HttpCore 教程

南磊 译

目录

前言

HttpCore 是一套实现了 HTTP 协议最基础方面的组件，尽管 HTTP 协议在使用最小占用来开发全功能的客户端和服务器的 HTTP 服务是足够的。

HttpCore 有如下的范围和目标：

1. HttpCore 范围

- 构建客户端/代理/服务器端 HTTP 服务一致的 API
- 构建同步和异步 HTTP 服务一致的 API
- 基于阻塞（经典的）和非阻塞（NIO）I/O 模型的一套低等级组件

2. HttpCore 目标

- 实现最基本的 HTTP 传输方面
- 良好性能和清晰度&表现力之间的平衡
- 小的（预测）内存占用
- 自我包含的类库（没有超越 JRE 的额外依赖）

3. 什么是 HttpCore 不能做的

- HttpClient 的替代
- Servlet 容器或 Servlet API 竞争对手的替代

关于翻译

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

第一章 基础

1.1 HTTP 报文

1.1.1 结构

HTTP 报文由头部和可选的内容体构成。HTTP 请求报文的头由请求行和头部字段的集合构成。HTTP 响应报文的头部由状态行和头部字段的集合构成。所有 HTTP 报文必须包含协议版本。一些 HTTP 报文可选地可以包含内容体。

HttpCore 定义了 HTTP 报文对象模型，它紧跟定义，而且提供对 HTTP 报文元素进行序列化（格式化）和反序列化（解析）的支持。

1.1.2 基本操作

1.1.2.1 HTTP 请求报文

HTTP 请求是由客户端向服务器端发送的报文。报文的第一行包含应用于资源的方法，资源的标识符，和使用的协议版本。

```
HttpRequest request = new BasicHttpRequest("GET", "/",
HttpVersion.HTTP_1_1);
System.out.println(request.getRequestLine().getMethod());
System.out.println(request.getRequestLine().getUri());
System.out.println(request.getProtocolVersion());
System.out.println(request.getRequestLine().toString());
```

输出内容为：

```
GET
/
HTTP/1.1
GET / HTTP/1.1
```

1.1.2.2 HTTP 响应报文

HTTP 响应是由服务器在收到和解释请求报文之后发回客户端的报文。报文的第一行包含了协议的版本，之后是数字状态码和相关的文本段。

```
HttpResponse response = new
BasicHttpResponse(HttpVersion.HTTP_1_1,HttpStatus.SC_OK, "OK");
System.out.println(response.getProtocolVersion());
System.out.println(response.getStatusLine().getStatusCode());
System.out.println(response.getStatusLine().getReasonPhrase());
System.out.println(response.getStatusLine().toString());
```

输出内容为：

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

1.1.2.3 HTTP 报文通用的属性和方法

HTTP 报文可以包含一些描述报文属性的头部信息，比如内容长度，内容类型等。HttpCore 提供方法来获取，添加，移除和枚举头部信息。

```
HttpResponse response = new
BasicHttpResponse(HttpVersion.HTTP_1_1,
HttpStatus.SC_OK, "OK");
response.addHeader("Set-Cookie",
"c1=a; path=/; domain=localhost");
response.addHeader("Set-Cookie",
"c2=b; path=\"/\", c3=c; domain=\"localhost\"");
Header h1 = response.getFirstHeader("Set-Cookie");
System.out.println(h1);
Header h2 = response.getLastHeader("Set-Cookie");
System.out.println(h2);
Header[] hs = response.getHeaders("Set-Cookie");
System.out.println(hs.length);
```

输出内容为：

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

有一个获得所有给定类型头部信息的有效途径，是使用 HeaderIterator 接口。

```
HttpResponse response = new
BasicHttpResponse(HttpVersion.HTTP_1_1,
HttpStatus.SC_OK, "OK");
response.addHeader("Set-Cookie",
"c1=a; path=/; domain=localhost");
response.addHeader("Set-Cookie",
"c2=b; path=\"/\", c3=c; domain=\"localhost\"");
HeaderIterator it = response.headerIterator("Set-Cookie");
while (it.hasNext()) {
   System.out.println(it.next());
}
```

输出内容为：

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

它也提供方便的方法来解析 HTTP 报文到独立头部元素。

```
HttpResponse response = new
BasicHttpResponse(HttpVersion.HTTP_1_1,
HttpStatus.SC_OK, "OK");
response.addHeader("Set-Cookie","c1=a; path=/;
domain=localhost");
response.addHeader("Set-Cookie",
"c2=b; path=\"/\", c3=c; domain=\"localhost\"");
HeaderElementIterator it = new BasicHeaderElementIterator(
response.headerIterator("Set-Cookie"));
while (it.hasNext()) {
   HeaderElement elem = it.nextElement();
   System.out.println(elem.getName() + " = " +
elem.getValue());
   NameValuePair[] params = elem.getParameters();
   for (int i = 0; i < params.length; i++) {
      System.out.println(" " + params[i]);
   }
}
```

输出内容为：

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

HTTP 头部信息仅在有需要时进行标记化并放入独立的头部元素中。从 HTTP 连接中获取的 HTTP 头部信息被作为字符数组存储在内部而且仅当它们的属性被访问时才延迟解析。

1.1.3 HTTP 实体

HTTP 报文可以携带和请求或响应相关的内容实体。实体可以在一些请求和一些响应中发现，因为它们是可选的。使用了实体的请求被称为包含请求的实体。HTTP 规范定义了两种包含方法的实体：POST 和 PUT。响应通常期望是包含内容实体的。这个规则也有一些例外，比如对 HEAD 方法的响应，204 没有内容，304 没有修改，205 重置内容响应。

HttpCore 区分三种类型的实体，这是基于它们的内容是在哪里生成的：

- **streamed 流式**：内容从流中获得，或者在运行中产生。特别是这种分类包含从 HTTP 响应中获取的实体。流式实体是不可重复生成的。
- **self-contained 自我包含式**：内容在内存中或通过独立的连接或其它实体中获得。自我包含式的实体是可以重复生成的。这种类型的实体会经常用于封闭 HTTP 请求的实体。
- **wrapping 包装式**：内容从另外一个实体中获得。

对于获得不同实体的连接管理的区别是很重要的。对于由应用程序创建的实体仅仅使用 HttpCore 框架来发送，流式和自我包含式之间的区别就没有多大的重要性了。那种情况下，建议将不可重复的实体视为流式的，而可以重复的视为自我包含式的。

1.1.3.1 重复实体

任何可以重复的实体，就是说它的内容可以被读取多余一次。这仅仅是对自我包含式实体来说的（比如 ByteArrayEntity 或 StringEntity）。

1.1.3.2 使用 HTTP 实体

由于实体既可以表示二进制内容，也可以表示字符内容，它对字符编码就有支持（对于后者来说的，也就是字符内容）。

实体是当使用封闭内容执行请求，或当请求已经成功执行，或当响应体结果发功到客户端时创建的。

要从实体中读取内容，可以通过 HttpEntity#getContent() 方法从输入流中获取，这会返回一个 java.io.InputStream 对象，或者提供一个输出流到 HttpEntity#writeTo(OutputStream) 方法中，这会一次返回所有写入到给定流中的内容。

EntityUtils 类，这会暴露出一些静态方法，这些方法可以更加容易地从实体中读取内容或信息。代替直接读取 java.io.InputStream，也可以使用这个类中的方法以字符串/字节数组的形式获取整个内容体。

当实体通过一个收到的报文获取时，HttpEntity#getContentType() 方法和 HttpEntity#getContentLength() 方法可以用来读取通用的元数据，如 Content-Type 和 Content-Length 头部信息（如果它们是可用的）。因为头部信息 Content-Type 可以包含对文本 MIME 类型的字符编码，比如 text/plain 或 text/html，

HttpEntity#getContentEncoding()方法用来读取这个信息。如果头部信息不可用，那么就返回长度-1，而对于内容类型返回 NULL。如果头部信息 Content-Type 是可用的，那么就会返回一个 Header 对象。

当为一个传出报文创建实体时，这个元数据不得不通过实体创建器来提供。

```
StringEntity myEntity = new StringEntity("important message",
"UTF-8");
System.out.println(myEntity.getContentType());
System.out.println(myEntity.getContentLength());
System.out.println(EntityUtils.getContentCharSet(myEntity));
System.out.println(EntityUtils.toString(myEntity));
System.out.println(EntityUtils.toByteArray(myEntity).length);
```

输出内容为

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

1.1.3.3 确保系统资源被释放

为了保证适当释放系统资源，我们必须关闭和实体相关的内容流。

```
HttpResponse response;
HttpEntity entity = response.getEntity();
if (entity != null) {
    InputStream instream = entity.getContent();
    try {
        // 做些有意义的事情
    } finally {
        instream.close();
    }
}
```

要注意一旦实体被完全写出后，HttpEntity#writeTo(OutputStream)方法也要求保证适当的释放系统资源。如果这个方法通过调用 HttpEntity#getContent()获得了一个 java.io.InputStream 实例，也期望在 finally 语句中关闭流。

当处理流式实体时，我们可以使用 EntityUtils#consume(HttpEntity)方法来保证实体内容被完全消耗而且低层的流被关闭。

1.1.4 创建实体

创建实体有一些方法。下面的实现是由 HttpCore 提供的：

- BasicHttpEntity
- ByteArrayEntity
- StringEntity
- InputStreamEntity
- FileEntity
- EntityTemplate
- HttpEntityWrapper
- BufferedHttpEntity

1.1.4.1 BasicHttpEntity 基本 HTTP 实体

这就像它的名字所描述的一样，代表低层流的基本实体。它通常用于从 HTTP 报文中获取的实体。

这个实体有一个空的构造方法。再创建之后它代表没有内容，而且有内容长度是一个负数。

我们需要来设置内容流，和可选的长度值。这分别可以使用 BasicHttpEntity#setContent(InputStream) 和 BasicHttpEntity#setContentLength(long)方法来完成。

```
BasicHttpEntity myEntity = new BasicHttpEntity();
myEntity.setContent(someInputStream);
myEntity.setContentLength(340); // 把长度设置为 340
```

1.1.4.2 ByteArrayEntity 字节数组实体

ByteArrayEntity 是自我包含的，可重复的，从给定的字节数组中获取内容的实体。字节数组是供给构造方法的。

```
String myData = "Hello world on the other side!!";
ByteArrayEntity myEntity =
new ByteArrayEntity(myData.getBytes());
```

1.1.4.3 StringEntity 字符串实体

StringEntity 是自我包含的，可重复的，从 java.lang.String 对象中获取内容的实体。它有两个构造方法，一个使用给定的 java.lang.String 对象来构造；另外那个也对字符串中的数据使用字符编码。

```
StringBuffer sb = new StringBuffer();
Map<String, String> env = System.getenv();
for (Entry<String, String> envEntry : env.entrySet()) {
    sb.append(envEntry.getKey()).append(": ")
    .append(envEntry.getValue()).append("\n");
}
```

```
// 不使用字符编码构建
HttpEntity myEntity1 = new StringEntity(sb.toString());
// 另外使用字符编码构建
HttpEntity myEntity2 = new StringEntity(sb.toString(),
"UTF-8");
```

1.1.4.4 InputStreamEntity 输入流实体

InputStreamEntity 是流式的，不能重复的实体，它获得输入流中的内容。它通过提供输入流和内容长度来构建。内容长度用来限制从 java.io.InputStream 对象中读取数据的数量。如果长度匹配输入流中可用的内容长度，那么所有数据都会被发送。另外负数的内容长度将会从输入流中读取所有数据，这和提供精确的内容长度是一样的，所以长度通常是用来限制长度的。

```
InputStream instream = getSomeInputStream();
InputStreamEntity myEntity = new InputStreamEntity(instream,
16);
```

