2018

HTTPCORE 源码分析

2016K8009929009

秦 宏

写在前面

面向对象的源码分析文档，内容不定时更新，主要目标依然是为了完成课堂任务，同时有助于大型工程的源码阅读分析。原有的内容也可能会修改，以最新文档内容为准。

我的 github：https://github.com/qinhongUcaser

目录

一、HTTP 初步了解

（一）概念

超文本传输协议（英语：HyperText Transfer Protocol，缩写：HTTP）是一种用于分布式、协作式和超媒体信息系统的应用层协议[1]。HTTP 是万维网的数据通信的基础。

（二）目的

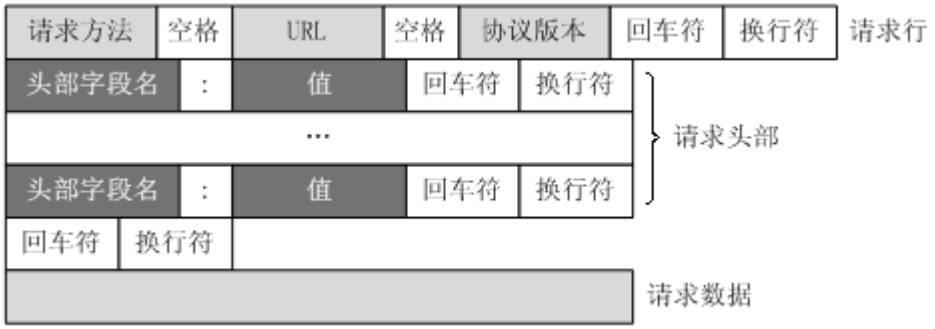
设计 HTTP 最初的目的是为了提供一种发布和接收 HTML 页面的方法。通过 HTTP 或者 HTTPS 协议请求的资源由统一资源标识符（Uniform Resource Identifiers，URI）来标识。

（三）详细内容

HTTP 是一个客户端终端（用户）和服务器端（网站）请求和应答的标准（TCP）。通过使用网页浏览器、网络爬虫或者其它的工具，客户端发起一个 HTTP 请求到服务器上指定端口（默认端口为 80）。我们称这个客户端为用户代理程序（user agent）。应答的服务器上存储着一些资源，比如 HTML 文件和图像。我们称这个应答服务器为源服务器（origin server）。在用户代理和源服务器中间可能存在多个“中间层”，比如代理服务器、网关或者隧道（tunnel）。

（四）消息(报文)格式

客户端发送一个 HTTP 请求到服务器的请求消息包括以下格式：请求行（request line）、请求头部（header）、空行和请求数据四个部分组成，下图给出了请求报文的一般格式。



服务器端返回的报文格式与上图类似，除了请求行变为状态行之外其他与上图大致相同。

（五）HTTP 实体

在执行带有附加内容的请求时或者请求成功并且使用响应主体将结果发送回客户端时，将创建实体。实体会在下面提到的八种方法的 POST 和 PUT 方法中传递的数据。实体作为消息传递时携带的“货物”，包含了实体头部和实体主体，存储在消息主体内。

实体的种类根据来源分为：

流实体：内容从流中接收，或在运行中生成。特别是，此类别包括从连接接收的实体。流实体通常不可重复。

自包含实体：内容在内存中或通过独立于连接或其他实体的方式获得。自包含实体通常是可重复的。

包装实体：内容从另一个实体中获得。

（六）八种方法

1.GET

向指定的资源发出“显示”请求。使用 GET 方法应该只用在读取数据，而不应当被用于产生“副作用”的操作中，例如在 Web Application 中。

2.HEAD

与 GET 方法一样，都是向服务器发出指定资源的请求。只不过服务器将不传回资源的本文部分。它的好处在于，使用这个方法可以在不必传输全部内容的情况下，就可以获取其中“关于该资源的信息”。

3.POST

向指定资源提交数据，请求服务器进行处理（例如提交表单或者上传文件）。数据被包含在请求本文中。这个请求可能会创建新的资源或修改现有资源，或二者皆有。

4.PUT

向指定资源位置上传其最新内容。

5.DELETE

6.TRACE

7.OPTIONS

8.CONNECT

（七）安全方法

对于 GET 和 HEAD 方法而言，除了进行获取资源信息外，这些请求不应当再有其他意义。也就是说，这些方法应当被认为是“安全的”。 客户端可能会使用其他“非安全”方法，例如 POST，PUT 及 DELETE，应该以特殊的方式（通常是按钮而不是超链接）告知客户可能的后果（例如一个按钮控制的资金交易），或请求的操作可能是不安全的（例如某个文件将被上传或删除）。

但是，不能想当然地认为服务器在处理某个 GET 请求时不会产生任何副作用。事实上，很多动态资源会把这作为其特性。这里重要的区别在于用户并没有请求这一副作用，因此不应由用户为这些副作用承担责任。

（八）关于 HTTPcore（摘自 HTTPcore 官网）

HttpCore 是一组底层 HTTP 传输组件，可用于以最小的占用空间构建自定义客户端和服务器端 HTTP 服务。HttpCore 支持两种 I / O 模型：基于经典 Java I / O 的阻塞 I / O 模型和基于 Java NIO 的非阻塞，事件驱动的 I / O 模型。

1. HTTPcore 的范围

构建客户端/代理/服务端的一致 API；构建同步和异步 HTTP 服务的一致 API；基于阻塞（经典）和非阻塞（NIO）I/O 模型的一组低级组件。

2. HTTP core 的目标

实现大多数的基本 HTTP 传输方面；平衡 API 的性能和清晰性和可表达性；低内存消耗(可预测)；独立的库(除 JRE 外，没有外部依赖)

3. HTTPcore 不是什么

不是 HttpClient 的替代品；不是 Servlet API 的替代品。

本次分析选取的版本为 HTTPcore 中 4.4.10 版本的源代码，具体包含了 HTTP 方法中的 GET 等通信方法。还未具体确定要选择的功能分析，代码结构仍然在学习阶段。

二、源代码文件结构解析

文件目录下一共有 4 个主要的文件夹，分别是 httpcore、httpcore-ab、httpcore-nio、httpcore-osgi。

（一）httpcore

该文件下存储了 4.4 版本下的 httpcore 的主要的核心代码。具体分析其中 main 文件夹下的文件内容。

Annotation 是注释文件夹；concurrent 是并发（使用未来模式）的文件夹；config 是配置文件夹；entity 是 http 消息的实体类文件夹；impl 是实现类文件夹；io 是输入输出类文件夹；message 是消息类文件夹；params 是参数文件夹，但是多被注释掉，不考虑使用；pool 的内容还不清楚；protocol 为协议处理器文件夹；ssl 为 HTTPS 以安全为目标建立的 HTTP 通道文件；util 包含集合框架、遗留的 collection 类、事件模型、日期和时间设施、国际化和各种实用工具类（字符串标记生成器、随机数生成器和位数组、日期 Date 类、堆栈 Stack 类、向量 Vector 类等）。集合类、时间处理模式、日期时间工具等各类常用工具包(摘自百度)。

