Урок 7

Модуль select, слоты

Особенности использования модуля select. Слоты, их назначение и применение.

На этом занятии продолжим изучать работу в сети: рассмотрим модуль **select** и обслуживание нескольких подключений. Дополнительно узнаем об использовании слотов (**__slots__**).

Особенности использования модуля select

Предыстория создания модуля select

В Python реализована группа модулей, обеспечивающих доступ к сетевым протоколам и форматам файлов. Эти модули зависят друг от друга. Один из них — **socket** — мы рассмотрели на третьем и пятом уроках.

Модуль **socket** реализует возможности объектно-ориентированного интерфейса для доступа к функциям сетевых библиотек операционной системы. При этом предусмотрена поддержка всех видов сокетов и всех сетевых протоколов: **TCP/IP** (в том числе **IPv6**), **UNIX Domain**, **IPX/SPX** и других.

У сетевого программиста есть два варианта взаимодействия с сокетом: синхронное и асинхронное. При первом поступает запрос от клиента -> открывается сокет -> передаются данные -> сокет закрывается. Пока не завершится локальный диалог с одним клиентом, не может быть запущено взаимодействие с новым.

Такой принцип заложен в основу простых серверов, не рассчитанных на работу с сотнями и тысячами клиентов. Если нагрузка возрастает некритично, можно создать еще один или нескольких потоков (или процессов) и выполнять обработку подключений и в них. Это стабильный и проверенный подход, реализованный, например, в сервере Apache. Данные, поступающие от клиентов, обрабатываются в порядке очередности. Запуск программного кода, реализующего сложные вычисления или операции с базой данных, на работу других клиентов не влияет.

Но сервер не может бесконечно генерировать процессы и потоки, так как при каждой такой операции тратятся существенные ресурсы. Поэтому внедрили асинхронный подход и реализацию системных вызовов для неблокирующего ввода-вывода. Множество сокетов и потоков, потребляющие значительные ресурсы, заменяются на прослушивание данных от многих клиентов на одном сокете.

Варианты реализации высоконагруженных проектов

В третьем уроке этого курса есть пример использования функции **accept()**, которая выполняется на серверном сокете и блокирует работу приложения до того, как будет получено сообщение или данные от клиента. Применение этой функции оправдано на простейших демонстрационных примерах. Реальные серверы с нагрузкой в несколько тысяч клиентов требуют других подходов, обеспечивающих нормальное функционирование сервера при многотысячной нагрузке.

Подходы, которые применяются на практике:

1. **Отдельный поток на каждого клиента**. Работоспособный вариант, но потребляет системные ресурсы. Поэтому не очень эффективен при решении сложных задач.

2. **Неблокирующие сокеты.** В Python 3 для их реализации предусмотрен метод **setblocking()**, в который передается параметр, равный 0. Пример неблокирующего сокета:

```
from socket import *
nbs = socket(AF_INET, SOCK_STREAM)
client_list = []
nc = 2
for j in range(nc):
   (client,ap) = nbs.accept()
    client.setblocking(0)
    client_list.append(client)
```

Главный недостаток этого подхода — в необходимости «вручную» выполнять проверку готовности каждого клиента.

3. **Применение select/poll**.

Введение в системный вызов select

Один из вариантов системных вызовов для реализации неблокирующего взаимодействия с вводом-выводом — **select()**. Логика работы этой команды в том, что данный системный вызов принимает фиксированный объем описанных в программе файловых дескрипторов (не более 1024 по умолчанию) и прослушивает события на них. Когда дескриптор готов к чтению или записи данных в коде, запускается соответствующий колбэк-метод. Системный вызов **select()** поддерживается всеми программными платформами, подразумевающими сетевое взаимодействие.

Модуль select в Python

Модуль **select** в Python позволяет использовать системные вызовы **select()** и **poll()**.