1.1.4.5 FileEntity 文件实体

FileEntity 是自我包含式，可以重复的实体，它从文件中获取内容。因为这通常是用于不同类型的大文件流，我们需要提供文件的类型，比如，发送一个 zip 文件需要 application/zip 类型，对于 XML 就是 application/xml。

```
HttpEntity entity = new FileEntity(staticFile,
"application/java-archive");
```

1.1.4.6 EntityTemplate 实体模板

这是从 ContentProducer 接口接收内容的实体。内容生产者是生成它们想要内容的对象，将它们写入到输出流中。它们被期望每次被请求时来生成内容。所以创建 EntityTemplate 后，我们要提供给内容生产者提供一个引用，这可以很有效地创建可重复实体。

在 HttpCore 中没有标准的内容生产者。基本上只是允许包装复杂逻辑到实体的方便的接口。要使用实体，我们需要创建实现 ContentProducer 接口的类，并覆盖 ContentProducer#writeTo(OutputStream) 方法。之后，自定义 ContentProducer 的实例会被用来将全部内容体写入到输出流中。比如，HTTP 服务器会使用 FileEntity 来服务静态文件，但是运行 CGI 程序可以通过 ContentProducer 完成，在其中可以实现自定义逻辑来提供可用的内容。这个方式我们不需要在字符串中来缓冲，之后使用 StringEntity 或 ByteArrayEntity。

```
ContentProducer myContentProducer = new ContentProducer() {
   public void writeTo(OutputStream out) throws IOException {
      out.write("ContentProducer rocks! ".getBytes());
      out.write(("Time requested: " + new Date()).getBytes());
   }
};
HttpEntity myEntity = new EntityTemplate(myContentProducer);
myEntity.writeTo(System.out);
```

输出内容为：

```
ContentProducer rocks! Time requested: Fri Sep 05 12:20:22 CEST
2008
```

1.1.4.7 HttpEntityWrapper HTTP 实体包装器

这是创建被包装实体的基类。包装实体持有被包装实体的引用，而且将所有调用都委派给它。包装实体的实现可以从这个类派生，而且需要覆盖仅仅那些不能被委派给被包装实体的方法。

1.1.4.8 BufferedHttpEntity 缓冲 HTTP 实体

BufferedHttpEntity 是 HttpEntityWrapper 的子类。它由提供另外一个实体来创建。它从所提供实体内读取内容，然后缓冲在内存中。

这使得从不可重复实体中生成可重复的实体成为可能。如果所提供的实体已经是可重复的，那么调用仅仅是传递给底层实体。

```
myNonRepeatableEntity.setContent(someInputStream);
BufferedHttpEntity myBufferedEntity = new BufferedHttpEntity(
myNonRepeatableEntity);
```

1.2 阻塞 HTTP 连接

HTTP 连接是负责 HTTP 报文序列化和反序列化的。我们应该极少直接使用 HTTP 连接对象。有高级协议组件来执行 HTTP 请求的处理。而在某些情况下直和 HTTP 连接直接交互是需要的，比如，访问如连接状态，套接字超时时间或本地和远程地址属性。

牢记 HTTP 连接是线程不安全的，这点非常重要。强烈建议限制所有 HTTP 连接对象的交互到一个线程中。HttpConnection 接口唯一的方法和它的从另外一个线程中安全调用子接口的 HttpConnection#shutdown() 方法。

1.2.1 使用阻塞的 HTTP 连接

HttpCore 不提供打开连接的完全支持，因为创建一个新连接的过程-特别是在客户端创建，当涉及到一个或多个认证和或/和通道代理时，这个过程可能是非常复杂的。相反，阻塞 HTTP 连接可以绑定到任意的网络套接字上。

```
Socket socket = new Socket();
// 初始化套接字
BasicHttpParams params = new BasicHttpParams();
DefaultHttpClientConnection conn = new
DefaultHttpClientConnection();
conn.bind(socket, params);
conn.isOpen();
HttpConnectionMetrics metrics = conn.getMetrics();
metrics.getRequestCount();
metrics.getResponseCount();
metrics.getReceivedBytesCount();
metrics.getSentBytesCount();
```

在客户端和服务器端的 HTTP 连接接口都可以在两个阶段发送和接收报文。报文头部首先传送。根据报文头部的属性，它可能跟着报文体。请注意通常关闭低层的内容流来给报文处理完毕发送信号是非常重要的。从底层连接输入流中直接获取内容流出的 HTTP 实体必须要保证报文体的内容被潜在的可重用的连接完全消耗。

简化客户端请求执行的程序可能看起来像这样：

```
Socket socket = new Socket();
// 初始化套接字
HttpParams params = new BasicHttpParams();
DefaultHttpClientConnection conn = new
DefaultHttpClientConnection();
conn.bind(socket, params);
HttpRequest request = new BasicHttpRequest("GET", "/");
conn.sendRequestHeader(request);
HttpResponse response = conn.receiveResponseHeader();
conn.receiveResponseEntity(response);
HttpEntity entity = response.getEntity();
if (entity != null) {
    // 用实体做点有用的事情，当完成时，报省所有内容都被消耗掉
    // 所以底层连接可能被重用。
EntityUtils.consume(entity);
}
```

简化服务器端请求处理的程序可能看起来像这样：

```
Socket socket = new Socket();
// 初始化套接字
HttpParams params = new BasicHttpParams();
DefaultHttpServerConnection conn = new
DefaultHttpServerConnection();
conn.bind(socket, params);
HttpRequest request = conn.receiveRequestHeader();
if (request instanceof HttpEntityEnclosingRequest) {
   conn.receiveRequestEntity((HttpEntityEnclosingRequest)
request);
   HttpEntity entity = ((HttpEntityEnclosingRequest) request)
.getEntity();
   if (entity != null) {
      // 用实体做点有用的事情，当完成时，报省所有内容都被消耗掉
      // 所以底层连接可能被重用。
      EntityUtils.consume(entity);
   }
}
HttpResponse response = new
BasicHttpResponse(HttpVersion.HTTP_1_1,
200, "OK");
response.setEntity(new StringEntity("Got it"));
conn.sendResponseHeader(response);
conn.sendResponseEntity(response);
```

请注意我们应该很少来使用低级别的方法来发送报文，相反，应该使用适合的高级别的 HTTP 服务实现。

1.2.2 内容传输与阻塞 I/O

HTTP 连接报文的内容传输处理使用 HttpEntity 接口。HTTP 连接生成封装了收到报文的内容流的实体对象。请注意 HttpServerConnection#receiveRequestEntity() 和 HttpClientConnection#receiveResponseEntity() 方法不会取出或缓冲任何收到的数据。它们仅仅是注入合适的依赖于收到报文属性的内容解码器。内容可以使用 HttpEntity#getContent() 方法，通过从包含实体的内容输入流中读取来获得。收到的数据将会完全自动被解码，透明地给数据消费者。同样，HTTP 连接依赖 HttpEntity#writeTo(OutputStream) 方法来生成输出报文的内容。如果输出报文包含实体，那么内容将会基于报文属性自动编码。

1.2.3 支持的内容传输机制

默认的 HTTP 连接支持的实现支持三种内容传机制输由 HTTP/1.1 规范定义：

- **Content-Length 分隔**：内容实体的结束是由头部信息 Content-Length 的值来决定的。最大的实体长度：Long#MAX_VALUE。
- **身份编码**：内容实体的结束是通过关闭底层连接（流结束的条件）来划定的。由于显而易见的原因，身份编码只能在服务器端使用。最大实体长度：不限制。
- **块编码**：内容是以小块发送的。最大实体长度：不限制。

合适的内容流类将会依赖于报文内包含实体的属性来自动创建。

1.2.4 终止 HTTP 连接

HTTP 连接的终止可以通过优雅地调用 HttpConnection#close()方法或强制地调用 HttpConnection#shutdown()方法。前者试图刷新所有缓冲数据，优先于终止连接也许会无限期阻塞。HttpConnection#close()方法是线程不安全的。后者终止连接不会去刷新内部缓冲而会把控制权尽快返回给调用者不会阻塞太长时间。HttpConnection#shutdown()方法是线程安全的。

1.3 HTTP 异常处理

所有的 HttpCore 组件会潜在地抛出两种类型的异常：在如套接字超时或重置的 I/O 失败时的 IOException 异常，还有如违反 HTTP 协议标识着 HTTP 失败的 HttpException 异常。通常 I/O 错误被认为是非致命的而且可以恢复的，而 HTTP 协议错误被认为是致命的而且是不能自动恢复的。

1.3.1 协议异常

ProtocolException 异常标识着致命的违反 HTTP 协议，通常会导致立即终止 HTTP 报文的处理。

1.4 HTTP 协议处理器

HTTP 协议拦截器是实现了 HTTP 协议特定方面的规则。通常情况下，协议拦截器期望作用于特定的头部信息或收到的报文的一组相关的头部信息，或者是使用特定的头部信息或一组相关的头部信息填充发出的报文。协议拦截器也可以操作包含在报文中的内容实体，透明的内容压缩/解压就是一个很好的示例。通常这可以使用“装饰者”模式来完成，其中一个包装器实体类可以用于装饰原实体。一些协议拦截器可以联合起来成为一个逻辑单元。

HTTP 协议处理器是一组协议拦截器的集合，并且实现了“责任链”模式，其中每个独立的协议拦截器期望处理它负责的 HTTP 协议的特定方面。

通常情况下，拦截器执行的顺序应该和依赖于特定的内容执行状态没有关系。如果协议拦截器有相互依赖的关系，那么就必须按照特定的顺序来执行，它们应该以被期望执行顺序的相同的序列被加入到协议处理器中。

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

1.4.1 标准协议拦截器

HttpCore 附带了一些很重要的协议拦截器，用于客户端和服务器端的 HTTP 处理。

1.4.1.1 RequestContent 请求内容

RequestContent 是对发出请求最重要的拦截器。它负责划定内容的长度，通过添加基于被包含实体和协议版本属性的Content-Length或Transfer-Content头部信息来实现。这个拦截器需要正确的客户端协议处理器的运行。

1.4.1.2 ResponseContent 响应内容

ResponseContent 是对发出响应最重要的拦截器。它负责划定内容的长度，通过添加基于被包含实体和协议版本属性的Content-Length或Transfer-Content头部信息来实现。这个拦截器需要正确的服务器端协议处理器的运行。

1.4.1.3 RequestConnControl 请求连接控制

RequestConnControl 负责添加头部信息 Connection 到发出的请求中，这对管理 HTTP/1.0 连接持久化是必须的。这个拦截器建议用于客户端协议处理器。

1.4.1.4 ResponseConnControl 响应连接控制

ResponseConnControl 负责添加头部信息 Connection 到发出的请求中，这对管理 HTTP/1.0 连接持久化是必须的。这个拦截器建议用于服务器端协议处理器。

1.4.1.5 RequestDate 请求日期

RequestDate 负责添加头部信息 Date 到发送请求中。这个拦截器对于客户端协议处理器来说是可选的。

1.4.1.6 ResponseDate 响应日期

ResponseDate 负责添加头部信息 Date 到发送响应中。这个拦截器建议用于服务器端协议处理器中。

1.4.1.7 RequestExpectContinue 请求希望继续

RequestExpectContinue 负责开启“expect-continue”（希望继续）握手，这是通过添加头部信息 Expect 来完成的。这个拦截器建议用于客户端协议处理器。

