServer(server_address, RequestHandlerClass, bind_and_activate=True):

Классы socketserver.UnixStreamServer и socketserver.UnixDatagramServer используются менее часто и похожи на классы TCP и UDP, но используют сокеты домена Unix. Они недоступны на платформах, отличных от Unix. Параметры такие же, как у socketserver.TCPServer().

Эти четыре класса обрабатывают запросы синхронно. Каждый запрос должен быть завершен до того, как можно будет начать следующий запрос. Если сервер обрабатывает большой объем информации или возвращает много данных, то такое поведение не подходит, т.к. для выполнения каждого запроса требуется много времени. Решение состоит в том, чтобы создать отдельный процесс или поток для обработки каждого запроса. В этом случае **для поддержки асинхронного поведения** могут использоваться классы socketserver.ForkingMixIn() и socketserver.ThreadingMixIn().

Шаги для создания сервера.

1. Во-первых, необходимо создать класс обработчика запросов, создав подкласс класса socketserver.BaseRequestHandler() и переопределив его метод .handle(). Этот метод будет обрабатывать входящие запросы.
2. Во-вторых, нужно создать экземпляр одного из классов сервера, передав ему адрес сервера и класс обработчика запросов. Рекомендуется использовать сервер совместно с оператором контекстного менеджера with.
3. Затем вызываются методы объекта сервера Server.handle_request() или Server.serve_forever() для обработки одного или нескольких запросов.

Вверх

4. Наконец, если сервер не использовался с оператором with, то необходимо вызвать его метод Server.server_close(), чтобы закрыть сокет.

Для обработки каждого соединения в своем потоке, нужно наследоваться от класса socketserver.ThreadingMixIn() , при этом несРЕКЛАМАходимо явно объявлять поведение потоков при внезапном завершении работы. Класс socketserver.ThreadingMixIn() определяет атрибут .daemon_threads, который указывает, должен ли сервер ждать завершения потока. Значение False (по умолчанию) означает, что Python не завершит работу, пока не завершатся все потоки.

Классы серверов имеют одинаковые внешние методы и атрибуты, независимо от того, какой сетевой протокол они используют.

Примечания по созданию сервера.

В диаграмме наследования есть пять классов, четыре из которых представляют синхронные серверы четырех типов:

```
+------------+
| BaseServer |
+------------+
      |
      v
+-----------+        +------------------+
| TCPServer |------->| UnixStreamServer |
+-----------+        +------------------+
      |
      v
+-----------+        +--------------------+
| UDPServer |------->| UnixDatagramServer |
+-----------+        +--------------------+
```

Обратите внимание, что UnixDatagramServer() является производным от UDPServer(), а не от UnixStreamServer - единственное различие между *TCP* и *UnixStream* сервером - это семейство адресов, которое повторяется в обоих классах Unix серверов.

Советы по созданию сервера.

При создании серверов необходимо работать головой!

Например, нет смысла использовать сервер, обрабатывающий запросы клиентов в отдельных процессах (ForkingMixIn), если служба содержит какой-то общий ресурс в оперативной памяти и может быть изменен различными запросами от клиентов, поскольку изменения в дочернем процессе никогда не достигнут начального состояния, хранящегося в родительском процессе и передаваемого каждому дочернему процессу. В этом случае можно использовать сервер, обрабатывающий запросы клиентов в потоках (ThreadingMixIn), но в случае общего ресурса придется использовать блокировки для защиты целостности общих данных.

Если создается HTTP-сервер, на который все данные приходят извне (с файловой системы или прокси), синхронный класс TCPServer или UDPServer, по сути, сделает службу "глухой", пока обрабатывается один запрос - что может быть ОЧЕНЬ долго. Здесь необходимо использовать сервер, который обрабатывает запросы клиентов в отдельных процессах или потоках.

В некоторых случаях может быть целесообразно начать обрабатывать запрос синхронно и в зависимости от данных запроса, продолжить обработку в отдельном дочернем процессе. Такое поведение можно реализовать, используя синхронный сервер и выполняя явный fork в методе класса обработчика запросов .handle().

Другой подход к асинхронной обработке нескольких запросов в среде, которая не поддерживает потоки или функцию os.fork() (или где они слишком дороги или не подходят для вашей службы), заключается в использовании модуля selectors, который позволяет решать, какой запрос следует обрабатывать дальше. Это особенно важно для *stream* сервисов, где каждый клиент потенциально может быть подключен в течение длительного времени.

Пример синхронного TCP сервера с использованием класса TCPServer.

Это серверная сторона:

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    Класс обработчика запросов для сервера.
    Создается один раз при каждом подключении к серверу и должен
    определить метод `handle()` для реализации связи с клиентом.
    """
```

Вверх

```
    def handle(self):
        # self.request - это TCP-сокет, подключенный к клиенту
        self.data = self.request.recv(1024).strip()
        print(f"{self.client_address[0]} отправил:")
        data = self.data.decode('utf-8')
        print(data)
        # отправим те же данные обратно, но в верхнем регистре
        self.request.sendall(data.upper().encode('utf-8'))


if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    # Создаем сервер, привязанный к `localhost` на порту 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # Активируем сервер. Работа будет продолжаться до тех пор,
        # пока не прервать программу при помощи Ctrl-C
        server.serve_forever()
```

РЕКЛАМА

Клиентская сторона для проверки работы синхронного TCP сервера. Первым запускается серверная сторона. Клиент запускается в другом терминале. Сообщение клиенту формируется как параметр командной строки при запуске клиентского скрипта, например $ python3 client.py Hello world!

```
import socket, sys

HOST, PORT = "localhost", 9999

# Данные для отправки получим
# как параметры командной строки
data = " ".join(sys.argv[1:])

# Создаем сокет (`SOCK_STREAM` означает сокет TCP)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Подключение к серверу и отправка данных
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data + "\n", "utf-8"))

    # Получаем данные с сервера и завершаем работу
    received = str(sock.recv(1024), "utf-8")

print(f"Отправлено: {data}")
print(f"Получено: {received}")
```

Вывод примера должен выглядеть примерно так:

Сервер:

```
$ python TCPServer.py
127.0.0.1 прислал:
b'hello world with TCP'
127.0.0.1 прислал:
b'python is nice'
```

Клиент:

```
$ python TCPClient.py hello world with TCP
Отправлено: hello world with TCP
Получено: HELLO WORLD WITH TCP

$ python TCPClient.py python is nice
Отправлено: python is nice
Получено: PYTHON IS NICE
```

Вверх

РЕКЛАМА


Содержание раздела:

DOCS-Python.ru™, 2023 г. (Внимание! При копировании материала ссылка на источник обязательна) @docs_python_ru

Вверх