Http core 项目具体功能及主要功能分析

肖家睿 2017K8009908013

1. HTTP 是什么

HTTP (HyperText Transfer Protocol) 即超文本传输协议。是互联网上应用最为广泛的一种网络协议。所有的 WWW 文件都必须遵守这个标准。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。HTTP 协议定义了浏览器（万维网客户进程）怎样向万维网服务器请求万维网文档，以及服务器怎样把文档传送给浏览器。从层次的角度看，HTTP 是面向事务的（Transaction-oriented）应用层协议，它规定了在浏览器和服务器之间的请求和响应的格式和规则，它是万维网上能够可靠交换文件（包括文本、声音、图像等各种多媒体文件）的重要基础。

HTTP 从提出到现在经历了很多代版本，现在最常用的是 HTTP 1.1。

2. Http Core 的重要组件及功能

Http Components Core 是一组底层 Http 传输协议组件，为构建客户端/代理/服务器端 HTTP 服务一致的 API。也就是说 Http Core 就是一种 IO 接口，用来实现应用端和服务供应商之间的信息交互。好处就是可以保证信息传递的效率和信息的安全性。

Http 其主要功能可以分为以下几块：

Http Headers，即 Http 请求中的消息头部分，其最主要的实现为 HttpMessage，包括 HttpRequest 和 HttpResponse，是 HttpCore 最核心的内容，也是整个项目最需要实现的对象。

Http Entity，Http 请求中的消息实体，包括其他附属内容；HttpConnection，包括网络服务器的连接状态。

Http Exception，其主要实现是协议例外。

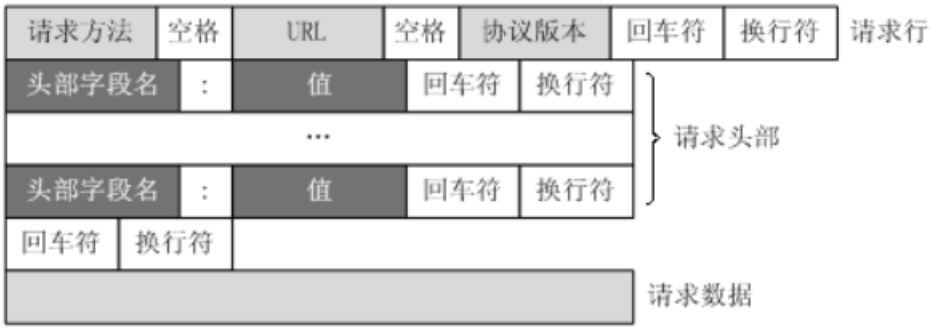
3. Http Message 主要功能

Http Message 包含客户端到服务端的请求以及服务端到客户端的响应。基本的传输接口有 HttpRequest（是客户端到服务端请求的接口）和 HttpResponse（是服务端收到客户端的请求之后对客户端响应的消息的接口）。接口中获取网页内容的请求形式主要有 Get，Post 两种。

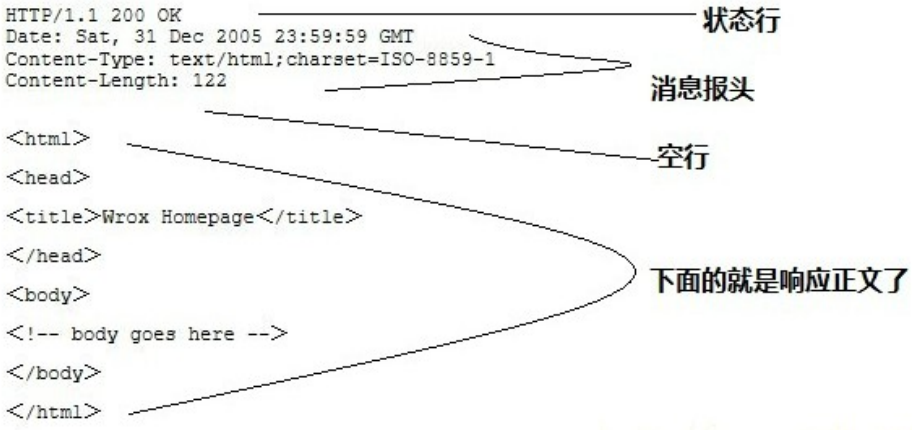
作为 Http Components-Core 的最重要功能，Http Message 实现了 HTTP 消息的有序传输，其中用到了多个接口、抽象类的设计，通过对于消息对象的抽象来达到信息隐藏和结构清晰的目的。

