**前言：**

目前HTTP（超文本传输协议）已然成为了互联网中重要的协议。在WEB服务、互联网应用和网络计算的增长继续扩大了HTTP协议的作用，超越了用户驱动的Web浏览器，同时增加了需要HTTP支持的应用程序的数量。

尽管java.net包提供了可以通过HTTP协议访问互联网资源的基础功能，但是它并没有许多应用所需的灵活功能全面的功能。HttpClient寻求通过提供一个高效的，最新的和功能丰富的软件包来实现客户端最新的HTTP标准和建议来填补这个空白。

在具有极大的扩展性的同时也基于HTTP协议提供了很多健壮性的支持，HttpClient可以构建HTTP感知客户端应用程序（如Web浏览器，Web服务客户端或利用或扩展用于分布式通信的HTTP协议的系统）。

**HttpClient scope**

* 基于HttpCore的客户端HTTP传输库
* 基于经典的I/O阻塞模型
* Content agnostic（我个人认为内容不可知为，不是想浏览器一样显示的）

**What HttpClient is NOT**

* HttpClien并不是浏览器.它只是一个HTTP的客户端 . HttpClient的作用只是传输和接收HTTP 信息. HttpClient不会去尝试处理HTTP协议中的内容, 或者执行在 HTML页面中的JavaScript脚本, 也不会尝试猜测传输内容的类型（Content-Type）,如果没有显示的设置 或者重新格式化request 和从定向 URIs,或者一些与HTTP传输无关的功能 。

**1.1. Request 的执行**

在HttpClient中最常用的功能就是执行HTTP的方法：GET/POST/PUT/DELETE/TRACE/OPTIONS/HEAD.HttpClient在执行一个HTTP方法时会涉及HTTP 请求/ HTTP 应答的数据交换. 使用者通常期望HttpClient可以提供一个传送request对象到目标服务中，然后返回一个与之对应的response对象, 或者在没有成功执行时抛出异常。

显然,HttpClient API的主要实体重点就是通过HttpClient接口定义上面以上描述的需求。

下面就是一个request执行过程的简单例子：

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. //1.将其想象成建一个浏览器的过程，HttpClients我个人感觉可以类比Collection和Collections的关系，提供HTTPClient的工具
2. CloseableHttpClient httpclient=HttpClients.createDefault();
3. //2.可以想象为用什么方法去访问服务，就像表单提交时候选择Get还是Post
4. HttpGet  httpget=**new** HttpGet("http://www.baidu.com");
5. //3.可以想象为点击鼠标的过程，或者是提交表单的过程。有返回值。。。。。
6. CloseableHttpResponse response=httpclient.execute(httpget);
7. **try** {
8. //业务处理层的东西
9. }**finally**{
10. response.close();
11. }

**1.1.1. HTTP request**

在HTTP协议中定义的请求报文的第一行必须包含：请求方法, 请求URI ，HTTP协议版本.

HttpClient 支持所有定义在HTTP/1.1中的请求方法的拆箱工作: GET, HEAD, POST, PUT,DELETE,TRACE OPTIONS. 分别对应的类是:HttpGet,HttpHead, HttpPost,HttpPut, HttpDelete,HttpTrace,  HttpOptions.

URI是一个统一资源定位符用来标识想要从哪获取资源. HTTP request URIs包含：protocol scheme（http://,https://）,host name ,port, resource path, 可选的query,可选的fragment.(注：不翻译成中文是因为在使用工具的时候，人家老外确实就是使用的这些英文来标识类或者字段、方法的)，例如：

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

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

HttpClient 提供了 URIBuilder工具类来简化request URIs的创建和修改.

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. //uri就是JDK中默认的java.net.URI。可以想象为在浏览器中输入网址的过程
2. java.net.URI uri=**new** URIBuilder().setScheme("http")
3. .setHost("www.baidu.com")
4. .setPath("/search")
5. .setParameter("q", "httpclient")
6. .setParameter("btnG", "Google Search")
7. .setParameter("aq", "f")
8. .setParameter("oq", "")
9. .build();
10. //可以想象为用什么方法去访问服务，就像表单提交时候选择Get还是Post
11. HttpGet  httpget=**new** HttpGet(uri);
12. //在get请求实体中同样可以拆箱获取uri
13. httpget.getURI();

**1.1.2. HTTP response**

HTTP response 是在服务器在已经接收和处理请求消息之后返回给客户端的信息. 消息中一定包含了：protocol version ，数值型的status code，状态解释短语textual phrase.

