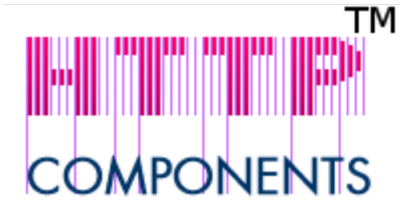


HTTPCOMPONENTS 学习报告

范楷元 2016K8009926017

目录

HttpComponents 版本：v4.4.10

本文档是基于国科大王伟老师的面向对象设计课程所做的一个对 HttpComponents 的较为全面的深入理解。主要内容分为 3 个部分：

第一部分：HttpComponents 的功能分析与建模

第二部分：HttpComponents 的核心类、对象、运作原理

第三部分：HttpComponents 的高级设计意图

第一部分

Apache HttpComponents 项目是负责创建和维护针对 HTTP 和相关协议的底层 Java 组件的工具集。超文本传输协议（HTTP）可能是当今网络上最重要的协议，基于网络的应用和网络计算的增

长对 HTTP 协议角色的扩展作用远胜于用户驱动的 Web 浏览器，同时不断增长的更多应用也需要 HTTP 协议的支持。HttpComponents 正是为了在支持基本的 HTTP 协议基础上，提供扩展而设计的。

HttpComponents 包含以下几部分：

- HttpComponents Core：

  HttpCore 实现了一系列的底层传输功能，利用这些功能，我们可以构建自己的 client 和 server。HttpCore 支持两种 I/O 模式：

  - 阻塞型 Blocking：基于典型的 Java 的 I/O 模型
  - 非阻塞型 Non-Blocking：基于 Java 的 NIO，事件驱动型

- HttpComponents Client：

  HttpClient 是基于 HttpCore 的一个代理侧的传输库（官方的话：Client-side HTTP transport library based on HttpCore）。

- HttpComponents AsyncClient：

  异步读写，基于 HttpCore NIO 和 HttpClient 的 HTTP/1.1 兼容代理，是 HttpClient 在特殊情况下的补充模块。

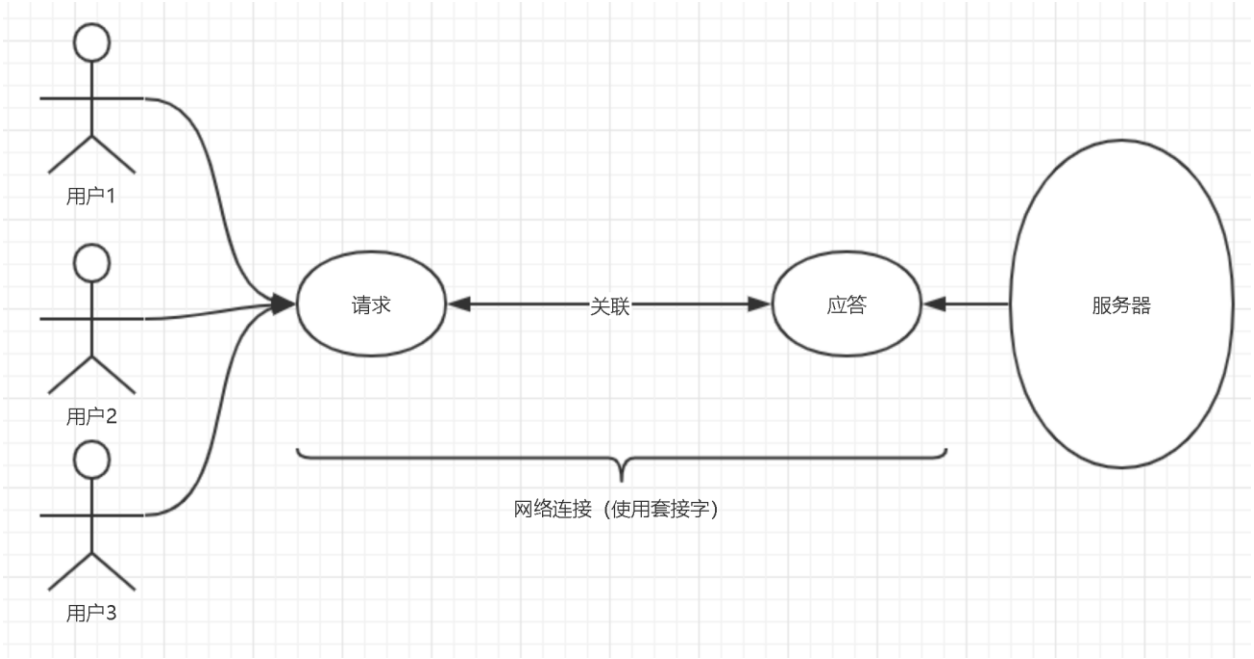
简单来说，HttpCore 是一套实现了 HTTP 协议的底层包，而 HttpClient 相当于使用了 HttpCore 这个方法的一个“客户端”。本报告主要分析 HttpCore。

用户在使用 HttpComponents 的时候，希望通过 HTTP 协议从网络上获取信息。可以建立用例模型：

用例名称	HTTP 请求
参与者	客户端，服务器
用例描述	该用例描述一个客户端和服务器之间的请求-响应行为
触发器	客户端需要向服务器请求数据时，该用例触发
前置条件	双方需要遵守 HTTP 协议

