HttpClient-4.0.1 官方教程

HttpClient 不是一个浏览器，它是一个客户端 HTTP 传输类库。HttpClient 作用是传输和接收 HTTP 消息。HttpClient 不尝试缓存内容，执行内嵌 HTML 页面中的 javascript 脚本，尝试猜测内容类型，重新定义请求/重定向 URI 位置，其他功能与 HTTP 传输无关。

第一章 Fundamentals（基础）

1.1 执行请求

HttpClient 最重要的功能是执行 HTTP 方法。执行一个 HTTP 方法涉及一个或多个 HTTP 请求/ HTTP 响应信息交流，通常是由 HttpClient 内部处理。用户提供一个请求对象，HttpClient 发送请求到目标服务器，希望服务器返回一个相应的响应对象，或者抛出一个异常（如果执行失败）。

很自然，该 HttpClient API 的主要切入点是 HttpClient 的接口定义。

下面是一个请求执行过程中的最简单形式的例子：

```
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 Request (HTTP请求)

所有的 HTTP 请求有一个请求行包含一个方法名，请求 URI 和 HTTP 协议版本。

HttpClient 支持在 HTTP/1.1 规范中定义的所有 HTTP 方法：GET、HEAD、POST、PUT、 DELETE,、TRACE and OPTIONS,用一些特殊的类来表示每一个方法:HttpGet、HttpHead、HttpPost、HttpPut、HttpDelete、HttpTrace 和 HttpOptions。

HTTP 请求 URI 的包括协议、主机名、可选的端口、资源的路径、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 提供了一个实用的方法来简化创建和修改请求的 URIs，URI 可以由程序编程组装：

```
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：

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

查询字符串也可以从各个参数生成：

```
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：

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

1.1.2 HTTP Response （HTTP响应）

HTTP Response 消息是由服务器发送回客户端，客户端接收并解释的请求消息。该消息的第一行由协议版本、数字状态码和相关的简短文本构成。

```
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：

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

1.1.3 Working with message headers （处理头部消息）

HTTP 信息可以包含一系列头部描述消息的属性，例如：内容长度，内容类型等等。HttpClient 提供方法检索、添加、删除、列举头部信息。

```
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：

```
Set-Cookie: c1=a; path=/; domain=localhost
Set-Cookie: c2=b; path="/", c3=c; domain="localhost"
2
```

最有效的方法来获得给定类型的所有头部信息是利用 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：

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

它还提供了便利的方法把 HTTP 消息解析成单独的头部信息元素。

```
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：

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

1.1.4 HTTP entity （HTTP实体）

HTTP 消息能传输内容实体的请求或响应，在一些请求和响应中可以发现实体，因为它们是可选的。HTTP 规范定义了两个实体包装的方法：POST 和 PUT，响应通常附上内容实体。

HttpClient 根据内容来源区分了三种不同类型的实体：

- Streamed：内容是从流接收或产生的，特别是，这类实体是从 HTTP 响应中接收，流媒体通常是不可重复的。
- self-contained：内容是在内存中或者从独立的一个连接或其它实体中获得，self-contained 实体通常是重复的。
- wrapping：内容从另一个实体获得。

1.1.4.1 Repeatable entities （可重复的实体）

一个实体可以是可重复的，这意味着它的内容可以被读取一次以上。这是唯一可能有自我包含的实体（像 ByteArrayEntity 或 StringEntity）。

1.1.4.2 Using HTTP entities （使用 HTTP 实体）

由于一个实体即能表示二进制内容又能表示字符内容，它支持字符编码（支持后者，即：字符内容）。从实体中读取内容，通过 HttpEntity 的 getContent 方法可以取回输入流，并返回一个 java.io.InputStream，或者给 HttpEntity 的 writeTo(OutputStream) 方法提供一个输出流，它将一次性返回所有的被写在流里的内容。

当实体已收到传入消息，HttpEntity＃getContentType（）和 HttpEntity＃getContentLength（）方法可用于读取常见的元数据，例如内容类型和内容长度（如果可用）。由于内容类型标题可以包含文本 MIME 类型的字符编码，像 text/plain 或者 text/html，HttpEntity＃getContentEncoding（）方法用来读取此信息。如果标题不可用，返回的长度为-1，并返回内容类型为 NULL，如果内容类型标题可用，返回标题对象。

当创建一个即将卸任的消息实体，此元数据必须由实体创建者提供。

```
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：

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

1.1.5 Ensuring release of low level resources （确保底层资源的释放）

当响应实体完成时，重要的是要确保所有的实体内容已被完全销毁，使该连接可以安全地返回到连接池，由以后的请求重复利用连接管理器，最简单的方法是调用 HttpEntity＃consumeContent（）方法来消毁任何可用的流内容。HttpClient 一旦检测到流的末尾的内容已经到达，就自动释放基础连接返回到连接管理器。HttpEntity＃consumeContent（）方法安全的调用一次以上。调用 HttpUriRequest＃abort（）方法可以简单地终止请求。

```
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();
}
```

该连接将不被重用，底层的资源将正确地释放。

1.1.6 Consuming entity content（销毁实体内容）

推荐的方法消耗的实体内容是通过使用其 HttpEntity＃getContent（）或 HttpEntity＃writeTo（OutputStream 中）方法。HttpClient 的还配备了 EntityUtils 类，它暴露了一些静态方法能够更容易地从一个实体读取内容或信息。而不是直接读取 java.io.InputStream，使用这个类的方法可以取回 string/byte array 整个内容体，然而，EntityUtils 不鼓励使用，除非实体响应来自一个可信赖的 HTTP 服务器和已知的有限长度。

```
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
```

```
    }
}
```

在某些情况下，它可能需要能够读取实体内容超过一次，在这种情况下实体的内容必须以某种方式被缓冲，无论是在内存或磁盘上。最简单的方法是通过包装原始的 BufferedHttpEntity 类来完成。这将导致原始的实体内容读入内存中的缓冲区。

```
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 Producing entity content（产生实体内容）

HttpClient 提供了常见的数据容器，例如字符串、字节数组、输入流和文件： 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，因为它只能从底层流中读取数据一次。一般来说，建议实现自定义 HttpEntity 类，而不是使用通用 InputStreamEntity。 FileEntity 可以是一个很好的起点。

1.1.7.1 Dynamic content entities （动态的内容实体）

通常的 HTTP 实体需要动态生成一个特定的执行上下文，HttpClient 提供了 ContentProducer 接口和 EntityTemplate 实体类支持动态实体。内容的生产者通过写出来到一个输出流产生他们的需求内容。因此，与 EntityTemplate 创建实体，一般是自给自足，重复性好。

```
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 forms （HTML 表单）

许多应用程序经常需要模拟的提交的 HTM 表单的过程，例如，按顺序登录到 Web 应用程序或提交的输入数据，HttpClient 提供特殊的实体类 UrlEncodedFormEntity 减轻了处理的困难。

```
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);
```

这 UrlEncodedFormEntity 实例将调用 URL 编码对参数进行编码，并产生以下内容：

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

1.1.7.3 Content chunking（内容组块）

一般来说，建议让 HttpClient 基于 HTTP 消息的属性传输选择最合适的传输编码，这是可能的，然而，真正的 HttpClient 通知该组块编码的，是首选的设置 HttpEntity＃setChunked（）为 true。请注意的 HttpClient 将使用该标志，当使用 HTTP 协议版本不支持组块编码时，这个值被忽略，例如：HTTP/1.0 协议。

```
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 Response handlers（响应处理器）