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. //在服务器中
2. sp;HttpResponse response=**new** BasicHttpResponse(org.apache.http.HttpVersion.HTTP\_1\_1,org.apache.http.HttpStatus.SC\_OK, "OK");
3. //同样可以进行拆箱的工作获取相关参数
4. System.out.println(response.getProtocolVersion());
5. //StatusLine 其实就是响应中的第一行，包含协议版本号，状态，短语
6. System.out.println(response.getStatusLine().toString());
7. System.out.println(response.getStatusLine().getStatusCode());
8. System.out.println(response.getStatusLine().getReasonPhrase());

**1.1.3. HTTP报文 headers**

一个HTTP报文可能包含很多个header来表述报文的属性例如： content length, content type等等. HttpClient提供了对http报文head进行查询、添加、移除、枚举的方法。

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. //在服务器中
2. HttpResponse response=**new** BasicHttpResponse(org.apache.http.HttpVersion.HTTP\_1\_1,org.apache.http.HttpStatus.SC\_OK, "OK");
3. response.addHeader("Set-Cookie","c1=a; path=/; domain=localhost");
4. response.addHeader("Set-Cookie", "c2=b; path=\"/\", c3=c; domain=\"localhost\"");
5. //同样可以进行拆箱的工作，获取Header
6. Header h1 = response.getFirstHeader("Set-Cookie");
7. System.out.println(h1);
8. Header h2 = response.getLastHeader("Set-Cookie");
9. System.out.println(h2);
10. Header[] hs = response.getHeaders("Set-Cookie");
11. System.out.println(hs.length);

最高效的获取所有header的方法是使用 HeaderIterator接口.

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. //相当于只是取到了header
2. HeaderIterator it = response.headerIterator("Set-Cookie");
3. **while** (it.hasNext()) {
4. System.out.println(it.next());
5. }

当然也提供了非常方便的解析header elements部分报文的方法.

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. HeaderElementIterator hei=**new** BasicHeaderElementIterator(response.headerIterator("Set-Cookie"));
2. **while** (it.hasNext()) {
3. HeaderElement elem = hei.nextElement();
4. System.out.println(elem.getName() + " = " + elem.getValue());
5. //当然可以以键值对的形式读取
6. NameValuePair[] params = elem.getParameters();
7. **for** (**int** i = 0; i < params.length; i++) {
8. System.out.println(" " + params[i]);
9. }
10. }

**1.1.4. HTTP entity**

在HTTP request 和response报文中可以传输一个content entity. Entities是optional（比如：HttpHead方法就不会返回entity）. Requests that use entities are referred to as entity enclosing requests（使用实体的请求被称为实体封装请求）.在HTTP定义了两种 entity enclosing 请求方法: POST 和 PUT. 响应通常期望包含内容实体（content entity）.此规则有例外情况，例如HEAD 方法和204 No Content,304 Not Modified, 205 Reset Content 响应.

HttpClient 依据content的来源将entity分为三种:（官网中的分类确实让人不知所云。。）

* **streamed:** 内容是从流中接收到的，或者是即时生成的（generated on the fly）。特别地，该类别包括从HTTP响应接收到的实体. Streamed entities 通常不可重复
* **self-contained:** 内容在内存中或通过独立于连接或其他实体的方式获取。Self-contained entities 通常是可重复的. 这是HTTP requests最常用的entities
* **wrapping:** 通常 content 来自于另一个entity.

当从HTTP响应（streaming out content）流出内容时，此区别对于连接管理（connection management）很重要.使用HttpClient仅当sent的时候request entities才会由应用创建，streamed 和 self-contained 的差异不是很重要.在这种情况下，建议将不可重复的实体视为streamed，将可重复的实体视为self-contained。

**1.1.4.1. Repeatable entities**

一个entity可以重复也就意味着它的content可以被多次读取. self contained entities (ByteArrayEntity、StringEntity)

**1.1.4.2. HTTP entities的使用**

因为一个entity可以使用二进制和字符两种形式表示, 所以entity支持字符encodings (用以支持latter, ie. character content).

