HttpClient 教程

南磊 译

目录

前言

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。

关于翻译

英文原版由 Oleg Kalnichevski 所著，若对 HttpClient 本身有问题的可以直接反馈到 Apache 官网 HttpClient 项目组。

第一章 基础

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：

```
HttpClient httpclient = new DefaultHttpClient();
HttpGet httpget = new HttpGet("http://localhost/");
HttpResponse response = httpclient.execute(httpget);
HttpEntity entity = response.getEntity();
if (entity != null) {
    InputStream instream = entity.getContent();
    int l;
    byte[] tmp = new byte[2048];
    while ((l = instream.read(tmp)) != -1) {
    }
}
```

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 = URIUtils.createURI("http", "www.google.com", -1,
"/search",
"q=httpclient&btnG=Google+Search&aq=f&oq=", null);
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=
```

查询字符串也可以从独立的参数中来生成：

```
List<NameValuePair> qparams = new ArrayList<NameValuePair>();
qparams.add(new BasicNameValuePair("q", "httpclient"));
qparams.add(new BasicNameValuePair("btnG", "Google Search"));
qparams.add(new BasicNameValuePair("aq", "f"));
qparams.add(new BasicNameValuePair("oq", null));
URI uri = URIUtils.createURI("http", "www.google.com", -1,
"/search",
URLEncodedUtils.format(qparams, "UTF-8"), null);
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());
```

输出内容为：

```
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",
"UTF-8");
System.out.println(myEntity.getContentType());
System.out.println(myEntity.getContentLength());
System.out.println(EntityUtils.getContentCharSet(myEntity));
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
UTF-8
important message
17
```

1.1.5 确保低级别资源释放

当完成一个响应实体，那么保证所有实体内容已经被完全消耗是很重要的，所以连接可

以安全的放回到连接池中，而且可以通过连接管理器对后续的请求重用连接。处理这个操作的最方便的方法是调用 HttpEntity#consumeContent()方法来消耗流中的任意可用内容。HttpClient 探测到内容流尾部已经到达后，会立即会自动释放低层连接，并放回到连接管理器。HttpEntity#consumeContent()方法调用多次也是安全的。

也可能会有特殊情况，当整个响应内容的一小部分需要获取，消耗剩余内容而损失性能，还有重用连接的代价太高，则可以仅仅通过调用 HttpUriRequest#abort()方法来中止请求。

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

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。

```
HttpGet httpget = new HttpGet("http://localhost/");
HttpResponse response = httpclient.execute(httpget);
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
    }
}
```

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。

```
HttpGet httpget = new HttpGet("http://localhost/");
HttpResponse response = httpclient.execute(httpget);
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, "text/plain;
charset=\"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 动态内容实体

通常来说，HTTP 实体需要基于特定的执行上下文来动态地生成。通过使用 EntityTemplate 实体类和 ContentProducer 接口，HttpClient 提供了动态实体的支持。内容生成器是按照需求生成它们内容的对象，将它们写入到一个输出流中。它们是每次被请求时来生成内容。所以用 EntityTemplate 创建的实体通常是自我包含而且可以重复的。

```
ContentProducer cp = new ContentProducer() {
public void writeTo(OutputStream outstream) throws IOException {
    Writer writer = new OutputStreamWriter(outstream, "UTF-8");
    writer.write("<response>");
    writer.write(" <content>");
    writer.write(" important stuff");
    writer.write(" </content>");
    writer.write("</response>");
    writer.flush();
    }
};
```

```
HttpEntity entity = new EntityTemplate(cp);
HttpPost httppost = new HttpPost("http://localhost/handler.do");
httppost.setEntity(entity);
```

1.1.7.2 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, "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
```

1.1.7.3 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",
"text/plain; charset=\"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。

```
HttpClient httpclient = new DefaultHttpClient();
HttpGet httpget = new HttpGet("http://localhost/");
ResponseHandler<byte[]> handler = new
ResponseHandler<byte[]>() {
   public byte[] handleResponse(
      HttpResponse response) throws ClientProtocolException,
   IOException {
      HttpEntity entity = response.getEntity();
      if (entity != null) {
         return EntityUtils.toByteArray(entity);
      } else {
         return null;
      }
   }
};
byte[] response = httpclient.execute(httpget, handler);
```

1.2 HTTP 执行的环境

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<String,Object>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：

- **'http.connection'**：HttpConnection实例代表了连接到目标服务器的真实连接。
- **'http.target_host'**：HttpHost实例代表了连接目标。
- **'http.proxy_host'**：如果使用了，HttpHost实例代表了代理连接。
- **'http.request'**：HttpRequest 实例代表了真实的 HTTP 请求。
- **'http.response'**：HttpResponse 实例代表了真实的 HTTP 响应。
- **'http.request_sent'**：java.lang.Boolean 对象代表了暗示真实请求是否被完全传送到目标连接的标识。

比如，为了决定最终的重定向目标，在请求执行之后，可以检查 http.target_host 属性的值：

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://www.google.com/");
HttpResponse response = httpclient.execute(httpget,
localContext);
HttpHost target = (HttpHost) localContext.getAttribute(
ExecutionContext.HTTP_TARGET_HOST);
```

```
System.out.println("Final target: " + target);
HttpEntity entity = response.getEntity();
if (entity != null) {
    entity.consumeContent();
}
```

输出内容为：

```
Final target: http://www.google.ch
```

1.3 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。

1.3.1 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。

1.3.2 幂等的方法

HTTP/1.1 明确地定义了幂等的方法，描述如下

[方法也可以有“幂等”属性在那些（除了错误或过期问题）N 的副作用>0 的相同请求和独立的请求是相同的]

换句话说，应用程序应该保证准备着来处理多个相同方法执行的实现。这是可以达到的，比如，通过提供一个独立的事务 ID 和其它避免执行相同逻辑操作的方法。

请注意这个问题对于 HttpClient 是不具体的。基于应用的浏览器特别受和非幂等的 HTTP 方法相关的相同问题的限制。

HttpClient 假设没有实体包含方法，比如 GET 和 HEAD 是幂等的，而实体包含方法，比如 POST 和 PUT 则不是。

1.3.3 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。

1.3.4 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。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpRequestRetryHandler myRetryHandler = new
HttpRequestRetryHandler() {
   public boolean retryRequest(IOException exception,
int executionCount,HttpContext context) {
   if (executionCount >= 5) {
      // 如果超过最大重试次数，那么就不要继续了
      return false;
   }
   if (exception instanceof NoHttpResponseException) {
      // 如果服务器丢掉了连接，那么就重试
      return true;
   }
   if (exception instanceof SSLHandshakeException) {
      // 不要重试SSL握手异常
      return false;
   }
   HttpRequest request = (HttpRequest) context.getAttribute(
ExecutionContext.HTTP_REQUEST);
   boolean idempotent = !(request instanceof
   HttpEntityEnclosingRequest);
   if (idempotent) {
      // 如果请求被认为是幂等的，那么就重试
      return true;
   }
```

```
    return false;
    }
};
httpclient.setHttpRequestRetryHandler(myRetryHandler);
```

1.4 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。

1.5 HTTP 协议拦截器

HTTP 协议拦截器是一个实现了特定 HTPP 协议方面的惯例。通常协议拦截器希望作用于一个特定头部信息上，或者一族收到报文的相关头部信息，或使用一个特定的头部或一族相关的头部信息填充发出的报文。协议拦截器也可以操纵包含在报文中的内容实体，透明的内容压缩/解压就是一个很好的示例。通常情况下这是由包装器实体类使用了“装饰者”模式来装饰原始的实体完成的。一些协议拦截器可以从一个逻辑单元中来结合。

协议拦截器也可以通过共享信息来共同合作-比如处理状态-通过 HTTP 执行上下文。协议拦截器可以使用 HTTP 内容来为一个或多个连续的请求存储一个处理状态。

通常拦截器执行的顺序不应该和它们基于的特定执行上下文状态有关。如果协议拦截器有相互依存关系，那么它们必须按特定顺序来执行，正如它们希望执行的顺序一样，它们应该在相同的序列中被加到协议处理器。

协议拦截器必须实现为线程安全的。和 Servlet 相似，协议拦截器不应该使用实例变量，除非访问的那些变量是同步的。

这个示例给出了本地内容在连续的请求中怎么被用于持久一个处理状态的：

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpContext localContext = new BasicHttpContext();
AtomicInteger count = new AtomicInteger(1);
localContext.setAttribute("count", count);
httpclient.addRequestInterceptor(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()));
    }
});
HttpGet httpget = new HttpGet("http://localhost/");
```

```
for (int i = 0; i < 10; i++) {
    HttpResponse response = httpclient.execute(httpget,
    localContext);
    HttpEntity entity = response.getEntity();
    if (entity != null) {
        entity.consumeContent();
    }
}
```

1.6 HTTP 参数

HttpParams 接口代表了定义组件运行时行为的一个不变的值的集合。很多情况下，HttpParams 和 HttpContext 相似。二者之间的主要区别是它们在运行时使用的不同。这两个接口表示了对象的集合，它们被视作为访问对象值的键的 Map，但是服务于不同的目的：

- HttpParams 旨在包含简单对象：整型，浮点型，字符串，集合，还有运行时不变的对象。
- HttpParams 希望被用在“一次写入-多处准备”模式下。HttpContext 旨在包含很可能在 HTTP 报文处理这一过程中发生改变的复杂对象
- HttpParams 的目标是定义其它组件的行为。通常每一个复杂的组件都有它自己的 HttpParams 对象。HttpContext 的目标是来表示一个 HTTP 处理的执行状态。通常相同的执行上下文在很多合作的对象中共享。