另外有多个独立的 Java 文件表示部分具体操作行为，具体操作将在后面分析。

（二）httpcore-ab

该文件下存储的是用于测试 httpcore 执行的 benchmark。

（三）httpcore-nio

该文件下存储了 Java NIO 的详细代码，用于实现非阻塞驱动模型。

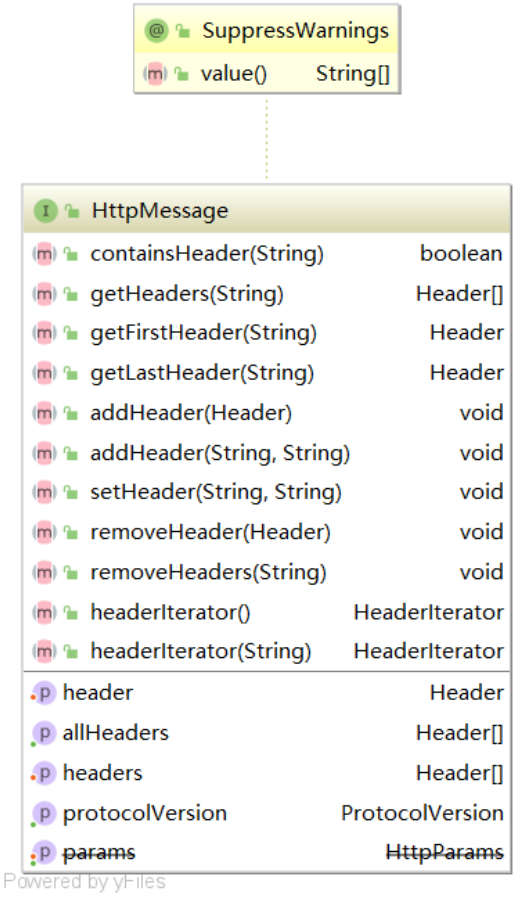
（四）httpcore-osgi

这里详细代码用途不清楚，看内容是存储了 OSGI 框架注释，用来区别一个叫做 IT 的变量。

三、代码具体功能解析

（一）以 request 和 response message 为例分析源码中的数据结构。

两种模式都连接了 HttpMessage 接口。可以看到定义的方法。



1.request message

如下图的一次 HTTP 请求消息(报文)，输入包含了使用消息的方法、标识符和协议版本。

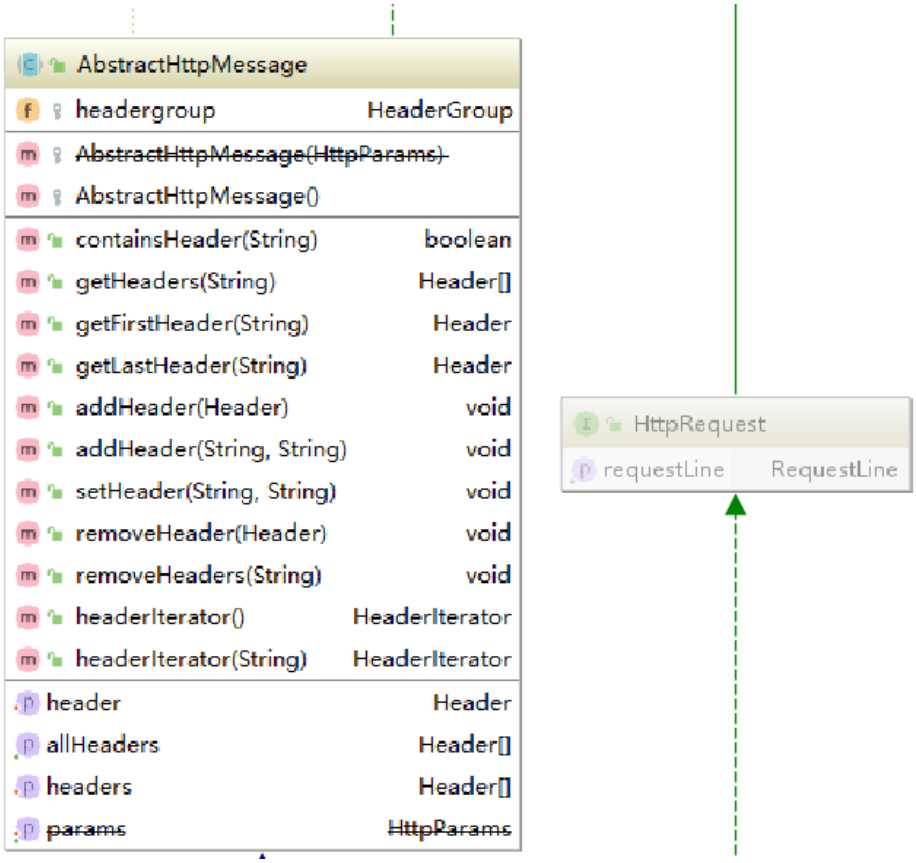
```
HttpRequest request = new BasicHttpRequest("GET", "/",
    HttpVersion.HTTP_1_1);

System.out.println(request.getRequestLine().getMethod());
System.out.println(request.getRequestLine().getUri());
System.out.println(request.getProtocolVersion());
System.out.println(request.getRequestLine().toString());
```

可以看到输出流的结果

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

方法和标识符的类型均为串类型，协议版本类型类包含了版本 UID、协议名和协议版本号。

BasicHTTPRequest 中类 BasicHTTPRequest 继承了抽象类 AbstractHttpMessage，两者又同时与接口 HTTPMessage 相连，HTTPMessage 中包含了如下的一些方法。下图是消息请求过程中可以用到的方法：



根据命名可以推断出方法的目的以及作用。

2. response message

下图表示 HTTP 响应是服务器在接收并解释请求消息后发送回客户端的消息。该消息的第一行包括协议版本，后跟数字状态代码及其相关的文本短语。

```
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());
```

输出流如下

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

总的类输入内容包含了状态行、协议版本、数字状态代码、文本短语、

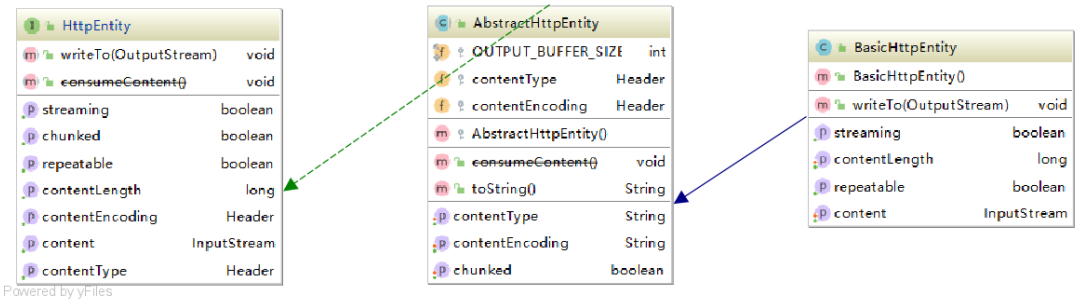
不同于请求消息时输入的参数是满的，这里只输入了 ProtocolVersion,code,reasonPhrase.

BasicHttpResponse 的结构依然是继承了 AbstractHttpMessage,连接了 HttpResponse 接口（继承了 HttpMessage）。这里消息(报文)的 header 内可以包含多个，用于表示消息的具体属性，包括请求内容的长度和类型等。

（二）分析三种实体的创建和数据结构

1. BasicHttpEntity

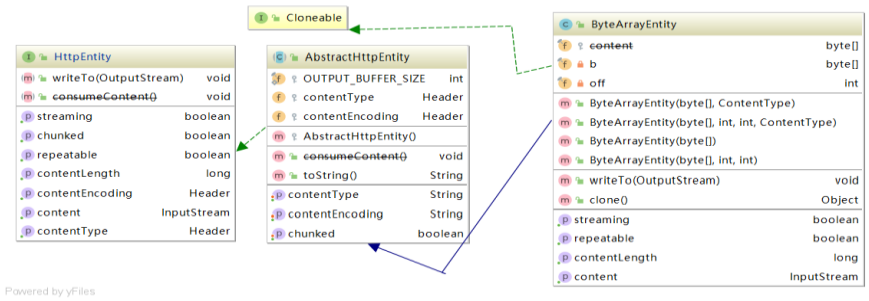
此类用于从 HTTP 消息接收的实体。该实体具有空构造函数。构建后，它表示没有内容，并且具有负内容长度。



从类图中可以明显的看到该类的继承关系，Basic 类继承了抽象类 HTTPEntity，并连接接口 HTTPEntity，从而实现了其中判断流、是否可重复、是否数据分块传递，以及请求返回内容长度和类型的方法。尽管 Basic 中实现了设置流长度、类型，但是根据注释该方法已被重载，这里不再记录。

