*转自：*http://www.yeetrack.com/?p=779

前言

Http 协议应该是互联网中最重要的协议。持续增长的 web 服务、可联网的家用电器等都在继承并拓展着 Http 协议，向着浏览器之外的方向发展。

虽然 jdk 中的 java.net 包中提供了一些基本的方法，通过 http 协议来访问网络资源，但是大多数场景下，它都不够灵活和强大。HttpClient 致力于填补这个空白，它可以提供有效的、最新的、功能丰富的包来实现 http 客户端。

为了拓展，HttpClient 即支持基本的 http 协议，还支持 http-aware 客户端程序，如 web 浏览器，Webservice 客户端，以及利用 or 拓展 http 协议的分布式系统。

1、HttpClient 的范围/特性

- 是一个基于 HttpCore 的客户端 Http 传输类库
- 基于传统的（阻塞）IO
- 内容无关

2、HttpClient 不能做的事情

- HttpClient 不是浏览器，它是一个客户端 http 协议传输类库。HttpClient 被用来发送和接受 Http 消息。HttpClient 不会处理 http 消息的内容，不会进行 javascript 解析，不会关心 content type，如果没有明确设置，httpclient 也不会对请求进行格式化、重定向 url，或者其他任何和 http 消息传输相关的功能。

第一章 基本概念

1.1. 请求执行

HttpClient 最基本的功能就是执行 Http 方法。一个 Http 方法的执行涉及到一个或者多个 Http 请求/Http 响应的交互，通常这个过程都会自动被 HttpClient 处理，对用户透明。用户只需要提供 Http 请求对象，HttpClient 就会将 http 请求发送给目标服务器，并且接收服务器的响应，如果 http 请求执行不成功，httpclient 就会抛出异样。

下面是个很简单的 http 请求执行的例子：

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

1.1.1. HTTP 请求

所有的 Http 请求都有一个请求行(request line)，包括方法名、请求的 URI 和 Http 版本号。

HttpClient 支持 HTTP/1.1 这个版本定义的所有 Http 方法：GET,HEAD,POST,PUT,DELETE,TRACE 和 OPTIONS。对于每一种 http 方法，HttpClient 都定义了一个相应的类：HttpGet, HttpHead, HttpPost, HttpPut, HttpDelete, HttpTrace 和 HttpOpquertions。

Request-URI 即统一资源定位符，用来标明 Http 请求中的资源。Http request URIs 包含协议名、主机名、主机端口（可选）、资源路径、query（可选）和片段信息（可选）。

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

HttpClient 提供 URIBuilder 工具类来简化 URIs 的创建和修改过程。

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

上述代码会在控制台输出：

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

1.1.2. HTTP 响应

服务器收到客户端的 http 请求后，就会对其进行解析，然后把响应发给客户端，这个响应就是 HTTP response.HTTP 响应第一行是协议版本，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());
```

上述代码会在控制台输出：

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

1.1.3. 消息头

一个 Http 消息可以包含一系列的消息头，用来对 http 消息进行描述，比如消息长度，消息类型等等。HttpClient 提供了方法来获取、添加、移除、枚举消息头。

```
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);
```

上述代码会在控制台输出：

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

最有效的获取指定类型的消息头的方法还是使用 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());
}
```

上述代码会在控制台输出：

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

HeaderIterator 也提供非常便捷的方式，将 Http 消息解析成单独的消息头元素。

```
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]);
        }
}
```

上述代码会在控制台输出：

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

1.1.4. HTTP 实体

Http 消息可以携带 http 实体，这个 http 实体既可以是 http 请求，也可以是 http 响应的。Http 实体，可以在某些 http 请求或者响应中发现，但不是必须的。Http 规范中定义了两种包含请求的方法：POST 和 PUT。HTTP 响应一般会包含一个内容实体。当然这条规则也有异常情况，如 Head 方法的响应，204 没有内容，304 没有修改或者 205 内容资源重置。

HttpClient 根据来源的不同，划分了三种不同的 Http 实体内容。

- **streamed 流式:** 内容是通过流来接受或者在运行中产生。特别是，streamed 这一类包含从 http 响应中获取的实体内容。一般说来，streamed 实体是不可重复的。
- **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。self-contained 类型的实体内容通常是可重复的。这种类型的实体通常用于关闭 http 请求。

- **wrapping 包装式:** 这种类型的内容是从另外的 http 实体中获取的。

当从 Http 响应中读取内容时，上面的三种区分对于连接管理器来说是非常重要的。对于由应用程序创建而且只使用 HttpClient 发送的请求实体，streamed 和 self-contained 两种类型的不同就不那么重要了。这种情况下，建议考虑如 streamed 流式这种不能重复的实体，和可以重复的 self-contained 自我包含式实体。

1.1.4.1. 可重复的实体

一个实体是可重复的，也就是说它的包含的内容可以被多次读取。这种多次读取只有 self contained（自包含）的实体能做到（比如 ByteArrayEntity 或者 StringEntity)。

1.1.4.2. 使用 Http 实体

由于一个 Http 实体既可以表示二进制内容，又可以表示文本内容，所以 Http 实体要支持字符编码（为了支持后者，即文本内容）。

当需要执行一个完整内容的 Http 请求或者 Http 请求已经成功，服务器要发送响应到客户端时，Http 实体就会被创建。

如果要从 Http 实体中读取内容，我们可以利用 HttpEntity 类的 getContent 方法来获取实体的输入流（java.io.InputStream)，或者利用 HttpEntity 类的 writeTo(OutputStream)方法来获取输出流，这个方法会把所有的内容写入到给定的流中。
当实体类已经被接受后，我们可以利用 HttpEntity 类的 getContentType()和 getContentLength()方法来读取 Content-Type 和 Content-Length 两个头消息（如果有的话）。由于 Content-Type 包含 mime-types 的字符编码，比如 text/plain 或者 text/html,HttpEntity 类的 getContentEncoding()方法就是读取这个编码的。如果头信息不存在，getContentLength（）会返回-1,getContentType()会返回 NULL。如果 Content-Type 信息存在，就会返回一个 Header 类。

当为发送消息创建 Http 实体时，需要同时附加 meta 信息。

```
1.  StringEntity myEntity = new StringEntity("important message",
2.     ContentType.create("text/plain", "UTF-8"));
3.
4.  System.out.println(myEntity.getContentType());
5.  System.out.println(myEntity.getContentLength());
6.  System.out.println(EntityUtils.toString(myEntity));
7.  System.out.println(EntityUtils.toByteArray(myEntity).length);
```

上述代码会在控制台输出：

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

1.1.5. 确保底层的资源连接被释放

为了确保系统资源被正确地释放，我们要么管理 Http 实体的内容流、要么关闭 Http 响应。

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

关闭 Http 实体内容流和关闭 Http 响应的区别在于，前者通过消耗掉 Http 实体内容来保持相关的 http 连接，然后后者会立即关闭、丢弃 http 连接。

请注意 HttpEntity 的 writeTo(OutputStream)方法，当 Http 实体被写入到 OutputStream 后，也要确保释放系统资源。如果这个方法内调用了 HttpEntity 的 getContent()方法，那么它会有一个 java.io.InpputStream 的实例，我们需要在 finally 中关闭这个流。

但是也有这样的情况，我们只需要获取 Http 响应内容的一小部分，而获取整个内容并、实现连接的可重复性代价太大，这时我们可以通过关闭响应的方式来关闭内容输入、输出流。

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpGet httpget = new HttpGet("http://localhost/");
```

```
CloseableHttpResponse response = httpclient.execute(httpget);
try {
    HttpEntity entity = response.getEntity();
    if (entity != null) {
        InputStream instream = entity.getContent();
        int byteOne = instream.read();
        int byteTwo = instream.read();
        // Do not need the rest
    }
} finally {
    response.close();
}
```

上面的代码执行后，连接变得不可用，所有的资源都将被释放。

1.1.6. 消耗 HTTP 实体内容

HttpClient 推荐使用 HttpEntity 的 getConent()方法或者 HttpEntity 的 writeTo(OutputStream)方法来消耗掉 Http 实体内容。HttpClient 也提供了 EntityUtils 这个类，这个类提供一些静态方法可以更容易地读取 Http 实体的内容和信息。和以 java.io.InputStream 流读取内容的方式相比，EntityUtils 提供的方法可以以字符串或者字节数组的形式读取 Http 实体。但是，**强烈不推荐使用 EntityUtils 这个类**，除非目标服务器发出的响应是可信任的，并且 http 响应实体的长度不会过大。

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

有些情况下，我们希望可以重复读取 Http 实体的内容。这就需要把 Http 实体内容缓存在内存或者磁盘上。最简单的方法就是把 Http Entity 转化成 BufferedHttpEntity，这样就把原 Http 实体的内容缓冲到了内存中。后面我们就可以重复读取 BufferedHttpEntity 中的内容。

```
1. CloseableHttpResponse response = <...>
2. HttpEntity entity = response.getEntity();
3. if (entity != null) {
4.     entity = new BufferedHttpEntity(entity);
5. }
```

1.1.7. 创建 HTTP 实体内容

HttpClient 提供了一些类，这些类可以通过 http 连接高效地输出 Http 实体内容。HttpClient 提供的这几个类涵盖的常见的数据类型，如 String，byte 数组，输入流，和文件类型：StringEntity,ByteArrayEntity,InputStreamEntity,FileEntity。

```
1. File file = new File("somefile.txt");
2. FileEntity entity = new FileEntity(file,
3.     ContentType.create("text/plain", "UTF-8"));
4.
5. HttpPost httppost = new HttpPost("http://localhost/action.do");
6. httppost.setEntity(entity);
```

请注意由于 InputStreamEntity 只能从下层的数据流中读取一次，所以它是不能重复的。推荐，通过继承 HttpEntity 这个自包含的类来自定义 HttpEntity 类，而不是直接使用 InputStreamEntity 这个类。FileEntity 就是一个很好的起点（FileEntity 就是继承的 HttpEntity）。

1.7.1.1. HTML 表单

很多应用程序需要模拟提交 Html 表单的过程，举个例子，登陆一个网站或者将输入内容提交给服务器。HttpClient 提供了 UrlEncodedFormEntity 这个类来帮助实现这一过程。

```
1. List<NameValuePair> formparams = new ArrayList<NameValuePair>();
2. formparams.add(new BasicNameValuePair("param1", "value1"));
3. formparams.add(new BasicNameValuePair("param2", "value2"));
4. UrlEncodedFormEntity entity = new UrlEncodedFormEntity(formparams, Consts.UT
   F_8);
5. HttpPost httppost = new HttpPost("http://localhost/handler.do");
6. httppost.setEntity(entity);
```

UrlEncodedFormEntity 实例会使用所谓的 Url 编码的方式对我们的参数进行编码，产生的结果如下：