后置条件	如果所请求的数据不存在，返回错误
基本事件流	1 客户端进行发送请求的准备工作 2 建立连接 3 根据请求方式，将所请求的 uri 按照 http 协议发送到服务器端 4 服务器接收到请求，分配一个线程来处理请求 5 服务器根据 uri 找到所请求的数据 6 将所请求的数据通过之前的连接传给客户端 7 关闭连接
结论	当客户端收到服务器发送的数据或其他异常信息时，用例结束。

UML 用例图如下,



第二部分

这一部分我们来针对 HttpComponents 的核心功能，来看看实现核心功能所需要的重要的类以及他们的相互关系。

在官方提供的源码中，有 example 这样一个文件夹，里面是实现一些基本功能的例子，我们就从这里入手。先看第一个较为简单的 HttpGet，源代码如下

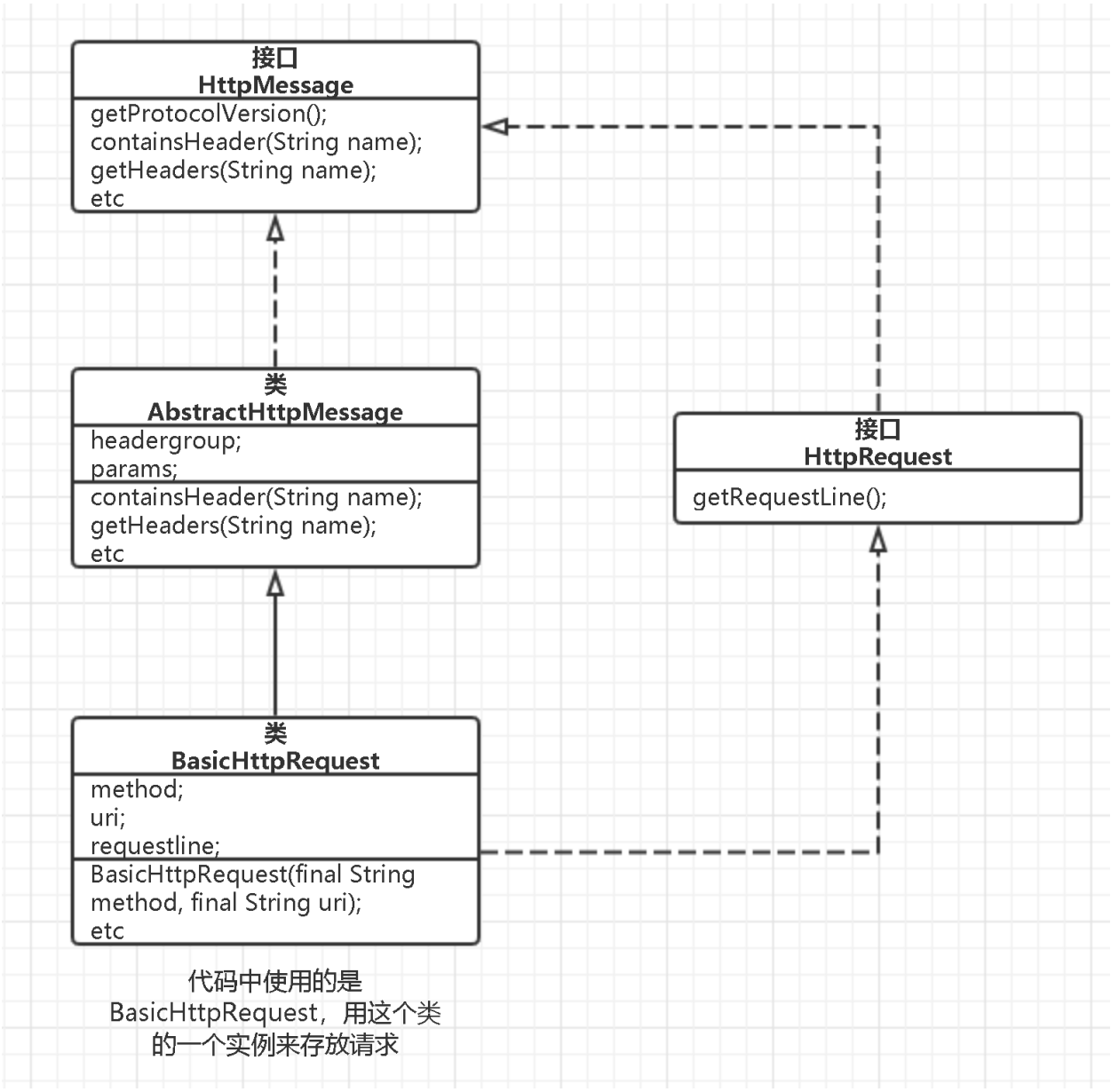
```
package org.apache.http.examples;


import java.net.Socket;


import org.apache.http.ConnectionReuseStrategy;
import org.apache.http.HttpHost;
import org.apache.http.HttpResponse;
import org.apache.http.impl.DefaultBHttpClientConnection;
import org.apache.http.impl.DefaultConnectionReuseStrategy;
import org.apache.http.message.BasicHttpRequest;
import org.apache.http.protocol.HttpCoreContext;
import org.apache.http.protocol.HttpProcessor;
import org.apache.http.protocol.HttpProcessorBuilder;
import org.apache.http.protocol.HttpRequestExecutor;
import org.apache.http.protocol.RequestConnControl;
import org.apache.http.protocol.RequestContent;
import org.apache.http.protocol.RequestExpectContinue;
import org.apache.http.protocol.RequestTargetHost;
import org.apache.http.protocol.RequestUserAgent;
import org.apache.http.util.EntityUtils;


/**
```

