21.21. `socketserver` — A framework for network servers

Source code: Lib/socketserver.py

The `socketserver` module simplifies the task of writing network servers.

There are four basic concrete server classes:

class `socketserver.`**`TCPServer`**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

> This uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. If *bind_and_activate* is true, the constructor automatically attempts to invoke `server_bind()` and `server_activate()`. The other parameters are passed to the `BaseServer` base class.

class `socketserver.`**`UDPServer`**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

> This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for `TCPServer`.

class `socketserver.`**`UnixStreamServer`**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
class `socketserver.`**`UnixDatagramServer`**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

> These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they're not available on non-Unix platforms. The parameters are the same as for `TCPServer`.

These four classes process requests *synchronously*; each request must be completed before the next request can be started. This isn't suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the `ForkingMixIn` and `ThreadingMixIn` mix-in classes can be used to support asynchronous behaviour.

Creating a server requires several steps. First, you must create a request handler class by subclassing the `BaseRequestHandler` class and overriding its `handle()` method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server's address and the request handler class. It is recommended to use the server in a `with` statement. Then call the handle_request() or serve_forever() method of the server object to process one or many requests. Finally, call server_close() to close the socket (unless you used a with statement).

When inheriting from ThreadingMixIn for threaded connection behavior, you should explicitly declare how you want your threads to behave on an abrupt shutdown. The ThreadingMixIn class defines an attribute *daemon_threads*, which indicates whether or not the server should wait for thread termination. You should set the flag explicitly if you would like threads to behave autonomously; the default is False, meaning that Python will not exit until all threads created by ThreadingMixIn have exited.

Server classes have the same external methods and attributes, no matter what network protocol they use.

21.21.1. Server Creation Notes

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types:

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

Note that UnixDatagramServer derives from UDPServer, not from UnixStreamServer — the only difference between an IP and a Unix stream server is the address family, which is simply repeated in both Unix server classes.

class socketserver.**ForkingMixIn**
class socketserver.**ThreadingMixIn**

Forking and threading versions of each type of server can be created using these mix-in classes. For instance, ThreadingUDPServer is created as follows:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

The mix-in class comes first, since it overrides a method defined in `UDPServer`. Setting the various attributes also changes the behavior of the underlying server mechanism.

`ForkingMixIn` and the Forking classes mentioned below are only available on POSIX platforms that support `fork()`.

class socketserver.**ForkingTCPServer**
class socketserver.**ForkingUDPServer**
class socketserver.**ThreadingTCPServer**
class socketserver.**ThreadingUDPServer**

These classes are pre-defined using the mix-in classes.

To implement a service, you must derive a class from `BaseRequestHandler` and redefine its `handle()` method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses `StreamRequestHandler` or `DatagramRequestHandler`.

Of course, you still have to use your head! For instance, it makes no sense to use a forking server if the service contains state in memory that can be modified by different requests, since the modifications in the child process would never reach the initial state kept in the parent process and passed to each child. In this case, you can use a threading server, but you will probably have to use locks to protect the integrity of the shared data.

On the other hand, if you are building an HTTP server where all data is stored externally (for instance, in the file system), a synchronous class will essentially render the service “deaf” while one request is being handled – which may be for a very long time if a client is slow to receive all the data it has requested. Here a threading or forking server is appropriate.

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a synchronous server and doing an explicit fork in the request handler class `handle()` method.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor `fork()` (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use `selectors` to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used). See `asyncore` for another way to manage this.

21.21.2. Server Objects

class `socketserver.`**`BaseServer`**(*server_address*, *RequestHandlerClass*)

This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses. The two parameters are stored in the respective `server_address` and `RequestHandlerClass` attributes.

`fileno`()

Return an integer file descriptor for the socket on which the server is listening. This function is most commonly passed to `selectors`, to allow monitoring multiple servers in the same process.

`handle_request`()

Process a single request. This function calls the following methods in order: `get_request()`, `verify_request()`, and `process_request()`. If the user-provided `handle()` method of the handler class raises an exception, the server's `handle_error()` method will be called. If no request is received within `timeout` seconds, `handle_timeout()` will be called and `handle_request()` will return.

`serve_forever`(*poll_interval=0.5*)

Handle requests until an explicit `shutdown()` request. Poll for shutdown every *poll_interval* seconds. Ignores the `timeout` attribute. It also calls `service_actions()`, which may be used by a subclass or mixin to provide actions specific to a given service. For example, the `ForkingMixIn` class uses `service_actions()` to clean up zombie child processes.

Changed in version 3.3: Added `service_actions` call to the `serve_forever` method.

`service_actions`()

This is called in the `serve_forever()` loop. This method can be overridden by subclasses or mixin classes to perform actions specific to a given service, such as cleanup actions.

New in version 3.3.

`shutdown`()

Tell the `serve_forever()` loop to stop and wait until it does.

`server_close()`

Clean up the server. May be overridden.

`address_family`

The family of protocols to which the server's socket belongs. Common examples are `socket.AF_INET` and `socket.AF_UNIX`.

`RequestHandlerClass`

The user-provided request handler class; an instance of this class is created for each request.

`server_address`

The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the `socket` module for details. For Internet protocols, this is a tuple containing a string giving the address, and an integer port number: `('127.0.0.1', 80)`, for example.

`socket`

The socket object on which the server will listen for incoming requests.

The server classes support the following class variables:

`allow_reuse_address`

Whether the server will allow the reuse of an address. This defaults to `False`, and can be set in subclasses to change the policy.

`request_queue_size`

The size of the request queue. If it takes a long time to process a single request, any requests that arrive while the server is busy are placed into a queue, up to `request_queue_size` requests. Once the queue is full, further requests from clients will get a "Connection denied" error. The default value is usually 5, but this can be overridden by subclasses.