1.6.1 参数层次

在 HTTP 请求执行过程中，HttpRequest 对象的 HttpParams 是和用于执行请求的 HttpClient 实例的 HttpParams 联系在一起的。这使得设置在 HTTP 请求级别的参数优先于设置在 HTTP 客户端级别的 HttpParams。推荐的做法是设置普通参数对所有的在 HTTP 客户端级别的 HTTP 请求共享，而且可以选择性重写具体在 HTTP 请求级别的参数。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.getParams().setParameter(CoreProtocolPNames.PROTO
COL_VERSION,HttpVersion.HTTP_1_0);
httpclient.getParams().setParameter(CoreProtocolPNames.HTTP_
CONTENT_CHARSET,"UTF-8");
HttpGet httpget = new HttpGet("http://www.google.com/");
httpget.getParams().setParameter(CoreProtocolPNames.PROTOCOL
_VERSION,HttpVersion.HTTP_1_1);
httpget.getParams().setParameter(CoreProtocolPNames.USE_EXPE
CT_CONTINUE,Boolean.FALSE);
httpclient.addRequestInterceptor(new
HttpRequestInterceptor() {
```

```
    public void process(final HttpRequest request,
    final HttpContext context) throws HttpException, IOException {
        System.out.println(request.getParams().getParameter(
            CoreProtocolPNames.PROTOCOL_VERSION));
        System.out.println(request.getParams().getParameter(
            CoreProtocolPNames.HTTP_CONTENT_CHARSET));
        System.out.println(request.getParams().getParameter(
            CoreProtocolPNames.USE_EXPECT_CONTINUE));
        System.out.println(request.getParams().getParameter(
            CoreProtocolPNames.STRICT_TRANSFER_ENCODING));
    }
});
```

输出内容为：

```
HTTP/1.1
UTF-8
false
null
```

1.6.2 HTTP 参数 bean

HttpParams 接口允许在处理组件的配置上很大的灵活性。很重要的是，新的参数可以被引入而不会影响老版本的二进制兼容性。然而，和常规的 Java bean 相比，HttpParams 也有一个缺点：HttpParams 不能使用 DI 框架来组合。为了缓解这个限制，HttpClient 包含了一些 bean 类，它们可以用来按顺序使用标准的 Java eban 惯例初始化 HttpParams 对象。

```
HttpParams params = new BasicHttpParams();
HttpProtocolParamBean paramsBean = new
HttpProtocolParamBean(params);
paramsBean.setVersion(HttpVersion.HTTP_1_1);
paramsBean.setContentCharset("UTF-8");
paramsBean.setUseExpectContinue(true);

System.out.println(params.getParameter(
    CoreProtocolPNames.PROTOCOL_VERSION));
System.out.println(params.getParameter(
    CoreProtocolPNames.HTTP_CONTENT_CHARSET));
System.out.println(params.getParameter(
    CoreProtocolPNames.USE_EXPECT_CONTINUE));
System.out.println(params.getParameter(
    CoreProtocolPNames.USER_AGENT));
```

输出内容为：

```
HTTP/1.1
UTF-8
false
null
```

1.7 HTTP 请求执行参数

这些参数会影响到请求执行的过程：

- **'http.protocol.version'**：如果没有在请求对象中设置明确的版本信息，它就定义了使用的 HTTP 协议版本。这个参数期望得到一个 ProtocolVersion 类型的值。如果这个参数没有被设置，那么就使用 HTTP/1.1。
- **'http.protocol.element-charset'**：定义了编码 HTTP 协议元素的字符集。这个参数期望得到一个 java.lang.String 类型的值。如果这个参数没有被设置，那么就使用 US-ASCII。
- **'http.protocol.eontent-charset'**：定义了为每个内容主体编码的默认字符集。这个参数期望得到一个 java.lang.String 类型的值。如果这个参数没有被设置，那么就使用 ISO-8859-1。
- **'http.useragent'**：定义了头部信息 User-Agent 的内容。这个参数期望得到一个 java.lang.String 类型的值。如果这个参数没有被设置，那么 HttpClient 将会为它自动生成一个值。
- **'http.protocol.strict-transfer-encoding'**：定义了响应头部信息中是否含有一个非法的 Transfer-Encoding，都要拒绝掉。
- **'http.protocol.expect-continue'**：为包含方法的实体激活 Expect: 100-Continue 握手。Expect: 100-Continue 握手的目的是允许客户端使用请求体发送一个请求信息来决定源服务器是否希望在客户端发送请求体之前得到这个请求（基于请求头部信息）。Expect: 100-Continue 握手的使用可以对需要目标服务器认证的包含请求的实体（比如 POST 和 PUT）导致明显的性能改善。Expect: 100-Continue 握手应该谨慎使用，因为它和 HTTP 服务器，不支持 HTTP/1.1 协议的代理使用会引起问题。这个参数期望得到一个 java.lang.Boolean 类型的值。如果这个参数没有被设置，那么 HttpClient 将会试图使用握手。
- **'http.protocol.wait-for-continue'**：定义了客户端应该等待 100-Continue 响应最大的毫秒级时间间隔。这个参数期望得到一个 java.lang.Integer 类型的值。如果这个参数没有被设置，那么 HttpClient 将会在恢复请求体传输之前为确认等待 3 秒。

第二章 连接管理

HttpClient 有一个对连接初始化和终止，还有在活动连接上 I/O 操作的完整控制。而连接操作的很多方面可以使用一些参数来控制。

2.1 连接参数

这些参数可以影响连接操作：

- **'http.socket.timeout'**：定义了套接字的毫秒级超时时间（SO_TIMEOUT），这就是等待数据，换句话说，在两个连续的数据包之间最大的闲置时间。如果超时时间是 0 就解释为是一个无限大的超时时间。这个参数期望得到一个 java.lang.Integer 类型的值。如果这个参数没有被设置，那么读取操作就不会超时（无限大的超时时间）。
- **'http.tcp.nodelay'**：决定了是否使用 Nagle 算法。Nagle 算法视图通过最小化发送的分组数量来节省带宽。当应用程序希望降低网络延迟并提高性能时，它们可以关闭 Nagle 算法（也就是开启 TCP_NODELAY）。数据将会更早发送，增加了带宽消耗的成文。这个参数期望得到一个 java.lang.Boolean 类型的值。如果这个参数没有被设置，那么 TCP_NODELAY 就会开启（无延迟）。
- **'http.socket.buffer-size'**：决定了内部套接字缓冲使用的大小，来缓冲数据同时接收/传输 HTTP 报文。这个参数期望得到一个 java.lang.Integer 类型的值。如果这个参数没有被设置，那么 HttpClient 将会分配 8192 字节的套接字缓存。
- **'http.socket.linger'**：使用指定的秒数拖延时间来设置 SO_LINGER。最大的连接超时值是平台指定的。值 0 暗示了这个选项是关闭的。值-1 暗示了使用了 JRE 默认的。这个设置仅仅影响套接字关闭操作。如果这个参数没有被设置，那么就假设值为-1（JRE 默认）。
- **'http.connection.timeout'**：决定了直到连接建立时的毫秒级超时时间。超时时间的值为 0 解释为一个无限大的时间。这个参数期望得到一个 java.lang.Integer 类型的值。如果这个参数没有被设置，连接操作将不会超时（无限大的超时时间）。
- **'http.connection.stalecheck'**：决定了是否使用旧的连接检查。当在一个连接之上执行一个请求而服务器端的连接已经关闭时，关闭旧的连接检查可能导致在获得一个 I/O 错误风险时显著的性能提升（对于每一个请求，检查时间可以达到 30 毫秒）。这个参数期望得到一个 java.lang.Boolean 类型的值。出于性能的关键操作，检查应该被关闭。如果这个参数没有被设置，那么旧的连接将会在每个请求执行之前执行。
- **'http.connection.max-line-length'**：决定了最大请求行长度的限制。如果设置为一个正数，任何 HTTP 请求行超过这个限制将会引发 java.io.IOException 异常。负数或零将会关闭这个检查。这个参数期望得到一个 java.lang.Integer 类型的值。如果这个参数没有被设置，那么就不强制进行限制了。
- **'http.connection.max-header-count'**：决定了允许的最大 HTTP 头部信息数量。如果设置为一个正数，从数据流中获得的 HTTP 头部信息数量超过这个限制就会引发 java.io.IOException 异常。负数或零将会关闭这个检查。这个参数期望得到一个 java.lang.Integer 类型的值。如果这个参数没有被设置，那么就不

强制进行限制了。

- **'http.connection.max-status-line-garbage'**：决定了在期望得到 HTTP 响应状态行之前可忽略请求行的最大数量。使用 HTTP/1.1 持久性连接，这个问题产生的破碎的脚本将会返回一个错误的 Content-Length（有比指定的字节更多的发送）。不幸的是，在某些情况下，这个不能在错误响应后来侦测，只能在下一次之前。所以 HttpClient 必须以这种方式跳过那些多余的行。这个参数期望得到一个 java.lang.Integer 类型的值。0 是不允许在状态行之前的所有垃圾/空行。使用 java.lang.Integer#MAX_VALUE 来设置不限制的数字。如果这个参数没有被设置那就假设是不限制的。

2.2 持久连接

从一个主机向另外一个建立连接的过程是相当复杂的，而且包含了两个终端之间的很多包的交换，它是相当费时的。连接握手的开销是很重要的，特别是对小量的 HTTP 报文。如果打开的连接可以被重用来执行多次请求，那么就可以达到很高的数据吞吐量。

HTTP/1.1 强调 HTTP 连接默认情况可以被重用于多次请求。HTTP/1.0 兼容的终端也可以使用相似的机制来明确地交流它们的偏好来保证连接处于活动状态，也使用它来处理多个请求。HTTP 代理也可以保持空闲连接处于一段时间的活动状态，防止对相同目标主机的一个连接也许对随后的请求需要。保持连接活动的能力通常被称作持久性连接。HttpClient 完全支持持久性连接。

2.3 HTTP 连接路由

HttpClient 能够直接或通过路由建立连接到目标主机，这会涉及多个中间连接，也被称为跳。HttpClient 区分路由和普通连接，通道和分层。通道连接到目标主机的多个中间代理的使用也称作是代理链。

普通路由由连接到目标或仅第一次的代理来创建。通道路由通过代理链到目标连接到第一通道来建立。没有代理的路由不是通道的，分层路由通过已存在连接的分层协议来建立。协议仅仅可以在到目标的通道上或在没有代理的直接连接上分层。

2.3.1 路由计算

RouteInfo 接口代表关于最终涉及一个或多个中间步骤或跳的目标主机路由的信息。HttpRoute 是 RouteInfo 的具体实现，这是不能改变的（是不变的）。HttpTracker 是可变的 RouteInfo 实现，由 HttpClient 在内部使用来跟踪到最大路由目标的剩余跳数。HttpTracker 可以在成功执行向路由目标的下一跳之后更新。HttpRouteDirector 是一个帮助类，可以用来计算路由中的下一跳。这个类由 HttpClient 在内部使用。

HttpRoutePlanner 是一个代表计算到基于执行上下文到给定目标完整路由策略的接口。HttpClient 附带两个默认的 HttpRoutePlanner 实现。ProxySelectorRoutePlanner 是基于 java.net.ProxySelector 的。默认情况下，它会从系统属性中或从运行应用程序的浏览器中选取 JVM 的代理设置。DefaultHttpRoutePlanner 实现既不使用任何 Java 系统属性，也不使用系统或浏览器

的代理设置。它只基于 HTTP 如下面描述的参数计算路由。

2.3.2 安全 HTTP 连接

如果信息在两个不能由非认证的第三方进行读取或修改的终端之间传输，HTTP 连接可以被认为是安全的。SSL/TLS 协议是用来保证 HTTP 传输安全使用最广泛的技术。而其它加密技术也可以被使用。通常来说，HTTP 传输是在 SSL/TLS 加密连接之上分层的。

2.4 HTTP 路由参数

这些参数可以影响路由计算：

- **'http.route.default-proxy'**：定义可以被不使用 JRE 设置的默认路由规划者使用的代理主机。这个参数期望得到一个 HttpHost 类型的值。如果这个参数没有被设置，那么就会尝试直接连接到目标。
- **'http.route.local-address'**：定义一个本地地址由所有默认路由规划者来使用。有多个网络接口的机器中，这个参数可以被用于从连接源中选择网络接口。这个参数期望得到一个 java.net.InetAddress 类型的值。如果这个参数没有被设置，将会自动使用本地地址。
- **'http.route.forced-route'**：定义一个由所有默认路由规划者使用的强制路由。代替了计算路由，给定的强制路由将会被返回，尽管它指向一个完全不同的目标主机。这个参数期望得到一个 HttpRoute 类型的值。如果这个参数没有被设置，那么就使用默认的规则建立连接到目标服务器。

2.5 套接字工厂

HTTP 连接内部使用 java.net.Socket 对象来处理数据在线路上的传输。它们依赖 SocketFactory 接口来创建，初始化和连接套接字。这会使得 HttpClient 的用户可以提供在运行时指定套接字初始化代码的应用程序。PlainSocketFactory 是创建和初始化普通的（不加密的）套接字的默认工厂。

创建套接字的过程和连接到主机的过程是不成对的，所以套接字在连接操作封锁时可以被关闭。