4. Http Message 的实现

作为 Http Components Core 的最重要的功能，Http Message 实现了 HTTP 消息的有序传输，其中用到了多个接口，抽象类的设计，通过对于消息对象的抽象来达到信息隐藏和结构清晰的目的。其大概的组成为：



上图为请求消息的基本组成格式，其中请求行是必须有的，之后的内容都可以视情况省略。



上图为服务器响应消息的内容，其中状态行是必须的，之后的响应可以视情况省略。

接下来我们针对这一功能的实现进行项目框架和源码的阅读（版本为 4.4.x，内容具体为 httpcomponents-core-4.4.x\httpcore\src\main\java\org\apache\http）

在 org.apache.http 目录下，有一个 HttpMessage 接口。首先我们来看这个接口提供了一些方法：

```java
@SuppressWarnings("deprecation")
public interface HttpMessage {

    /**
     * Returns the protocol version this message is compatible with.
     */
    ProtocolVersion getProtocolVersion();

    /**
     * Checks if a certain header is present in this message. Header values are
     * ignored.
     *
     * @param name the header name to check for.
     * @return true if at least one header with this name is present.
     */
    boolean containsHeader(String name);

    /**
     * Returns all the headers with a specified name of this message. Header values
     * are ignored. Headers are orderd in the sequence they will be sent over a
     * connection.
     *
     * @param name the name of the headers to return.
     * @return the headers whose name property equals {@code name}.
     */
    Header[] getHeaders(String name);

    /**
     * Returns the first header with a specified name of this message. Header
     * values are ignored. If there is more than one matching header in the
     * message the first element of {@link #getHeaders(String)} is returned.
     * If there is no matching header in the message {@code null} is
     * returned.
     *
     * @param name the name of the header to return.
     * @return the first header whose name property equals {@code name}
     *   or {@code null} if no such header could be found.
     */
    Header getFirstHeader(String name);
```

如上图所示，包含有：

getProtocolVersion，用来获取协议的版本；

containsHeader 用于判断消息中是否含有 certain header；

getHeaders 用来返回所有与 name 相匹配的消息头；

getFirstHeader 返回第一个与 name 匹配的消息头，与之类似的还有 getLastHeader；

该文件中还有很多方法比如 addHeader，setHeader，removeHeader 等等一系列方法，我们可以看出 Message 这个接口主要就是对 Header 部分的操作，提供了信息处理方法，分析消息头信息方法和处理消息头的方法，进而思考发现这个接口并没有针对不同的消息类型进行区别，而是对所有消息的一种抽象，只是单单从概念上提供了对消息头的许多处理方法。体现出抽象的思想。

在 Http Message 的基础上，HttpRequest 和 HttpResponse 这两个接口继承了这个接口的设计，同时也增加了相关信息的特殊方法。

```java
public interface HttpRequest extends HttpMessage {

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

}
```

