# 1特性：

1. 基于标准、纯净的**[Java](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "http://blog.csdn.net/wangpeng047/article/details/_blank)**语言。实现了Http1.0和Http1.1

2. 以可扩展的面向对象的结构实现了Http全部的方法（GET, POST, PUT, DELETE, HEAD, OPTIONS, and TRACE）。

3. 支持HTTPS协议。

4. 通过Http代理建立透明的连接。

5. 利用CONNECT方法通过Http代理建立隧道的https连接。

6. Basic, Digest, NTLMv1, NTLMv2, NTLM2 Session, SNPNEGO/Kerberos认证方案。

7. 插件式的自定义认证方案。

8. 便携可靠的套接字工厂使它更容易的使用第三方解决方案。

9. 连接管理器支持多线程应用。支持设置最大连接数，同时支持设置每个主机的最大连接数，发现并关闭过期的连接。

10. 自动处理Set-Cookie中的Cookie。

11. 插件式的自定义Cookie策略。

12. Request的输出流可以避免流中内容直接缓冲到socket服务器。

13. Response的输入流可以有效的从socket服务器直接读取相应内容。

14. 在http1.0和http1.1中利用KeepAlive保持持久连接。

15. 直接获取服务器发送的response code和 headers。

16. 设置连接超时的能力。

17. 实验性的支持http1.1 response caching。

18. 源代码基于Apache License 可免费获取。

# 2 使用方法：

使用HttpClient发送请求、接收响应很简单，一般需要如下几步即可。

1. 创建HttpClient对象。

2. 创建请求方法的实例，并指定请求URL。如果需要发送GET请求，创建HttpGet对象；如果需要发送POST请求，创建HttpPost对象。

3. 如果需要发送请求参数，可调用HttpGet、HttpPost共同的setParams(HetpParams params)方法来添加请求参数；对于HttpPost对象而言，也可调用setEntity(HttpEntity entity)方法来设置请求参数。

4. 调用HttpClient对象的execute(HttpUriRequest request)发送请求，该方法返回一个HttpResponse。

5. 调用HttpResponse的getAllHeaders()、getHeaders(String name)等方法可获取服务器的响应头；调用HttpResponse的getEntity()方法可获取HttpEntity对象，该对象包装了服务器的响应内容。程序可通过该对象获取服务器的响应内容。

6. 释放连接。无论执行方法是否成功，都必须释放连接

# 3 Httpclient4.3实现：

## 3.1 新增简单的url请求内容返回, 比较时髦的链调用

1. **try** {
2. Content returnContent = Request.Get("http://www.qq.com")
3. .execute().returnContent();
4. System.out.println(returnContent.toString());
5. } **catch** (ClientProtocolException e) {
6. // TODO Auto-generated catch block
7. e.printStackTrace();
8. } **catch** (IOException e) {
9. // TODO Auto-generated catch block
10. e.printStackTrace();
11. }

## 3.2 一套demo方法， 含模拟登录，获取cookie，并把cookie带入后面的请求中

1. CookieStore cookieStore = **new** BasicCookieStore();
3. HttpClientContext context = HttpClientContext.create();
4. context.setCookieStore(cookieStore);
6. RequestConfig globalConfig = RequestConfig.custom().setCookieSpec(CookieSpecs.BROWSER\_COMPATIBILITY).build();
8. CloseableHttpClient httpclient = HttpClients.custom().
9. setDefaultRequestConfig(globalConfig)
10. .setDefaultCookieStore(cookieStore).build();
11. // 登录
12. HttpPost loginPost = **new** HttpPost("https://www.xxx.com/account/login.php");
13. List<NameValuePair> nvps = **new** ArrayList<NameValuePair>();
14. nvps.add(**new** BasicNameValuePair("email", email));
15. nvps.add(**new** BasicNameValuePair("password", pwd));
16. loginPost.setEntity(**new** UrlEncodedFormEntity(nvps,"UTF-8"));
17. CloseableHttpResponse  loginResp = httpclient.execute(loginPost,context);
18. **try** {
19. System.out.println(loginResp.getStatusLine());
20. HttpEntity entity1 = loginResp.getEntity();
21. EntityUtils.consume(entity1);
22. } **finally** {
23. loginResp.close();
24. }
26. **for** (**int** i = START\_PAGE; i < END\_PAGE; i++) {
27. Thread.sleep(SLEEP\_TIME);
28. //获取交易
29. HttpGet get = **new** HttpGet(
30. "https://www.xxx.com/trade/index.php?a=history&t=0&amt\_begin=0&amt\_end=0&date\_begin=&date\_end=&pn="+i);
31. HttpClientContext context1 = HttpClientContext.create();
32. context1.setCookieStore(cookieStore);
33. HttpResponse dealResp = httpclient.execute(get, context1);
34. **try** {
35. HttpEntity dealEntity = dealResp.getEntity();
36. String body = EntityUtils.toString(dealEntity);
37. prasepage(body);
38. EntityUtils.consume(dealEntity);
39. } **finally** {
40. loginResp.close();
41. }
42. }

## 3.3 HttpGet 参数拼装代码示例：

1. String base = "http://api.map.baidu.com/geosearch/v3/nearby?";
3. List<NameValuePair> params = **new** LinkedList<NameValuePair>();
4. params.add(**new** BasicNameValuePair("ak", JobConstans.ak));
5. params.add(**new** BasicNameValuePair("geotable\_id", geotable\_id));
6. params.add(**new** BasicNameValuePair("filter", filter.filter));
7. params.add(**new** BasicNameValuePair("page\_index", String.valueOf(filter.page\_index)));
8. params.add(**new** BasicNameValuePair("page\_size", String.valueOf(filter.page\_size)));
9. params.add(**new** BasicNameValuePair("location", filter.location));
10. params.add(**new** BasicNameValuePair("sortby", filter.sortby));
11. params.add(**new** BasicNameValuePair("radius", String.valueOf(filter.radius)));
12. String paramString = URLEncodedUtils.format(params, "utf-8");
13. base += paramString;

## 3.4 httpclient 连接 ssl

1. **public** **void** ssl() {
2. CloseableHttpClient httpclient = **null**;
3. **try** {
4. KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
5. FileInputStream instream = **new** FileInputStream(**new** File("d:\\tomcat.keystore"));
6. **try** {
7. // 加载keyStore d:\\tomcat.keystore
8. trustStore.load(instream, "123456".toCharArray());
9. } **catch** (CertificateException e) {
10. e.printStackTrace();
11. } **finally** {
12. **try** {
13. instream.close();
14. } **catch** (Exception ignore) {
15. }
16. }
17. // 相信自己的CA和所有自签名的证书
18. SSLContext sslcontext = SSLContexts.custom().loadTrustMaterial(trustStore, **new** TrustSelfSignedStrategy()).build();
19. // 只允许使用TLSv1协议
20. SSLConnectionSocketFactory sslsf = **new** SSLConnectionSocketFactory(sslcontext, **new** String[] { "TLSv1" }, **null**,
21. SSLConnectionSocketFactory.BROWSER\_COMPATIBLE\_HOSTNAME\_VERIFIER);
22. httpclient = HttpClients.custom().setSSLSocketFactory(sslsf).build();
23. // 创建http请求(get方式)
24. HttpGet httpget = **new** HttpGet("https://localhost:8443/myDemo/Ajax/serivceJ.action");
25. System.out.println("executing request" + httpget.getRequestLine());
26. CloseableHttpResponse response = httpclient.execute(httpget);
27. **try** {
28. HttpEntity entity = response.getEntity();
29. System.out.println("----------------------------------------");
30. System.out.println(response.getStatusLine());
31. **if** (entity != **null**) {
32. System.out.println("Response content length: " + entity.getContentLength());
33. System.out.println(EntityUtils.toString(entity));
34. EntityUtils.consume(entity);
35. }
36. } **finally** {
37. response.close();
38. }
39. } **catch** (ParseException e) {
40. e.printStackTrace();
41. } **catch** (IOException e) {
42. e.printStackTrace();
43. } **catch** (KeyManagementException e) {
44. e.printStackTrace();
45. } **catch** (NoSuchAlgorithmException e) {
46. e.printStackTrace();
47. } **catch** (KeyStoreException e) {
48. e.printStackTrace();
49. } **finally** {
50. **if** (httpclient != **null**) {
51. **try** {
52. httpclient.close();
53. } **catch** (IOException e) {
54. e.printStackTrace();
55. }
56. }
57. }

## 3.5 post 方式提交表单

1. **public** **void** postForm() {
2. // 创建默认的httpClient实例.
3. CloseableHttpClient httpclient = HttpClients.createDefault();
4. // 创建httppost
5. HttpPost httppost = **new** HttpPost("http://localhost:8080/myDemo/Ajax/serivceJ.action");
6. // 创建参数队列
7. List<namevaluepair> formparams = **new** ArrayList<namevaluepair>();
8. formparams.add(**new** BasicNameValuePair("username", "admin"));
9. formparams.add(**new** BasicNameValuePair("password", "123456"));
10. UrlEncodedFormEntity uefEntity;
11. **try** {
12. uefEntity = **new** UrlEncodedFormEntity(formparams, "UTF-8");
13. httppost.setEntity(uefEntity);
14. System.out.println("executing request " + httppost.getURI());
15. CloseableHttpResponse response = httpclient.execute(httppost);
16. **try** {
17. HttpEntity entity = response.getEntity();
18. **if** (entity != **null**) {
19. System.out.println("--------------------------------------");
20. System.out.println("Response content: " + EntityUtils.toString(entity, "UTF-8"));
21. System.out.println("--------------------------------------");
22. }
23. } **finally** {
24. response.close();
25. }
26. } **catch** (ClientProtocolException e) {
27. e.printStackTrace();
28. } **catch** (UnsupportedEncodingException e1) {
29. e1.printStackTrace();
30. } **catch** (IOException e) {
31. e.printStackTrace();
32. } **finally** {
33. // 关闭连接,释放资源
34. **try** {
35. httpclient.close();
36. } **catch** (IOException e) {
37. e.printStackTrace();
38. }
39. }
40. }

## 3.6 发送post请求访问本地应用，并根据参数不同返回不同的结果

1. **public** **void** post() {
2. // 创建默认的httpClient实例.
3. CloseableHttpClient httpclient = HttpClients.createDefault();
4. // 创建httppost
5. HttpPost httppost = **new** HttpPost("http://localhost:8080/myDemo/Ajax/serivceJ.action");
6. // 创建参数队列
7. List<namevaluepair> formparams = **new** ArrayList<namevaluepair>();
8. formparams.add(**new** BasicNameValuePair("type", "house"));
9. UrlEncodedFormEntity uefEntity;
10. **try** {
11. uefEntity = **new** UrlEncodedFormEntity(formparams, "UTF-8");
12. httppost.setEntity(uefEntity);
13. System.out.println("executing request " + httppost.getURI());
14. CloseableHttpResponse response = httpclient.execute(httppost);
15. **try** {
16. HttpEntity entity = response.getEntity();
17. **if** (entity != **null**) {
18. System.out.println("--------------------------------------");
19. System.out.println("Response content: " + EntityUtils.toString(entity, "UTF-8"));
20. System.out.println("--------------------------------------");
21. }
22. } **finally** {
23. response.close();
24. }
25. } **catch** (ClientProtocolException e) {
26. e.printStackTrace();
27. } **catch** (UnsupportedEncodingException e1) {
28. e1.printStackTrace();
29. } **catch** (IOException e) {
30. e.printStackTrace();
31. } **finally** {
32. // 关闭连接,释放资源
33. **try** {
34. httpclient.close();
35. } **catch** (IOException e) {
36. e.printStackTrace();
37. }
38. }
39. }

## 3.7 发送get请求

1. **public** **void** get() {
2. CloseableHttpClient httpclient = HttpClients.createDefault();
3. **try** {
4. // 创建httpget.
5. HttpGet httpget = **new** HttpGet("http://www.baidu.com/");
6. System.out.println("executing request " + httpget.getURI());
7. // 执行get请求.
8. CloseableHttpResponse response = httpclient.execute(httpget);
9. **try** {
10. // 获取响应实体
11. HttpEntity entity = response.getEntity();
12. System.out.println("--------------------------------------");
13. // 打印响应状态
14. System.out.println(response.getStatusLine());
15. **if** (entity != **null**) {
16. // 打印响应内容长度
17. System.out.println("Response content length: " + entity.getContentLength());
18. // 打印响应内容
19. System.out.println("Response content: " + EntityUtils.toString(entity));
20. }
21. System.out.println("------------------------------------");
22. } **finally** {
23. response.close();
24. }
25. } **catch** (ClientProtocolException e) {
26. e.printStackTrace();
27. } **catch** (ParseException e) {
28. e.printStackTrace();
29. } **catch** (IOException e) {
30. e.printStackTrace();
31. } **finally** {
32. // 关闭连接,释放资源
33. **try** {
34. httpclient.close();
35. } **catch** (IOException e) {
36. e.printStackTrace();
37. }
38. }
39. }