```
param1=value1&m2=value2
```

1.1.7.2. 内容分块

一般来说，推荐让 HttpClient 自己根据 Http 消息传递的特征来选择最合适的传输编码。当然，如果非要手动控制也是可以的，可以通过设置 HttpEntity 的 setChunked()为 true。请注意：HttpClient 仅会将这个参数看成是一个建议。如果 Http 的版本（如 http 1.0)不支持内容分块，那么这个参数就会被忽略。

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

1.1.8. RESPONSE HANDLERS

最简单也是最方便的处理 http 响应的方法就是使用 ResponseHandler 接口，这个接口中有 handleResponse(HttpResponse response)方法。使用这个方法，用户完全不用关心 http 连接管理器。当使用 ResponseHandler 时，HttpClient 会自动地将 Http 连接释放给 Http 管理器，即使 http 请求失败了或者抛出了异常。

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpGet httpget = new HttpGet("http://localhost/json");

ResponseHandler<MyJsonObject> rh = new ResponseHandler<MyJsonObject>() {

    @Override
    public JsonObject handleResponse(
            final HttpResponse response) throws IOException {
        StatusLine statusLine = response.getStatusLine();
        HttpEntity entity = response.getEntity();
        if (statusLine.getStatusCode() >= 300) {
            throw new HttpResponseException(
                    statusLine.getStatusCode(),
                    statusLine.getReasonPhrase());
        }
        if (entity == null) {
```

```
            throw new ClientProtocolException("Response contains no content");
        }
        Gson gson = new GsonBuilder().create();
        ContentType contentType = ContentType.getOrDefault(entity);
        Charset charset = contentType.getCharset();
        Reader reader = new InputStreamReader(entity.getContent(), charset);
        return gson.fromJson(reader, MyJsonObject.class);
    }
};
MyJsonObject myjson = client.execute(httpget, rh);
```

1.2. HttpClient 接口

对于 Http 请求执行过程来说，HttpClient 的接口有着必不可少的作用。HttpClient 接口没有对 Http 请求的过程做特别的限制和详细的规定，连接管理、状态管理、授权信息和重定向处理这些功能都单独实现。这样用户就可以更简单地拓展接口的功能（比如缓存响应内容）。

一般说来，HttpClient 实际上就是一系列特殊的 handler 或者说策略接口的实现，这些 handler（测试接口）负责着处理 Http 协议的某一方面，比如重定向、认证处理、有关连接持久性和 keep alive 持续时间的决策。这样就允许用户使用自定义的参数来代替默认配置，实现个性化的功能。

```
ConnectionKeepAliveStrategy keepAliveStrat = 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;
    }

};
CloseableHttpClient httpclient = HttpClients.custom()
        .setKeepAliveStrategy(keepAliveStrat)
        .build();
```

1.2.1. HTTPCLIENT 的线程安全性

HttpClient 已经实现了线程安全。所以希望用户在实例化 HttpClient 时，也要支持为多个请求使用。

1.2.2. HTTPCLIENT 的内存分配

当一个 CloseableHttpClient 的实例不再被使用，并且它的作用范围即将失效，和它相关的连接必须被关闭，关闭方法可以调用 CloseableHttpClient 的 close()方法。

```
CloseableHttpClient httpclient = HttpClients.createDefault();
try {
    <...>
} finally {
    httpclient.close();
}
```

1.3. Http 执行上下文

最初，Http 被设计成一种无状态的、面向请求-响应的协议。然而，在实际使用中，我们希望能够在一些逻辑相关的请求-响应中，保持状态信息。为了使应用程序可以保持 Http 的持续状态，HttpClient 允许 http 连接在特定的 Http 上下文中执行。如果在持续的 http 请求中使用了同样的上下文，那么这些请求就可以被分配到一个逻辑会话中。HTTP 上下文就和一个 java.util.Map<String, Object>功能类似。它实际上就是一个任意命名的值的集合。应用程序可以在 Http 请求执行前填充上下文的值，也可以在请求执行完毕后检查上下文。

HttpContext 可以包含任意类型的对象，因此如果在多线程中共享上下文会不安全。推荐每个线程都只包含自己的 http 上下文。

在 Http 请求执行的过程中，HttpClient 会自动添加下面的属性到 Http 上下文中：

- HttpConnection 的实例，表示客户端与服务器之间的连接
- HttpHost 的实例，表示要连接的目标服务器
- HttpRoute 的实例，表示全部的连接路由
- HttpRequest 的实例，表示 Http 请求。在执行上下文中，最终的 HttpRequest 对象会代表 http 消息的状态。Http/1.0 和 Http/1.1 都默认使用相对的 uri。但是如果使用了非隧道模式的代理服务器，就会使用绝对路径的 uri。
- HttpResponse 的实例，表示 Http 响应
- java.lang.Boolean 对象，表示是否请求被成功的发送给目标服务器
- RequestConfig 对象，表示 http request 的配置信息
- java.util.List<Uri>对象，表示 Http 响应中的所有重定向地址

我们可以使用 HttpClientContext 这个适配器来简化和上下文交互的过程。

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

同一个逻辑会话中的多个 Http 请求，应该使用相同的 Http 上下文来执行，这样就可以自动地在 http 请求中传递会话上下文和状态信息。

在下面的例子中，我们在开头设置的参数，会被保存在上下文中，并且会应用到后续的 http 请求中。

```
CloseableHttpClient httpclient = HttpClients.createDefault();
RequestConfig requestConfig = RequestConfig.custom()
        .setSocketTimeout(1000)
        .setConnectTimeout(1000)
        .build();

HttpGet httpget1 = new HttpGet("http://localhost/1");
httpget1.setConfig(requestConfig);
CloseableHttpResponse response1 = httpclient.execute(httpget1, context);
try {
    HttpEntity entity1 = response1.getEntity();
} finally {
    response1.close();
}
HttpGet httpget2 = new HttpGet("http://localhost/2");
CloseableHttpResponse response2 = httpclient.execute(httpget2, context);
try {
    HttpEntity entity2 = response2.getEntity();
} finally {
    response2.close();
}
```

1.4. 异常处理

HttpClient 会被抛出两种类型的异常，一种是 java.io.IOException，当遇到 I/O 异常时抛出（socket 超时，或者 socket 被重置）;另一种是 HttpException,表示 Http 失败，如 Http 协议使用不正确。通常认为，I/O 错误时不致命、可修复的，而 Http 协议错误是致命了，不能自动修复的错误。

1.4.1. HTTP 传输安全

Http 协议不能满足所有类型的应用场景，我们需要知道这点。Http 是个简单的面向协议的请求/响应的协议，当初它被设计用来支持静态或者动态生成的内容检索，之前从来没有人想过让它支持事务性操作。例如，Http 服务器成功接收、处理请求后，生成响应消息，并且把状态码发送给客户端，这个过程是 Http 协议应该保证的。但是，如果客户端由于读取超时、取消请求或者系统崩溃导致接收响应失败，服务器不会回滚这一事务。如果客户端重新发送这个请求，服务器就会重复的解析、执行这个事务。在一些情况下，这会导致应用程序的数据损坏和应用程序的状态不一致。

即使 Http 当初设计是不支持事务操作，但是它仍旧可以作为传输协议为某些关键程序提供服务。为了保证 Http 传输层的安全性，系统必须保证应用层上的 http 方法的幂等性。

1.4.2. 方法的幂等性

HTTP/1.1 规范中是这样定义幂等方法的，Methods can also have the property of “idempotence” in that (aside from error or expiration issues) the side-effects of N > 0 identical requests is the same as for a single request。用其他话来说，应用程序需要正确地处理同一方法多次执行造成的影响。添加一个具有唯一性的 id 就能避免重复执行同一个逻辑请求，问题解决。

请知晓，这个问题不只是 HttpClient 才会有，基于浏览器的应用程序也会遇到 Http 方法不幂等的问题。

HttpClient 默认把非实体方法 get、head 方法看做幂等方法，把实体方法 post、put 方法看做非幂等方法。

1.4.3. 异常自动修复

默认情况下，HttpClient 会尝试自动修复 I/O 异常。这种自动修复仅限于修复几个公认安全的异常。

- HttpClient 不会尝试修复任何逻辑或者 http 协议错误（即从 HttpException 衍生出来的异常）。
- HttpClient 会自动再次发送幂等的方法（如果首次执行失败)。
- HttpClient 会自动再次发送遇到 transport 异常的方法，前提是 Http 请求仍旧保持着连接（例如 http 请求没有全部发送给目标服务器，HttpClient 会再次尝试发送）。

1.4.4. 请求重试 HANDLER

如果要自定义异常处理机制，我们需要实现 HttpRequestRetryHandler 接口。

```
1.  HttpRequestRetryHandler myRetryHandler = new HttpRequestRetryHandler() {
2.
3.      public boolean retryRequest(
```

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

1.5. 终止请求

有时候由于目标服务器负载过高或者客户端目前有太多请求积压，http 请求不能在指定时间内执行完毕。这时候终止这个请求，释放阻塞 I/O 的进程，就显得很必要。通过 HttpClient 执行的 Http 请求，在任何状态下都能通过调用 HttpUriRequest 的 abort()方法来终止。这个方法是线程安全的，并且能在任何线程中调用。当 Http 请求被终止了，本线程（即使现在正在阻塞 I/O）也会通过抛出一个 InterruptedIOException 异常，来释放资源。

1.6. Http 协议拦截器

HTTP 协议拦截器是一种实现一个特定的方面的 HTTP 协议的代码程序。通常情况下，协议拦截器会将一个或多个头消息加入到接受或者发送的消息中。协议拦截器也可以操作消息的内容实体—消息内容的压缩/解压缩就是个很好的例子。通常，这是通过使用"装饰"开发模式，一个包装实体类用于装饰原来的实体来实现。一个拦截器可以合并，形成一个逻辑单元。

协议拦截器可以通过共享信息协作——比如处理状态——通过 HTTP 执行上下文。协议拦截器可以使用 Http 上下文存储一个或者多个连续请求的处理状态。

通常，只要拦截器不依赖于一个特定状态的 http 上下文，那么拦截执行的顺序就无所谓。如果协议拦截器有相互依赖关系，必须以特定的顺序执行，那么它们应该按照特定的顺序加入到协议处理器中。

