Httpcore主要功能代码阅读和设计意图分析

2016K8009908053

1603班 孙越

1. **HTTP是什么**

根据RFC2616的定义，超文本传输协议（HTTP）是一种为共有的超文本信息传输系统定制的应用层协议。HTTP是一个客户端和服务器端请求和应答的标准（Transmission Control Protocol, TCP）。客户端通过使用Web浏览器、网络爬虫或者其它的工具发起一个到服务器上指定端口的HTTP请求。HTTP协议采用了请求/响应模型。整个基本的工作流程是客户端发送一个HTTP请求，说明客户端想要访问的资源和请求的动作，服务端收到请求之后开始处理，并根据请求做出相应的动作访问服务器资源，并以一个状态行作为响应，响应的内容包括消息协议的版本，成功或者错误编码加上包含服务器信息、实体元信息以及可能的实体内容。最后通过发送HTTP响应把结果返回给客户端。

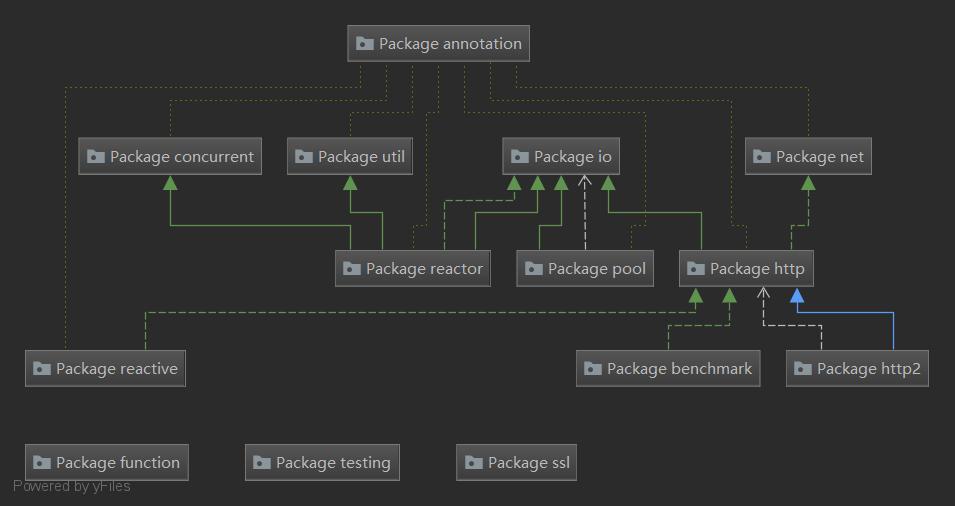
HTTP经历了多个版本，目前最常用的版本为HTTP1.1。版本号在每次HTTP请求的消息中是被要求的，一般处于消息头中。

1. **HttpCore的内容和主要功能**

HttpCore是基于超文本传输协议的一种实现方式，是一套实现了HTTP协议最基础功能的组件。HttpCore可以理解为一种人为编写的应用接口，这种接口实现了用户（应用端）与服务供应商（服务器端）还有中间的代理端之间信息交互的一致性，有效确保了信息传递的效率和安全性。