如上图所示，HttpRequest 在继承 HttpMessage 接口的基础上又新加入了获取请求行的操作 getRequstLine 的方法，也只加入了这一种方法，所以我们可以推断出一个请求的最低要求是需要有消息头 Header 和请求行 request line 的信息就足够了，也应证了之前我们所说的请求行是必须有的。

```
 */
public interface HttpResponse extends HttpMessage {

    /**
     * Obtains the status line of this response.
     * The status line can be set using one of the
     * {@link #setStatusLine setStatusLine} methods,
     * or it can be initialized in a constructor.
     *
     * @return  the status line, or {@code null} if not yet set
     */
    StatusLine getStatusLine();

    /**
     * Sets the status line of this response.
     *
     * @param statusline the status line of this response
     */
    void setStatusLine(StatusLine statusline);

    /**
     * Sets the status line of this response.
     * The reason phrase will be determined based on the current
     * {@link #getLocale locale}.
     *
     * @param ver       the HTTP version
     * @param code      the status code
     */
    void setStatusLine(ProtocolVersion ver, int code);

    /**
     * Sets the status line of this response with a reason phrase.
     *
     * @param ver       the HTTP version
     * @param code      the status code
     * @param reason    the reason phrase, or {@code null} to omit
     */
    void setStatusLine(ProtocolVersion ver, int code, String reason);
```

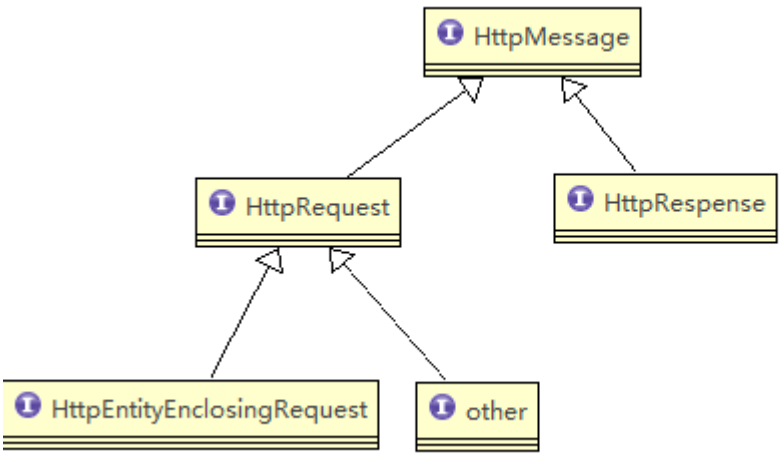
如上图所示，HttpResponse 不同于 HttpRequest，他在继承 HttpMessage 接口的基础上新加入了许多方法，比如 getSatusLine，获取状态行；setStatusLine，写状态行以及图片中没有展示的 setStatusCode，设定状态代码；setReasonPhrase 编写解释行；getEntity 获取消息实体，setEntity 将消息实体编写入消息；getLocale 获取国际化地址，setLocale 设定地区。

作为一个响应消息的接口，HttpResponse 需要从整个系统的其他部分去获取相关信息，完成对于 HttpResponse 消息的搭建。值得注意的是，接口中出现了三种对于 setStatusLine 的重载方法，以适用于不同的相应消息头。而考虑到 Http 的特殊性，代码中还对国际化进行了实现。

```java
public interface HttpEntityEnclosingRequest extends HttpRequest {

    /**
     * Tells if this request should use the expect-continue handshake.
     * The expect continue handshake gives the server a chance to decid
     * whether to accept the entity enclosing request before the possib
     * lengthy entity is sent across the wire.
     * @return true if the expect continue handshake should be used, fa
     * not.
     */
    boolean expectContinue();

    /**
     * Associates the entity with this request.
     *
     * @param entity the entity to send.
     */
    void setEntity(HttpEntity entity);

    /**
     * Returns the entity associated with this request.
     *
     * @return entity
     */
    HttpEntity getEntity();

}
```

对于一些复杂的的请求 Httpcore 提供了 HttpEntityEnclosingRequest 这一接口，它在继承通用的 HttpRequest 接口的基础上又提供了 expectContinue 用于判定是否采用 expect-continue 模式处理消息实体；getEntity、setEntity 两个方法对消息实体操作。