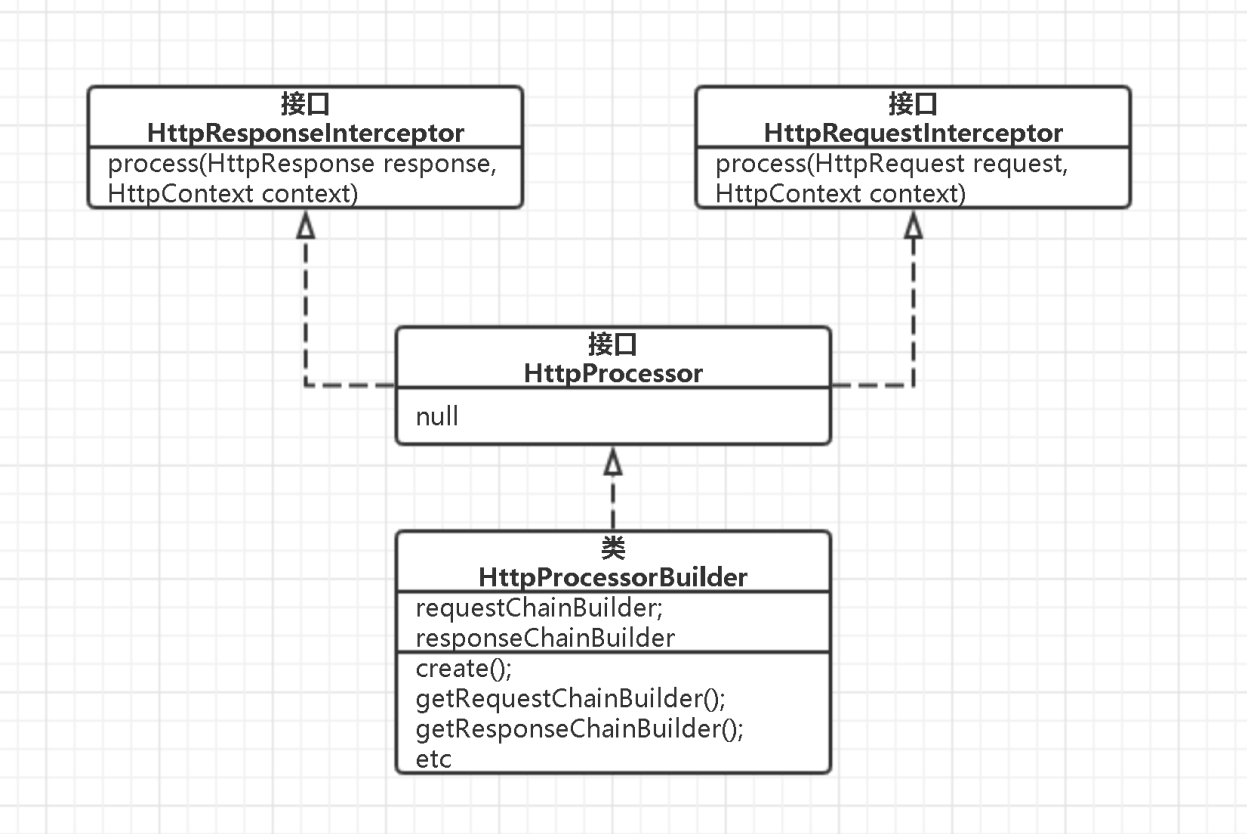
```
 * Elemental example for executing multiple GET requests sequentially.
 */
public class ElementalHttpGet {

    public static void main(String[] args) throws Exception {
        HttpProcessor httpproc = HttpProcessorBuilder.create()
            .add(new RequestContent())
            .add(new RequestTargetHost())
            .add(new RequestConnControl())
            .add(new RequestUserAgent("Test/1.1"))
            .add(new RequestExpectContinue(true)).build();

        HttpRequestExecutor httpexecutor = new HttpRequestExecutor();

        HttpCoreContext coreContext = HttpCoreContext.create();
        HttpHost host = new HttpHost("localhost", 8080);
        coreContext.setTargetHost(host);

        DefaultBHttpClientConnection conn = new DefaultBHttpClientConnection(8 * 1024);
        ConnectionReuseStrategy connStrategy = DefaultConnectionReuseStrategy.INSTANCE;

        try {

            String[] targets = {
                    "/",
                    "/servlets-examples/servlet/RequestInfoExample",
                    "/somewhere%20in%20pampa"};

            for (int i = 0; i < targets.length; i++) {
                if (!conn.isOpen()) {
```

```
                Socket socket = new Socket(host.getHostName(), host.getPort());
                conn.bind(socket);
            }
            BasicHttpRequest request = new BasicHttpRequest("GET", targets[i]);
            System.out.println(">> Request URI: " + request.getRequestLine().getUri());

            httpexecutor.preProcess(request, httpproc, coreContext);
            HttpResponse response = httpexecutor.execute(request, conn, coreContext);
            httpexecutor.postProcess(response, httpproc, coreContext);

            System.out.println("<< Response: " + response.getStatusLine());
            System.out.println(EntityUtils.toString(response.getEntity()));
            System.out.println("==============");
            if (!connStrategy.keepAlive(response, coreContext)) {
                conn.close();
            } else {
                System.out.println("Connection kept alive...");
            }
          }
        } finally {
          conn.close();
        }
      }
    }
```