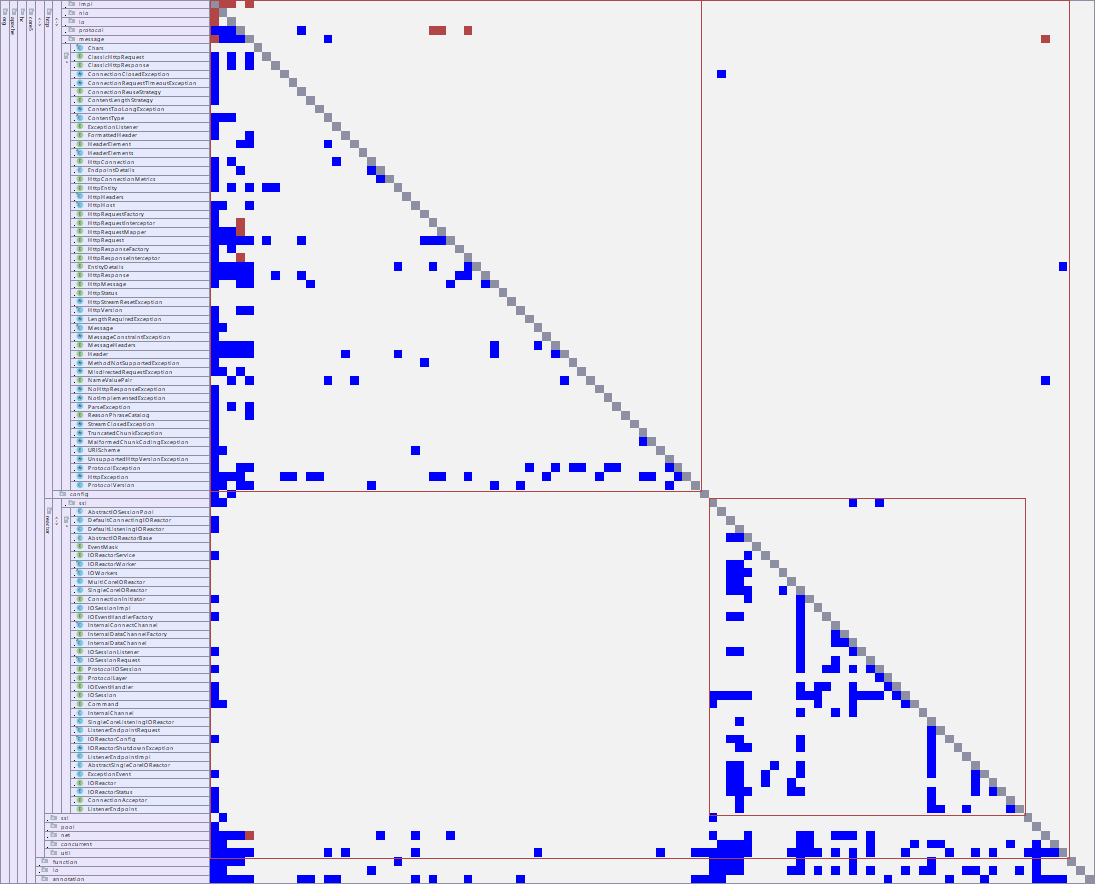
HttpCore需要实现最基本的HTTP传输，包括HttpRequest、HttpResponse等，以及具体方法比如GET和POST。这是最简单的模型，同时还要实现这些简单模型简单操作的复杂要求，包括阻塞与非阻塞的I/O模型等。

可以从代码的角度对整个HttpCore项目进行分析。本文针对httpcomponents-core-4.4.x\httpcore\src\main\java\org\apache\http文件夹的代码进行分析，这些代码足够体现HttpCore的主体设计思想、主要需求实现和代码规范，其他文件夹中的代码与分析代码的相关性有限并且有较多重复，将不在本文报告中进行进一步阐述。【注:由于core5结构更清晰，背景介绍用core5代码】