协议处理器必须是线程安全的。类似于 servlets，协议拦截器不应该使用变量实体，除非访问这些变量是同步的（线程安全的）。

下面是个例子，讲述了本地的上下文时如何在连续请求中记录处理状态的：

```
1.  CloseableHttpClient httpclient = HttpClients.custom()
2.          .addInterceptorLast(new HttpRequestInterceptor() {
3.
4.              public void process(
5.                      final HttpRequest request,
6.                      final HttpContext context) throws HttpException, IOExcep
    tion {
7.                  AtomicInteger count = (AtomicInteger) context.getAttribute("
    count");
8.                  request.addHeader("Count", Integer.toString(count.getAndIncr
    ement()));
9.              }
10.
11.         })
12.         .build();
13.
14. AtomicInteger count = new AtomicInteger(1);
15. HttpClientContext localContext = HttpClientContext.create();
16. localContext.setAttribute("count", count);
17.
18. HttpGet httpget = new HttpGet("http://localhost/");
19. for (int i = 0; i < 10; i++) {
20.     CloseableHttpResponse response = httpclient.execute(httpget, localContex
    t);
21.     try {
22.         HttpEntity entity = response.getEntity();
23.     } finally {
24.         response.close();
25.     }
```

```
}
```

上面代码在发送 http 请求时，会自动添加 Count 这个 header，可以使用 wireshark 抓包查看。

1.7.1. 重定向处理

HttpClient 会自动处理所有类型的重定向，除了那些 Http 规范明确禁止的重定向。See Other (status code 303) redirects on POST and PUT requests are converted to GET requests as required by the HTTP specification. 我们可以使用自定义的重定向策略来放松 Http 规范对 Post 方法重定向的限制。

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

HttpClient 在请求执行过程中，经常需要重写请求的消息。 HTTP/1.0 和 HTTP/1.1 都默认使用相对的 uri 路径。同样，原始的请求可能会被一次或者多次的重定向。最终结对路径的解释可以使用最初的请求和上下文。URIUtils 类的 resolve 方法可以用于将拦截的绝对路径构建成最终的请求。这个方法包含了最后一个分片标识符或者原始请求。

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

    // Expected to be an absolute URI
} finally {
    response.close();
}
```

第二章 连接管理

2.1. 持久连接

两个主机建立连接的过程是很复杂的一个过程，涉及到多个数据包的交换，并且也很耗时间。Http 连接需要的三次握手开销很大，这一开销对于比较小的 http 消息来说更大。但是如果我们直接使用已经建立好的 http 连接，这样花费就比较小，吞吐率更大。

HTTP/1.1 默认就支持 Http 连接复用。兼容 HTTP/1.0 的终端也可以通过声明来保持连接，实现连接复用。HTTP 代理也可以在一定时间内保持连接不释放，方便后续向这个主机发送 http 请求。这种保持连接不释放的情况实际上是建立的持久连接。HttpClient 也支持持久连接。

2.2. HTTP 连接路由

HttpClient 既可以直接、又可以通过多个中转路由（hops）和目标服务器建立连接。HttpClient 把路由分为三种 plain（明文 ），tunneled（隧道）和 layered（分层）。隧道连接中使用的多个中间代理被称作代理链。

客户端直接连接到目标主机或者只通过了一个中间代理，这种就是 Plain 路由。客户端通过第一个代理建立连接，通过代理链 tunnelling，这种情况就是 Tunneled 路由。不通过中间代理的路由不可能时 tunneled 路由。客户端在一个已经存在的连接上进行协议分层，这样建立起来的路由就是 layered 路由。协议只能在隧道—>目标主机，或者直接连接（没有代理），这两种链路上进行分层。

2.2.1. 路由计算

RouteInfo 接口包含了数据包发送到目标主机过程中，经过的路由信息。HttpRoute 类继承了 RouteInfo 接口，是 RouteInfo 的具体实现，这个类是不允许修改的。HttpTracker 类也实现了 RouteInfo 接口，它是可变的，HttpClient 会在内部使用这个类来探测到目标主机的剩余路由。HttpRouteDirector 是个辅助类，可以帮助计算数据包的下一步路由信息。这个类也是在 HttpClient 内部使用的。

HttpRoutePlanner 接口可以用来表示基于 http 上下文情况下，客户端到服务器的路由计算策略。HttpClient 有两个 HttpRoutePlanner 的实现类。SystemDefaultRoutePlanner 这个类基于 java.net.ProxySelector，它默认使用 jvm 的代理配置信息，这个配置信息一般来自系统配置或者浏览器配置。DefaultProxyRoutePlanner 这个类既不使用 java 本身的配置，也不使用系统或者浏览器的配置。它通常通过默认代理来计算路由信息。

2.2.2. 安全的 HTTP 连接

为了防止通过 Http 消息传递的信息不被未授权的第三方获取、截获，Http 可以使用 SSL/TLS 协议来保证 http 传输安全，这个协议是当前使用最广的。当然也可以使用其他的加密技术。但是通常情况下，Http 信息会在加密的 SSL/TLS 连接上进行传输。

2.3. HTTP 连接管理器

2.3.1. 管理连接和连接管理器

Http 连接是复杂，有状态的，线程不安全的对象，所以它必须被妥善管理。一个 Http 连接在同一时间只能被一个线程访问。HttpClient 使用一个叫做 Http 连接管理器的特殊实体类来管理 Http 连接，这个实体类要实现 HttpClientConnectionManager 接口。Http 连接管理器在新建 http 连接时，作为工厂类;管理持久 http 连接的生命周期;同步持久连接（确保线程安全，即一个 http 连接同一时间只能被一个线程访问）。Http 连接管理器和 ManagedHttpClientConnection 的实例类一起发挥作用，ManagedHttpClientConnection 实体类可以看做 http 连接的一个代理服务器，管理着 I/O 操作。如果一个 Http 连接被释放或者被它的消费者明确表示要关闭，那么底层的连接就会和它的代理进行分离，并且该连接会被交还给连接管理器。这是，即使服务消费者仍然持有代理的引用，它也不能再执行 I/O 操作，或者更改 Http 连接的状态。

下面的代码展示了如何从连接管理器中取得一个 http 连接：

```
1.  HttpClientContext context = HttpClientContext.create();
2.  HttpClientConnectionManager connMrg = new BasicHttpClientConnectionManager();
3.  HttpRoute route = new HttpRoute(new HttpHost("localhost", 80));
4.  // 获取新的连接. 这里可能耗费很多时间
5.  ConnectionRequest connRequest = connMrg.requestConnection(route, null);
6.  // 10 秒超时
7.  HttpClientConnection conn = connRequest.get(10, TimeUnit.SECONDS);
8.  try {
9.      // 如果创建连接失败
10.     if (!conn.isOpen()) {
11.         // establish connection based on its route info
12.         connMrg.connect(conn, route, 1000, context);
13.         // and mark it as route complete
14.         connMrg.routeComplete(conn, route, context);
15.     }
16.      // 进行自己的操作.
17. } finally {
18.     connMrg.releaseConnection(conn, null, 1, TimeUnit.MINUTES);
19. }
```

如果要终止连接，可以调用 ConnectionRequest 的 cancel()方法。这个方法会解锁被 ConnectionRequest 类 get()方法阻塞的线程。

2.3.2. 简单连接管理器

BasicHttpClientConnectionManager 是个简单的连接管理器，它一次只能管理一个连接。尽管这个类是线程安全的，它在同一时间也只能被一个线程使用。BasicHttpClientConnectionManager 会尽量重用旧的连接来发送后续的请求，并且使用相同的路由。如果后续请求的路由和旧连接中的路由不匹

配，BasicHttpClientConnectionManager 就会关闭当前连接，使用请求中的路由重新建立连接。如果当前的连接正在被占用，会抛出 java.lang.IllegalStateException 异常。

2.3.3. 连接池管理器

相对 BasicHttpClientConnectionManager 来说，PoolingHttpClientConnectionManager 是个更复杂的类，它管理着连接池，可以同时为很多线程提供 http 连接请求。Connections are pooled on a per route basis.当请求一个新的连接时，如果连接池有有可用的持久连接，连接管理器就会使用其中的一个，而不是再创建一个新的连接。

PoolingHttpClientConnectionManager 维护的连接数在每个路由基础和总数上都有限制。默认，每个路由基础上的连接不超过 2 个，总连接数不能超过 20。在实际应用中，这个限制可能会太小了，尤其是当服务器也使用 Http 协议时。

下面的例子演示了如果调整连接池的参数：

```
1.  PoolingHttpClientConnectionManager cm = new PoolingHttpClientConnectionManag
    er();
2.  // Increase max total connection to 200
3.  cm.setMaxTotal(200);
4.  // Increase default max connection per route to 20
5.  cm.setDefaultMaxPerRoute(20);
6.  // Increase max connections for localhost:80 to 50
7.  HttpHost localhost = new HttpHost("locahost", 80);
8.  cm.setMaxPerRoute(new HttpRoute(localhost), 50);
9.
10. CloseableHttpClient httpClient = HttpClients.custom()
11.         .setConnectionManager(cm)
12.         .build();
```

2.3.4. 关闭连接管理器

当一个 HttpClient 的实例不在使用，或者已经脱离它的作用范围，我们需要关掉它的连接管理器，来关闭掉所有的连接，释放掉这些连接占用的系统资源。

```
1.  CloseableHttpClient httpClient = <...>
2.  httpClient.close();
```

2.4. 多线程请求执行

当使用了请求连接池管理器（比如 PoolingClientConnectionManager）后，HttpClient 就可以同时执行多个线程的请求了。

PoolingClientConnectionManager 会根据它的配置来分配请求连接。如果连接池中的所有连接都被占用了，那么后续的请求就会被阻塞，直到有连接被释放回连接池中。为了防止永远阻塞的情况发生，我们可以把 http.conn-manager.timeout 的值设置成一个整数。如果在超时时间内，没有可用连接，就会抛出 ConnectionPoolTimeoutException 异常。