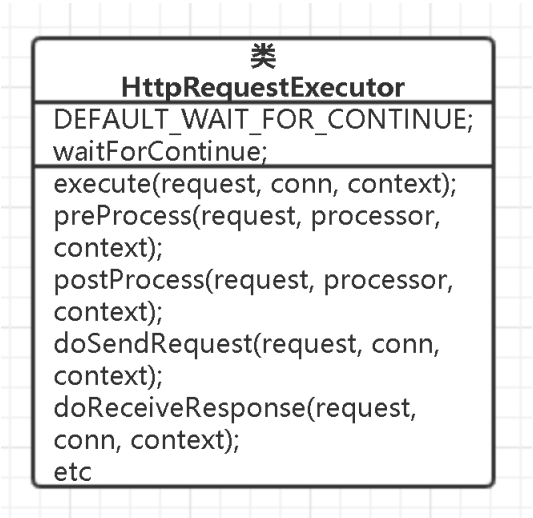
一、主要的类

对于其中较为重要的类和接口，首先整理类图如下

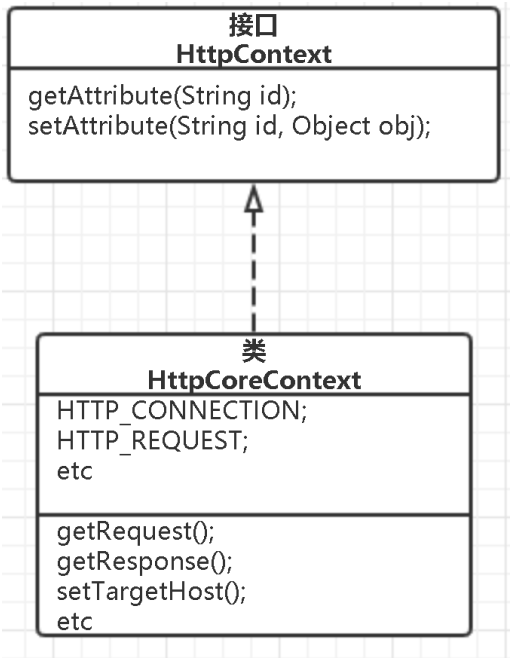

接口
HttpMessage
getProtocolVersion();
containsHeader(String name);
getHeaders(String name);
etc
类
AbstractHttpMessage
headergroup;
params;
containsHeader(String name);
getHeaders(String name);
etc
接口
HttpRequest
getRequestLine();
类
BasicHttpRequest
method;
uri;
requestline;
BasicHttpRequest(final String method, final String uri);
etc
代码中使用的是BasicHttpRequest，用这个类的一个实例来存放请求



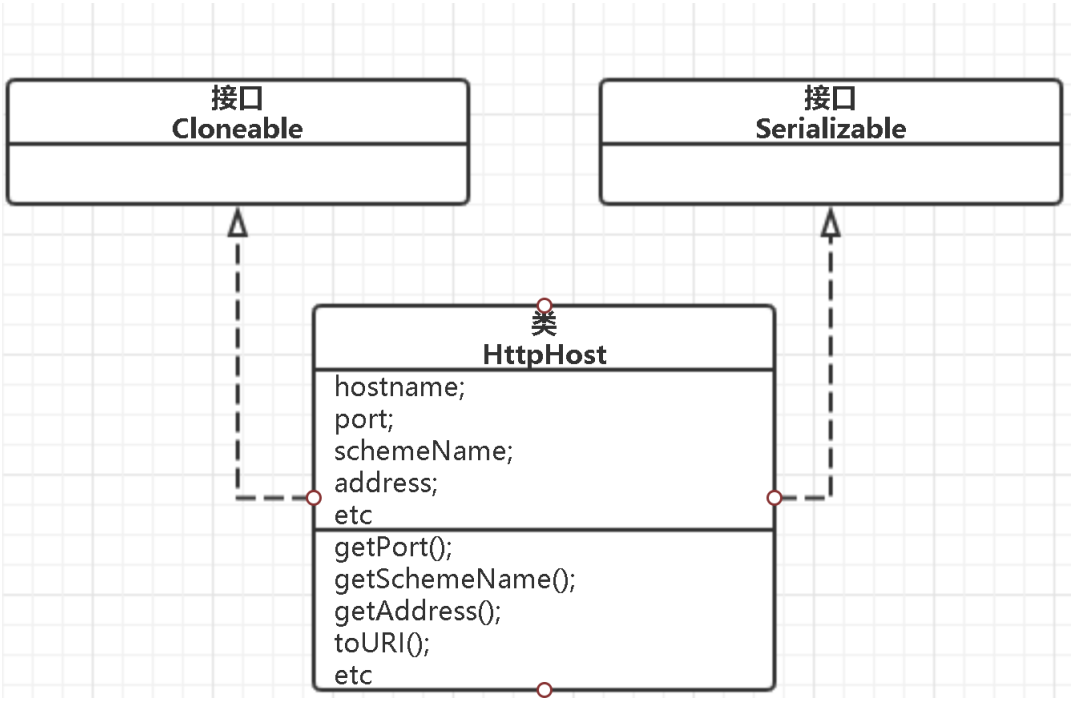
HttpProcessor 的实例用来表示 HTTP 请求的处理器，但是这里的处理器和我们说的硬件的处理器不同，主要工作并不是由处理器完成的。他的主要功能（根据类中的方法推测）应该是将若干个请求和响应放到一个 chain 中排队。



回收前面埋下的伏笔，在 HttpCore 中，真正完成工作，或者说把完成工作的功能封装在了一起的是这个执行器，在调用 execute 方法处理 request 的时候，会进一步调用 executor 的私有方法 doSendRequest。在 doSendRequest 中，会调用 connection 的方法来完成请求的发送，具体说明放到下面 connection 相关类图之后了。