可以看出，对于简单以及复杂两种请求 Httpcore 提供了两个层次的接口，这样的设计使得代码的层次结构更加分明；它们都是 HttpRequest 的接口，但是彼此之间又明显的层次关系，这体现了面向对象程序设计中关键的一个理念或方法：从同一类事物中提取共性，再根据不同进行层次化。而前面介绍过的 HttpResponse 并没有这样的差别。到此为止，HttpMessage 的继承关系我们基本上已经理清楚了，可以用以下图片来表示：



以上都只是提供了对消息的请求和响应以及信息进行操作的接口，并没有展示实现的方法，在 http/message 文件夹中 AbstactHttpMessage 这一抽象类实现了 HttpMessage 这个接口，

并且对内部变量进行了严格的保护，防止外部对其进行修改，保护了数据和程序的安全。BasicHttpRequest 和 BasicHttpResponse 继承了抽象类 AbstactHttpMessage，并且分别实现了 HttpRequest 和 HttpResponse 接口，下面就分别分析一下相关代码。

```
    @Override
    public boolean containsHeader(final String name) {
        return this.headergroup.containsHeader(name);
    }

    // non-javadoc, see interface HttpMessage
    @Override
    public Header[] getHeaders(final String name) {
        return this.headergroup.getHeaders(name);
    }

    // non-javadoc, see interface HttpMessage
    @Override
    public Header getFirstHeader(final String name) {
        return this.headergroup.getFirstHeader(name);
    }

    // non-javadoc, see interface HttpMessage
    @Override
    public Header getLastHeader(final String name) {
        return this.headergroup.getLastHeader(name);
    }

    // non-javadoc, see interface HttpMessage
    @Override
    public Header[] getAllHeaders() {
        return this.headergroup.getAllHeaders();
    }

    // non-javadoc, see interface HttpMessage
    @Override
    public void addHeader(final Header header) {
        this.headergroup.addHeader(header);
    }

    // non-javadoc, see interface HttpMessage
```

如上图所示，对于之前接口中提供的种种方法，在这个抽象类中都进行了 override 操作，而且在这个抽象类中也没有具体实现接口里面的方法，而是调用 headergroup 来实现的。这又是一层封装，对于消息头的实质操作是由 headergroup 来完成的，而上面一层两层抽象类只需要调用 headergroup 来实现就可以了。这样使得代码的层次更加清晰，也体现了封装的思想。headergoup → AbstactHttpMessage → HttpMessage 从底层到顶层，维护的时候也只需要修改 headergroup 的代码，无需对上层进行重构，使得维护方便。

http/message 文件夹下的 BasicHttpRequest 和 BasicHttpResponse 继承了抽象类 AbstactHttpMessage，并且分别实现了 HttpRequest 和 HttpResponse 接口，下面就分析一下 BasicHttpResponse

```
48 public class BasicHttpResponse extends AbstractHttpMessage implements HttpResponse {
49
50     private StatusLine          statusline;
51     private ProtocolVersion     ver;
52     private int                 code;
53     private String              reasonPhrase;
54     private HttpEntity          entity;
55     private final ReasonPhraseCatalog reasonCatalog;
56     private Locale              locale;
57
```

其变量依次为状态行 statusline，协议版本 ver，协议编码 code，原因 reasonPhrase，消息实体 entity，原因目录 reasonCatalog，国际化 locate。这些变量都是 private，只能通过内部方法访问。

```
69    public BasicHttpResponse(final StatusLine statusline,
70                             final ReasonPhraseCatalog catalog,
71                             final Locale locale) {
72        super();
73        this.statusline = Args.notNull(statusline, "Status line");
74        this.ver = statusline.getProtocolVersion();
75        this.code = statusline.getStatusCode();
76        this.reasonPhrase = statusline.getReasonPhrase();
77        this.reasonCatalog = catalog;
78        this.locale = locale;
79    }
80
```

```
88    public BasicHttpResponse(final StatusLine statusline) {
89        super();
90        this.statusline = Args.notNull(statusline, "Status line");
91        this.ver = statusline.getProtocolVersion();
92        this.code = statusline.getStatusCode();
93        this.reasonPhrase = statusline.getReasonPhrase();
94        this.reasonCatalog = null;
95        this.locale = null;
96    }
```

```
108    public BasicHttpResponse(final ProtocolVersion ver,
109                             final int code,
110                             final String reason) {
111        super();
112        Args.notNegative(code, "Status code");
113        this.statusline = null;
114        this.ver = ver;
115        this.code = code;
116        this.reasonPhrase = reason;
117        this.reasonCatalog = null;
118        this.locale = null;
119    }
```