```
PlainSocketFactory sf = PlainSocketFactory.getSocketFactory();
Socket socket = sf.createSocket();
HttpParams params = new BasicHttpParams();
params.setParameter(CoreConnectionPNames.CONNECTION_TIMEOUT,
1000L);
sf.connectSocket(socket, "locahost", 8080, null, -1, params);
```

2.5.1 安全套接字分层

LayeredSocketFactory 是 SocketFactory 接口的扩展。分层的套接字工厂可

以创建在已经存在的普通套接字之上的分层套接字。套接字分层主要通过代理来创建安全的套接字。HttpClient 附带实现了 SSL/TLS 分层的 SSLSocketFactory。请注意 HttpClient 不使用任何自定义加密功能。它完全依赖于标准的 Java 密码学（JCE）和安全套接字（JSEE）扩展。

2.5.2 SSL/TLS 的定制

HttpClient 使用 SSLSocketFactory 来创建 SSL 连接。SSLSocketFactory 允许高度定制。它可以使用 javax.net.ssl.SSLContext 的实例作为参数，并使用它来创建定制 SSL 连接。

```
TrustManager easyTrustManager = new X509TrustManager() {
   @Override
   public void checkClientTrusted(X509Certificate[] chain,
      String authType) throws CertificateException {
   // 哦，这很简单!
   }
   @Override
   public void checkServerTrusted(X509Certificate[] chain,
      String authType) throws CertificateException {
   //哦，这很简单!
   }
   @Override
   public X509Certificate[] getAcceptedIssuers() {
      return null;
   }
};
SSLContext sslcontext = SSLContext.getInstance("TLS");
sslcontext.init(null, new TrustManager[] { easyTrustManager },
null);
SSLSocketFactory sf = new SSLSocketFactory(sslcontext);
SSLSocket socket = (SSLSocket) sf.createSocket();
socket.setEnabledCipherSuites(new String[]
{ "SSL_RSA_WITH_RC4_128_MD5" });
HttpParams params = new BasicHttpParams();
params.setParameter(CoreConnectionPNames.CONNECTION_TIMEOUT,
1000L);
sf.connectSocket(socket, "locahost", 443, null, -1, params);
```

SSLSocketFactory的定制暗示出一定程度SSL/TLS协议概念的熟悉，这个详细的解释超出了本文档的范围。请参考Java的安全套接字扩展[http://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/JSSERefGuide.html]，这是javax.net.ssl.SSLContext和相关工具的详细描述。

2.5.3 主机名验证

除了信任验证和客户端认证在 SSL/TLS 协议级上进行，一旦连接建立之后，HttpClient 能可选地验证目标主机名匹配存储在服务器的 X.509 认证中的名字。这个认证可以提供额外的服务器信任材料的真实保证。X509 主机名验证接口代表了主机名验证的策略。HttpClient 附带了 3 个 X509 主机名验证器。很重要的一点是：主机名验证不应该混淆 SSL 信任验证。

- **StrictHostnameVerifier**：严格的主机名验证在 Sun Java 1.4，Sun Java 5 和 Sun Java 6 中是相同的。而且也非常接近 IE6。这个实现似乎是兼容 RFC 2818 处理通配符的。主机名必须匹配第一个 CN 或任意的 subject-alt。在 CN 和其它任意的 subject-alt 中可能会出现通配符。
- **BrowserCompatHostnameVerifier**：主机名验证器和 Curl 和 Firefox 的工作方式是相同的。主机名必须匹配第一个 CN 或任意的 subject-alt。在 CN 和其它任意的 subject-alt 中可能会出现通配符。BrowserCompatHostnameVerifier 和 StrictHostnameVerifier 的唯一不同是使用 BrowserCompatHostnameVerifier 匹配所有子域的通配符（比如"*.foo.com"），包括"a.b.foo.com"。
- **AllowAllHostnameVerifier**：这个主机名验证器基本上是关闭主机名验证的。这个实现是一个空操作，而且不会抛出 javax.net.ssl.SSLException 异常。

每一个默认的 HttpClient 使用 BrowserCompatHostnameVerifier 的实现。如果需要的话，它可以指定不同的主机名验证器实现。

```
SSLSocketFactory sf = new
SSLSocketFactory(SSLContext.getInstance("TLS"));
sf.setHostnameVerifier(SSLSocketFactory.STRICT_HOSTNAME_VERI
FIER);
```

2.6 协议模式

Scheme 类代表了一个协议模式，比如“http”或“https”同时包含一些协议属性，比如默认端口，用来为给定协议创建 java.net.Socket 实例的套接字工厂。SchemeRegistry 类用来维持一组 Scheme，当去通过请求 URI 建立连接时，HttpClient 可以从中选择：

```
Scheme http = new Scheme("http",
PlainSocketFactory.getSocketFactory(), 80);
SSLSocketFactory sf = new
SSLSocketFactory(SSLContext.getInstance("TLS"));
sf.setHostnameVerifier(SSLSocketFactory.STRICT_HOSTNAME_VERI
FIER);
Scheme https = new Scheme("https", sf, 443);
SchemeRegistry sr = new SchemeRegistry();
sr.register(http);
sr.register(https);
```

2.7 HttpClient 代理配置

尽管 HttpClient 了解复杂的路由模式和代理链，它仅支持简单直接的或开箱的跳式代理连接。

告诉 HttpClient 通过代理去连接到目标主机的最简单方式是通过设置默认的代理参数：

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpHost proxy = new HttpHost("someproxy", 8080);
httpclient.getParams().setParameter(ConnRoutePNames.DEFAULT_
PROXY, proxy);
```

也可以构建 HttpClient 使用标准的 JRE 代理选择器来获得代理信息：

```
DefaultHttpClient httpclient = new DefaultHttpClient();
ProxySelectorRoutePlanner routePlanner = new
ProxySelectorRoutePlanner(
httpclient.getConnectionManager().getSchemeRegistry(),
ProxySelector.getDefault());
httpclient.setRoutePlanner(routePlanner);
```