```
1.  PoolingHttpClientConnectionManager cm = new PoolingHttpClientConnectionManag
    er();
2.  CloseableHttpClient httpClient = HttpClients.custom()
3.          .setConnectionManager(cm)
4.          .build();
5.
6.  // URIs to perform GETs on
7.  String[] urisToGet = {
8.      "http://www.domain1.com/",
9.      "http://www.domain2.com/",
10.     "http://www.domain3.com/",
11.     "http://www.domain4.com/"
12. };
13.
14. // create a thread for each URI
15. GetThread[] threads = new GetThread[urisToGet.length];
16. for (int i = 0; i < threads.length; i++) {
17.     HttpGet httpget = new HttpGet(urisToGet[i]);
18.     threads[i] = new GetThread(httpClient, httpget);
19. }
20.
21. // start the threads
22. for (int j = 0; j < threads.length; j++) {
23.     threads[j].start();
24. }
25.
26. // join the threads
27. for (int j = 0; j < threads.length; j++) {
28.     threads[j].join();
29. }
```

即使 HttpClient 的实例是线程安全的，可以被多个线程共享访问，但是仍旧推荐每个线程都要有自己专用实例的 HttpContext。

下面是 GetThread 类的定义：

```
1.  static class GetThread extends Thread {
2.
3.      private final CloseableHttpClient httpClient;
4.      private final HttpContext context;
5.      private final HttpGet httpget;
6.
```

```
7.      public GetThread(CloseableHttpClient httpClient, HttpGet httpget) {
8.          this.httpClient = httpClient;
9.          this.context = HttpClientContext.create();
10.         this.httpget = httpget;
11.     }
12.
13.     @Override
14.     public void run() {
15.         try {
16.             CloseableHttpResponse response = httpClient.execute(
17.                     httpget, context);
18.             try {
19.                 HttpEntity entity = response.getEntity();
20.             } finally {
21.                 response.close();
22.             }
23.         } catch (ClientProtocolException ex) {
24.             // Handle protocol errors
25.         } catch (IOException ex) {
26.             // Handle I/O errors
27.         }
28.     }
29.
30. }
```

2.5. 连接回收策略

经典阻塞 I/O 模型的一个主要缺点就是只有当组侧 I/O 时，socket 才能对 I/O 事件做出反应。当连接被管理器收回后，这个连接仍然存活，但是却无法监控 socket 的状态，也无法对 I/O 事件做出反馈。如果连接被服务器端关闭了，客户端监测不到连接的状态变化（也就无法根据连接状态的变化，关闭本地的 socket）。

HttpClient 为了缓解这一问题造成的影响，会在使用某个连接前，监测这个连接是否已经过时，如果服务器端关闭了连接，那么连接就会失效。这种过时检查并不是 100%有效，并且会给每个请求增加 10 到 30 毫秒额外开销。唯一一个可行的，且 does not involve a one thread per socket model for idle connections 的解决办法，是建立一个监控线程，来专门回收由于长时间不活动而被判定为失效的连接。这个监控线程可以周期性的调用 ClientConnectionManager 类的 closeExpiredConnections()方法来关闭过期的连接，回收连接池中被关闭的连接。它也可以选择性的调用 ClientConnectionManager 类的 closeIdleConnections()方法来关闭一段时间内不活动的连接。

```
1.  public static class IdleConnectionMonitorThread extends Thread {
2.
3.      private final HttpClientConnectionManager connMgr;
4.      private volatile boolean shutdown;
```

```
5.
6.      public IdleConnectionMonitorThread(HttpClientConnectionManager connMgr)
    {
7.          super();
8.          this.connMgr = connMgr;
9.      }
10.
11.     @Override
12.     public void run() {
13.         try {
14.             while (!shutdown) {
15.                 synchronized (this) {
16.                     wait(5000);
17.                     // Close expired connections
18.                     connMgr.closeExpiredConnections();
19.                     // Optionally, close connections
20.                     // that have been idle longer than 30 sec
21.                     connMgr.closeIdleConnections(30, TimeUnit.SECONDS);
22.                 }
23.             }
24.         } catch (InterruptedException ex) {
25.             // terminate
26.         }
27.     }
28.
29.     public void shutdown() {
30.         shutdown = true;
31.         synchronized (this) {
32.             notifyAll();
33.         }
34.     }
35.
36. }
```

2.6. 连接存活策略

Http 规范没有规定一个持久连接应该保持存活多久。有些 Http 服务器使用非标准的 Keep-Alive 头消息和客户端进行交互，服务器端会保持数秒时间内保持连接。HttpClient 也会利用这个头消息。如果服务器返回的响应中没有包含 Keep-Alive 头消息，HttpClient 会认为这个连接可以永远保持。然而，很多服务器都会在不通知客户端的情况下，关闭一定时间内不活动的连接，来节省服务器资源。在某些情况下默认的策略显得太乐观，我们可能需要自定义连接存活策略。

```
1.  ConnectionKeepAliveStrategy myStrategy = new ConnectionKeepAliveStrategy() {
2.
```

```
    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(
                HttpClientContext.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;
        }
    }

};
CloseableHttpClient client = HttpClients.custom()
        .setKeepAliveStrategy(myStrategy)
        .build();
```

2.7. socket 连接工厂

Http 连接使用 java.net.Socket 类来传输数据。这依赖于 ConnectionSocketFactory 接口来创建、初始化和连接 socket。这样也就允许 HttpClient 的用户在代码运行时，指定 socket 初始化的代码。PlainConnectionSocketFactory 是默认的创建、初始化明文 socket（不加密）的工厂类。

创建 socket 和使用 socket 连接到目标主机这两个过程是分离的，所以我们可以在连接发生阻塞时，关闭 socket 连接。

```
HttpClientContext clientContext = HttpClientContext.create();
PlainConnectionSocketFactory sf = PlainConnectionSocketFactory.getSocketFactory();
Socket socket = sf.createSocket(clientContext);
int timeout = 1000; //ms
```

```
HttpHost target = new HttpHost("localhost");
InetSocketAddress remoteAddress = new InetSocketAddress(
        InetAddress.getByAddress(new byte[] {127,0,0,1}), 80);
sf.connectSocket(timeout, socket, target, remoteAddress, null, clientContext);
```

2.7.1. 安全 SOCKET 分层

LayeredConnectionSocketFactory 是 ConnectionSocketFactory 的拓展接口。分层 socket 工厂类可以在明文 socket 的基础上创建 socket 连接。分层 socket 主要用于在代理服务器之间创建安全 socket。HttpClient 使用 SSLSocketFactory 这个类实现安全 socket，SSLSocketFactory 实现了 SSL/TLS 分层。请知晓，HttpClient 没有自定义任何加密算法。它完全依赖于 Java 加密标准（JCE）和安全套接字（JSEE）拓展。

2.7.2. 集成连接管理器

自定义的 socket 工厂类可以和指定的协议（Http、Https）联系起来，用来创建自定义的连接管理器。

```
ConnectionSocketFactory plainsf = <...>
LayeredConnectionSocketFactory sslsf = <...>
Registry<ConnectionSocketFactory> r = RegistryBuilder.<ConnectionSocketFactory>create()
        .register("http", plainsf)
        .register("https", sslsf)
        .build();

HttpClientConnectionManager cm = new PoolingHttpClientConnectionManager(r);

HttpClients.custom()
        .setConnectionManager(cm)
        .build();
```

2.7.3. SSL/TLS 定制

HttpClient 使用 SSLSocketFactory 来创建 ssl 连接。SSLSocketFactory 允许用户高度定制。它可以接受 javax.net.ssl.SSLContext 这个类的实例作为参数，来创建自定义的 ssl 连接。

```
HttpClientContext clientContext = HttpClientContext.create();
KeyStore myTrustStore = <...>
SSLContext sslContext = SSLContexts.custom()
        .useTLS()
```

```
        .loadTrustMaterial(myTrustStore)
        .build();
SSLConnectionSocketFactory sslsf = new SSLConnectionSocketFactory(sslContext);
```

2.7.4. 域名验证

除了信任验证和在 ssl/tls 协议层上进行客户端认证，HttpClient 一旦建立起连接，就可以选择性验证目标域名和存储在 X.509 证书中的域名是否一致。这种验证可以为服务器信任提供额外的保障。X509HostnameVerifier 接口代表主机名验证的策略。在 HttpClient 中，X509HostnameVerifier 有三个实现类。重要提示：主机名有效性验证不应该和 ssl 信任验证混为一谈。

- StrictHostnameVerifier: 严格的主机名验证方法和 java 1.4,1.5,1.6 验证方法相同。和 IE6 的方式也大致相同。这种验证方式符合 RFC 2818 通配符。The hostname must match either the first CN, or any of the subject-alts. A wildcard can occur in the CN, and in any of the subject-alts.
- BrowserCompatHostnameVerifier: 这种验证主机名的方法，和 Curl 及 firefox 一致。The hostname must match either the first CN, or any of the subject-alts. A wildcard can occur in the CN, and in any of the subject-alts.StrictHostnameVerifier 和 BrowserCompatHostnameVerifier 方式唯一不同的地方就是，带有通配符的域名（比如 *.yeetrack.com),BrowserCompatHostnameVerifier 方式在匹配时会匹配所有的的子域名，包括 a.b.yeetrack.com .
- AllowAllHostnameVerifier: 这种方式不对主机名进行验证，验证功能被关闭，是个空操作，所以它不会抛出 javax.net.ssl.SSLException 异常。HttpClient 默认使用 BrowserCompatHostnameVerifier 的验证方式。如果需要，我们可以手动执行验证方式。

```
SSLContext sslContext = SSLContexts.createSystemDefault();
SSLConnectionSocketFactory sslsf = new SSLConnectionSocketFactory(
        sslContext,
        SSLConnectionSocketFactory.STRICT_HOSTNAME_VERIFIER);
```

2.8. HttpClient 代理服务器配置

尽管，HttpClient 支持复杂的路由方案和代理链，它同样也支持直接连接或者只通过一跳的连接。

使用代理服务器最简单的方式就是，指定一个默认的 proxy 参数。

```
HttpHost proxy = new HttpHost("someproxy", 8080);
DefaultProxyRoutePlanner routePlanner = new DefaultProxyRoutePlanner(proxy);

CloseableHttpClient httpclient = HttpClients.custom()
        .setRoutePlanner(routePlanner)
```

```
5.          .build();
```

我们也可以让 HttpClient 去使用 jre 的代理服务器。

```
1.  SystemDefaultRoutePlanner routePlanner = new SystemDefaultRoutePlanner(
2.          ProxySelector.getDefault());
3.  CloseableHttpClient httpclient = HttpClients.custom()
4.          .setRoutePlanner(routePlanner)
5.          .build();
```

又或者，我们也可以手动配置 RoutePlanner，这样就可以完全控制 Http 路由的过程。