Системный вызов **select()** в данном случае можно использовать для опроса — или мультиплексирования — обработки нескольких потоков ввода-вывода, не используя потоки управления или дочерние процессы. В системах UNIX эти вызовы можно применять для работы с сокетами, каналами и многими другими типами файлов. В Windows — только для работы с сокетами.

- **select(iwtd, owtd, ewtd [, timeout])** — запрашивает информацию о готовности к вводу, выводу и о наличии исключений для группы дескрипторов файлов. В первых трех аргументах передаются списки с целочисленными дескрипторами файлов или с объектами, обладающими методом **fileno()**, который возвращает дескриптор файла. Аргумент **iwtd** определяет список объектов, которые проверяются на готовность к вводу; **owtd** — список объектов, которые проверяются на готовность к выводу; **ewtd** — список объектов, которые проверяются на наличие исключительных ситуаций. В любом из аргументов можно передавать пустой список. В аргументе **timeout** передается число с плавающей точкой, определяющее предельное время ожидания в секундах. При вызове без **timeout** функция ожидает, пока хотя бы один из дескрипторов окажется в требуемом состоянии. Если в этом аргументе передать число 0, функция просто выполнит опрос и тут же вернет управление. Системный вызов **select()** возвращает кортеж списков с объектами, находящимися в требуемом состоянии. Эти списки включают подмножества объектов в первых трех аргументах. Если по истечении предельного времени ожидания ни один из дескрипторов не приходит в требуемое состояние, возвращается три пустых списка. В случае ошибки вызывается исключение **select.error**. В качестве его значения возвращается та же информация, что и в исключениях **IOError** и **OSError**.

- **poll()** — создает объект, выполняющий опрос с помощью системного вызова **poll()**. Эта функция доступна только в системах, поддерживающих его.

Системный вызов **poll()** был разработан, чтобы реализовывать высоконагруженные сетевые приложения в различных ОС. В Linux он используется давно, а в Windows — с выпуска Windows Vista. Данный вызов вместо несвязанного набора сокетов принимает на вход структуру со списком дескрипторов и связанных с ними событий. После этого обходит структуру в цикле и «отлавливает» события.

Реализация системного вызова **poll()** расширила возможности метода **select()**, но и эта команда имеет недостатки. Главный — линейность обхода структуры с дескрипторами с точки зрения алгоритмики. Это касается и отслеживания событий, и реакции на них, и передачи данных.

Тем не менее системный вызов **select()** остается основным и проверенным способом обеспечить сетевое взаимодействие при значительных нагрузках на сервер. Благодаря этой команде не надо вручную проверять готовность каждого клиента. **select()** перекладывает проверку непосредственно на операционную систему. Так же эффективно можно применять более позднюю реализацию системного вызова **select()** — команду **poll()**.

Логика работы системного вызова select

Классическая задача с применением системного вызова **select** — написать серверную часть приложения. Например, при разработке чата. При использовании традиционного подхода (без функции **select**) взаимодействие сервера с клиентами строится по следующему алгоритму:

1. Существует массив дескрипторов клиентских сокетов.
2. При сетевом взаимодействии в бесконечном цикле массив обходят и «прослушивают» каждый сокет на наличие запросов.
3. Если передаваемых данных нет, переходят к следующему сокету.

В этом случае 90% времени выполнения цикла будет затрачено впустую, поскольку данные от клиентов могут не поступать. Процессорное время уходит впустую, а им мог бы воспользоваться другой процесс, решающий важную задачу.

Массив дескрипторов клиентских сокетов можно передавать в качестве аргумента функции **select**, а она в свою очередь предоставляет список сокетов, которые:

- Готовы принять новые данные;
- Имеют новые данные для чтения;
- Содержат ошибки выполнения.

Конструкция **select** — это даже не функция, а системный вызов. Она заложена не в сам Python, а в операционную систему. Таким образом, традиционный подход с простым перебором заменяется на более эффективный, с особым оптимизированным алгоритмом.