当执行带有enclosed content的request请求（post、put）或者当请求成功返回响应体时entity会被创建.

从 entity读取content, 既可以使用 HttpEntity#getContent()方法, 将返回java.io.InputStream, 也可以向HttpEntity＃writeTo（OutputStream）方法提供输出流，一旦所有内容都已写入给定流，该方法将返回。

entity的 HttpEntity#getContentType()和 HttpEntity#getContentLength()方法对应的参数是header中：Content-Type和Content-Length headers (如果可访问的话).因为在HttpEntity#getContentEncoding()可以获取Content-Type header 含mime-types ：text/plain 、 text/html. 如果header不可访问则Content-Length返回-1 ，Content-Type返回NULL .如果Content-Type header 可以访问则会返回Header对象.

当为outgoing message外发消息创建实体时，该元数据必须由实体的创建者提供。

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. StringEntity entity=**new** StringEntity("important message",ContentType.create("text/plain","utf-8"));
3. //Entity中可以包含的信息Content-Type
4. entity.getContentType();
5. //Content-Length
6. entity.getContentLength();
7. //支持的编码：Content-Encoding;Content-Type: text/plain; charset=utf-8
8. entity.getContentEncoding();
9. //显示entity的content ：important message
10. EntityUtils.toString(entity);
11. //将entity转化为Byte字节数组
12. EntityUtils.toByteArray(entity);

**1.1.5. 确保释放低级别资源**

为了保证能正确释放系统资源，必须关闭与实体或响应本身相关联的内容流。

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. //将其想象成建一个浏览器的过程，HttpClients我个人感觉可以类比Collection和Collections的关系，提供HTTPClient的工具
2. CloseableHttpClient httpclient=HttpClients.createDefault();
3. //可以想象为用什么方法去访问服务，就像表单提交时候选择Get还是Post
4. HttpGet  httpget=**new** HttpGet("www.baidu.com");
5. //可以想象为点击鼠标的过程，或者是提交表单的过程。有返回值。。。。。
6. CloseableHttpResponse response=httpclient.execute(httpget);
7. **try** {
8. //业务处理层的东西
9. HttpEntity entity=response.getEntity();
10. **if**(entity!=**null**) {
11. InputStream is=entity.getContent();
12. **try** {
13. //dosomething
15. }**finally** {
16. //关闭entity的输入流
17. is.close();
18. }
19. }
20. }**finally**{
21. //关闭响应的流
22. response.close();
23. }
24. }

关闭entity的content stream和关闭response的区别在于前者将尝试通过消费（取走）实体内容来保持底层连接，而后者立即关闭并丢弃连接。

请注意，一旦实体完全写出，还需要使用HttpEntity＃writeTo（OutputStream）方法来确保正确释放系统资源。如果此方法通过调用HttpEntity＃getContent（）获取java.io.InputStream的实例，那么也希望在finally子句中关闭流

当使用流实体时，可以使用EntityUtils＃consume（HttpEntity）方法来确保实体内容已被完全消费（其实就是我们所说的取走了），底层流已经被关闭 。

然而，可能会有情况，当只需要查询整个响应内容的一小部分，并且取走剩余内容并使连接可重用的性能损失太高，在这种情况下，可以通过关闭终止内容流响应。

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. **try** {
2. //业务处理层的东西
3. HttpEntity entity=response.getEntity();
4. **if**(entity!=**null**) {
5. InputStream is=entity.getContent();
6. **try** {
7. //一字节一字节的读取
8. **int** byteOne = is.read();
9. **int** byteTwo = is.read();
10. //60,30
11. System.out.println(byteOne);
12. System.out.println(byteTwo);
13. // Do not need the rest
14. }**finally** {
15. //关闭entity的输入流
16. is.close();
17. }

连接不会重复使用，但由其持有的所有级别资源将被正确地分配。

**1.1.6. Consuming entity content（我个人感觉为是使用entity的内容）**

使用 HttpEntity#getContent() 和 HttpEntity#writeTo(OutputStream) 方法都能获取到entity的内容. HttpClient  也提供了EntityUtils工具类, 提供了许多static methods可以很容易读取entity中的内容信息.可以直接使用java.io.InputStream方法，而不是直接读取java.io.InputStream，可以通过使用该类的方法来查询字符串/字节数组中的整个内容.但是，强烈建议不要使用EntityUtils，除非响应实体来自受信任的HTTP服务器，并且已知其长度有限。