最简单和最方便的方式来处理的响应是使用 ResponseHandler 接口。这种方法完全缓解连接管理，用户不必担心。当使用 ResponseHandler 的 HttpClient 会自动采取异常谨慎，确保释放连接返回到连接管理器，无论是否成功或异常原因。

```
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 execution context （HTTP执行上下文）

最初的 HTTP 被设计成一个无状态，面向响应/请求的协议。然而，现实世界应用程序通常需要能够坚持响应交换状态信息通过几个逻辑上相关的请求。为了使应用程序能够保持状态的 HttpClient 处理 HTTP 请求允许将在一个特定的上下文中执行的执行，简称为 HTTP 上下文。多个逻辑相关的请求可以参与逻辑会话，如果同样的情况下连续请求之间重用。 HTTP 上下文功能类似于 java.util.Map 的<String, Object> 。它只是一个任意命名

的值的集合。应用程序在执行请求之前填充的上下文属性或请求完成之后检查上下文。

在 HttpClient 的 HTTP 请求的执行过程中添加下列属性的执行情况：

- 'http.connection'：HttpConnection 实例代表实际连接到目标服务器。
- 'http.target_host'：HttpHost 实例代表连接的目标。
- 'http.proxy_host'：HttpHost 实例代表连接代理，如果使用。
- 'http.request'：HttpRequest 实例代表实际的 HTTP 请求。
- 'http.response'：HttpResponse 实例代表了实际的 HTTP 响应。
- 'http.request_sent'：java.lang.Boolean 的对象，表示该标志指示是否实际的请求已完全传输到连接的目标。

例如，以确定最终的重定向目标，请求执行之后，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：

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

1.3 Exception handling （异常处理）

HttpClient 可以抛出两种类型的异常：假如发生 I/O 故障抛出 java.io.IOException，例如 socket 超时，socket 重置，HttpException 这种信号的 HTTP 故障，例如违反了 HTTP 协议。通常的 I / O 错误被视为非致命性和可恢复的，而 HTTP 协议被认为是致命的错误并不可恢复的。

1.3.1 HTTP transport safety （HTTP 传输安全性）

最重要的一点:HTTP 协议并不适合所有类型的应用程序。HTTP 是一个简单的要求/响应导向的协议是用来支持静态或开始动态生成的内容检索。它从未打算支持交互操作。例如,将 HTTP 服务器及其部分合同履行如果它成功地接受和加工要求,产生反应和发送给客户端状态码。该服务器将不作任何尝试回滚事务，如果客户端无法接收失事事故的全部原因是读超时，要求取消或系统。如果客户决定重试相同的请求，服务器将不可避免地最终执行相同的交易超过一次。在某些情况下，这可能会导致数据损坏或不一致的应用程序应用程序的状态。

虽然从来没有被设计的 HTTP 支持事务处理，它仍然可以用来作为传输协议的关键任务应用程序提供某些条件得到满足。为了确保 HTTP 传输层安全系统必须确保层幂等的 HTTP 方法应用程序。

1.3.2 Idempotent methods （幂等的方法）

HTTP/1.1 规范定义了幂等方法,【该方法也有 “幂等的”属性（除了错误或过期的问题之外），N> 0 相同请

求】。换句话说，应用程序应该预先确保相同的方法多个执行的意义。这可以实现的，例如，通过提供一个唯一的事务 ID 和其他手段避免执行相同的逻辑运算。

请注意，此问题不是仅限于 HttpClient。基于浏览器应用程序是完全符合 HttpClient 的同样的问题与 HTTP 方法非幂等相关。

1.3.3 Automatic exception recovery （自动的异常恢复）

默认情况下的 HttpClient 尝试自动恢复从 I / O 异常。默认的自动恢复机制是限于一个是安全的少数例外的是众所周知的。

- HttpClient 不会试图恢复任何逻辑错误或 HTTP 协议错误（那些来自 HttpException 类）。
- HttpClient 会自动重试那些被假定为幂等方法。
- HttpClient 会自动重试那些传输异常失败的方法，HTTP 请求仍然被传输到目标服务器。
- HttpClient 会自动重试的完全传输到服务器方法，但服务器没有响应 HTTP 状态代码（服务器仅仅放弃连接不发送任何东西），在这种情况下，假定要求服务器和应用程序没有改变状态。如果这个假设未必如此你的应用程序服务器强烈建议提供一个自定义的异常处理器。

1.3.4 Request retry handler （请求重试处理程序）

自定义异常恢复机制将提供 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) {
            // Do not retry if over max retry count
            return false;
        }
        if (exception instanceof NoHttpResponseException) {
            // Retry if the server dropped connection on us
            return true;
        }
        if (exception instanceof SSLHandshakeException) {
            // Do not retry on SSL handshake exception
            return false;
        }
        HttpRequest request = (HttpRequest) context.getAttribute(
                ExecutionContext.HTTP_REQUEST);
        boolean idempotent = !(request instanceof HttpEntityEnclosingRequest);
        if (idempotent) {
            // Retry if the request is considered idempotent
```

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

1.4 Aborting requests（中止请求）

调用 HttpUriRequest#abort()方法可以终止请求，这个方法是线程安全的，可以从任何线程调用。当 HTTP 请求中止时抛出 InterruptedIOException。

1.5 HTTP protocol interceptors （HTTP协议拦截器）

HTTP 协议拦截器协议是一个例行程序，实现了 HTTP 协议的具体内容。拦截器还可以操纵的消息封闭的实体内容，透明的内容压缩/解压是一个很好的例子。通常这是通过使用'装饰'设计模式，其中一个包装实体类是用来装饰原始实体。几个拦截器可以组合成一个逻辑单元。

拦截器可进行协作共享信息，如处理状态。拦截器可以使用 HTTP 上下文来存储一个或多个连续请求的处理状态。

通常情况下拦截器顺序的被执行，只要他们不依赖于执行上下文某一特定状态。如果拦截器互相依赖，因此必须在一个特定的顺序中执行。

协议拦截器必须线程安全的实现，同 servlet 一样，协议拦截不应使用实例变量，除非获得这些变量的同步。

这是一个例子，如何在局部的上下文中保存连续请求的处理状态：

```
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拦截器必须实现为线程安全的parameters （HTTP参数）

HttpParams 接口代表一个不可改变值的集合，定义一个组件运行时行为。在许多方面 HttpParams 与 HttpContext 是相似的。两个接口都代表一个对象集合，该集合是一个键到值的映射，但是，服务于不同的用途。

- HttpParams 适用于包含简单的对象：integers, doubles, strings, collections 和 objects 在运行时保持不变。
- HttpParams 预计用在“'写一次，读多次”方式，HttpContext 适用于包含复杂对象，这些对象很可能在 HTTP 消息处理过程中发生变异。
- HttpParams 作用是定义其他组件的行为，一般每个复杂的组件都有它自己的 HttpParams 对象。HttpContext 作用是代表了一个 HTTP 进程的执行状态。通常是相同的执行上下文之间共享许多合作对象。

1.6.1 Parameter hierarchies （参数的层次体系）

HttpRequest 对象是 HttpClient 的实例用来执行连在一起的请求 ParamsHttp。
允许参数设置成 HTTP 请求级别优先于设置成 HTTP 客户端级别。
建议的做法是通过设置在 HTTP 客户端级别的共同所有的 HTTP 请求参数和选择性覆盖在 HTTP 请求级别的具体参数。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.getParams().setParameter(CoreProtocolPNames.PROTOCOL_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_EXPECT_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));
```

```
        }
});
```

