httpclient4.4中文教程(初级篇)

kmg343

Published
with GitBook

目錄

httpclient4.4中文教程（初级篇）

超文本传输协议（HTTP）也许是当今互联网上使用的最重要的协议了。Web服务，有网络功能的设备和网络计算的发展，都持续扩展了HTTP协议的角色，超越了用户使用的Web浏览器范畴，同时，也增加了需要HTTP协议支持的应用程序的数量。

尽管java.net包提供了基本通过HTTP访问资源的功能，但它没有提供全面的灵活性和其它很多应用程序需要的功能。HttpClient就是寻求弥补这项空白的组件，通过提供一个有效的，保持更新的，功能丰富的软件包来实现客户端最新的HTTP标准和建议。

为扩展而设计，同时为基本的HTTP协议提供强大的支持，HttpClient组件也许就是构建HTTP客户端应用程序，比如web浏览器，web服务端，利用或扩展HTTP协议进行分布式通信的系统的开发人员的关注点。

1.HttpClient的范围

- 基于HttpCore[http://hc.apache.org/httpcomponents-core/index.html] 的客户端HTTP运输实现库
- 基于经典（阻塞）I/O
- 内容无关

2.什么是HttpClient不能做的

HttpClient不是一个浏览器。它是一个客户端的HTTP通信实现库。HttpClient的目标是发送和接收HTTP报文。HttpClient不会去缓存内容，执行嵌入在HTML页面中的javascript代码，猜测内容类型，重新格式化请求/重定向URI，或者其它和HTTP运输无关的功能。

First Chapter

GitBook allows you to organize your book into chapters, each chapter is stored in a separate file like this one.

1.1 执行请求

HttpClient最重要的功能是执行HTTP方法。一个HTTP方法的执行包含一个或多个HTTP请求/HTTP响应交换，通常由HttpClient的内部来处理。而期望用户提供一个要执行的请求对象，而HttpClient期望传输请求到目标服务器并返回对应的响应对象，或者当执行不成功时抛出异常。 很自然地，HttpClient API的主要切入点就是定义描述上述规约的HttpClient接口。

这里有一个很简单的请求执行过程的示例：

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpGet httpget = new HttpGet("http://localhost/");
CloseableHttpResponse response = httpclient.execute(httpget);
try {
<...>
} finally {
response.close();
}
```

1.1.1 HTTP请求

所有HTTP请求有一个组合了方法名，请求URI和HTTP协议版本的请求行。

HttpClient支持所有定义在HTTP/1.1版本中的HTTP方法：GET，HEAD，POST，PUT，DELETE，TRACE和OPTIONS。对于每个方法类型都有一个特殊的类：HttpGet，HttpHead，HttpPost，HttpPut，HttpDelete，HttpTrace和HttpOptions。

请求的URI是统一资源定位符，它标识了应用于哪个请求之上的资源。HTTP请求URI包含一个协议模式，主机名称，可选的端口，资源路径，可选的查询和可选的片段。

```
HttpGet httpget = new HttpGet(
"http://www.google.com/search?hl=en&q=httpclient&btnG=Google+Search&aq=f&oq=");
```

HttpClient提供很多工具方法来简化创建和修改执行URI。

URI也可以编程来拼装：

```
URI uri = new URIBuilder()
.setScheme("http")
.setHost("www.google.com")
.setPath("/search")
.setParameter("q", "httpclient")
.setParameter("btnG", "Google Search")
.setParameter("aq", "f")
.setParameter("oq", "")
.build();
HttpGet httpget = new HttpGet(uri);
System.out.println(httpget.getURI());
```

输出内容为：

```
http://www.google.com/search?q=httpclient&btnG=Google+Search&aq=f&oq=
```

1.1.2 HTTP响应

HTTP响应是由服务器在接收和解释请求报文之后返回发送给客户端的报文。响应报文的第一行包含了协议版本，之后是数字状态码和相关联的文本段。