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. //业务处理层的东西
2. HttpEntity entity=response.getEntity();
3. **if**(entity!=**null**) {
4. **long** len=entity.getContentLength();
5. //在entity内容非空且<2k的时候可以使用EntityUtils,否则使用流
6. **if**(len!=-1&&len<2048) {
7. System.out.println(EntityUtils.toString(entity));
8. }**else** {
9. // Stream content out
10. }
11. }

在某些情况下，可能需要多次读取entity内容。在这种情况下，entity内容必须以某种方式缓存到内存或者磁盘上。最简单的方法是通过使用BufferedHttpEntity类包装原始实体。这将导致将原来的entity内容读入内存缓冲区。其他的wrapper也是同样可以获取原始内容。

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. //业务处理层的东西
2. HttpEntity entity=response.getEntity();
3. **if**(entity!=**null**) {
4. //会将entity的内容缓存到内存中
5. entity = **new** BufferedHttpEntity(entity);
6. }

**1.1.7. Producing entity content（生成 entity）**

HttpClient 提供了许多实现类，通过HTTP connections可以高效输出entity内容 . 这些类的实例可以与实体封装请求相关联，例如POST和PUT，以便将实体内容包含到传出的HTTP请求中。HttpClient为大多数常见的数据容器提供了几个类，例如字符串，字节数组，输入流和文件:StringEntity, ByteArrayEntity,InputStreamEntity, FileEntity.

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. //将其想象成建一个浏览器的过程，HttpClients我个人感觉可以类比Collection和Collections的关系，提供HTTPClient的工具
2. CloseableHttpClient httpclient=HttpClients.createDefault();
3. //FileEntity
4. FileEntity entity=**new** FileEntity(**new** File(""),ContentType.create("text/plain","utf-8"));
5. //post和put方法相关联的实体
6. HttpPost post=**new** HttpPost("http://www.baidu.com");
7. post.setEntity(entity);
8. httpclient.execute(post);

需要注意的是 InputStreamEntity 是不可repeatable,因为它只能从底层数据流读取一次.通常建议实现自定义HttpEntity类，它是自包含的，而不是使用通用的InputStreamEntity。FileEntity可以是一个很好的起点。

**1.1.7.1. HTML forms**

许多应用需要模拟表单提交的过程, 例如,  HttpClient 为了这种需求，记录提交的数据提供了 UrlEncodedFormEntity.

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. //将其想象成建一个浏览器的过程，HttpClients我个人感觉可以类比Collection和Collections的关系，提供HTTPClient的工具
2. CloseableHttpClient httpclient=HttpClients.createDefault();
3. List<NameValuePair> formparams=**new** ArrayList<>();
4. formparams.add(**new** BasicNameValuePair("param1", "value1"));
5. formparams.add(**new** BasicNameValuePair("param2", "value2"));
6. UrlEncodedFormEntity entity=**new** UrlEncodedFormEntity(formparams);
7. HttpPost httpPost=**new** HttpPost("http://www.baidu.com");
8. httpPost.setEntity(entity);
9. httpclient.execute(httpPost);

UrlEncodedFormEntity实例将使用所谓的URL编码来对参数进行编码并产生以下内容：param1=value1&param2=value2

**1.1.7.2. Content chunking（内容分块）**

一般建议让HttpClient根据正在传输的HTTP消息的属性选择最合适的传输编码。但是，通过将HttpEntity＃setChunked（）设置为true，通知HttpClient可以优先使用块编码。请注意，HttpClient只会使用此标志作为提示。当使用不支持块编码的HTTP协议版本（如HTTP / 1.0）时，此值将被忽略.