Примеры использования модуля select

Рассмотрим пример эхо-сервера, обслуживающего одновременно несколько клиентов. Реализован с использованием функции **select** (файл **examples/01_select/01_time_server_select.py**):

```
import time
import select
from socket import socket, AF_INET, SOCK_STREAM

def new_listen_socket(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(5)
    sock.settimeout(0.2)    # Таймаут для операций с сокетом
                            # Таймаут необходим, чтобы не ждать появления данных
в сокете
    return sock


def mainloop():
    ''' Основной цикл обработки запросов клиентов
    '''
    address = ('', 8888)
    clients = []
    sock = new_listen_socket(address)

    while True:
        try:
            conn, addr = sock.accept()  # Проверка подключений
        except OSError as e:
            pass                        # timeout вышел
        else:
            print("Получен запрос на соединение с %s" % str(addr))
            clients.append(conn)
        finally:
            # Проверить наличие событий ввода-вывода без таймаута
            w = []
            try:
                r, w, e = select.select([], clients, [], 0)
            except Exception as e:
            # Исключение произойдет, если какой-то клиент отключится
                pass        # Ничего не делать, если какой-то клиент отключился

            # Обойти список клиентов, читающих из сокета
            for s_client in w:
                timestr = time.ctime(time.time()) + "\n"
            try:
                    s_client.send(timestr.encode('utf-8'))
            except:
                # Удаляем клиента, который отключился
                clients.remove(s_client)

print('Эхо-сервер запущен!')
mainloop()
```

Клиентское приложение теперь находится в постоянном опросе сервера (файл **examples/01_select/02_time_client_random.py**):

```
from socket import *

s = socket(AF_INET, SOCK_STREAM)     # Создать сокет TCP
s.connect(('localhost', 8888))       # Соединиться с сервером

while True:                          # Постоянный опрос сервера
    tm = s.recv(1024)
    print("Текущее время: %s" % tm.decode('utf-8'))

s.close()
```

Такой скрипт позволяет поэкспериментировать, чтобы увидеть, как сервер будет обрабатывать клиентские подключения. Создадим дополнительный служебный скрипт, запускающий несколько «клиентов» с использованием модуля **subprocess** (файл **examples/01_select/03_start_clients.py**)

```
from subprocess import Popen, CREATE_NEW_CONSOLE

p_list = []         # Список клиентских процессов

while True:
    user = input("Запустить 10 клиентов (s) / Закрыть клиентов (x) / Выйти (q)
")

    if user == 'q':
        break
    elif user == 's':
        for _ in range(10):
            # Флаг CREATE_NEW_CONSOLE нужен для ОС Windows,
            # чтобы каждый процесс запускался в отдельном окне консоли
            p_list.append(Popen('python time_client_random.py',
                                creationflags=CREATE_NEW_CONSOLE))

        print(' Запущено 10 клиентов')
    elif user == 'x':
        for p in p_list:
            p.kill()
        p_list.clear()
```

- Модуль **subprocess** изучим на занятии, посвященном процессам и потокам.