```
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());
```

輸出内容为：

```
HTTP/1.1
200
OK
HTTP/1.1 200 OK
```

1.1.3 处理报文头部

一个HTTP报文可以包含很多描述如内容长度，内容类型等信息属性的头部信息。HttpClient提供获取，添加，移除和枚举头部信息的方法。

```
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"
2
```

获得给定类型的所有头部信息最有效的方式是使用HeaderIterator接口。

```
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\"");
HeaderIterator it = response.headerIterator("Set-Cookie");
while (it.hasNext()) {
System.out.println(it.next());
}
```

输出内容为：

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

它也提供解析HTTP报文到独立头部信息元素的方法方法。

```
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\"");
HeaderElementIterator it = new BasicHeaderElementIterator(
response.headerIterator("Set-Cookie"));
while (it.hasNext()) {
HeaderElement elem = it.nextElement();
System.out.println(elem.getName() + " = " + elem.getValue());
NameValuePair[] params = elem.getParameters();
for (int i = 0; i < params.length; i++) {
System.out.println(" " + params[i]);
}
}
```

输出内容为：

```
c1 = a
path=/
domain=localhost
c2 = b
path=/
c3 = c
domain=localhost
```

1.1.4. 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响应中找到，因为它们也是可选的。使用了实体的请求被称为封闭实体请求。HTTP规范定义了两种封闭实体的方法：POST和PUT。响应通常期望包含一个内容实体。这个规则也有特例，比如HEAD方法的响应和204 No Content，304 Not Modified和205 Reset Content响应。

HttpClient根据其内容出自何处区分三种类型的实体：

- 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。

当从一个HTTP响应中获取流式内容时，这个区别对于连接管理很重要。对于由应用程序创建而且只使用HttpClient发送的请求实体，流式和自我包含式的不同就不那么重要了。这种情况下，建议考虑如流式这种不能重复的实体，和可以重复的自我包含式实体。

1.1.4.1 重复实体

实体可以重复，意味着它的内容可以被多次读取。这就仅仅是自我包含式的实体了（像ByteArrayEntity或StringEntity）。

1.1.4.2 使用HTTP实体

因为一个实体既可以代表二进制内容又可以代表字符内容，它也支持字符编码（支持后者也就是字符内容）。

实体是当使用封闭内容执行请求，或当请求已经成功执行，或当响应体结果发功到客户端时创建的。

要从实体中读取内容，可以通过HttpEntity#getContent()方法从输入流中获取，这会返回一个java.io.InputStream对象，或者提供一个输出流到HttpEntity#writeTo(OutputStream)方法中，这会一次返回所有写入到给定流中的内容。 当实体通过一个收到的报文获取时，HttpEntity#getContentType()方法和HttpEntity#getContentLength()方法可以用来读取通用的元数据，如Content-Type和Content-Length头部信息（如果它们是可用的）。因为头部信息Content-Type可以包含对文本MIME类型的字符编码，比如text/plain或text/html，HttpEntity#getContentEncoding()方法用来读取这个信息。如果头部信息不可用，那么就返回长度-1，而对于内容类型返回NULL。如果头部信息Content-Type是可用的，那么就会返回一个Header对象。

当为一个传出报文创建实体时，这个元数据不得不通过实体创建器来提供。

```
StringEntity myEntity = new StringEntity("important message",
ContentType.create("text/plain", "UTF-8"));
System.out.println(myEntity.getContentType());
System.out.println(myEntity.getContentLength());
System.out.println(EntityUtils.toString(myEntity));
System.out.println(EntityUtils.toByteArray(myEntity).length);
```

输出内容为：