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. StringEntity entity = **new** StringEntity("important message", ContentType.create("plain/text", Consts.UTF\_8));
2. entity.setChunked(**true**);

**1.1.8. Response handlers（响应处理器）**

处理响应的最简单和最方便的方法是使用ResponseHandler接口，其中包括handleResponse（HttpResponse response）方法。这种方法可以完全避免用户不必担心连接管理。当使用ResponseHandler时，HttpClient将自动处理确保将连接释放回连接管理器，无论请求执行是成功还是导致异常。

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. //将其想象成建一个浏览器的过程，HttpClients我个人感觉可以类比Collection和Collections的关系，提供HTTPClient的工具
2. CloseableHttpClient httpclient=HttpClients.createDefault();
3. HttpPost httpPost=**new** HttpPost("http://www.baidu.com");
4. ResponseHandler<JsonObject> rh = **new** ResponseHandler<JsonObject>() {
6. @Override
7. **public** JsonObject handleResponse(
8. **final** HttpResponse response) **throws** IOException {
9. //对HttpResponse 进行处理
10. StatusLine statusLine = response.getStatusLine();
11. HttpEntity entity = response.getEntity();
12. **if** (statusLine.getStatusCode() >= 300) {
13. **throw** **new** HttpResponseException(
14. statusLine.getStatusCode(),
15. statusLine.getReasonPhrase());
16. }
17. **if** (entity == **null**) {
18. **throw** **new** ClientProtocolException("Response contains no content");
19. }
20. Gson gson = **new** GsonBuilder().create();//创建Google的GSON对象
21. ContentType contentType = ContentType.getOrDefault(entity);//获取Content-type中的编码
22. Charset charset = contentType.getCharset();
23. Reader reader = **new** InputStreamReader(entity.getContent(), charset);//获取字符流
24. **return** gson.fromJson(reader, JsonObject.**class**);//将其格式化为对象返回
25. }
26. };
28. JsonObject json=httpclient.execute(httpPost, rh);

**1.2. HttpClient interface（ConnectionKeepAliveStrategy）**

HttpClient接口代表HTTP请求执行最基本的流程契约。它对请求执行过程只定义了过程方法，并将连接管理，状态管理，身份验证和重定向处理的具体细节留给各个实现类。这应该使得更容易使用附加功能（如响应内容缓存）来装饰接口。

一般来说，HttpClient实现作为一个特殊用途处理程序或策略接口实现的外观，负责处理HTTP协议的特定方面，例如重定向或身份验证处理或决定连接持久性并保持活动持续时间。这使得用户能够选择性地将这些方面的默认实现替换为具有特定应用程序的那些方面。ConnectionKeepAliveStrategy：

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. //设置连接KeepAlive策略
2. ConnectionKeepAliveStrategy keepAliveStrategy=**new** DefaultConnectionKeepAliveStrategy() {
3. @Override
4. ublic **long** getKeepAliveDuration(HttpResponse response, HttpContext context) {
5. // TODO Auto-generated method stub
6. **long** keeAlive=**super**.getKeepAliveDuration(response, context);
7. **if**(keeAlive==-1) {
8. // 如果服务器没有显示的设置 keep-alive的时间，就设置为保持连接5秒
9. keeAlive=5000;
10. }
11. **return** keeAlive;
13. };
14. CloseableHttpClient httpclient=HttpClients.custom().setKeepAliveStrategy(keepAliveStrategy).build();

**1.2.1. HttpClient 是线程安全的**

HttpClient实现是线程安全的。建议将此类的同一个实例重用于多个请求执行。

**1.2.2. HttpClient resource deallocation**

当CloseableHttpClient不再需要，与之关联的连接管理器必须通过调用CloseableHttpClient＃close（）方法来关闭.

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

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

**1.3. HTTP 的执行上下文**

最初HTTP被设计为无状态的，响应请求的协议。然而，现实世界的应用程序通常需要通过几个逻辑相关的请求 - 响应交换来保持状态信息。为了使应用程序能够保持处理状态HttpClient允许在特定执行上下文（称为HTTP上下文）中执行HTTP请求。如果在连续请求之间重复使用相同的上下文，则多个逻辑相关请求可以使用逻辑session。HTTP上下文功能类似于java.util.Map <String，Object>。 它只是一个任意命名值的集合。应用程序可以在请求执行之前填充上下文属性，或在执行完成后检查上下文。