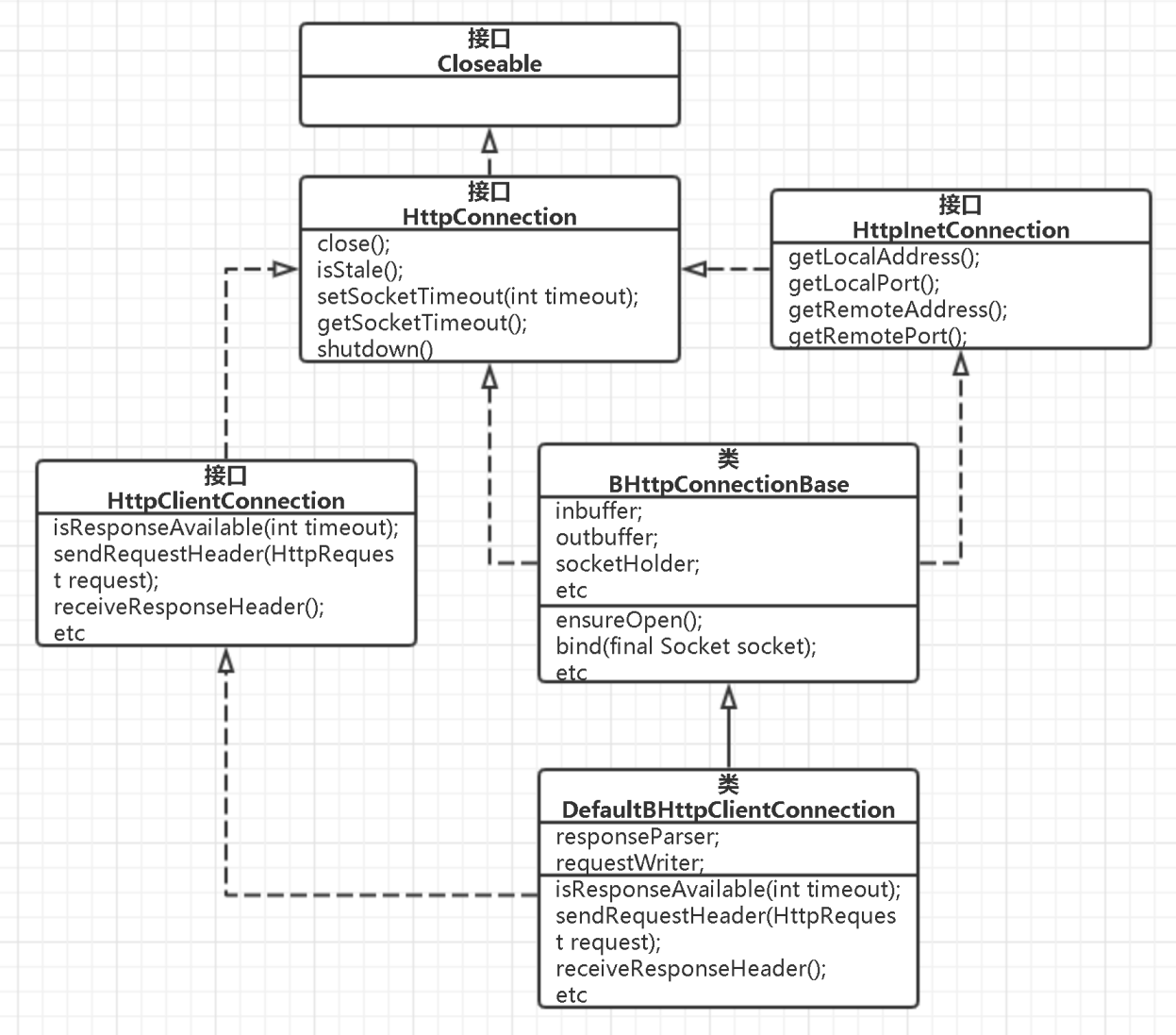
HttpCoreContext 类实现了 HttpContext 接口，在代码中体现为 coreContext 实例，用来记录 HttpCore 的上下文，包括请求、响应、目标服务器等。在使用执行器 executor 的时候是必不可少的一部分。



HttpHost 类用来设置请求的 Host，在这个例子里是 localhost 和 8080 端口，是本地计算机的默认 WEB 发布路径。这里对 Host 做一下简要说明。HTTP 协议本身的解释是这样的：“Host”是服

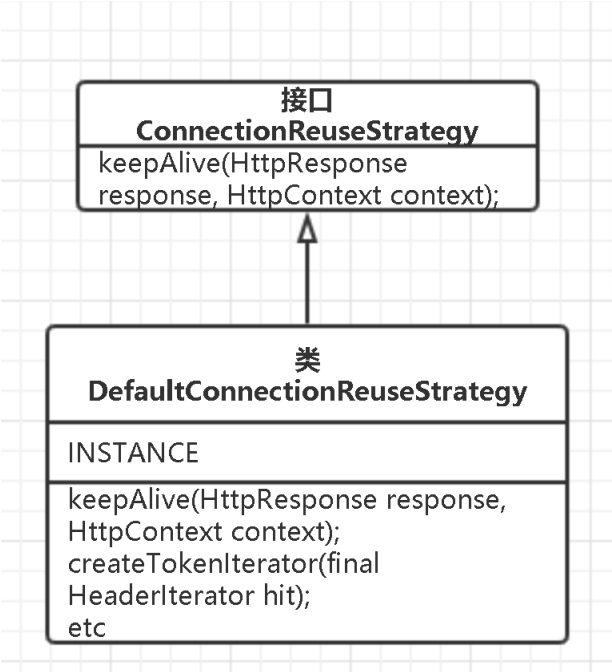
务器在处理对一个 IP 地址的多个请求时，用于区分资源的。在 HTTP1.1 协议中，用户发送请求时必须包含 Host 字段（如果没有指定 host，则返回 404）。而且因为 Host 非常重要，一般是请求语句之后的第一个 header。举一个简单的例子，向原始服务器请求 http://www.example.org/pub/WWW/的 GET 请求会以下面的语句开始：

```
GET /pub/WWW/ HTTP/1.1
Host: www.example.org
```