另外一种选择，可以提供一个定制的 RoutePlanner 实现来获得 HTTP 路由计算处理上的复杂的控制：

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.setRoutePlanner(new HttpRoutePlanner() {
   public HttpRoute determineRoute(HttpHost target,
      HttpRequest request,
      HttpContext context) throws HttpException {
         return new HttpRoute(target, null, new
      HttpHost("someproxy", 8080),
      "https".equalsIgnoreCase(target.getSchemeName()));
   }
});
```

2.8 HTTP 连接管理器

2.8.1 连接操作器

连接操作是客户端的低层套接字或可以通过外部实体，通常称为连接操作的被操作的状态的连接。OperatedClientConnection 接口扩展了 HttpClientConnection 接口而且定义了额外的控制连接套接字的方法。ClientConnectionOperator 接口代表了创建实例和更新那些对象低层套接字的策略。实现类最有可能利用 SocketFactory 来创建 java.net.Socket 实例。ClientConnectionOperator 接口可以让 HttpClient 的用户提供一个连接操作的定制策略和提供可选实现 OperatedClientConnection 接

口的能力。

2.8.2 管理连接和连接管理器

HTTP 连接是复杂的，有状态的，线程不安全的对象需要正确的管理以便正确地执行功能。HTTP 连接在同一时间仅仅只能由一个执行线程来使用。HttpClient 采用一个特殊实体来管理访问 HTTP 连接，这被称为 HTTP 连接管理器，代表了 ClientConnectionManager 接口。一个 HTTP 连接管理器的目的是作为工厂服务于新的 HTTP 连接，管理持久连接和同步访问持久连接来确保同一时间仅有一个线程可以访问一个连接。

内部的 HTTP 连接管理器和 OperatedClientConnection 实例一起工作，但是它们为服务消耗器 ManagedClientConnection 提供实例。ManagedClientConnection 扮演连接之上管理状态控制所有 I/O 操作的 OperatedClientConnection 实例的包装器。它也抽象套接字操作，提供打开和更新去创建路由套接字便利的方法。ManagedClientConnection 实例了解产生它们到连接管理器的链接，而且基于这个事实，当不再被使用时，它们必须返回到管理器。ManagedClientConnection 类也实现了 ConnectionReleaseTrigger 接口，可以被用来触发释放连接返回给管理器。一旦释放连接操作被触发了，被包装的连接从 ManagedClientConnection 包装器中脱离，OperatedClientConnection 实例被返回给管理器。尽管服务消耗器仍然持有 ManagedClientConnection 实例的引用，它也不再去执行任何 I/O 操作或有意无意地改变的 OperatedClientConnection 状态。

这里有一个从连接管理器中获取连接的示例：

```
HttpParams params = new BasicHttpParams();
Scheme http = new Scheme("http",
PlainSocketFactory.getSocketFactory(), 80);
SchemeRegistry sr = new SchemeRegistry();
sr.register(http);
ClientConnectionManager connMrg = new
SingleClientConnManager(params, sr);
// 请求新连接。这可能是一个很长的过程。
ClientConnectionRequest connRequest =
connMrg.requestConnection(
   new HttpRoute(new HttpHost("localhost", 80)), null);
// 等待连接10秒
ManagedClientConnection conn = connRequest.getConnection(10,
TimeUnit.SECONDS);
try {
   // 用连接在做有用的事情。当完成时释放连接。
   conn.releaseConnection();
} catch (IOException ex) {
   // 在I/O error之上终止连接。
   conn.abortConnection();
   throw ex;
}
```

如果需要，连接请求可以通过调用来ClientConnectionRequest#abortRequest()方法过早地中断。这会解锁在ClientConnectionRequest#getConnection()方法中被阻止的线程。

一旦响应内容被完全消耗后，BasicManagedEntity包装器类可以用来保证自动释放低层的连接。HttpClient内部使用这个机制来实现透明地对所有从HttpClient#execute()方法中获得响应释放连接：

```
ClientConnectionRequest connRequest =
connMrg.requestConnection(
new HttpRoute(new HttpHost("localhost", 80)), null);
ManagedClientConnection conn = connRequest.getConnection(10,
TimeUnit.SECONDS);
try {
   BasicHttpRequest request = new BasicHttpRequest("GET", "/");
   conn.sendRequestHeader(request);
   HttpResponse response = conn.receiveResponseHeader();
   conn.receiveResponseEntity(response);
   HttpEntity entity = response.getEntity();
   if (entity != null) {
      BasicManagedEntity managedEntity = new
      BasicManagedEntity(entity, conn, true);
      // 替换实体
      response.setEntity(managedEntity);
   }
   // 使用响应对象做有用的事情。当响应内容被消耗后这个连接将会自动释放。
} catch (IOException ex) {
   //在I/O error之上终止连接。
   conn.abortConnection();
   throw ex;
}
```

2.8.3 简单连接管理器

SingleClientConnManager是一个简单的连接管理器，在同一时间它仅仅维护一个连接。尽管这个类是线程安全的，但它应该被用于一个执行线程。SingleClientConnManager对于同一路由的后续请求会尽量重用连接。而如果持久连接的路由不匹配连接请求的话，它也会关闭存在的连接之后对给定路由再打开一个新的。如果连接已经被分配，将会抛出java.lang.IllegalStateException异常。

对于每个默认连接，HttpClient使用SingleClientConnManager。

2.8.4 连接池管理器

ThreadSafeClientConnManager是一个复杂的实现来管理客户端连接池，它也

可以从多个执行线程中服务连接请求。对每个基本的路由，连接都是池管理的。对于路由的请求，管理器在池中有可用的持久性连接，将被从池中租赁连接服务，而不是创建一个新的连接。

ThreadSafeClientConnManager 维护每个基本路由的最大连接限制。每个默认的实现对每个给定路由将会创建不超过两个的并发连接，而总共也不会超过 20 个连接。对于很多真实的应用程序，这个限制也证明很大的制约，特别是他们在服务中使用 HTTP 作为传输协议。连接限制，也可以使用 HTTP 参数来进行调整。

这个示例展示了连接池参数是如何来调整的：

```
HttpParams params = new BasicHttpParams();
// 增加最大连接到200
ConnManagerParams.setMaxTotalConnections(params, 200);
// 增加每个路由的默认最大连接到20
ConnPerRouteBean connPerRoute = new ConnPerRouteBean(20);
// 对localhost:80增加最大连接到50
HttpHost localhost = new HttpHost("locahost", 80);
connPerRoute.setMaxForRoute(new HttpRoute(localhost), 50);
ConnManagerParams.setMaxConnectionsPerRoute(params,
connPerRoute);
SchemeRegistry schemeRegistry = new SchemeRegistry();
schemeRegistry.register(
new Scheme("http", PlainSocketFactory.getSocketFactory(),
80));
schemeRegistry.register(
new Scheme("https", SSLSocketFactory.getSocketFactory(),
443));
ClientConnectionManager cm = new
ThreadSafeClientConnManager(params, schemeRegistry);
HttpClient httpClient = new DefaultHttpClient(cm, params);
```

2.8.5 连接管理器关闭

当一个 HttpClient 实例不再需要时，而且即将走出使用范围，那么关闭连接管理器来保证由管理器保持活动的所有连接被关闭，由连接分配的系统资源被释放是很重要的。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpGet httpget = new HttpGet("http://www.google.com/");
HttpResponse response = httpclient.execute(httpget);
HttpEntity entity = response.getEntity();
System.out.println(response.getStatusLine());
if (entity != null) {
   entity.consumeContent();
}
httpclient.getConnectionManager().shutdown();
```

2.9 连接管理参数

这些是可以用于定制标准 HTTP 连接管理器实现的参数：

- **'http.conn-manager.timeout'**：定义了当从 ClientConnectionManager 中检索 ManagedClientConnection 实例时使用的毫秒级的超时时间。这个参数期望得到一个 java.lang.Long 类型的值。如果这个参数没有被设置，连接请求就不会超时（无限大的超时时间）。
- **'http.conn-manager.max-per-route'**：定义了每个路由连接的最大数量。这个限制由客户端连接管理器来解释，而且应用于独立的管理器实例。这个参数期望得到一个 ConnPerRoute 类型的值。
- **'http.conn-manager.max-total'**：定义了总共连接的最大数目。这个限制由客户端连接管理器来解释，而且应用于独立的管理器实例。这个参数期望得到一个 java.lang.Integer 类型的值。

2.10 多线程执行请求

当配备连接池管理器时，比如 ThreadSafeClientConnManager，HttpClient 可以同时被用来执行多个请求，使用多线程执行。

ThreadSafeClientConnManager 将会分配基于它的配置的连接。如果对于给定路由的所有连接都被租出了，那么连接的请求将会阻塞，直到一个连接被释放回连接池。它可以通过设置'http.conn-manager.timeout'为一个正数来保证连接管理器不会在连接请求执行时无限期的被阻塞。如果连接请求不能在给定的时间周期内被响应，将会抛出 ConnectionPoolTimeoutException 异常。

```
HttpParams params = new BasicHttpParams();
SchemeRegistry schemeRegistry = new SchemeRegistry();
schemeRegistry.register(
new Scheme("http", PlainSocketFactory.getSocketFactory(), 80));
ClientConnectionManager cm = new
ThreadSafeClientConnManager(params, schemeRegistry);
HttpClient httpClient = new DefaultHttpClient(cm, params);
// 执行GET方法的URI
String[] urisToGet = {
   "http://www.domain1.com/",
   "http://www.domain2.com/",
   "http://www.domain3.com/",
   "http://www.domain4.com/"
};
// 为每个URI创建一个线程
GetThread[] threads = new GetThread[urisToGet.length];
for (int i = 0; i < threads.length; i++) {
   HttpGet httpget = new HttpGet(urisToGet[i]);
   threads[i] = new GetThread(httpClient, httpget);
}
```

```
// 开始执行线程
for (int j = 0; j < threads.length; j++) {
   threads[j].start();
}
// 合并线程
for (int j = 0; j < threads.length; j++) {
   threads[j].join();
}
```

```
static class GetThread extends Thread {
   private final HttpClient httpClient;
   private final HttpContext context;
   private final HttpGet httpget;
   public GetThread(HttpClient httpClient, HttpGet httpget) {
      this.httpClient = httpClient;
      this.context = new BasicHttpContext();
      this.httpget = httpget;
   }
   @Override
   public void run() {
      try {
         HttpResponse response =
      this.httpClient.execute(this.httpget, this.context);
         HttpEntity entity = response.getEntity();
         if (entity != null) {
            // 对实体做些有用的事情...
            // 保证连接能释放回管理器
            entity.consumeContent();
         }
      } catch (Exception ex) {
         this.httpget.abort();
      }
   }
}
```