HttpContext可以包含任意对象，因此可能不安全地在多个线程之间共享。建议每个执行线程都维护自己的上下文。

在HTTP请求执行过程中，HttpClient将以下属性添加到执行上下文中：

* HttpConnection：表示与目标服务器的实际连接的实例
* HttpHost：表示连接目标的实例.
* HttpRoute ：表示完整连接路由的实例
* HttpRequest：表示实际的HTTP请求实例。执行上下文中的最终HttpRequest对象总是表示消息的状态与发送到目标服务器的状态完全相同。 默认HTTP / 1.0和HTTP / 1.1使用相对请求URI。但是，如果请求是通过代理在非隧道模式（non-tunneling mode）下发送的，则URI将是绝对地址。
* HttpResponse ：表示实际的HTTP响应.
* java.lang.Boolean ：表示表示实际请求是否被完全发送到连接目标的标志的对象
* RequestConfig ：代表实际的请求配置.
* java.util.List<URI> ：表示在请求执行过程中接收的所有重定向位置的集合.

可以使用HttpClientContext适配器类来简化与上下文状态的交互。

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

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();

应该使用相同的HttpContext实例来执行表示逻辑相关会话的多个请求序列，以确保在请求之间自动传播会话上下文和状态信息。

在以下示例中，由初始请求设置的请求配置将保留在执行上下文中，并被传播到共享相同上下文的连续请求。

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. HttpContext context=<...>;
2. //设置连接KeepAlive策略
3. ConnectionKeepAliveStrategy keepAliveStrategy=**new** DefaultConnectionKeepAliveStrategy() {
4. @Override
5. **public** **long** getKeepAliveDuration(HttpResponse response, HttpContext context) {
6. // TODO Auto-generated method stub
7. **long** keeAlive=**super**.getKeepAliveDuration(response, context);
8. **if**(keeAlive==-1) {
9. // 如果服务器没有显示的设置 keep-alive的时间，就设置为保持连接5秒
10. keeAlive=5000;
11. }
12. **return** keeAlive;
13. }
14. };
15. CloseableHttpClient httpclient=HttpClients.custom().setKeepAliveStrategy(keepAliveStrategy).build();
16. //设置请求配置
17. RequestConfig requestConfig=RequestConfig.custom()
18. .setSocketTimeout(1000)
19. .setConnectTimeout(1000).build();
21. HttpPost httpPost=**new** HttpPost("http://www.baidu.com");
22. httpPost.setConfig(requestConfig);
23. CloseableHttpResponse response1 = httpclient.execute(httpPost, context);
24. **try** {
25. HttpEntity entity1 = response1.getEntity();
26. } **finally** {
27. response1.close();
28. }
29. HttpGet httpget2 = **new** HttpGet("http://localhost/2");
30. CloseableHttpResponse response2 = httpclient.execute(httpget2, context);
31. **try** {
32. HttpEntity entity2 = response2.getEntity();
33. } **finally** {
34. response2.close();
35. }

**1.4. HTTP协议拦截器（interceptors）**

HTTP协议拦截器是一个实现HTTP协议特定方面的例程。通常，协议拦截器预期作用于输入消息的一个特定头部或一组相关头部，或者使用一个特定头部或一组相关头部填充输出消息。协议拦截器还可以操纵包含消息的内容实体 - 透明内容压缩/解压缩就是一个很好的例子。 通常这是通过使用“装饰器”模式来实现的，其中使用包装器实体类来装饰原始实体。 几个协议拦截器可以组合形成一个逻辑单元.

协议拦截器可以通过HTTP执行上下文共享信息（如处理状态）进行协作。 协议拦截器可以使用HTTP上下文来存储一个请求或多个连续请求的处理状态。

通常执行拦截器的顺序不要紧，因为它们不依赖于执行上下文的特定状态。 如果协议拦截器具有相互依赖关系，因此必须以特定顺序执行，则应将它们按照与其预期执行顺序相同的顺序添加到协议处理器。