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

1.6.2. HTTP parameters beans

HttpParams 接口在处理组件的配置方面考虑到很多灵活性。最重要的事，可以引进新的参数，而不会影响旧版本的二进制兼容性。然而，HttpParams 与 JAVA BEAN 规则相比也有一定的缺点：HttpParams 不能使用 DI 框架装配，为了减轻限制，使用标准的 JAVA BEAN 约定初始化 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：

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

1.7 HTTP request execution parameters （HTTP请求执行参数）

这些参数会影响请求执行的进程：

- 'http.protocol.version'：定义使用 HTTP 协议版本，如果没有设定明确的请求对象。此参数要求是 ProtocolVersion 类型的值，如果不设置此参数将使用 HTTP/1.1。
- 'http.protocol.element-charset'：定义了要使用的字符集编码的 HTTP 协议的元素。此参数要求是 java.lang.String 类型的值，如果不设置此参数将使用 US-ASCII。
- 'http.protocol.content-charset'：定义字符集用于每个内容体的默认编码。此参数要求是 java.lang.String 类型的值，如果不设置此参数将使用 ISO-8859-1。
- 'http.useragent'：定义 User-Agent 头的内容。此参数要求是 java.lang.String 类型的值，如果此参数不设置，HttpClient 会自动为它生成值。
- 'http.protocol.strict-transfer-encoding'：定义是否以无效 Transfer-Encoding 头的响应，应予以拒绝。此参数要求是 java.lang. Boolean 类型的值，如果该参数没有设置无效 Transfer-Encoding 值将被忽略。

- 'http.protocol.expect-continue'：激活 Expect: 100-Continue 实体内附的方法握手，Expect: 100-Continue 握手的作用是让客户端随着请求主体发送一个请求消息，如果源服务器愿意接受请求（基于请求头），然后客户端发送请求主体。使用 Expect: 100-Continue 握手，实体附入的请求结果性能得到明显的改善（例如 POST 和 PUT），谨慎使用 Expect: 100-continue，因为它可能引起问题，HTTP 服务器和代理不支持 HTTP 1.1 协议。此参数要求是 java.lang. Boolean 类型的值，如果此参数不设置，HttpClient 会试图使用握手。
- 'http.protocol.wait-for-continue'：定义以毫秒为单位，客户端等待 100-continue 响应花费的最大一段时间。此参数要求是 java.lang. Integer 类型的值，如果此参数不设置，HttpClient 的请求体将等待 3 秒为一个确认，然后恢复传输。

第二章 Connection management（连接管理）

HttpClient 的具有完全控制初始化连接和终止连接进程以及 I / O 操作的活动连接，然而可以使用一个参数来控制连接方面的操作。当值为 0 被解释成一个无限暂停，此参数要求是 java.lang. Integer 类型的值，如果此参数不设置，读操作不会超时（无限暂停）。

2.1 Connection parameters （连接参数）

这些参数可以影响连接操作：

- 'http.socket.timeout'：以毫秒为单位定义套接字超时（SO_TIMEOUT）。当值为 0 被解释成一个无限的暂停，此参数要求是 java.lang. Integer 类型的值，如果此参数不设置，读操作不会超时（无限暂停）。
- 'http.tcp.nodelay'：确定是否使用 Nagle 算法。Nagle 算法尝试通过将被发送的段数量减到最少节省带宽。当应用程序希望降低网络延迟提高性能，他们可以禁用 Nagle 算法（启用 TCP_NODELAY，以增加消耗带宽为代价发送数据，此参数要求是 java.lang. Boolean 类型的值，如果此参数不设置，启用 TCP_NODELAY）。
- 'http.socket.buffer-size'：接收/传输 HTTP 消息时，确定 socket 内部缓冲区缓存数据的大小。此参数要求是 java.lang. Integer 类型的值，如果此参数不设置，HttpClient 将分配 8192 字节 scoket 缓冲区。
- 'http.socket.linger'：
- 'http.connection.timeout'：建立连接的超时时间（以毫秒为单位）。当值为 0 被解释成一个无限超时，此参数要求是 java.lang. Integer 类型的值，如果此参数不设置，连接操作不会超时（无限超时）。
- 'http.connection.stalecheck'：
- 'http.connection.max-line-length'：确定行数最大长度的限制。如果设置为正值，任何 HTTP 行超过此限制将导致 IOException，负或零值将有效地禁用检查。此参数要求是 java.lang. Integer 类型的值，如果此参数不设置，没有限制。
- 'http.connection.max-header-count'：确定最大允许 HTTP 头数。如果设置一个正值，从数据流收到的 HTTP 头数超过此限制将导致一个 IOException，负或零值将有效地禁用检查，此参数要求是 java.lang. Integer 类型的值，如果此参数不设置，没有限制。
- 'http.connection.max-status-line-garbage'：定义忽略的 HTTP 响应的状态行最大行数。HTTP/1.1 持续连接，该问题产生中断脚本返回错误的内容长度（），不幸的是，在某些情况下，检测不到这个错误的响应，所以的 HttpClient 必须能够跳过这些多余的行，此参数要求是 java.lang. Integer 类型的值，值为 0 拒绝所有的垃圾行/空行之前的状态行。

2.2 Connection persistence （持续连接）

从主机到另一个主机建立连接的过程是相当复杂的，涉及到两个端点之间多个包交换是相当耗时的。人们可以实现更高的数据吞吐量，如果打开的连接可以被重新用于执行多个请求。

HTTP/1.1 中指出 HTTP 连接默认可以重用多个请求。HTTP/1.0 也符合该机制，HttpClient 完全支持持续连接。

2.3 HTTP connection routing（HTTP连接路由）

HttpClient 直接或间接路由建立连接到目标主机，路由涉及到多个中间连接（跳板）。HttpClient 路由连接的区别是进入线路、隧道、分层。使用多个中间代理的连接到目标主机被称为代理链。

2.3.1 Route computation

RouteInfo 接口信息明确表示了路由到目标主机涉及一个或多个中间步骤或跳板。HttpRoute 是 RouteInfo 一个具体的实现，HttpRouteDirector 是一个辅助类，可以用来计算下一步路线，这个类由 HttpClient 内部使用。

HttpRoutePlanner 接口代表一个完整的路径来计算给定目标的基础上的运行环境，HttpClient 有两个 HttpRoutePlanner 默认实现。ProxySelectorRoutePlanner 基于 java.net.ProxySelector，默认情况下，它取出 JVM 代理设置，无论是从系统属性或从浏览器中运行的应用程序。DefaultHttpRoutePlanner 实现不使用 JAVA 系统属性，也不使用 JAVA 代理或浏览器代理设置，它计算路线完全基于 HTTP 的描述参数。

2.3.2 Secure HTTP connections

HTTP 连接可以被视为安全的，如果信息在两个连接端点之间传输不能被读取或未经授权的第三方篡改。SSL / TLS 协议是最广泛使用的技术，以确保 HTTP 传输安全，然而,其它加密技术也可以使用。通常,HTTP 传输层使用 SSL / TLS 加密连接。

2.4 HTTP route parameters

这些参数可以影响线路计算：

- 'http.route.default-proxy'：定义一个代理主机使用默认的路线规划者不使用 JRE 设置，此参数要求是 HttpHost 类型的值，如果此参数不设置，尝试直接连接到目标地址。
- 'http.route.local-address'：定义一个本地地址使用默认路由规划者。在具有多个网络接口的机器上，这个参数可以用来选择从哪个网络接口连接。此参数要求是 java.net.InetAddress 类型的值，如果这个参数未设置自动默认本地地址将被使用。
- 'http.route.forced-route'：定义一个强制路由使用默认路由规划者。代替计算路由，返回强制路由，即使它指向一个完全不同的目标主机。此参数要求是 HttpRoute 类型的值。