```
1.  HttpRoutePlanner routePlanner = new HttpRoutePlanner() {
2.
3.      public HttpRoute determineRoute(
4.              HttpHost target,
5.              HttpRequest request,
6.              HttpContext context) throws HttpException {
7.          return new HttpRoute(target, null,  new HttpHost("someproxy", 8080),
8.                  "https".equalsIgnoreCase(target.getSchemeName()));
9.      }
10.
11. };
12. CloseableHttpClient httpclient = HttpClients.custom()
13.         .setRoutePlanner(routePlanner)
14.         .build();
15.     }
16. }
```

第三章 Http 状态管理

最初，Http 被设计成一个无状态的，面向请求/响应的协议，所以它不能在逻辑相关的 http 请求/响应中保持状态会话。由于越来越多的系统使用 http 协议，其中包括 http 从来没有想支持的系统，比如电子商务系统。因此，http 支持状态管理就很必要了。

当时的 web 客户端和服务器软件领先者，网景(netscape)公司，最先在他们的产品中支持 http 状态管理，并且制定了一些专有规范。后来，网景通过发规范草案，规范了这一机制。这些努力促成 RFC standard track 制定了标准的规范。但是，现在多数的应用的状态管理机制都在使用网景公司的规范，而网景的规范和官方规定是不兼容的。因此所有的浏览器开发这都被迫兼容这两种协议，从而导致协议的不统一。

3.1. Http cookies

所谓的 Http cookie 就是一个 token 或者很短的报文信息，http 代理和服务器可以通过 cookie 来维持会话状态。网景的工程师把它们称作“magic cookie”。

HttpClient 使用 Cookie 接口来代表 cookie。简单说来，cookie 就是一个键值对。一般，cookie 也会包含版本号、域名、路径和 cookie 有效期。

SetCookie 接口可以代表服务器发给 http 代理的一个 set-cookie 响应头，在浏览器中，这个 set-cookie 响应头可以写入 cookie，以便保持会话状态。SetCookie2 接口对 SetCookie 接口进行了拓展，添加了 Set-Cookie2 方法。

ClientCookie 接口继承了 Cookie 接口，并进行了功能拓展，比如它可以取出服务器发送过来的原始 cookie 的值。生成头消息是很重要的，因为只有当 cookie 被指定为 Set-Cookie 或者 Set-Cookie2 时，它才需要包括一些特定的属性。

3.1.1 COOKIES 版本

兼容网景的规范，但是不兼容官方规范的 cookie，是版本 0. 兼容官方规范的版本，将会是版本 1。版本 1 中的 Cookie 可能和版本 0 工作机制有差异。

下面的代码，创建了网景版本的 Cookie：

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

下面的代码，创建标准版本的 Cookie。注意，标准版本的 Cookie 必须保留服务器发送过来的 Cookie 所有属性。

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

下面的代码，创建了 Set-Cookie2 兼容 cookie。

```
1.  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 规范

CookieSpec 接口代表了 Cookie 管理规范。Cookie 管理规范规定了：

- 解析 Set-Cookie 和 Set-Cookie2(可选）头消息的规则
- 验证 Cookie 的规则
- 将指定的主机名、端口和路径格式化成 Cookie 头消息

HttpClient 有下面几种 CookieSpec 规范：

- Netscape draft: 这种符合网景公司指定的规范。但是尽量不要使用，除非一定要保证兼容很旧的代码。
- Standard: RFC 2965 HTTP 状态管理规范
- Browser compatibility: 这种方式，尽量模仿常用的浏览器，如 IE 和 firefox
- Best match: 'Meta' cookie specification that picks up a cookie policy based on the format of cookies sent with the HTTP response.它基本上将上面的几种规范积聚到一个类中。
  ++ Ignore cookies: 忽略所有 Cookie
  强烈推荐使用 Best Match 匹配规则，让 HttpClient 根据运行时环境自己选择合适的规范。

3.3. 选择 Cookie 策略

我们可以在创建 Http client 的时候指定 Cookie 测试，如果需要，也可以在执行 http 请求的时候，进行覆盖指定。

```
RequestConfig globalConfig = RequestConfig.custom()
        .setCookieSpec(CookieSpecs.BEST_MATCH)
        .build();
CloseableHttpClient httpclient = HttpClients.custom()
        .setDefaultRequestConfig(globalConfig)
        .build();
RequestConfig localConfig = RequestConfig.copy(globalConfig)
```

```
        .setCookieSpec(CookieSpecs.BROWSER_COMPATIBILITY)
        .build();
HttpGet httpGet = new HttpGet("/");
httpGet.setConfig(localConfig);
```

3.4. 自定义 Cookie 策略

如果我们要自定义 Cookie 测试，就要自己实现 CookieSpec 接口，然后创建一个 CookieSpecProvider 接口来新建、初始化自定义 CookieSpec 接口，最后把 CookieSpecProvider 注册到 HttpClient 中。一旦我们注册了自定义策略，就可以像其他标准策略一样使用了。

```
CookieSpecProvider easySpecProvider = new CookieSpecProvider() {

    public CookieSpec create(HttpContext context) {

        return new BrowserCompatSpec() {
            @Override
            public void validate(Cookie cookie, CookieOrigin origin)
                    throws MalformedCookieException {
                // Oh, I am easy
            }
        };
    }

};
Registry<CookieSpecProvider> r = RegistryBuilder.<CookieSpecProvider>create()
        .register(CookieSpecs.BEST_MATCH,
            new BestMatchSpecFactory())
        .register(CookieSpecs.BROWSER_COMPATIBILITY,
            new BrowserCompatSpecFactory())
        .register("easy", easySpecProvider)
        .build();

RequestConfig requestConfig = RequestConfig.custom()
        .setCookieSpec("easy")
        .build();

CloseableHttpClient httpclient = HttpClients.custom()
        .setDefaultCookieSpecRegistry(r)
        .setDefaultRequestConfig(requestConfig)
        .build();
```

3.5. Cookie 持久化

HttpClient 可以使用任何存储方式的 cookie store，只要这个 cookie store 实现了 CookieStore 接口。默认的 CookieStore 通过 java.util.ArrayList 简单实现了 BasicCookieStore。存在在 BasicCookieStore 中的 Cookie，当载体对象被当做垃圾回收掉后，就会丢失。如果必要，用户可以自己实现更为复杂的方式。

```
// Create a local instance of cookie store
CookieStore cookieStore = new BasicCookieStore();
// 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
CloseableHttpClient httpclient = HttpClients.custom()
        .setDefaultCookieStore(cookieStore)
        .build();
```

3.6. HTTP 状态管理和执行上下文

在 Http 请求执行过程中，HttpClient 会自动向执行上下文中添加下面的状态管理对象：

- Lookup 对象 代表实际的 cookie 规范 registry。在当前上下文中的这个值优先于默认值。
- CookieSpec 对象 代表实际的 Cookie 规范。
- CookieOrigin 对象 代表实际的 origin server 的详细信息。
- CookieStore 对象 表示 Cookie store。这个属性集中的值会取代默认值。

本地的 HttpContext 对象可以用来在 Http 请求执行前，自定义 Http 状态管理上下文;或者测试 http 请求执行完毕后上下文的状态。我们也可以在不同的线程中使用不同的执行上下文。我们在 http 请求层指定的 cookie 规范集和 cookie store 会覆盖在 http Client 层级的默认值。

```
CloseableHttpClient httpclient = <...>

Lookup<CookieSpecProvider> cookieSpecReg = <...>
CookieStore cookieStore = <...>

HttpClientContext context = HttpClientContext.create();
context.setCookieSpecRegistry(cookieSpecReg);
context.setCookieStore(cookieStore);
HttpGet httpget = new HttpGet("http://somehost/");
CloseableHttpResponse response1 = httpclient.execute(httpget, context);
<...>
// Cookie origin details
CookieOrigin cookieOrigin = context.getCookieOrigin();
```

```
// Cookie spec used
CookieSpec cookieSpec = context.getCookieSpec();
```

第四章 HTTP 认证

HttpClient 既支持 HTTP 标准规范定义的认证模式，又支持一些广泛使用的非标准认证模式，比如 NTLM 和 SPNEGO。

4.1. 用户凭证

任何用户认证的过程，都需要一系列的凭证来确定用户的身份。最简单的用户凭证可以是用户名和密码这种形式。UsernamePasswordCredentials 这个类可以用来表示这种情况，这种凭据包含明文的用户名和密码。

这个类对于 HTTP 标准规范中定义的认证模式来说已经足够了。

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

上述代码会在控制台输出：

```
user
pwd
```

NTCredentials 是微软的 windows 系统使用的一种凭据，包含 username、password，还包括一系列其他的属性，比如用户所在的域名。在 Microsoft Windows 的网络环境中，同一个用户可以属于不同的域，所以他也就有不同的凭据。

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

上述代码输出：

```
DOMAIN/user
pwd
```

4.2. 认证方案

AutoScheme 接口表示一个抽象的面向挑战/响应的认证方案。一个认证方案要支持下面的功能：

- 客户端请求服务器受保护的资源，服务器会发送过来一个 chanllenge(挑战），认证方案（Authentication scheme）需要解析、处理这个挑战
- 为 processed challenge 提供一些属性值：认证方案的类型，和此方案需要的一些参数，这种方案适用的范围
- 使用给定的授权信息生成授权字符串;生成 http 请求，用来响应服务器发送来过的授权 challenge

请注意：一个认证方案可能是有状态的，因为它可能涉及到一系列的挑战/响应。

HttpClient 实现了下面几种 AutoScheme:

- Basic: Basic 认证方案是在 RFC2617 号文档中定义的。这种授权方案用明文来传输凭证信息，所以它是不安全的。虽然 Basic 认证方案本身是不安全的，但是它一旦和 TLS/SSL 加密技术结合起来使用，就完全足够了。
- Digest: Digest（摘要）认证方案是在 RFC2617 号文档中定义的。Digest 认证方案比 Basic 方案安全多了，对于那些受不了 Basic+TLS/SSL 传输开销的系统，digest 方案是个不错的选择。
- NTLM: NTLM 认证方案是个专有的认证方案，由微软开发，并且针对 windows 平台做了优化。NTLM 被认为比 Digest 更安全。
- SPNEGO: SPNEGO(Simple and Protected GSSAPI Negotiation Mechanism)是 GSSAPI 的一个"伪机制"，它用来协商真正的认证机制。SPNEGO 最明显的用途是在微软的 HTTP 协商认证机制拓展上。可协商的子机制包括 NTLM、Kerberos。目前，HttpCLient 只支持 Kerberos 机制。（原文：The negotiable sub-mechanisms include NTLM and Kerberos supported by Active Directory. At present HttpClient only supports the Kerberos sub-mechanism.）