协议拦截器必须实现为线程安全。 与servlet类似，协议拦截器不应使用实例变量，除非对这些变量的访问是同步的

这是一个例子，说明如何使用本地上下文来持续连续请求之间的处理状态：

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. //设置连接KeepAlive策略
2. ConnectionKeepAliveStrategy keepAliveStrategy=**new** DefaultConnectionKeepAliveStrategy() {
3. @Override
4. **public** **long** getKeepAliveDuration(HttpResponse response, HttpContext context) {
5. // TODO Auto-generated method stub
6. **long** keeAlive=**super**.getKeepAliveDuration(response, context);
7. **if**(keeAlive==-1) {
8. // 如果服务器没有显示的设置 keep-alive的时间，就设置为保持连接5秒
9. keeAlive=5000;
10. }
11. **return** keeAlive;
12. }
13. };
14. //配置Request的interceptors拦截器，在请求中添加或者拦截特殊的头部信息或者上下文信息
15. HttpRequestInterceptor httpRequestInterceptor=**new** HttpRequestInterceptor() {
16. @Override
17. **public** **void** process(HttpRequest request, HttpContext context)
18. **throws** HttpException, IOException {
19. AtomicInteger count = (AtomicInteger) context.getAttribute("count");
20. request.addHeader("Count", Integer.toString(count.getAndIncrement()));
22. }
23. ;
24. CloseableHttpClient httpclient=HttpClients.custom().setKeepAliveStrategy(keepAliveStrategy)
25. .addInterceptorLast(httpRequestInterceptor)
26. .build();
27. AtomicInteger count=**new** AtomicInteger(1);
28. HttpClientContext localContext=HttpClientContext.create();
29. localContext.setAttribute("count", count);
30. RequestConfig requestConfig=RequestConfig.custom()
31. .setSocketTimeout(1000)
32. .setConnectTimeout(1000).build();
34. HttpPost httpPost=**new** HttpPost("http://www.baidu.com");
35. httpPost.setConfig(requestConfig);
36. **for**(**int** i=0;i<10;i++) {
37. CloseableHttpResponse response=httpclient.execute(httpPost,localContext);
38. **try** {
39. HttpEntity entity=response.getEntity();
40. }**finally** {
41. response.close();
42. }
43. }

**1.5. 异常处理**

HTTP protocol processors 可以抛出两种类型的异常：java.io.IOException （当 I/O出现异常时例如socket超时或者socket重置），HttpException表示HTTP故障，如违反HTTP协议。通常I / O错误被认为是非致命和可恢复的，而HTTP协议错误被认为是致命的，不能自动恢复。 请注意，HttpClient实现将HttpExceptions重新抛出为ClientProtocolException，它是java.io.IOException的子类。 这使HttpClient的用户能够从单个catch子句处理I / O错误和协议违规。

**1.5.1. HTTP 事务安全**

重要的是要了解HTTP协议不是很适合所有类型的应用程序。HTTP是一种简单的面向请求/响应的协议，最初被设计为支持静态或动态生成的内容检索。从来没有考虑到事务的处理方面。例如，如果HTTP服务器成功接收和处理请求，生成响应并将状态代码发送回客户端，则HTTP服务器将考虑其部分的contract。如果客户端由于读取超时，请求取消或系统崩溃而无法全部收到响应，服务器将不会尝试回滚事务。如果客户端决定重试相同的请求，服务器将不可避免地最终不止一次地执行相同的事务。 在某些情况下，这可能导致应用程序数据损坏或应用程序状态不一致。

即使HTTP从未被设计为支持事务处理，但如果满足某些条件，它仍然可以用作任务关键应用程序的传输协议。 为了确保HTTP传输层安全，系统必须确保应用层上的HTTP方法的等效性。

**1.5.2. Idempotent methods（幂等方法，不是很明白这个）**

在HTTP/1.1  idempotent method定义如下：

[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]

换句话说，应用应该确保能够正确处理执行多个相同方法的含义(也就是说执行相同的方法能够确保相互独立执行). 这可以通过例如提供唯一的事务ID和避免执行相同逻辑操作的其他手段来实现。