## 3.8 上传文件

1. **public** **void** upload() {
2. CloseableHttpClient httpclient = HttpClients.createDefault();
3. **try** {
4. HttpPost httppost = **new** HttpPost("http://localhost:8080/myDemo/Ajax/serivceFile.action");
6. FileBody bin = **new** FileBody(**new** File("F:\\image\\sendpix0.jpg"));
7. StringBody comment = **new** StringBody("A binary file of some kind", ContentType.TEXT\_PLAIN);
9. HttpEntity reqEntity = MultipartEntityBuilder.create().addPart("bin", bin).addPart("comment", comment).build();
11. httppost.setEntity(reqEntity);
13. System.out.println("executing request " + httppost.getRequestLine());
14. CloseableHttpResponse response = httpclient.execute(httppost);
15. **try** {
16. System.out.println("----------------------------------------");
17. System.out.println(response.getStatusLine());
18. HttpEntity resEntity = response.getEntity();
19. **if** (resEntity != **null**) {
20. System.out.println("Response content length: " + resEntity.getContentLength());
21. }
22. EntityUtils.consume(resEntity);
23. } **finally** {
24. response.close();
25. }
26. } **catch** (ClientProtocolException e) {
27. e.printStackTrace();
28. } **catch** (IOException e) {
29. e.printStackTrace();
30. } **finally** {
31. **try** {
32. httpclient.close();
33. } **catch** (IOException e) {
34. e.printStackTrace();
35. }
36. }
37. }
38. }</namevaluepair></namevaluepair></namevaluepair></namevaluepair>

# 4 Httpclient 4.5 实现：

## 4.1 cttpclient工具类

### 4.1.1初始化设置

1. **private** **static** PoolingHttpClientConnectionManager cm;
2. **private** **static** String EMPTY\_STR = "";
3. **private** **static** String UTF\_8 = "UTF-8";
5. **private** **static** **void** init() {
6. **if** (cm == **null**) {
7. cm = **new** PoolingHttpClientConnectionManager();
8. cm.setMaxTotal(50);// 整个连接池最大连接数
9. cm.setDefaultMaxPerRoute(5);// 每路由最大连接数，默认值是2
10. }
11. }

### 4.1.2 通过连接池获取httpclient

1. **private** **static** CloseableHttpClient getHttpClient() {
2. init();
3. **return** HttpClients.custom().setConnectionManager(cm).build();
4. }
5. **public** **static** String httpGetRequest(String url) {
6. HttpGet httpGet = **new** HttpGet(url);
7. **return** getResult(httpGet);
8. }
10. **public** **static** String httpGetRequest(String url, Map<String, Object> params) **throws** URISyntaxException {
11. URIBuilder ub = **new** URIBuilder();
12. ub.setPath(url);
14. ArrayList<NameValuePair> pairs = covertParams2NVPS(params);
15. ub.setParameters(pairs);
17. HttpGet httpGet = **new** HttpGet(ub.build());
18. **return** getResult(httpGet);
19. }
21. **public** **static** String httpGetRequest(String url, Map<String, Object> headers, Map<String, Object> params)
22. **throws** URISyntaxException {
23. URIBuilder ub = **new** URIBuilder();
24. ub.setPath(url);
26. ArrayList<NameValuePair> pairs = covertParams2NVPS(params);
27. ub.setParameters(pairs);
29. HttpGet httpGet = **new** HttpGet(ub.build());
30. **for** (Map.Entry<String, Object> param : headers.entrySet()) {
31. httpGet.addHeader(param.getKey(), String.valueOf(param.getValue()));
32. }
33. **return** getResult(httpGet);
34. }
36. **public** **static** String httpPostRequest(String url) {
37. HttpPost httpPost = **new** HttpPost(url);
38. **return** getResult(httpPost);
39. }
41. **public** **static** String httpPostRequest(String url, Map<String, Object> params) **throws** UnsupportedEncodingException {
42. HttpPost httpPost = **new** HttpPost(url);
43. ArrayList<NameValuePair> pairs = covertParams2NVPS(params);
44. httpPost.setEntity(**new** UrlEncodedFormEntity(pairs, UTF\_8));
45. **return** getResult(httpPost);
46. }
48. **public** **static** String httpPostRequest(String url, Map<String, Object> headers, Map<String, Object> params)
49. **throws** UnsupportedEncodingException {
50. HttpPost httpPost = **new** HttpPost(url);
52. **for** (Map.Entry<String, Object> param : headers.entrySet()) {
53. httpPost.addHeader(param.getKey(), String.valueOf(param.getValue()));
54. }
56. ArrayList<NameValuePair> pairs = covertParams2NVPS(params);
57. httpPost.setEntity(**new** UrlEncodedFormEntity(pairs, UTF\_8));
59. **return** getResult(httpPost);
60. }
62. **private** **static** ArrayList<NameValuePair> covertParams2NVPS(Map<String, Object> params) {
63. ArrayList<NameValuePair> pairs = **new** ArrayList<NameValuePair>();
64. **for** (Map.Entry<String, Object> param : params.entrySet()) {
65. pairs.add(**new** BasicNameValuePair(param.getKey(), String.valueOf(param.getValue())));
66. }
68. **return** pairs;
69. }

### 4.1.3 处理http请求

1. **private** **static** String getResult(HttpRequestBase request) {
2. // CloseableHttpClient httpClient = HttpClients.createDefault();
3. CloseableHttpClient httpClient = getHttpClient();
4. **try** {
5. CloseableHttpResponse response = httpClient.execute(request);
6. // response.getStatusLine().getStatusCode();
7. HttpEntity entity = response.getEntity();
8. **if** (entity != **null**) {
9. // long len = entity.getContentLength();// -1 表示长度未知
10. String result = EntityUtils.toString(entity);
11. response.close();
12. // httpClient.close();
13. **return** result;
14. }
15. } **catch** (ClientProtocolException e) {
16. e.printStackTrace();
17. } **catch** (IOException e) {
18. e.printStackTrace();
19. } **finally** {
21. }
23. **return** EMPTY\_STR;
24. }

## 4.2 简单实例

### 4.2.1 初始配置

1. **private** RequestConfig requestConfig = RequestConfig.custom()
2. .setSocketTimeout(15000)
3. .setConnectTimeout(15000)
4. .setConnectionRequestTimeout(15000)
5. .build();
7. **private** **static** HttpClientUtil instance = **null**;
8. **private** HttpClientUtil(){}
9. **public** **static** HttpClientUtil getInstance(){
10. **if** (instance == **null**) {
11. instance = **new** HttpClientUtil();
12. }
13. **return** instance;
14. }

### 4.2.2 发送post请求

1. \*/
2. **public** String sendHttpPost(String httpUrl) {
3. HttpPost httpPost = **new** HttpPost(httpUrl);// 创建httpPost
4. **return** sendHttpPost(httpPost);
5. }
6. /\*\*
7. \* 发送 post请求
8. \* @param httpUrl 地址
9. \* @param params 参数(格式:key1=value1&key2=value2)
10. \*/
11. **public** String sendHttpPost(String httpUrl, String params) {
12. HttpPost httpPost = **new** HttpPost(httpUrl);// 创建httpPost
13. **try** {
14. //设置参数
15. StringEntity stringEntity = **new** StringEntity(params, "UTF-8");
16. stringEntity.setContentType("application/x-www-form-urlencoded");
17. httpPost.setEntity(stringEntity);
18. } **catch** (Exception e) {
19. e.printStackTrace();
20. }
21. **return** sendHttpPost(httpPost);
22. }
23. /\*\*
24. \* 发送 post请求
25. \* @param httpUrl 地址
26. \* @param maps 参数
27. \*/
28. **public** String sendHttpPost(String httpUrl, Map<String, String> maps) {
29. HttpPost httpPost = **new** HttpPost(httpUrl);// 创建httpPost
30. // 创建参数队列
31. List<NameValuePair> nameValuePairs = **new** ArrayList<NameValuePair>();
32. **for** (String key : maps.keySet()) {
33. nameValuePairs.add(**new** BasicNameValuePair(key, maps.get(key)));
34. }
35. **try** {
36. httpPost.setEntity(**new** UrlEncodedFormEntity(nameValuePairs, "UTF-8"));
37. } **catch** (Exception e) {
38. e.printStackTrace();
39. }
40. **return** sendHttpPost(httpPost);
41. }
42. /\*\*
43. \* 发送 post请求（带文件）
44. \* @param httpUrl 地址
45. \* @param maps 参数
46. \* @param fileLists 附件
47. \*/
48. **public** String sendHttpPost(String httpUrl, Map<String, String> maps, List<File> fileLists) {
49. HttpPost httpPost = **new** HttpPost(httpUrl);// 创建httpPost
50. MultipartEntityBuilder meBuilder = MultipartEntityBuilder.create();
51. **for** (String key : maps.keySet()) {
52. meBuilder.addPart(key, **new** StringBody(maps.get(key), ContentType.TEXT\_PLAIN));
53. }
54. **for**(File file : fileLists) {
55. FileBody fileBody = **new** FileBody(file);
56. meBuilder.addPart("files", fileBody);
57. }
58. HttpEntity reqEntity = meBuilder.build();
59. httpPost.setEntity(reqEntity);
60. **return** sendHttpPost(httpPost);
61. }
63. /\*\*
64. \* 发送Post请求
65. \* @param httpPost
66. \* @return
67. \*/
68. **private** String sendHttpPost(HttpPost httpPost) {
69. CloseableHttpClient httpClient = **null**;
70. CloseableHttpResponse response = **null**;
71. HttpEntity entity = **null**;
72. String responseContent = **null**;
73. **try** {
74. // 创建默认的httpClient实例.
75. httpClient = HttpClients.createDefault();
76. httpPost.setConfig(requestConfig);
77. // 执行请求
78. response = httpClient.execute(httpPost);
79. entity = response.getEntity();
80. responseContent = EntityUtils.toString(entity, "UTF-8");
81. } **catch** (Exception e) {
82. e.printStackTrace();
83. } **finally** {
84. **try** {
85. // 关闭连接,释放资源
86. **if** (response != **null**) {
87. response.close();
88. }
89. **if** (httpClient != **null**) {
90. httpClient.close();
91. }
92. } **catch** (IOException e) {
93. e.printStackTrace();
94. }
95. }
96. **return** responseContent;
97. }

### 4.2.3 发送get请求

1. /\*\*
2. \* 发送 get请求
3. \* @param httpUrl
4. \*/
5. **public** String sendHttpGet(String httpUrl) {
6. HttpGet httpGet = **new** HttpGet(httpUrl);// 创建get请求
7. **return** sendHttpGet(httpGet);
8. }
10. /\*\*
11. \* 发送 get请求Https
12. \* @param httpUrl
13. \*/
14. **public** String sendHttpsGet(String httpUrl) {
15. HttpGet httpGet = **new** HttpGet(httpUrl);// 创建get请求
16. **return** sendHttpsGet(httpGet);
17. }
19. /\*\*
20. \* 发送Get请求
21. \* @param httpPost
22. \* @return
23. \*/
24. **private** String sendHttpGet(HttpGet httpGet) {
25. CloseableHttpClient httpClient = **null**;
26. CloseableHttpResponse response = **null**;
27. HttpEntity entity = **null**;
28. String responseContent = **null**;
29. **try** {
30. // 创建默认的httpClient实例.
31. httpClient = HttpClients.createDefault();
32. httpGet.setConfig(requestConfig);
33. // 执行请求
34. response = httpClient.execute(httpGet);
35. entity = response.getEntity();
36. responseContent = EntityUtils.toString(entity, "UTF-8");
37. } **catch** (Exception e) {
38. e.printStackTrace();
39. } **finally** {
40. **try** {
41. // 关闭连接,释放资源
42. **if** (response != **null**) {
43. response.close();
44. }
45. **if** (httpClient != **null**) {
46. httpClient.close();
47. }
48. } **catch** (IOException e) {
49. e.printStackTrace();
50. }
51. }
52. **return** responseContent;
53. }
55. /\*\*
56. \* 发送Get请求Https
57. \* @param httpPost
58. \* @return
59. \*/
60. **private** String sendHttpsGet(HttpGet httpGet) {
61. CloseableHttpClient httpClient = **null**;
62. CloseableHttpResponse response = **null**;
63. HttpEntity entity = **null**;
64. String responseContent = **null**;
65. **try** {
66. // 创建默认的httpClient实例.
67. PublicSuffixMatcher publicSuffixMatcher = PublicSuffixMatcherLoader.load(**new** URL(httpGet.getURI().toString()));
68. DefaultHostnameVerifier hostnameVerifier = **new** DefaultHostnameVerifier(publicSuffixMatcher);
69. httpClient = HttpClients.custom().setSSLHostnameVerifier(hostnameVerifier).build();
70. httpGet.setConfig(requestConfig);
71. // 执行请求
72. response = httpClient.execute(httpGet);
73. entity = response.getEntity();
74. responseContent = EntityUtils.toString(entity, "UTF-8");
75. } **catch** (Exception e) {
76. e.printStackTrace();
77. } **finally** {
78. **try** {
79. // 关闭连接,释放资源
80. **if** (response != **null**) {
81. response.close();
82. }
83. **if** (httpClient != **null**) {
84. httpClient.close();
85. }
86. } **catch** (IOException e) {
87. e.printStackTrace();
88. }
89. }
90. **return** responseContent;
91. }
92. }