```
Content-Type: text/plain; charset=utf-8
17
important message
17
```

1.1.5 确保低级别资源释放

为了保障系统资源能够正常释放，需要关闭与内容流相关得实体与相应请求对象。

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpGet httpget = new HttpGet("http://localhost/");
CloseableHttpResponse response = httpclient.execute(httpget);
try {
HttpEntity entity = response.getEntity();
if (entity != null) {
InputStream instream = entity.getContent();
try {
// do something useful
} finally {
instream.close();
}
}
} finally {
response.close();
}
```

关闭返回内容和关闭请求的不同在于，前者在链接没有关闭得同时获取相关得数据内容，随后将关闭相关得链接。

请注意 HttpEntity#writeTo(OutputStream)方法也需要确认相关得系统资源已经释放掉，如果方法包含java.io.InputStream实例，例如调用HttpEntity#getContent()方法，他也需要在最后关闭数据流。

当我们利用流实体得时候，可以使用EntityUtils#consume(HttpEntity)方法，这样既可以在我们获取相关返回内容的时候，最后，相关得资源链接会被释放。

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpGet httpget = new HttpGet("http://localhost/");
CloseableHttpResponse response = httpclient.execute(httpget);
try {
HttpEntity entity = response.getEntity();
if (entity != null) {
InputStream instream = entity.getContent();
int byteOne = instream.read();
int byteTwo = instream.read();
// Do not need the rest
}
} finally {
response.close();
}
```

连接不会被重用，但是由它持有的所有级别的资源将会被正确释放。

1.1.6 消耗实体内容

推荐消耗实体内容的方式是使用它的HttpEntity#getContent()或HttpEntity#writeTo(OutputStream)方法。HttpClient也自带EntityUtils类，这会暴露出一些静态方法，这些方法可以更加容易地从实体中读取内容或信息。代替直接读取java.io.InputStream，也可以使用这个类中的方法以字符串/字节数组的形式获取整个内容体。然而，EntityUtils的使用是强烈不鼓励的，除非响应实体源自可靠的HTTP服务器和已知的长度限制。

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpGet httpget = new HttpGet("http://localhost/");
CloseableHttpResponse response = httpclient.execute(httpget);
try {
HttpEntity entity = response.getEntity();
if (entity != null) {
long len = entity.getContentLength();
if (len != -1 && len < 2048) {
System.out.println(EntityUtils.toString(entity));
} else {
// Stream content out
}
}
} finally {
response.close();
}
```

在一些情况下可能会不止一次的读取实体。此时实体内容必须以某种方式在内存或磁盘上被缓冲起来。最简单的方法是通过使用BufferedHttpEntity类来包装源实体完成。这会引起源实体内容被读取到内存的缓冲区中。在其它所有方式中，实体包装器将会得到源实体。

```
CloseableHttpResponse response = <...>
HttpEntity entity = response.getEntity();
if (entity != null) {
entity = new BufferedHttpEntity(entity);
}
```

1.1.7 生成实体内容

HttpClient提供一些类，它们可以用于生成通过HTTP连接获得内容的有效输出流。为了封闭实体从HTTP请求中获得的输出内容，那些类的实例可以和封闭如POST和PUT请求的实体相关联。HttpClient为很多公用的数据容器，比如字符串，字节数组，输入流和文件提供了一些类：StringEntity，ByteArrayEntity，InputStreamEntity和FileEntity。

```
File file = new File("somefile.txt");
FileEntity entity = new FileEntity(file,
ContentType.create("text/plain", "UTF-8"));
HttpPost httppost = new HttpPost("http://localhost/action.do");
httppost.setEntity(entity);
```

请注意InputStreamEntity是不可重复的，因为它仅仅能从低层数据流中读取一次内容。通常来说，我们推荐实现一个定制的HttpEntity类，这是自我包含式的，用来代替使用通用的InputStreamEntity。FileEntity也是一个很好的起点。