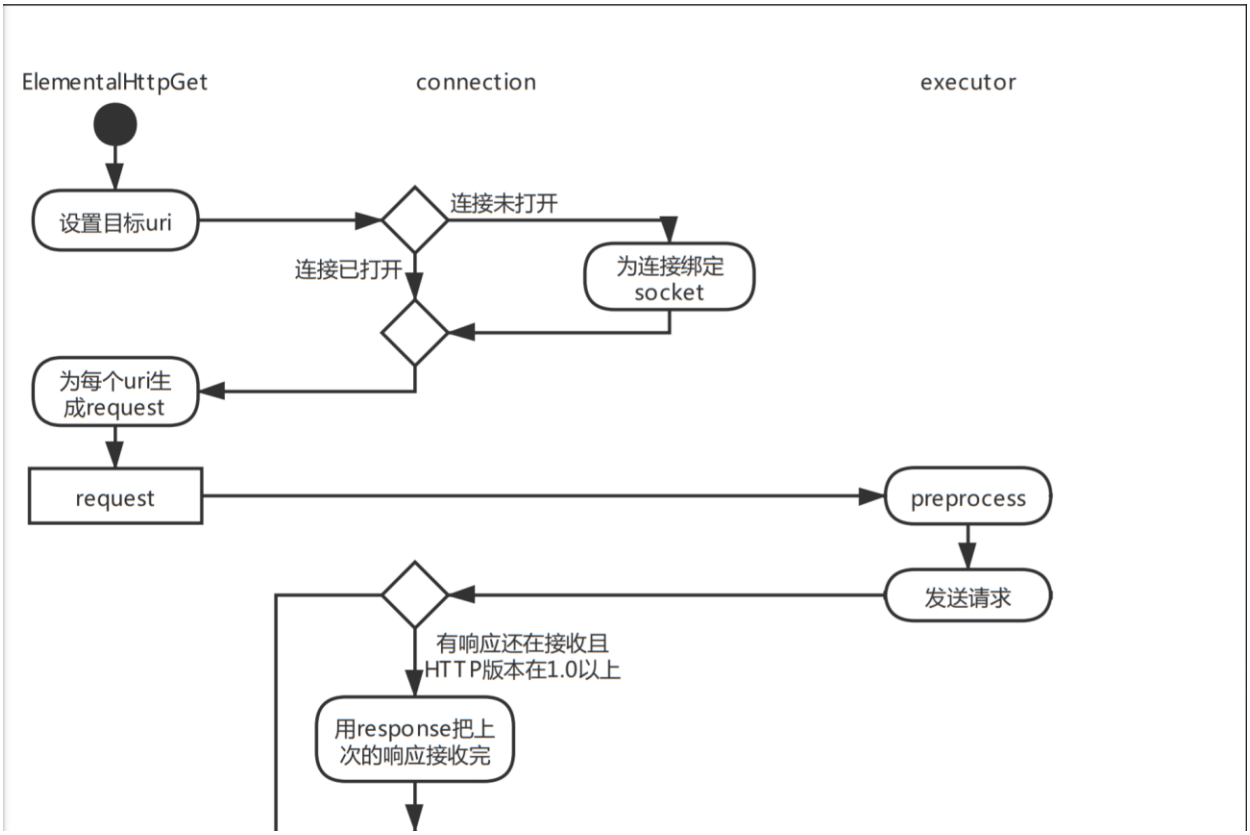
从上面的类图可以看出来连接 connection 涉及的类和接口要复杂的多，但是其实只要理清楚接口主要实现了什么功能，理解起来并没有看上去那么复杂。在官方给出的例程中，在外面使用到了 connection 的 3 个方法，一个是判断连接是否畅通的 isOpen，一个是将连接与 socket 绑定的 bind，还有一个是关闭连接的 close，这三个方法实现的功能还比较直观。Connection 核心功能的实现是在执行器的使用里。

