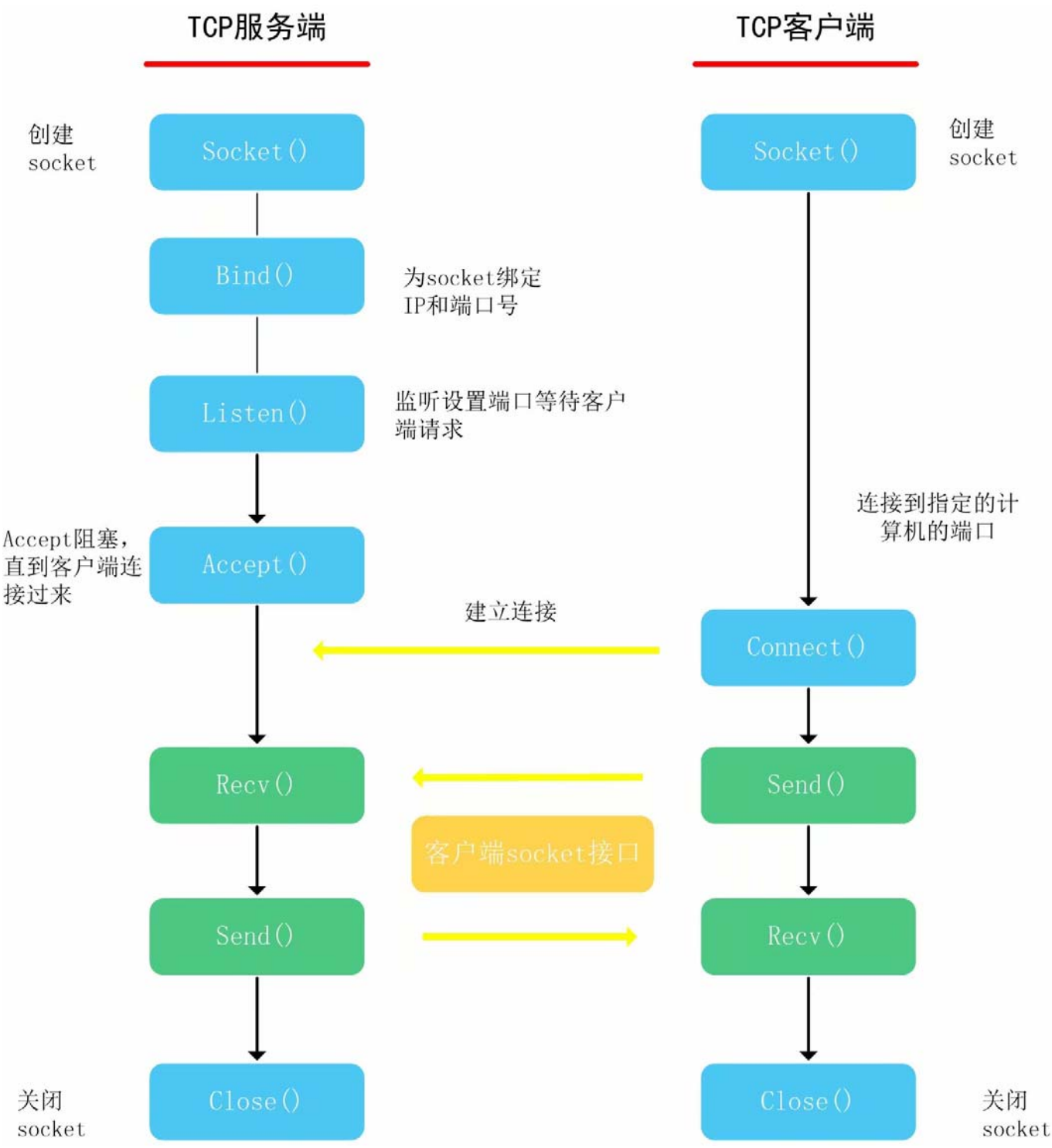
python网络编程——socke,socketserver

socket

流程框架



流程描述

1. 服务器根据地址类型(ipv4, ipv6)，**socket**类型，协议类型创建**socket**

2. 服务器为**socket**绑定ip地址和端口号

3. 服务器**socket**监听端口请求，随时准备接受客户端发来的连接，这时候服务器的**socket**并没有被打开

4. 客户端创建**socket**

5. 客户端打开**socket**，根据服务器ip地址和端口号试图连接服务器端**socket**

6. 服务器端**socket**接收到客户端**socket**请求，被动打开，开始接收客户端请求，直到客户端端返回连接信息，此时**socket**进入阻塞状态，所谓阻塞状态即**accept()**方法一直等到客户端返回连接信息后才继续执行，开始接收下一个客户端请求。

7. 客户端连接成功，向服务器发送连接状态信息

8. 服务器**accept()**方法返回，连接成功

9. 客户端向**socket**写入信息（或服务器端向**socket**写入信息）

10. 服务器读取信息（客户端读取信息）

11. 客户端关闭

12. 服务器端关闭

相关方法及参数

```
socket.bind(address)
    # 绑定连接地址。address地址格式取决于地址族。在AF_INET下，以元组(host, post)的形式表示地址

socket.listen(backlog)
    # 开始监听传入连接，backlog指定在拒绝连接之前，可以挂起的最大连接数。

socket.setblocking(bool)
    # 是否阻塞（默认为True），如果设置成False，那么accept和recv一旦无数据，则报错

socket.accept()