**Figure 1. Relation between packages**

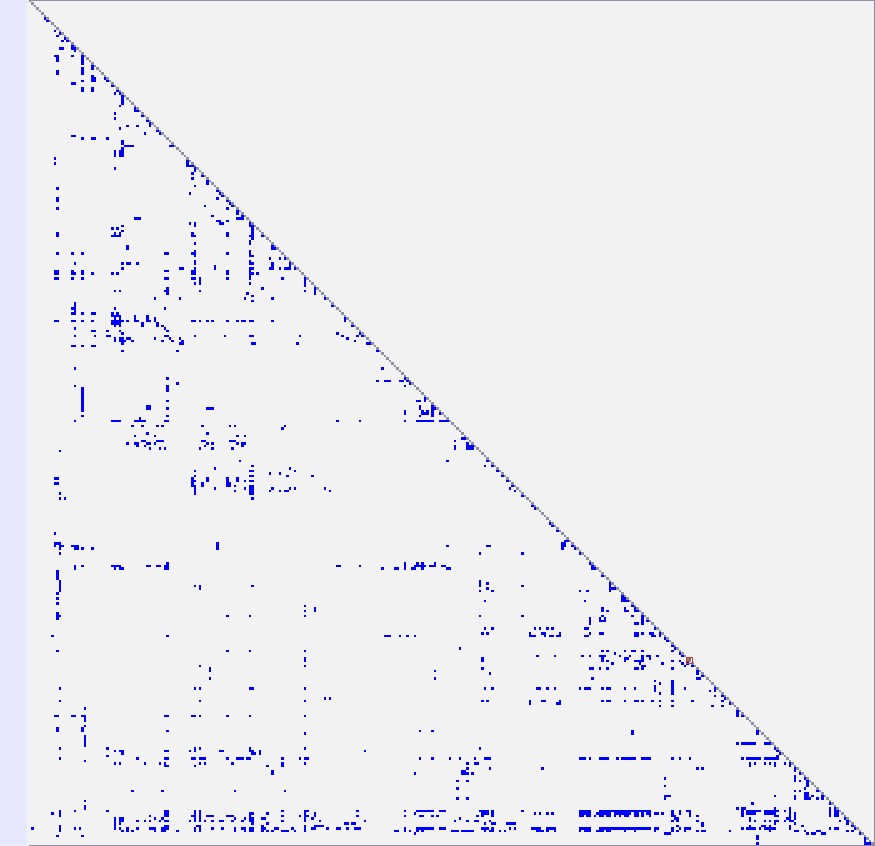
如图为本文分析的文件夹中包含的多个包之间的对应关系，不难发现其中的function、testing和ssl与代码主体的相关性有限。根据UML图对于包之间的描述，包reactor中有对包concurrent、util和io的继承，有对包io的实现；包pool中有对包io的继承和局部变量、方法或者静态方法的调用（依赖关系）；包http有对包io的继承和包net的实现。以上三个包构成了分析问题的核心。下面的三个包reactive、benchmark、http2是对包http的继承和依赖，所以包http是我们分析的重点。



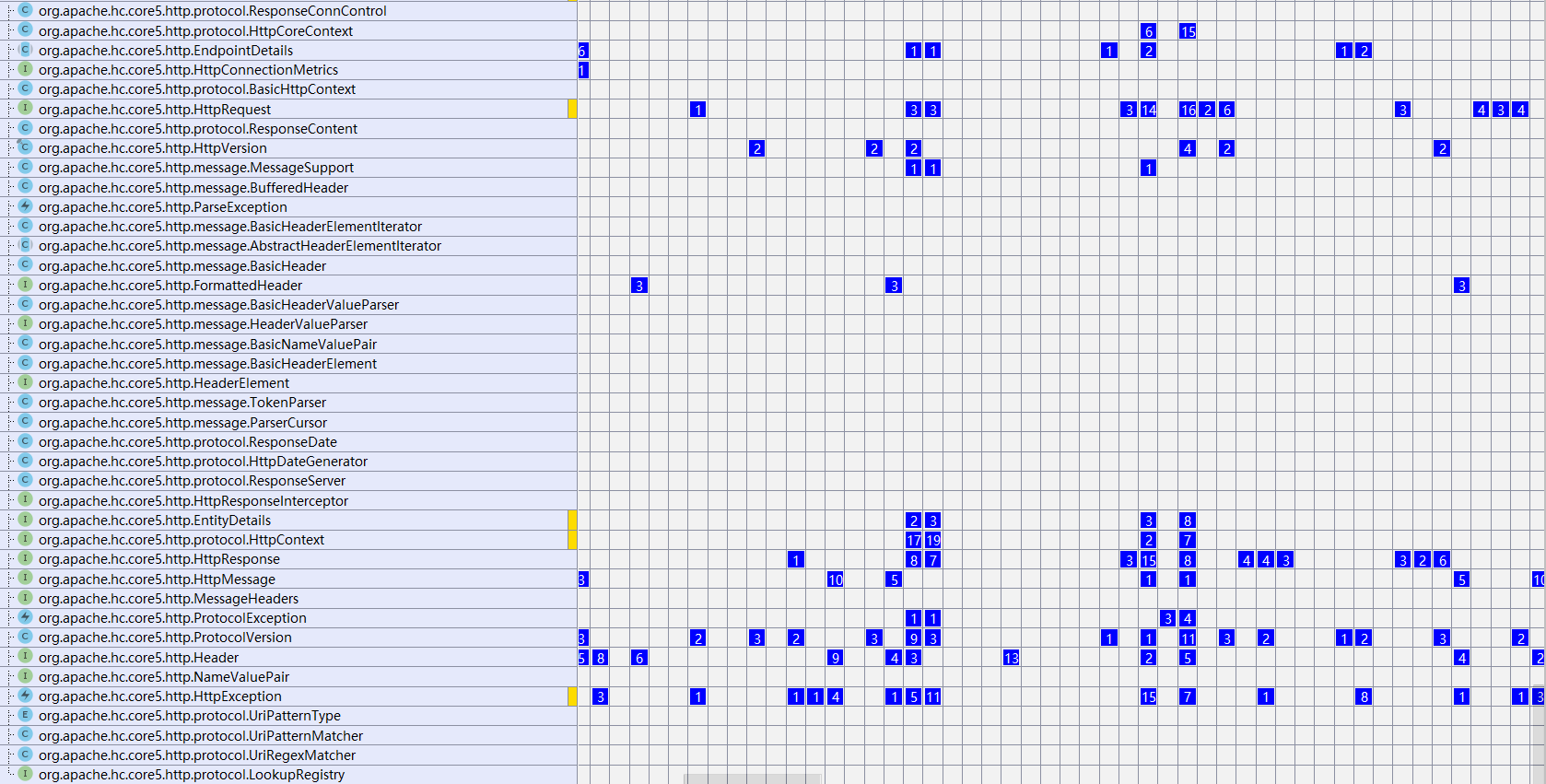
**Figure 2. DSM plot of the main frame----org.apache.hc.core5**