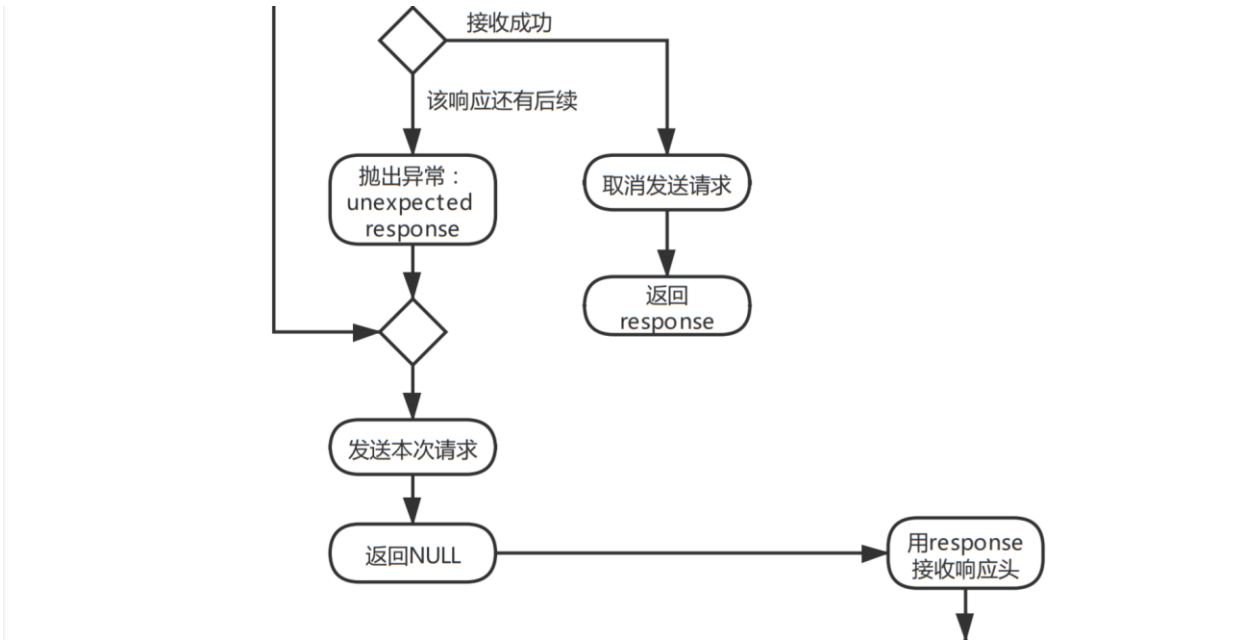
从上面的类图中，我们可以看到 connection 有发送请求和接收响应的方法，而这才是 connection 的主要功能。执行器调用.execute 方法的时候，会首先使用 executor 的内部方法 doSendRequest 并接收其返回值，如果返回值为 NULL 则 doReceiveResponse。这里面用到的两个内部方法的实现都是通过 connection 提供的上述接口实现的。而在 connection 内部，这些函数的具体实现就直接和字符流处理相关了。详情请看下一节的活动图。

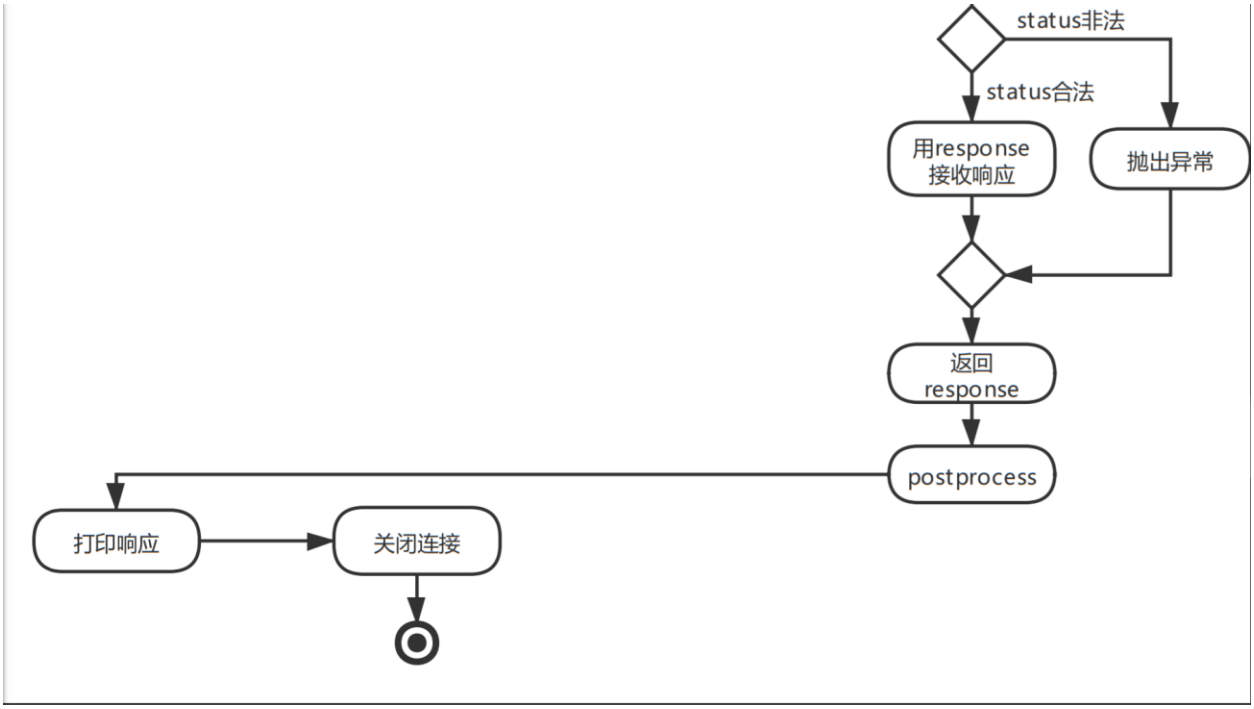


最后一个涉及到的类就是这个连接复用策略了，他的主要作用是记录连接在传输完成后是否保持打开的信息。在使用时会利用 keepAlive 来判断是否要保持连接 open，否则 close。

二、活动图


ElementalHttpGet
connection
executor
设置目标uri
连接未打开
连接已打开
为连接绑定
socket
为每个uri生
成request
request
preprocess
发送请求
有响应还在接收且
HTTP版本在1.0以上
用response把上
次的响应接收完