    # 接收并返回(conn, address)，其中conn是新的套接字对象，可以用来接收和发送数据，address是客户端地址信息

socket.connect(address)
    # 连接address处的套接字，如果连接出错，则返回socket.error错误

socket.connect_ex(address)
    # 同上，正常连接返回0，连接失败返回错误编码，例如：10061

socket.close()
    # 关闭套接字

socket.recv(bufsize[, flags])
    # 接收发送的数据，数据以字节形式返回，bufsize指定最多可以一次接收多少字节，flags提供有关消息的其他信息，通常可以忽略

socket.recvfrom(bufsize[, flags])
    # 与recv类似，但返回值是(data, address)，其中data是传送的数据，address是发送端的地址
```

```
socket.send(data[, flags])
    # 将字节数据data发送，返回值是发送的字节数量，该数量可能小于data的字节大小，因为可能未将制定内容全部发送。

socket.sendall(data[, flags])
    # 将字节数据data发送，但在返回之前会尝试发送所有数据，成功返回None，失败则抛出异常 -----> 内部是通过递归调用send，将所有内容发送出去

socket.sendto(data[, flags], address)
    # 制定地址发送数据，主要用户UDP协议，返回值是发送出去的字节数

socket.settimeout(timeout)
    # 设置套接字操作的超时时间，timeout是一个浮点数，单位是秒，值为None表示没有超时。

socket.getpeername() # 返回套接字的远程地址
socket.getsockname() # 返回套接字本身地址
socket.fileno() # 套接字的文件描述符
```

