

## SocketServer

类的继承关系

编程接口

实现EchoServer

练习——改写ChatServer

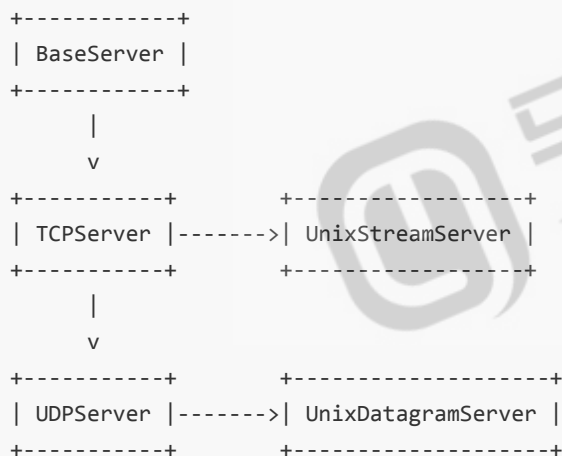
解决客户端主动连接断开问题

总结

# SocketServer

socket编程过于底层，编程虽然有套路，但是想要写出健壮的代码还是比较困难的，所以很多语言都对socket底层API进行封装，Python的封装就是——socketserver模块。它是网络服务编程框架，便于企业级快速开发。

## 类的继承关系



SocketServer简化了网络服务器的编写。

它有4个同步类：TCPServer，UDPServer，UnixStreamServer，UnixDatagramServer。

2个Mixin类：ForkingMixIn 和 ThreadingMixIn 类，用来支持异步。

```
class ForkingUDPServer(ForkingMixIn, UDPServer): pass
```

```
class ForkingTCPServer(ForkingMixIn, TCPServer): pass
```

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

```
class ThreadingTCPServer(ThreadingMixIn, TCPServer): pass
```

fork是创建多进程，thread是创建多线程

## 编程接口

socketserver.BaseServer(server\_address, RequestHandlerClass)

需要提供服务器绑定的地址信息，和用于处理请求的RequestHandlerClass类。

RequestHandlerClass类必须是BaseRequestHandler类的子类，在BaseServer中代码如下：

```
# BaseServer代码
class BaseServer:
    def __init__(self, server_address, RequestHandlerClass):
        """Constructor. May be extended, do not override."""
        self.server_address = server_address
        self.RequestHandlerClass = RequestHandlerClass
        self.__is_shut_down = threading.Event()
        self.__shutdown_request = False

    def finish_request(self, request, client_address): # 处理请求的方法
        """Finish one request by instantiating RequestHandlerClass."""
        self.RequestHandlerClass(request, client_address, self) # RequestHandlerClass构造
```

BaseRequestHandler类

它是和用户连接的用户请求处理类的基类，定义为BaseRequestHandler(request, client\_address, server)

服务端Server实例接收用户请求后，最后会实例化这个类。

它被初始化时，送入3个构造参数：request, client\_address, server自身

以后就可以在BaseRequestHandler类的实例上使用以下属性：

self.request是和客户端的连接socket对象

self.server是TCPServer本身

self.client\_address是客户端地址

这个类在初始化的时候，它会依次调用3个方法。子类可以覆盖这些方法。

```
# BaseRequestHandler要子类覆盖的方法
class BaseRequestHandler:
    def __init__(self, request, client_address, server):
        self.request = request
        self.client_address = client_address
        self.server = server
        self.setup()
        try:
            self.handle()
        finally:
            self.finish()

    def setup(self): # 每一个连接初始化
        pass

    def handle(self): # 每一次请求处理
        pass

    def finish(self): # 每一个连接清理
        pass
```

## 测试代码

```
import threading
import socketserver

class MyHandler(socketserver.BaseRequestHandler):
    def handle(self):
        # super().handle() # 可以不调用, 父类handle什么都没有做
        print('-'*30)
        print(self.server) # 服务
        print(self.request) # 服务端负责客户端连接请求的socket对象
        print(self.client_address) # 客户端地址
        print(self.__dict__)
        print(self.server.__dict__) # 能看到负责accept的socket

        print(threading.enumerate())
        print(threading.current_thread())
        print('-'*30)

addr = ('192.168.142.1', 9999)
server = socketserver.ThreadingTCPServer(addr, MyHandler)

server.serve_forever() # 永久
```

测试结果说明, handle方法相当于socket的recv方法。

每个不同的连接上的请求过来后, 生成这个连接的socket对象即self.request, 客户端地址是self.client\_address。

### 问题

测试过程中, 上面代码, 连接后立即断开了, 为什么?

怎样才能客户端和服务端长时间连接?

```
import threading
import socketserver
import logging

FORMAT = "%(asctime)s %(threadName)s %(thread)d %(message)s"
logging.basicConfig(format=FORMAT, level=logging.INFO)

class MyHandler(socketserver.BaseRequestHandler):
    def handle(self):
        # super().handle() # 可以不调用, 父类handle什么都没有做
        print('-'*30)
        print(self.server) # 服务
        print(self.request) # 服务端负责客户端连接请求的socket对象
        print(self.client_address) # 客户端地址
        print(self.__dict__)
        print(self.server.__dict__) # 能看到负责accept的

        print(threading.enumerate())
        print(threading.current_thread())
```

```

print('-'*30)
for i in range(3):
    data = self.request.recv(1024)
    logging.info(data)
    logging.info('====end====')

addr = ('192.168.142.1', 9999)
server = socketserver.ThreadingTCPServer(addr, MyHandler)

server.serve_forever() # 永久