1.4.1.8 RequestTargetHost 请求目标主机

RequestTargetHost 负责添加头部信息 Host。这个拦截器需要用于客户端协议处理器。

1.4.1.9 RequestUserAgent 请求用户代理

RequestUserAgent 负责添加头部信息 User-Agent。这个拦截器建议用于客户端协议处理器。

1.4.1.10 ResponseServer 响应服务器

ResponseServer 负责添加头部信息 Server。这个拦截器建议用于服务器端协议处理器。

1.4.2 使用协议处理器

通常情况下，HTTP 协议处理器用于在执行应用特定处理逻辑之前预处理接收的报文，在发送出报文之后也要一些后续处理工作。

```
BasicHttpProcessor httpproc = new BasicHttpProcessor();
// 必须的协议拦截器
httpproc.addInterceptor(new RequestContent());
httpproc.addInterceptor(new RequestTargetHost());
// 推荐的协议拦截器
httpproc.addInterceptor(new RequestConnControl());
httpproc.addInterceptor(new RequestUserAgent());
httpproc.addInterceptor(new RequestExpectContinue());
HttpContext context = new BasicHttpContext();
HttpRequest request = new BasicHttpRequest("GET", "/");
httpproc.process(request, context);
HttpResponse response = null;
```

将请求发送到目标服务器，获取响应。

```
httpproc.process(response, context);
```

请注意 BasicHttpProcessor 类没有同步对它内部结构的访问，因此可能是线程不安全的。

1.4.3 HTTP 上下文

协议拦截器可以通过共享信息来合作-比如处理状态-通过 HTTP 执行上下文。HTTP 上下文是一个用于映射属性名称到属性值的结构。从内部来说，HTTP 上下文实现通常是用一个 HashMap。HTTP 上下文主要的作用是促进信息在多个逻辑相关组件之间的共享。HTTP 上下文。HTTP 上下文可以用来存储一个或几个连续报文的处理状态。如果相同的上下文在连续报文之间被重用，多个逻辑相关报文可以参与到一个逻辑会话中。

```
BasicHttpProcessor httpproc = new BasicHttpProcessor();
httpproc.addInterceptor(new HttpRequestInterceptor() {
   public void process(HttpRequest request,
   HttpContext context) throws HttpException, IOException {
      String id = (String) context.getAttribute("session-id");
      if (id != null) {
         request.addHeader("Session-ID", id);
      }
   }
});
HttpRequest request = new BasicHttpRequest("GET", "/");
httpproc.process(request, context);
```

HttpContext 的实例可以连接起来形成一个层次结构。在最简单的形式中，一个上下文可以使用另外一个上下文的内容来获取本地上下文中没有表现出的属性的默认值。

```
HttpContext parentContext = new BasicHttpContext();
parentContext.setAttribute("param1", Integer.valueOf(1));
parentContext.setAttribute("param2", Integer.valueOf(2));
HttpContext localContext = new BasicHttpContext();
localContext.setAttribute("param2", Integer.valueOf(0));
localContext.setAttribute("param3", Integer.valueOf(3));
HttpContext stack = new DefaultedHttpContext(localContext,
parentContext);
System.out.println(stack.getAttribute("param1"));
System.out.println(stack.getAttribute("param2"));
System.out.println(stack.getAttribute("param3"));
System.out.println(stack.getAttribute("param4"));
```

输出内容为：

```
1
0
3
null
```

1.5 HTTP 参数

HttpParams 接口代表定义组件运行时行为不变的值的集合。在很多方面，HttpParams 和 HttpContext 是很相似的。二者之间的主要区别是它们在运行时的使用。两个接口都代表了被组织为文本名称和对象值映射的对象集合，但是服务于不同的目的：

- HttpParams 意在包含简单对象：整数，双精度浮点数，字符串，集合和在运行时保持不变的对象。HttpParams 期望用于“一次编写-多处准备”模式。HttpContext 意在包含在 HTTP 报文处理期间可以轻易改变的复杂对象。
- HttpParams 的目标是定义其它组件的行为。通常情况下，每个复杂组件都有它自己的 HttpParams 对象。HttpContext 的目标是用来代表一个 HTTP 处理的执行状态。通常情况下，相同执行上下文在很多合作的对象中是共享的。

就像 HttpContext，HttpParams 也可以连接在一起形成层次结构。在最简单的形式中，参数的集合可以使用另外一个的内容来获取没有在这个集合中出现的参数的默认值。

```
HttpParams parentParams = new BasicHttpParams();
parentParams.setParameter(CoreProtocolPNames.PROTOCOL_VERSION,
HttpVersion.HTTP_1_0);
parentParams.setParameter(CoreProtocolPNames.HTTP_CONTENT_CHARSET,
"UTF-8");
HttpParams localParams = new BasicHttpParams();
localParams.setParameter(CoreProtocolPNames.PROTOCOL_VERSION,
HttpVersion.HTTP_1_1);
localParams.setParameter(CoreProtocolPNames.USE_EXPECT_CONTINUE,
Boolean.FALSE);
HttpParams stack = new DefaultedHttpParams(localParams,
parentParams);
System.out.println(stack.getParameter(
CoreProtocolPNames.PROTOCOL_VERSION));
System.out.println(stack.getParameter(
CoreProtocolPNames.HTTP_CONTENT_CHARSET));
System.out.println(stack.getParameter(
CoreProtocolPNames.USE_EXPECT_CONTINUE));
System.out.println(stack.getParameter(
CoreProtocolPNames.USER_AGENT));
```

输出内容为：

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

BasicHttpParams请注意类没有同步访问它内部的结构，因此可能是线程不安全的。

1.5.1 HTTP 参数 bean

HttpParams 接口允许在处理组件配置时的很大的灵活性。很重要的是，新的参数可以被引入而不影响老版本的二进制组件。但是和传统的 Java bean 相比，HttpParams 也有一个确定的缺点：HttpParams 不能使用 DI（Dependency Injection，依赖注入，译者注）框架来组装。为了环节限制，HttpCore 包含了很多 bean 类，它们可以使用标准的 Java bean 约定来初始化对象。

```
HttpParams params = new BasicHttpParams();
HttpProtocolParamBean paramsBean = new
HttpProtocolParamBean(params);
paramsBean.setVersion(HttpVersion.HTTP_1_1);
paramsBean.setContentCharset("UTF-8");
paramsBean.setUseExpectContinue(true);
System.out.println(params.getParameter(
CoreProtocolPNames.PROTOCOL_VERSION));
System.out.println(params.getParameter(
CoreProtocolPNames.HTTP_CONTENT_CHARSET));
System.out.println(params.getParameter(
CoreProtocolPNames.USE_EXPECT_CONTINUE));
System.out.println(params.getParameter(
CoreProtocolPNames.USER_AGENT));
```

输出内容为：

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

1.6 阻塞 HTTP 协议处理程序

1.6.1 HTTP 服务

HttpService 是基于阻塞 I/O 模型来为由 RFC 2616（HTTP/1.1 规范定义的 RFC 文档，译者注）描述的服务器端报文处理，实现 HTTP 协议的基本要求的服务器端 HTTP 协议处理程序。

HttpService 依赖于 HttpProcessor 实例对所有发出的报文来生成强制协议头部并对所有收到的和发出的报文应用通用的，交叉消息转换，而 HTTP 请求处理程序期望保护应用程序特定的内容生成和处理。

```
HttpParams params;
// 初始化HTTP参数
HttpProcessor httpproc;
// 初始化HTTP处理器
HttpService httpService = new HttpService(
   httpproc,new DefaultConnectionReuseStrategy(),
   new DefaultHttpResponseFactory());
httpService.setParams(params);
```

1.6.1.1 HTTP 请求处理程序

HttpRequestHandler 接口代表了处理指定 HTTP 请求组的规则。HttpService 被设计用来关注协议的指定方面，而独立的请求处理程序期望去关注应用程序特定的 HTTP 处理。请求处理程序的主要目的是使用内容实体生成响应对象送回客户端来响应给定的请求。

```
HttpRequestHandler myRequestHandler = new HttpRequestHandler() {
   public void handle(HttpRequest request,HttpResponse response,
      HttpContext context) throws HttpException, IOException {
      response.setStatusCode(HttpStatus.SC_OK);
      response.addHeader("Content-Type", "text/plain");
      response.setEntity(
         new StringEntity("some important message"));
   }
};
```

1.6.1.2 请求处理程序解析器

HTTP 请求处理程序通常是由 HttpRequestHandlerResolver 来管理的，它匹配请求 URI 到请求处理程序。HttpCore 包含一个基于琐碎模式匹配算法：HttpRequestHandlerRegistry 支持仅仅 3 种格式：*，<uri>*和*<uri>的请求处理程序解析器的非常简单的实现

```
HttpService httpService;
// 初始化HTTP服务
HttpRequestHandlerRegistry handlerResolver =
new HttpRequestHandlerRegistry();
handlerReqistry.register("/service/*", myRequestHandler1);
handlerReqistry.register("*.do", myRequestHandler2);
handlerReqistry.register("*", myRequestHandler3);
// 注入处理程序解析器
httpService.setHandlerResolver(handlerResolver);
```

我们鼓励用户提供更复杂的 HttpRequestHandlerResolver 实现，比如，基于正

则表达式的实现。

1.6.1.3 使用 HTTP 服务来处理请求

当完全加载和配置后，HttpService 可以用于执行和处理对活动 HTTP 连接的请求。HttpService#handleRequest()方法读取传入的请求，生成响应并发回客户端。这个方法可以在循环中执行来处理一个持久化连接中的多个请求。HttpService#handleRequest()方法在多线程中执行是安全的。这就允许同时对多个连接的请求进行处理，只要所有的协议拦截器和请求处理程序由 HttpService 使用是线程安全的就行。

```
HttpService httpService;
// 初始化HTTP服务
HttpServerConnection conn;
// 初始化连接
HttpContext context;
// 初始化HTTP上下文
boolean active = true;
try {
   while (active && conn.isOpen()) {
      httpService.handleRequest(conn, context);
   }
} finally {
   conn.shutdown();
}
```

1.6.2 HTTP 请求执行器

HttpRequestExecutor 是基于阻塞 I/O 模型的客户端 HTTP 协议处理程序，它实现了 HTTP 协议对客户端报文处理的基本要求，这在 RFC 2616 中被描述出来了。HttpRequestExecutor 依赖于 HttpProcessor 实例对所有发送的报文来生成强制性的协议头部信息，而且对所有收到和发送的报文应用通用的，交叉报文转换。应用程序指定的处理可以当请求被执行和收到响应时在 HttpRequestExecutor 外部实现。

```
HttpClientConnection conn;
// 创建连接
HttpParams params;
// 初始化HTTP参数
HttpProcessor httpproc;
// 初始化HTTP处理器
HttpContext context;
// 初始化HTTP上下文
HttpRequestExecutor httpexecutor = new HttpRequestExecutor();
```

```
BasicHttpRequest request = new BasicHttpRequest("GET", "/");
request.setParams(params);
httpexecutor.preProcess(request, httpproc, context);
HttpResponse response = httpexecutor.execute(
request, conn, context);
response.setParams(params);
httpexecutor.postProcess(response, httpproc, context);
HttpEntity entity = response.getEntity();
EntityUtils.consume(entity);
```

HttpRequestExecutor 的方法在多线程中执行是安全的。这就允许同时对多个连接的请求进行处理，只要所有的协议拦截器由 HttpRequestExecutor 使用是线程安全的就行。

1.6.3 连接持久化/重用

ConnectionReuseStrategy 接口意在决定是否底层的连接可以对将来在当前报文完成传输之后报文的处理继续重用。默认的连接重用策略无论何时都尝试保持连接活动。首先，它会检查用于传输报文的 HTTP 协议的版本。HTTP/1.1 连接默认是持久性的，而 HTTP/1.0 连接则不是。其次，它会检查 Connection 头部信息的值。它的相同作用的实例也可以通过在 Connection 头部信息中发送 Keep-Alive 或值指示是否在另一端重用连接。第三，这个策略基于被包含实体的属性（如果属性可用时）决定连接是否安全地重用。