上图是对httpcomponents-core/src/main/java/org.apache.hc.core5/进行依赖分析(dependency structure matrix, DSM)后得到的一张DSM图像。DSM图像可以形象的展现类与类之间、类与接口之间的关系。对于大型复杂系统来说，相比起UML图而言，DSM图对于任务之间的信息交互的展示更加直观，可以明显的体现出依赖关系。比如上图中是将core5核心代码中的两个最主要的package文件夹http和reactor展开之后的结果，可以发现文件的架构呈现出高聚合性和低相关性的特点，即包与包之间的联系较少，而包内联系较为紧密。

下图是对package http的所有接口与类进行依赖矩阵分析之后得到的图。整个结构看上去较为清晰，没有明显的过于依赖其他模块的结构，说明程序设计的基本思路较为清晰，没有冗杂。同时不难发现底层的几行模块被大量其他模块使用。这说明文件整体（或者说HttpCore这个API整体）对于底层几行模块的依赖性较大。我们一般称这类模块为工具模块或者基础模块。这些模块也将成为之后对于程序分析的切入点。



**Figure 3. DSM plot of the main package----org.apache.hc.core5/http**



**Figure 4. The bottom of DSM plot of the main package----org.apache.hc.core5/http**

从上图可以发现那些基础模块包括：HttpMessage, HttpResponse, HttpRequest, HttpHeader, HttpException等，而这些恰好是HttpCore实现的基本功能。

1. **具体功能分析**



**Figure 5. The UML plot of the package of httpcore5**

以上是对于整个http包的UML分析图，其主要功能可以分为以下几块：HttpHeaders，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；HttpEntity，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；HttpException，其主要实现是协议例外。其他的包与整个功能主体的相互关联较小，本文就暂且不做过多的分析。将对每一个功能的具体实现和包之间的相互调用关系进行梳理。