Рассмотрим более сложный пример, где реализуется сетевое взаимодействие эхо-сервера и нескольких клиентов с использованием функции **select** (файл **examples/01_select/04_echo_server_select.py**):

```
# -------- Эхо-сервер, обрабатывающий "одновременно" несколько клиентов -------
#              Обработка клиентов осуществляется функцией select

import select
from socket import socket, AF_INET, SOCK_STREAM


def read_requests(r_clients, all_clients):
    """ Чтение запросов из списка клиентов
    """
    responses = {}  # Словарь ответов сервера вида {сокет: запрос}

    for sock in r_clients:
        try:
            data = sock.recv(1024).decode('utf-8')
            responses[sock] = data
        except:
            print('Клиент {} {} отключился'.format(sock.fileno(), sock.getpeername()))
            all_clients.remove(sock)

    return responses


def write_responses(requests, w_clients, all_clients):
    """ Эхо-ответ сервера клиентам, от которых были запросы
    """

    for sock in w_clients:
        if sock in requests:
            try:
                # Подготовить и отправить ответ сервера
                resp = requests[sock].encode('utf-8')
                # Эхо-ответ сделаем чуть непохожим на оригинал
                sock.send(resp.upper())
            except:  # Сокет недоступен, клиент отключился
                print('Клиент {} {} отключился'.format(sock.fileno(),
sock.getpeername()))
                sock.close()
                all_clients.remove(sock)


def mainloop():
    """ Основной цикл обработки запросов клиентов
    """
    address = ('', 10000)
    clients = []

    s = socket(AF_INET, SOCK_STREAM)
```

```
    s.bind(address)
    s.listen(5)
    s.settimeout(0.2)  # Таймаут для операций с сокетом
    while True:
        try:
            conn, addr = s.accept()  # Проверка подключений
        except OSError as e:
            pass  # timeout вышел
        else:
            print("Получен запрос на соединение от %s" % str(addr))
            clients.append(conn)
        finally:
            # Проверить наличие событий ввода-вывода
            wait = 10
            r = []
            w = []
            try:
                r, w, e = select.select(clients, clients, [], wait)
            except:
                pass  # Ничего не делать, если какой-то клиент отключился

            requests = read_requests(r, clients)  # Сохраним запросы клиентов
            if requests:
                write_responses(requests, w, clients)  # Выполним отправку ответов
клиентам


print('Эхо-сервер запущен!')
mainloop()
```

Пояснения к коду:

1. Функции **read_requests** и **write_responses** достаточно просты:
    a. **в read_requests** обходится список сокетов клиентов на чтение и формируется словарь запросов в формате **{сокет: запрос}**;
    b. в **write_responses** похожим образом обходится список сокетов на запись. Отсылаемые ответы сервера формируются на основании словаря запросов **requests**;
2. **mainloop** — основная функция сервера:
    a. открывает сокет на прослушивание и устанавливает таймаут для всех операций с ним;
    b. далее в цикле проверяет наличие новых подключений функцией **accept()** и добавляет новое подключение в список **clients**. При установленном таймауте **accept()** создаст исключение по его истечении, если подключение отсутствует;
    c. посредством функции **select()** проверяет возможность чтения/записи из/в сокеты (список **clients**); **select()** возвращает кортеж из трех списков: дескрипторы на чтение, на запись и имеющие исключительную ситуацию.

Код эхо-клиента, который отправляет «запрос» и сразу же читает ответ от сервера (файл **examples/01_select/05_echo_client_select.py**):

```
from socket import *
from select import select
import sys

ADDRESS = ('localhost', 10000)

def echo_client():
    # Начиная с Python 3.2 сокеты имеют протокол менеджера контекста
    # При выходе из оператора with сокет будет автоматически закрыт
    with socket(AF_INET, SOCK_STREAM) as sock: # Создать сокет TCP
        sock.connect(ADDRESS)    # Соединиться с сервером
        while True:
            msg = input('Ваше сообщение: ')
            if msg == 'exit':
                break
            sock.send(msg.encode('utf-8'))       # Отправить!
            data = sock.recv(1024).decode('utf-8')
            print('Ответ:', data)

if __name__ == '__main__':
      echo_client()
```

TCP-протокол, обеспечивающий сетевое взаимодействие, надежен и гарантирует доставку данных. При разработке клиент-серверных приложений, работающих через протокол **TCP**, его функции распределяются между программой и операционной системой. Сокеты, обеспечивающие непосредственный обмен данными, могут быть неблокирующими и блокирующими. Неблокирующие сервисы поддерживают сервер при высоких нагрузках. Альтернативой им выступают системные вызовы **select()** и **poll()**, функции которых в Python 3 предоставляет модуль **select**. Благодаря им могут работать высоконагруженные серверы с подключением множества клиентов.