第二章 NIO 的扩展

2.1 非阻塞 I/O 模型的优点和缺点

和大众的看法相反，NIO 的性能在数据吞吐量方面显著地比阻塞 I/O 要低。NIO 不一定适用于所有的情况，应该被用在合适的地方：

- 处理成千上万的连接，其中相当数量的连接可以闲置。
- 处理高延迟的连接。
- 请求/响应处理需要脱钩的。

2.2 和其它 NIO 框架的差异

解决其它框架类似的问题，但是有一定的显著特性：

简约，对数据量密集协议的优化，比如 HTTP

有效的内存管理：数据消费者可以读取它能处理的数据输入量而不去分配更多的内存。

直接访问可能的 NIO 通道

2.3 I/O 反应器

HttpCore NIO 是基于由 Doug Lea 定义描述的反应器模式的。I/O 反应器的目标是反应 I/O 事件然后分发事件通知到独立的 I/O 会话中。I/O 反应器模式的主要思想是从每次连接模型摆脱一个由经典阻塞 I/O 模型强加的线程。`IOReactor` 接口代表了反应器模式抽象的对象实现。在其内部，`IOReactor` 实现封装了了 NIO `java.nio.channels.Selector` 的功能。

I/O 反应器通常使用一小部分分发线程（通常是一个）来分发 I/O 事件通知到一个很大数量（通常是好几千）的 I/O 会话或连接。通常建议每个 CPU 核心只有一个分发线程。

```
HttpParams params = new BasicHttpParams();
int workerCount = 2;
IOReactor ioreactor = new
DefaultConnectingIOReactor(workerCount,params);
```

2.3.1 I/O 分发器

`IOReactor` 实现使用了 `IOEventDispatch` 接口来通知事件客户端为特定的会话挂起。`IOEventDispatch` 的所有方法在 I/O 反应器的分发线程上执行。因此，在事件方法上的处理不会阻塞分发线程很长时间，这一点是很重要的，而且 I/O 反应器也不可能对其它事件做出反应。

```
HttpParams params = new BasicHttpParams();
IOReactor ioreactor = new DefaultConnectingIOReactor(2,
params);
IOEventDispatch eventDispatch = new MyIOEventDispatch();
ioreactor.execute(eventDispatch);
```

由 IOEventDispatch 接口定义的通用 I/O 事件：

- **connected**：当一个新的会话创建时触发。
- **inputReady**：当新的会话等待输入时触发。
- **outputReady**：当会话准备输出时触发。
- **timeout**：当会话超时时触发。
- **disconnected**：当会话终止时触发。

2.3.2 I/O 反应器关闭

I/O 反应器的关闭是一个复杂的过程而且可能需要一些时间来完成。I/O 反应器将会试图在规定的宽限期内优雅地终止几乎所有活动的 I/O 会话和分发线程。如果任何一个 I/O 会话没有正确终止，I/O 反应器将会强制关闭剩余会话。

```
long gracePeriod = 3000L; // 毫秒
ioreactor.shutdown(gracePeriod);
```

IOReactor#shutdown(long)方法从任何线程中调用都是安全的。

2.3.3 I/O 会话

IOSession 接口代表了两个端点之间逻辑相关的数据交换的序列。IOSession 封装了 NIO java.nio.channels.SelectionKey 和 java.nio.channels.SocketChannel 的功能。和 IOSession 相关的通道可以被用来从会话读取数据和向会话中写入数据。

```
IOSession iosession;
ReadableByteChannel ch = (ReadableByteChannel)
iosession.channel();
ByteBuffer dst = ByteBuffer.allocate(2048);
ch.read(dst);
```

2.3.4 I/O 会话状态管理

I/O 会话没有绑定到执行线程，因此我们不能使用线程的上下文来存储会话的状态。所有关于特定会话的细节必须存储在会话自身中。

```
IOSession iosession;
Object someState;
iosession.setAttribute("state", someState);
Object currentState = iosession.getAttribute("state");
```

请注意如果一些会话使用了共享对象，访问那些对象必须是线程安全的。

2.3.5 I/O 会话事件掩码

我们可以在特定类型的 I/O 事件中对特定的 I/O 会话通过设置它的事件掩码来声明兴趣点。

```
IOSession iosession;
iosession.setEventMask(SelectionKey.OP_READ |
SelectionKey.OP_WRITE);
```

我们也可以独自切换 OP_READ 和 OP_WRITE 标志。

```
iosession.setEvent(SelectionKey.OP_READ);
iosession.clearEvent(SelectionKey.OP_READ);
```

如果响应的兴趣点标志没有设置，那么事件通知就不会发生。

2.3.6 I/O 会话缓冲

I/O 会话常常需要来维护内部的 I/O 缓冲来转换输入/输出数据，优先于返回数据到消费者中或写入低层通道中。HttpCore NIO 的内存管理基于数据消费者只能读取它能处理的最大数据量而不用分配更多内存的根本原则。这就是说，常常有一些输入数据可能在内部或外部会话缓冲中保持未读状态。I/O 反应器可以查询这些会话缓冲的状态，而且保证消费者当更多数据存储在会话缓冲中时，可以得到正确的通知，因此一旦有能力处理数据时，允许消费者读取剩余数据。I/O 会话可以使用 SessionBufferStatus 接口来了解外部会话缓冲的状态。

```
IOSession iosession;
SessionBufferStatus myBufferStatus = new
MySessionBufferStatus();
iosession.setBufferStatus(myBufferStatus);
iosession.hasBufferedInput();
iosession.hasBufferedOutput();
```

2.3.7 I/O 会话关闭

我们可以通过调用允许会话有序关闭的 IOSession#close()方法或强制关闭底层通道的 IOSession#shutdown()方法来优雅地关闭 I/O 会话。两个方法的区别是对于那

些涉及会话终止握手（如 SSL/TLS 连接）的排序的 I/O 会话类型的首要的重要性。

2.3.8 监听 I/O 反应器

ListeningIOReactor 代表了有能力监听在一些端口上到达连接的 I/O 反应器。

```
ListeningIOReactor ioreactor;
ListenerEndpoint ep1 = ioreactor.listen(new
InetSocketAddress(8081));
ListenerEndpoint ep2 = ioreactor.listen(new
InetSocketAddress(8082));
ListenerEndpoint ep3 = ioreactor.listen(new
InetSocketAddress(8083));
// 等待，直到所有端点都注册
ep1.waitFor();
ep2.waitFor();
ep3.waitFor();
```

一旦端点完全初始化，它就开始接收收到的连接，传播 I/O 活动通知给 IOEventDispatch 接口。

我们可以在运行时获取一组注册的端点，查询运行时一个端点的状态，如果需要也可以关闭它。

```
ListeningIOReactor ioreactor;
Set<ListenerEndpoint> eps = ioreactor.getEndpoints();
for (ListenerEndpoint ep: eps) {
   // 仍然是活动的?
   System.out.println(ep.getAddress());
   if (ep.isClosed()) {
      // 如果不是，它是由于异常而终止的吗?
      if (ep.getException() != null) {
         ep.getException().printStackTrace();
      }
   } else {
      ep.close();
   }
}
```

2.3.9 连接 I/O 反应器

ConnectingIOReactor 代表了能够和远程主机建立连接的 I/O 反应器。

```
ConnectingIOReactor ioreactor;
SessionRequest sessionRequest = ioreactor.connect(
new InetSocketAddress("www.google.com", 80),null, null, null);
```

打开到远程主机的连接通常往往是一个耗费时间的过程，而且需要一定时间来完成。我们可以凭借 SessionRequest 接口监控和控制会话初始化的过程。

```
// 如果连接在1秒之后没有建立，确保请求超时
sessionRequest.setConnectTimeout(1000);
// 等待请求完成
sessionRequest.waitFor();
// 请求因为异常而终止了吗?
if (sessionRequest.getException() != null) {
    sessionRequest.getException().printStackTrace();
}
// 获取新的I/O会话
IOSession iosession = sessionRequest.getSession();
```

SessionRequest 实现希望是线程安全的。会话请求可以从其它任意执行线程中通过调用 IOSession#cancel()方法在任何时候终止。

```
if (!sessionRequest.isCompleted()) {
    sessionRequest.cancel();
}
```

我们可以传递一些可选的参数给 ConnectingIOReactor#connect()方法来发挥对会话初始化过程的更强控制。

非空的本地套接字地址参数可以用于绑定这个套接字到指定的本地地址上。

```
ConnectingIOReactor ioreactor;
SessionRequest sessionRequest = ioreactor.connect(
new InetSocketAddress("www.google.com", 80),
new InetSocketAddress("192.168.0.10", 1234),null, null);
```

我们可以提供附件对象，它可以在初始化时加到新的会话上下文中。这个对象可以用于给协议初期程序传递初始处理状态。

```
SessionRequest sessionRequest = ioreactor.connect(
new InetSocketAddress("www.google.com", 80),
null, new HttpHost("www.google.ru"), null);
IOSession iosession = sessionRequest.getSession();
HttpHost virtualHost = (HttpHost) iosession.getAttribute(
IOSession.ATTACHMENT_KEY);
```

它通常需要能够反应异步会话请求的完成情况，而不需要来等待，阻塞当前的执行线程。我们可以提供可选的 SessionRequestCallback 接口的实现来通知事件相关的会话请求，必须请求完成，取消，失败或超时。

```
ConnectingIOReactor ioreactor;
SessionRequest sessionRequest = ioreactor.connect(
   new InetSocketAddress("www.google.com", 80), null, null,
   new SessionRequestCallback() {
      public void cancelled(SessionRequest request) {
   }
   public void completed(SessionRequest request) {
      System.out.println("new connection to " +
      request.getRemoteAddress());
   }
   public void failed(SessionRequest request) {
      if (request.getException() != null) {
         request.getException().printStackTrace();
      }
   }
   public void timeout(SessionRequest request) {
   }
});
```

2.3.10 I/O 兴趣点集合操作的队列

一些老的 JRE 实现（主要是 IBM）包含 Java API 文档参考作为一个本地的 java.nio.channels.SelectionKey 类的实现。在这样的 JRE 中，java.nio.channels.SelectionKey 的问题是读取或写入 I/O 兴趣点集合时，如果 I/O 选择器在执行一个选择操作过程中，可能会无限期地阻塞。HttpCore NIO 可以配置来在特殊模式下操作，比如在 I/O 兴趣点操作是队列时，在仅当 I/O 选择器没有参与选择操作的分发线程上执行时。

```
HttpParams params = new BasicHttpParams();
NIOReactorParams.setInterestOpsQueueing(params, true);
ListeningIOReactor ioreactor = new
DefaultListeningIOReactor(2, params);
```

2.4 I/O 反应器异常处理

协议特定的异常和那些在和会话通道交互过程中抛出的 I/O 异常期望被特定的协议处理程序来处理。这些异常可能导致独立会话的终止但是不会影响 I/O 反应器和其它活动的会话。在某些情况下，比方说，当 I/O 反应器本身遇到了如在底层 NIO 类的 I/O 异常或未处理的运行时异常等内部问题。那些类型的异常通常是致命的，而且会引起 I/O 反应器自动关闭。

覆盖这种行为并阻止 I/O 反应器在运行时发生异常时自动关闭或内部类的 I/O 异常是有可能的。这可以通过提供自定义的 IOReactorExceptionHandler 接口的实现来完成。

```
DefaultConnectingIOReactor ioreactor;
ioreactor.setExceptionHandler(new
IOReactorExceptionHandler() {
   public boolean handle(IOException ex) {
      if (ex instanceof BindException) {
         // 绑定被确认为OK的故障为忽略
         return true;
      }
      return false;
   }
   public boolean handle(RuntimeException ex) {
      if (ex instanceof UnsupportedOperationException) {
         // 被认为是OK的不支持的操作被忽略
         return true;
      }
   return false;
   }
});
```