4.3. 凭证 provider

凭证 providers 旨在维护一套用户的凭证，当需要某种特定的凭证时，providers 就应该能产生这种凭证。认证的具体内容包括主机名、端口号、realm name 和认证方案名。当使用凭据 provider 的时候，我们可以很模糊的指定主机名、端口号、realm 和认证方案，不用写的很精确。因为，凭据 provider 会根据我们指定的内容，筛选出一个最匹配的方案。

只要我们自定义的凭据 provider 实现了 CredentialsProvider 这个接口，就可以在 HttpClient 中使用。默认的凭据 provider 叫做 BasicCredentialsProvider，它使用 java.util.HashMap 对 CredentialsProvider 进行了简单的实现。

```
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")));
```

上面代码输出：

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

4.4. HTTP 授权和执行上下文

HttpClient 依赖 AuthState 类去跟踪认证过程中的状态的详细信息。在 Http 请求过程中，HttpClient 创建两个 AuthState 实例：一个用于目标服务器认证，一个用于代理服务器认证。如果服务器或者代理服务器需要用户的授权信息，AuthScope、AutoScheme 和认证信息就会被填充到两个 AuthScope 实例中。通过对 AutoState 的检测，我们可以确定请求的授权类型，确定是否有匹配的 AuthScheme，确定凭据 provider 根据指定的授权类型是否成功生成了用户的授权信息。

在 Http 请求执行过程中，HttpClient 会向执行上下文中添加下面的授权对象：

- Lookup 对象，表示使用的认证方案。这个对象的值可以在本地上下文中进行设置，来覆盖默认值。

- CredentialsProvider 对象，表示认证方案 provider，这个对象的值可以在本地上下文中进行设置，来覆盖默认值。
- AuthState 对象，表示目标服务器的认证状态，这个对象的值可以在本地上下文中进行设置，来覆盖默认值。
- AuthState 对象，表示代理服务器的认证状态，这个对象的值可以在本地上下文中进行设置，来覆盖默认值。
- AuthCache 对象，表示认证数据的缓存，这个对象的值可以在本地上下文中进行设置，来覆盖默认值。

我们可以在请求执行前，自定义本地 HttpContext 对象来设置需要的 http 认证上下文;也可以在请求执行后，再检测 HttpContext 的状态，来查看授权是否成功。

```
1.  CloseableHttpClient httpclient = <...>
2.
3.  CredentialsProvider credsProvider = <...>
4.  Lookup<AuthSchemeProvider> authRegistry = <...>
5.  AuthCache authCache = <...>
6.
7.  HttpClientContext context = HttpClientContext.create();
8.  context.setCredentialsProvider(credsProvider);
9.  context.setAuthSchemeRegistry(authRegistry);
10. context.setAuthCache(authCache);
11. HttpGet httpget = new HttpGet("http://somehost/");
12. CloseableHttpResponse response1 = httpclient.execute(httpget, context);
13. <...>
14.
15. AuthState proxyAuthState = context.getProxyAuthState();
16. System.out.println("Proxy auth state: " + proxyAuthState.getState());
17. System.out.println("Proxy auth scheme: " + proxyAuthState.getAuthScheme());
18. System.out.println("Proxy auth credentials: " + proxyAuthState.getCredential
    s());
19. AuthState targetAuthState = context.getTargetAuthState();
20. System.out.println("Target auth state: " + targetAuthState.getState());
21. System.out.println("Target auth scheme: " + targetAuthState.getAuthScheme());
22. System.out.println("Target auth credentials: " + targetAuthState.getCredenti
    als());
```

4.5. 缓存认证数据

从版本 4.1 开始，HttpClient 就会自动缓存验证通过的认证信息。但是为了使用这个缓存的认证信息，我们必须在同一个上下文中执行逻辑相关的请求。一旦超出该上下文的作用范围，缓存的认证信息就会失效。

4.6. 抢先认证

HttpClient 默认不支持抢先认证，因为一旦抢先认证被误用或者错用，会导致一系列的安全问题，比如会把用户的认证信息以明文的方式发送给未授权的第三方服务器。因此，需要用户自己根据自己应用的具体环境来评估抢先认证带来的好处和带来的风险。

即使如此，HttpClient 还是允许我们通过配置来启用抢先认证，方法是提前填充认证信息缓存到上下文中，这样，以这个上下文执行的方法，就会使用抢先认证。

```
CloseableHttpClient httpclient = <...>

HttpHost targetHost = new HttpHost("localhost", 80, "http");
CredentialsProvider credsProvider = new BasicCredentialsProvider();
credsProvider.setCredentials(
        new AuthScope(targetHost.getHostName(), targetHost.getPort()),
        new UsernamePasswordCredentials("username", "password"));

// Create AuthCache instance
AuthCache authCache = new BasicAuthCache();
// Generate BASIC scheme object and add it to the local auth cache
BasicScheme basicAuth = new BasicScheme();
authCache.put(targetHost, basicAuth);

// Add AuthCache to the execution context
HttpClientContext context = HttpClientContext.create();
context.setCredentialsProvider(credsProvider);
context.setAuthCache(authCache);

HttpGet httpget = new HttpGet("/");
for (int i = 0; i < 3; i++) {
    CloseableHttpResponse response = httpclient.execute(
            targetHost, httpget, context);
    try {
        HttpEntity entity = response.getEntity();

    } finally {
        response.close();
    }
}
```

4.7. NTLM 认证

从版本 4.1 开始，HttpClient 就全面支持 NTLMv1、NTLMv2 和 NTLM2 认证。当人我们可以仍旧使用外部的 NTLM 引擎（比如 Samba 开发的 JCIFS 库）作为与 Windows 互操作性程序的一部分。

4.7.1. NTLM 连接持久性

相比 Basic 和 Digest 认证，NTLM 认证要明显需要更多的计算开销，性能影响也比较大。这也可能是微软把 NTLM 协议设计成有状态连接的主要原因之一。也就是说，NTLM 连接一旦建立，用户的身份就会在其整个生命周期和它相关联。NTLM 连接的状态性使得连接持久性更加复杂，The stateful nature of NTLM connections makes connection persistence more complex, as for the obvious reason persistent NTLM connections may not be re-used by users with a different user identity. HttpClient 中标准的连接管理器就可以管理有状态的连接。但是，同一会话中逻辑相关的请求，必须使用相同的执行上下文，这样才能使用用户的身份信息。否则，HttpClient 就会结束旧的连接，为了获取被 NTLM 协议保护的资源，而为每个 HTTP 请求，创建一个新的 Http 连接。更新关于 Http 状态连接的信息，点击此处。

由于 NTLM 连接是有状态的，一般推荐使用比较轻量级的方法来处罚 NTLM 认证（如 GET、Head 方法），然后使用这个已经建立的连接在执行相对重量级的方法，尤其是需要附件请求实体的请求（如 POST、PUT 请求）。

```
1.  CloseableHttpClient httpclient = <...>
2.
3.  CredentialsProvider credsProvider = new BasicCredentialsProvider();
4.  credsProvider.setCredentials(AuthScope.ANY,
5.          new NTCredentials("user", "pwd", "myworkstation", "microsoft.com"));
6.
7.  HttpHost target = new HttpHost("www.microsoft.com", 80, "http");
8.
9.  // Make sure the same context is used to execute logically related requests
10. HttpClientContext context = HttpClientContext.create();
11. context.setCredentialsProvider(credsProvider);
12.
13. // Execute a cheap method first. This will trigger NTLM authentication
14. HttpGet httpget = new HttpGet("/ntlm-protected/info");
15. CloseableHttpResponse response1 = httpclient.execute(target, httpget, contex
    t);
16. try {
17.     HttpEntity entity1 = response1.getEntity();
18. } finally {
19.     response1.close();
20. }
21.
22. // Execute an expensive method next reusing the same context (and connection)
23. HttpPost httppost = new HttpPost("/ntlm-protected/form");
24. httppost.setEntity(new StringEntity("lots and lots of data"));
25. CloseableHttpResponse response2 = httpclient.execute(target, httppost, conte
    xt);
26. try {
```

```
    HttpEntity entity2 = response2.getEntity();
} finally {
    response2.close();
}
```

4.8. SPNEGO/Kerberos 认证

SPNEGO(Simple and Protected GSSAPI Negotiation Mechanism），当双方均不知道对方能使用/提供什么协议的情况下，可以使用 SP 认证协议。这种协议在 Kerberos 认证方案中经常使用。It can wrap other mechanisms, however the current version in HttpClient is designed solely with Kerberos in mind.

4.8.1. 在 HTTPCIENT 中使用 SPNEGO

SPNEGO 认证方案兼容 Sun java 1.5 及以上版本。但是强烈推荐 jdk1.6 以上。Sun 的 JRE 提供的类就已经几乎完全可以处理 Kerberos 和 SPNEGO token。这就意味着，需要设置很多的 GSS 类。SpnegoScheme 是个很简单的类，可以用它来 handle marshalling the tokens and 读写正确的头消息。

最好的开始方法就是从示例程序中找到 KerberosHttpClient.java 这个文件，尝试让它运行起来。运行过程有可能会出现很多问题，但是如果人品比较高可能会顺利一点。这个文件会提供一些输出，来帮我们调试。

在 Windows 系统中，应该默认使用用户的登陆凭据;当然我们也可以使用 kinit 来覆盖这个凭据，比如$JAVA_HOME\bin\kinit testuser@AD.EXAMPLE.NET，这在我们测试和调试的时候就显得很有用了。如果想用回 Windows 默认的登陆凭据，删除 kinit 创建的缓存文件即可。

确保在 krb5.conf 文件中列出 domain_realms。这能解决很多不必要的问题。

4.8.2. 使用 GSS/JAVA KERBEROS

下面的这份文档是针对 Windows 系统的，但是很多信息同样适合 Unix。

org.ietf.jgss 这个类有很多的配置参数，这些参数大部分都在 krb5.conf/krb5.ini 文件中配置。更多的信息，参考此处。

login.conf 文件

下面是一个基本的 login.conf 文件，使用于 Windows 平台的 IIS 和 JBoss Negotiation 模块。

系统配置文件 java.security.auth.login.config 可以指定 login.conf 文件的路径。
login.conf 的内容可能会是下面的样子：

```
com.sun.security.jgss.login {
  com.sun.security.auth.module.Krb5LoginModule required client=TRUE useTicketCache=true;
};

com.sun.security.jgss.initiate {
  com.sun.security.auth.module.Krb5LoginModule required client=TRUE useTicketCache=true;
};

com.sun.security.jgss.accept {
  com.sun.security.auth.module.Krb5LoginModule required client=TRUE useTicketCache=true;
};
```

4.8.4. KRB5.CONF / KRB5.INI 文件

如果没有手动指定，系统会使用默认配置。如果要手动指定，可以在 java.security.krb5.conf 中设置系统变量，指定 krb5.conf 的路径。krb5.conf 的内容可能是下面的样子：

```
[libdefaults]
    default_realm = AD.EXAMPLE.NET
    udp_preference_limit = 1
[realms]
    AD.EXAMPLE.NET = {
        kdc = KDC.AD.EXAMPLE.NET
    }
[domain_realms]
.ad.example.net=AD.EXAMPLE.NET
ad.example.net=AD.EXAMPLE.NET
```

4.8.5. WINDOWS 详细的配置

为了允许 Windows 使用当前用户的 tickets，javax.security.auth.useSubjectCredsOnly 这个系统变量应该设置成 false，并且需要在 Windows 注册表中添加 allowtgtsessionkey 这个项，而且要 allow session keys to be sent in the Kerberos Ticket-Granting Ticket.