```

将ThreadingTCPServer换成TCPServer，同时连接2个客户端观察效果。

ThreadingTCPServer是异步的，可以同时处理多个连接。

TCPServer是同步的，一个连接处理完了，即一个连接的handle方法执行完了，才能处理另一个连接，且只有主线程。

## 总结

创建服务器需要几个步骤：

1. 从BaseRequestHandler类派生出子类，并覆盖其handle()方法来创建请求处理程序类，此方法将处理传入请求
2. 实例化一个服务器类，传参服务器的地址和请求处理类
3. 调用服务器实例的handle\_request()或serve\_forever()方法
4. 调用server\_close()关闭套接字

## 实现EchoServer

顾名思义，Echo，来什么消息回显什么消息  
客户端发来什么信息，返回什么信息

```

import threading
from socketserver import ThreadingTCPServer, BaseRequestHandler
import sys

class EchoHandler(BaseRequestHandler):
    def setup(self):
        super().setup()
        self.event = threading.Event() # 初始工作

    def finish(self):
        super().finish()
        self.event.set()

    def handle(self):
        super().handle()

        while not self.event.is_set():
            data = self.request.recv(1024).decode()
            msg = "{} {}".format(self.client_address, data).encode()

```

```

        self.request.send(msg)
    print('End')

addr = ('0.0.0.0', 9999)
server = ThreadingTCPServer(addr, EchoHandler)

server_thread = threading.Thread(target=server.serve_forever, name='EchoServer', daemon=True)
server_thread.start()

try:
    while True:
        cmd = input('>>>')
        if cmd.strip() == 'quit':
            break
        print(threading.enumerate())
except Exception as e:
    print(e)
except KeyboardInterrupt:
    pass
finally:
    print('Exit')
    sys.exit(0)

```

## 练习——改写ChatServer

使用ThreadingTCPServer改写ChatServer

```

import threading
from socketserver import ThreadingTCPServer, BaseRequestHandler
import sys
import logging

FORMAT = "%(asctime)s %(threadName)s %(thread)d %(message)s"
logging.basicConfig(format=FORMAT, level=logging.INFO)

class ChatHandler(BaseRequestHandler):
    clients = {}

    def setup(self):
        super().setup()
        self.event = threading.Event() # 初始工作
        self.clients[self.client_address] = self.request

    def finish(self):
        super().finish() # 清理工作
        self.clients.pop(self.client_address) # 能执行到吗?
        self.event.set()

    def handle(self):
        super().handle()

```

```

while not self.event.is_set():
    data = self.request.recv(1024).decode()
    if data == 'quit':
        break
    msg = "{} {}".format(self.client_address, data).encode()
    logging.info(msg)
    for c in self.clients.values():
        self.request.send(msg)
print('End')

addr = ('0.0.0.0', 9999)
server = ThreadingTCPServer(addr, ChatHandler)

server_thread = threading.Thread(target=server.serve_forever, name='ChatServer', daemon=True)
server_thread.start()

try:
    while True:
        cmd = input('>>>')
        if cmd.strip() == 'quit':
            break
        print(threading.enumerate())
except Exception as e:
    print(e)
except KeyboardInterrupt:
    pass
finally:
    print('Exit')
    sys.exit(0)

```

## 问题

上例 `self.clients.pop(self.client_address)` 能执行到吗？

如果连接的线程中 `handle` 方法中抛出异常，例如客户端主动断开导致的异常，线程崩溃，`self.clients` 的 `pop` 方法还能执行吗？

当然能执行，基类源码保证了即使异常，也能执行 `finish` 方法。但不代表不应该不捕获客户端各种异常。

## 解决客户端主动连接断开问题

如果客户端主动断开，总是抛出一个异常。看看到底发生了什么，在 `handle` 方法中增加一些语句。

```

class ChatHandler(BaseRequestHandler):
    clients = {}

    def setup(self):
        super().setup()
        self.event = threading.Event() # 初始工作
        self.clients[self.client_address] = self.request

    def finish(self):

```

```

        super().finish() # 清理工作
        self.clients.pop(self.client_address) # 能执行到吗？
        self.event.set()

    def handle(self):
        super().handle()

    while not self.event.is_set():
        data = self.request.recv(1024).decode()
        print(data, '~~~~~') # 增加
        if data == 'quit':
            break
        msg = "{} {}".format(self.client_address, data).encode()
        logging.info(msg)
        for c in self.clients.values():
            print('+++++') # 增加
            self.request.send(msg)
    print('End')

```

通过打印可以看到，客户端主动断开，会导致recv方法立即返回一个空bytes，**并没有同时抛出异常**。当循环回到recv这一句的时候就会抛出异常。所以，可以通过判断data数据是否为空来判断客户端是否断开。

```

class ChatHandler(BaseRequestHandler):
    clients = {}

    def setup(self):
        super().setup()
        self.event = threading.Event() # 初始工作
        self.clients[self.client_address] = self.request

    def finish(self):
        super().finish() # 清理工作
        self.clients.pop(self.client_address) # 能执行到吗？
        self.event.set()

    def handle(self):
        super().handle()

    while not self.event.is_set():
        data = self.request.recv(1024).decode()
        print(data, '~~~~~')
        if not data or data == 'quit':
            print('Broken pipe')
            break
        msg = "{} {}".format(self.client_address, data).encode()
        logging.info(msg)
        for c in self.clients.values():
            self.request.send(msg)
    print('End')

```

## 总结

---

为每一个连接提供RequestHandlerClass类实例，依次调用setup、handle、finish方法，且使用了try...finally结构保证finish方法一定能被调用。这些方法依次执行完成，如果想维持这个连接和客户端通信，就需要在handle函数中使用循环。

socketserver模块提供的不同的类，但是编程接口是一样的，即使是多进程、多线程的类也是一样，大大减少了编程的难度。