请注意，这个问题不是HttpClient特有的。 基于浏览器的应用程序受到与HTTP方法非幂等性相关的完全相同的问题。

默认情况下，HttpClient仅假定非实体封闭方法（如GET和HEAD）为幂等，并且实体封装方法（如POST和PUT）不是出于兼容性原因。

**1.5.3. 自动异常恢复**

默认情况下，HttpClient尝试自动从I / O异常恢复。 默认的自动恢复机制仅限于一些已知是安全的异常。

* HttpClient不会尝试从任何逻辑或HTTP协议错误（从HttpException类派生的错误）中恢复。

HttpClient会自动重试那些假定为幂等的方法。

* 当HTTP请求仍被传送到目标服务器（即请求尚未完全发送到服务器）时，HttpClient将自动重试那些失败的传输异常的方法。

**1.5.4. 请求重试处理器**

为了启用自定义异常恢复机制，应该提供HttpRequestRetryHandler接口的实现。

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. //重新请求策略处理器
2. pRequestRetryHandler myRetryHandler = **new** HttpRequestRetryHandler() {
4. **public** **boolean** retryRequest(
5. IOException exception,
6. **int** executionCount,
7. HttpContext context) {
8. **if** (executionCount >= 5) {
9. // 超过设置的最大次数则不再重试
10. **return** **false**;
11. }
12. **if** (exception **instanceof** InterruptedIOException) {
13. // Timeout
14. **return** **false**;
15. }
16. **if** (exception **instanceof** UnknownHostException) {
17. // Unknown host
18. **return** **false**;
19. }
20. **if** (exception **instanceof** ConnectTimeoutException) {
21. // Connection refused
22. **return** **false**;
23. }
24. **if** (exception **instanceof** SSLException) {
25. // SSL handshake exception
26. **return** **false**;
27. }
28. HttpClientContext clientContext = HttpClientContext.adapt(context);
29. HttpRequest request = clientContext.getRequest();
30. **boolean** idempotent = !(request **instanceof** HttpEntityEnclosingRequest);
31. **if** (idempotent) {
32. // Retry if the request is considered idempotent
33. **return** **true**;
34. }
35. **return** **false**;
36. }

39. CloseableHttpClient httpclient=HttpClients.custom()
40. .setRetryHandler(myRetryHandler)
41. .build();

请注意，可以使用StandardHttpRequestRetryHandler而不是默认使用的HandlerRequestRetryHandler，以便通过RFC-2616定义为幂等的请求方法可以自动重试：GET，HEAD，PUT，DELETE，OPTIONS和TRACE。

**1.6. 终止请求**

在某些情况下，由于目标服务器的负载过高或客户端发出的并发请求太多，HTTP请求执行在预期的时间内无法完成。在这种情况下，可能需要提前终止请求，并在I / O操作中解除阻塞执行线程。由HttpClient执行的HTTP请求可以通过调用HttpUriRequest＃abort（）方法在执行的任何阶段中止。这种方法是线程安全的，可以从任何线程调用。 当一个HTTP请求被中止时，它的执行线程 - 即使当前在I / O操作中阻塞 - 也被保证通过抛出一个InterruptedIOException来解除阻塞；

HttpUriRequest的子类：HttpGet、HttpPost等等

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

1. HttpPost httpPost=**new** HttpPost("http://www.baidu.com");
2. httpPost.setConfig(requestConfig);
3. **for**(**int** i=0;i<10;i++) {
4. httpPost.abort();
5. CloseableHttpResponse response=httpclient.execute(httpPost,localContext);
6. **try** {
7. HttpEntity entity=response.getEntity();
8. }**finally** {
9. response.close();
10. }
11. }

**1.7. 重定向处理**

HttpClient会自动处理所有类型的重定向，除了要求用户干预之外由HTTP规范明确禁止的重定向。请参阅POST上的其他（状态码303）重定向，并将PUT请求按照HTTP规范的要求转换为GET请求。 

可以使用自定义重定向策略来放宽对HTTP规范强加的POST方法的自动重定向的限制。

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

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

HttpClient经常在其执行过程中重写请求消息。 默认情况下，HTTP / 1.0和HTTP / 1.1通常使用相对请求URI。 同样，原始请求可能会从位置重定向到另一次。 最终解释的绝对HTTP位置可以使用原始请求和上下文构建。 URIUtils＃resolve的实用方法可以用来构建用于生成最终请求的解释的绝对URI。 该方法包括来自重定向请求或原始请求的最后一个片段标识符。

**[java]** [view plain](https://blog.csdn.net/chenpeng19910926/article/details/71482347) [copy](https://blog.csdn.net/chenpeng19910926/article/details/71482347)

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. }

**总结：基础部分已经完结，其实这一篇介绍的很多确实是基础部分也提到了很多基础概念。。**