2.11 连接收回策略

一个经典的阻塞 I/O 模型的主要缺点是网络套接字仅当 I/O 操作阻塞时才可以响应 I/O 事件。当一个连接被释放返回管理器时，它可以被保持活动状态而却不能监控套接字的状态和响应任何 I/O 事件。如果连接在服务器端关闭，那么客户端连接也不能去侦测连接状态中的变化和关闭本端的套接字去作出适当响应。

HttpClient 通过测试连接是否是过时的来尝试去减轻这个问题，这已经不再有效了，因为它已经在服务器端关闭了，之前使用执行 HTTP 请求的连接。过时的连接检查也并不是 100%

的稳定，反而对每次请求执行还要增加 10 到 30 毫秒的开销。唯一可行的而不涉及到每个对空闲连接的套接字模型线程解决方案，是使用专用的监控线程来收回因为长时间不活动而被认为是过期的连接。监控线程可以周期地调用 ClientConnectionManager#closeExpiredConnections() 方法来关闭所有过期的连接，从连接池中收回关闭的连接。它也可以选择性调用 ClientConnectionManager#closeIdleConnections() 方法来关闭所有已经空闲超过给定时间周期的连接。

```
public static class IdleConnectionMonitorThread extends Thread {
    private final ClientConnectionManager connMgr;
    private volatile boolean shutdown;
    public IdleConnectionMonitorThread(ClientConnectionManager
    connMgr) {
        super();
        this.connMgr = connMgr;
    }
    @Override
    public void run() {
        try {
            while (!shutdown) {
                synchronized (this) {
                    wait(5000);
                    // 关闭过期连接
                    connMgr.closeExpiredConnections();
                    // 可选地，关闭空闲超过30秒的连接
                    connMgr.closeIdleConnections(30,
                TimeUnit.SECONDS);
                }
            }
        } catch (InterruptedException ex) {
            // 终止
        }
    }
    public void shutdown() {
        shutdown = true;
        synchronized (this) {
            notifyAll();
        }
    }
}
```

2.12 连接保持活动的策略

HTTP 规范没有确定一个持久连接可能或应该保持活动多长时间。一些 HTTP 服务器使用

非标准的头部信息Keep-Alive来告诉客户端它们想在服务器端保持连接活动的周期秒数。如果这个信息可用，HttClient 就会利用这个它。如果头部信息 Keep-Alive 在响应中不存在，HttpClient 假设连接无限期的保持活动。然而许多现实中的 HTTP 服务器配置了在特定不活动周期之后丢掉持久连接来保存系统资源，往往这是不通知客户端的。如果默认的策略证明是过于乐观的，那么就会有人想提供一个定制的保持活动策略。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.setKeepAliveStrategy(new
ConnectionKeepAliveStrategy() {
public long getKeepAliveDuration(HttpResponse response,
HttpContext context) {
   // 兑现'keep-alive'头部信息
   HeaderElementIterator it = new BasicHeaderElementIterator(
      response.headerIterator(HTTP.CONN_KEEP_ALIVE));
   while (it.hasNext()) {
      HeaderElement he = it.nextElement();
      String param = he.getName();
      String value = he.getValue();
      if (value != null && param.equalsIgnoreCase("timeout")) {
         try {
            return Long.parseLong(value) * 1000;
         } catch(NumberFormatException ignore) {
         }
      }
   }
   HttpHost target = (HttpHost) context.getAttribute(
   ExecutionContext.HTTP_TARGET_HOST);
   if
   ("www.naughty-server.com".equalsIgnoreCase(target.getHostNa
   me())) {
      // 只保持活动5秒
      return 5 * 1000;
   } else {
      // 否则保持活动30秒
      return 30 * 1000;
   }
}
});
```

第三章 HTTP 状态管理

原始的 HTTP 是被设计为无状态的，面向请求/响应的协议，没有特殊规定有状态的，贯穿一些逻辑相关的请求/响应交换的会话。由于 HTTP 协议变得越来越普及和受欢迎，越来越多的从前没有打算使用它的系统也开始为应用程序来使用它，比如作为电子商务应用程序的传输方式。因此，支持状态管理就变得非常必要了。

网景公司，一度成为 Web 客户端和服务器软件开发者的领导方向，在它们基于专有规范的产品中实现了对 HTTP 状态管理的支持。之后，网景公司试图通过发布规范草案来规范这种机制。它们的努力通过 RFC 标准跟踪促成了这些规范定义。然而，在很多应用程序中的状态管理仍然基于网景公司的草案而不兼容官方的规范。很多主要的 Web 浏览器开发者觉得有必要保留那些极大促进标准片段应用程序的兼容性。

3.1 HTTP cookies

Cookie 是 HTTP 代理和目标服务器可以交流保持会话的状态信息的令牌或短包。网景公司的工程师用它来指“魔法小甜饼”和粘住的名字。

HttpClient 使用 Cookie 接口来代表抽象的 cookie 令牌。在它的简单形式中 HTTP 的 cookie 几乎是名/值对。通常一个 HTTP 的 cookie 也包含一些属性，比如版本号，合法的域名，指定 cookie 应用所在的源服务器 URL 子集的路径，cookie 的最长有效时间。

SetCookie接口代表由源服务器发送给HTTP代理的响应中的头部信息Set-Cookie来维持一个对话状态。SetCookie2接口和指定的Set-Cookie2方法扩展了SetCookie。

SetCookie 接口和额外的如获取原始 cookie 属性的能力，就像它们由源服务器指定的客户端特定功能扩展了 Cookie 接口。这对生成 Cookie 头部很重要，因为一些 cookie 规范需要。Cookie 头部应该包含在 Set-Cookie 或 Set-Cookie2 头部中指定的特定属性。

3.1.1 Cookie 版本

Cookie 兼容网景公司的草案标准，但是版本 0 被认为是不符合官方规范的。符合标准的 cookie 的期望版本是 1。HttpClient 可以处理基于不同版本的 cookie。

这里有一个重新创建网景公司草案 cookie 示例：

```
BasicClientCookie netscapeCookie = new
BasicClientCookie("name", "value");
netscapeCookie.setVersion(0);
netscapeCookie.setDomain(".mycompany.com");
netscapeCookie.setPath("/");
```

这是一个重新创建标准 cookie 的示例。要注意符合标准的 cookie 必须保留由源服务器发送的所有属性：

```
BasicClientCookie stdCookie = new BasicClientCookie("name",
"value");
stdCookie.setVersion(1);
stdCookie.setDomain(".mycompany.com");
stdCookie.setPath("/");
stdCookie.setSecure(true);
// 精确设置由服务器发送的属性
stdCookie.setAttribute(ClientCookie.VERSION_ATTR, "1");
stdCookie.setAttribute(ClientCookie.DOMAIN_ATTR,
".mycompany.com");
```

这是一个重新创建 Set-Cookie2 兼容 cookie 的实例。要注意符合标准的 cookie 必须保留由源服务器发送的所有属性：

```
BasicClientCookie2 stdCookie = new BasicClientCookie2("name",
"value");
stdCookie.setVersion(1);
stdCookie.setDomain(".mycompany.com");
stdCookie.setPorts(new int[] {80,8080});
stdCookie.setPath("/");
stdCookie.setSecure(true);
// 精确设置由服务器发送的属性
stdCookie.setAttribute(ClientCookie.VERSION_ATTR, "1");
stdCookie.setAttribute(ClientCookie.DOMAIN_ATTR,
".mycompany.com");
stdCookie.setAttribute(ClientCookie.PORT_ATTR, "80,8080");
```

3.2 Cookie 规范

CookieSpec 接口代表了 cookie 管理的规范。Cookie 管理规范希望如下几点：

- 解析的 Set-Cookie 规则还有可选的 Set-Cookie2 头部信息。
- 验证解析 cookie 的规则。
- 格式化给定主机的 Cookie 头部信息，原始端口和路径。

HttpClient 附带了一些 CookieSpec 的实现：

- **网景公司草案**：这个规范符合由网景通讯发布的原始草案规范。应当避免，除非有绝对的必要去兼容遗留代码。
- **RFC 2109**：官方 HTTP 状态管理规范并取代的老版本，被 RFC 2965 取代。
- **RFC 2965**：官方 HTTP 状态管理规范。
- **浏览器兼容性**：这个实现努力去密切模仿（mis）通用 Web 浏览器应用程序的实现。比如微软的 Internet Explorer 和 Mozilla 的 FireFox 浏览器。
- **最佳匹配**：'Meta'（元）cookie 规范采用了一些基于又 HTTP 响应发送的 cookie 格式的 cookie 策略。它基本上聚合了以上所有的实现到以一个类中。

强烈建议使用 Best Match 策略，让 HttpClient 在运行时基于执行上下文采用一些合适的兼容等级。

3.3 HTTP cookie 和状态管理参数

这些是用于定制 HTTP 状态管理和独立的 cookie 规范行为的参数。

- **'http.protocol.cookie-datepatterns'**：定义了用于解析非标准的 expires 属性的合法日期格式。只是对兼容不符合规定的，仍然使用网景公司草案定义的 expires 而不使用标准的 max-age 属性服务器需要。这个参数期望得到一个 java.util.Collection 类型的值。集合元素必须是 java.lang.String 类型，来兼容 java.text.SimpleDateFormat 的语法。如果这个参数没有被设置，那么默认的选择就是 CookieSpec 实现规范的值。要注意这个参数的应用。
- **'http.protocol.single-cookie-header'**：定义了是否 cookie 应该强制到一个独立的 Cookie 请求头部信息中。否则，每个 cookie 就被当作分离的 Cookie 头部信息来格式化。这个参数期望得到一个 java.lang.Boolean 类型的值。如果这个参数没有被设置，那么默认的选择就是 CookieSpec 实现规范的值。要注意这个参数仅仅严格应用于 cookie 规范（RFC 2109 和 RFC 2965）。浏览器兼容性和网景公司草案策略将会放置所有的 cookie 到一个请求头部信息中。
- **'http.protocol.cookie-policy'**：定义了用于 HTTP 状态管理的 cookie 规范的名字。这个参数期望得到一个 java.lang.String 类型的值。如果这个参数没有被设置，那么合法的日期格式就是 CookieSpec 实现规范的值。

3.4 Cookie 规范注册表

HttpClient 使用 CookieSpecRegistry 类维护一个可用的 cookie 规范注册表。下面的规范对于每个默认都是注册过的：

- **兼容性**：浏览器兼容性（宽松策略）。
- **网景**：网景公司草案。
- **rfc2109**：RFC 2109（过时的严格策略）。
- **rfc2965**：RFC 2965（严格策略的标准符合）。
- **best-match**：最佳匹配 meta（元）策略。

3.5 选择 cookie 策略

Cookie 策略可以在 HTTP 客户端被设置，如果需要，在 HTTP 请求级重写。

