# 解析

# httpcomponents-core-master

李欢洋
2016K8009929042

# 目录

# 1 简介

超文本传输协议(http)是目前互联网上极其普遍的传输协议,它为构建功能丰富，绚丽多彩的网页提供了强大的支持。构建一个网站，通常无需直接操作http协议，目前流行的WEB框架已经透明的将这些底层功能封装的很好了，如常见的J2EE，.NET，php等框架或语言。

除了作为网站系统的底层支撑，http同样可以在其它的一些场景中使用，如游戏服务器和客户端的传输协议、web service、 网络爬虫、HTTP代理、网站后台数据接口等。

HttpComponents Core简称HttpCore，包括用于构建客户机/代理/服务器端HTTP服务的一致API，用于构建同步和异步HTTP服务的一致API和一组基于阻塞(经典)和非阻塞(NIO) I/O模型的低级别组件
，可用于以最小的占用空间构建客户端和服务器端HTTP服务，支持两种I/O模型：阻塞I/O模型和和非阻塞I/O模型。上层组件(HttpComponents Client，HttpComponents AsyncClient)依赖此组件实现数据传输。

阻塞I/O模型基于基本的JAVA I/O实现，非阻塞模型基于JAVA NIO实现。阻塞I/O模型可能更适合于数据密集、低延迟的场景，而非阻塞模型可能更适合于高延迟的场景，在这些场景中，原始数据吞吐量不如以资源高效的方式处理数千个并发HTTP连接的能力重要。HttpCore目标在于，实现最基本的HTTP传输方面功能，在良好的性能和API的清晰表达之间做到平衡，满足小的(可预测的)内存占用和自包含库(JRE之外没有外部依赖项)

## 1.1 HTTP 报文

### 1.1.1 结构

HTTP 报文由头部和可选的内容体构成。HTTP 请求报文的头由请求行和头部字段的集合构成。HTTP 响应报文的头部由状态行和头部字段的集合构成。所有 HTTP 报文必须包含协议版本。一些 HTTP 报文可选地可以包含内容体。

HttpCore 定义了 HTTP 报文对象模型，它紧跟定义，而且提供对 HTTP 报文元素进行序列化（格式化）和反序列化（解析）的支持。

### 1.1.2 基本操作

#### 1.1.2.1 HTTP 请求报文

HTTP 请求是由客户端向服务器端发送的报文。报文的第一行包含应用于资源的方法，资源的标识符，和使用的协议版本。

```
HttpRequest request = new
BasicHttpRequest("GET", "/", HttpVersion.HTTP_1_1);
System.out.println(request.getRequestLine().getMethod());
System.out.println(request.getRequestLine().getUri());
System.out.println(request.getProtocolVersion());
System.out.println(request.getRequestLine().toString());
```

输出内容为：

```
GET
/
HTTP/1.1
GET / HTTP/1.1
```

#### 1.1.2.2 HTTP 响应报文

HTTP 响应是由服务器在收到和解释请求报文之后发回客户端的报文。报文的第一行包含了协议的版本，之后是数字状态码和相关的文本段。

```
HttpResponse response = new
BasicHttpResponse(HttpVersion.HTTP_1_1,HttpStatus.SC_OK, "OK");
System.out.println(response.getProtocolVersion());
System.out.println(response.getStatusLine().getStatusCode());
System.out.println(response.getStatusLine().getReasonPhrase());
System.out.println(response.getStatusLine().toString());
```

输出内容为：

HTTP/1.1

200

OK

HTTP/1.1 200 OK

1.1.2.3 HTTP 报文通用的属性和方法

HTTP 报文可以包含一些描述报文属性的头部信息，比如内容长度，内容类型等。HttpCore 提供方法来获取，添加，移除和枚举头部信息。

```
HttpResponse response = new
BasicHttpResponse(HttpVersion.HTTP_1_1, HttpStatus.SC_OK, "OK");
response.addHeader("Set-Cookie", "c1=a; path=/; domain=localhost");
response.addHeader("Set-Cookie", "c2=b; path=\"/\", c3=c;
domain=\"localhost\"");
Header h1 = response.getFirstHeader("Set-Cookie");
System.out.println(h1);
Header h2 = response.getLastHeader("Set-Cookie");
System.out.println(h2);
Header[] hs = response.getHeaders("Set-Cookie");
System.out.println(hs.length);
```

输出内容为：

```
Set-Cookie: c1=a; path=/; domain=localhost
Set-Cookie: c2=b; path="/", c3=c; domain="localhost"
```

HTTP 头部信息仅在有需要时进行标记化并放入独立的头部元素中。从 HTTP 连接中获取的 HTTP 头部信息被作为字符数组存储在内部而且仅当它们的属性被访问时才延迟解析。