Слоты, их назначение и применение

Введение в слоты

Понятие слотов в Python тесно связано с определением классов. Слоты относятся к специально зарезервированным методам, которые выделяются префиксом —двойным символом подчеркивания. Слоты — это список атрибутов, определяемый в заголовке класса с помощью ключевого слова **__slots__**. Чтобы использовать в таком классе какой-либо атрибут, его необходимо указать в списке слотов.

Простейший пример с конструкцией **__slots__**:

```
class resume(object):
    __slots__ = ['name', 'surname', 'address']

>>> r = resume()
>>> x.name = 'john'
```

Значением конструкции **__slots__** может быть не только список, а любой итерируемый объект, а также строка или набор строк.

Зачем нужны слоты

В языке Python каждый класс может иметь атрибуты, для хранения которых по умолчанию используется словарь. Словари в Python отличаются тем, что занимают значительную часть «оперативки». Интерпретатор Python не может выделить определенный объем памяти при создании экземпляра класса для хранения его атрибутов.

Поэтому большое количество экземпляров классов будут потреблять оперативную память. Использование **__slots__** позволяет обойти эту проблему. Данная конструкция указывает, чтобы Python не генерировал словари, а выделял ограниченный объем памяти, который необходим для хранения указанного числа атрибутов. Рассмотрим пример программного кода класса с использованием конструкции **__slots__** и без нее.

Без **__slots__**:

```
Class MyResume(object):
    def __init__(self, name, address):
        self.name = name
        self.address = address
        self.set_data()
    # ...
```

С применением **__slots__**:

```
Class MyResume(object):
    __slots__ = ['name', 'address']
    def __init__(self, name, address):
        self.name = name
        self.address = address
        self.set_data()
    # ...
```

Код из второго фрагмента уменьшает потребление оперативной памяти. Разработчики отмечают 40- и даже 50-процентную экономию ресурса благодаря этому решению.

Еще один пример использования слотов:

```
class Account(object):
    __slots__ = ('name', 'balance')
    ...

acc = Account()
acc.x = 13

# Traceback (most recent call last):
# File "<pyshell#12>", line 1, in <module>
# a.x = 13
# AttributeError: 'Account' object has no attribute 'x'
```

В данном случае исключение было сгенерировано из-за динамического назначения экземпляру класса **Account()** атрибута **x** со значением 13. При этом в списке **__slots__** такого атрибута нет, и возможность динамического назначения также не определена.

Если в классе определена переменная **__slots__**, его экземпляры смогут иметь атрибуты только с указанными именами. В противном случае будет вызываться исключение **AttributeError**. Это ограничение исключает добавление новых атрибутов к существующим экземплярам и не позволяет присваивать значения атрибутам, в именах которых допущена опечатка.

Переменная **__slots__** задумывалась совсем не как мера предосторожности. Фактически, это инструмент оптимизации объема занимаемой памяти и скорости выполнения. Экземпляры классов, где определена переменная **__slots__**, уже не используют словарь для хранения данных экземпляра. Вместо него используется более компактная структура данных, в основе которой лежит массив. Применение переменной **__slots__** в программах, создающих огромное число объектов, может существенно уменьшить объем потребляемой памяти и ускорить выполнение.

__slots__ по-особенному воздействует на наследование. Если она используется в базовом классе, производный также должен объявлять атрибут **__slots__** со списком имен своих атрибутов (даже если он не добавляет новых). Только так он сможет использовать преимущества этой переменной. Если этого не сделать, производный класс будет работать медленнее и занимать больше памяти, чем если бы переменной **__slots__** не было ни в одном из классов!

__slots__ может нарушить работоспособность программного кода, который ожидает, что экземпляры будут иметь атрибут **__dict__**. Хотя такая ситуация не характерна для прикладного программного кода, вспомогательные библиотеки и другие инструменты поддержки объектов могут использовать словарь **__dict__** для отладки, сериализации объектов и других операций.