2.5 Socket factories

HTTP 连接使用一个 java.net.Socket 对象内部通过线路处理数据传输。不过，他们依靠 SocketFactory 接口来创建，初始化连接套接字。这使得 HttpClient 用户在应用程序运行时提供特定的套接字初始化代码。PlainSocketFactory 是一个默认的工厂用来创建和初始化简单的（未加密）套接字。

建立一个套接字进程和连接到一个主机是分离的，当连接操作阻塞时，套接字可以被关闭。

```
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 Secure socket layering

LayeredSocketFactory 扩展了 SocketFactory 接口，套接字工厂 layer 在现有的普通套接字上创建套接字，套接字 layering 通过代理创建安全的套接字。HttpClient SSLSocketFactory 实现了 SSL/TSL 层，请注意的 HttpClient 不使用任何自定义加密功能，它完全依赖于标准的 Java 扩展加密（JCE）和安全套接字（JSEE）。

2.5.2 SSL/TLS customization

HttpClient 的使用 SSLSocketFactory 创建 SSL 连接，SSLSocketFactory 允许高度定制，它可以采取 javax.net.ssl.SSLContext 实例作为一个参数，并使用它来创建自定义配置 SSL 连接。

```
TrustManager easyTrustManager = new X509TrustManager() {
    @Override
    public void checkClientTrusted(
            X509Certificate[] chain,
            String authType) throws CertificateException {
        // Oh, I am easy!
    }
    @Override
    public void checkServerTrusted(
            X509Certificate[] chain,
            String authType) throws CertificateException {
        // Oh, I am easy!
    }
    @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 定制意味着对 SSL/TLS 协议概念有一定的了解，详细解释超出对本的范围，请参阅【 http://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/ JSSERefGuide.html 】 的 详 细 描 述 javax.net.ssl.SSLContext 和扩展 Java 安全套接字相关的工具。

2.5.3. Hostname verification

除了信任验证和客户端身份验证级别上执行的 SSL / TLS 协议，一旦连接已建立，HttpClient 的可以选择验证目标主机名是否与存储在服务器侧的 X.509 证书名称匹配。这种验证可以对服务器的信任材料真实性提供额外的保证。X509HostnameVerifier 接口表示主机名称验证策略，重要的是：主机名验证不应与 SSL 的信任验证混淆。

- StrictHostnameVerifier：
- BrowserCompatHostnameVerifier：
- AllowAllHostnameVerifier：

2.6 Protocol schemes

Scheme 类表示一个协议方案，例如"http"或者"https"和包含许多的协议属性，例如缺省的端口和 socket 工厂常用于为指定的协议创建 java.net.Socket 实例，SchemeRegistry 类被用来维护一个 Schemes 的集合，HttpClient 可以选择通过一个请求 URI 设法建立一个连接。

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

2.7 HttpClient proxy configuration

即使 HttpClient 明白复杂的路由计划和一连串的代理，它也仅仅支持从 box 开始的简单直接或一级跳转代理连接。最简单的方式来通知 HttpClient 通过代理设置一个缺省的代理参数连接目标主机。

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

也可以指示 HttpClient 使用标准的 JRE 代理选择器去获得代理信息。

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

为了完全的控制 Http 路由计算，提供一个自定义 RouterPlanner 实现。

```
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 connection managers

2.8.1 Connection operators

操作连接是指客户端连接，它的基础套接字或状态能被外部实体操作，通常是指一个连接操作者。OperatedClientConnection 接口继承 HttpClientConnection 接口，和定义额外的方法管理连接套接字。ClientConnectionOperator 接口表示创建 ClientConnectionOperator 实例和更新这些对象基础的套接字一个策略。应用充分利用 SocketFactorys 来创建 java.net.socket 实例。ClientConnectionOperator 接口使 HttpClient 的用户为连接操作者提供传统的策略，和提供交替应用 OperatedClientConnection 接口的能力。

2.8.2 Managed connections and connection managers

HTTP 连接是复杂、有状态的，非线程安全的对象需要适当的管理正确的功能。HTTP 连接每次仅被一个执行的线程使用，HttpClient 利用一个特殊的实体管理访问 HTTP 连接，称为 HTTP 连接管理器,由 ClientConnectionManager 接口表示。HTTP 连接管理器的充当一个新的 HTTP 连接工厂，管理持续的连接和同步的访问持续的连接，确保每次只有一个线程能访问连接。

内部的 HTTP 连接管理器与 OperatedClientConnection 的实例协同工作。但是他们取出 ManagedClientConnection 实例来服务客户。ManagedClientConnection 充当一个 OperatedClientConnection 实例的包装器，管理它的状态和控制连接的所有的 I/O 操作。为了建立路由，为打开和更新 socket 提供抽象的 socket 操作和便利的方法。ManagedClientConnection 实例注意到他们对连接管理的连接来创建他们，事实上当他们不再使用时必须返回管理器。ManagedClientConnection 类也应用 ConnectionReleaseTrigger 接口，这个接口被用来触发连接释放，返回到管理器。一旦连接释放被触发，封装的连接从 ManagedClientConnection 封装和 OperatedClientConnection 实例中分离返回到管理器。即使 service consumer 任然控制着 ManagedClientConnection 实例，不能执行任何 i/o 操作或者改变 ManagedClientConnection 有意图或者无意图的状态。

这是一个从连接管理器获得连接的例子。

```
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);
// Request new connection. This can be a long process
ClientConnectionRequest connRequest = connMrg.requestConnection(
        new HttpRoute(new HttpHost("localhost", 80)), null);
// Wait for connection up to 10 sec
ManagedClientConnection conn = connRequest.getConnection(10, TimeUnit.SECONDS);
try {
    // Do useful things with the connection.
    // Release it when done.
    conn.releaseConnection();
} catch (IOException ex) {
    // Abort connection upon an I/O error.
    conn.abortConnection();
    throw ex;
}
```

如果必要的话，连接请求可以调用 ClientConnectionRequest#abortRequest() 终止．解除ClientConnectionRequest#getConnection()方法中阻塞的线程。

一旦响应内容被接收，BasicManagedEntity 封装类能确保基础连接自动释放。HttpClient 使用内部机制取得透明的连接，从 HttpClient#execute()方法中得到所有的响应，。

```
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);
        // Replace entity
        response.setEntity(managedEntity);
    }
    // Do something useful with the response
    // The connection will be released automatically
    // as soon as the response content has been consumed
} catch (IOException ex) {
    // Abort connection upon an I/O error.
    conn.abortConnection();
    throw ex;
```

```
}
```

2.8.3. Simple connection manager

SingleClientConnManager 是简单的连接管理器，它每次仅维护一个连接。即使这个类是线程安全的，它也只能被一个执行线程使用。SingleClientConnManager 努力重用具有相同路由的连续请求的连接。然而，如果持久连接的路由不匹配连接请求，它将会关闭已有的连接和打开指定的路由连接。如果连接已经被分配，抛出 java.lang.IllegalStateException。

HttpClient 缺省使用 SingleClientConnManager。

2.8.4. Pooling connection manager

ThreadSafeClientConnManager 是一个很复杂的实现，它管理一个客户连接池，服务多个执行线程的连接请求，连接被每个路由放入池中。连接池中可用的已经存在持久连接的路由请求由池中租借的连接进行服务，而不是创建一个新的连接分支。

ThreadSafeClientConnManager 在每个路由中维持一个最大的连接限制。缺省的应用对每个路由创建仅仅 2 个 concurrent 连接，总数不超过 20 个 连接。对于很多现实的应用，这些限制可能出现太多的限制，特别是如果他们使用 HTTP 作为一个 传输协议进行服务。连接限制.然而，可以通过 HTTP 参数进行调整。

这个例子演示如何调整连接池参数：