1.1.7.1 HTML表单

许多应用程序需要频繁模拟提交一个HTML表单的过程，比如，为了来记录一个Web应用程序或提交输出数据。HttpClient提供了特殊的实体类UrlEncodedFormEntity来这个满足过程。

```
List<NameValuePair> formparams = new ArrayList<NameValuePair>();
formparams.add(new BasicNameValuePair("param1", "value1"));
formparams.add(new BasicNameValuePair("param2", "value2"));
UrlEncodedFormEntity entity = new UrlEncodedFormEntity(formparams, Consts.UTF_8);
HttpPost httppost = new HttpPost("http://localhost/handler.do");
httppost.setEntity(entity);
```

UrlEncodedFormEntity实例将会使用URL编码来编码参数，生成如下的内容：

```
param1=value1&param2=value2
```

1.1.7.2 内容分块

通常，我们推荐让HttpClient选择基于被传递的HTTP报文属性的最适合的编码转换。这是可能的，但是，设置HttpEntity#setChunked()方法为true是通知HttpClient分块编码的首选。请注意HttpClient将会使用标识作为提示。当使用的HTTP协议版本，如HTTP/1.0版本，不支持分块编码时，这个值会被忽略。

```
StringEntity entity = new StringEntity("important message",
ContentType.create("plain/text", Consts.UTF_8));
entity.setChunked(true);
HttpPost httppost = new HttpPost("http://localhost/acrtion.do");
httppost.setEntity(entity);
```

1.1.8 响应控制器

控制响应的最简便和最方便的方式是使用ResponseHandler接口。这个放完完全减轻了用户关于连接管理的担心。当使用ResponseHandler时，HttpClient将会自动关注并保证释放连接到连接管理器中去，而不管请求执行是否成功或引发了异常。

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpGet httpget = new HttpGet("http://localhost/json");
ResponseHandler<MyJsonObject> rh = new ResponseHandler<MyJsonObject>() {
@Override
public JsonObject handleResponse(
final HttpResponse response) throws IOException {
StatusLine statusLine = response.getStatusLine();
HttpEntity entity = response.getEntity();
if (statusLine.getStatusCode() >= 300) {
throw new HttpResponseException(
statusLine.getStatusCode(),
statusLine.getReasonPhrase());
}
if (entity == null) {
throw new ClientProtocolException("Response contains no content");
}
Gson gson = new GsonBuilder().create();
ContentType contentType = ContentType.getOrDefault(entity);
Charset charset = contentType.getCharset();
Reader reader = new InputStreamReader(entity.getContent(), charset);
return gson.fromJson(reader, MyJsonObject.class);
}
};
MyJsonObject myjson = client.execute(httpget, rh);
```

1.2 httpclient接口

1.2.1. HttpClient 线程安全

HttpClient的实施预计将是线程安全的。我们建议同一个实例 可以用于多个请求执行。

1.2.2. HttpClient 资源释放

当一个CloseableHttpClient实例不再需要得时候，或者跳出connection manager得管理范围得时候，你必须关闭它，并调用CloseableHttpClient#close() 方法，它会释放相关资源。

1.3. HTTP 执行环境

最初，HTTP是被设计成无状态的，面向请求-响应的协议。然而，真实的应用程序经常需要通过一些逻辑相关的请求-响应交换来持久状态信息。为了开启应用程序来维持一个过程状态，HttpClient允许HTTP请求在一个特定的执行环境中来执行，简称为HTTP上下文。如果相同的环境在连续请求之间重用，那么多种逻辑相关的请求可以参与到一个逻辑会话中。HTTP上下文功能和java.util.Map很相似。它仅仅是任意命名参数值的集合。应用程序可以在请求之前或在检查上下文执行完成之后来填充上下文属性。

在HTTP请求执行的这一过程中，HttpClient添加了下列属性到执行上下文中：

- HttpConnection实例代表了连接到目标服务器的真实连接。
- HttpHost实例代表了链接得目标地址.
- HttpRoute 实例代表了链接路由
- HttpRequest实例代表了真实的HTTP请求。
- HttpResponse实例代表了真实的HTTP响应。
- RequestConfig 对象代表了实际得请求配置。
- java.util.List 对象代表请求执行的过程中收到的所有重定向位置的集合。