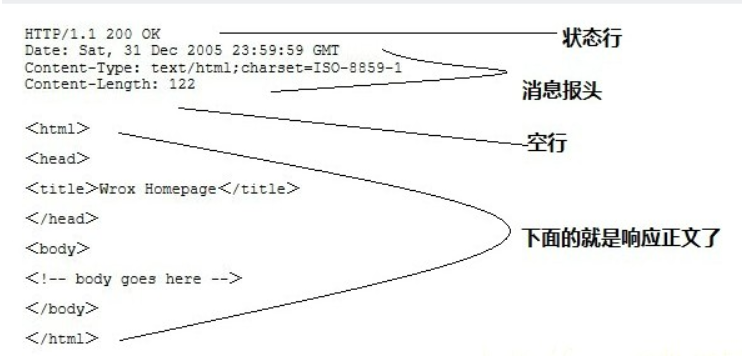
1. **HttpMessage的实现**

作为httpcomponents-core的最重要功能，HttpMessage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。我们先来回顾一下HttpMessage的大概组成：



**Figure 6. The basic structure of Http request message**

上图为请求消息的基本组成，期中请求行的内容是必须的，之后的内容可以视情况省略。



**Figure 7. The basic structure of Http response message**

上图为服务器端给出的响应消息的内容，状态行是必须的，之后的响应正文可以视情况省略。下面我们针对这一功能的实现进行项目的框架和源代码阅读。【作者注：具体功能分析的代码参照httpcorecomponents-4.4.x的代码而非core5，因为现在的发行版是4.4.x，而core5还在开发完善当中。两者在前三节所述的功能上差异不大，但是具体实现稍有差别，在之后会有对比和总结。】

在org.apache.http目录下，有一个HttpMessage接口，含有：

信息处理类方法getProtocolVersion协议版本获取, containsHeader返回消息头；

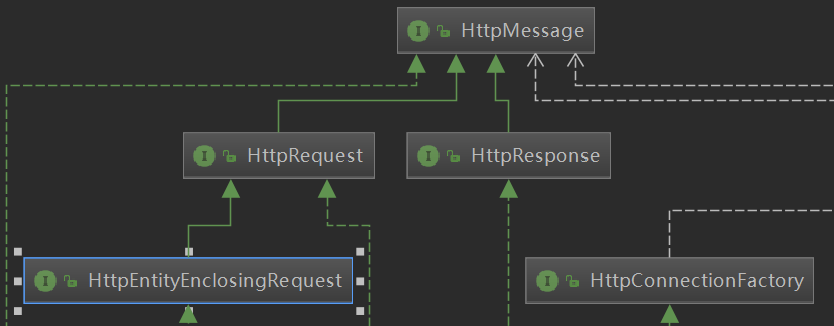
分析消息头信息方法 getHeaders, getFirstHeader, getLastHeader, getAllHeaders；

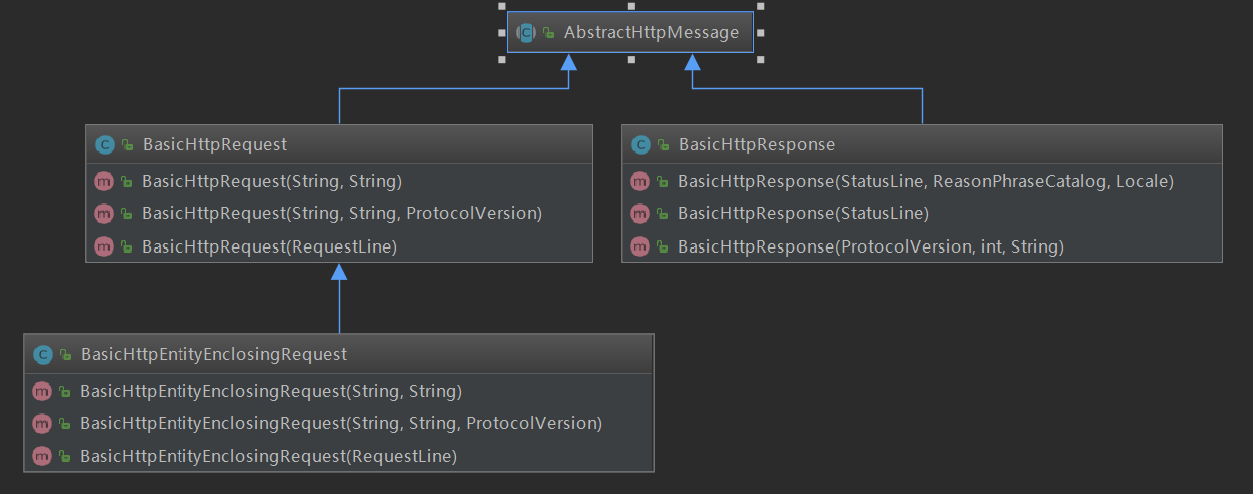
处理消息头方法 addHeader, setHeader, removeHeader, headerIterator。