### 4.2.4 测试代码

1. @RequestMapping(value = "/test/send")
2. @ResponseBody
3. **public** Map<String, String> sendPost(HttpServletRequest request) {
4. Map<String, String> maps = **new** HashMap<String, String>();
5. String username = request.getParameter("username");
6. String password = request.getParameter("password");
7. maps.put("username", username);
8. maps.put("password", password);
9. **return** maps;
10. }
12. @RequestMapping(value = "/test/sendpost/file",method=RequestMethod.POST)
13. @ResponseBody
14. **public** Map<String, String> sendPostFile(@RequestParam("files") MultipartFile [] files,HttpServletRequest request) {
15. Map<String, String> maps = **new** HashMap<String, String>();
16. String username = request.getParameter("username");
17. String password = request.getParameter("password");
18. maps.put("username", username);
19. maps.put("password", password);
21. **try** {
22. **for**(MultipartFile file : files){
23. String fileName = file.getOriginalFilename();
24. fileName = **new** String(fileName.getBytes(),"UTF-8");
25. InputStream is = file.getInputStream();
26. **if** (fileName != **null** && !("".equals(fileName))) {
27. File directory = **new** File("D://test//httpclient//file");
28. **if** (!directory.exists()) {
29. directory.mkdirs();
30. }
31. String filePath = ("D://test//httpclient//file") + File.separator + fileName;
32. FileOutputStream fos = **new** FileOutputStream(filePath);
33. **byte**[] buffer = **new** **byte**[1024];
34. **while** (is.read(buffer) > 0) {
35. fos.write(buffer, 0, buffer.length);
36. }
37. fos.flush();
38. fos.close();
39. maps.put("file--"+fileName, "uploadSuccess");
40. }
41. }
42. } **catch** (Exception e) {
43. e.printStackTrace();
44. }
45. **return** maps;
46. }

# 5 详细教程

## 5.1 基础

### 1.1 Request 的执行

HttpClient最必不可少的功能就是执行HTTP的方法，执行HTTP方法会涉及到一个或者多个HTTP request/HTTP response交换，而这些过程通常会在HttpClient内部完成。使用者提交一个request的对象去执行，HttpClient会发送这个request到目标服务器并且获得一个对应的response对象，如果不成功的话则抛出一个异常。

自然而然，满足了上面描述的HttpClient接口就是HttpClient API的主要入口。

下面是一个最简单的request执行过程：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_256](https://code.csdn.net/snippets/1597743)[IMG_257](https://code.csdn.net/snippets/1597743/fork)

1. CloseableHttpClient httpclient = HttpClients.createDefault();
2. HttpGet httpget = **new** HttpGet("http://localhost/");
3. CloseableHttpResponse response = httpclient.execute(httpget);
4. **try** {
5. <...>
6. } **finally** {
7. response.close();
8. }

**1.1.1 HTTP request**

所有的HTTP request都有一个请求线，包含了请求的方法名称，请求的URI和HTTP协议版本。

HttpClient开箱即用的支持所有HTTP/1.1中定义的HTTP方法，包括GET,HEAD,POST,PUT,DELETE,TRACE 和 OPTIONS，这些HTTP方法类型都有一个与之对应的类：HttpGet,HttpHead,HttpPost,HttpPut,HttpDelete,HttpTrace 和 HttpOptions。

Request-URI是一个统一资源定位符，它指明了用于处理该request的资源的位置。HTTP request URIs包含了一个协议类型，主机名，可选端口，资源路径，可选查询参数，可选片段。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_258](https://code.csdn.net/snippets/1597743)[IMG_259](https://code.csdn.net/snippets/1597743/fork)

1. HttpGet httpget = **new** HttpGet(
2. "http://www.google.com/search?hl=en&q=httpclient&btnG=Google+Search&aq=f&oq=");
3. HttpClient提供URIBuilder实用类用于简化request URIS的创建和修改。
4. URI uri = **new** URIBuilder()
5. .setScheme("http")
6. .setHost("www.google.com")
7. .setPath("/search")
8. .setParameter("q", "httpclient")
9. .setParameter("btnG", "Google Search")
10. .setParameter("aq", "f")
11. .setParameter("oq", "")
12. .build();
13. HttpGet httpget = **new** HttpGet(uri);
14. System.out.println(httpget.getURI());

输出：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_260](https://code.csdn.net/snippets/1597743)[IMG_261](https://code.csdn.net/snippets/1597743/fork)

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

**1.1.2 HTTP response**

HTTP response是服务器在接收并且处理完request消息之后返回给客户端的消息，消息的第一行包含了协议版本，status code和与之关联的文本。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_262](https://code.csdn.net/snippets/1597743)[IMG_263](https://code.csdn.net/snippets/1597743/fork)

1. HttpResponse response = **new** BasicHttpResponse(HttpVersion.HTTP\_1\_1,
2. HttpStatus.SC\_OK, "OK");
4. System.out.println(response.getProtocolVersion());
5. System.out.println(response.getStatusLine().getStatusCode());
6. System.out.println(response.getStatusLine().getReasonPhrase());
7. System.out.println(response.getStatusLine().toString());

输出：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_264](https://code.csdn.net/snippets/1597743)[IMG_265](https://code.csdn.net/snippets/1597743/fork)

1. HTTP/1.1
2. 200
3. OK
4. HTTP/1.1 200 OK

**1.1.3 消息头处理**

HTTP消息包含多个header，这些header描述了消息的属性比如content length,content type等等，HttpClient提供了取出，添加，删除和遍历header的方法。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_266](https://code.csdn.net/snippets/1597743)[IMG_267](https://code.csdn.net/snippets/1597743/fork)

1. HttpResponse response = **new** BasicHttpResponse(HttpVersion.HTTP\_1\_1, HttpStatus.SC\_OK, "OK");
2. response.addHeader("Set-Cookie", "c1=a; path=/; domain=localhost");
3. response.addHeader("Set-Cookie", "c2=b; path=\"/\", c3=c; domain=\"localhost\"");
4. Header h1 = response.getFirstHeader("Set-Cookie");
5. System.out.println(h1);
6. Header h2 = response.getLastHeader("Set-Cookie");
7. System.out.println(h2);
8. Header[] hs = response.getHeaders("Set-Cookie");
9. System.out.println(hs.length);

输出：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_268](https://code.csdn.net/snippets/1597743)[IMG_269](https://code.csdn.net/snippets/1597743/fork)

1. Set-Cookie: c1=a; path=/; domain=localhost
2. Set-Cookie: c2=b; path="/", c3=c; domain="localhost"
3. 2

获取所有指定headers的最高效的方式是使用HeaderIterator接口。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_270](https://code.csdn.net/snippets/1597743)[IMG_271](https://code.csdn.net/snippets/1597743/fork)

1. HttpResponse response = **new** BasicHttpResponse(HttpVersion.HTTP\_1\_1, HttpStatus.SC\_OK, "OK");
2. response.addHeader("Set-Cookie", "c1=a; path=/; domain=localhost");
3. response.addHeader("Set-Cookie", "c2=b; path=\"/\", c3=c; domain=\"localhost\"");
5. HeaderIterator it = response.headerIterator("Set-Cookie");
7. **while** (it.hasNext()) {
8. System.out.println(it.next());
9. }

输出：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_272](https://code.csdn.net/snippets/1597743)[IMG_273](https://code.csdn.net/snippets/1597743/fork)

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

该接口同时也提供HTTP消息转换到独立header元素的便利方法。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_274](https://code.csdn.net/snippets/1597743)[IMG_275](https://code.csdn.net/snippets/1597743/fork)

1. HttpResponse response = **new** BasicHttpResponse(HttpVersion.HTTP\_1\_1, HttpStatus.SC\_OK, "OK");
2. response.addHeader("Set-Cookie", "c1=a; path=/; domain=localhost");
3. response.addHeader("Set-Cookie", "c2=b; path=\"/\", c3=c; domain=\"localhost\"");
5. HeaderElementIterator it = **new** BasicHeaderElementIterator(response.headerIterator("Set-Cookie"));
7. **while** (it.hasNext()) {
8. HeaderElement elem = it.nextElement();
9. System.out.println(elem.getName() + " = " + elem.getValue());
10. NameValuePair[] params = elem.getParameters();
11. **for** (**int** i = 0; i < params.length; i++) {
12. System.out.println(" " + params[i]);
13. }
14. }

输出：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_276](https://code.csdn.net/snippets/1597743)[IMG_277](https://code.csdn.net/snippets/1597743/fork)

1. c1 = a
2. path=/
3. domain=localhost
4. c2 = b
5. path=/
6. c3 = c
7. domain=localhost

**1.1.4 HTTP entity**

HTTP消息可以携带跟request和response相关联的content entity。由于这些Entity是可选的，所以他们在某些request和response中可以查找到，有些则查不到。使用了entity的request被称为entity封装请求，HTTP定义了两种entity封装请求方法：POST 和 PUT。Response通常会封装一个content entity。不同的方法会有对应的异常，比如对于HEAD来说就有204 No Content,304 Not Modified, 205 Reset Content异常。

HttpClient根据内容来源将Entities区分为3种，根据内容来源区分为：

**streamed:** 内容来自于流或者实时产生。特别的是，这个分类包含了从HTTP responses接收到的entities，Streamed entities通常不可重复。

**self-contained:** 内容来自于内存或者通过独立于connection或者其他entity的手段获得，self-contained entities通常可以重复，其也是封装了entity的HTTP request最常用的Entities类型。

**wrapping :** 内容来自于其他的entity。

这种区分对于HTTP response输出内容时的连接管理是非常重要的。对于request entites来说，由于其是被应用创建并且通过HttpClient发送，streamed和self-contained有什么不同就没有那么重要。

由此，建议将不可重复的entites视为streamed，可以重复的entities视为self-contained。

**1.1.4.1 可重复的entities**

一个entity能够重复，表明它的内容可以被读取多次，这种情况只会在self contained entities中发生（比如ByteArrayEntity 或者 StringEntity）

**1.1.4.2 使用HTTP entities**

Entity可以同时代表二进制和文字内容，支持对字符进行编码，比如对character content进行编码。

Entity产生于request（封装内容 ）执行时，或者request成功请求并且response body用于发送结果数据到客户端时。

从Entity中读取内容，可以通过HttpEntity#getContent()获取input stream，也可以向HttpEntity#writeTo(OutputStream)提供一个output stream，该方法会将内容一次性全部写回给指派的stream。

当消息到达并且Entity已经被接收时，可以用 HttpEntity#getContentType() 和 HttpEntity#getContentLength() 方法读取常用的元数据，如Content-Type 和 Content-Length 头信息，Content-Type头信息包含文本的mime-type编码信息如 text/plain 或者 text/html，该信息可以通过 HttpEntity#getContentType() 获取到，如果Header里面不包含这些信息，那么 HttpEntity#getContentLength()  会返回-1 并且  HttpEntity#getContentType()  返回 NULL，如果Header包含这些信息，那么 HttpEntity#getContentType() 会返回 Header 对象。

当创建一个出站消息的entity时，必须同时指定其相关的参数。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_278](https://code.csdn.net/snippets/1597743)[IMG_279](https://code.csdn.net/snippets/1597743/fork)

1. StringEntity myEntity = **new** StringEntity("important message",
2. ContentType.create("text/plain", "UTF-8"));
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);

输出：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_280](https://code.csdn.net/snippets/1597743)[IMG_281](https://code.csdn.net/snippets/1597743/fork)

1. Content-Type: text/plain; charset=utf-8
2. 17
3. important message
4. 17

**1.1.5 确保释放低级资源**

为了保证恰当的释放系统资源，使用完毕后必须关闭entity相关的stream和response。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_282](https://code.csdn.net/snippets/1597743)[IMG_283](https://code.csdn.net/snippets/1597743/fork)

1. CloseableHttpClient httpclient = HttpClients.createDefault();
2. HttpGet httpget = **new** HttpGet("http://localhost/");
3. CloseableHttpResponse response = httpclient.execute(httpget);
4. **try** {
5. HttpEntity entity = response.getEntity();
6. **if** (entity != **null**) {
7. InputStream instream = entity.getContent();
8. **try** {
9. // do something useful
10. } **finally** {
11. instream.close();
12. }
13. }
14. } **finally** {
15. response.close();
16. }

关闭内容流和关闭response的不同之处在于，关闭内容流会尝试通过消费entity的content来保持底层连接，而关闭response则是直接关闭和丢弃掉这个连接。

注意 HttpEntity#writeTo(OutputStream)  方法也需要在entity被写出到流之后恰当的释放系统资源。如果通过HttpEntity#getContent()获取了**[Java](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "http://blog.csdn.net/ktlifeng/article/details/_blank)**.io.InputStream流的实例，那么在finally阶段也需要关闭该流。

当处理streaming entites时，可以使用 EntityUtils#consume(HttpEntity) 方法来确保entity内容已经被完全消费并且底层stream已经被关闭。

有另一种情况是，当我们只需要从response content里面获取一小部分数据，但是消费剩余数据和保持连接复用的性能损失又太高时，我们可以直接关闭response。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_284](https://code.csdn.net/snippets/1597743)[IMG_285](https://code.csdn.net/snippets/1597743/fork)