对应不同的 Http 消息，HttpResponse 有三种不同的构建方式，其中协议版本，协议编码和原因是必要参数。因为这些内容都在状态行中，所以当输入数据中有状态行，就可以调用这些内容，其余的信息和输入的其他数据有关。

对于获取状态行函数 getStatusline

```
130    public StatusLine getStatusLine() {
131        if (this.statusline == null) {
132            this.statusline = new BasicStatusLine(
133                    this.ver != null ? this.ver : HttpVersion.HTTP_1_1,
134                    this.code,
135                    this.reasonPhrase != null ? this.reasonPhrase : getReason(this.code));
136        }
137        return this.statusline;
138    }
```

如果状态行为空，就构造一个新的状态行，并将协议版本改为默认版本（HTTP 1.1），也应证了最开始所说的当今默认的最常用的版本就是 HTTP 1.1

BasicHttpResponse 中还有其他许多函数，其作用就是设置这些 private 变量，还有很多细节的东西，由于时间关系我并没有一个个去读。

之前也说过 header 中的信息是 HttpMessage 中非常重要的一个部分

```java
public interface Header extends NameValuePair {

    /**
     * Parses the value.
     *
     * @return an array of {@link HeaderElement} entries, may be empty, but is never {@coc
     * @throws ParseException in case of a parsing error
     */
    HeaderElement[] getElements() throws ParseException;

}
```

Header 这个接口十分简单，直接继承的 NameValuePair，所以说 header 就是一个 Name—Value。

对于 Header 的实现在 http/message 的 BasicHeader 中定义了一个类

```java
public class BasicHeader implements Header, Cloneable, Serializable {

    private static final HeaderElement[] EMPTY_HEADER_ELEMENTS = new HeaderElement[] {};

    private static final long serialVersionUID = -5427236326487562174L;

    private final String name;
    private final String value;
```

BasicHeader 继承了 Header 的接口也继承了 Cloneable，Serializable，可见消息头是可以拷贝，序列化的。

```java
    @Override
    public HeaderElement[] getElements() throws ParseException {
        if (this.getValue() != null) {
            // result intentionally not cached, it's probably not used again
            return BasicHeaderValueParser.parseElements(this.getValue(), null);
        }
        return EMPTY_HEADER_ELEMENTS;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public String getValue() {
        return value;
    }
```

BasicHeader 中重写了 getValue 和 getName 来得到 Name-Value 的值，还有 getrElements 来获取 Header 中除了 Name 和 Value 的其他元素，但是如果 Value 值为空，就会告知这个 Header 是空的。

分析完单个 Header 的操作，现在我们来看封装在 HeaderGroup 中的函数对于 Header 的操作。

```java
public class HeaderGroup implements Cloneable, Serializable {

    private static final long serialVersionUID = 2608834160639271617L;

    private static final Header[] EMPTY = new Header[] {};

    /** The list of headers for this group, in the order in which they were added */
    private final List<Header> headers;

    /**
```