```
HttpParams params = new BasicHttpParams();
// Increase max total connection to 200
ConnManagerParams.setMaxTotalConnections(params, 200);
// Increase default max connection per route to 20
ConnPerRouteBean connPerRoute = new ConnPerRouteBean(20);
// Increase max connections for localhost:80 to 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. Connection manager shutdown

当 HttpClient 不再需要和超过作用域范围时，连接管理器关闭所有的活动连接，确保系统资源被释放掉。

```
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. Connection management parameters

自定义标准的 HTTP 连接管理实现参数：

- 'http.conn-manager.timeout'：定义以毫秒为单位，从 ClientConnectionManager 获得一个 ManagedClientConnection 实例超时时间。该参数要求是一个 java.lang.Long 类型的值，如果该参数没有被设置，连接请求将没有时间限制。
- 'http.conn-manager.max-per-route'：定义路由最大的连接数。这个限制被客户连接管理器解释。应用到单个的管理器实例。该参数要求是一个 ConnPerRoute 类型的值。
- 'http.conn-manager.max-total'：定义总连接数最大值。这个限制被客户端连接管理器解释和应用到单个的管理器实例。该参数要求是一个 java.lang.Integer 类型的值。

2.10 Multithreaded request execution

当配置了连接池管理器，例如 ThreadSafeClientConnManager，HttpClient 使用多线程同时执行多个请求。ThreadSafeClientConnManager 类基于它的配置分配连接。如果所有的链接线路被租用，连接请求将被阻塞直到一个连接被释放回到池中。在连接请求操作中设置'http.conn-manager.timeout'为正值，确保连接管理器不会无限期的阻塞。如果连接请求在规定的时间段内没有响应，将抛出 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);
// URIs to perform GETs on
String[] urisToGet = {
    "http://www.domain1.com/",
    "http://www.domain2.com/",
    "http://www.domain3.com/",
    "http://www.domain4.com/"
};
// create a thread for each URI
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);
}
// start the threads
```

```
for (int j = 0; j < threads.length; j++) {
    threads[j].start();
}
// join the threads
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) {
                // do something useful with the entity
                // ...
                // ensure the connection gets released to the manager
                entity.consumeContent();
            }
        } catch (Exception ex) {
            this.httpget.abort();
        }
    }

}
```

2.11 Connection eviction policy

典型的阻塞 I / O 模型的主要缺点之一是当一个 I/O 操作被阻塞时，网络套接字可以对 I/O 事件作出反应。当连接被释放回管理器时，它可以保持活动，然而，它不能监控 socket 的状态并对任何 I/O 事件作出反应。如果连接在服务端被关闭，那么客户端不会侦测到这个连接状态的变化而及时的作出反应来关闭 socket。

HttpClient 尝试以减轻该问题通过测试连接是否'陈旧'，这是不再有效，因为它是在服务器端关闭，才能使用该连接执行一个 HTTP 请求。陈旧的连接检查不是 100％可靠，并增加了 10 到 30 毫秒的开销每个请求执行。唯

一可行的解决方案，不涉及一个每一个线程空闲连接插座模型是一个专门用来监视的线程以驱逐那些被认为过期由于长期不活动的连接。在监视器线程可以定期调用 ClientConnectionManager＃closeExpiredConnections()方法关闭所有 过期的连接和驱逐从池中关闭连接。它也可以随意调用 ClientConnectionManager＃closeIdleConnections()方法关闭所有已闲置一段时间一段时间的连接。

```
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);
                    // Close expired connections
                    connMgr.closeExpiredConnections();
                    // Optionally, close connections
                    // that have been idle longer than 30 sec
                    connMgr.closeIdleConnections(30, TimeUnit.SECONDS);
                }
            }
        } catch (InterruptedException ex) {
            // terminate
        }
    }
    public void shutdown() {
        shutdown = true;
        synchronized (this) {
            notifyAll();
        }
    }
}
```

2.12. Connection keep alive strategy

HTTP 规范没有指定一个持久连接可以而且应该保持活动多久。一些 HTTP 服务器使用非标准 keep-alive 头传达给客户的时间在几秒钟的时间，他们打算保持连接在服务器端活动。 HttpClient 是这个信息变得可用。如果 keep-alive 头中没有反应，目前的 HttpClient 假定连接可以无限期地持续下去。然而，许多 HTTP 服务器内有配置取消掉在一定的时间内没有活动的连接，以节省系统的持久连接资源，往往不通知客户。如果默认的策略证明

是过于乐观，可能要提供一个自定义 keep-alive 的策略。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.setKeepAliveStrategy(new ConnectionKeepAliveStrategy() {
    public long getKeepAliveDuration(HttpResponse response, HttpContext context) {
        // Honor 'keep-alive' header
        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.getHostName())) {
            // Keep alive for 5 seconds only
            return 5 * 1000;
        } else {
            // otherwise keep alive for 30 seconds
            return 30 * 1000;
        }
    }
});
```

第三章 HTTP state management（HTTP状态管理）

最初的 HTTP 作为一个无状态的，面向协议请求/响应被设计，这个协议没有提供有状态的会话进行横跨多个逻辑相关的请求/响应交换，作为 HTTP 协议变的很流行和被采用，越来越多的没有打算应用它的系统开始使用它，例如在电子商务应用中作为传输工具。因此对状态管理的支持变成必要。

Netscape 通信，在那个时期引领网络客户和服务软件，在他们的产品中依据一个专有规格对 HTTP 状态管理实施支持。后来,Netscape 通过发布一个规格草稿试着使机制标准化。这个努力有助于通过 RFC 标准轨道使正式规格被定义，然而，在许多应用中状态管理大部分依靠 Netscape 草稿和官方规格不兼容。所有主要 web 浏览器的开发者被迫保留和那些应用的兼容性有助于破碎的标准。

3.1. HTTP cookies

Cookie 是一个标志或者是状态信息的封装， HTTP 代理和目标服务进行交换来保持一个会话，Netscape 工程师过去常常称它为一个魔术 cookie。

HttpClient 使用 cookie 接口来代表一个抽象的 cookie 标志，用简单的形式，一个 HTTP cookie 仅仅是一个键值对。通常一个 HTTP cookie 也包括许多属性，例如版本，有效的域名，cookie 向源服务申请的明确的 URL 子集路径和最大的有效期。

SetCookie 接口表示为了维护一个会话状态源服务向 HTTP 代理发送的一个 set-cookie 的响应头。setCookie2 接口继承 SetCookie，SetCookie2 带有的特定的方法。

ClientCookie 接口继承 Cookie 接口，增加了客户端特有的功能，能准确的获得原始 cookie 属性，它们被原始服务指定，这对于生成 cookie 头部很重要，因为一些 cookie 规范需要 cookie 头部包含某个属性，只有他们 set-Cookie 和 set-Cookie2 头部中被指定。

3.1.1. Cookie versions

Netscape 草稿规范 Cookies 的兼容性没有遵从官方的规范，被认为是 0 版本，标准合格 cookies 被认为是 1 版本，HttpClient 可能依靠版本处理不同的 cookies。

这有一个重新创建 Netscape cookie 的例子：

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

这里有一个重新创建标准 cookie 的例子，请注意标准合格的 cookie 必须保留原始服务发送的所有的属性：