在抛弃异常时我们必须非常小心。让 I/O 反应器自行关闭是最好的，之后重启它而不是将它留在易变，不稳定的状态。

2.4.1 I/O 反应器审计日志

如果 I/O 反应器不能自动从 I/O 或运行时异常中修复，那么它可能会进入关闭模式。首先，它会关闭所有的活动监听器，取消所有等待新会话的请求。之后它会试图去关闭所有活动的 I/O 会话，给它们一定的时间来刷新等待输出数据和终止。最后，它会强制关闭那些在这宽限期间仍然保持活动的 I/O 会话。这是一个相当复杂的过程，很多事情可能会在相同时间失败，而且很多不同的异常可能在关闭处理中抛出。I/O 反应器将会记录关闭处理过程中所有抛出的异常，包括引发关闭的最初的，并记录在审计日志中。我们可以检查审计日志来决定是否可以安全地重启 I/O 反应器。

```
DefaultConnectingIOReactor ioreactor;
// 给5秒的宽限期
ioreactor.shutdown(5000);
List<ExceptionEvent> events = ioreactor.getAuditLog();
for (ExceptionEvent event: events) {
   System.err.println("Time: " + event.getTimestamp());
   event.getCause().printStackTrace();
}
```

2.5 非阻塞的 HTTP 连接

有效的非阻塞 HTTP 连接是使用了 HTTP 指定功能的围绕 IOSession 的包装器。非阻塞的 HTTP 连接是有状态的，而不是线程安全的。非阻塞 HTTP 连接之上的输入/输出操作应该被限制去派发由 I/O 事件分发进程触发的事件。

2.5.1 非阻塞 HTTP 连接的执行上下文

非阻塞 HTTP 连接没有绑定到特定线程的执行，因此它们需要来维护它们自己的执行上下文。每个非阻塞 HTTP 连接都有一个 HttpContext 实例和它相关，这可以用来维护处理状态。HttpContext 实例是线程安全的，可以从多线程中来操作。

```
// 获取非阻塞的HTTP连接
DefaultNHttpClientConnection conn;
// 状态
Object myStateObject;
HttpContext context = conn.getContext();
context.setAttribute("state", myStateObject);
```

2.5.2 使用非阻塞的 HTTP 连接

在任何时间点我们都能获取目前被转义到非阻塞 HTTP 连接的请求和响应对象。这两个对象中的任何一个，或者两者，如果没有到来或当前被转义要发送的报文，那么它们可以为空。

```
NHttpConnection conn;
HttpRequest request = conn.getHttpRequest();
if (request != null) {
   System.out.println("Transferring request: " +
   request.getRequestLine());
}
HttpResponse response = conn.getHttpResponse();
if (response != null) {
   System.out.println("Transferring response: " +
   response.getStatusLine());
}
```

请注意当前的请求和当前的响应不一定代表相同的报文交换！非阻塞 HTTP 连接可以在全双工模式下操作。我们可以处理收到和发出报文彼此间完全独立。这使得非阻塞 HTTP 连接完全拥有流水线能力，但是在同一时刻意味着这是协议处理程序的工作，来匹配逻辑相关的请求和响应报文。

在简化提交客户端的请求的处理可以像这样进行：

```
// 获得HTTP连接
NHttpClientConnection conn;
// 获得执行上下文
HttpContext context = conn.getContext();
// 获得处理状态
Object state = context.getAttribute("state");
// 生成基于状态信息的请求
HttpRequest request = new BasicHttpRequest("GET", "/");
conn.submitRequest(request);
System.out.println(conn.isRequestSubmitted());
```

在简化提交服务器端响应的处理可以像这样进行：

```
// 获得HTTP连接
NHttpServerConnection conn;
// 获得执行上下文
HttpContext context = conn.getContext();
// 获得处理状态
Object state = context.getAttribute("state");
// 生成基于状态信息的响应
HttpResponse response = new
BasicHttpResponse(HttpVersion.HTTP_1_1,
HttpStatus.SC_OK, "OK");
BasicHttpEntity entity = new BasicHttpEntity();
entity.setContentType("text/plain");
entity.setChunked(true);
response.setEntity(entity);
conn.submitResponse(response);
System.out.println(conn.isResponseSubmitted());
```

请注意我们应该很少需要来使用这些低级别方法来发送报文，相反，应该使用合适的高级 HTTP 服务实现。

2.5.3 HTTP I/O 控制

所有非阻塞 HTTP 连接类实现了 IOControl 接口，这代表了在连接控制功能子集的 I/O 甚至是通知。IOControl 实例期望完全的线程安全。因此 IOControl 可以用来从其它线程中请求/暂停 I/O 事件通知。

当和非阻塞连接交互时，我们必须采取特别的预防措施。HttpRequest 和 HttpResponse 是线程不安全的。一般建议所有在非阻塞连接上的输入/输出操作从 I/O 事件分发线程中执行。

建议使用下列模式：

- 使用 IOControl 接口来传递连接的 I/O 事件到另外一个线程/会话之上的控制。
- 如果输入/输出操作需要在特定的连接之上执行，在连接上下文中存储所有需要的信息（状态），而且通过调用 IOControl#requestInput() 或

IOControl#requestOutput()方法来请求适当的I/O操作。

- 从分发线程中的事件方法中执行需要的操作，使用存储在连接上下文中的信息。

请注意所有发生在事件方法中的操作不应该阻塞太长事件，因为之后分发线程在会话中依然是阻塞的，它不能对其它的会话处理事件。使用会话底层通道的I/O操作不是问题，因为它们保证是非阻塞的。

2.5.4 非阻塞内容传输

对于非阻塞连接的内容传输处理的工作方式和阻塞连接相比是完全不同的，因为非阻塞连接需要容纳NIO模型的异步性质。两种连接类型的主要区别是无力使用普通的，但本质阻塞的java.io.InputStream和java.io.OutputStream类来代表输入和输出内容的流。HttpCore NIO提供ContentEncoder和ContentDecoder接口来处理一步内容传输的过程。非阻塞HTTP连接将会实例化适合的基于包含在报文实体属性内容解编码器实现。

非阻塞HTTP连接将触发输入事件，直到内容实体完全传输。

```
//获得内容解码器
ContentDecoder decoder;
//读入数据
ByteBuffer dst = ByteBuffer.allocate(2048);
decoder.read(dst);
// 当内容实体完全传输时，解码会标记为成功
if (decoder.isCompleted()) {
    // 完成
}
```

非阻塞HTTP连接将会触发输出事件，直到内容实体被标记为完全传输。

```
// 获得内容编码器
ContentEncoder encoder;
// 准备输出数据
ByteBuffer src = ByteBuffer.allocate(2048);
// 将输出写出
encoder.write(src);
// 当完成时标记内容实体为完全传输
encoder.complete();
}
```

请注意，当提交包含报文的实体到非阻塞HTTP连接时，我们仍然需要提供HttpEntity实例。实体的属性将被用来初始化ContentEncoder实例用于转换实体内容。非阻塞HTTP连接，忽略被包含实体的本质阻塞HttpEntity#getContent()和HttpEntity#writeTo()方法。

```
// 获得HTTP连接
NHttpServerConnection conn;
HttpResponse response = new
BasicHttpResponse(HttpVersion.HTTP_1_1,
HttpStatus.SC_OK, "OK");
BasicHttpEntity entity = new BasicHttpEntity();
entity.setContentType("text/plain");
entity.setChunked(true);
entity.setContent(null);
response.setEntity(entity);
conn.submitResponse(response);
```

同样地，收到包含报文的实体会有一个和它们相关的 HttpEntity 实例，但试图调用 HttpEntity#getContent() 或 HttpEntity#writeTo() 方法将会引起 java.lang.IllegalStateException 异常。HttpEntity 实例可以用于决定收到实体的属性，比如内容长度。

```
// 获得HTTP连接
NHttpClientConnection conn;
HttpResponse response = conn.getHttpResponse();
HttpEntity entity = response.getEntity();
if (entity != null) {
    System.out.println(entity.getContentType());
    System.out.println(entity.getContentLength());
    System.out.println(entity.isChunked());
}
```

2.5.5 支持的非阻塞内容转换机制

默认的非阻塞 HTTP 连接接口的实现支持由 HTTP/1.1 规范定义的三种内容转换机制：

- **Content-Length 分隔**：内容实体的末尾由 Content-Length 头部的值来分隔。最大实体长度：Long#MAX_VALUE。
- **身份编码**：内容实体的末尾由关闭底层连接（流条件结束）来划定。由于显而易见的原因，身份编码仅仅能用于服务器端。最大实体长度：不限制。
- **块编码**：内容以小块来发送。最大实体长度：不限制。

适当的内容解编码器将会依靠被包含在报文中的实体属性来自动创建。

2.5.6 直接通道 I/O

内容代码对直接从底层 I/O 会话通道读取数据和直接写入数据到底层 I/O 会话通道进行了优化，尽可能避免在会话缓冲区的中间缓冲。此外，那些解编码器不执行任何内容转换，比如 Content-Length 分隔和身份可以利用 NIO java.nio.FileChannel 的方法来大幅提高文件转换操作性能，包括输入和输出。

如果实际内容解码器实现 FileContentDecoder，我们可以使用它的方法来直接读取输出的内容通过传递中间的 java.nio.ByteBuffer 到文件中。

```
//获取内容解码器
ContentDecoder decoder;
//准备文件通道
FileChannel dst;
//如果可能，直接使用文件I/O
if (decoder instanceof FileContentDecoder) {
    long Bytesread = ((FileContentDecoder) decoder)
    .transfer(dst, 0, 2048);
    // 当内容实体完全转换时解码就会被标记为完成
    if (decoder.isCompleted()) {
        // 完成
    }
}
```

如果实际内容编码器实现 FileContentEncoder，我们可以使用它的方法来通过传递中间 java.nio.ByteBuffer 直接从文件中写输出内容。

```
// 获得内容编码器
ContentEncoder encoder;
// 准备文件通道
FileChannel src;
//如果可能，直接使用文件I/O
if (encoder instanceof FileContentEncoder) {
    // 将数据写出
    long bytesWritten = ((FileContentEncoder) encoder)
    .transfer(src, 0, 2048);
    // 当完成时标记内容实体为完全转换
    encoder.complete();
}
```

2.6 HTTP 1/O 事件分发器

HTTP I/O 事件分发器服务于由 HTTP 协议指定事件的 I/O 反应器转换通用 I/O 事件触发器。它们依赖于 NHttpClientHandler 和 NHttpServiceHandler 接口来传播 HTTP 协议事件到 HTTP 协议控制程序。

服务器端 HTTP 1/O 事件由 NHttpServiceHandler 接口定义：

- **connected**：当新的传入连接被创建时触发。
- **requestReceived**：当新的 HTTP 请求收到时触发。传递的连接作为一个参数给方法来保证返回有效的 HTTP 请求对象。如果收到的请求包含请求实体，这个方法会被一系列的 inputReady 事件跟着来转换请求内容。
- **inputReady**：当底层通道准备通过对应的内容解码器读取新的请求实体部分时触发。如果内容消费者不能处理收到的内容，那么输入事件通知可以使用

IOControl 接口暂时停止。

- **responseReady**：当连接准备接受新的 HTTP 响应时触发。如果没有准备好，如果没有准备好，协议处理程序不需要提交响应。
- **outputReady**：当底层通道准备通过对应的内容编码器写新的响应实体部分时触发。如果内容生产者不能生成要发出的内容，输出事件通知可以使用 IOControl 接口暂时停止。
- **exception**：当 I/O 错误发生而从底层通道读取或写入底层通道，或当 HTTP 协议违规发生而接收 HTTP 请求时触发。
- **timeout**：当这个连接超过最大活动期而没有侦测到输入时触发。
- **closed**：当连接关闭时触发。