同 Header 一样，HeaderGroup 也是可拷贝的，可序列化的，但这不是一个 Header，而是对一个 Header 的一个 List 来进行操作的。这个不难理解，因为一个 Message 中往往含有不止一个 header。

```
67    public void clear() {
68        headers.clear();
69    }
```

```
77    public void addHeader(final Header header) {
78        if (header == null) {
79            return;
80        }
81        headers.add(header);
82    }
```

```
89    public void removeHeader(final Header header) {
90        if (header == null) {
91            return;
92        }
93        headers.remove(header);
94    }
```

```
127   public void setHeaders(final Header[] headers) {
128       clear();
129       if (headers == null) {
130           return;
131       }
132       Collections.addAll(this.headers, headers);
133   }
```

```
    public void updateHeader(final Header header) {
        if (header == null) {
            return;
        }
        // HTTPCORE-361 : we don't use the for-each syntax, i.e.
        //     for (Header header : headers)
        // as that creates an Iterator that needs to be garbage-collected
        for (int i = 0; i < this.headers.size(); i++) {
            final Header current = this.headers.get(i);
            if (current.getName().equalsIgnoreCase(header.getName())) {
                this.headers.set(i, header);
                return;
            }
        }
        this.headers.add(header);
    }
```

其中 clear，add，remove 操作都是可以通过链表来实现的，get，set，update 操作也是直接对链表元素进行操作，都是一些比较常规的操作。但这些方法给 HttpMessage 提供了消息头的调用和操作方法，使得 HttpMessage 在接收到变化后能第一时间修正其 HeaderGroup 下的 Headers。

```
public Header getCondensedHeader(final String name) {
    final Header[] hdrs = getHeaders(name);

    if (hdrs.length == 0) {
        return null;
    } else if (hdrs.length == 1) {
        return hdrs[0];
    } else {
        final CharArrayBuffer valueBuffer = new CharArrayBuffer(128);
        valueBuffer.append(hdrs[0].getValue());
        for (int i = 1; i < hdrs.length; i++) {
            valueBuffer.append(", ");
            valueBuffer.append(hdrs[i].getValue());
        }

        return new BasicHeader(name.toLowerCase(Locale.ROOT), valueBuffer.toString());
    }
}
```

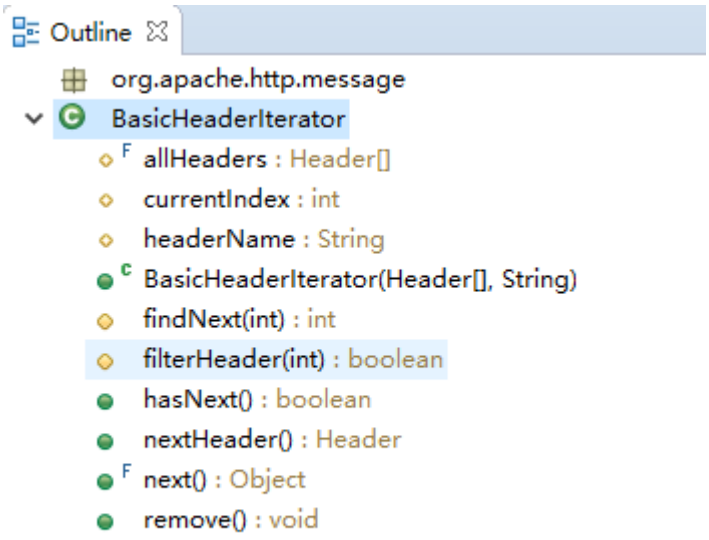
getCondensedHeader 根据传入的 name，把所有具有相同 name 的消息头的 value 拼接起来得到 valueBuffer，然后返回这个 name 和 valueBuffer 组成的新 Header。

以上是 HeaderGroup 中的一些方法。既然 HeaderGroup 是一个 List，那么从中找到某一个消息头一定会有一个遍历的过程。最简单的遍历方法就是直接对 list 进行索引，索引也分外部索引和内部索引。前者会把 list 里面的信息全部暴露出来，不是很安全。后者比前者稍微好一些，但是会使得容器内部十分复杂，不便于管理。所以我们引入了 Iterator 模式来实现遍历的操作。

```
public HeaderIterator iterator() {
    return new BasicListHeaderIterator(this.headers, null);
}
```

```
public HeaderIterator iterator(final String name) {
    return new BasicListHeaderIterator(this.headers, name);
}
```