```
BasicClientCookie stdCookie = new BasicClientCookie("name", "value");
stdCookie.setVersion(1);
stdCookie.setDomain(".mycompany.com");
stdCookie.setPath("/");
stdCookie.setSecure(true);
// Set attributes EXACTLY as sent by the server
stdCookie.setAttribute(ClientCookie.VERSION_ATTR, "1");
stdCookie.setAttribute(ClientCookie.DOMAIN_ATTR, ".mycompany.com");
```

这里有一个重建 Set-cookie2 合格 cookie 的例子，请注意标准合格的 cookie 必须保留原始服务发送的所有属性：

```
BasicClientCookie2 stdCookie = new BasicClientCookie2("name", "value")
stdCookie.setVersion(1);
stdCookie.setDomain(".mycompany.com");
stdCookie.setPorts(new int[] {80,8080});
stdCookie.setPath("/");
stdCookie.setSecure(true);
// Set attributes EXACTLY as sent by the server
stdCookie.setAttribute(ClientCookie.VERSION_ATTR, "1");
stdCookie.setAttribute(ClientCookie.DOMAIN_ATTR, ".mycompany.com");
stdCookie.setAttribute(ClientCookie.PORT_ATTR, "80,8080");
```

3.2. Cookie specifications

CookieSpec 接口代表一个 cookie 管理规范，cookie 管理规范强制要求：

- 解析 set-cooke 和可选的 set-cookie2 头部的规则
- 解析 cookie 验证规则
- 对于主机，端口和原始路径的 cookie 头部规格。

HttpClient 有几个 CookieSpec 实现：

- Netscape 草稿： 该规范遵从原 Netscape 通信发布的原始草稿规范，它应该避免，除非对遗留代码兼容性绝对需要。
- RFC 2109：官方 HTTP 状态管理规范的旧版本被 RFC 2965 取代。
- RFC 2965：官方 HTTP 状态管理规范。
- 浏览器兼容：这个实现努力去精密的模仿常见的 WEB 浏览器应用程序，比如 IE 和火狐。
- 最好的匹配：'Meta' cookie 明确说出获得一个 cookie 规则依据 Http 响应发送的 cookie 格式，它基本上把所有的应用积聚到一个类中。

强烈推荐使用最好匹配策略，让 HttpClient 在运行时依据执行上下文获取适当的遵从级别。

3.3 HTTP cookie和状态管理参数

这些参数被用于自定义 HTTP 状态管理和 cookie 规范行为：

- 'http.protocol.cookie-datepatterns'：定义有效的日期格式被用来解析非标准的 expires 属性。只有在兼容非遵从的服务中需要，这个服务仍然用 expires 定义在 Netscape 草稿而不是标准的 max-age 属性。该参数要求一个 java.util.Collection 类型的值。这个集合元素必须是 java.lang.String 类型，和语法 java.text.SimpleDateFormat 语法兼容。如果该参数没有设置，缺省值的选择由 CookieSpec 应用指定，请注意这个参数应用。
- 'http.protocol.single-cookie-header'：定义 cookies 是否被强迫进入一个单个的 cookie 请求头部，否则，每个 cookie 被格式为分开的 cookie 头部。该参数要求一个 java.lang.Boolean 类型的值。如果该参数没有设置，缺省的值的选择由 cookieSpec 应用指定。请注意该参数应用在一个严格的 cookie 规范(RFC2109 和 RFC2965),浏览器兼容和草稿规范将总是把所有 cookies 放在一个请求的头部。
- 'http.protocol.cookie-policy'：定义一个 cookie 名字的规则被用在 HTTP 状态管理。该参数要求一个 java.lang.String 的值。如果该参数没有设置，有效的日期格式由 CookieSpec 实现指定。

3.4 Cookie specification registry（cookie规范注册表）

HTTPClient 用 CookiespecRegistry 类维护一个可用规范的注册表，下列规范被每个缺省注册：

- compatibility：浏览器兼容。
- netscape：Netscape 草稿。
- rfc2109：RFC2109(严格规范过时)。
- rfc2965：RFC2965(标准遵从严格规则)。
- best-match：最好的匹配 meta-policy。

3.5 Choosing cookie policy（选择cookie策略）

Cookievv 策略被设置在 HTTP 客户端,如果需要的话可以重写 HTTP 请求级别：

```
HttpClient httpclient = new DefaultHttpClient();
// force strict cookie policy per default
httpclient.getParams().setParameter(ClientPNames.COOKIE_POLICY, CookiePolicy.RFC_2965);
HttpGet httpget = new HttpGet("http://www.broken-server.com/");
// Override the default policy for this request
httpget.getParams().setParameter(
        ClientPNames.COOKIE_POLICY, CookiePolicy.BROWSER_COMPATIBILITY);
```

3.6 Custom cookie policy（自定义cookie策略）

为了实现一个自定义的 cookie 策略,必须创建一个 CookieSpec 接口自定义实现。创建一个 CookieSpecFactory 实现来创建和初始化自定义规则和注册一个 HTTPClient 工厂实例。一旦自定义的规则被注册，它可以用和标准 cookie 规范相同的方法被激活。

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

3.7 Cookie persistence（cookie持久化）

HTTpClient 能从事持久化 cookie 的物理代表工作，被储存在 CookieStore 接口，缺省的 CookieStore 实现调用 BasicClientCookie,是一个依靠 java.util.ArrayList 简单的应用 ，当容器对象取得垃圾收集时，保存在一个 BasiscClientCookie 对象中的 cookie 被丢失，当必要时，用户能提供更多复杂的应用。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
// Create a local instance of cookie store
CookieStore cookieStore = new MyCookieStore();
// Populate cookies if needed
BasicClientCookie cookie = new BasicClientCookie("name", "value");
cookie.setVersion(0);
```

```
cookie.setDomain(".mycompany.com");
cookie.setPath("/");
cookieStore.addCookie(cookie);
// Set the store
httpclient.setCookieStore(cookieStore);
```

3.8 HTTP state management and execution context

在 HTTP 请求执行过程中，HTTPClient 添加下列状态管理相关对象来执行上下文：

- 'http.cookiespec-registry'：CookieSpecRegistry 实例代表一个实际的 cookie 规范注册表，这个属性值被设置到本地上下文中优先于缺省。
- 'http.cookie-spec'：CookieSpec 实例代表实际的 cookie 规范
- 'http.cookie-origin'：CookieOrigin 实例代表实际的源服务细节。
- 'http.cookie-store'：CookieStore 实例代表实际的 cookie 存储。属性值被设置在本地上下文中优先于缺省。

本地 HttpContext 对象被用来自定义 HTTP 状态管理上下文，在请求执行之前或者当请求被执行后，检查他的状态。

```
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 Per user / thread state management

使用单一的本地执行上下文为了实现每个用户或每个线程状态管理。cookie 规范注册表和 cookie 存储被定义在本地上下文中，将优先于缺省设置在 HTTP 客户级别。

```
HttpClient httpclient = new DefaultHttpClient();
// Create a local instance of cookie store
CookieStore cookieStore = new BasicCookieStore();
// Create local HTTP context
HttpContext localContext = new BasicHttpContext();
// Bind custom cookie store to the local context
localContext.setAttribute(ClientContext.COOKIE_STORE, cookieStore);
HttpGet httpget = new HttpGet("http://www.google.com/");
// Pass local context as a parameter
HttpResponse response = httpclient.execute(httpget, localContext);
```

第四章 HTTP authentication（HTTP认证）

HttpClient 完全支持 HTTP 标准规范中定义的认证模式。HttpClient 的认证框架也可以进行扩展以支持 NTLM 和 SPNEGO 等非标准认证模式。

4.1. User credentials（用户凭证）

任何用户身份验证过程需要一套可以用来确定用户的身份凭据。在最简单的形式用户 crednetials 可以只是一个用户名/密码对。 UsernamePasswordCredentials 代表了一个安全主体组成的凭据和一个明文密码。这是执行标准认证由 HTTP 规范中定义的标准计划足够了。

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

输出：