```
HttpContext context = <...>
HttpClientContext clientContext = HttpClientContext.adapt(context);
HttpHost target = clientContext.getTargetHost();
HttpRequest request = clientContext.getRequest();
HttpResponse response = clientContext.getResponse();
RequestConfig config = clientContext.getRequestConfig();
```

一个逻辑上得session实例会关联到多个request请求序列，在这些请求中其实是有相同得HttpContext，用以确保不同得请求之间共享上下文信息。

在接下来得一个例子中，通过初始request请求并保存相关信息到上下文中，并用来作为共享信息送到下一个请求中。

```
CloseableHttpClient httpclient = HttpClients.createDefault();
RequestConfig requestConfig = RequestConfig.custom()
.setSocketTimeout(1000)
.setConnectTimeout(1000)
.build();
HttpGet httpget1 = new HttpGet("http://localhost/1");
httpget1.setConfig(requestConfig);
CloseableHttpResponse response1 = httpclient.execute(httpget1, context);
try {
HttpEntity entity1 = response1.getEntity();
} finally {
response1.close();
}
```

```
HttpGet httpget2 = new HttpGet("http://localhost/2");
CloseableHttpResponse response2 = httpclient.execute(httpget2, context);
try {
HttpEntity entity2 = response2.getEntity();
} finally {
response2.close();
}
```

1.4. HTTP协议拦截器

HTTP协议拦截器是实现了HTTP协议的一个通常协议拦截器具体方面的程序。 协议拦截器还可以操纵附在邮件内容的实体 - 透明的内容压缩/解压就是一个很好的例子。几个协议拦截器可以被组合以形成一个逻辑单元。

这是如何局部上下文可用于持久连续之间的处理状态的一例 要求：

```
CloseableHttpClient httpclient = HttpClients.custom()
.addInterceptorLast(new HttpRequestInterceptor() {
public void process(
final HttpRequest request,
final HttpContext context) throws HttpException, IOException {
AtomicInteger count = (AtomicInteger) context.getAttribute("count");
request.addHeader("Count", Integer.toString(count.getAndIncrement()));
}
})
.build();
AtomicInteger count = new AtomicInteger(1);
HttpClientContext localContext = HttpClientContext.create();
localContext.setAttribute("count", count);
HttpGet httpget = new HttpGet("http://localhost/");
for (int i = 0; i < 10; i++) {
CloseableHttpResponse response = httpclient.execute(httpget, localContext);
try {
HttpEntity entity = response.getEntity();
} finally {
response.close();
}
}
```

1.5 异常处理

HttpClient能够抛出两种类型的异常：在I/O失败时，如套接字连接超时或被重置的java.io.IOException异常，还有标志HTTP请求失败的信号，如违反HTTP协议的HttpException异常。通常I/O错误被认为是非致命的和可以恢复的，而HTTP协议错误则被认为是致命的而且是不能自动恢复的。

1.5.1 HTTP运输安全

要理解HTTP协议并不是对所有类型的应用程序都适合的，这一点很重要。HTTP是一个简单的面向请求/响应的协议，最初被设计用来支持取回静态或动态生成的内容。它从未向支持事务性操作方向发展。比如，如果成功收到和处理请求，HTTP服务器将会考虑它的其中一部分是否完成，生成一个响应并发送一个状态码到客户端。如果客户端因为读取超时，请求取消或系统崩溃导致接收响应实体失败时，服务器不会试图回滚事务。如果客户端决定重新这个请求，那么服务器将不可避免地不止一次执行这个相同的事务。在一些情况下，这会导致应用数据损坏或者不一致的应用程序状态。

尽管HTTP从来都没有被设计来支持事务性处理，但它也能被用作于一个传输协议对关键的任务应用提供被满足的确定状态。要保证HTTP传输层的安全，系统必须保证HTTP方法在应用层的幂等性。

1.5.2 幂等的方法

1.5.3 异常自动恢复

默认情况下，HttpClient会试图自动从I/O异常中恢复。默认的自动恢复机制是受很少一部分已知的异常是安全的这个限制。

- HttpClient不会从任意逻辑或HTTP协议错误（那些是从HttpException类中派生出的）中恢复的。
- HttpClient将会自动重新执行那么假设是幂等的方法。
- HttpClient将会自动重新执行那些由于运输异常失败，而HTTP请求仍然被传送到目标服务器（也就是请求没有完全被送到服务器）失败的方法。
- HttpClient将会自动重新执行那些已经完全被送到服务器，但是服务器使用HTTP状态码（服务器仅仅丢掉连接而不会发回任何东西）响应时失败的方法。在这种情况下，假设请求没有被服务器处理，而应用程序的状态也没有改变。如果这个假设可能对于你应用程序的目标Web服务器来说不正确，那么就强烈建议提供一个自定义的异常处理器。

1.5.4 请求重试处理

为了开启自定义异常恢复机制，应该提供一个HttpRequestRetryHandler接口的实现。