Windows Server 2003 和 Windows 2000 SP4,配置如下：

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\Kerberos\Parameters

Value Name: allowtgtsessionkey
Value Type: REG_DWORD
Value: 0x01
```

Windows XP SP2 配置如下：

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\Kerberos\
Value Name: allowtgtsessionkey
Value Type: REG_DWORD
Value: 0x01
```

第五章 快速 API

5.1. Easy to use facade API

HttpClient 从 4.2 开始支持快速 api。快速 api 仅仅实现了 HttpClient 的基本功能，它只要用于一些不需要灵活性的简单场景。例如，快速 api 不需要用户处理连接管理和资源释放。

下面是几个使用快速 api 的例子：

```
// Execute a GET with timeout settings and return response content as String.
Request.Get("http://somehost/")
        .connectTimeout(1000)
        .socketTimeout(1000)
        .execute().returnContent().asString();
```

```
// Execute a POST with the 'expect-continue' handshake, using HTTP/1.1,
// containing a request body as String and return response content as byte array.
Request.Post("http://somehost/do-stuff")
        .useExpectContinue()
        .version(HttpVersion.HTTP_1_1)
        .bodyString("Important stuff", ContentType.DEFAULT_TEXT)
        .execute().returnContent().asBytes();
```

```
// Execute a POST with a custom header through the proxy containing a request body
// as an HTML form and save the result to the file
Request.Post("http://somehost/some-form")
        .addHeader("X-Custom-header", "stuff")
        .viaProxy(new HttpHost("myproxy", 8080))
        .bodyForm(Form.form().add("username", "vip").add("password", "secret").build())
```

```
7.          .execute().saveContent(new File("result.dump"));
```

如果需要在指定的安全上下文中执行某些请求，我们也可以直接使用 Exector，这时候用户的认证信息就会被缓存起来，以便后续的请求使用。

```
1.  Executor executor = Executor.newInstance()
2.          .auth(new HttpHost("somehost"), "username", "password")
3.          .auth(new HttpHost("myproxy", 8080), "username", "password")
4.          .authPreemptive(new HttpHost("myproxy", 8080));
5.
6.  executor.execute(Request.Get("http://somehost/"))
7.          .returnContent().asString();
8.
9.  executor.execute(Request.Post("http://somehost/do-stuff")
10.         .useExpectContinue()
11.         .bodyString("Important stuff", ContentType.DEFAULT_TEXT))
12.         .returnContent().asString();
```

5.1.1. 响应处理

一般情况下，HttpClient 的快速 api 不用用户处理连接管理和资源释放。但是，这样的话，就必须在内存中缓存这些响应消息。为了避免这一情况，建议使用使用 ResponseHandler 来处理 Http 响应。

```
1.  Document result = Request.Get("http://somehost/content")
2.          .execute().handleResponse(new ResponseHandler<Document>() {
3.
4.      public Document handleResponse(final HttpResponse response) throws IOExc
    eption {
5.          StatusLine statusLine = response.getStatusLine();
6.          HttpEntity entity = response.getEntity();
7.          if (statusLine.getStatusCode() >= 300) {
8.              throw new HttpResponseException(
9.                      statusLine.getStatusCode(),
10.                     statusLine.getReasonPhrase());
11.         }
12.         if (entity == null) {
13.             throw new ClientProtocolException("Response contains no content")
    ;
14.         }
15.         DocumentBuilderFactory dbfac = DocumentBuilderFactory.newInstance();

16.         try {
17.             DocumentBuilder docBuilder = dbfac.newDocumentBuilder();
18.             ContentType contentType = ContentType.getOrDefault(entity);
19.             if (!contentType.equals(ContentType.APPLICATION_XML)) {
20.                 throw new ClientProtocolException("Unexpected content type:"
    +
```

```
                    contentType);
            }
            String charset = contentType.getCharset();
            if (charset == null) {
                charset = HTTP.DEFAULT_CONTENT_CHARSET;
            }
            return docBuilder.parse(entity.getContent(), charset);
        } catch (ParserConfigurationException ex) {
            throw new IllegalStateException(ex);
        } catch (SAXException ex) {
            throw new ClientProtocolException("Malformed XML document", ex);
        }
    }

});
```

第六章 HTTP 缓存

6.1. 基本概念

HttpClient 的缓存机制提供一个与 HTTP/1.1 标准兼容的缓存层 – 相当于 Java 的浏览器缓存。HttpClient 缓存机制的实现遵循责任链（Chain of Responsibility）设计原则，默认的 HttpClient 是没有缓存的，有缓存机制的 HttpClient 可以用来临时替代默认的 HttpClient，如果开启了缓存，我们的请求结果就会从缓存中获取，而不是从目标服务器中获取。如果在 Get 请求头中设置了 If-Modified-Since 或者 If-None-Match 参数，那么 HttpClient 会自动向服务器校验缓存是否过期。

HTTP/1.1 版本的缓存是语义透明的，意思是无论怎样，缓存都不应该修改客户端与服务器之间传输的请求/响应数据包。因此，在 existing compliant client-server relationship 中使用带有缓存的 HttpClient 也应该是安全的。虽然缓存是客户端的一部分，但是从 Http 协议的角度来看，缓存机制是为了兼容透明的缓存代理。

最后，HttpClient 缓存也支持 RFC 5861 规定的 Cache-Control 拓展（stale-if-error'和 stale-while-revalidate`）。

当开启缓存的 HttpClient 执行一个 Http 请求时，会经过下面的步骤：

- 检查 http 请求是否符合 HTTP 1.1 的基本要求，如果不符合就尝试修正错误。
- 刷新该请求无效的缓存项。（Flush any cache entries which would be invalidated by this request.）
- 检测该请求是否可以从缓存中获取。如果不能，直接将请求发送给目标服务器，获取响应并加入缓存。

- 如果该请求可以从缓存中获取，HttpClient 就尝试读取缓存中的数据。如果读取失败，就会发送请求到目标服务器，如果可能的话，就把响应缓存起来。
- 如果 HttpClient 缓存的响应可以直接返回给请求，HttpClient 就会构建一个包含 ByteArrayEntity 的 BasicHttpResponse 对象，并将它返回给 http 请求。否则，HttpClient 会向服务器重新校验缓存。
- 如果 HttpClient 缓存的响应，向服务器校验失败，就会向服务器重新请求数据，并将其缓存起来（如果合适的话）。

  当开启缓存的 HttpClient 收到服务器的响应时，会经过下面的步骤：
- 检查收到的响应是否符合协议兼容性
- 确定收到的响应是否可以缓存
- 如果响应是可以缓存的，HttpClient 就会尽量从响应消息中读取数据（大小可以在配置文件进行配置），并且缓存起来。
- 如果响应数据太大，缓存或者重构消耗的响应空间不够，就会直接返回响应，不进行缓存。

  需要注意的是，带有缓存的 HttpClient 不是 HttpClient 的另一种实现，而是通过向 http 请求执行管道中插入附加处理组件来实现的。

6.2. RFC-2616 Compliance

HttpClient 的缓存机制和 RFC-2626 文档规定是无条件兼容的。也就是说，只要指定了 MUST，MUST NOT，SHOULD 或者 SHOULD NOT 这些 Http 缓存规范，HttpClient 的缓存层就会按照指定的方式进行缓存。即当我们使用 HttpClient 的缓存机制时，HttpClient 的缓存模块不会产生异常动作。

6.3. 使用范例

下面的例子讲述了如何创建一个基本的开启缓存的 HttpClient。并且配置了最大缓存 1000 个 Object 对象，每个对象最大占用 8192 字节数据。代码中出现的数据，只是为了做演示，而过不是推荐使用的配置。

```
1.  CacheConfig cacheConfig = CacheConfig.custom()
2.          .setMaxCacheEntries(1000)
3.          .setMaxObjectSize(8192)
4.          .build();
5.  RequestConfig requestConfig = RequestConfig.custom()
6.          .setConnectTimeout(30000)
7.          .setSocketTimeout(30000)
8.          .build();
9.  CloseableHttpClient cachingClient = CachingHttpClients.custom()
10.         .setCacheConfig(cacheConfig)
11.         .setDefaultRequestConfig(requestConfig)
12.         .build();
```

```
13.
14. HttpCacheContext context = HttpCacheContext.create();
15. HttpGet httpget = new HttpGet("http://www.mydomain.com/content/");
16. CloseableHttpResponse response = cachingClient.execute(httpget, context);
17. try {
18.     CacheResponseStatus responseStatus = context.getCacheResponseStatus();
19.     switch (responseStatus) {
20.         case CACHE_HIT:
21.             System.out.println("A response was generated from the cache with
    " +
22.                     "no requests sent upstream");
23.             break;
24.         case CACHE_MODULE_RESPONSE:
25.             System.out.println("The response was generated directly by the "
    +
26.                     "caching module");
27.             break;
28.         case CACHE_MISS:
29.             System.out.println("The response came from an upstream server");

30.             break;
31.         case VALIDATED:
32.             System.out.println("The response was generated from the cache "
    +
33.                     "after validating the entry with the origin server");
34.             break;
35.     }
36. } finally {
37.     response.close();
38. }
```

6.4. 配置

有缓存的 HttpClient 继承了非缓存 HttpClient 的所有配置项和参数（包括超时时间，连接池大小等配置项）。如果需要对缓存进行具体配置，可以初始化一个 CacheConfig 对象来自定义下面的参数：