在 HeaderGroup 中提供了两种 Iterator 的方法，一个是遍历所有的 header，一个是遍历指定 name 的 header。一个 Iterator 的实现类如下图所示



一个 Iterator 至少需要 hasNext 和 Next 两个方法来完成遍历。而对于不同的容器而言，

遍历时的具体操作不同，Iterator 的设计模式就是把便利的方法封装到一个接口中。对于用户来说我们只需要在容器中调通 Iterator 这个接口就可以得到专门的迭代器，从而使用它的遍历方法。

在 Iterator 的帮助下，对于 HeaderGroup 的遍历操作变得简单。Iterator 的设计模式在 Httpcore 中有多次出现，因为 HttpMessage 包含了大量信息，必须用 List、ArrayList 等数据结构收集管理，为了避免并行遍历导致的冲突，模块化各个功能，减少各个组件之间的依赖，使用 Iterator 模式可以最大程度降低程序块之间的耦合度。

5. Http Core 高级设计意图分析

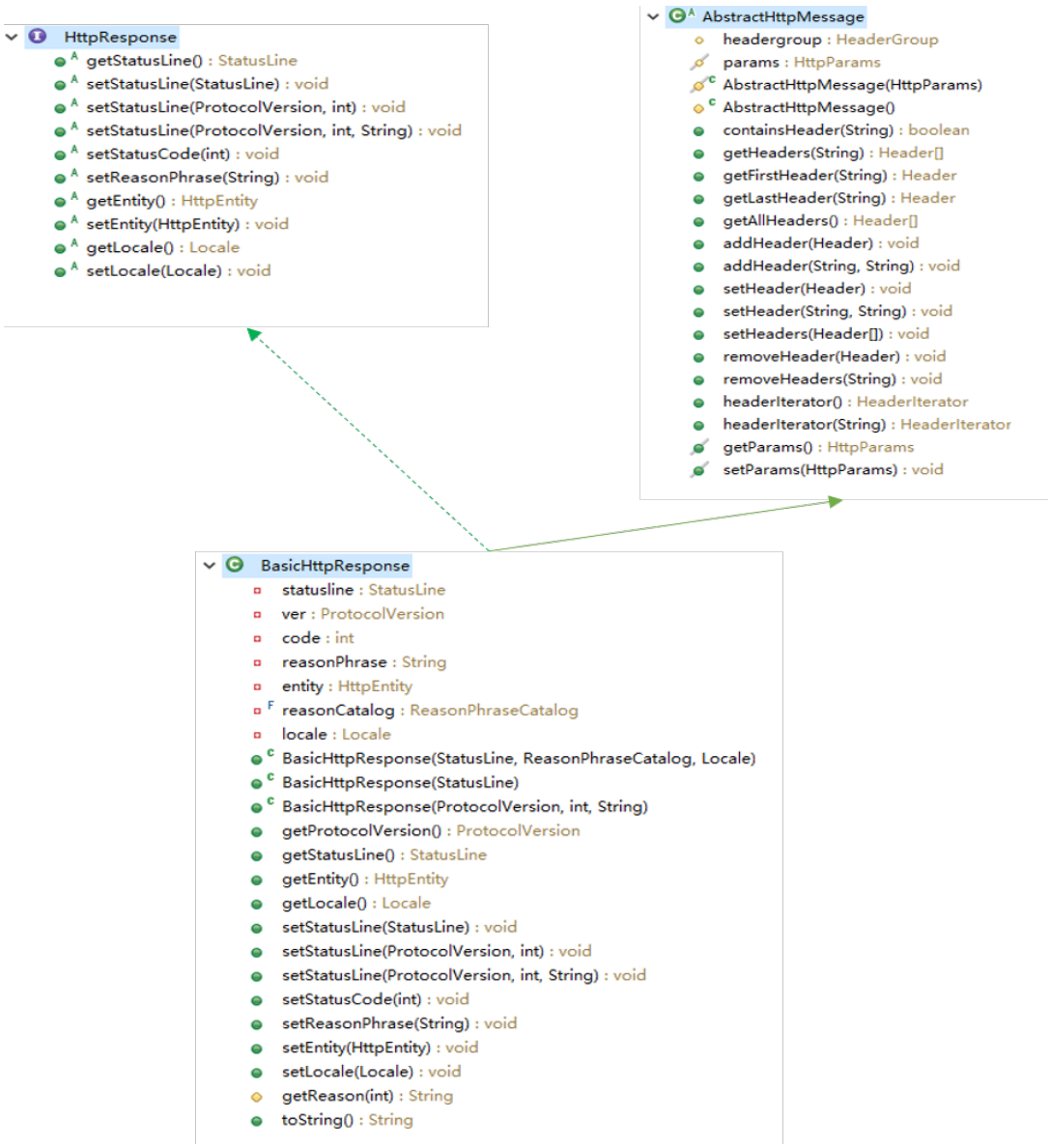
（一）设计模式分析

1. 适配器模式

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。

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

之前也分析过 BasicHttpResponse，这次从设计模式的角度来看，他继承了一个抽象类 AbstractHttpMessage 和一个接口 HttpResponse，形成了一个针对特定应答消息的类。



从图中我们可以看出，BasicHttpResponse 有对方法的重写（getStatusLine），代码中也有继承父类的构造器（super）。对于实际情况来说也是如此，消息答复时需要对消息头的内容进行修改，也需要对状态行的内容进行修改，所以才会实现一个这样更加具体的类。体现了适配器模式在 Http Core 中设计时的优点。

2. 迭代器模式

之前对于源码的分析中已经描述

（二）基于设计原则的分析

上个部分分析了 HttpMessage 的实现过程。接下来从面向对象的程序设计的五个设计原则（单一职责，开放封闭，里氏替换，接口隔离和依赖倒转）出发，分析其中的高级设计意图。