```
HttpClient httpclient = new DefaultHttpClient();
// 对每个默认的强制严格cookie策略
httpclient.getParams().setParameter(
ClientPNames.COOKIE_POLICY, CookiePolicy.RFC_2965);
HttpGet httpget = new
HttpGet("http://www.broken-server.com/");
// 对这个请求覆盖默认策略
httpget.getParams().setParameter(
ClientPNames.COOKIE_POLICY,
CookiePolicy.BROWSER_COMPATIBILITY);
```

3.6 定制 cookie 策略

为了实现定制 cookie 策略，我们应该创建 CookieSpec 接口的定制实现类，创建一个 CookieSpecFactory 实现来创建和初始化定制实现的实例并和 HttpClient 注册这个工厂。一旦定制实现被注册了，它可以和标准的 cookie 实现有相同的活性。

```
CookieSpecFactory csf = new CookieSpecFactory() {
   public CookieSpec newInstance(HttpParams params) {
      return new BrowserCompatSpec() {
         @Override
         public void validate(Cookie cookie, CookieOrigin
      origin)
         throws MalformedCookieException {
         // 这相当简单
         }
      };
   }
};
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.getCookieSpecs().register("easy", csf);
httpclient.getParams().setParameter(
ClientPNames.COOKIE_POLICY, "easy");
```

3.7 Cookie 持久化

HttpClient 可以和任意物理表示的实现了 CookieStore 接口的持久化 cookie 存储一起使用。默认的 CookieStore 实现称为 BasicClientCookie，这是凭借 java.util.ArrayList 的一个简单实现。在 BasicClientCookie 对象中存储的 cookie 当容器对象被垃圾回收机制回收时会丢失。如果需要，用户可以提供更复杂的实现。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
// 创建一个本地的cookie store实例
CookieStore cookieStore = new MyCookieStore();
// 如果需要填充cookie
BasicClientCookie cookie = new BasicClientCookie("name",
"value");
cookie.setVersion(0);
cookie.setDomain(".mycompany.com");
cookie.setPath("/");
cookieStore.addCookie(cookie);
// 设置存储
httpclient.setCookieStore(cookieStore);
```

3.8 HTTP 状态管理和执行上下文

在 HTTP 请求执行的过程中，HttpClient 添加了下列和状态管理相关的对象到执行上下文中：

- **'http.cookiespec-registry'**：CookieSpecRegistry 实例代表了实际的 cookie 规范注册表。这个属性的值设置在本地内容中，优先于默认的。
- **'http.cookie-spec'**：CookieSpec 实例代表真实的 cookie 规范。
- **'http.cookie-origin'**：CookieOrigin 实例代表了真实的源服务器的详细信息。
- **'http.cookie-store'**：CookieStore 实例代表了真实的 cookie 存储。设置在本地内容中的这个属性的值优先于默认的。

本地的 HttpContext 对象可以被用来定制 HTTP 状态管理内容，先于请求执行或在请求执行之后检查它的状态：

```
HttpClient httpclient = new DefaultHttpClient();
HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://localhost:8080/");
HttpResponse response = httpclient.execute(httpget,
localContext);
CookieOrigin cookieOrigin = (CookieOrigin)
localContext.getAttribute(
ClientContext.COOKIE_ORIGIN);
System.out.println("Cookie origin: " + cookieOrigin);
CookieSpec cookieSpec = (CookieSpec)
localContext.getAttribute(
ClientContext.COOKIE_SPEC);
System.out.println("Cookie spec used: " + cookieSpec);
```

3.9 每个用户/线程的状态管理

我们可以使用独立的本地执行上下文来实现对每个用户（或每个线程）状态的管理。定义在本地内容中的 cookie 规范注册表和 cookie 存储将会优先于设置在 HTTP 客户端级别中默认的那些。

```
HttpClient httpclient = new DefaultHttpClient();
// 创建cookie store的本地实例
CookieStore cookieStore = new BasicCookieStore();
// 创建本地的HTTP内容
HttpContext localContext = new BasicHttpContext();
// 绑定定制的cookie store到本地内容中
localContext.setAttribute(ClientContext.COOKIE_STORE,
cookieStore);
HttpGet httpget = new HttpGet("http://www.google.com/");
// 作为参数传递本地内容
HttpResponse response = httpclient.execute(httpget,
localContext);
```

第四章 HTTP 认证

HttpClient 提供对由 HTTP 标准规范定义的认证模式的完全支持。HttpClient 的认证框架可以扩展支持非标准的认证模式，比如 NTLM 和 SPNEGO。

4.1 用户凭证

任何用户身份验证的过程都需要一组可以用于建立用户身份的凭据。用户凭证的最简单的形式可以仅仅是用户名/密码对。UsernamePasswordCredentials 代表了一组包含安全规则和明文密码的凭据。这个实现对由 HTTP 标准规范中定义的标准认证模式是足够的

```
UsernamePasswordCredentials creds = new
UsernamePasswordCredentials("user", "pwd");
System.out.println(creds.getUserPrincipal().getName());
System.out.println(creds.getPassword());
```

输出内容为：

```
user
pwd
```

NTCredentials是微软Windows指定的实现，它包含了除了用户名/密码对外，一组额外的Windows指定的属性，比如用户域名的名字，比如在微软的Windows网络中，相同的用户使用不同设置的认证可以属于不同的域。

```
NTCredentials creds = new NTCredentials("user", "pwd",
"workstation", "domain");
System.out.println(creds.getUserPrincipal().getName());
System.out.println(creds.getPassword());
```

输出内容为：

```
DOMAIN/user
pwd
```

4.2 认证模式

AuthScheme 接口代表了抽象的，面向挑战-响应的认证模式。一个认证模式期望支持如下的功能：

- 解析和处理由目标服务器在对受保护资源请求的响应中发回的挑战。
- 提供处理挑战的属性：认证模式类型和它的参数，如果可用，比如这个认证模型可应用的领域。
- 对给定的凭证组和 HTTP 请求对响应真实认证挑战生成认证字符串。

要注意认证模式可能是有状态的，涉及一系列的挑战-响应交流。HttpClient 附带了一些

AuthScheme 实现：

- **Basic**（基本）：Basic 认证模式定义在 RFC 2617 中。这个认证模式是不安全的，因为凭据以明文形式传送。尽管它不安全，如果用在和 TLS/SSL 加密的组合中，Basic 认证模式是完全够用的。
- **Digest**（摘要）：Digest 认证模式定义在 RFC 2617 中。Digest 认证模式比 Basic 有显著的安全提升，对不想通过 TLS/SL 加密在完全运输安全上开销的应用程序来说也是很好的选择。
- **NTLM**：NTLM 是一个由微软开发的优化 Windows 平台的专有认证模式。NTLM 被认为是比 Digest 更安全的模式。这个模式需要外部的 NTLM 引擎来工作。要获取更多详情请参考包含在 HttpClient 发布包中的 NTLM_SUPPORT.txt 文档。

4.3 HTTP 认证参数

有一些可以用于定制 HTTP 认证过程和独立认证模式行为的参数：

- **'http.protocol.handle-authentication'**：定义了是否认证应该被自动处理。这个参数期望的得到一个 java.lang.Boolean 类型的值。如果这个参数没有被设置，HttpClient 将会自动处理认证。
- **'http.auth.credential-charset'**：定义了当编码用户凭证时使用的字符集。这个参数期望得到一个 java.lang.String 类型的值。如果这个参数没有被设置，那么就会使用 US-ASCII。

4.4 认证模式注册表

HttpClient 使用 AuthSchemeRegistry 类维护一个可用的认证模式的注册表。对于每个默认的下面的模式是注册过的：

- **Basic**：基本认证模式
- **Digest**：摘要认证模式

请注意 NTLM 模式*没有*对每个默认的进行注册。NTLM 不能对每个默认开启是应为许可和法律上的原因。要获取更详细的关于如何开启 NTLM 支持的内容请看这部分。