- Cache size（缓存大小）. 如果后台存储支持，我们可以指定缓存的最大条数，和每个缓存中存储的 response 的最大 size。
- Public/private cacheing(公用/私有 缓存）. 默认情况下，缓存模块会把缓存当做公用的缓存，所以缓存机制不会缓存带有授权头消息或者指定 Cache-Control:private 的响应。但是如果缓存只会被一个逻辑上的用户使用（和浏览器饿缓存类似），我们可能希望关闭缓存共享机制。
- Heuristic caching（启发式缓存）。即使服务器没有明确设置缓存控制 headers 信息，每个 RFC2616 缓存也会存储一定数目的缓存。这个特征在 HttpClient 中默认是关闭的，如果服务器不设置控制缓存的 header 信息，但是我们仍然希望对响应进行缓存，就需要在 HttpClient

中打开这个功能。激活启发式缓存，然后使用默认的刷新时间或者自定义刷新时间。更多启发式缓存的信息，可以参考 Http/1.1 RFC 文档的 13.2.2 小节，13.2.4 小节。

- Background validation(后台校验）。HttpClient 的缓存机制支持 RFC5861 的 stale-while-revalidate 指令，它允许一定数目的缓存在后台校验是否过期。我们可能需要调整可以在后台工作的最大和最小的线程数，以及设置线程在回收前最大的空闲时间。当没有足够线程来校验缓存是否过期时，我们可以指定排队队列的大小。

6.5.存储介质

默认，HttpClient 缓存机制将缓存条目和缓存的 response 放在本地程序的 jvm 内存中。这样虽然提供高性能，但是当我们的程序内存有大小限制的时候，这就会变得不太合理。因为缓存的生命中期很短，如果程序重启，缓存就会失效。当前版本的 HttpClient 使用 EhCache 和 memchached 来存储缓存，这样就支持将缓存放到本地磁盘或者其他存储介质上。如果内存、本地磁盘、外地磁盘，都不适合你的应用程序，HttpClient 也支持自定义存储介质，只需要实现 HttpCacheStorage 接口，然后在创建 HttpClient 时，使用这个接口的配置。这种情况，缓存会存储在自定义的介质中，但是 you will get to reuse all of the logic surrounding HTTP/1.1 compliance and cache handling. 一般来说，可以创建出支持任何键值对指定存储（类似 Java Map 接口）的 HttpCacheStorage，用于进行原子更新。

最后，通过一些额外的工作，还可以建立起多层次的缓存结构;磁盘中的缓存，远程 memcached 中的缓存，虚拟内存中的缓存，L1/L2 处理器中的缓存等。

第七章　高级主题

7.1 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。

```
<span style="font-family:SimSun;">class MyLineParser extends BasicLineParser {

    @Override
    public Header parseHeader(
            CharArrayBuffer buffer) throws ParseException {
        try {
            return super.parseHeader(buffer);
        } catch (ParseException ex) {
            // Suppress ParseException exception
            return new BasicHeader(buffer.toString(), null);
        }
    }

}</span>
```

提过一个自定义的 HttpConnectionFactory 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。

```
<span style="font-family:SimSun;">HttpConnectionFactory<HttpRoute, ManagedHttpClientConnection> connFactory =
        new ManagedHttpClientConnectionFactory(
            new DefaultHttpRequestWriterFactory(),
            new DefaultHttpResponseParserFactory(
                    new MyLineParser(), new DefaultHttpResponseFactory()));</span>
```

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。

```
<span style="font-family:SimSun;">PoolingHttpClientConnectionManager cm = new PoolingHttpClientConnectionManager(
    connFactory);
CloseableHttpClient httpclient = HttpClients.custom()
        .setConnectionManager(cm)
        .build();</span>
```

7.2 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。

7.2.1 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：

```
1.  <span style="font-
    family:SimSun;">CloseableHttpClient httpclient = HttpClients.createDefault();
2.  HttpClientContext context = HttpClientContext.create();
3.  HttpGet httpget = new HttpGet("http://localhost:8080/");
4.  CloseableHttpResponse response = httpclient.execute(httpget, context);
5.  try {
6.      Principal principal = context.getUserToken(Principal.class);
7.      System.out.println(principal);
8.  } finally {
9.      response.close();
10. }</span>
```

如果默认的不能满足需求，用户可以提供一个特定的实现：

```
1.  <span style="font-
    family:SimSun;">UserTokenHandler userTokenHandler = new UserTokenHandler() {
2.
3.      public Object getUserToken(HttpContext context) {
4.          return context.getAttribute("my-token");
5.      }
6.
7.  };
8.  CloseableHttpClient httpclient = HttpClients.custom()
9.          .setUserTokenHandler(userTokenHandler)
10.         .build();</span>
```

7.2.2 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。

```
<span style="font-family:SimSun;">CloseableHttpClient httpclient = HttpClients.createDefault();

HttpClientContext context1 = HttpClientContext.create();
HttpGet httpget1 = new HttpGet("http://localhost:8080/");
CloseableHttpResponse response1 = httpclient.execute(httpget1, context1);
try {
    HttpEntity entity1 = response1.getEntity();
} finally {
    response1.close();
}
Principal principal = context1.getUserToken(Principal.class);

HttpClientContext context2 = HttpClientContext.create();
context2.setUserToken(principal);
HttpGet httpget2 = new HttpGet("http://localhost:8080/");
CloseableHttpResponse response2 = httpclient.execute(httpget2, context2);
try {
    HttpEntity entity2 = response2.getEntity();
} finally {
    response2.close();
}</span>
```

7.3. 使用 FutureRequestExecutionService

通过使用 FutureRequestExecutionService，你可以调度 HTTP 调用以及把 response 当作一个 Future。这是非常有用的，比如当多次调用一个 Web 服务。使用 FutureRequestExecutionService 的优势在于您可以使用多个线程来调度请求同时，对任务设置超时或取消，当 response 不再需要的时候。

FutureRequestExecutionService 用 HttpRequestFutureTask（继承 FutureTask）包装 request。这个类允许你取消 Task 以及保持跟踪各项指标，如 request duration。

7.3.1. 构造 FutureRequestExecutionService

futureRequestExecutionService 的构造方法包括两个参数：httpClient 实例和 ExecutorService 实例。当配置两个参数的时候，您要使用的线程数等于最大连接数是很重要的。当线程比连接多的时候，连接可能会开始超时，因为没有可用的连接。当连接多于线程时，futureRequestExecutionService 不会使用所有的连接。

```
<span style="font-family:SimSun;">HttpClient httpClient = HttpClientBuilder.create().setMaxConnPerRoute(5).build();
ExecutorService executorService = Executors.newFixedThreadPool(5);
FutureRequestExecutionService futureRequestExecutionService =
    new FutureRequestExecutionService(httpClient, executorService);</span>
```

7.3.2. 安排 requests

要安排一个请求，只需提供一个 HttpUriRequest，HttpContext 和 ResponseHandler。因为 request 是由 executor service 处理的，而 ResponseHandler 的是强制性的。

```
<span style="font-family:SimSun;">private final class OkidokiHandler implements ResponseHandler<Boolean> {
    public Boolean handleResponse(
            final HttpResponse response) throws ClientProtocolException, IOException {
        return response.getStatusLine().getStatusCode() == 200;
    }
}

HttpRequestFutureTask<Boolean> task = futureRequestExecutionService.execute(
    new HttpGet("http://www.google.com"), HttpClientContext.create(),
    new OkidokiHandler());
// blocks until the request complete and then returns true if you can connect to Google
boolean ok=task.get();</span>
```

7.3.3. 取消 tasks

预定的任务可能会被取消。如果任务尚未执行，但仅仅是排队等待执行，它根本就不会执行。如果任务在执行中且 mayInterruptIfRunning 参数被设置为 true，请求中的 abort（）函数将被调用;否则 response 会简单地忽略，但该请求将被允许正常完成。任何后续调用 task.get()会产生一个 IllegalStateException。应当注意到，取消任务仅可以释放客户端的资源。该请求可能实际上是在服务器端正常处理。

```
<span style="font-family:SimSun;">task.cancel(true)
task.get() // throws an Exception</span>
```

7.3.4. 回调

不用手动调用 task.get()，您也可以在请求完成时使用 FutureCallback 实例获取回调。这里采用的是和 HttpAsyncClient 相同的接口

```
<span style="font-family:SimSun;">private final class MyCallback implements FutureCallback<Boolean> {

    public void failed(final Exception ex) {
        // do something
    }

    public void completed(final Boolean result) {
        // do something
    }

    public void cancelled() {
        // do something
    }
}

HttpRequestFutureTask<Boolean> task = futureRequestExecutionService.execute(

    new HttpGet("http://www.google.com"), HttpClientContext.create(),
    new OkidokiHandler(), new MyCallback());</span>
```

7.3.5. 指标

FutureRequestExecutionService 通常用于大量 Web 服务调用的应用程序之中。为了便于例如监视或配置调整，FutureRequestExecutionService 跟踪了几个指标。

HttpRequestFutureTask 会提供一些方法来获得任务时间：从被安排，开始，直到结束。此外，请求和任务持续时间也是可用的。这些指标都聚集在 FutureRequestExecutionService 中的 FutureRequestExecutionMetrics 实例，可以通过 FutureRequestExecutionService.metrics()获取。

```
<span style="font-family:SimSun;">task.scheduledTime() // returns the timestamp the task was scheduled
task.startedTime() // returns the timestamp when the task was started
task.endedTime() // returns the timestamp when the task was done executing
task.requestDuration // returns the duration of the http request
task.taskDuration // returns the duration of the task from the moment it was scheduled

FutureRequestExecutionMetrics metrics = futureRequestExecutionService.metrics()
```

```
metrics.getActiveConnectionCount() // currently active connections
metrics.getScheduledConnectionCount(); // currently scheduled connections
metrics.getSuccessfulConnectionCount(); // total number of successful reques
ts
metrics.getSuccessfulConnectionAverageDuration(); // average request duratio
n
metrics.getFailedConnectionCount(); // total number of failed tasks
metrics.getFailedConnectionAverageDuration(); // average duration of failed
tasks
metrics.getTaskCount(); // total number of tasks scheduled
metrics.getRequestCount(); // total number of requests
metrics.getRequestAverageDuration(); // average request duration
metrics.getTaskAverageDuration(); // average task duration</span>
```

参考:http://hc.apache.org/httpcomponents-client-ga/tutorial/html/index.html