之前在分析 Header 的遍历时提到的 Iterator 模式，就能很好的体现单一职责的原则，Iterator 本身只提供遍历的服务，而且兼容性强，即使外部的环境发生变化，迭代器经过很小

的改动就能满足使用需要。

关于开放封闭方面，Http Core 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。

在 HttpMessage 这个功能的实现中，AbstractHttpMessage 从所有消息中被抽象出来，而其子类 BasicHttpRequest 和 BasicHttpResponse 都能实现 AbstractHttpMessage 中所有的方法。两个子类在编译过程中出现的 override 都是对于接口提供的基于子类情况的特殊方法的重写，父类的方法被完整的继承了下来，满足里氏替换原则。

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，因为可能消息同时包含发送和回答的消息，因此要区分开请求方法和应答方法，所以采用了两层接口。这样 request 接口和 response 接口就不会产生干扰。满足接口隔离和依赖倒转的原则

（三） 并发实现

```
public class HeaderGroup implements Cloneable, Serializable {
```

```
public class BasicHeaderElement implements HeaderElement, Cloneable {
```

```
public class BasicRequestLine implements RequestLine, Cloneable, Serializable {
```

在很多类中都继承了 Cloneable 的接口，说明是支持多线程的。

6. 后记

由于本学期大部分时间献给了操作系统和体系结构的实验，再加上个人所掌握 java 的知识和在阅读源码能力的不足，则学期仅对 httpcore 中的 HttpMessage 功能的实现进行了分析，

理解。

Httpcore 中还定义了客户端与服务器的连接方法（阻塞、非阻塞)、协议版本控制、连接池管理和安全套接层 SSL 等功能。由于时间关系也只是在搜索资料时看了一眼，并没有深入地去研究代码。

Http 项目对后续课程中的计算机网络可能也会有所帮助（猜测)，以后有时间会继续研究。