客户端 I/O 事件由 NHttpClientHandler 接口定义：

- **connected**：当新的传出连接创建时触发。传递的作为参数给这个事件的附件对象是任意附加到会话请求中的对象。
- **requestReady**：当连接准备接受新的 HTTP 请求时触发。如果没有准备好，协议处理程序不需要提交请求。
- **outputReady**：当底层通道准备通过对应的内容编码器写请求实体的下一部分时触发。如果内容生产者不能生成要发出的内容，输出事件通知可以使用 IOControl 接口暂时停止。
- **responseReceived**：当 HTTP 响应收到时触发。当作参数传递给方法的连接要保证返回合法的 HTTP 响应对象。如果接收到的响应包含响应实体，这个方法会被一系列 inputReady 事件跟着来转换响应内容。
- **inputReady**：当底层通道准备通过对应的内容解码器读取新的请求实体部分时触发。如果内容消费者不能处理收到的内容，那么输入事件通知可以使用 IOControl 接口暂时停止。
- **exception**：当 I/O 错误发生而从底层通道读取或写入底层通道，或当 HTTP 协议违规发生而接收 HTTP 请求时触发。
- **timeout**：当这个连接超过最大活动期而没有侦测到输入时触发。
- **closed**：当连接关闭时触发。

2.7 非阻塞 HTTP 实体

如之前讨论的，非阻塞连接的内容传输过程和阻塞连接相比是完全不同的。出于显而易见的原因，经典 I/O 抽象化是基于本质阻塞的 java.io.InputStream 和 java.io.OutputStream 类，它们不能应用于数据传输的异步处理。因此，非阻塞的 HTTP 实体提供 NIO 对 HttpEntity 接口特定的扩展：ProducingNHttpEntity 和 ConsumingNHttpEntity 接口。如果特定的实现不能代表它的内容流作为的 java.io.InputStream 实例，或者不能将它内容流式读出到 java.io.OutputStream 中，这些接口的实现类可能从 HttpEntity#getContent() 或 HttpEntity#writeTo() 方法抛出 java.lang.UnsupportedOperationException 异常。

2.7.1 内容消耗非阻塞 HTTP 实体

ConsumingNHttpEntity 接口代表允许内容从内容解码器中被消耗的非阻塞实体。ConsumingNHttpEntity 使用了 NIO 特定的通知方法扩展了基础的 HttpEntity 接口：

- **consumeContent**：通知内容可以从解码器中读取。IOControl 实例作为参数传递给方法，如果实体暂时不能分配更多的存储空间来容纳所有收到的内容，可以用来暂停输入事件。
- **finish**：通知任意被分配用来做读取操作的资源被释放。

下列 ConsumingNHttpEntity 的实现由 HttpCore NIO 提供：

- BufferingNHttpEntity
- ConsumingNHttpEntityTemplate

2.7.1.1 BufferingNHttpEntity

BufferingNHttpEntity 是 HttpEntityWrapper 的子类,它可以消耗所有收到的内容到内存中。一旦内容体被全部收到，它可以通过 HttpEntity#getContent()方法以 java.io.InputStream 来获取，或者通过 HttpEntity#writeTo()方法被写入到一个输出流中。

2.7.1.2 ConsumingNHttpEntityTemplate

ConsumingNHttpEntityTemplate 是 HttpEntityWrapper 的子类，用来装饰传入的 HTTP 实体，代理处理传入的内容到 ContentListener 实例中。

```
static class FileWriteListener implements ContentListener {
   private final FileChannel fileChannel;
   private long idx = 0;
   public FileWriteListener(File file) throws IOException {
      this.fileChannel = new
      FileInputStream(file).getChannel();
   }
   public void contentAvailable(ContentDecoder decoder,
IOControl ioctrl)throws IOException {
      long transferred;
      if (decoder instanceof FileContentDecoder) {
         transferred = ((FileContentDecoder) decoder).
         transfer(fileChannel, idx, Long.MAX_VALUE);
      } else {
         transferred = fileChannel.transferFrom(new
      ContentDecoderChannel(decoder),idx, Long.MAX_VALUE);
      }
      if (transferred > 0) {
         idx += transferred;
      }
   }
```

```
    public void finished() {
        try {
            fileChannel.close();
        } catch(IOException ignored) {}
    }
}
HttpEntity incomingEntity;
File file = new File("buffer.bin");
ConsumingNHttpEntity entity = new
ConsumingNHttpEntityTemplate(incomingEntity,
new FileWriteListener(file));
```

2.7.2 内容生成非阻塞 HTTP 实体

ProducingNHttpEntity 接口代表非阻塞实体，它允许内容被写入到内容编码器中。ProducingNHttpEntity 使用一组 NIO 特定的通知方法扩展了基础的 HttpEntity 接口：

- **produceContent**：通知内容可以被写入到编码器中。如果实体不能用来生产更多内容，作为参数传递给方法的 IOControl 实例可以用来暂停输出事件。请注意我们必须调用 ContentEncoder#complete()方法来通知底层连接所有内用已经写完了。如果不这么做可能导致实体不能正确被分隔。
- **finish**：通知任意为写操作分配的资源被释放。

下列的 ProducingNHttpEntity 实现由 HttpCore NIO 提供：

- NByteArrayEntity
- NStringEntity
- NFileEntity

2.7.2.1 NByteArrayEntity

这是一个简单自我包含的可重复实体，从给定的字节数组中获取内容。字节数组提供给构造方法。

```
String myData = "Hello world on the other side!!";
NByteArrayEntity entity = new
NByteArrayEntity(myData.getBytes());
```

2.7.2.2 NStringEntity

这是一个简单的，自我包含的，可重复的实体，它可以从 java.lang.String 对象中获取数据。它有 2 个构造方法，一个简单使用给定的字符串来构造，而另外一个需要对 java.lang.String 中的数据使用字符编码。

```
String myData = "Hello world on the other side!!";
// 不使用字符编码来构造
NStringEntity myEntity1 = new NStringEntity(myData);
// 使用编码来构造
NStringEntity myEntity2 = new NStringEntity(myData, "UTF-8");
```

2.7.2.3 NFileEntity

这个实体从文件中读取内容体。这个类主要使用流式用于不同类型的大文件，所以我们需要提供文件的内容类型来确保内容可以被正确识别并被接受者处理。

```
File staticFile = new File("/path/to/myapp.jar");
NHttpEntity entity = new NFileEntity(staticFile,
"application/java-archive");
```

NHttpEntity 会尽可能使用直接通道 I/O，提供内容编码器能够直接从文件到底层连接的套接字中转换数据。

2.8 非阻塞 HTTP 协议处理程序

2.8.1 异步的 HTTP 服务处理程序

AsyncNHttpServiceHandler 是完全的异步 HTTP 服务器端协议处理程序，它实现了 RFC 2616 中描述的 HTTP 协议对服务器端报文处理的基本要求。AsyncNHttpServiceHandler 能够以几乎不变的内存占用为独立的 HTTP 连接处理 HTTP 请求。这个处理程序在内存中存储 HTTP 报文的头部信息，而报文体的内容使用 ConsumingNHttpEntity 和 ProducingNHttpEntity 接口直接从实体中流式读入底层通道（反之亦然）。

当使用这个实现时，保证为编写实现的 ProducingNHttpEntity 提供实体是很重要的。这么做允许实体异步地写出。如果为编写提供的实体没有实现 ProducingNHttpEntity 接口，将会加入代理，在内存中缓冲全部内容。此外，缓冲区可能占用 I/O 分发线程，这可能会引起暂时的 I/O 阻塞。对于最好的结果，我们必须保证从 NHttpRequestHandler 中所有 HTTP 响应的实体集实现 ProducingNHttpEntity。

如果传入的实体包含内容实体，NHttpRequestHandler 实例期望返回 ConsumingNHttpEntity 来读取内容。在实体完成数据读取之后，NHttpRequestHandler#handle() 方法被调用来生成响应。

AsyncNHttpServiceHandler 依赖于 HttpProcessor 来为输出报文生成强制性协议报头，对所有收到的哈发送的报文应用通用的，交叉消息转换，而独立的 HTTP 请求处理程序期望关注特定内容生成和处理的应用程序。

```
HttpParams params;
// 初始化HTTP参数
HttpProcessor httpproc;
// 初始话HTTP处理器
AsyncNHttpServiceHandler handler = new
AsyncNHttpServiceHandler(
httpproc,new DefaultHttpResponseFactory(),
new DefaultConnectionReuseStrategy(),params);
```

2.8.1.1 非阻塞 HTTP 请求处理程序

NHttpRequestHandler 接口代表了处理特定非阻塞 HTTP 请求组的规则。NHttpRequestHandler 实现期望关注协议特定的方面，而独立的请求处理程序期望关注应用程序特定的 HTTP 处理。请求处理程序的主要目的是使用内容实体生成响应对象，并发回客户端来详情给定的请求。

```
NHttpRequestHandler myRequestHandler = new
NHttpRequestHandler() {
   public ConsumingNHttpEntity entityRequest(
      HttpEntityEnclosingRequest request,
   HttpContext context) throws HttpException, IOException {
      // 在内存中简单缓冲收到的内容
      return new BufferingNHttpEntity(request.getEntity(),
      new HeapByteBufferAllocator());
   }
   public void handle(HttpRequest request,
      HttpResponse response,NHttpResponseTrigger trigger,
   HttpContext context) throws HttpException, IOException {
      response.setStatusCode(HttpStatus.SC_OK);
      response.addHeader("Content-Type", "text/plain");
      response.setEntity(
         new NStringEntity("some important message"));
      // 为简单起见，立即提交响应
      trigger.submitResponse(response);
   }
};
```

请求处理程序必须实现为线程安全的方式。和 Servlet 相似，请求处理程序不应该使用实例变量，除非访问那些同步的变量。

2.8.1.2 异步的响应触发器

和阻塞组件相比，非阻塞请求处理程序的最根本的区别是推迟 HTTP 响应发回客户端，

而不需要通过代理 HTTP 请求到工作线程的处理过程的阻塞 I/O 线程的传输能力。工作线程可以使用 NHttpResponseTrigger 实例并作为参数传递给 NHttpRequestHandler#handle 方法来提交响应，如一旦响应可用的稍后的时间点。

```
NHttpRequestHandler myRequestHandler = new
NHttpRequestHandler() {
   public ConsumingNHttpEntity entityRequest(
HttpEntityEnclosingRequest request,
   HttpContext context) throws HttpException, IOException {
   // 为简单起见，在内存中缓冲收到的内容
      return new BufferingNHttpEntity(request.getEntity(),
      new HeapByteBufferAllocator());
   }
   public void handle(HttpRequest request,
   HttpResponse response,NHttpResponseTrigger trigger,
   HttpContext context)throws HttpException, IOException {
      new Thread() {
         @Override
         public void run() {
            try {
               Thread.sleep(10);
            }
            catch(InterruptedException ie) {}
            try {
         URI uri = new URI(request.getRequestLine().getUri());
         response.setStatusCode(HttpStatus.SC_OK);
         response.addHeader("Content-Type", "text/plain");
         response.setEntity(
            new NStringEntity("some important message"));
         trigger.submitResponse(response);
         } catch(URISyntaxException ex) {
            trigger.handleException(
            new HttpException("Invalid request URI: " +
               ex.getInput()));
         }
      }
   }.start();
}
};
```

请注意 HttpResponse 对象是线程不安全的，所以不能同时进行修改。非阻塞请求处理程序必须保证 HTTP 响应不能被多余一个线程同时访问。

2.8.1.3 非阻塞请求处理程序解析器

和阻塞 HTTP 请求处理程序相比，非阻塞 HTTP 请求处理程序的管理非常简单。通常一个 NHttpRequestHandlerResolver 实例可以用来维护请求处理程序注册，并可以匹配请求 URI 到特定的请求处理程序。HttpCore 仅包含一个请求处理程序解析器简单实现，它是基于简单模式的匹配算法：NHttpRequestHandlerRegistry 仅支持三种格式：*，<uri>*和*<uri>。

```
// 初始化异步协议处理程序
AsyncNHttpServiceHandler handler;
NHttpRequestHandlerRegistry handlerResolver =
new NHttpRequestHandlerRegistry();
handlerReqistry.register("/service/*", myRequestHandler1);
handlerReqistry.register("*.do", myRequestHandler2);
handlerReqistry.register("*", myRequestHandler3);
handler.setHandlerResolver(handlerResolver);
```

