# HTTPServer

**注意：针对tornado是2.4.1版本 以下代码有删减，具体见源代码**

实现代码

tornado/httpserver.py HTTPServer

Tornado/netutil.py TCPServer （3.0.x版本在tornado/tcpserver.py）

## 结构图

httpsvr\_callback

HTTPConnection1

HTTPSVR

I OLOOP

HTTPConnection2

## HTTPServer

class HTTPServer(TCPServer):

def \_\_init\_\_(self, request\_callback, no\_keep\_alive=False, io\_loop=None,

xheaders=False, ssl\_options=None, \*\*kwargs):

self.request\_callback = request\_callback

self.no\_keep\_alive = no\_keep\_alive

self.xheaders = xheaders

TCPServer.\_\_init\_\_(self, io\_loop=io\_loop, ssl\_options=ssl\_options,

\*\*kwargs)

def handle\_stream(self, stream, address):

HTTPConnection(stream, address, self.request\_callback,

self.no\_keep\_alive, self.xheaders)

HTTPServer 实现很简单 继承TCPServer，并重写父类的handle\_stream 方法，

处理客户端客户端过来的连接。

## TCPServer

非堵塞，单线程 TCP server.

用 `TCPServer`,定义子类必须重写 `handle\_stream`。

`TCPServer` 初始化有3种方式：

1. `listen`: simple single-process::

server = TCPServer()

server.listen(8888)

IOLoop.instance().start()

2. `bind`/`start`: simple multi-process::

server = TCPServer()

server.bind(8888)

server.start(0) # Forks multiple sub-processes

IOLoop.instance().start()

When using this interface, an `IOLoop` must \*not\* be passed

to the `TCPServer` constructor. `start` will always start

the server on the default singleton `IOLoop`.

3. `add\_sockets`: advanced multi-process::

sockets = bind\_sockets(8888)

tornado.process.fork\_processes(0)

server = TCPServer()

server.add\_sockets(sockets)

IOLoop.instance().start()

The `add\_sockets` interface is more complicated, but it can be

used with `tornado.process.fork\_processes` to give you more

flexibility in when the fork happens. `add\_sockets` can

also be used in single-process servers if you want to create

your listening sockets in some way other than

`bind\_sockets`.

看一种方式实现：

server = TCPServer()

server.listen(8888)

IOLoop.instance().start()

Listen函数实现

创建绑定socket

向ioloop 注册连接处理函数（READ事件）

连接处理函数，最终调用的是handle\_stream函数，该函数需要子类实现

def \_handle\_connection(self, connection, address):

....

try:

if self.ssl\_options is not None:

stream = SSLIOStream(connection, io\_loop=self.io\_loop)

else:

stream = IOStream(connection, io\_loop=self.io\_loop)

self.handle\_stream(stream, address)

except Exception:

logging.error("Error in connection callback", exc\_info=True)

#子类要覆盖的，第一个参数是IOStream 第二个是客户端地址

def handle\_stream(self, stream, address):

"""Override to handle a new `IOStream` from an incoming connection."""

raise NotImplementedError()

回过头看Httpserver的处理函数，就调用了一个HTTPConnection

他会解析HTTP的headers和bodies，最后执行请求回调（callback（httprequest））

## HTTPConnection

大致处理流程

HTTPConnection --> self.\_on\_headers ==> 构造一个self.\_request = HTTPRequest()对象

--> self.\_on\_request\_body # 如果没有Content-Length 跳过 此步骤

==》进一步解析HTTPRequest 回调 self.request\_callback(self.\_request)

最后回调函数传人的参数是HTTPRequest 对象