2. ByteArrayEntity

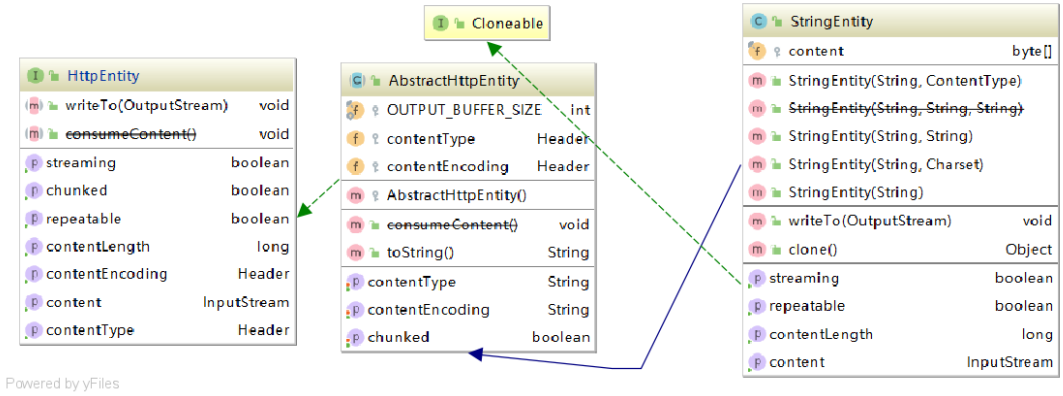
ByteArrayEntity 是一个自包含，可重复的实体，从给定的字节数组中获取其内容。将字节数组提供给构造函数。



相较于 Basic 类，ByteArray 类创建的是一个自包含可重复的实体，这就要求 ByteArray 还要连接可重复的接口 Cloneable，方法中实现了重复的方法（clone）,另外该类的作用是根据输入的字节数组中创建实体，因此实现的方法中包含了针对不同输入情况下的方法处理，包括只输入字节数组或者内容类型的方法。

3.StringEntity

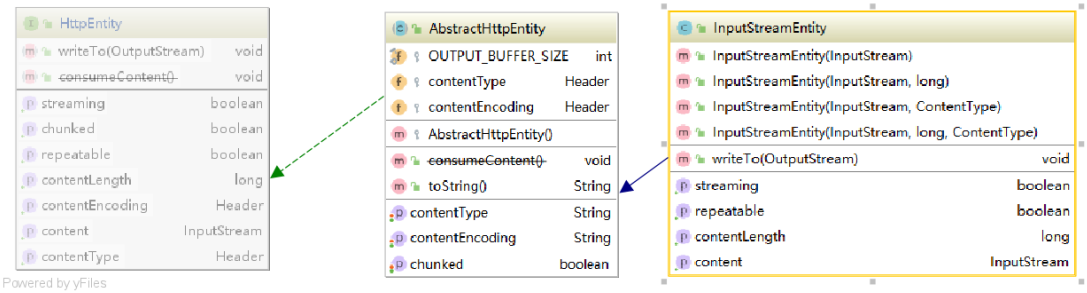
StringEntity 是一个自包含，可重复的实体，从 java.lang.String 对象中获取其内容。它有三个构造方法，一个是用给定的 java.lang.String 对象构造; 第二个也对字符串中的数据进行字符编码; 第三个允许指定 mime 类型。(Multipurpose Internet Mail Extensions)



按照构造方法输入可见上图中 StringEntity 中三个构造方法自上至下依次为用给定的 String 类型构造，对字符串编码构造，接收 MIME 类型的构造。其余部分与 Byte 的关系相同。

4.InputStreamEntity

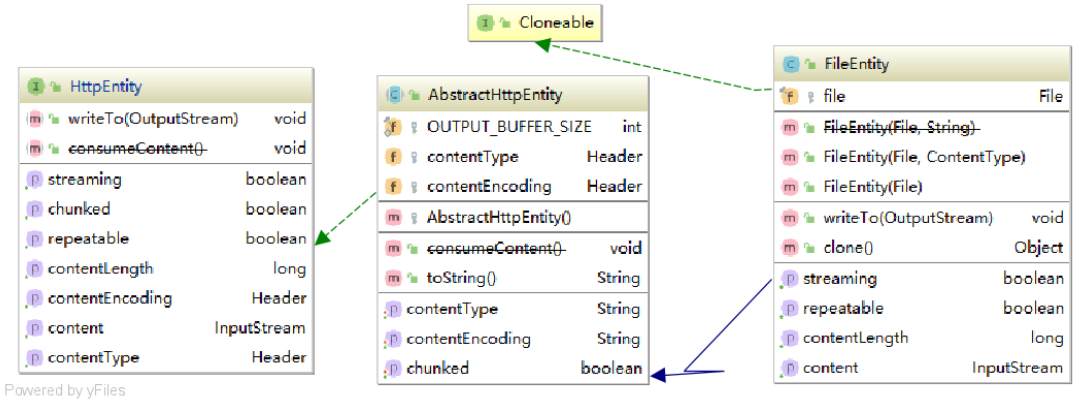
InputStreamEntity 是一个流式，不可重复的实体，从输入流中获取其内容。通过提供输入流和内容长度来构造它。使用内容长度来限制从中读取的数据量 java.io.InputStream。如果长度与输入流上可用的内容长度匹配，则将发送所有数据。或者，负内容长度将读取输入流中的所有数据，这与提供确切的内容长度相同，因此使用长度来限制要读取的数据量。



不同于 Basic 类的创建方法，InputSstream 要求创建实体时提供内容长度，并据此从输入流中选取内容。其他继承关系均与 Basic 一致，注意不可重复，故，没有与 cloneable 接口相连。

5.FileEntity

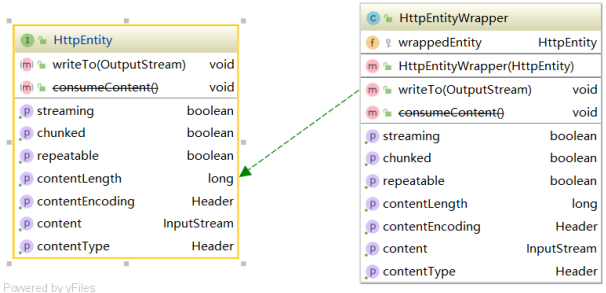
FileEntity 是一个自包含，可重复的实体，从文件中获取其内容。主要用于流式传输不同类型的大型文件，需要提供文件的内容类型，例如，发送 zip 文件需要内容类型 application/zip，用于 XML application/xml。



根据类的定义，要求输入文件，其他方法可以添加输入内容类型，可重复因此与 cloneable 接口相连。其他继承关系与 Byte 相同。

6.HttpEntityWrapper

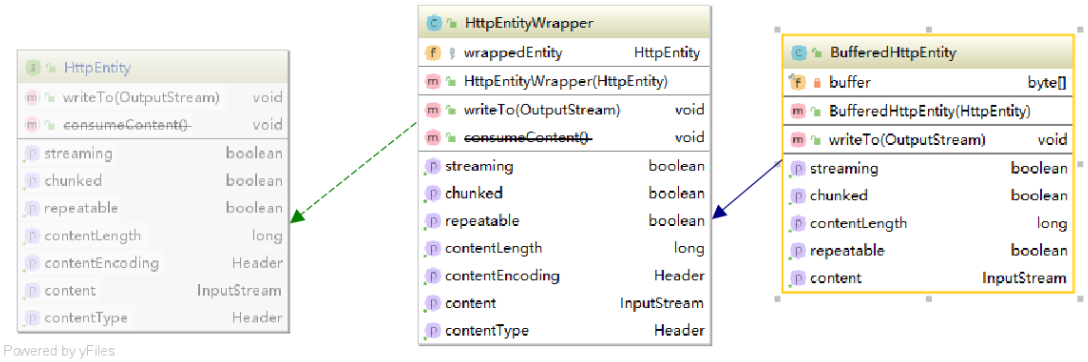
这是用于创建包装实体的基类。包装实体保存对包装实体的引用，并委托对它的所有调用。包装实体的实现可以从此类派生，并且只需要覆盖那些不应该委托给包装实体的方法。



Wrapper 类没有继承抽象类，而是直接与 HTTPEntity 接口相连，另外该类是不可重复的，原本在抽象类中声明的方法转移到了 Wrapper 中。

7. BufferedHttpEntity

BufferedHttpEntity 是 HttpEntityWrapper 的子类。通过提供另一个实体来构建它。它从提供的实体读取内容，并将其缓存在内存中。这使得可以从不可重复的实体制作可重复的实体。如果提供的实体已经可重复，它只是将调用传递给底层实体。