1. CloseableHttpClient httpclient = HttpClients.createDefault();
2. HttpGet httpget = **new** HttpGet("http://localhost/");
3. CloseableHttpResponse response = httpclient.execute(httpget);
4. **try** {
5. HttpEntity entity = response.getEntity();
6. **if** (entity != **null**) {
7. InputStream instream = entity.getContent();
8. **int** byteOne = instream.read();
9. **int** byteTwo = instream.read();
10. // Do not need the rest
11. }
12. } **finally** {
13. response.close();
14. }

这样这个连接不会被复用，连接的所有资源都会被正确的释放掉。

**1.1.6 消费entity内容**

消费Entity内容的推荐方式是使用 HttpEntity#getContent() 或者 HttpEntity#writeTo(OuptputStream) 方法，也可以使用EntityUtils类，该类提供了一些静态方法来简化读取Entity内容或者其他信息。通过使用该类的某些方法，你可以以String /byte[] 方式来获取整个content body，从而替代直接读取 InputStream 的方式。但是，除非你知道response entity来自于可信HTTP server 并且其长度有限，否则不推荐使用EntityUtils。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_286](https://code.csdn.net/snippets/1597743)[IMG_287](https://code.csdn.net/snippets/1597743/fork)

1. CloseableHttpClient httpclient = HttpClients.createDefault();
2. HttpGet httpget = **new** HttpGet("http://localhost/");
3. CloseableHttpResponse response = httpclient.execute(httpget);
4. **try** {
5. HttpEntity entity = response.getEntity();
6. **if** (entity != **null**) {
7. **long** len = entity.getContentLength();
8. **if** (len != -1 && len < 2048) {
9. System.out.println(EntityUtils.toString(entity));
10. } **else** {
11. // Stream content out
12. }
13. }
14. } **finally** {
15. response.close();
16. }

某些情况下你可能读取entity content不止一次，这时就需要将内容通过内存或者磁盘的方式缓存起来，最简单的实现方式就是使用BufferedHttpEntity包装源Entity，该类会将源 Entity 的内容读取到内存中。如果通过其他方式来实现，那么就必须要保存一个源entity了。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_288](https://code.csdn.net/snippets/1597743)[IMG_289](https://code.csdn.net/snippets/1597743/fork)

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

**1.1.7 生产 entity content**

HttpClient提供多个可以通过HTTP连接高效的输出内容的类，这些类的实例可以跟POST和PUT等request相关联并且为request封装entity内容，HttpClient提供了最常用的数据容器如string, byte array , input stream 和文件 ： StringEntity , ByteArrayEntity,InputStreamEntity,和FileEntity。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_290](https://code.csdn.net/snippets/1597743)[IMG_291](https://code.csdn.net/snippets/1597743/fork)

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

注意 InputStreamEntity 不可以重复，因为它只能从底层数据流读取一次。通常建议用HttpEntity的实现（self-contained）来替代普通的InputStreamEntity，FileEntity是一个不错的起始点。

**1.1.7.1 HTML表单**

许多应用程序需要模拟表单提交的过程，比如登录web应用或者提交input 数据，HttpClient提供了UrlEncodedFormEntity类来简化这个过程。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_292](https://code.csdn.net/snippets/1597743)[IMG_293](https://code.csdn.net/snippets/1597743/fork)

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.UTF\_8);
5. HttpPost httppost = **new** HttpPost("http://localhost/handler.do");
6. httppost.setEntity(entity);

UrlEncodedFormEntity实例会使用URL encoding来编码参数并且产生出如下内容：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_294](https://code.csdn.net/snippets/1597743)[IMG_295](https://code.csdn.net/snippets/1597743/fork)

1. param1=value1¶m2=value2

**1.1.7.2 内容分块**

通常来说，建议让HttpClient选择最合适的传输编码，HttpClient选择的传输编码会基于被传输的HTTP消息的属性而定。你可以通过设置HttpEntity#setChunked()为true来通知HttpClient需要进行chunk编码。注意HttpClient只是把这一标识当做一个提示使用，该值在不支持chunk编码的HTTP协议版本中如HTTP/1.0中会被忽略。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_296](https://code.csdn.net/snippets/1597743)[IMG_297](https://code.csdn.net/snippets/1597743/fork)

1. StringEntity entity = **new** StringEntity("important message",
2. ContentType.create("plain/text", Consts.UTF\_8));
3. entity.setChunked(**true**);
4. HttpPost httppost = **new** HttpPost("http://localhost/acrtion.do");
5. httppost.setEntity(entity);

**1.1.8 Response handlers**

最简单和方便处理response的方式是使用ResponseHandler，该类包含了handleResponse(HttpResponse response)方法。该方法使得用户完全不需要去操心连接管理的事情。当使用ResponseHandler时，HttpClient会自动确保将连接释放回连接管理器，无论request是否执行成功。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50805895" \o "copy) [IMG_298](https://code.csdn.net/snippets/1597743)[IMG_299](https://code.csdn.net/snippets/1597743/fork)

1. CloseableHttpClient httpclient = HttpClients.createDefault();
2. HttpGet httpget = **new** HttpGet("http://localhost/json");
4. ResponseHandler<MyJsonObject> rh = **new** ResponseHandler<MyJsonObject>() {
6. @Override
7. **public** JsonObject handleResponse(
8. **final** HttpResponse response) **throws** IOException {
9. StatusLine statusLine = response.getStatusLine();
10. HttpEntity entity = response.getEntity();
11. **if** (statusLine.getStatusCode() >= 300) {
12. **throw** **new** HttpResponseException(
13. statusLine.getStatusCode(),
14. statusLine.getReasonPhrase());
15. }
16. **if** (entity == **null**) {
17. **throw** **new** ClientProtocolException("Response contains no content");
18. }
19. Gson gson = **new** GsonBuilder().create();
20. ContentType contentType = ContentType.getOrDefault(entity);
21. Charset charset = contentType.getCharset();
22. Reader reader = **new** InputStreamReader(entity.getContent(), charset);
23. **return** gson.fromJson(reader, MyJsonObject.**class**);
24. }
25. };
26. MyJsonObject myjson = client.execute(httpget, rh);

### 1.2 HttpClient 接口

HttpClient接口代表HTTP request执行的最基本的协议。它不规定任何request执行过程中的限制或者细节，并且将连接管理，状态管理，认证和重定向处理交给各自独立实现，这使得它更容易去使用附加功能去装饰接口，如response content cacheing。

通常HttpClient的实现作为一系列特定的handler或者strategy接口的入口，而这些特定的handler或者strategy各自负责HTTP协议处理的一个方面，比如重定向或者认证处理，或者连接持久化和保活周期处理。这使得用户可以有选择的替代这些方面默认实现。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "copy) [IMG_256](https://code.csdn.net/snippets/1602161)[IMG_257](https://code.csdn.net/snippets/1602161/fork)

1. ConnectionKeepAliveStrategy keepAliveStrat = **new** DefaultConnectionKeepAliveStrategy() {
3. @Override
4. **public** **long** getKeepAliveDuration(
5. HttpResponse response,
6. HttpContext context) {
7. **long** keepAlive = **super**.getKeepAliveDuration(response, context);
8. **if** (keepAlive == -1) {
9. // Keep connections alive 5 seconds if a keep-alive value
10. // has not be explicitly set by the server
11. keepAlive = 5000;
12. }
13. **return** keepAlive;
14. }
16. };
17. CloseableHttpClient httpclient = HttpClients.custom()
18. .setKeepAliveStrategy(keepAliveStrat)
19. .build();

1.2.1 HttpClient线程安全

HttpClient的实现是线程安全的，建议不同的request 执行过程用同一个该类的实例。

1.2.2 HttpClient资源释放

当CloseableHttpClient不再使用并且将离开其所关联的连接管理器的范围时，其必须通过调用CloseableHttpClient#close方法关闭。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "copy) [IMG_258](https://code.csdn.net/snippets/1602161)[IMG_259](https://code.csdn.net/snippets/1602161/fork)

1. CloseableHttpClient httpclient = HttpClients.createDefault();
2. **try** {
3. <...>
4. } **finally** {
5. httpclient.close();
6. }

### 1.3 HTTP运行期上下文

最初HTTP被设计为无状态的，面向response-request的协议。但是现实世界中应用程序经常需要通过一些逻辑相关的request-response交换来保持状态信息。为了让应用程序能够保持处理状态，HttpClient允许HTTP requests在一个特别的运行期上下文中执行，即HTTP context。多数连续的逻辑相关的request如果使用相同的context，那么他们可以分享同一个逻辑session，HTTP context有点像**[Java](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "http://blog.csdn.net/ktlifeng/article/details/_blank)**.util.Map<String,Object>，它只是简单的手机任意形式的name-value。应用程序可以在request执行之前填充context或者市在执行之后检查context。

HttpContext可以包含任意的对象，因此在不同的线程之间共享是不安全的，建议每个线程保存其自己的Context。

在HTTP request请求的过程中，HttpClient将会添加如下属性到运行上下文中。

HttpConnection 实例代表着到目标服务器的真实连接。

HttpHost实例代表着连接的目标

HttpRoute实例代表着完整的连接路由

HttpRequest 实例代表着实际的HTTP request请求，因为HttpRequest是发送到目标服务器中的，所以它最终在执行上下文中的状态即表示着这条消息的状态。一般来说HTTP/1.0和HTTP、1.1的request都默认使用相对路径，但是如果request是通过代理服务器，而且工作在非隧道模式下，那么这时就必须使用绝对路径了。

HttpResponse 实例代表着实际的HTTP response

java.lang.Boolean 对象标识着request是否完全发送到了目标服务器

RequestConfig 对象表示实际的request配置

java.util.List<URI> 对象表示着在执行request过程中受到的所有重定向请求。

你可以使用HttpClientContext适配器类简化与context的交互操作。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "copy) [IMG_260](https://code.csdn.net/snippets/1602161)[IMG_261](https://code.csdn.net/snippets/1602161/fork)

1. HttpContext context = <...>
2. HttpClientContext clientContext = HttpClientContext.adapt(context);
3. HttpHost target = clientContext.getTargetHost();
4. HttpRequest request = clientContext.getRequest();
5. HttpResponse response = clientContext.getResponse();
6. RequestConfig config = clientContext.getRequestConfig();

多个逻辑相关，依赖于session的request应该在同样的HttpContext上下文环境中执行以保证会话上下文和状态能够在不同的request中传递。

在下面的例子中，request初始化时创建了request configuration配置，此配置会保存在执行上下文中并且在连续不同的request之间传递用以分享同样的上下文。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "copy) [IMG_262](https://code.csdn.net/snippets/1602161)[IMG_263](https://code.csdn.net/snippets/1602161/fork)

1. CloseableHttpClient httpclient = HttpClients.createDefault();
2. RequestConfig requestConfig = RequestConfig.custom()
3. .setSocketTimeout(1000)
4. .setConnectTimeout(1000)
5. .build();
7. HttpGet httpget1 = **new** HttpGet("http://localhost/1");
8. httpget1.setConfig(requestConfig);
9. CloseableHttpResponse response1 = httpclient.execute(httpget1, context);
10. **try** {
11. HttpEntity entity1 = response1.getEntity();
12. } **finally** {
13. response1.close();
14. }
15. HttpGet httpget2 = **new** HttpGet("http://localhost/2");
16. CloseableHttpResponse response2 = httpclient.execute(httpget2, context);
17. **try** {
18. HttpEntity entity2 = response2.getEntity();
19. } **finally** {
20. response2.close();
21. }

### 1.4 HTTP protocol 拦截器

HTTP protocol拦截器是实现了HTTP protocol某些特定方面的一段程序。通常，protocol拦截器会对入栈消息的某一个或者一组指定的headers起作用，或者用某一个或者一组特定的headers填充出栈消息。protocol拦截器也可以操作封装好内容实体的消息，比如进行内容压缩或者解压缩，通常我们会通过‘Decorator’模式来完成这些操作，特别是在使用实体包装器类来封装源实体时。多个protocol拦截器可以组合形成一个逻辑性的单元。

Protocol拦截器可以通过共享信息（比如HTTP执行上下文中的处理状态）来进行合作，Protocol拦截器也可以通过HTTPContext来存储一个或者一系列request的处理状态。

由于拦截器不依赖于执行环境中的某个特定的状态，所以他们的执行顺序并不重要，但是如果你对protocol的执行顺序有要求，那么他们应该按照执行的顺序添加到protocol processor中去。

Procotol必须是线程安全的，跟Servlets类似，protocol拦截器不应该使用实例变量，除非这些实例变量是同步的。

下面是一个local context在连续的request中存储处理状态的例子。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "copy) [IMG_264](https://code.csdn.net/snippets/1602161)[IMG_265](https://code.csdn.net/snippets/1602161/fork)

1. CloseableHttpClient httpclient = HttpClients.custom()
2. .addInterceptorLast(**new** HttpRequestInterceptor() {
4. **public** **void** process(
5. **final** HttpRequest request,
6. **final** HttpContext context) **throws** HttpException, IOException {
7. AtomicInteger count = (AtomicInteger) context.getAttribute("count");
8. request.addHeader("Count", Integer.toString(count.getAndIncrement()));
9. }
11. })
12. .build();
14. AtomicInteger count = **new** AtomicInteger(1);
15. HttpClientContext localContext = HttpClientContext.create();
16. localContext.setAttribute("count", count);
18. HttpGet httpget = **new** HttpGet("http://localhost/");
19. **for** (**int** i = 0; i < 10; i++) {
20. CloseableHttpResponse response = httpclient.execute(httpget, localContext);
21. **try** {
22. HttpEntity entity = response.getEntity();
23. } **finally** {
24. response.close();
25. }
26. }