```
user
pwd
```

NTCredentials 是一个 Microsoft Windows 的具体实现，包括用户名/密码对，增加一套 windows 特有的属性，例如用户的域名，微软 Windows 网络相同的用户可以属于多个域使用不同的认证设置。

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

输出：

```
DOMAIN/user
pwd
```

4.2. Authentication schemes（认证模式）

该 AuthScheme 接口表示面向应答式的验证模式。验证模式要求支持以下功能：

- 在分析和处理发送目标服务器响应请求的受保护资源的挑战。
- xx
- yy

注意验证模式可能涉及的状态应答式交换。HttpClient 有几个 AuthScheme 实现：

- Basic：基本身份验证模式在 RFC 2617 中定义。这种身份验证模式是不安全的，因为凭据以明文形式传输。尽管它不安全，如果与 TLS / SSL 加密技术结合使用，基本验证模式是十分满足需求的。
- Digest：摘要式身份验证模式在 RFC 2617 中定义。摘要式身份验证方案是大大超过基本安全，对于那些不想应用程序通过 TLS / SSL 加密安全传输，Basic 是一个不错的选择。
- NTLM：NTLM 是微软 Windows 平台开发和优化专有的认证模式。 NTLM 被认为比摘要更安全。这个模式是需要一个外部 NTLM 身份引擎能够发挥作用。有关详情请参阅 NTLM_SUPPORT.txt 文件包含在 HttpClient 发行版中。

4.3. HTTP authentication parameters（HTTP认证参数）

这些都是用来定制的 HTTP 身份验证过程和个人身份验证方案的行为参数：

- 'http.protocol.handle-authentication'：定义是否应自动处理身份验证。此参数要求 java.lang.Boolean 类型的值。如果该参数没有设置，HttpClient 会自动处理身份验证。
- 'http.auth.credential-charset'：定义的字符集编码时要使用的用户凭据。此参数要求 java.lang.String 类型的值。如果此参数不设置，US-ASCII 将被使用。

4.4 Authentication scheme registry

HttpClient 使用 AuthSchemeRegistry 类来维护可用的身份验证模式注册。下面的模式被默认注册：

- Basic：基本身份验证模式
- Digest：摘要身份验证模式

请注意 NTLM 模式不是默认注册的。NTLM 不能启用，由于默认许可和法律原因，有关如何启用 NTLM 支持，请参见本节详细介绍。

4.5 Credentials provider

凭证提供者是值维护一套用户凭证，并能够产生特定的认证范围的用户凭证。认证范围包括主机名称，端口号，一个域名和身份验证模式的名称。在与供应商可以提供凭证外卡（任何主机，任何端口，任何领域，任何模式），而不是一个具体的属性值登记证书。提供的凭证，然后将能够找到一个特定的范围最接近的匹配，如果匹配的直接无法找到。 HttpClient 的可以与任何一个凭证提供程序接口实现 CredentialsProvider 物理表示。默认 CredentialsProvider 实现所谓 BasicCredentialsProvider 是一个简单实现的 java.util.HashMap。

```
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：

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

4.6 HTTP authentication and execution context

HttpClient 依赖有关身份验证过程的状态详细信息跟踪的 AuthState 类。在 HTTP 请求执行期间创建两个 AuthState 实例：一个用于目标主机的认证，另一个用于代理认证。如果目标服务器或代理服务器需要用户身份验证的实例将各自 AuthScope 与 AuthScope，AuthScheme 并在验证过程中使用 Crednetials 填充该 AuthState 可以检查，以找出什么样的认证要求，是否匹配 AuthScheme 执行结果和提供的凭据，是否能够找到给定的认证范围的用户凭据。

在 HTTP 请求执行的 HttpClient 过程中添加下面的身份验证来执行上下文相关的对象：

- 'http.authscheme -registry'：AuthSchemeRegistry 实例代表实际的身份验证方案的注册表。由于在本地范围内设置此属性的默认值接管一个优先级。
- 'http.auth.credentials-provider'：CookieSpec 实例代表提供实际凭据。由于在本地范围内设置此属性的默认值接管一个优先级。
- 'http.auth.target-scope'：HTTP 认证 AuthState 实例代表实际的目标身份验证状态。由于在本地的情况设置此属性的默认值接管一个优先级。
- 'http.auth.proxy-scope'：AuthState 实例代表实际的代理身份验证状态。由于在本地范围内设置此属性的默认值接管一个优先级。

本地 HttpContext 对象被用来自定义 HTTP 认证上下文，在请求执行之前或者当请求被执行后，检查他的状态：

```
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 Preemptive authentication

HttpClient 的不支持超前验证开箱，因为如果滥用或使用不当的超前身份验证可以导致严重的安全问题，如明文发送用户凭据的未经授权的第三方。因此，用户将评估在他们的特定应用环境中先发制人的认证与安全风险的潜在利益，并需添加先发制人的身份验证支持使用标准的 HttpClient 扩展机制，例如拦截协议。

这是一个简单的协议拦截器，抢先介绍 BasicScheme 实例的执行上下文，举例来说，如果认证没有完成。请注意，此拦截器在标准的认证拦截器之前必须添加协议处理链。

```
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);
            // If not auth scheme has been initialized yet
            if (authState.getAuthScheme() == null) {
                AuthScope authScope = new AuthScope(targetHost.getHostName(),targetHost.getPort());
                // Obtain credentials matching the target host
              Credentials creds = credsProvider.getCredentials(authScope);
                // If found, generate BasicScheme preemptively
                if (creds != null) {
                    authState.setAuthScheme(new BasicScheme());
                    authState.setCredentials(creds);
                }
            }
        }
};
DefaultHttpClient httpclient = new DefaultHttpClient();
// Add as the very first interceptor in the protocol chain
httpclient.addRequestInterceptor(preemptiveAuth, 0);
```

4.8 NTLM Authentication

目前的 HttpClient 不提供对 NTLM 验证模式的支持，可能永远也不会。可能是法律原因而不是技术的原因。但是，启用 NTLM 身份验证可以通过使用外部的 NTLM 引擎，例如 JCIFS [http:/jcifs.samba.org/]类库由 Samba [http:/www.samba.org/]作为 Windows 的一部分的操作性成套的方案。有关详情请参阅 NTLM_SUPPORT.txt 文件包含的 HttpClient 发布版。

4.8.1. NTLM connection persistence

NTLM 验证模式是贵得多的计算开销和性能比标准的基本和摘要模式的影响方面。这可能是为什么微软选择做验证计划状态的主要原因之一。也就是说，一旦通过认证，用户身份是，对于其整个生命跨度连接相关联。连接状态的 NTLM 持久性，使连接更复杂，因为持续的理由很明显，使用 NTLM 连接可能不会再由用户使用不同的用户身份。标准连接与 HttpClient 的发运经理完全有能力管理有状态连接。然而，至关重要的是在同一会期内使用相同的执行上下文，以使他们认识到当前用户的身份在逻辑上相关的要求。否则，最终的 HttpClient 将创建一个新的 HTTP 对 NTLM 身份为每个 HTTP 请求连接保护的资源。有关 HTTP 连接状态的详细讨论请参阅本节。

由于 NTLM 身份连接状态，一般建议使用 NTLM 身份验证触发一个相对廉价的方法，例如 GET 或 HEAD 和重新使用相同的连接来执行更昂贵的方法，特别是那些要求附上一个实体，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");
// Make sure the same context is used to execute logically related requests
HttpContext localContext = new BasicHttpContext();
// Execute a cheap method first. This will trigger NTLM authentication
HttpGet httpget = new HttpGet("/ntlm-protected/info");
```

```
HttpResponse response1 = httpclient.execute(target, httpget, localContext);
HttpEntity entity1 = response1.getEntity();
if (entity1 != null) {
    entity1.consumeContent();
}
// Execute an expensive method next reusing the same context (and connection)
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 client service（HTTP客户服务）

5.1 HTTP FACADE

HttpClient 的接口代表了最重要的 HTTP 请求执行合同。它并没有强加限制或执行请求的过程特殊的细节和交付给连接管理，状态管理，认证和重新处理了个别实现的细节。这应该更容易装饰，如响应内容缓存等附加功能的接口。

DefaultHttpClient 是 HttpClient 的接口的默认实现。作为一个特殊用途的处理程序或接口实现策略的一个具体方面的 HTTP 协议处理，如处理或重定向或验证作出有关决定，并保持连接持续时间或着负责人数外观这个类的行为。这使用户有选择性地取代默认与自定义这些方面的执行情况，具体的应用。