（三）HTTP 协议处理器

HTTP 协议拦截器是一段实现了 HTTP 协议某一特定方面的程序。通常协议拦截器会对到来报文的特定头部或相关的一组头部进行处理，或者用特定头部或相关的一组头部对要发送的报文进行填充。协议拦截器也可以操纵内容实体，将其附加在报文中。关于透明地对报文内容进行解压和压缩也是一个很好的例子。一般使用这种方式称为“装饰者”模式，即用一个包裹实体来装饰那个原始的实体。几个协议拦截器可以组合成一个逻辑单元。

HTTP 协议拦截器是一组实现了“责任链模式”的协议拦截器，每个协议拦截器都在它的责职范围内进行某些特定方面的工作（针对 HTTP 协议的不同方面）。一般来说只要不在特别状态的执行上下文，拦截器们的执行顺序不是一个问题。如果协议拦截器互相依赖而必须以特定的顺序执行，那么应该将它们以应有的执行顺序放在协议处理器的一个序列中。协议拦截器应该实现为线程安全的。协议拦截器不能使用实例变量，除非访问这些变量是同步的。（实例变量即是要将类实例化后才能使用的类变量）。

1. 标准协议拦截器

首先需要了解一下所有针对请求的协议拦截器的唯一接口 HttpRequestInterceptor 和针对答复的协议拦截器唯一接口 HttpResponseInterceptor。

```
public interface HttpRequestInterceptor {

    /**
     * Processes a request.
     * On the client side, this step is performed before the request is
     * sent to the server. On the server side, this step is performed
```

```
     * on incoming messages before the message body is evaluated.
     *
     * @param request   the request to preprocess
     * @param context   the context for the request
     *
     * @throws HttpException in case of an HTTP protocol violation
     * @throws IOException in case of an I/O error
     */
    void process(HttpRequest request, HttpContext context)
        throws HttpException, IOException;

}
```

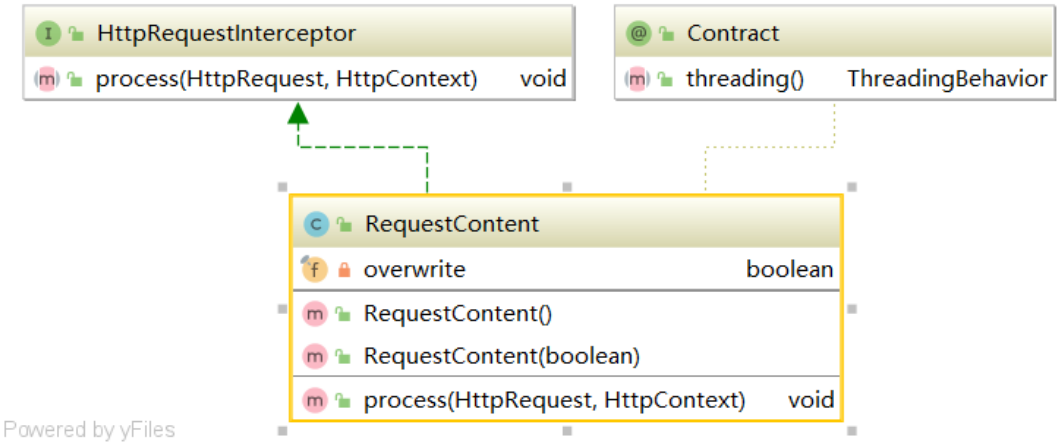
由此看到拦截器接口的输入内容包括了预处理的请求和请求包含的内容，同时通过 throw 定义了两种错误形式，分别是违反协议内容的错误和输入输出端口的错误。

```
public interface HttpResponseInterceptor {

    /**
     * Processes a response.
     * On the server side, this step is performed before the response is
     * sent to the client. On the client side, this step is performed
     * on incoming messages before the message body is evaluated.
     *
     * @param response  the response to postprocess
     * @param context   the context for the request
     *
     * @throws HttpException in case of an HTTP protocol violation
     * @throws IOException in case of an I/O error
     */
    void process(HttpResponse response, HttpContext context)
        throws HttpException, IOException;

}
```

HTTP 响应的端口与请求不同之处在于输入的不是请求方式，而是答复方式。其他错误定义等都没有变化。

（1） RequestContent

RequestContent 是最重要的向外请求拦截器，它的责任是：通过加上 Content-Length 来界定向外请求报文的内容长度，或根据报文附加实体的特征及协议版本加上某些 Transfer-Content 头部（即补充头部）。这个拦截器起作用的条件是，客户端的协议处理器正确运行。



RequestContent 类中定义了布尔变量 overwrite，用来判断是否对报文进行重写，并在方法 RequestContent 中对是否跳转到 HTTP 错误进行判断。Threading 表示可以进行多个请求拦截工作。对接口方法 process 进行了重载，下图是对 process 方法的详细解释：

```
public void process(final HttpRequest request, final HttpContext context)
        throws HttpException, IOException {
    Args.notNull(request, "HTTP request");
    if (request instanceof HttpEntityEnclosingRequest) { //是否是右边类的实例，是否带有实体
        if (this.overwrite) {
            request.removeHeaders(HTTP.TRANSFER_ENCODING);//移除分块编码
            request.removeHeaders(HTTP.CONTENT_LEN);//移除内容长度
        } else {
            if (request.containsHeader(HTTP.TRANSFER_ENCODING)) {
                throw new ProtocolException("Transfer-encoding header already present");//
不可重写则跳转到错误处理
            }
            if (request.containsHeader(HTTP.CONTENT_LEN)) {
                throw new ProtocolException("Content-Length header already present");
            }
        }
        final ProtocolVersion ver = request.getRequestLine().getProtocolVersion();
        final HttpEntity entity = ((HttpEntityEnclosingRequest)request).getEntity();
        if (entity == null) {
            request.addHeader(HTTP.CONTENT_LEN, "0");//长度为0
            return;
        }
```

```
        // Must specify a transfer encoding or a content length
        if (entity.isChunked() || entity.getContentLength() < 0) {
            if (ver.lessEquals(HttpVersion.HTTP_1_0)) {
                throw new ProtocolException(
                        "Chunked transfer encoding not allowed for " + ver);//过低协议版本
            }
            request.addHeader(HTTP.TRANSFER_ENCODING, HTTP.CHUNK_CODING);//添加分块编写传输
        } else {
            request.addHeader(HTTP.CONTENT_LEN,
Long.toString(entity.getContentLength()));//添加内容长度
        }
        // Specify a content type if known
        if (entity.getContentType() != null && !request.containsHeader(
                HTTP.CONTENT_TYPE )) {
            request.addHeader(entity.getContentType());//添加类型
        }
        // Specify a content encoding if known
        if (entity.getContentEncoding() != null && !request.containsHeader(
                HTTP.CONTENT_ENCODING)) {
            request.addHeader(entity.getContentEncoding());//添加内容编码格式：内容编码格式
gzip 和 deflate
        }
    }
}
```

（2） ResponseContent

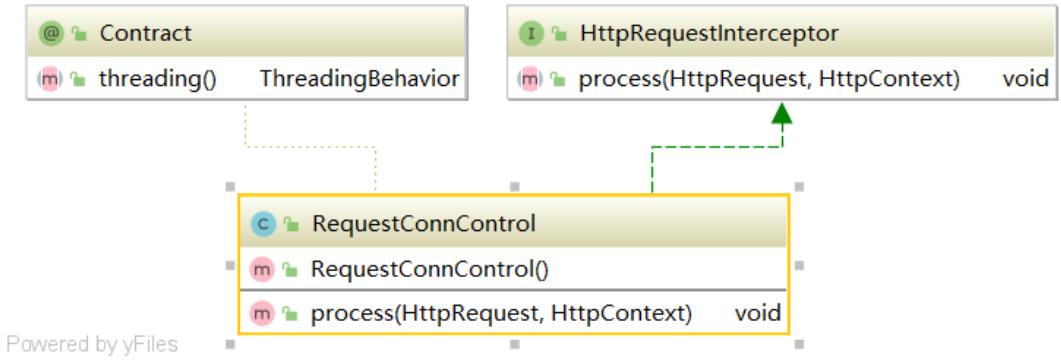
Response 的处理位置在服务器端，其主要内容与请求端的操作近似，最后添加了对所请求内容的状态判定，

```
if (status != HttpStatus.SC_NO_CONTENT//204
        && status != HttpStatus.SC_NOT_MODIFIED//304
        && status != HttpStatus.SC_RESET_CONTENT) {//205
    response.addHeader(HTTP.CONTENT_LEN, "0");
}
```