`socket_type`

The type of socket used by the server; `socket.SOCK_STREAM` and `socket.SOCK_DGRAM` are two common values.

`timeout`

Timeout duration, measured in seconds, or `None` if no timeout is desired. If `handle_request()` receives no incoming requests within the timeout period, the `handle_timeout()` method is called.

There are various server methods that can be overridden by subclasses of base server classes like `TCPServer`; these methods aren't useful to external users of the server object.

`finish_request`(*request*, *client_address*)

Actually processes the request by instantiating `RequestHandlerClass` and calling its `handle()` method.

`get_request`()

Must accept a request from the socket, and return a 2-tuple containing the *new* socket object to be used to communicate with the client, and the client's address.

`handle_error`(*request*, *client_address*)

This function is called if the `handle()` method of a `RequestHandlerClass` instance raises an exception. The default action is to print the traceback to standard error and continue handling further requests.

Changed in version 3.6: Now only called for exceptions derived from the `Exception` class.

`handle_timeout`()

This function is called when the `timeout` attribute has been set to a value other than `None` and the timeout period has passed with no requests being received. The default action for forking servers is to collect the status of any child processes that have exited, while in threading servers this method does nothing.

`process_request`(*request*, *client_address*)

Calls `finish_request()` to create an instance of the `RequestHandlerClass`. If desired, this function can create a new process or thread to handle the request; the `ForkingMixIn` and `ThreadingMixIn` classes do this.

`server_activate`()

Called by the server's constructor to activate the server. The default behavior for a TCP server just invokes `listen()` on the server's socket. May be overridden.

`server_bind`()

Called by the server's constructor to bind the socket to the desired address. May be overridden.

`verify_request`(*request*, *client_address*)

Must return a Boolean value; if the value is `True`, the request will be processed, and if it's `False`, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always returns `True`.

Changed in version 3.6: Support for the context manager protocol was added. Exiting the context manager is equivalent to calling `server_close()`.

21.21.3. Request Handler Objects

class `socketserver.`**`BaseRequestHandler`**

This is the superclass of all request handler objects. It defines the interface, given below. A concrete request handler subclass must define a new `handle()` method, and can override any of the other methods. A new instance of the subclass is created for each request.

`setup`()

Called before the `handle()` method to perform any initialization actions required. The default implementation does nothing.

`handle`()

This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it; the request is available as `self.request`; the client address as `self.client_address`; and the server instance as `self.server`, in case it needs access to per-server information.

The type of `self.request` is different for datagram or stream services. For stream services, `self.request` is a socket object; for datagram services, `self.request` is a pair of string and socket.

`finish`()

Called after the `handle()` method to perform any clean-up actions required. The default implementation does nothing. If `setup()` raises an exception, this function will not be called.

class `socketserver.`**`StreamRequestHandler`**
class `socketserver.`**`DatagramRequestHandler`**

These `BaseRequestHandler` subclasses override the `setup()` and `finish()` methods, and provide `self.rfile` and `self.wfile` attributes. The `self.rfile` and `self.wfile` attributes can be read or written, respectively, to get the request data or return data to the client.

The `rfile` attributes of both classes support the `io.BufferedIOBase` readable interface, and `DatagramRequestHandler.wfile` supports the `io.BufferedIOBase` writable interface.

Changed in version 3.6: `StreamRequestHandler.wfile` also supports the `io.BufferedIOBase` writable interface.

21.21.4. Examples

21.21.4.1. `socketserver.TCPServer` Example

This is the server side:

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # Activate the server; this will keep running until you
        # interrupt the program with Ctrl-C
        server.serve_forever()
```

An alternative request handler class that makes use of streams (file-like objects that simplify communication by providing the standard file interface):

```
class MyTCPHandler(socketserver.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the handler;
```

```
        # we can now use e.g. readline() instead of raw recv() calls
        self.data = self.rfile.readline().strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # Likewise, self.wfile is a file-like object used to write bac
        # to the client
        self.wfile.write(self.data.upper())
```

The difference is that the `readline()` call in the second handler will call `recv()` multiple times until it encounters a newline character, while the single `recv()` call in the first handler will just return what has been sent from the client in one `sendall()` call.

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data + "\n", "utf-8"))

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

The output of the example should look something like this:

Server:

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

Client:

```
$ python TCPClient.py hello world with TCP
Sent:     hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
```

```
Sent:     python is nice
Received: PYTHON IS NICE
```

21.21.4.2. `socketserver.UDPServer` Example

This is the server side:

```
import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and sir
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print("{} wrote:".format(self.client_address[0]))
        print(data)
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
        server.serve_forever()
```

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

The output of the example should look exactly like for the TCP server example.

21.21.4.3. Asynchronous Mixins

To build asynchronous handlers, use the `ThreadingMixIn` and `ForkingMixIn` classes.

An example for the `ThreadingMixIn` class:

```
import socket
import threading
import socketserver

class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data), 'asci
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPS
    pass

def client(ip, port, message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((ip, port))
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))

if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler
    with server:
        ip, port = server.server_address

        # Start a thread with the server -- that thread will then star
        # more thread for each request
        server_thread = threading.Thread(target=server.serve_forever)
        # Exit the server thread when the main thread terminates
        server_thread.daemon = True
        server_thread.start()
        print("Server loop running in thread:", server_thread.name)

        client(ip, port, "Hello World 1")
        client(ip, port, "Hello World 2")
        client(ip, port, "Hello World 3")

        server.shutdown()
```

The output of the example should look something like this:

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

The `ForkingMixIn` class is used in the same way, except that the server will spawn a new process for each request. Available only on POSIX platforms that support `fork()`.