Объявление переменной **__slots__** не требует переопределения в классе методов, **__getattribute__()**, **__getattr__()** и **__setattr__()**. По умолчанию они учитывают возможность ее наличия. Не надо добавлять имена методов и свойств в переменную **__slots__**, так как они хранятся не на уровне экземпляров, а на уровне класса.

Еще один пример, демонстрирующий сравнение подходов: без использования конструкции **__slots__** и с ней (файл **examples/02_slots/01_slots.py**).

Без **__slots__**:

```
class BasicClass(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

# Создадим объект нашего класса.
bc = BasicClass(13, 42)
# В обычной ситуации все атрибуты объекта хранятся во внутреннем словаре
__dict__:
print(bc.__dict__)

# Строка ниже не вызовет ошибку. Атрибут z будет добавлен в уже созданный
объект
bc.z = 777
print(bc.__dict__)
```

Результат:

```
{'x': 13, 'y': 42}
{'x': 13, 'z': 777, 'y': 42}
```

С **__slots__**:

```
class StrictClass():
    __slots__ = ('x', 'y')

    def __init__(self, x, y):
        self.x = x
        self.y = y

sc = StrictClass(13, 42)
sc2 = StrictClass(11, 44)

# У объекта такого класса не будет атрибута __dict__:
print(sc.__dict__)
```

Результат:

```
AttributeError: 'StrictClass' object has no attribute '__dict__'
```

Рекомендации при использовании слотов

1. Если класс наследуется от родителя, не имеющего конструкции **__slots__**, ее определение для потомка бесполезно, поскольку родителем будет создан словарь **(__dict__)**.
2. При попытке связать с классом атрибут, не перечисленный в **__slots__**, возникнет исключение **AttributeError**.
3. Если для экземпляра класса необходимо динамическое назначение атрибутов, следует установить данную опцию в перечислении слотов — добавить конструкцию **__dict__**.
4. Действие слотов распространяется только на тот класс, в котором они определены. Поэтому у наследников для атрибутов будет генерироваться словарь (**__dict__**). Либо необходимо отдельно определять слоты для каждого класса-потомка. Если для наследников также определены слоты, в перечислении необходимо указывать только дополнительные.

Специальная конструкция **__slots__** ориентирована на явное указание атрибутов, которые разработчик ожидает от экземпляра класса. Слоты реализуются на основе создания дескриптора (указателя) для каждого из атрибутов экземпляра класса. Данный подход позволит ускорить выполнение кода и сэкономить память.

На занятии мы рассмотрели, как использовать модуль **select**, чтобы обслуживать одновременно несколько клиентских взаимодействий. Осветили тему слотов и продолжили знакомство с сетевым программированием.

Практическое задание

Продолжаем работу над проектом «Мессенджер»:

1. Реализовать обработку нескольких клиентов на сервере, используя функцию **select**. Клиенты должны общаться в «общем чате»: каждое сообщение участника отправляется всем, подключенным к серверу.

2. Реализовать функции отправки/приема данных на стороне клиента. Чтобы упростить разработку на данном этапе, пусть клиентское приложение будет либо только принимать, либо только отправлять сообщения в общий чат. Эти функции надо реализовать в рамках отдельных скриптов.

Дополнительные материалы

1. Что такое select.
2. Разбираемся с устройством асинхронных фреймворков для Python.
3. Программирование на Python. Специальные методы и атрибуты классов.
4. Использование __slots__.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. David Beazley, Brian K. Jones. Python Cookbook. Third Edition (каталог «Дополнительные материалы»).
2. Лучано Ромальо. Python. К вершинам мастерства (каталог «Дополнительные материалы»).
3. Дэвид Бизли. Python. Подробный справочник (каталог «Дополнительные материалы»).
4. Сетевое программирование.
5. Язык программирования Python — стандартная библиотека.
6. Select — Waiting for I/O completion.
7. How to Work with TCP Sockets in Python.
8. Object.__slots__.
9. Магия __slots__.