鼓励用户提供 NHttpRequestHandlerResolver 的更复杂实现，比如，基于正则表达式的实现。

2.8.2 异步的 HTTP 客户端处理程序

AsyncNHttpClientHandler 是完全的异步 HTTP 客户端协议处理程序，它实现了 RFC 2616 中描述的 HTTP 协议对客户端报文处理的基本要求。AsyncNHttpClientHandler 能够以几乎不变的内存占用为独立的 HTTP 连接处理 HTTP 请求。这个处理程序在内存中存储 HTTP 报文的头部信息，而报文体的内容使用 ConsumingNHttpEntity 和 ProducingNHttpEntity 接口直接从实体中流式读入底层通道（反之亦然）。

当使用这个实现时，保证为编写实现的 ProducingNHttpEntity 提供实体是很重要的。这么做允许实体异步地写出。如果为编写提供的实体没有实现 ProducingNHttpEntity 接口，将会加入代理，在内存中缓冲全部内容。此外，缓冲区可能占用 I/O 分发线程，这可能会引起暂时的 I/O 阻塞。对于最好的结果，我们必须保证从 NHttpRequestHandler 中所有 HTTP 响应的实体集实现 ProducingNHttpEntity。

如果传入的实体包含内容实体，NHttpRequestExecutionHandler 实例期望从内容中返回 ConsumingNHttpEntity。在实体完成数据读取之后，NHttpRequestExecutionHandler#handleResponse() 方法被调用来处理响应。

AsyncNHttpClientHandler 依赖于 HttpProcessor 来为输出报文生成强制性协议报头，对所有收到的哈发送的报文应用通用的，交叉消息转换，而 HTTP 请求处理器希望关注应用程序特定内容的生成和处理。

```
// 初始化HTTP参数
HttpParams params;
//初始化HTTP处理器
HttpProcessor httpproc;
//创建HTTP请求执行处理程序
NHttpRequestExecutionHandler execHandler;
AsyncNHttpClientHandler handler = new AsyncNHttpClientHandler(
httpproc,execHandler,new DefaultConnectionReuseStrategy(),
params);
```

2.8.2.1 异步的 HTPP 请求执行处理程序

异步的 HTTP 请求执行处理程序可以由客户端协议处理程序来使用，触发心得 HTTP 请求的提交和 HTTP 响应的处理。

HTTP 请求执行事件由 NHttpRequestExecutionHandler 接口定义：

- **initalizeContext**：当新的连接建立和 HTTP 上下文需要被初始化时触发。传递给这个方法的附件对象是和当连接请求生成时传递给连接 I/O 反应器的是相同的对象。附件可能包含一些需要的状态信息来正确初始化 HTTP 上下文。
- **submitRequest**：当底层连接准备发送新的 HTTP 请求到目标主机时触发。如果客户端没有准备好去发送请求，这个方法可能返回 null。这种情况下连接会保持打开并在以后激活。如果连接包含实体，实体必须是 ProducingNHttpEntity 实例。
- **responseEntity**：当收到包含实体的响应时触发。这个方法应该返回 ConsumingNHttpEntity，它会被用来消耗实体。Null 是合法的响应值，并暗示这个实体应该被忽略。在实体被完全消耗后，handleResponse 方法被调用通知全面响应，被包含的实体也准备好处理。
- **handleResponse**：当 HTTP 响应准备去处理时触发。
- **finalizeContext**：当连接被终止时触发。这个事件可以用来释放存储在上下文中的对象，或者做一些清洁工作。

```
NHttpRequestExecutionHandler execHandler =
   new NHttpRequestExecutionHandler() {
      private final static String DONE_FLAG = "done";
      public void initalizeContext(
         HttpContext context,Object attachment) {
         if (attachment != null) {
            HttpHost virtualHost = (HttpHost) attachment;
            context.setAttribute(ExecutionContext.HTTP_TARG
         ET_HOST,virtualHost);
         }
      }
      public void finalizeContext(HttpContext context) {
         context.removeAttribute(DONE_FLAG);
      }
```

```
        public HttpRequest submitRequest(HttpContext context) {
            // 提交一次HTTP GET
            Object done = context.getAttribute(DONE_FLAG);
            if (done == null) {
                context.setAttribute(DONE_FLAG, Boolean.TRUE);
                return new BasicHttpRequest("GET", "/");
            } else {
                return null;
            }
        }
        public ConsumingNHttpEntity responseEntity(
        HttpResponse response,
        HttpContext context) throws IOException {
            // 简单在内存中缓冲收到的内容
        return new BufferingNHttpEntity(response.getEntity(),
        new HeapByteBufferAllocator());
    }
    public void handleResponse(HttpResponse response,
        HttpContext context) throws IOException {
        System.out.println(response.getStatusLine());
        if (response.getEntity() != null) {
            System.out.println(
            EntityUtils.toString(response.getEntity()));
        }
    }
};
```

2.8.3 兼容阻塞 I/O

除了上面描述的异步协议处理程序，HttpCore 附带两个 HTTP 协议处理程序的变种，在非阻塞对象头部仿真阻塞 I/O 模型，允许生成报文内容，使用标准的 java.io.OutputStream / java.io.InputStream API 来消耗。兼容协议处理程序可以使用 HTTP 请求处理程序和请求执行器，它依赖于阻塞的 HttpEntity 实现。

兼容协议处理程序依赖 HttpProcessor 来对发出的报文生成强制的协议头部，并对所有收到的和发出的报文应用通用的，交叉报文转换，而独立的 HTTP 请求执行器/HTTP 请求处理器期望关心应用程序指定的内容生成和处理。

2.8.3.1 缓冲协议处理程序

BufferingHttpServiceHandler 和 BufferingHttpClientHandler 是协议处理程序的实现，它们在内存中存储完整的 HTTP 报文内容提供对阻塞 I/O 的兼容。请求/响应处理回调仅当完整报文内容被读取到内存缓冲区时进行。请注意请求执行/请求处理占

用主要 I/O 线程，因此独立 HTTP 请求执行器/请求处理程序必须保证它们不会无限期的阻塞。

缓冲协议处理程序应该用于当处理 HTTP 报文被限制长度时。

2.8.3.2 节流协议处理程序

`ThrottlingHttpServiceHandler` 和 `ThrottlingHttpClientHandler` 是协议处理程序的实现，它们利用共享内容缓冲和一个小的工作线程池提供阻塞 I/O 模型的支持。节流协议处理程序在初始化之后分配固定长度输入/输出缓冲，并控制 I/O 事件率来保证那些内容缓冲不会溢出。这帮助保证了当收到报文时 HTTP 连接立即回调的不变内存占用。节流协议处理程序代理请求处理和响应内容生成任务到执行器中，期望使用专用工作线程执行那些任务来避免阻塞 I/O 线程。

通常情况下，节流协议处理程序仅仅需要适度的工作线程数，远少于并发连接的数量。如果报文的长度较小，或对共享内容的缓冲区大小工作线程仅存储内容在缓冲中，并几乎立即终止而不阻塞。反过来，I/O 分发线程将会关注发出的异步缓冲内容。工作线程当处理大报文和共享缓冲填满时不得不阻塞。通常情况下，建议分配共享缓冲的大小是最佳性能下内容体大小的平均值。

2.8.4 连接事件监听器

协议处理程序就像其它类别的 HttpCore 类，不进行日志记录，为了不对用户强加日志框架的选择。而我们可以对重要的连接添加日志，注入 `EventListener` 实现到协议处理程序中即可。

由 `EventListener` 接口定义的连接事件：

- **`fatalIOException`**：当 I/O 错误引起连接终止时触发。
- **`fatalProtocolException`**：当 HTTP 协议错误引起连接终止时触发。
- **`connectionOpen`**：当新的连接建立时触发。
- **`connectionClosed`**：当连接终止时触发。
- **`connectionTimeout`**：当连接超时时触发。

2.9 非阻塞的 TLS/SSL

2.9.1 SSL I/O 会话

`SSLIOSession` 是一个装饰器类，为了透明地用传输层安全功能来扩展任意的基于 SSL/TLS 协议的 `IOSession`。独立的协议处理程序应该可以和 SSL 会话一起工作，而不需要特定的先决条件或修改。而 I/O 分发器需要使用一些额外的动作来保证运输层加密的正确功能。

- 当底层 I/O 会话被创建时，I/O 分发必须调用 `SSLIOSession#bind()` 方法来将 SSL 会话放置到客户端或服务器端模式。
- 当底层 I/O 会话准备输入时，I/O 分发器应该检查 SSL I/O 会话是否准备调用 `SSLIOSession#isAppInputReady()` 方法来处理输入数据，如果是的话，传

递控制和协议处理程序，最终调用 SSLIOSession#inboundTransport()方法来做一些 SSL 握手和加密输入数据的必要操作。

- 当底层 I/O 会话准备输出时，I/O 分发器应该检查 SSL I/O 会话是否准备调用 SSLIOSession#isAppOutputReady()方法接收输出数据，如果是的话，传递控制给协议处理程序，最终调用 SSLIOSession#outboundTransport()方法来做一些 SSL 握手和加密应用程序数据的必要操作。

2.9.1.1 SSL I/O 会话处理程序

应用程序可以通过传递自定义实现的 SSLIOSessionHandler 接口来自定义很多 TLS/SSL 协议的方面内容。

SSL 事件由 SSLIOSessionHandler 接口来定义：

- **initalize**：当 SSL 连接建立时触发。处理程序可以使用这个回调来自定义 javax.net.ssl.SSLEngine 的属性来建立 SSL 连接。
- **verify**：当 SSL 连接建立和初始的 SSL 握手成功完成时触发。这个处理程序可以使用回调来验证 SSL 会话的属性。比如这会是实施 SSL 加密强度的地方，验证证书链并做些主机名检查。

```
// 获取新的I/O会话
IOSession iosession;
// 初始化默认SSL上下文
SSLContext sslcontext = SSLContext.getInstance("SSL");
sslcontext.init(null, null, null);
SSLIOSession sslsession = new SSLIOSession(
   iosession, sslcontext, new SSLIOSessionHandler() {
      public void initalize(SSLEngine sslengine,
      HttpParams params) throws SSLException {
      // 询问客户端来验证
      sslengine.setWantClientAuth(true);
      // 密码加强
      sslengine.setEnabledCipherSuites(new String[] {
      "TLS_RSA_WITH_AES_256_CBC_SHA",
      "TLS_DHE_RSA_WITH_AES_256_CBC_SHA",
      "TLS_DHE_DSS_WITH_AES_256_CBC_SHA" });
   }
   public void verify(SocketAddress remoteAddress,
      SSLSession session) throws SSLException {
         X509Certificate[] certs =
      session.getPeerCertificateChain();
      // 检查同等的证书链
      for (X509Certificate cert: certs) {
         System.out.println(cert.toString());
      }
   }
});
```

2.9.2 SSL I/O 事件分发

HttpCore 提供 SSLClientIOEventDispatch 和 SSLServerIOEventDispatch I/O 分发实现，它可以用来由任意 I/O 反应器开启 SSL 连接管理。这个分发使用所有必须的动作使用 SSL I/O 会话装饰器来包装激活 I/O 会话并确保 SSL 协议握手的正确处理。

第三章 高级主题

3.1 HTTP 报文解析和格式化框架

HTTP 报文处理框架被设计得很有表现力和灵活性，而剩余内存也很有效，快速。HttpCore 的 HTTP 报文处理代码对解析和格式化操作基本是达到零垃圾和零缓冲拷贝。相同的 HTTP 报文解析和格式化 API，还有实现使用了阻塞和非阻塞的传输实现，这帮助保证了 HTTP 服务而不管 I/O 模型的一致性行为。