### 1.5. Exception 处理

HTTP protocol processors 会抛出两种异常：

java.io.IOException 在I/O失败时，如socket超时或者socket重置。

HttpException  标识HTTP 失败，如违反HTTP协议。

通常 I/O 失败是非致命并且可以恢复的，而HTTP protocol错误是致命的并且不可自动恢复的。请注意HttpClient的实现类将ClientProtocolException当做HttpException抛出，ClientProtocolException是java.io.IOException的子类，这使得用户在一个catch里可以同时处理I/O错误和协议错误。

1.5.1 HTTP 安全传输

要明白HTTP 协议并不是适合所有的应用。HTTP 是一个简单的request/response协议，它在最初的设计用于静态/动态内容检索，它从来没有试图去支持事务操作。举个例子来说，HTTP 服务器在接收并且处理request时只会考虑完成自己负责的这一部分协议的工作，生成response和发送状态码给客户端，如果客户端接收失败（读取超时，请求取消或者系统崩溃），服务器不会尝试去回滚这个操作，如果服务器支持事务，那么当客户端决定重试请求，服务器将不止一次的强制终止执行相同的事务，而这样会导致应用的数据损坏或者状态不一致。

即使HTTP从来没有考虑过支持事务处理，在某些特定的情况下它也可以作为关键应用的传输协议，为了保证HTTP传输层的安全，使用系统必须在应用层确保HTTP 方法的幂等性。

1.5.2 方法幂等性

HTTP/1.1 规范将幂等性做如下定义：

[ 方法有幂等性即表示N>0个相同请求的副作用跟单一请求的副作用一样 ]

换句话来说就表示应用应该确保已准备好处理执行多个相同方法带来的影响，而这是有办法可以实现的，比如提供一个唯一的事务id，意味着避免执行相同的逻辑操作。

请注意这个问题并不是专门针对于HttpClient，浏览器都会受到相同问题的影响。

基于兼容性的原因，HttpClient假设只有非实体封装的方法，比如GET 和HEAD是幂等性的，而实体封装方法如POST和PUT不是幂等性的。

1.5.3 异常自动恢复

HttpClient默认会自动尝试从I/O异常中恢复，自动恢复机制只作用于一些已经被认为是安全的异常：

HttpClient不会尝试从任何逻辑或者HTTP protocol错误中恢复（HttpException的衍生类）。

HttpClient会自动重试那些被认为是幂等性的方法。

HttpClient会自动重试在Http request发送到目标服务器时出现传输失败的方法。

1.5.4 Request retry handler

为了启用自定义的错误恢复机制，你应该提供一个HttpRequestRetryHandler接口的实现。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "copy) [IMG_266](https://code.csdn.net/snippets/1602161)[IMG_267](https://code.csdn.net/snippets/1602161/fork)

1. HttpRequestRetryHandler myRetryHandler = **new** HttpRequestRetryHandler() {
3. **public** **boolean** retryRequest(
4. IOException exception,
5. **int** executionCount,
6. HttpContext context) {
7. **if** (executionCount >= 5) {
8. // Do not retry if over max retry count
9. **return** **false**;
10. }
11. **if** (exception **instanceof** InterruptedIOException) {
12. // Timeout
13. **return** **false**;
14. }
15. **if** (exception **instanceof** UnknownHostException) {
16. // Unknown host
17. **return** **false**;
18. }
19. **if** (exception **instanceof** ConnectTimeoutException) {
20. // Connection refused
21. **return** **false**;
22. }
23. **if** (exception **instanceof** SSLException) {
24. // SSL handshake exception
25. **return** **false**;
26. }
27. HttpClientContext clientContext = HttpClientContext.adapt(context);
28. HttpRequest request = clientContext.getRequest();
29. **boolean** idempotent = !(request **instanceof** HttpEntityEnclosingRequest);
30. **if** (idempotent) {
31. // Retry if the request is considered idempotent
32. **return** **true**;
33. }
34. **return** **false**;
35. }
37. };
38. CloseableHttpClient httpclient = HttpClients.custom()
39. .setRetryHandler(myRetryHandler)
40. .build();

请注意你可以使用StandardHttpRequestRetryHandler来替换默认实现，从而实现RFC-2616中定义的幂等性方法的自动重试（GET,HEAD,PUT,DELETE,OPTIONS,AND TRACE）。

### 1.6 终止请求

在某些情况下，HTTP request在期望的时间内无法执行完成，导致目标服务器的高负载或者客户端存在许多同步的请求，从而你会需要去提前中断request请求并且将阻塞在I/O操作的执行线程释放出来，在HttpClient中执行的request可以在执行过程中的任何阶段通过调用HttpUriRequest#abort()终止，这个方法是线程安全的并且可以从任何线程起调。当HTTP request在执行线程中终止时，会抛出InterruptedIOException。

### 1.7 Redirect Handling

HttpClient自动处理所有类型的重定向，除了在HTTP规范中明确禁止并且要求用户干预的。在HTTP规范中，通过POST和PUT请求的303 redirect会被转换成GET请求。你可以使用自定义重定向策略去放宽HTTP规范中关于POST方法重定向的限制。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "copy) [IMG_268](https://code.csdn.net/snippets/1602161)[IMG_269](https://code.csdn.net/snippets/1602161/fork)

1. LaxRedirectStrategy redirectStrategy = **new** LaxRedirectStrategy();
2. CloseableHttpClient httpclient = HttpClients.custom()
3. .setRedirectStrategy(redirectStrategy)
4. .build();

HttpClient在其执行过程中经常需要重写request消息。HTTP/1.0和HTTP/1.1经常使用相对请求，同样的，原始请求可能会被重定向多次，最终确定的HTTP路径可以通过原始request和context组合出来，URIUtils#resolve实用方法可以用来明确的最终请求的URI路径。这个方法包括重定向请求或者原始request的最后一个片段标识符。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50829825" \o "copy) [IMG_270](https://code.csdn.net/snippets/1602161)[IMG_271](https://code.csdn.net/snippets/1602161/fork)

1. 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. HttpHost target = context.getTargetHost();
7. List<URI> redirectLocations = context.getRedirectLocations();
8. URI location = URIUtils.resolve(httpget.getURI(), target, redirectLocations);
9. System.out.println("Final HTTP location: " + location.toASCIIString());
10. // Expected to be an absolute URI
11. } **finally** {
12. response.close();
13. }

## 5.2 连接管理

### 5.2.1 连接持久化

        在两个主机之间建立连接的过程复杂并且可能相当耗时，这一过程涉及到多个数据包交换，，连接（特别是短连接）握手的开销会非常的大，我们可以通过多个request重用HTTP 连接来达到高吞吐数据量避免这一问题。

HTTP/1.1 默认HTTP连接可以被多个请求重用。HTTP/1.0标准的终端可以使用某些机制去显示的表达他们要想重用连接的意图。HTTP代理也能够保持一段时间内的空闲连接从而提供给同一主机的连续请求使用。保持连接的能力通常叫做连接持久化，HttpClient对连接持久化提供了全面的支持。

### 5.2.2 HTTP 路由

        HttpClient可以直接跟目标主机建立连接，也可以通过路由来建立多个连接--我们称之为“跳”。HttpClient将连接划分为plain（直连），tunneled（隧道），layered（分层连接），多个中间代理的隧道连接被称之为代理链。

直接到目标或者第一个并且是唯一一个代理的连接称之为直连，与目标之间的多个中间代理建立的连接称之为隧道路由，隧道路由不能脱离代理而存在，通过已有连接的分层协议的路由则称之为分层路由，分层协议只能基于隧道或者直连。

**2.2.1 路由计算**

        RouteInfo接口代表了到目标主机的一系列明确的路由信息，HttpRoute是RouteInfo的一个具体实现，该类是不可修改的。

        HttpTracker是一个可变的RouteInfo的实现，用于HttpClient内部来追踪到最终目标主机的剩余路由，HttpTracker可以在进行完一个成功的路由之后被更新。

        HttpRouteDirector则是用于计算到下一步路由的帮助类，该类也是在HttpClient内部使用。

        HttpRoutePlanner基于执行上下文来计算到目标主机的路由策略，HttpClient附带两个默认的HttpRoutePlanner实现，

        SystemDefaultRoutePlanner基于**[Java](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "http://blog.csdn.net/ktlifeng/article/details/_blank)[.NET](http://lib.csdn.net/base/dotnet" \o ".NET知识库" \t "http://blog.csdn.net/ktlifeng/article/details/_blank)**.ProxySelector，它默认从JVM，系统属性或者浏览器中提取代理配置。

        DefaultProxyRoutePlanner总是通过默认的策略来计算路由。

**2.2.2 HTTP安全连接**

        如果两个主机之间传递的信息不能被未授权的第三方读取或者篡改，那么这个连接就认为是安全的，SSL/TLS是使用最广泛的HTTP安全传输协议，虽然其他加密技术也可以很好的实现安全传输，但是HTTP传输则普遍使用SSL/TLS。

### 5.2.3 HTTP 连接管理器

**2.3.1 连接管理与连接管理器**

        HTTP连接是复杂的，无状态的，非线程安全的，需要妥善的管理以保证正确工作，一个HTTP连接同一时间只能被一个线程执行，HttpClient使用HttpClientConnectionManager接口作为连接管理器去管理HTTP连接，HTTP连接管理器的目标是

        1 成为创建HTTP新连接的工厂

        2 管理持久化连接的生命周期

        3 同步持久化连接的访问，保证同一时间只有一个线程可以访问同一个连接。

内部的HTTP连接管理器与ManagedHttpClientConnection实例协同工作，能够实现代理连接，管理连接状态，控制执行I/O操作。如果一个连接被释放或者被消费端明确的关闭了，底层连接将会从其代理分离出来交还连接管理器，即便是消费端仍然持有代理实例的引用，它也没办法做任何I/O操作或者是改变真实连接的状态。

下面是从连接管理器获取HTTP连接的一个例子：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_256](https://code.csdn.net/snippets/1605188)[IMG_257](https://code.csdn.net/snippets/1605188/fork)

1. HttpClientContext context = HttpClientContext.create();
2. HttpClientConnectionManager connMrg = **new** BasicHttpClientConnectionManager();
3. HttpRoute route = **new** HttpRoute(**new** HttpHost("localhost", 80));
4. // Request new connection. This can be a long process
5. ConnectionRequest connRequest = connMrg.requestConnection(route, **null**);
6. // Wait for connection up to 10 sec
7. HttpClientConnection conn = connRequest.get(10, TimeUnit.SECONDS);
8. **try** {
9. // If not open
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. // Do useful things with the connection.
17. } **finally** {
18. connMrg.releaseConnection(conn, **null**, 1, TimeUnit.MINUTES);
19. }

连接可以通过调用ConnectionRequest#cancel()提前终止，这会将ConnectionRequest#get()方法中阻塞的线程解除。

**3.2 简单连接管理器**

        BasicHttpClientConnectionManager是一个简单连接管理器，一次只保持一条连接，即便这个类是线程安全的，它也只应该被一个执行线程使用，BasicHttpClientConnectionManager对相同路由的连续请求将重用连接，如果新的连接跟已经持久化保持的连接不同，那么它会关闭已有的连接，根据所给的路由重新开启一个新的连接来使用，如果连接已经被分配出去了，那么java.lang.IllegalStateException异常会被抛出。

该连接管理器的实现应该在EJB容器内使用。

**2.3.3 连接池管理器**

        PoolingHttpClientConnectionManager是一个更加复杂的实现，其管理了一个连接池，能够为多个执行线程提供连接，连接依据路由归类放入到池中，当一个请求在连接池中有对应路由的连接时，连接管理器会从池中租借出一个持久化连接而不是创建一个带有标记的连接。

        PoolingHttpClientConnectionManager在总的和每条路由上都会保持最大数量限制的连接，默认该实现会为每个路由保持2个并行连接，总的数量上不超过20个连接，在现实使用中，这是限制可能太过于苛刻，尤其对于那些将HTTP作为传输协议的服务来说。

下面是一个如何调整连接池参数的例子：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_258](https://code.csdn.net/snippets/1605188)[IMG_259](https://code.csdn.net/snippets/1605188/fork)

1. PoolingHttpClientConnectionManager cm = **new** PoolingHttpClientConnectionManager();
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);
10. CloseableHttpClient httpClient = HttpClients.custom()
11. .setConnectionManager(cm)
12. .build();

**2.3.4 连接池关闭**

        当HttpClient实例不再使用并且将要离开作用域时，需要关闭连接管理器，确保所有的连接都被关闭，系统资源被正确释放，这一点非常重要。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_260](https://code.csdn.net/snippets/1605188)[IMG_261](https://code.csdn.net/snippets/1605188/fork)