如果不是上述内容状态，则设置长度为 0，即内容为空。

（3） RequestConnControl

RequestConnControl 是为请求报文添加连接头部的拦截器，对于管理 HTTP/1.0 的持续连接是很重要的。这个拦截器建议用在客户端协议处理器中。



端口连接与请求内容拦截器一致，具体重载的 process 方法自然不同。

```
public class RequestConnControl implements HttpRequestInterceptor {

    public RequestConnControl() {
        super();
    }

    @Override
    public void process(final HttpRequest request, final HttpContext context)
            throws HttpException, IOException {
        Args.notNull(request, "HTTP request");

        final String method = request.getRequestLine().getMethod();
        if ("CONNECT".equalsIgnoreCase(method)) {//字符串比较，忽略大小写，，判断是否已经连接
            return;
        }

        if (!request.containsHeader(HTTP.CONN_DIRECTIVE)) {
            // Default policy is to keep connection alive
            // whenever possible
            request.addHeader(HTTP.CONN_DIRECTIVE, HTTP.CONN_KEEP_ALIVE);//添加连接头部，添
加keppalive 保持连接
        }
    }

}
```

（4） ResponseConnControl

Response 作用在服务器端，作用目的与请求类似，只是 process 方法根据不同情况分类讨论：

```
public void process(final HttpResponse response, final HttpContext context)
        throws HttpException, IOException {
    Args.notNull(response, "HTTP response");
```

```
    final HttpCoreContext corecontext = HttpCoreContext.adapt(context);

    // Always drop connection after certain type of responses
    final int status = response.getStatusLine().getStatusCode();
    if (status == HttpStatus.SC_BAD_REQUEST ||
            status == HttpStatus.SC_REQUEST_TIMEOUT ||
            status == HttpStatus.SC_LENGTH_REQUIRED ||
            status == HttpStatus.SC_REQUEST_TOO_LONG ||
            status == HttpStatus.SC_REQUEST_URI_TOO_LONG ||
            status == HttpStatus.SC_SERVICE_UNAVAILABLE ||
            status == HttpStatus.SC_NOT_IMPLEMENTED) {
        response.setHeader(HTTP.CONN_DIRECTIVE, HTTP.CONN_CLOSE);//非正常答复下的断开连接
        return;
    }
    final Header explicit = response.getFirstHeader(HTTP.CONN_DIRECTIVE);
    if (explicit != null && HTTP.CONN_CLOSE.equalsIgnoreCase(explicit.getValue())) {
        // Connection persistence explicitly disabled 不能长时间连接
        return;
    }
    // Always drop connection for HTTP/1.0 responses and below
    // if the content body cannot be correctly delimited 限制
    final HttpEntity entity = response.getEntity();
    if (entity != null) {
        final ProtocolVersion ver = response.getStatusLine().getProtocolVersion();
        if (entity.getContentLength() < 0 &&
                (!entity.isChunked() || ver.lessEquals(HttpVersion.HTTP_1_0))) {//强制关闭
            response.setHeader(HTTP.CONN_DIRECTIVE, HTTP.CONN_CLOSE);
            return;
        }
    }
    // Drop connection if requested by the client or request was <= 1.0
    final HttpRequest request = corecontext.getRequest();
    if (request != null) {
        final Header header = request.getFirstHeader(HTTP.CONN_DIRECTIVE);
        if (header != null) {
            response.setHeader(HTTP.CONN_DIRECTIVE, header.getValue());//客户端申请关闭
        } else if (request.getProtocolVersion().lessEquals(HttpVersion.HTTP_1_0)) {
            response.setHeader(HTTP.CONN_DIRECTIVE, HTTP.CONN_CLOSE);//请求版本低
        }
    }
}
```

（5） RequestDate

RequestDate 是用来为请求实体添加时间头部的拦截器，这个拦截器对于客户端是可选的。同样的 RequestDate 的结构与上面请求拦截器的结构类似，process 方法用于添加 Date 头。这里由于代码较为简单，不再展示。

（6） ResponseDate

ResponseDate 是用来为响应实体添加时间头部的拦截器，这个拦截器建议用在服务器端。其端口与内容和请求 Date 基本一致。

（7） RequestExpectContinue

RequestExpectContinue 通过添加 Expect 头部使得'expect-continue'握手可以进行，建议用在客户端协议处理器。其连接结构同上，代码中 process 方法如下

```
public void process(final HttpRequest request, final HttpContext context)
        throws HttpException, IOException {
    Args.notNull(request, "HTTP request");

    if (!request.containsHeader(HTTP.EXPECT_DIRECTIVE)) {
        if (request instanceof HttpEntityEnclosingRequest) {//带有实体的请求
            final ProtocolVersion ver = request.getRequestLine().getProtocolVersion();
            final HttpEntity entity =
((HttpEntityEnclosingRequest)request).getEntity();//类型转换
            // Do not send the expect header if request body is known to be empty
            if (entity != null
                    && entity.getContentLength() != 0
&& !ver.lessEquals(HttpVersion.HTTP_1_0)) {
                final boolean active = request.getParams().getBooleanParameter(
                        CoreProtocolPNames.USE_EXPECT_CONTINUE, this.activeByDefault);
                if (active) {//判断是否使用 expect-continue
                    request.addHeader(HTTP.EXPECT_DIRECTIVE, HTTP.EXPECT_CONTINUE);
                }
            }
        }
    }
}
```