```
HttpRequestRetryHandler myRetryHandler = new HttpRequestRetryHandler() {
public boolean retryRequest(
IOException exception,
int executionCount,
HttpContext context) {
if (executionCount >= 5) {
// Do not retry if over max retry count
return false;
}
if (exception instanceof InterruptedIOException) {
// Timeout
return false;
}
if (exception instanceof UnknownHostException) {
// Unknown host
return false;
}
if (exception instanceof ConnectTimeoutException) {
// Connection refused
return false;
}
if (exception instanceof SSLException) {
// SSL handshake exception
return false;
}
HttpClientContext clientContext = HttpClientContext.adapt(context);
HttpRequest request = clientContext.getRequest();
boolean idempotent = !(request instanceof HttpEntityEnclosingRequest);
if (idempotent) {
// Retry if the request is considered idempotent
return true;
}
return false;
}
};
CloseableHttpClient httpclient = HttpClients.custom()
.setRetryHandler(myRetryHandler)
.build();
```

1.6 中止请求

在一些情况下，由于目标服务器的高负载或客户端有很多活动的请求，那么HTTP请求执行会在预期的时间框内而失败。这时，就可能不得不过早地中止请求，解除封锁在I/O执行中的线程封锁。被HttpClient执行的HTTP请求可以在执行的任意阶段通过调用HttpUriRequest#abort()方法而中止。这个方法是线程安全的，而且可以从任意线程中调用。当一个HTTP请求被中止时，它的执行线程就封锁在I/O操作中了，而且保证通过抛出InterruptedIOException异常来解锁。

1.7 请求重定向

HttpClient能够自动支持所有类型的重定向请求。当POST请求返回状态为303的时候，我们就需要重定向的策略。

```
LaxRedirectStrategy redirectStrategy = new LaxRedirectStrategy();
CloseableHttpClient httpclient = HttpClients.custom()
.setRedirectStrategy(redirectStrategy)
.build();
```

HttpClient在我们进行执行请求处理的过程中会经常碰到请求的重定向。一般我们利用URIUtils#resolve的方法进行相关的调用。

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpClientContext context = HttpClientContext.create();
HttpGet httpget = new HttpGet("http://localhost:8080/");
CloseableHttpResponse response = httpclient.execute(httpget, context);
try {
HttpHost target = context.getTargetHost();
List<URI> redirectLocations = context.getRedirectLocations();
URI location = URIUtils.resolve(httpget.getURI(), target, redirectLocations);
System.out.println("Final HTTP location: " + location.toASCIIString());
// Expected to be an absolute URI
} finally {
response.close();
}
```