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

### 5.2.4 多线程request执行

        当使用如PoolingClientConnectionManager的连接池管理器时，HttpClient可以通过多个执行线程同时执行多个请求。

        PoolingClientConnectionManager将会根据其配置分配连接，如果某个路由的所有连接都已经被分配出去了，新进来的请求将会阻塞直到某个连接被释放回连接池，你可以通过配置http.conn-manager.timeout 这个参数来配置新的请求进来时阻塞的超时时间，从而避免无限期等待，如果在给定时间内连接没有被获取到，那么将会抛出ConnectionPoolTimeoutException异常。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_262](https://code.csdn.net/snippets/1605188)[IMG_263](https://code.csdn.net/snippets/1605188/fork)

1. PoolingHttpClientConnectionManager cm = **new** PoolingHttpClientConnectionManager();
2. CloseableHttpClient httpClient = HttpClients.custom()
3. .setConnectionManager(cm)
4. .build();
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. };
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. }
21. // start the threads
22. **for** (**int** j = 0; j < threads.length; j++) {
23. threads[j].start();
24. }
26. // join the threads
27. **for** (**int** j = 0; j < threads.length; j++) {
28. threads[j].join();
29. }

由于HttpClient实例是线程安全的能够在多个执行线程之间共享，强烈建议每个线程维持其自己的专有HttpContext实例。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_264](https://code.csdn.net/snippets/1605188)[IMG_265](https://code.csdn.net/snippets/1605188/fork)

1. **static** **class** GetThread **extends** Thread {
3. **private** **final** CloseableHttpClient httpClient;
4. **private** **final** HttpContext context;
5. **private** **final** HttpGet httpget;
7. **public** GetThread(CloseableHttpClient httpClient, HttpGet httpget) {
8. **this**.httpClient = httpClient;
9. **this**.context = HttpClientContext.create();
10. **this**.httpget = httpget;
11. }
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. }
30. }

### 5.2.5 连接回收策略

        经典阻塞I/O模型的一个主要缺点就是网络socket只有在I/O操作阻塞的情况下才会对I/O事件作出反应。当连接释放回管理器时，它虽然能够保持存活，但是它无法监控socket的状态也无法对任何I/O事件作出反应。如果连接在Server端被关闭，client端连接无法侦测到连接状态的改变并且作出适当的回应。

        HttpClient尝试通过测试连接是否'stale'来缓解这个问题，stale的意思是连接不再有效因为在执行Http请求之前其已经被服务端关闭。过期连接检查不是100%可靠的，唯一可行解决方案是提供一个专用监控线程用于回收那些长时间内不活动的连接，而该解决方案不会影响到一个socket一个线程的空闲连接模型。监控线程可以定期调用        ClientConnectionManager#closeExpiredConnections()方法来关闭所有过期的连接，同时从连接池中回收已经被关闭的连接，也可以有选择性的调用        ClientConnectionManager#closeIdleConnection()方法去关闭那些在给定时间范围内空闲的连接。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_266](https://code.csdn.net/snippets/1605188)[IMG_267](https://code.csdn.net/snippets/1605188/fork)

1. **public** **static** **class** IdleConnectionMonitorThread **extends** Thread {
3. **private** **final** HttpClientConnectionManager connMgr;
4. **private** **volatile** **boolean** shutdown;
6. **public** IdleConnectionMonitorThread(HttpClientConnectionManager connMgr) {
7. **super**();
8. **this**.connMgr = connMgr;
9. }
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. }
29. **public** **void** shutdown() {
30. shutdown = **true**;
31. **synchronized** (**this**) {
32. notifyAll();
33. }
34. }
36. }

### 5.2.6连接保活策略

        HTTP规范没有明确指出一个连接应该被持久化保持多长时间，一些HTTP 服务器使用非标准的Keep-Aliveheader去同客户端沟通他们想要在server端保持多久的连接，HttpClient也会利用这个信息，如果Keep-Alive消息头在response中不存在，HttpClient假设连接可以被无限期的保持存活，但是很多HTTP服务端通常配置为在持久化连接一段时间内不活动时就丢弃以节省系统资源，而丢弃时往往不通知客户端，在这种情况下，默认的策略会变得过于乐观，你可能想要提供一个自定义keep-alive策略。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_268](https://code.csdn.net/snippets/1605188)[IMG_269](https://code.csdn.net/snippets/1605188/fork)

1. ConnectionKeepAliveStrategy myStrategy = **new** ConnectionKeepAliveStrategy() {
3. **public** **long** getKeepAliveDuration(HttpResponse response, HttpContext context) {
4. // Honor 'keep-alive' header
5. HeaderElementIterator it = **new** BasicHeaderElementIterator(
6. response.headerIterator(HTTP.CONN\_KEEP\_ALIVE));
7. **while** (it.hasNext()) {
8. HeaderElement he = it.nextElement();
9. String param = he.getName();
10. String value = he.getValue();
11. **if** (value != **null** && param.equalsIgnoreCase("timeout")) {
12. **try** {
13. **return** Long.parseLong(value) \* 1000;
14. } **catch**(NumberFormatException ignore) {
15. }
16. }
17. }
18. HttpHost target = (HttpHost) context.getAttribute(
19. HttpClientContext.HTTP\_TARGET\_HOST);
20. **if** ("www.naughty-server.com".equalsIgnoreCase(target.getHostName())) {
21. // Keep alive for 5 seconds only
22. **return** 5 \* 1000;
23. } **else** {
24. // otherwise keep alive for 30 seconds
25. **return** 30 \* 1000;
26. }
27. }
29. };
30. CloseableHttpClient client = HttpClients.custom()
31. .setKeepAliveStrategy(myStrategy)
32. .build();

### 5.2.7 连接套接字工厂

        HTTP连接内部使用java**[.Net](http://lib.csdn.net/base/dotnet" \o ".NET知识库" \t "http://blog.csdn.net/ktlifeng/article/details/_blank)**.Socket对象来处理线路上的数据传输，但是他们依赖于ConnectionSocketFactory接口来创建，初始化和连接socket，这使得HttpClient的用户可以在运行时提供应用程序特定的socket初始化代码。PlainConnectionSocketFactory是创建和初始化普通socket(非加密)的默认工厂类。

创建socket和连接socket的过程是分离的，这样socket在连接操作阻塞时就可以被关闭掉。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_270](https://code.csdn.net/snippets/1605188)[IMG_271](https://code.csdn.net/snippets/1605188/fork)

1. HttpClientContext clientContext = HttpClientContext.create();
2. PlainConnectionSocketFactory sf = PlainConnectionSocketFactory.getSocketFactory();
3. Socket socket = sf.createSocket(clientContext);
4. **int** timeout = 1000; //ms
5. HttpHost target = **new** HttpHost("localhost");
6. InetSocketAddress remoteAddress = **new** InetSocketAddress(
7. InetAddress.getByAddress(**new** **byte**[] {127,0,0,1}), 80);
8. sf.connectSocket(timeout, socket, target, remoteAddress, **null**, clientContext);

**2.7.1 安全套接字层**

        LayeredConnectionSocketFactory是ConnectionSocketFactory接口的一个扩展，分层套接字工厂能够在已存在的普通socket上创建分层socket，Socket分层主要用于通过代理创建安全socket，HttpClient附带的SSLSocketFactory实现了SSL/TLS层协议，请注意HttpClient不会使用任何定制的加密功能，它完全依赖JCE（标准JAVA加密）和JSEE（安全套接字）扩展。

**2.7.2 同连接管理器整合**

自定义连接套接字工厂可以跟特定的协议（如HTTP,HTTPS）关联起来用于创建自定义的连接管理器。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_272](https://code.csdn.net/snippets/1605188)[IMG_273](https://code.csdn.net/snippets/1605188/fork)

1. ConnectionSocketFactory plainsf = <...>
2. LayeredConnectionSocketFactory sslsf = <...>
3. Registry<ConnectionSocketFactory> r = RegistryBuilder.<ConnectionSocketFactory>create()
4. .register("http", plainsf)
5. .register("https", sslsf)
6. .build();
8. HttpClientConnectionManager cm = **new** PoolingHttpClientConnectionManager(r);
9. HttpClients.custom()
10. .setConnectionManager(cm)
11. .build();

**2.7.3 SSL/TLS 定制**

        HttpClient使用SSLConnectionSocketFactory来创建SSL连接，SSLConnectionSocketFactory允许高度定制化，它可以通过 javax.net.ssl.SSLContext 参数来创建定制的SSL连接。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_274](https://code.csdn.net/snippets/1605188)[IMG_275](https://code.csdn.net/snippets/1605188/fork)

1. KeyStore myTrustStore = <...>
2. SSLContext sslContext = SSLContexts.custom()
3. .loadTrustMaterial(myTrustStore)
4. .build();
5. SSLConnectionSocketFactory sslsf = **new** SSLConnectionSocketFactory(sslContext);

        定制SSLConnectionSocketFactory意味着对SSL/TLS协议有一定程度的了解，其详细说明已经超出本文档的范围，请查看http://docs.**[Oracle](http://lib.csdn.net/base/oracle" \o "Oracle知识库" \t "http://blog.csdn.net/ktlifeng/article/details/_blank)**.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html

来获取javax.net.ssl.SSLContext和其相关工具的详细说明。

**2.7.4 主机名验证**

        除了信任验证和客户端身份验证在SSL/TLS协议层进行之外，HttpClient可以有选择的验证目标主机名是否跟服务端存储在X.509认证里的一致，一旦连接已经建立，这种验证可以为服务器认证提供额外的保障，javax.net.ssl.HostnameVerifier 接口代表了主机名验证的一种策略，HttpClient附带了两中javax.net.ssl.HostnameVerifier的实现，注意：不要把主机名验证跟SSL信任验证混淆

        DefaultHostnameVerifier: HttpClient使用的默认实现，与RFC2818兼容，主机名必须匹配证书指定的任何可替换的名称，或者没有可替换名称下证书主体中指定的具体的CN，CN和可替换名称中都可能有通配符。

        NoopHostnameVerifier: 这个主机名验证器基本上就是把主机名验证关闭了，它接受任何有效的SSL会话来匹配目标主机。

默认HttpClient使用DefaultHostnameVerifier实现，如果有需要的话你可以指定一个不同的主机名验证器

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_276](https://code.csdn.net/snippets/1605188)[IMG_277](https://code.csdn.net/snippets/1605188/fork)

1. SSLContext sslContext = SSLContexts.createSystemDefault();
2. SSLConnectionSocketFactory sslsf = **new** SSLConnectionSocketFactory(
3. sslContext,
4. NoopHostnameVerifier.INSTANCE);

        HttpClient4.4使用Mozilla基金会维护的公共后缀列表去确保SSL证书的通配符不会被多个通用顶级域名误用，HttpClient会附带一个该列表的最新的拷贝，最新的修正版在[https://publicsuffix.org/list/](https://publicsuffix.org/list/effective_tld_names.dat" \t "http://blog.csdn.net/ktlifeng/article/details/_blank)，强烈建议从源数据每天更新一次并且保持一份本地拷贝。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_278](https://code.csdn.net/snippets/1605188)[IMG_279](https://code.csdn.net/snippets/1605188/fork)

1. PublicSuffixMatcher publicSuffixMatcher = PublicSuffixMatcherLoader.load(
2. PublicSuffixMatcher.**class**.getResource("my-copy-effective\_tld\_names.dat"));
3. DefaultHostnameVerifier hostnameVerifier = **new** DefaultHostnameVerifier(publicSuffixMatcher);

你可以通过使用null匹配来关闭公共后缀列表验证

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_280](https://code.csdn.net/snippets/1605188)[IMG_281](https://code.csdn.net/snippets/1605188/fork)

1. DefaultHostnameVerifier hostnameVerifier = **new** DefaultHostnameVerifier(**null**);

### 5.2.8 HttpClient 代理配置

        尽管HttpClient已经有复杂路由计算和代理量，他也只支持简单的直连和单跳代理连接。

让HttpClient通过代理连接目标主机最简单的方式是配置默认的代理参数：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_282](https://code.csdn.net/snippets/1605188)[IMG_283](https://code.csdn.net/snippets/1605188/fork)

1. HttpHost proxy = **new** HttpHost("someproxy", 8080);
2. DefaultProxyRoutePlanner routePlanner = **new** DefaultProxyRoutePlanner(proxy);
3. CloseableHttpClient httpclient = HttpClients.custom()
4. .setRoutePlanner(routePlanner)
5. .build();

你也可以让HttpClient使用标准JRE代理选择器来获取代理配置：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_284](https://code.csdn.net/snippets/1605188)[IMG_285](https://code.csdn.net/snippets/1605188/fork)

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

或者，你可以提供定制的RoutePlanner实现，来实现HTTP路由的复杂计算：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50849075" \o "copy) [IMG_286](https://code.csdn.net/snippets/1605188)[IMG_287](https://code.csdn.net/snippets/1605188/fork)

1. HttpRoutePlanner routePlanner = **new** HttpRoutePlanner() {
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. }
11. };
12. CloseableHttpClient httpclient = HttpClients.custom()
13. .setRoutePlanner(routePlanner)
14. .build();
15. }
16. }

## 5.3 状态管理

 最初的HTTP呗设计成为无状态的，面向request/response的协议，并没有为跨越多个逻辑依赖的request/response交换的有状态的session做准备，随着HTTP协议被越来越广泛的使用，许多系统已经使用它来为超出预期的应用服务，如电子商务应用的传输，于是，状态管理逐渐变得有必要。