（8） RequestTargetHost

RequestTargetHost 是用来添加主机 Host 头部的，这个拦截器对于客户端协议处理器是必需的。

（9） RequestUserAgent

RequestUserAgent 用来添加用户代理 User-Agent 头部，建议用在客户端协议处理器中。

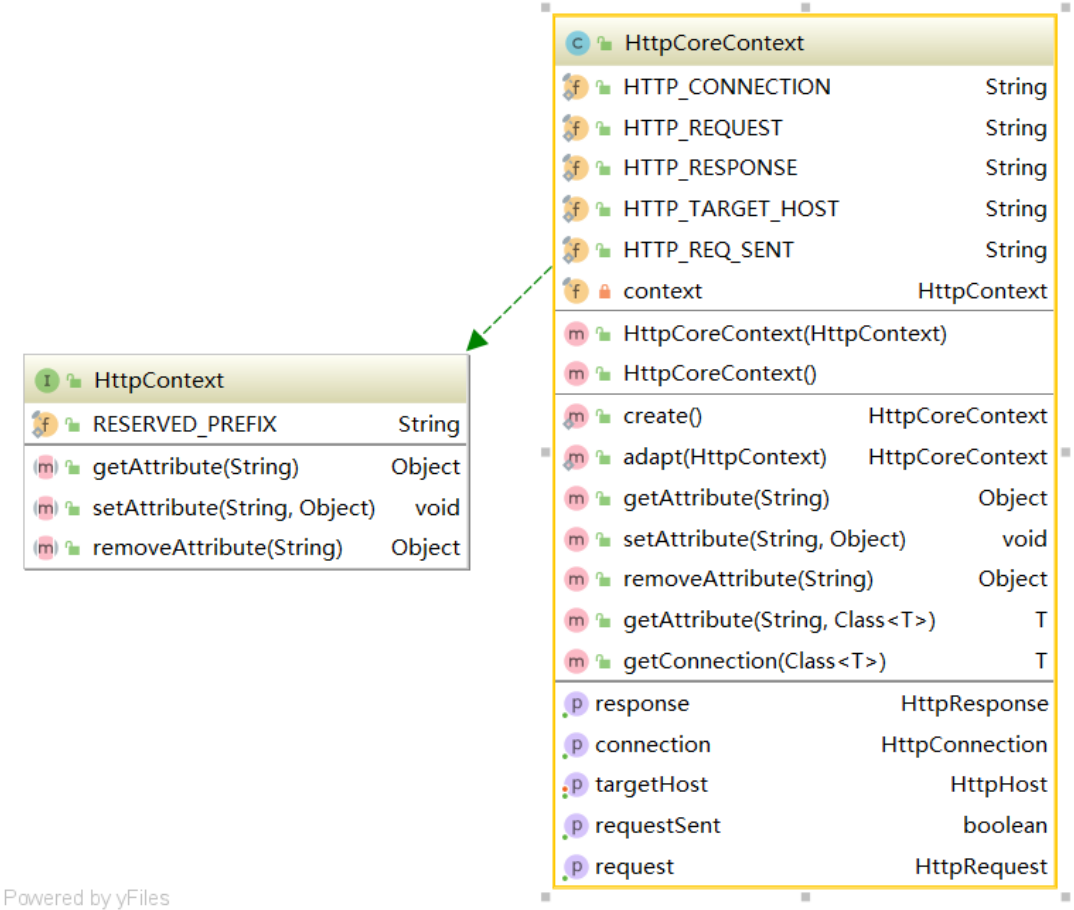
（10） ResponseServer

ResponseServer 用来添加服务器 Server 头部，建议用在服务器端协议处理器中。

（四）HTTP 上下文

最初 HTTP 被设计为无状态，面对“请求-响应”的协议。可是，真实世界的应用程序经常需要在几个逻辑“请求-响应”报文交换中持续化状态信息。为了能够保持应用程序的进程状态，HttpCore 允许 HTTP 报文在一个特定的执行上下文中被处理，这个上下文我们叫它 HTTP 上下文(HTTP Context)。如果在连续的 HTTP 请求中，同一个上下文被重复使用，那么多个逻辑相关的报文就可以参与到同一个逻辑会话中（即，一个连接一直不断开）。

请注意，HttpContext 能保存任意的对象，所以在多个进程中共享可能是不安全的，请确保 HttpContext 的实例在同一时间内只能被一个进程访问（其实有些实现类是线程安全的）。



上图所示的 HTTPcoreContext 中连接了 HttpContext 接口，其中包括了获取、修改、移除属性等方法。右边的类中额外包含了创造、匹配上下文的方法，以及获取上下文中内容的

响应、连接、目标主机、是否请求等属性。

四、阻塞 I/O 模型分析

前面的功能分析基于 HTTP 报文的产生和拦截器的使用过程分析，并未涉及真正的 HTTP 报文传输，下面将基于实际模型分析整个过程中的一些具体实现。这里根据官方教程研究阻塞 I/O 模型，若仍有时间可继续研究。

（一）模型简介

阻塞（或典型）I/O 在 Java 中有高效方便的 I/O 模型，在并发量不是很大的情况下非常适合高性能的应用程序。现代 JVM 虚拟机能够有效进行上下文切换，并且在并发连接低于 1000，大多数连接都在忙着传送数据这种情况下，阻塞 I/O 模型应该提供最好的性能（事实它做到了）。可是，对于某些应用程序来说，大多数的连接都会被闲置，那么上下文切换的代价会很大，消耗大量资源，此时用非阻塞 I/O 模型可能是更好的选择。

阻塞就是要排队执行，连接闲置了都不能断开，如连接后等了好久都不见报文来，可是又不能保证报文不来，所以一直等待。

（二）阻止 HTTP 连接

HTTP 连接的作用是为了 HTTP 报文的序列化和反序列化。一般很少直接使用 HTTP connection 对象，因为有更高层次的协议部分可以用来执行和处理 HTTP 请求。可是有些时候还是需要直接用到这种 HTTP 连接，举个例子，访问某些属性如连接状态，套接字超时或者本地和远程 IP 地址。谨记 HTTP 连接不是线程安全的。所以限制所有 HTTP 连接对象只用在一个线程上。唯一可以在不同进程中安全调用的方法是 HttpConnection 接口和其子接口中的 HttpConnection.shutdown()。

HTTP 报文的序列化和反序列化，其中序列化的目的是将对象的字节序列永久保存在硬盘中，通常存放在一个文件中，或者在网络上传输的通常是对象的字节序列。接收端一般接收到字节序列后反序列化就可以将其转化为对象。

（三）HTTPcore 支持的内容传递机制

1. 内容长度限定

内容长度最大值受内容长度决定，内容长度存储在内容头部。

2. 身份编码

内容在关闭最底层连接时被区分处理，这时的身份编码只能被用在服务器端。

3. 块编码

内容实体被分割为等大的数据块 chunk，进行传播。

（四）终止 HTTP 连接

终止 HTTP 连接有 close 和 shutdown 两种方式，但是 close 方法尝试清理缓存中的内容并阻塞，在多线程情况下会导致线程之间不安全，而 shutdown 仅仅是切断了连接，是线程安全的。

（五）HTTP 例外处理

HTTP 例外有两种类型，一种是输入输出例外，例如 socket 建立时超时等，另一种是违反了 HTTP 用户协议而触发的且不可复位的例外。

（六）阻塞 HTTP 协议处理程序

HttpRequestHandler 用于处理特定的 http 请求，会响应特定的 http 请求并返回特定的实体对象。

（七）请求处理器

HTTP 请求处理通过 HttpRequestHandlerResolver 将请求的 URI 对应到一个请求处理模块，URI 的三种形式为*，<uri>*,*<uri>.

（八）HTTP 服务处理请求

HTTP 处理请求通过 HttpService 中的 handleRequest 处理请求，

```
public void handleRequest(
        final HttpServerConnection conn,
        final HttpContext context) throws IOException, HttpException {

    context.setAttribute(HttpCoreContext.HTTP_CONNECTION, conn);

    HttpRequest request = null;
    HttpResponse response = null;

    try {
        request = conn.receiveRequestHeader();
        if (request instanceof HttpEntityEnclosingRequest) {

```

```
            if (((HttpEntityEnclosingRequest) request).expectContinue()) {
                response = this.responseFactory.newHttpResponse(HttpVersion.HTTP_1_1,
                        HttpStatus.SC_CONTINUE, context);
                if (this.expectationVerifier != null) {
                    try {
                        this.expectationVerifier.verify(request, response, context);
                    } catch (final HttpException ex) {
                        response =
this.responseFactory.newHttpResponse(HttpVersion.HTTP_1_0,
                                HttpStatus.SC_INTERNAL_SERVER_ERROR, context);
                        handleException(ex, response);
                    }
                }
                if (response.getStatusLine().getStatusCode() < 200) {
                    // Send 1xx response indicating the server expections
                    // have been met
                    conn.sendResponseHeader(response);
                    conn.flush();
                    response = null;
                    conn.receiveRequestEntity((HttpEntityEnclosingRequest) request);
                }
            } else {
                conn.receiveRequestEntity((HttpEntityEnclosingRequest) request);
            }
        }

        context.setAttribute(HttpCoreContext.HTTP_REQUEST, request);

        if (response == null) {
            response = this.responseFactory.newHttpResponse(HttpVersion.HTTP_1_1,
                    HttpStatus.SC_OK, context);
            this.processor.process(request, context);
            doService(request, response, context);
        }

        // Make sure the request content is fully consumed
        if (request instanceof HttpEntityEnclosingRequest) {
            final HttpEntity entity = ((HttpEntityEnclosingRequest)request).getEntity();
            EntityUtils.consume(entity);
        }

    } catch (final HttpException ex) {
        response = this.responseFactory.newHttpResponse
            (HttpVersion.HTTP_1_0, HttpStatus.SC_INTERNAL_SERVER_ERROR,
```

```
                context);
        handleException(ex, response);
    }

    context.setAttribute(HttpCoreContext.HTTP_RESPONSE, response);

    this.processor.process(response, context);
    conn.sendResponseHeader(response);
    if (canResponseHaveBody(request, response)) {
        conn.sendResponseEntity(response);
    }
    conn.flush();

    if (!this.connStrategy.keepAlive(response, context)) {
        conn.close();
    }
}
```