1.1.3 HTTP 实体

HTTP 报文可以携带和请求或响应相关的内容实体。实体可以在一些请求和一些响应中发现,因为它们是可选的。使用了实体的请求被称为包含请求的实体。HTTP 规范定义了两种包含方法的实体:POST 和 PUT。响应通常期望是包含内容实体的。这个规则也有一些例外,比如对 HEAD 方法的响应,204 没有内容,304 没有修改,205 重置内容响应。

HttpCore 区分三种类型的实体，这是基于它们的内容是在哪里生成的：

streamed 流式：内容从流中获得，或者在运行中产生。特别是这种分类包含从 HTTP 响应中获取的实体。流式实体是不可重复生成的。

self-contained 自我包含式：内容在内存中或通过独立的连接或其它实体中获得。自我包含式的实体是可以重复生成的。这种类型的实体会经常用于封闭 HTTP 请求的实体。

wrapping 包装式：内容从另外一个实体中获得。

## 2. Get 请求

Get、Post 是最常见的获取网页内容的请求形式，当然，返回内容并非必须是 html 代码，任何的 xml、json 或文字字符串都可以作为返回内容。

下面是用 Get 请求获取一个 html 网页内容的代码

```
// (1) 创建 HttpGet 实例
HttpGet get = new HttpGet("http://www.126.com");
// (2) 使用 HttpClient 发送 get 请求，获得返回结果 HttpResponse
HttpClient http = new DefaultHttpClient();
HttpResponse response = http.execute(get);
// (3) 读取返回结果
if (response.getStatusLine().getStatusCode() == 200) {
    HttpEntity entity = response.getEntity();
    InputStream in = entity.getContent();
    readResponse(in);
}
```

（1）HttpGet 的实例就是一个 get 请求，构造函数只有一个字符串参数，即要获取的网页地址。另外一种构造形式是使用 URI 实例作为 HttpGet 的参数。HttpComponents 提供了 URIUtils 类，它的 createURI()返回一个 URI 实例。

（2）请求最后被 HttpClient 发送出去，new DefaultHttpClient()创建一个基本的 HttpClient 实例。由于底层是基于阻塞的 JAVA I/O 模型，执行 execute()的时间与具体请求的远程服务器和网络速度有关，在实际运行场景中应特别注意此问题。如果是在 tomcat 等环境中执行可能会造成线程等待，浪费服务器资源，或拒绝其它的连接。

（3）请求返回后就可以读取返回内容了，但服务器地址错误，或请求的页面不存在等问题都会让请求失败。为了确保得到了正确的响应首先应判断返回码是否正确。调用 response.getStatusLine()返回一个 StatusLine 的实例，此实例描述了一次请求的响应信息。一个成功响应的 StatusLine 实例本身包含如下信息：

HTTP/1.0 200 OK

HTTP/1.0：是请求协议和版本号

200：是响应码

StatusLine 的下面 2 个方法分别用于获取响应信息的各部分内容

getProtocolVersion()： 得到请求协议和协议版本号，如 HTTP/1.0

getStatusCode()：得到响应码，如 200

HttpEntity entity = response.getEntity()返回一个 HttpEntity 实例，进而调用 getContent()就得到了一个输入流。后面的事情应该很明确了。
readResponse()是一个自己写的读取输入流中字符串的方法，代码如下：