4.5 凭据提供器

凭据提供器意来维护一组用户凭据，还有能够对特定认证范围生产用户凭据。认证范围包括主机名，端口号，领域名称和认证模式名称。当使用凭据提供器来注册凭据时，我们可以提供一个通配符（任意主机，任意端口，任意领域，任意模式）来替代确定的属性值。如果直接匹配没有发现，凭据提供器期望被用来发现最匹配的特定范围。

HttpClient 可以和任意实现了 CredentialsProvider 接口的凭据提供器的物理代表一同工作。默认的 CredentialsProvider 实现被称为 BasicCredentialsProvider，它是简单的凭借 java.util.HashMap 的实现。

```
CredentialsProvider credsProvider = new
BasicCredentialsProvider();
credsProvider.setCredentials(
new AuthScope("somehost", AuthScope.ANY_PORT),
new UsernamePasswordCredentials("u1", "p1"));
credsProvider.setCredentials(
new AuthScope("somehost", 8080),
new UsernamePasswordCredentials("u2", "p2"));
credsProvider.setCredentials(
new AuthScope("otherhost", 8080, AuthScope.ANY_REALM, "ntlm"),
new UsernamePasswordCredentials("u3", "p3"));
System.out.println(credsProvider.getCredentials(
new AuthScope("somehost", 80, "realm", "basic")));
System.out.println(credsProvider.getCredentials(
new AuthScope("somehost", 8080, "realm", "basic")));
System.out.println(credsProvider.getCredentials(
new AuthScope("otherhost", 8080, "realm", "basic")));
System.out.println(credsProvider.getCredentials(
new AuthScope("otherhost", 8080, null, "ntlm")));
```

输出内容为：

```
[principal: u1]
[principal: u2]
null
[principal: u3]
```

4.6 HTTP 认证和执行上下文

HttpClient 依赖于 AuthState 类来跟踪关于认证过程状态的详细信息。在 HTTP 请求执行过程中，HttpClient 创建 2 个 AuthState 的实例：一个对于目标主机认证，另外一个对于代理认证。如果目标服务器或代理需要用户认证，那么各自的 AuthState 实例将会被在认证处理过程中使用的 AuthScope，AuthScheme 和 Crednetials 来填充。AuthState 可以被检查来找出请求的认证是什么类型的，是否匹配 AuthScheme 的实现，是否凭据提供器对给定的认证范围去找用户凭据。

在 HTTP 请求执行的过程中，HttpClient 添加了下列和认证相关的对象到执行上下文中：

'http.authscheme-registry'：AuthSchemeRegistry 实例代表真实的认证模式注册表。在本地内容中设置的这个属性的值优先于默认的。

'http.auth.credentials-provider'：CookieSpec 实例代表了真实的凭据提供器。在本地内容中设置的这个属性的值优先于默认的。

'http.auth.target-scope'：AuthState 实例代表了真实的目标认证状态。在本地内容中设置的这个属性的值优先于默认的。

'http.auth.proxy-scope'：AuthState 实例代表了真实的代理认证状态。在本地内容中设置的这个属性的值优先于默认的。

本地的 HttpContext 对象可以用于定制 HTTP 认证内容，并先于请求执行或在请求被执行之后检查它的状态：

```
HttpClient httpclient = new DefaultHttpClient();
HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://localhost:8080/");
HttpResponse response = httpclient.execute(httpget,
localContext);
AuthState proxyAuthState = (AuthState)
localContext.getAttribute(
ClientContext.PROXY_AUTH_STATE);
System.out.println("Proxy auth scope: " +
proxyAuthState.getAuthScope());
System.out.println("Proxy auth scheme: " +
proxyAuthState.getAuthScheme());
System.out.println("Proxy auth credentials: " +
proxyAuthState.getCredentials());
AuthState targetAuthState = (AuthState)
localContext.getAttribute(
ClientContext.TARGET_AUTH_STATE);
System.out.println("Target auth scope: " +
targetAuthState.getAuthScope());
System.out.println("Target auth scheme: " +
targetAuthState.getAuthScheme());
System.out.println("Target    auth    credentials:    "    +
targetAuthState.getCredentials());
```

4.7 抢占认证

HttpClient 不支持开箱的抢占认证，因为滥用或重用不正确的抢占认证可能会导致严重的安全问题，比如将用户凭据以明文形式发送给未认证的第三方。因此，用户期望评估抢占认证和在它们只能应用程序环境内容安全风险潜在的好处，而且要求使用如协议拦截器的标准 HttpClient 扩展机制添加对抢占认证的支持。

这是一个简单的协议拦截器，如果没有企图认证，来抢先引入 BasicScheme 的实例到执行上下文中。请注意拦截器必须在标准认证拦截器之前加入到协议处理链中。

```
HttpRequestInterceptor preemptiveAuth = new
HttpRequestInterceptor() {
public void process(final HttpRequest request,
final HttpContext context) throws HttpException, IOException {
    AuthState authState = (AuthState) context.getAttribute(
    ClientContext.TARGET_AUTH_STATE);
    CredentialsProvider credsProvider = (CredentialsProvider)
    context.getAttribute(ClientContext.CREDS_PROVIDER);
```

```
    HttpHost targetHost = (HttpHost) context.getAttribute(
    ExecutionContext.HTTP_TARGET_HOST);
    // 如果没有初始化auth模式
    if (authState.getAuthScheme() == null) {
        AuthScope authScope = new AuthScope(
        targetHost.getHostName(),
        targetHost.getPort());
        // 获得匹配目标主机的凭据
        Credentials creds =
    credsProvider.getCredentials(authScope);
        // 如果发现了，抢先生成BasicScheme
        if (creds != null) {
            authState.setAuthScheme(new BasicScheme());
            authState.setCredentials(creds);
        }
    }
}
};
DefaultHttpClient httpclient = new DefaultHttpClient();
// 作为第一个拦截器加入到协议链中
httpclient.addRequestInterceptor(preemptiveAuth, 0);
```

4.8 NTLM 认证

当前 HttpClient 没有提对开箱的 NTLM 认证模式的支持也可能永远也不会。这个原因是法律上的而不是技术上的。然而，NTLM 认证可以使用外部的 NTLM 引擎比如 JCIFS[http://jcifs.samba.org/]来开启，类库由 Samba[http://www.samba.org/]项目开发，作为它们 Windows 的交互操作程序套装的一部分。要获取详细内容请参考 HttpClient 发行包中包含的 NTLM_SUPPORT.txt 文档。

4.8.1 NTLM 连接持久化

NTLM 认证模式是在计算开销方面昂贵的多的，而且对标准的 Basic 和 Digest 模式的性能影响也很大。这很可能是为什么微软选择 NTLM 认证模式为有状态的主要原因之一。也就是说，一旦认证通过，用户标识是和连接的整个生命周期相关联的。NTLM 连接的状态特性使得连接持久化非常复杂，对于明显的原因，持久化 NTLM 连接不能被使用不同用户标识的用户重用。标准的连接管理器附带 HttpClient 是完全能够管理状态连接的。而逻辑相关的，使用同一 session 和执行上下文为了让它们了解到当前的用户标识的请求也是极为重要的。否则，HttpClient 将会终止对每个基于 NTLM 保护资源的 HTTP 请求创建新的 HTTP 连接。要获取关于有状态的 HTTP 连接的详细讨论，请参考这个部分。

因为 NTLM 连接是有状态的，通常建议使用相对简单的方法触发 NTLM 认证，比如 GET 或 HEAD，而重用相同的连接来执行代价更大的方法，特别是它们包含请求实体，比如 POST

或 PUT。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
NTCredentials creds = new NTCredentials("user", "pwd",
"myworkstation", "microsoft.com");
httpclient.getCredentialsProvider().setCredentials(AuthScope
.ANY, creds);
HttpHost target = new HttpHost("www.microsoft.com", 80,
"http");
// 保证相同的内容来用于执行逻辑相关的请求
HttpContext localContext = new BasicHttpContext();
// 首先执行简便的方法。这会触发NTLM认证
HttpGet httpget = new HttpGet("/ntlm-protected/info");
HttpResponse response1 = httpclient.execute(target, httpget,
localContext);
HttpEntity entity1 = response1.getEntity();
if (entity1 != null) {
   entity1.consumeContent();
}
//之后使用相同的内容（和连接）执行开销大的方法。
HttpPost httppost = new HttpPost("/ntlm-protected/form");
httppost.setEntity(new StringEntity("lots and lots of data"));
HttpResponse response2 = httpclient.execute(target, httppost,
localContext);
HttpEntity entity2 = response2.getEntity();
if (entity2 != null) {
   entity2.consumeContent();
}
```

第五章 HTTP 客户端服务

5.1 HttpClient 门面

HttpClient 接口代表了最重要的 HTTP 请求执行的契约。它没有在请求执行处理上强加限制或特殊细节，而在连接管理，状态管理，认证和处理重定向到具体实现上留下了细节。这应该使得很容易使用额外的功能，比如响应内容缓存来装饰接口。

DefaultHttpClient 是 HttpClient 接口的默认实现。这个类扮演了很多特殊用户程序或策略接口实现负责处理特定 HTTP 协议方面，比如重定向到处理认证或做出关于连接持久化和保持活动的持续时间决定的门面。这使得用户可以选择使用定制，具体程序等来替换某些方面默认实现。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
   httpclient.setKeepAliveStrategy(new
   DefaultConnectionKeepAliveStrategy() {
      @Override
      public long getKeepAliveDuration(HttpResponse response,
         HttpContext context) {
         long keepAlive =
      super.getKeepAliveDuration(response, context);
         if (keepAlive == -1) {
      // 如果keep-alive值没有由服务器明确设置，那么保持连接持续5秒。
            keepAlive = 5000;
         }
      return keepAlive;
   }
});
```