接收成功
该响应还有后续
抛出异常：
unexpected
response
取消发送请求
返回
response
发送本次请求
返回NULL
用response
接收响应头



第三部分：高级设计意图

一、整体架构

客户层：HttpClient

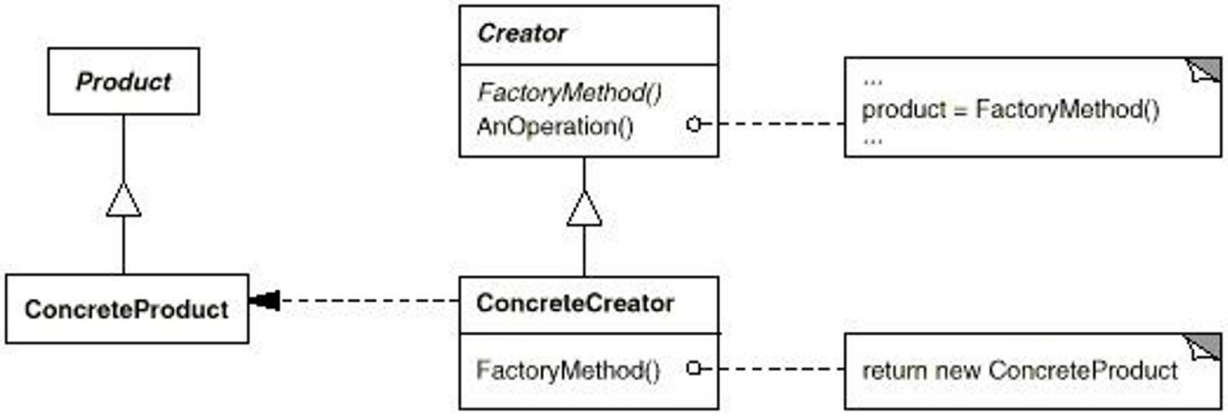
执行层：HttpExecutor，HttpMessageParser，HttpMessageWriter

连接层：Connection，ConnPool，Host

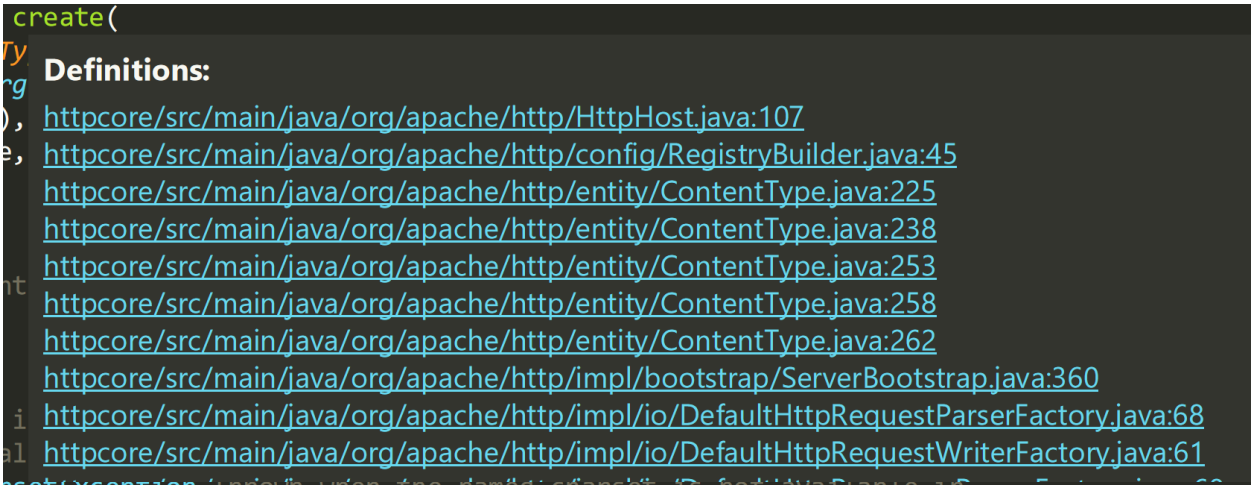
二、设计模式

1. 工厂方法模式（FactoryMethod）

结构:



工厂方法模式的一个典型特点是在类中创建对象，在 HttpComponents 中多处使用到了这种设计模式。



2. 适配器模式（Adapter）

适配器模式的作用是将一个类的接口转换成客户希望的另一个接口，使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。在 HttpComponents 中的一个典型例子是基本的读写函数 write 和 parser，这两个方法的功能非常简单，也是真正去完成收发功能的方法（其实就是对于

字符流的操作）。但是在用到他们的时候，connection 对其进行了包装（适配），帮助客户完成了参数的转换，降低了使用的复杂程度。

3. 代理模式（Proxy）

代理模式能够为其他对象提供一种代理以控制对这个对象的访问。在上面分析的例子中，executor 将这一点体现的淋漓尽致。在真正发送请求和接收响应的过程中，绝大部分方法都包含在 connection 中，但是用户看到的是 executor。对于 connection 的控制全部由 executor 完成，进一步降低了用户的使用难度。

4. 策略模式（Stategy）

策略模式是针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。在 HttpComponents 中，有两个向互联网请求数据的基本操作，Get 和 Post。在使用的时候，字符串”Get”和”Post”作为参数被传递给 BasicHttpRequest 以生成请求，之后 connection 在发送请求的时候会根据 request 的 method 来使用不同的算法。