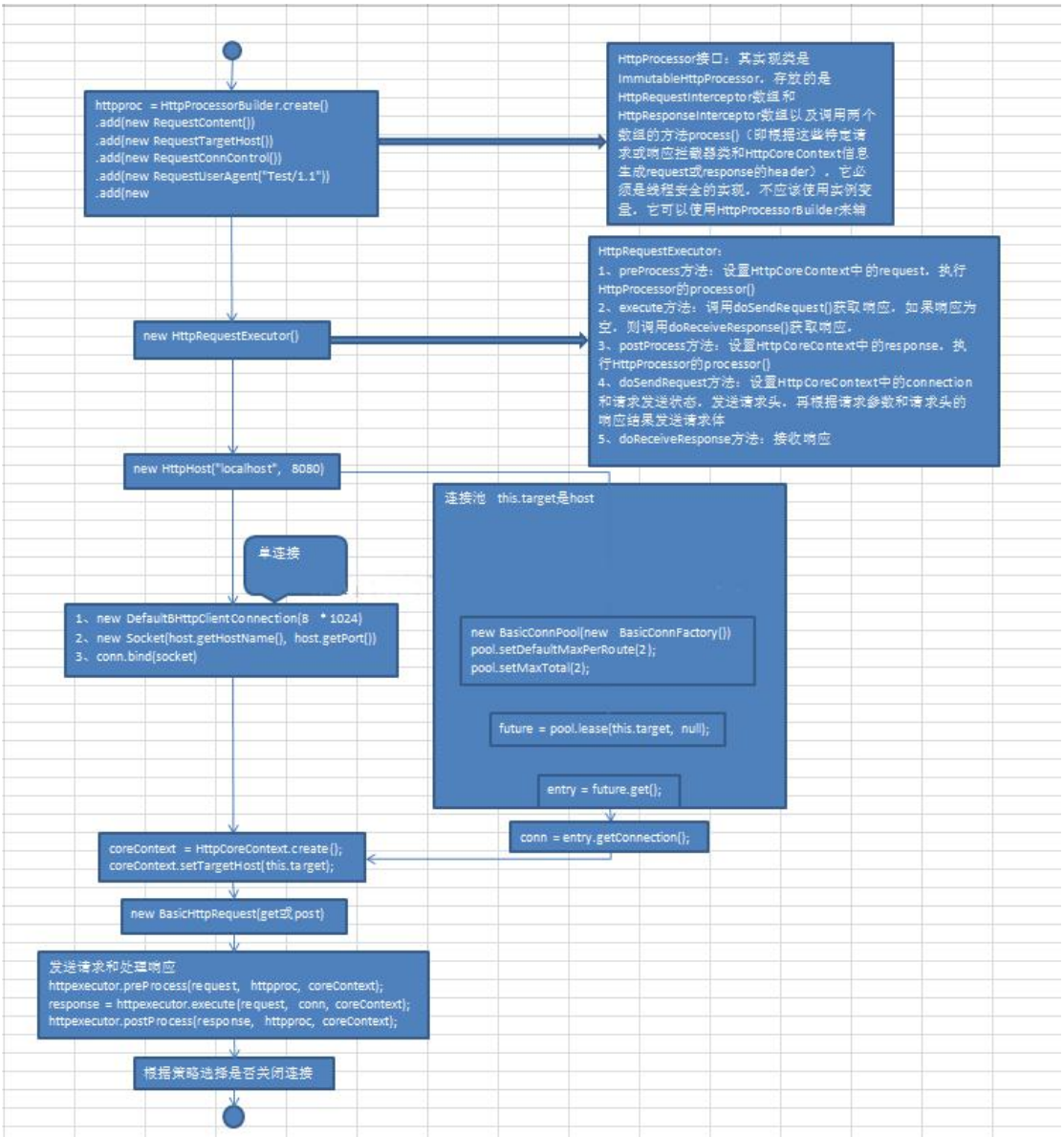
```
public static void readResponse(InputStream in) throws Exception{
    BufferedReader reader = new BufferedReader(new
    InputStreamReader(in));
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}
```

## 3. Post 请求

Post 请求在代码上与 Get 请求的主要区别是将 HttpGet 换成了 HttpPost，其余部分代码基本一致。

```
// (1) 创建 HttpGet 实例
<span style="color: #ff0000;">HttpPost post = new
HttpPost("http://www.126.com");</span>
// (2) 使用 HttpClient 发送 get 请求，获得返回结果 HttpResponse
HttpClient http = new DefaultHttpClient();
HttpResponse response = http.execute(<span style="color:
#ff0000;">post</span>);
// (3) 读取返回结果
if (response.getStatusLine().getStatusCode() == 200) {
    HttpEntity entity = response.getEntity();
    InputStream in = entity.getContent();
    readResponse(in);
}
```

4.HttpRequst 执行流程




HttpProcessor 是 HTTP protocol processor 的缩写，它的作用就是给我们写的 request 设置默认参数（比如协议版本，是否保持连接等），一般我们写的 HttpRequest 只有 url 和数据，事实上，一个完整的请求不只有这两个数据，查看一下 HTTP 协议就知道一个请求需要很多数据了，而之所以我们不用写这些数据，是因为 HttpCore 或 HttpClient 帮我们完成了，而负责这部分功能的就是 HttpProcessor。HttpRequestExecutor 的作用就是调用 HttpProcessor 完善原始 HttpRequest，根据请求参数使用 HttpClientConnection 执行 HttpRequest 并接受 HttpResponse，以及设置 HttpCoreContext 参数。HttpClientConnection

获取方法有两种：1、自己新建 conn 绑定根据 host 新建的 socket；2、新建 connpool，根据 host 从 pool 中获取 conn。HttpCoreContext：是为了完成会话功能