DefaultHttpClient 也维护一组协议拦截器，意在处理即将离开的请求和即将到达的响应，而且提供管理那些拦截器的方法。新的协议拦截器可以被引入到协议处理器链中，或在需要时从中移除。内部的协议拦截器存储在一个简单的 java.util.ArrayList 中。它们以被加入到 list 中的自然顺序来执行。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.removeRequestInterceptorByClass(RequestUserAgent.
class);
httpclient.addRequestInterceptor(new
HttpRequestInterceptor() {
   public void process(
      HttpRequest request, HttpContext context)
      throws HttpException, IOException {
         request.setHeader(HTTP.USER_AGENT,
      "My-own-client");
   }
});
```

DefaultHttpClient 是线程安全的。建议相同的这个类的实例被重用于多个请求的执行。当一个 DefaultHttpClient 实例不再需要而且要脱离范围时，和它关联的连接管理器必须调用 ClientConnectionManager#shutdown() 方法关闭。

```
HttpClient httpclient = new DefaultHttpClient();
// 做些有用的事
httpclient.getConnectionManager().shutdown();
```

5.2 HttpClient 参数

这些是可以用于定制默认 HttpClient 实现行为的参数：

- **'http.protocol.handle-redirects'**：定义了重定向是否应该自动处理。这个参数期望得到一个 java.lang.Boolean 类型的值。如果这个参数没有被设置，HttpClient 将会自动处理重定向。
- **'http.protocol.reject-relative-redirect'**：定义了是否相对的重定向应该被拒绝。HTTP 规范需要位置值是一个绝对 URI。这个参数期望得到一个 java.lang.Boolean 类型的值。如果这个参数没有被设置，那么就允许相对重定向。
- **'http.protocol.max-redirects'**：定义了要遵循重定向的最大数量。这个重定向数字的限制意在防止由破碎的服务器端脚本引发的死循环。这个参数期望得到一个 java.lang.Integer 类型的值。如果这个参数没有被设置，那么只允许不多余 100 次重定向。
- **'http.protocol.allow-circular-redirects'**：定义环形重定向（重定向到相同路径）是否被允许。HTTP 规范在环形重定向没有足够清晰的允许表述，因此这作为可选的是可以开启的。这个参数期望得到一个 java.lang.Boolean 类型的值。如果这个参数没有被设置，那么环形重定向就不允许。
- **'http.connection-manager.factory-class-name'**：定义了默认的 ClientConnectionManager 实现的类型。这个参数期望得到一个 java.lang.String 类型的值。如果这个参数没有被设置，对于每个默认的将使用 SingleClientConnManager。
- **'http.virtual-host'**：定义了在头部信息 Host 中使用的虚拟主机名称，来代替物理主机名称。这个参数期望得到一个 HttpHost 类型的值。如果这个参数没有被设置，那么将会使用目标主机的名称或 IP 地址。
- **'http.default-headers'**：定义了每次请求默认发送的头部信息。这个参数期望得到一个包含 Header 对象的 java.util.Collection 类型值。
- **'http.default-host'**：定义了默认主机。如果目标主机没有在请求 URI（相对 URI）中明确指定，那么就使用默认值。这个参数期望得到一个 HttpHost 类型的值。

5.3 自动重定向处理

HttpClient 处理所有类型的自动重定向，除了那些由 HTTP 规范明令禁止的，比如需要用户干预的。参考其它（状态码 303）POST 和 PUT 请求重定向转换为由 HTTP 规范需要的 GET 请求。

5.4 HTTP 客户端和执行上下文

DefaultHttpClient 将 HTTP 请求视为不变的对象，也从来不会假定在请求执行期间改变。相反，它创建了一个原请求对象私有的可变副本，副本的属性可以基于执行上下文来更新。因此，如目标主键和请求 URI 的 final 类型的请求参数可以在请求执行之后，由检查本地 HTTP 上下文来决定。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://localhost:8080/");
HttpResponse response = httpclient.execute(httpget,
localContext);
HttpHost target = (HttpHost) localContext.getAttribute(
ExecutionContext.HTTP_TARGET_HOST);
HttpUriRequest req = (HttpUriRequest)
localContext.getAttribute(
ExecutionContext.HTTP_REQUEST);
System.out.println("Target host: " + target);
System.out.println("Final request URI: " + req.getURI());
System.out.println("Final request method: " + req.getMethod());
```

第六章 高级主题

6.1 自定义客户端连接

在特定条件下，也许需要来定制 HTTP 报文通过线路传递，越过了可能使用的 HTTP 参数来处理非标准不兼容行为的方式。比如，对于 Web 爬虫，它可能需要强制 HttpClient 接受格式错误的响应头部信息，来抢救报文的内容。

通常插入一个自定义的报文解析器的过程或定制连接实现需要几个步骤：

- 提供一个自定义 LineParser/LineFormatter 接口实现。如果需要，实现报文解析/格式化逻辑。

```
class MyLineParser extends BasicLineParser {
    @Override
    public Header parseHeader(
        final CharArrayBuffer buffer) throws ParseException {
        try {
            return super.parseHeader(buffer);
        } catch (ParseException ex) {
            // 压制ParseException异常
            return new BasicHeader("invalid",
        buffer.toString());
        }
    }
}
```

- 提过一个自定义的 OperatedClientConnection 实现。替换需要自定义的默认请求/响应解析器，请求/响应格式化器。如果需要，实现不同的报文写入/读取代码。

```
class MyClientConnection extends DefaultClientConnection {
    @Override
    protected HttpMessageParser createResponseParser(
        final SessionInputBuffer buffer,
        final HttpResponseFactory responseFactory,
        final HttpParams params) {
            return new DefaultResponseParser(buffer,
            new MyLineParser(),responseFactory,params);
    }
}
```

- 为了创建新类的连接，提供一个自定义的 ClientConnectionOperator 接口实现。如果需要，实现不同的套接字初始化代码。

```
class MyClientConnectionOperator extends
    DefaultClientConnectionOperator {
    public MyClientConnectionOperator(
        final SchemeRegistry sr) {
        super(sr);
    }
    @Override
    public OperatedClientConnection createConnection() {
        return new MyClientConnection();
    }
}
```

- 为了创建新类的连接操作，提供自定义的 ClientConnectionManager 接口实现。

```
class MyClientConnManager extends SingleClientConnManager {
    public MyClientConnManager(
        final HttpParams params,
        final SchemeRegistry sr) {
        super(params, sr);
    }
    @Override
    protected ClientConnectionOperator
createConnectionOperator(
        final SchemeRegistry sr) {
        return new MyClientConnectionOperator(sr);
    }
}
```

6.2 有状态的 HTTP 连接

HTTP 规范假设 session 状态信息通常是以 HTTP cookie 格式嵌入在 HTTP 报文中的，因此 HTTP 连接通常是无状态的，这个假设在现实生活中通常是不对的。也有一些情况，当 HTTP 连接使用特定的用户标识或特定的安全上下文来创建时，因此不能和其它用户共享，只能由该用户重用。这样的有状态的 HTTP 连接的示例就是 NTLM 认证连接和使用客户端证书认证的 SSL 连接。

6.2.1 用户令牌处理器

HttpClient 依赖 UserTokenHandler 接口来决定给定的执行上下文是否是用户指定的。如果这个上下文是用户指定的或者如果上下文没有包含任何资源或关于当前用户指定详情而是 null，令牌对象由这个处理器返回，期望唯一地标识当前的用户。用户令牌将被用来保证用户指定资源不会和其它用户来共享或重用。

如果它可以从给定的执行上下文中来获得，UserTokenHandler 接口的默认实现是使用主类的一个实例来代表 HTTP 连接的状态对象。UserTokenHandler 将会使用基于如 NTLM 或开启的客户端认证 SSL 会话认证模式的用户的主连接。如果二者都不可用，那么就不会返回令牌。

如果默认的不能满足它们的需要，用户可以提供一个自定义的实现：

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.setUserTokenHandler(new UserTokenHandler() {
   public Object getUserToken(HttpContext context) {
      return context.getAttribute("my-token");
   }
});
```

6.2.2 用户令牌和执行上下文

在 HTTP 请求执行的过程中，HttpClient 添加了下列和用户标识相关的对象到执行上下文中：

- **'http.user-token'**：对象实例代表真实的用户标识，通常期望 Principle 接口的实例。

我们可以在请求被执行后，通过检查本地 HTTP 上下文的内容，发现是否用于执行请求的连接是有状态的。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://localhost:8080/");
HttpResponse response = httpclient.execute(httpget,
localContext);
HttpEntity entity = response.getEntity();
if (entity != null) {
   entity.consumeContent();
}
Object userToken =
localContext.getAttribute(ClientContext.USER_TOKEN);
System.out.println(userToken);
```

6.2.2.1 持久化有状态的连接

请注意带有状态对象的持久化连接仅当请求被执行时，相同状态对象被绑定到执行上下文时可以被重用。所以，保证相同上下文重用于执行随后的相同用户，或用户令牌绑定到之前请求执行上下文的 HTTP 请求是很重要的。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpContext localContext1 = new BasicHttpContext();
HttpGet httpget1 = new HttpGet("http://localhost:8080/");
HttpResponse response1 = httpclient.execute(httpget1,
localContext1);
HttpEntity entity1 = response1.getEntity();
if (entity1 != null) {
   entity1.consumeContent();
}
Principal principal = (Principal) localContext1.getAttribute(
ClientContext.USER_TOKEN);
HttpContext localContext2 = new BasicHttpContext();
localContext2.setAttribute(ClientContext.USER_TOKEN,
principal);
HttpGet httpget2 = new HttpGet("http://localhost:8080/");
HttpResponse response2 = httpclient.execute(httpget2,
localContext2);
HttpEntity entity2 = response2.getEntity();
if (entity2 != null) {
   entity2.consumeContent();
}
```