这里的 handleRequest 只能处理一次 Http 请求，在判断完毕请求是否附加实体和请求是否输入完毕之后，会进行返回，并判断是否需要返回实体，最后执行 close 关闭当前 Http 请求。

（九）HTTP 请求执行器

HttpRequestExecutor 是基于阻塞 I/O 模型的协议处理器，可以在判断没有接收或者发送请求的情况下自动处理将要执行的发送请求或者接收请求的操作。

（十）连接保持/再利用

ConnectionReuseStrategy 判断当前的连接是否可以再次利用，该接口的使用要求该连接是线程安全的,当该接口用于多线程时连接复用需要保证线程安全。

（十一）连接池技术

高效的客户端 HTTP 传输通常需要复用持久的连接，HTTPcore 提供了持久连接的管理池，连接池是线程安全的，可以同时处理多个请求。连接池总计允许 20 个并发连接，实际的情况下，动态可变的连接池上限能够根据上下文增加更多的并发连接。

一个连接是否应该返回连接池需要连接的用户判断处理，即使该连接不可以再被复用，也应该由用户将其释放。

BasicConnPool 使得用户能够跟踪并获取当前被使用的池和空闲池的状态，保证了池的使用者有选择性地中止连接。

（十二）TLS/SSL 支持

阻塞连接能够绑定任意一个 socket，使得 SSL 非常直接的实现，SSL 绑定一个 socket 保证了消息传递过程中的保密性。

（十三）嵌入式 HTTP 服务器

建立了基于阻塞 I/O 组件的服务器。

五、基于源代码例子分析请求过程

（一）发送请求过程分析

```
public class ElementalHttpGet {

    public static void main(String[] args) throws Exception {
        HttpProcessor httpproc = HttpProcessorBuilder.create()
            .add(new RequestContent())
            .add(new RequestTargetHost())
            .add(new RequestConnControl())
            .add(new RequestUserAgent("Test/1.1"))
            .add(new RequestExpectContinue(true)).build();

        HttpRequestExecutor httpexecutor = new HttpRequestExecutor();

        HttpCoreContext coreContext = HttpCoreContext.create();
        HttpHost host = new HttpHost("localhost", 8080);
        coreContext.setTargetHost(host);

        DefaultBHttpClientConnection conn = new DefaultBHttpClientConnection(8 * 1024);
        ConnectionReuseStrategy connStrategy = DefaultConnectionReuseStrategy.INSTANCE;

        try {

            String[] targets = {
                    "/",
                    "/servlets-examples/servlet/RequestInfoExample",
                    "/somewhere%20in%20pampa"};

            for (int i = 0; i < targets.length; i++) {
                if (!conn.isOpen()) {
```

```
                    Socket socket = new Socket(host.getHostName(), host.getPort());
                    conn.bind(socket);
                }
                BasicHttpRequest request = new BasicHttpRequest("GET", targets[i]);
                System.out.println(">> Request URI: " +
request.getRequestLine().getUri());

                httpexecutor.preProcess(request, httpproc, coreContext);
                HttpResponse response = httpexecutor.execute(request, conn, coreContext);
                httpexecutor.postProcess(response, httpproc, coreContext);

                System.out.println("<< Response: " + response.getStatusLine());
                System.out.println(EntityUtils.toString(response.getEntity()));
                System.out.println("==============");
                if (!connStrategy.keepAlive(response, coreContext)) {
                    conn.close();
                } else {
                    System.out.println("Connection kept alive...");
                }
            }
        } finally {
            conn.close();
        }
    }

}
```

简单分析发送请求过程中客户端要做的事。

1. 首先是客户端建立发送请求的进程，包括新建报文的内容、目标 host 地址、代理协议版本、连接控制方式、是否与服务器保持连接等需求。
2. 然后 HttpRequestExcutor 在客户端申请发送请求，即该类中的方法可以被调用来执行发送消息或者接收消息或者切断连接等操作。
3. 将 1 中建立的变量的内容的定义进一步完善，如 host 等。
4. 调用 `DefaultBHttpClientConnection` 和 `DefaultConnectionReuseStrategy` 与服务器建立连接并且建立了保持连接，或者使得当前连接是可以被复用的。
5. 接下来通过建立一个 socket 模式的连接来与服务器互通，开始传输信息。
6. 通过 BasicHttpRequest 建立了一个 request 类用来对传递的消息进行操作，包括获得请求行等。
7. HttpRequestExcutor 类对消息头的内容处理，并且生成返回的内容，设置返回的 ID 序列号。
8. 判断是否需要保持连接，否则关闭连接。

总结：以上就是一次基于 httpcore 的请求过程，其中使用了 close 来关闭请求连接，显然这个例子下的请求只能够是单线程的，不能够并发申请。

（二）多线程发送请求分析

```
public class ElementalPoolingHttpGet {

    public static void main(String[] args) throws Exception {
        final HttpProcessor httpproc = HttpProcessorBuilder.create()
            .add(new RequestContent())
            .add(new RequestTargetHost())
            .add(new RequestConnControl())
            .add(new RequestUserAgent("Test/1.1"))
            .add(new RequestExpectContinue(true)).build();

        final HttpRequestExecutor httpexecutor = new HttpRequestExecutor();

        final BasicConnPool pool = new BasicConnPool(new BasicConnFactory());
        pool.setDefaultMaxPerRoute(2);
        pool.setMaxTotal(2);

        HttpHost[] targets = {
                new HttpHost("www.google.com", 80),
                new HttpHost("www.yahoo.com", 80),
                new HttpHost("www.apache.com", 80)
        };

        class WorkerThread extends Thread {

            private final HttpHost target;

            WorkerThread(final HttpHost target) {
                super();
                this.target = target;
            }

            @Override
            public void run() {
                ConnectionReuseStrategy connStrategy =
DefaultConnectionReuseStrategy.INSTANCE;
                try {
                    Future<BasicPoolEntry> future = pool.lease(this.target, null);

                    boolean reusable = false;
                    BasicPoolEntry entry = future.get();
                    try {
                        HttpClientConnection conn = entry.getConnection();
```

```
                    HttpCoreContext coreContext = HttpCoreContext.create();
                    coreContext.setTargetHost(this.target);

                    BasicHttpRequest request = new BasicHttpRequest("GET", "/");
                    System.out.println(">> Request URI: " +
request.getRequestLine().getUri());

                    httpexecutor.preProcess(request, httpproc, coreContext);
                    HttpResponse response = httpexecutor.execute(request, conn,
coreContext);
                    httpexecutor.postProcess(response, httpproc, coreContext);

                    System.out.println("<< Response: " + response.getStatusLine());
                    System.out.println(EntityUtils.toString(response.getEntity()));

                    reusable = connStrategy.keepAlive(response, coreContext);
                } catch (IOException ex) {
                    throw ex;
                } catch (HttpException ex) {
                    throw ex;
                } finally {
                    if (reusable) {
                        System.out.println("Connection kept alive...");
                    }
                    pool.release(entry, reusable);
                }
            } catch (Exception ex) {
                System.out.println("Request to " + this.target + " failed: " +
ex.getMessage());
            }
        }

    };

    WorkerThread[] workers = new WorkerThread[targets.length];
    for (int i = 0; i < workers.length; i++) {
        workers[i] = new WorkerThread(targets[i]);
    }
    for (int i = 0; i < workers.length; i++) {
        workers[i].start();
    }
    for (int i = 0; i < workers.length; i++) {
        workers[i].join();
    }
```

```
    }

}
```

同样的代码过程不再重复，这里只介绍与（一）中不同之处来体现多线程请求。

1. 在建立了执行器之后建立了连接池，并设置了连接池的大小为 2。
2. 通过 WorkThread 中 target 来建立区分多线程请求实例。为每个请求在连接池中分配一个请求处理线程，并且判断连接是否切断从而将连接池内的线程释放。
3. 按照设计的三个 http 请求产生三个请求线程，执行，最后合并。