Netscape公司是当时web客户端和服务端软件开发的领导者，基于私有规范在其产品中实现了HTTP状态管理，随后，Netscape尝试通过发布规范草案来标准化该机制，这些努力有助于通过RFC形成规范文档，然而在大多数的应用里，状态管理依旧是很大程度上基于Netscape的草案并且跟官方规范不兼容，web开发者都可以感觉到强行保留与这些应用的兼容性导致了标准规范碎片化。

### 5.3.1 HTTP cookies

        HTTP cookie是HTTP客户端和服务端之间交互的一小段状态信息或者标记用于维持一个会话信息，Netscape工程师用一个魔法饼干来形容它，而这一名称就被保留了下来。

        HttpClient使用Cookie接口来表示抽象的cookie标记，HTTP cookie最简单的形式就是一个键值对，通常HTTP cookie包含一系列属性比如合法域名，cookie可以被接收的URL的子集，cookie保存的最大时间。

        SetCookie接口代表Set-Cookie response头消息，该消息从服务端发送给客户端用于保持一个会话状态。

        ClientCookie接口继承自Cookie接口，其可以实现额外的客户端特定的功能，比如精确的检索源服务器指定的源cookie属性，这对于生成Cookie头信息非常重要，因为一些cookie规范要求特定的属性仅当他们在Set-Cookie头消息指定时才能被Cookie头包含进去。

 下面是一个创建客户端Cookie对象的例子：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50858348" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50858348" \o "copy) [IMG_256](https://code.csdn.net/snippets/1606563)[IMG_257](https://code.csdn.net/snippets/1606563/fork)

1. BasicClientCookie cookie = **new** BasicClientCookie("name", "value");
2. // Set effective domain and path attributes
3. cookie.setDomain(".mycompany.com");
4. cookie.setPath("/");
5. // Set attributes exactly as sent by the server
6. cookie.setAttribute(ClientCookie.PATH\_ATTR, "/");
7. cookie.setAttribute(ClientCookie.DOMAIN\_ATTR, ".mycompany.com");

### 5.3.2 Cookie规范

        CookieSpec接口代表了cookie管理的规范，cookie管理规范包含如下内容：

        解析Set-Cookie头部的规则

        验证已解析cookie的规则

        为已给定主机，端口和源路径格式化Cookie头部

        HttpClient附带多种CookieSpec实现：

**Standard strict（严格）：**状态管理策略行为完全符合RFC6265第四章的行为定义。

**Standard（标准）：**状态管理策略较为符合RFC6265第四章定义的行为，以期望在不是完全遵守该行为之间的服务器之间进行交互。

**Netscape 草案（已过时）：**该策略遵守Netscape公司公布最初的规范草案，除非确实需要与旧代码兼容，否则尽量避免使用它。

**RFC 2965（已过时）：** 状态管理策略符合过时的RFC2965定义的状态管理规范。请不要在新应用中使用。

**RFC2109（已过时）：**状态管理策略符合过时的RFC2109定义的状态管理规范，请不要在新应用中使用。

**浏览器兼容性Browser compatibility（已过时）：**该策略尝试尽量去模拟老旧的浏览器版本如微软IE和Mozilla FireFox，请不要在新应用中使用。

**Default：**默认cookie策略是一种综合性的策略，其基于HTTP response返回的cookie属性如version信息，过期信息，与RFC2965,RFC2109或者Netscape草案兼容，该策略将会在下一个HttpClient小版本（基于RFC6265）中废弃。

**Ignore cookies：**所有的cookie都被忽略

强烈建议在新应用中使用Standard或者Standard strict策略，过时规范应该仅仅是在与旧系统兼容时使用。下一个HttpClient版本将会停止对过时规范的支持。

### 5.3.3 选择Cookie策略

Cookie策略能够通过HTTP客户端设置，并且可以在HTTP request时被覆盖掉。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50858348" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50858348" \o "copy) [IMG_258](https://code.csdn.net/snippets/1606563)[IMG_259](https://code.csdn.net/snippets/1606563/fork)

1. RequestConfig globalConfig = RequestConfig.custom()
2. .setCookieSpec(CookieSpecs.DEFAULT)
3. .build();
4. CloseableHttpClient httpclient = HttpClients.custom()
5. .setDefaultRequestConfig(globalConfig)
6. .build();
7. RequestConfig localConfig = RequestConfig.copy(globalConfig)
8. .setCookieSpec(CookieSpecs.STANDARD\_STRICT)
9. .build();
10. HttpGet httpGet = **new** HttpGet("/");
11. httpGet.setConfig(localConfig);

### 5.3.4 定制cookie策略

        为了实现定制cookie策略，你应该创建一个自定义的CookieSpec接口的实现，创建一个CookieSpecProvider的实现类，然后用该实现类去创建和初始化自定义规范的实例，然后使用HttpClient进行注册，一旦自定义规范被注册，它就会如同标准cookie规范一样被触发。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50858348" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50858348" \o "copy) [IMG_260](https://code.csdn.net/snippets/1606563)[IMG_261](https://code.csdn.net/snippets/1606563/fork)

1. PublicSuffixMatcher publicSuffixMatcher = PublicSuffixMatcherLoader.getDefault();
3. Registry<CookieSpecProvider> r = RegistryBuilder.<CookieSpecProvider>create()
4. .register(CookieSpecs.DEFAULT,
5. **new** DefaultCookieSpecProvider(publicSuffixMatcher))
6. .register(CookieSpecs.STANDARD,
7. **new** RFC6265CookieSpecProvider(publicSuffixMatcher))
8. .register("easy", **new** EasySpecProvider())
9. .build();
11. RequestConfig requestConfig = RequestConfig.custom()
12. .setCookieSpec("easy")
13. .build();
15. CloseableHttpClient httpclient = HttpClients.custom()
16. .setDefaultCookieSpecRegistry(r)
17. .setDefaultRequestConfig(requestConfig)
18. .build();

### 5.3.5 Cookie持久化

        HttpClient可以同任何实现了CookieStore接口的实际Cookie存储器协同工作，模拟的CookieStore实现叫做BasicCookieStore，是一个基于**[Java](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "http://blog.csdn.net/ktlifeng/article/details/_blank)**.util.ArrayList的简单实现，当容器进行垃圾回收时，如果BasicClientCookie对象被回收掉了，其存储的Cookie也会丢失，你可以提供更复杂的实现来满足自己的需求。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50858348" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50858348" \o "copy) [IMG_262](https://code.csdn.net/snippets/1606563)[IMG_263](https://code.csdn.net/snippets/1606563/fork)

1. // Create a local instance of cookie store
2. CookieStore cookieStore = **new** BasicCookieStore();
3. // Populate cookies if needed
4. BasicClientCookie cookie = **new** BasicClientCookie("name", "value");
5. cookie.setDomain(".mycompany.com");
6. cookie.setPath("/");
7. cookieStore.addCookie(cookie);
8. // Set the store
9. CloseableHttpClient httpclient = HttpClients.custom()
10. .setDefaultCookieStore(cookieStore)
11. .build();

### 5.3.6 HTTP状态管理和运行上下文

        在HTTP request执行过程中，HttpClient将如下的状态管理相关的对象添加到运行上下文中。

        Lookup 实例代表实际的cookie注册规范，该属性的值的优先级为当前上下文大于默认上下文。

        CookieSpec实例代表着实际的cookie规范。

        CookieOrigin实例代表着源服务器的详细Cookie信息。

        CookieStore实例代表着实际的Cookie存储，该属性的值的优先级为当前上下文大于默认上下文。

        当前HttpContext对象能够在请求执行之前用来定制HTTP状态管理上下文，或者在请求执行之后检查其状态，你也可以通过单独的运行上下文来实现各自的状态管理，在HTTP客户端级别，当前上下文的ookie注册规范和cookie存储优先级大于默认的上下文。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/50858348" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/50858348" \o "copy) [IMG_264](https://code.csdn.net/snippets/1606563)[IMG_265](https://code.csdn.net/snippets/1606563/fork)

1. CloseableHttpClient httpclient = <...>
3. Lookup<CookieSpecProvider> cookieSpecReg = <...>
4. CookieStore cookieStore = <...>
6. HttpClientContext context = HttpClientContext.create();
7. context.setCookieSpecRegistry(cookieSpecReg);
8. context.setCookieStore(cookieStore);
9. HttpGet httpget = **new** HttpGet("http://somehost/");
10. CloseableHttpResponse response1 = httpclient.execute(httpget, context);
11. <...>
12. // Cookie origin details
13. CookieOrigin cookieOrigin = context.getCookieOrigin();
14. // Cookie spec used
15. CookieSpec cookieSpec = context.getCookieSpec();

## 5.4 HTTP身份认证

HttpClient对HTTP标准规范中定义的认证机制和非标准的认证机制如NTLM和SPNEGO提供了全面的支持。

### 5.4.1 用户凭证

        大部分的用户认证过程都需要一组凭证用于鉴定用户的身份，用户凭证最简单的方式就是一组 用户名/密码 对，UsernamePasswordCredentials用明文形式表示一组安全主体和密码凭证，该类一般可满足HTTP标准规范中的认证机制的要求。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "copy) [IMG_256](https://code.csdn.net/snippets/1639550)[IMG_257](https://code.csdn.net/snippets/1639550/fork)

1. //安全主体user 密码凭证pwd
2. UsernamePasswordCredentials creds = **new** UsernamePasswordCredentials("user", "pwd");
3. System.out.println(creds.getUserPrincipal().getName());
4. System.out.println(creds.getPassword());

输出：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "copy) [IMG_258](https://code.csdn.net/snippets/1639550)[IMG_259](https://code.csdn.net/snippets/1639550/fork)

1. user
2. pwd

       NTCredentials是Microsoft Windows特定的实现，包含了用户名/密码 对，附加了Windows特定的属性如用户所属的域名。在Microsoft Windows网络中同一个用户可以属于不同的域，每一个域都有不同的认证机制。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "copy) [IMG_260](https://code.csdn.net/snippets/1639550)[IMG_261](https://code.csdn.net/snippets/1639550/fork)

1. //wordstation指发出请求的机器名称,一般是电脑名,domain是域的名称
2. NTCredentials creds = **new** NTCredentials("user", "pwd", "workstation", "domain");
3. System.out.println(creds.getUserPrincipal().getName());
4. System.out.println(creds.getPassword());

输出：

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "copy) [IMG_262](https://code.csdn.net/snippets/1639550)[IMG_263](https://code.csdn.net/snippets/1639550/fork)

1. DOMAIN/user
2. pwd

### 5.4.2 认证机制

       AuthScheme接口表示一个抽象的面向挑战-应答的认证机制。一个认证机制一般会期望有如下的功能：

              解析和处理目标服务器发送的挑战并且对受保护资源的请求做出应答。

              提供处理后挑战的属性：认证机制类型和其参数，比如认证机制适用的范围（可能的话）

              为所给的凭证和HTTP request生成认证字符串用于认证挑战的应答。

       请注意认证机制可能是有状态的，应该避免一系列的挑战-应答交换。

       HttpClient附带多个AuthScheme的实现：

**Basic**: 如RFC2617所定义的基本认证机制，由于凭证通过明文传递，该认证机制是不安全的，尽管其安全性比较低，但是如果与TLS/SSL加密协同工作，其认证机制仍然是可以接受的。

**Digest**: 如RFC2617所定义的Digest认证机制，Digest认证机制明显比Basic要更安全，该机制对于那些顾虑TLS/SSL加密所带来的开销的应用会是一个比较好的选择。

**NTLM**: NTLM是一个微软开发的并且针对Windows平台进行了优化的专有认证机制，NTML被认为比Digest更安全。

**SPNEGO**: SPNEGO(Simple and Protected GSSAPI Negotiation Mechanism)是一个GSSAPI伪机制，用于协商那些潜在的真正机制，SPNEGO最明显的用途是在微软的HTTP Negotiate验证扩展，可协商的子机制包含活动目录中支持的NTML和Kerberos，现在HttpClient只支持Kerberos子机制。

**Kerberos**: Kerberos认证的实现。

### 5.4.3 凭证提供器

       凭证提供器旨在维护一组用户凭证并且能够为特定的认证域生产用户凭证，认证领域包含一个主机名，一个端口名，一个域名称和一个认证机制名，当使用凭证提供器（能够提供任意的主机，任意端口，任意域名和任意机制）来替代具体属性值注册凭证时，其期望能够为某个特定认证域找到一个最贴近的匹配值（在直接匹配找不到的情况下）。

       HttpClient可以与任意实现了CredentialsProvider接口的实际凭证提供器协同工作，默认的CredentialsProvider实现叫做BasicCredentialsProvider，其是一个依赖于**[Java](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "http://blog.csdn.net/ktlifeng/article/details/_blank)**.util.HashMap的简单实现。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "copy) [IMG_264](https://code.csdn.net/snippets/1639550)[IMG_265](https://code.csdn.net/snippets/1639550/fork)