socketserver

socketserver模块简化了编写网络服务程序的任务。同时socketserver模块也是python标准库中很多服务器框架的基础

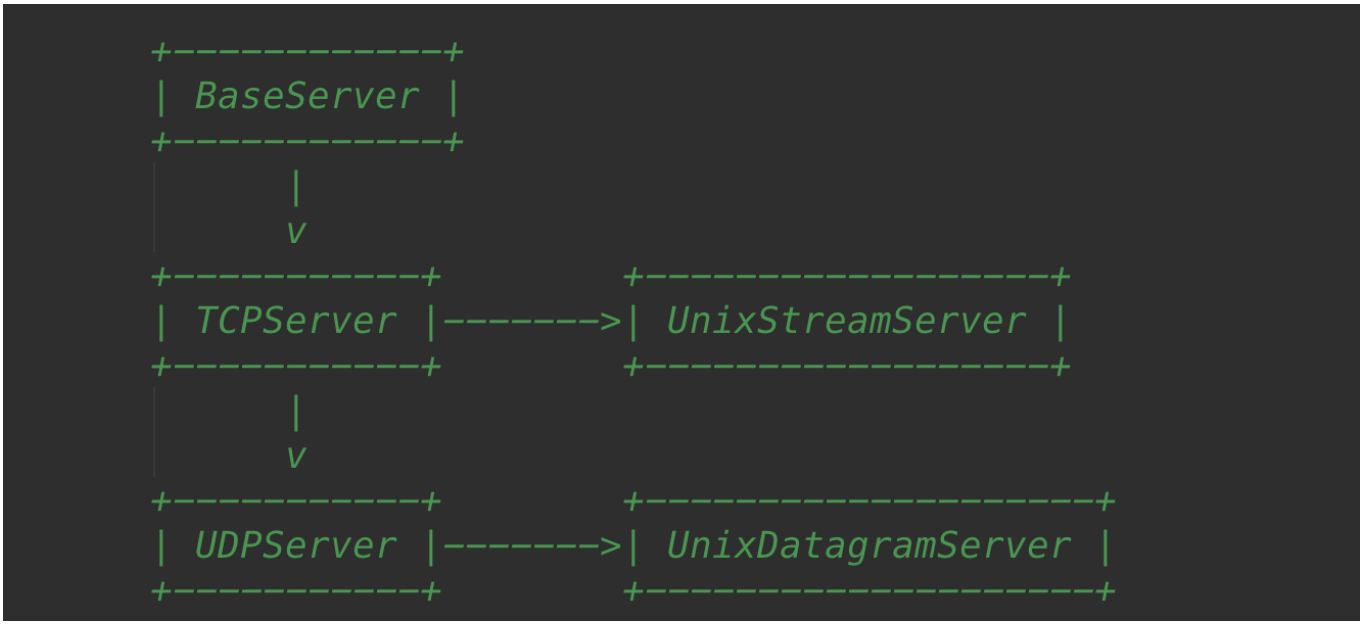
python把网络服务抽象成两个主要的类

- Server类，用于处理连接相关的网络操作，
- RequestHandler类，用于处理请求的相关操作类
- （MixIn类，用于扩展Server类，实现多进程和多线程）

Server类

- BaseServer
- TCPServer
- UDPServer
- UnixStreamServer
- UnixDatagramServer

> TCPServer 和 UDPServer 相对于 UnixStreamServer 和 UnixDatagramServer之间差别仅仅在于在Unix环境下使用 AF_UNIX 字段



RequestHandler类

所有RequstHandler类都继承BaseRequestHandler基类

使用socketserver流程

1. 首先，你必须通过继承**BaseRequestHandler**类，并且重写**handle()**方法来定制一个用于处理请求的类；其中**handle()**方法用于处理客户端发来的请求

2. 接着，你必须实例化一个**Server**类对象(上述的Server类中，根据自己的需求进行实例化)，并传入服务器地址和刚刚定制的用于处理请求的类。

3. 然后，通过**Server**类对象调用**handle_request()**或者**serve_forever()**方法来处理客户端发来的请求。

4. 最后，调用**serve_close()**方法来关闭**socket**。

一个简单的实例：

```
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
    server = socketserver.TCPServer((HOST, PORT), MyTCPHandler)

    # Activate the server; this will keep running until you
    # interrupt the program with Ctrl-C
    server.serve_forever()
```

如果想让服务器端实现并发处理请求的效果，必须选择使用以下类中的一个来替换上面创建的Server类。

- ForkingTCPServer
- ForkingUDPServer
- ThreadingTCPServer
- ThreadingUDPServer

```
server = socketserver.TCPServer((HOST, PORT), MyTCPHandler)
#替换为
server = socketserver.ThreadingTCPServer((HOST, PORT), MyTCPHandler)
```