```
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) {
            // Keep connections alive 5 seconds if a keep-alive value
            // has not be explicitly set by the server
            keepAlive = 5000;
        }
        return keepAlive;
    }

});
```

efaultHttpClient 还维护拦截器的协议列表处理传入请求和传出响应，并提供管理那些拦截器的方法。新

的协议拦截器可以引进到协议处理器链或从中删除如果需要的话。内部协议拦截器存储在一个简单的java.util.ArrayList。它们按照已添加到列表中顺序被执行。

```
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。建议类的同一个实例是复用多个请求。当一个 DefaultHttpClient 实例不再需要和超出作用域，与它相关的连接管理器必须关闭调用 ClientConnectionManager#shutdown()方法。

```
HttpClient httpclient = new DefaultHttpClient();
// Do something useful
httpclient.getConnectionManager().shutdown();
```

5.2 HttpClient parameters

这些参数用于 HttpClient 默认执行自定义行为：

- 'http.protocol.handle-redirects'：定义是否应自动处理重定向。此参数期望是 java.lang.Boolean 类型的值。如果此参数没有定义 HttpClient 将不会自动处理重定向。
- 'http.protocol.reject-relative-redirect'：定义是否相对重定向应予以拒绝。 HTTP 规范规定位置值是一个绝对的 URI。此参数期望是 java.lang.Boolean 类型的值。如果该参数没有设置相对重定向将被允许。
- 'http.protocol.max-redirects'：定义的最大数量应遵循的重定向。关于重定向的数量限制是为了防止破坏服务器端脚本造成无限循环。预计此参数类型 java.lang.Integer 价值。如果不设置此参数不超过 100 重定向将被允许。
- 'http.protocol.allow-circular-redirects'：定义是否（循环重定向重定向到相同的位置）应该被允许。的 HTTP 规范不够明确，是否允许重定向循环，因此他们可以选择启用。预计此参数类型 java.lang.Boolean 的价值。如果该参数没有设置循环重定向将被禁止。
- 'http.connection-manager.factory-class-name'：定义了默认的 ClientConnectionManager 实现类的名称。此参数预计 java.lang.String 类型的值。如果不设置此参数为默认将使用 SingleClientConnManager。
- 'http.virtual-host'：定义的虚拟主机名，而不是在物理主机使用主机头名。预计此参数类型 HttpHost 价值。如果未设置此参数的名称或 IP 地址将目标主机使用。
- 'http.default-headers'：定义了请求为每发送的每个请求的默认标题。预计此参数的类型包含头对象为
  java.util.Collection 价值。
- 'http.default-host'：定义的默认主机。默认值将用于如果目标主机不明确的请求的 URI（相对 URI）中指定。预计此参数类型 HttpHost 价值。

5.3 Automcatic redirect handling

HttpClient 处理所有类型的重定向，除了那些显式地需要用户干预的 HTTP 规范所禁止的。见其他（状态代码 303）重定向的 GET 转换按照 HTTP 规范所要求的 POST 和 PUT 请求。

5.4 HTTP client and execution context

该 DefaultHttpClient 视为应该是从来没有改变的请求执行过程中的不可变对象的 HTTP 请求。相反，它创造了一个原始请求对象，其可以更新取决于执行上下文属性私有可变的副本。因此，最终的请求，例如目标主机和请求 URI 可以通过检查本地 HTTP 上下文内容确定后，要求被执行。

```
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 Advanced topics（高级主题）

6.1. Custom client connections

在某些情况下，定制 HTTP 消息通过线路被传输的方式是必要的，而不是为了非标准，非遵守的行为而使用 Http 参数。对于 web 爬虫来说，为了抢救消息的内容，强迫 HttpClient 接受畸形的头反应是有必要的。

通常插件在习惯的消息解析或者习惯的连接应用的处理涉及到几个步骤。

- 提供一个自定义的 LineParser/LineFormatter 接口实现，实现消息解析/格式化必需的逻辑。

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

```
        }
    }
}
```

- 提供一个自定义的 OperatedClientConnection 实现，替换缺省的请求和响应解析，请求和响应格式化，如果有必要的话，实现不同的消息读写代码。

```
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);
    }
}
```

- 提供一个自定义 ClientConnectionOperator 接口实现为了创建一个新连接，如果有必要的话，实现 socket 初始化代码。

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

- 提供一个自定义 ClientConnectionManager 接口实现为了创建新类的连接操作。

```
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. Stateful HTTP connections

HTTP 详细说明如果会话状态信息总是以 HTTP cookie 的形式嵌入 HTTP 消息中，因此 HTTP 连接总是无状态的，这个假象在真实的生活中不总是有效。这种情况下，HTTP 连接被一个特别的用户创建或者在一个特别的安全环境下被创建，因此不能被其他用户分享，只能被同一个用户重用。有状态的 HTTP 连接的例子是 NTLM 认证连接和 SSL 连接带有客户凭证认证。

6.2.1. User token handler

如果特定的执行上下文具有用户或者没有，HTTPClient 依靠 UserTokenHandler 接口来确定。处理返回的标记对象唯一标示当前用户,是否上下文具有用户或者为空。是否上下文不包含任何资源或者当前用户具体的细节。用户令牌被用来确定用户具体的资源不能被分享或者被其他用户重用。

UserTokenHandler 接口缺省实现使用一个 Principal 类的实例来代表 HTTP 连接的一个状态对象，如果它从特定的执行的上下文中获得，DefaultUserTokenHandler 将在认证模式基础上使用用户连接规则。例如 NTLM 或者 SSL 会话打开客户认证，如果两者都不可用，空令牌将会返回。

如果默认是实现不能满足需求，用户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. User token and execution context

在 HTTP 请求执行过程中，HTTPClient 添加以下与用户识别相关对象来执行上下文。

- 'http.user-token'：对象实例代表实际用户标识。通常要求是 Principle 接口的一个实例。

在请求被执行以后，通过检查本地 HTTP 上下文的内容。你可以查明是否被用来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. Persistent stateful connections

请注意，携带一个状态对象的持续连接不能被用，只有当请求被执行，同一个状态对象绑定到执行上下文，连接才能用。所以确保相同的上下文被相同的用户用来执行连续的 HTTP 请求，或者用户令牌被绑定到上下文优先于请求被执行很重要。

```
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();
}
```