1. //凭证注册器
2. CredentialsProvider credsProvider = **new** BasicCredentialsProvider();
3. //注册1 somehost主机名  任意PORT  指定用户U1 密码P1
4. credsProvider.setCredentials(
5. **new** AuthScope("somehost", AuthScope.ANY\_PORT),
6. **new** UsernamePasswordCredentials("u1", "p1"));
7. //注册2 somehost主机名  8080端口  用户U2 密码P2
8. credsProvider.setCredentials(
9. **new** AuthScope("somehost", 8080),
10. **new** UsernamePasswordCredentials("u2", "p2"));
11. //注册3  otherhost主机名   8080端口   任意域   认证机制ntlm   用户U3  密码P3
12. credsProvider.setCredentials(
13. **new** AuthScope("otherhost", 8080, AuthScope.ANY\_REALM, "ntlm"),
14. **new** UsernamePasswordCredentials("u3", "p3"));
16. //与somehost 端口80  realm域  basic认证机制匹配的凭证
17. System.out.println(credsProvider.getCredentials(
18. **new** AuthScope("somehost", 80, "realm", "basic")));
19. //与somehost 端口8080  realm域  basic认证机制匹配的凭证
20. System.out.println(credsProvider.getCredentials(
21. **new** AuthScope("somehost", 8080, "realm", "basic")));
22. //与otherhost 端口8080  realm域  basic认证机制匹配的凭证
23. System.out.println(credsProvider.getCredentials(
24. **new** AuthScope("otherhost", 8080, "realm", "basic")));
25. //与otherhost 端口8080  realm域  basic认证机制匹配的凭证
26. System.out.println(credsProvider.getCredentials(
27. **new** AuthScope("otherhost", 8080, **null**, "ntlm")));

输出

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "copy) [IMG_266](https://code.csdn.net/snippets/1639550)[IMG_267](https://code.csdn.net/snippets/1639550/fork)

1. //与注册1最类似
2. [principal: u1]
3. //与注册2最类似
4. [principal: u2]
5. //都不类似 因为没有对应的主机名和认证机制
6. **null**
7. //与注册3最类似
8. [principal: u3]

### 5.4.4 HTTP身份认证和执行环境

       HttpClient依靠AuthState类来对认证处理状态的详细信息进行跟踪，HttpClient在HTTP请求执行过程中创建了两个AuthState实例：一个用于目标主机认证另一个用于代理认证，由于目标服务器或者代理服务器都可能要求用户身份认证，所以各自的AuthScope实例将会被认证过程中所使用的AuthScope,AuthScheme和Credentials所填充，AuthState可以用于检查请求是属于何种认证类型，无论是否找到匹配的AuthScheme实现，也不管是否有凭证提供器用于查找特定的用户凭证。

在HTTP请求执行的过程中，HttpClient添加如下的认证相关对象到执行环境中：

**Lookup**实例代表实际注册的认证机制，该属性值优先级为局部环境中的值大于默认值。

**CredentialsProvider**实例代表了实际的凭证提供器，该属性值优先级为局部环境中的值大于默认值。

**AuthState**实例代表了实际的目标认证状态，该属性值优先级为局部环境中的值大于默认值。

**AuthState**实例代表了实际的代理认证状态，该属性值优先级为局部环境中的值大于默认值。

**AuthCache**实例代表了实际的认证数据缓存，该属性值优先级为局部环境中的值大于默认值。

       局部**HttpContext**对象可以先于request执行环来定制化Http认证环境，或者在请求执行之后用于检查其状态。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "copy) [IMG_268](https://code.csdn.net/snippets/1639550)[IMG_269](https://code.csdn.net/snippets/1639550/fork)

1. CloseableHttpClient httpclient = <...>
3. CredentialsProvider credsProvider = <...>
4. Lookup<AuthSchemeProvider> authRegistry = <...>
5. AuthCache authCache = <...>
6. //HttpClient上下文
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. //代理认证状态
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.getCredentials());
19. //目标主机认证状态
20. AuthState targetAuthState = context.getTargetAuthState();
21. System.out.println("Target auth state: " + targetAuthState.getState());
22. System.out.println("Target auth scheme: " + targetAuthState.getAuthScheme());
23. System.out.println("Target auth credentials: " + targetAuthState.getCredentials());

5.4.5 认证数据缓存

       自从4.1版本开始，HttpClient会自动缓存其认证成功的主机的相关信息，请注意你必须使用相同的执行环境去执行逻辑相关的请求，以此保证缓存的认证数据能够在不同的request之间传输，当执行环境超出范围时认证数据会立即被丢弃。

### 5.4.6 先占式认证

       HttpClient没有现成的方式支持先占式认证，因为如果先占认证被误用或者使用不当会导致重大安全问题，比如明文发送用户凭证到非授权的第三方，所以，希望用户能够在其具体的应用环境中评估先占式认证的好处和其潜在的风险。

       尽管如此，你可以通过预填充认证数据缓存的方式来配置HttpClient实现先占式认证。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "copy) [IMG_270](https://code.csdn.net/snippets/1639550)[IMG_271](https://code.csdn.net/snippets/1639550/fork)

1. CloseableHttpClient httpclient = <...>
3. HttpHost targetHost = **new** HttpHost("localhost", 80, "http");
4. //基础凭证提供器,明文传输数据
5. CredentialsProvider credsProvider = **new** BasicCredentialsProvider();
6. credsProvider.setCredentials(
7. **new** AuthScope(targetHost.getHostName(), targetHost.getPort()),
8. **new** UsernamePasswordCredentials("username", "password"));
10. // 创建认证缓存
11. AuthCache authCache = **new** BasicAuthCache();
12. // 创建基础认证机制 添加到缓存
13. BasicScheme basicAuth = **new** BasicScheme();
14. authCache.put(targetHost, basicAuth);
16. // 将认证缓存添加到执行环境中  即预填充
17. HttpClientContext context = HttpClientContext.create();
18. context.setCredentialsProvider(credsProvider);
19. context.setAuthCache(authCache);
21. HttpGet httpget = **new** HttpGet("/");
22. **for** (**int** i = 0; i < 3; i++) {
23. CloseableHttpResponse response = httpclient.execute(
24. targetHost, httpget, context);
25. **try** {
26. HttpEntity entity = response.getEntity();
28. } **finally** {
29. response.close();
30. }
31. }

### 5.4.7 NTLM认证

       自从4.1版本后，HttpClient对NTMLv1,NTLMv2,和NTML2绘画认证提供了现成的支持，你可以继续使用外部的NTLM引擎比如JCIFS库（Samba工程作为他们的windows协同编程套件研发）。

**4.7.1 NTLM连接持久化**

       NTLM认证机制在计算开销和性能影响上明显比Basic和Digest机制更昂贵，这很可能是微软选择将NTLM认证机制做成有状态化的主要原因之一，换句话说，一旦已经认证过了，用户的身份就跟整个生命周期的连接关联起来了，NTML连接这种有状态的性质使得连接持久化更加复杂，一个明显的情形是持久化的NTLM连接不会被不同身份的另一个用户重用。而HttpClient附带的标准连接管理器完全有能力管理有状态的连接，但是，有一点非常重要的就是逻辑相关request要使用同一个执行上下文和会话，以便于他们共享同一个当前用户。否则，对于每个请求NTML保护资源的request，HttpClient最终会为其创建一个新的HTTP连接。

       由于NTLM连接是有状态的，其一般建议通过一个相对便宜的方式来触发NTLM认证，如GET或者HEAD，然后重用相同的连接来执行更加昂贵的方法，特别是那些封装了请求实体的如POST或者PUT。

**[java]** [view plain](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "copy) [IMG_272](https://code.csdn.net/snippets/1639550)[IMG_273](https://code.csdn.net/snippets/1639550/fork)

1. CloseableHttpClient httpclient = <...>
3. CredentialsProvider credsProvider = **new** BasicCredentialsProvider();
4. credsProvider.setCredentials(AuthScope.ANY,
5. **new** NTCredentials("user", "pwd", "myworkstation", "microsoft.com"));
7. HttpHost target = **new** HttpHost("www.microsoft.com", 80, "http");
9. //确保相同逻辑的request使用同一个context
10. HttpClientContext context = HttpClientContext.create();
11. context.setCredentialsProvider(credsProvider);
13. //使用GET来触发NTLM认证
14. HttpGet httpget = **new** HttpGet("/ntlm-protected/info");
15. CloseableHttpResponse response1 = httpclient.execute(target, httpget, context);
16. **try** {
17. HttpEntity entity1 = response1.getEntity();
18. } **finally** {
19. response1.close();
20. }
22. //使用POST来提交请求
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, context);
26. **try** {
27. HttpEntity entity2 = response2.getEntity();
28. } **finally** {
29. response2.close();
30. }

### 5.4.8 SPNEGO/Kerberos 认证

       SPNEGO(Simple and Protected GSSAPI Negotiation Mechanism) 设计成在不知道服务可以提供或者使用什么时为其提供认证，其最普遍的用法是扩展Kerberos认证，它可以包装其他机制，但是在HttpClient中，SPNEGO的当前版本唯一关心的只有Kerberos。

**4.8.1 HttpClient中的SPNEGO支持**

       SPNEGO认证机制能够工作在Sun Java的1.5版本及以上，但是建议使用1.6及以上的版本，对SPNEGO认证的支持会更加完整一些。

       SUN JRE提供了支持类几乎能够处理所有的Kerberos和SPNEGO的token，这意味着为了使用GSS类你需要做很多的设置，SPNegoScheme是一个简单的用于处理token编组和正确读写headers的类。

       开始的最好方式是用KerberosHttpClient.java类，尝试编写一个示例并跑起来，中途可能会发生很多异常情况，但是幸运的话还是能够正常工作的，为了调试你应该打印出足够多的调试信息。

       在Windows里面，它应该默认使用已存在的登录凭证，该方式可以通过使用'kinit'重写，如$JAVA\_HOME\bin\kinit testuser@AD.EXAMPLE**[.NET](http://lib.csdn.net/base/dotnet" \o ".NET知识库" \t "http://blog.csdn.net/ktlifeng/article/details/_blank)** ,kinit对于调试异常和进行测试非常有用，你可以通过删除kinit的缓存文件来回滚到windows的Kerberos。

       确信在krb5.conf文件中列出了domain\_realms，这通常是发生问题的主要原因之一。

**4.8.2 GSS/[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "http://blog.csdn.net/ktlifeng/article/details/_blank)Kerberos配置**

       该文档假设你使用windows，但是许多信息在Unix会更加适用。

       org.ietf.jgss类有许多的配置参数，主要都在krb5.conf/krb5.ini文件，更多的信息可以查看http://web.mit.edu/kerberos/krb5-1.4/krb5-1.4.1/doc/krb5-admin/krb5.conf.html

**4.8.3 log.conf文件**

       以下是在windows中配置IIS和JBoss Negotiation模块的基本配置

       系统属性 java.security.auth.login.config可以通过在login.conf文件中指明来使用

       login.conf内容如下所示

**[html]** [view plain](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "copy) [IMG_274](https://code.csdn.net/snippets/1639550)[IMG_275](https://code.csdn.net/snippets/1639550/fork)

1. com.sun.security.jgss.login {
2. com.sun.security.auth.module.Krb5LoginModule required client=TRUE useTicketCache=true;
3. };
5. com.sun.security.jgss.initiate {
6. com.sun.security.auth.module.Krb5LoginModule required client=TRUE useTicketCache=true;
7. };
9. com.sun.security.jgss.accept {
10. com.sun.security.auth.module.Krb5LoginModule required client=TRUE useTicketCache=true;
11. };

**4.8.4 krb5.conf/krb5.ini文件**

       如果没有指定的话，系统默认使用该文件，你可以设置krb5.conf文件中的java.security.krb5.conf系统属性来重写该配置。

       krb5.conf内容看起来如下所示：

**[html]** [view plain](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "copy) [IMG_276](https://code.csdn.net/snippets/1639550)[IMG_277](https://code.csdn.net/snippets/1639550/fork)

1. [libdefaults]
2. default\_realm = AD.EXAMPLE.NET
3. udp\_preference\_limit = 1
4. [realms]
5. AD.EXAMPLE.NET = {
6. kdc = KDC.AD.EXAMPLE.NET
7. }
8. [domain\_realms]
9. .ad.example.net=AD.EXAMPLE.NET
10. ad.example.net=AD.EXAMPLE.NET

**4.8.5 Windows特定配置**

       要允许Windows使用当前用户的身份票据，必须设置系统属性javax.security.auth.useSubjectCredsOnly为false并且Windows注册表项"allowtgtsessionkey"应该被正确的添加和设置，以允许session的key能够被发送到Kerberos Ticket-Granting Tikect（Kerberos授权票据）

       在Windows Server 2003和Windows 2000 SP4,如下是注册表设置：

**[html]** [view plain](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "copy) [IMG_278](https://code.csdn.net/snippets/1639550)[IMG_279](https://code.csdn.net/snippets/1639550/fork)

1. HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Control\Lsa\Kerberos\Parameters
2. Value Name: allowtgtsessionkey
3. Value Type: REG\_DWORD
4. Value: 0x01

       以下是Windows XP SP2注册表设置

**[html]** [view plain](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "view plain) [copy](http://blog.csdn.net/ktlifeng/article/details/51099370" \o "copy) [IMG_280](https://code.csdn.net/snippets/1639550)[IMG_281](https://code.csdn.net/snippets/1639550/fork)

1. HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Control\Lsa\Kerberos\
2. Value Name: allowtgtsessionkey
3. Value Type: REG\_DWORD
4. Value: 0x01