3.1.1 HTTP 的行解析和格式化

HttpCore 使用一组低级别的组件来用于对所有行解析和格式化方法。

CharArrayBuffer 代表字符的序列，通常是 HTTP 报文流中的单独一行，比如请求行，状态行或头部信息。在内部，CharArrayBuffer 的真实实现是字符数组，如果需要它可以扩充来容纳更多输入。CharArrayBuffer 也提供一组工具方法来操纵缓冲的内容，存储数据和检索数据的子集。

```
CharArrayBuffer buf = new CharArrayBuffer(64);
buf.append("header: data ");
int i = buf.indexOf(':');
String s = buf.substringTrimmed(i + 1, buf.length());
System.out.println(s);
System.out.println(s.length());
```

输出内容为：

```
data
4
```

ParserCursor代表了解析操作的上下文：这个边界限制了解析操作的范围还有当前解析操作期望开始的位置。

```
CharArrayBuffer buf = new CharArrayBuffer(64);
buf.append("header: data ");
int i = buf.indexOf(':');
ParserCursor cursor = new ParserCursor(0, buf.length());
cursor.updatePos(i + 1);
System.out.println(cursor);
```

输出内容为：

```
[0>7>14]
```

LineParser 是在 HTTP 报文的头部区域解析行的接口。它有解析请求行，状态行，或头部行的独立方法。要解析的行在内存中传递，解析器不需要依赖任何特定的 I/O 机制。

```
CharArrayBuffer buf = new CharArrayBuffer(64);
buf.append("HTTP/1.1 200");
ParserCursor cursor = new ParserCursor(0, buf.length());
LineParser parser = new BasicLineParser();
ProtocolVersion ver = parser.parseProtocolVersion(buf,
cursor);
System.out.println(ver);
System.out.println(buf.substringTrimmed(cursor.getPos(),
cursor.getUpperBound()));
```

输出内容为：

```
HTTP/1.1
200
```

（再看一段）

```
CharArrayBuffer buf = new CharArrayBuffer(64);
buf.append("HTTP/1.1 200 OK");
ParserCursor cursor = new ParserCursor(0, buf.length());
LineParser parser = new BasicLineParser();
StatusLine sl = parser.parseStatusLine(buf, cursor);
System.out.println(sl.getReasonPhrase());
```

输出内容为：

```
OK
```

LineFormatter 来格式化 HTTP 报文头部的元素。这是对 LineParser 的补充。对格式化请求行，状态行或头部行有独立的方法。

请注意格式化不包含尾部换行符序列 CR-LF。

```
CharArrayBuffer buf = new CharArrayBuffer(64);
LineFormatter formatter = new BasicLineFormatter();
formatter.formatRequestLine(buf,
new BasicRequestLine("GET", "/", HttpVersion.HTTP_1_1));
System.out.println(buf.toString());
formatter.formatHeader(buf,
new BasicHeader("Content-Type", "text/plain"));
System.out.println(buf.toString());
```

输出内容为：

```
GET / HTTP/1.1
Content-Type: text/plain
```

HeaderValueParser 是解析头部值到元素中的接口。

```
CharArrayBuffer buf = new CharArrayBuffer(64);
HeaderValueParser parser = new BasicHeaderValueParser();
buf.append("name1=value1; param1=p1, " +
"name2 = \"value2\", name3 = value3");
ParserCursor cursor = new ParserCursor(0, buf.length());
System.out.println(parser.parseHeaderElement(buf, cursor));
System.out.println(parser.parseHeaderElement(buf, cursor));
System.out.println(parser.parseHeaderElement(buf, cursor));
```

输出内容为：

```
name1=value1; param1=p1
name2=value2
name3=value3
```

HeaderValueFormatter 是格式化头部信息值元素的接口。这是对 HeaderValueParser 的补充。

```
CharArrayBuffer buf = new CharArrayBuffer(64);
HeaderValueFormatter formatter = new
BasicHeaderValueFormatter();
HeaderElement[] hes = new HeaderElement[] {
   new BasicHeaderElement("name1", "value1",
      new NameValuePair[] {
         new BasicNameValuePair("param1", "p1")} ),
         new BasicHeaderElement("name2", "value2"),
      new BasicHeaderElement("name3", "value3"),
   };
formatter.formatElements(buf, hes, true);
System.out.println(buf.toString());
```

输出内容为：

```
name1="value1"; param1="p1", name2="value2", name3="value3"
```

3.1.2 HTTP 报文流和会话 I/O 缓冲

HttpCore 为阻塞和非阻塞 I/O 模型提供一组工具类，它们可以便利的处理 HTTP 报文流，简化 HTTP 报文 CR-LF 分隔行和管理直接数据缓冲的处理。

HTTP 连接实现通常依赖会话输入/输出缓冲从 HTTP 报文流中来读取数据和写入数据到 HTTP 报文流中。会话输入/输出缓冲实现是特定的 I/O 模型，可以优化对阻塞和非阻塞的操作。

阻塞 HTTP 连接使用套接字绑定会话缓冲到传输数据。会话缓冲接口和 java.io.InputStream/java.io.OutputStream 类相似，但是它们也提供读取和写入 CR-LF 分隔换行方法。

```
Socket socket1;
Socket socket2;
HttpParams params = new BasicHttpParams();
SessionInputBuffer inbuffer = new SocketInputBuffer(
socket1, 4096, params);
SessionOutputBuffer outbuffer = new SocketOutputBuffer(
socket2, 4096, params);
CharArrayBuffer linebuf = new CharArrayBuffer(1024);
inbuffer.readLine(linebuf);
outbuffer.writeLine(linebuf);
```

非阻塞 HTTP 连接使用会话缓冲来优化从非阻塞 NIO 通道中读取数据和将数据写入非阻塞 NIO 通道。NIO 输入/输出会话帮助在非阻塞 I/O 模型中处理 CR-LF 分隔换行。

```
ReadableByteChannel channel1;
WritableByteChannel channel2;
HttpParams params = new BasicHttpParams();
SessionInputBuffer inbuffer = new SessionInputBufferImpl(
4096, 1024, params);
SessionOutputBuffer outbuffer = new SessionOutputBufferImpl(
4096, 1024, params);
CharArrayBuffer linebuf = new CharArrayBuffer(1024);
boolean endOfStream = false;
int bytesRead = inbuffer.fill(channel1);
if (bytesRead == -1) {
   endOfStream = true;
}
if (inbuffer.readLine(linebuf, endOfStream)) {
   outbuffer.writeLine(linebuf);
}
if (outbuffer.hasData()) {
   outbuffer.flush(channel2);
}
```

3.1.3 HTTP 报文解析器和格式化

HttpCore 提供接口粗粒度的门面类型来解析和格式化 HTTP 报文。那些接口的默认实现建立在由 SessionInputBuffer/SessionOutputBuffer 和 HttpLineParser/HttpLineFormatter 实现的功能之上。

HTTP 请求解析/写入阻塞 HTTP 连接的示例：

```
SessionInputBuffer inbuffer;
SessionOutputBuffer outbuffer;
HttpParams params = new BasicHttpParams();
HttpMessageParser requestParser = new HttpRequestParser(
inbuffer,new BasicLineParser(),
new DefaultHttpRequestFactory(),params);
HttpRequest request = (HttpRequest) requestParser.parse();
HttpMessageWriter requestWriter = new HttpRequestWriter(
outbuffer,new BasicLineFormatter(),params);
requestWriter.write(request);
```

HTTP 响应解析/写入阻塞 HTTP 连接的示例：

```
SessionInputBuffer inbuffer;
SessionOutputBuffer outbuffer;
HttpParams params = new BasicHttpParams();
HttpMessageParser responseParser = new HttpResponseParser(
inbuffer,new BasicLineParser(),
new DefaultHttpResponseFactory(),params);
HttpResponse response = (HttpResponse) responseParser.parse();
HttpMessageWriter responseWriter = new HttpResponseWriter(
outbuffer,new BasicLineFormatter(),params);
responseWriter.write(response);
```

HTTP 请求解析/写入非阻塞 HTTP 连接的示例：

```
SessionInputBuffer inbuffer;
SessionOutputBuffer outbuffer;
HttpParams params = new BasicHttpParams();
NHttpMessageParser requestParser = new HttpRequestParser(
inbuffer,new BasicLineParser(),
new DefaultHttpRequestFactory(),params);
HttpRequest request = (HttpRequest) requestParser.parse();
NHttpMessageWriter requestWriter = new HttpRequestWriter(
outbuffer,new BasicLineFormatter(),params);
requestWriter.write(request);
```

HTTP 响应解析/写入非阻塞 HTTP 连接的示例：

```
SessionInputBuffer inbuffer;
SessionOutputBuffer outbuffer;
HttpParams params = new BasicHttpParams();
NHttpMessageParser responseParser = new HttpResponseParser(
inbuffer,new BasicLineParser(),
new DefaultHttpResponseFactory(),params);
HttpResponse response = (HttpResponse) responseParser.parse();
NHttpMessageWriter responseWriter = new HttpResponseWriter(
outbuffer,new BasicLineFormatter(),params);
responseWriter.write(response);
```

3.1.4 HTTP 头部解析需求

默认的 HttpMessageParser 和 NHttpMessageParser 接口实现不立即解析 HTTP 头部信息。头部信息值的解析被推迟到它的属性被访问时。那些从不被应用程序使用的头部信息不会被解析。CharArrayBuffer 支持头可以通过可选的 FormattedHeader 接口来获得。

```
Header h1 = response.getFirstHeader("Content-Type");
if (h1 instanceof FormattedHeader) {
   CharArrayBuffer buf = ((FormattedHeader) h1).getBuffer();
   System.out.println(buf);
}
```

3.2 自定义 HTTP 连接

我们可以自定义 HTTP 连接解析和扩展默认实现，覆盖工厂方法，替换默认解析器或格式化实现格式化 HTTP 报文的方式。

对于阻塞 HTTP 连接，我们也可以提供自定义会话输入/输出缓冲的实现。

```
class MyDefaultNHttpClientConnection
extends DefaultNHttpClientConnection {
public MyDefaultNHttpClientConnection(
   IOSession session,HttpResponseFactory responseFactory,
   ByteBufferAllocator allocator,HttpParams params) {
      super(session, responseFactory, allocator, params);
   }
   @Override
   protected NHttpMessageWriter createRequestWriter(
   SessionOutputBuffer buffer,HttpParams params) {
      return new HttpRequestWriter(
      buffer, new BasicLineFormatter(), params);
   }
   @Override
   protected NHttpMessageParser createResponseParser(
   SessionInputBuffer buffer,HttpResponseFactory
   responseFactory,HttpParams params) {
      return new HttpResponseParser(
      buffer, new BasicLineParser(), responseFactory, params);
   }
};
```

对于非阻塞 HTTP 连接实现，我们可以替换默认的 HTTP 报文解析器和格式化的实现。会话输入/输出缓冲实现可以在 I/O 反应器级来覆盖。

```
class MyDefaultHttpClientConnection
extends DefaultHttpClientConnection {
   @Override
   protected SessionInputBuffer createSessionInputBuffer(
   Socket socket,int buffersize,
   HttpParams params) throws IOException {
      return new MySocketInputBuffer(socket, buffersize,
   params);
   }
   @Override
   protected SessionOutputBuffer createSessionOutputBuffer(
   Socket socket,int buffersize,
   HttpParams params) throws IOException {
      return new MySocketOutputBuffer(socket, buffersize,
   params);
   }
   @Override
   protected HttpMessageWriter createRequestWriter(
   SessionOutputBuffer buffer,HttpParams params) {
      return new MyHttpRequestWriter(
      buffer, new BasicLineFormatter(), params);
   }
   @Override
   protected HttpMessageParser createResponseParser(
   SessionInputBuffer buffer,
   HttpResponseFactory responseFactory,
   HttpParams params) {
      return new MyHttpResponseParser(
      buffer, new BasicLineParser(), responseFactory, params);
   }
};
```