5.1 基于 ClassicGetExecutionExample 的分析

在 ClassicGetExecutionExample 这个 public 类中，

```
public static void main(String[] args) throws Exception {
```

告诉编译器这个方法中间可能有些地方要抛出异常，相应的，在代码最后对异常做了处理。

接下来，调用 RequesterBootstrap.bootstrap()方法创建了一个网站请求程序对象 httpRequester ，并对 ClassicGetExecutionExample 继承自其父类 examples 的 onRequestHead、onResponseHead、onExchangeComplete 三个方法进行重写.

```
HttpRequester httpRequester = RequesterBootstrap.bootstrap()
.setStreamListener(new Http1StreamListener() {

@Override
public void onRequestHead(final HttpConnection connection, final HttpRequest request) {
System.out.println(connection + " " + new RequestLine(request));
}
```

```
@Override
public void onResponseHead(final HttpConnection connection, final HttpResponse response) {
System.out.println(connection + " " + new StatusLine(response));
}
```

```
@Override
public void onExchangeComplete(final HttpConnection connection, final boolean keepAlive) {
if (keepAlive) {
System.out.println(connection + " exchange completed (connection kept alive)");
} else {
System.out.println(connection + " exchange completed (connection closed)");
}
}
```

```
})
```

接下来调用配置端口号方法，包括自定义设置、设置超时时间。

```
.setSocketConfig(SocketConfig.custom()
.setSoTimeout(5, TimeUnit.SECONDS)
.build())
```

调用 create()方法创建上下文。创建一个网站主机对象：目标（target）。创建一个串对象：路径（requestUris）。

```
HttpCoreContext coreContext = HttpCoreContext.create();
HttpHost target = new HttpHost("httpbin.org");
String[] requestUris = new String[] {"/", "/ip", "/user-agent", "/headers"};
```

最后，对路径长度建立循环，使用带参数的 try() {}语法

打印除去 host（域名或者 ip）部分的路径->页请求状态码，打印从响应中获得的消息实体。

```
for (int i = 0; i < requestUris.length; i++) {
String requestUri = requestUris[i];
ClassicHttpRequest request = new BasicClassicHttpRequest("GET", target, requestUri);
try (ClassicHttpResponse response = httpRequester.execute(target, request, Timeout.ofSeconds(5),
coreContext)) {
System.out.println(requestUri + "->" + response.getCode());
System.out.println(EntityUtils.toString(response.getEntity()));
System.out.println("==============");
}
}
```

5.2 基于 ClassicPostExecutionExample 的分析

抛出异常，调用 RequesterBootstrap.bootstrap()方法创建对象 ，对继承自其父类的三个方法进行重写等与 Get 实例类似之处就不过多赘述了

```
public class ClassicPostExecutionExample {
public static void main(String[] args) throws Exception {
HttpRequester httpRequester = RequesterBootstrap.bootstrap()
.setStreamListener(new Http1StreamListener() {...
}
```

```
})
.setSocketConfig(SocketConfig.custom()
.setSoTimeout(5, TimeUnit.SECONDS)
.build())
.create();
HttpCoreContext coreContext = HttpCoreContext.create();
HttpHost target = new HttpHost("httpbin.org");
```

自然地，更多的关注不同之处。这里创建了一个对象：请求体。属于 http 实体一类。其中包含新创建的串实体，比特队列实体，输入流实体，比特列输入流实体四个对象。

```
HttpEntity[] requestBodies = {
new StringEntity(
"This is the first test request",
ContentType.create("text/plain", StandardCharsets.UTF_8)),
new ByteArrayEntity(
"This is the second test request".getBytes(StandardCharsets.UTF_8),
ContentType.APPLICATION_OCTET_STREAM),
new InputStreamEntity(
new ByteArrayInputStream(
"This is the third test request (will be chunked)"
.getBytes(StandardCharsets.UTF_8)),
ContentType.APPLICATION_OCTET_STREAM)
};
```

### 5.3 基于 ClassicPostWithTrailersExecutionExample 的分析

与 ClassicPostExecutionExample 相比，ClassicPostWithTrailersExecutionExample 并没有对继承自其父类的三个方法进行重写：

```
public static void main(String[] args) throws Exception {
HttpRequester httpRequester = RequesterBootstrap.bootstrap()
.setSocketConfig(SocketConfig.custom()
.setSoTimeout(5, TimeUnit.SECONDS)
.build())
.create();
```

此外，另一个不同之处是在创建请求体这个对象时，只包含串实体和基态头。

```
HttpEntity requestBody = new HttpEntityWithTrailers(
new StringEntity("Chunked message with trailers", ContentType.TEXT_PLAIN),
new BasicHeader("t1","Hello world"));
```