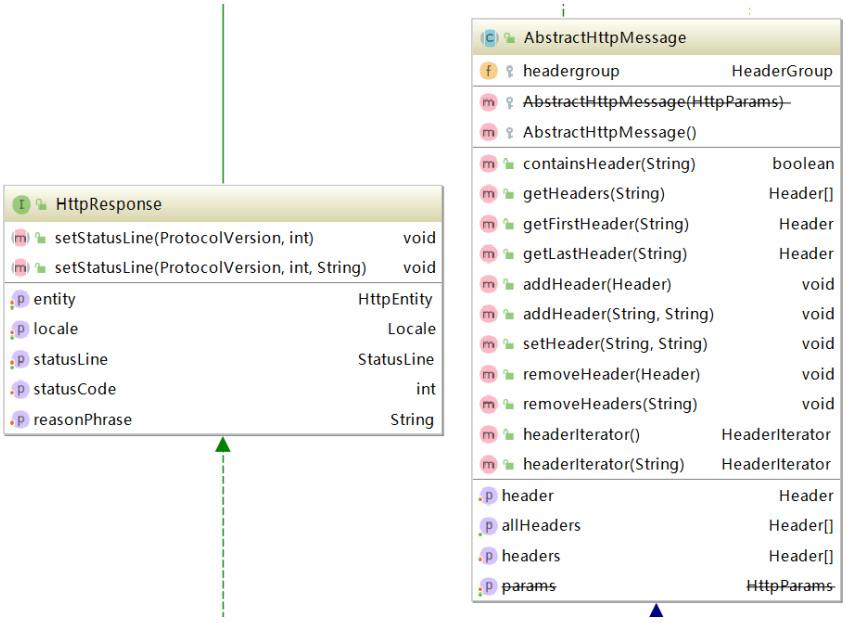
六、高级设计意图分析

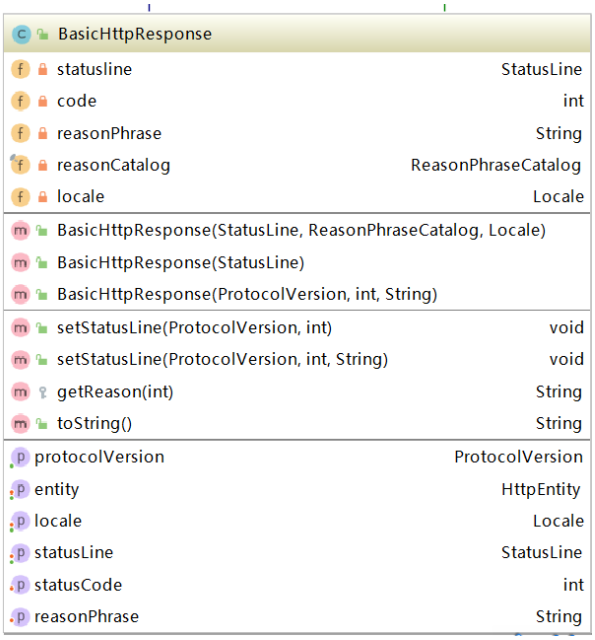
（一）设计模式分析

1. 适配器模式

```
public class BasicHttpResponse extends AbstractHttpMessage implements HttpResponse
```

如上图是 httpcore 中对一个应答类的定义，采用了 extend 一个类和 implement 一个接口的方式，形成了一个针对特定应答消息类型的类，

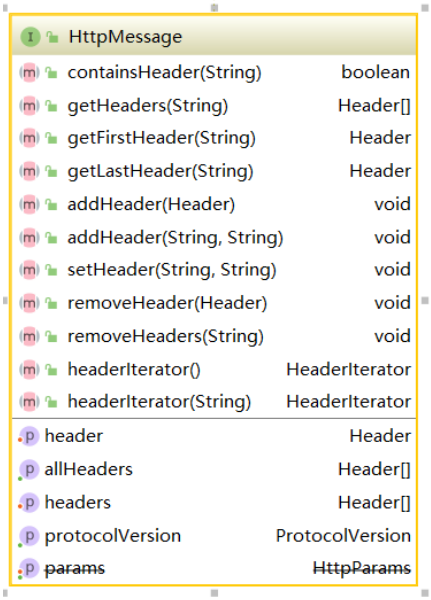




可以看到 BasicHttpResponse 中既有对上面接口方法的继承(getHeaders)，也有对方法的重写(getStatusLine)，而实际情况中一个消息答复时需要对消息头的内容修改，也需要对状态行的内容修改，因此才会实现这样一个更加具体的类。这体现了适配器模式在 httpcore 接口设计时的优点。

2. 观察者模式

观察者模式体现最明显的是在 Message 的处理上面，



上图中可以清晰地看到，HttpMessage 中定义了 9 个针对消息头的方法，可以说是作为监视者的方法，当请求发送或者回答发生时，实现该接口的类可以同时调用这些方法对消息头进行处理，而不受其他因素影响。体现了观察者模式下对消息头的灵活处理。

（二）基于设计原则的分析

Httpcore 项目本身包含大量的类和接口，在阅读时一度让我苦不堪言，分不清楚类和类之间的联系。了解了代码中 example 的流程已经设计原则的内容我才逐渐明白了接口和类十分多得原因。

首先就是单一职责和开放封闭方面，Httpcore 传递消息本身的数据结构就会涉及到大量信息，包括 host 地址，请求类型、是否有实体等大量信息，而这就需要对每一个因素有对应的处理方法，比如 header 就有 getheader、addheader 等方法，这些需要单独封装为接口的，不能够都添加在一个抽象类或者接口当中。同时由于每个因素都定义了对应的接口，那么一个消息类实现过程就是不断实现上述接口和继承上述类的过程，而具体的消息类只需要在方法中对上述的方法进行重写或者继承就可以获得对应的效果，反而接口中的方法处理是高度抽象简化的，具体的处理方法需要到具体的消息中单独实现，充分体现了开放封闭的特性，便于实现请求时的消息方法修改。

其次是里氏替换，如上所述，消息具体实现到请求上面需要将接口中的所需方法一一实现，而不是直接继承，因此只需要在消息类中重写方法即可产生所需要的方法。消息中的类可以完全替换父类中的方法。

最后是接口隔离和依赖倒转，httpcore 中的接口层次复杂，一个消息类一直追溯到最顶层可以找到一层甚至是两层接口，但正是这样复杂的接口，才让一个报文中消息的各个部分内容不会受到其他来源的方法的影响，还是 HttpMessage,HttpRequest 同样是接口，只有一个方法，但是它依然实现了 HttpMessage，

```
public interface HttpRequest extends HttpMessage {

    /**
     * Returns the request line of this request.
     * @return the request line.
     */
    RequestLine getRequestLine();

}
```

而不是在 HttpMessage 中直接添加 HttpRequest 的方法，此处原因可能是消息同时包含发送或者回答的消息，因此要区分开请求方法和应答方法，故采用两层接口。这样 request 接口和 response 接口就不会产生干扰。

（三）并发实现

Httpcore 中还实现了并发请求的过程，阅读代码发现对应的部分接口有实现了 clonable 的接口，说明是支持多线程的，同时 example 中建立了基于连接池的并发请求实现，这在阻塞 I/O 模型中的影响是非常大的。

七、总结

Httpcore 项目代码有很多的接口和类的复杂关系，一开始阅读时产生很大疑问是非常

正常的，最好的流程是首先从宏观层面了解一次请求操作的具体内容，再结合例子在代码中寻找需要的类和接口，进一步查找实现的方法。这是我目前阅读到现在总结出来的阅读复杂代码的方法，希望对后面的阅读经历有所帮助。

Httpcore 的代码结构非常严谨，接口和类的设计都严格按照消息的结构去设计，这对后面我自己可能的代码实现上面有很大的启发意义，无论是接口设计和代码风格等都十分值得学习。

学期结束了，我目前阅读进度至阻塞 I/O 模型，仍有非阻塞 I/O 模型没有得到分析，希望后续有时间或者相关的任务会继续完成未读完的部分。