这个接口体现出HttpMessage的一般操作方法，是对所有消息的抽象，特别是对所有消息头的处理。在这个基础上，有两个接口HttpRequest和HttpResponse继承了这个接口的设计，同时增加了相关消息的特殊方法。HttpRequest只加了一个getRequestLine方法，用于从消息中提取出Http请求行。HttpResponse相比请求消息多了很多其他的内容，包括getStatusLine获取状态行，setStatusLine编写状态行，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。

HttpRequest接口还被HttpEntityEnclosingRequest接口继承，出现了三个特殊方法：expectContinue用于判定是否采用expect-continue模式处理消息实体；setEntity用于将消息实体编入消息；getEntity用于获取消息实体。这个接口用于处理带有实体的请求消息。

这些接口的相互关系在package message中被非常好的继承和实现了，同时被继承的还有这些接口之间的继承关系，如下图所示：





**Figure 8. Relation between Interfaces and (Abstract) Classes**

抽象类AbstactHttpMessage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。

**BasicHttpResponse：**

public class BasicHttpResponse extends AbstractHttpMessage implements HttpResponse {  
  
 private StatusLine statusline;//状态行  
 private ProtocolVersion ver;//协议版本  
 private int code;//协议编码  
 private String reasonPhrase;//原因解释  
 private HttpEntity entity;//消息实体  
 private final ReasonPhraseCatalog reasonCatalog;  
 private Locale locale;//国际化

所有内部变量都限定为private，仅能通过内部方法访问。

public BasicHttpResponse(final StatusLine statusline,  
 final ReasonPhraseCatalog catalog,  
 final Locale locale) {  
 super();  
 this.statusline = Args.*notNull*(statusline, "Status line");  
 this.ver = statusline.getProtocolVersion();  
 this.code = statusline.getStatusCode();  
 this.reasonPhrase = statusline.getReasonPhrase();  
 this.reasonCatalog = catalog;  
 this.locale = locale;  
}

public BasicHttpResponse(final StatusLine statusline) {  
 super();  
 this.statusline = Args.*notNull*(statusline, "Status line");  
 this.ver = statusline.getProtocolVersion();  
 this.code = statusline.getStatusCode();  
 this.reasonPhrase = statusline.getReasonPhrase();  
 this.reasonCatalog = null;  
 this.locale = null;  
}

public BasicHttpResponse(final ProtocolVersion ver,  
 final int code,  
 final String reason) {  
 super();  
 Args.*notNegative*(code, "Status code");  
 this.statusline = null;  
 this.ver = ver;  
 this.code = code;  
 this.reasonPhrase = reason;  
 this.reasonCatalog = null;  
 this.locale = null;  
}

HttpResponse消息的构建函数重载了三次，面向不同的Http相应消息的参数进行构建。一个消息中除了必须要有必要的参数以外（必要的参数是指协议版本、协议编码和原因字段，这些内容都包含于状态行中，所以当输入数据中有状态行时，直接调用状态行这个类的方法来调用这部分内容即可），其余部分可以选择性没有。

下面对于程序中较为值得分析的接口方法实现和内部方法实现进行一些分析。

@Override  
public StatusLine getStatusLine() {//该方法用于获取状态行  
 if (this.statusline == null) {//判定状态行是否为空，若空则构造一个然后返回  
 this.statusline = new BasicStatusLine(  
 this.ver != null ? this.ver : HttpVersion.*HTTP\_1\_1*,//默认版本号为HTTP 1.1  
 this.code,  
 this.reasonPhrase != null ? this.reasonPhrase : getReason(this.code));//如果原因字段为空，那么从编码中获取原因字段，方法见下  
 }  
 return this.statusline;  
}

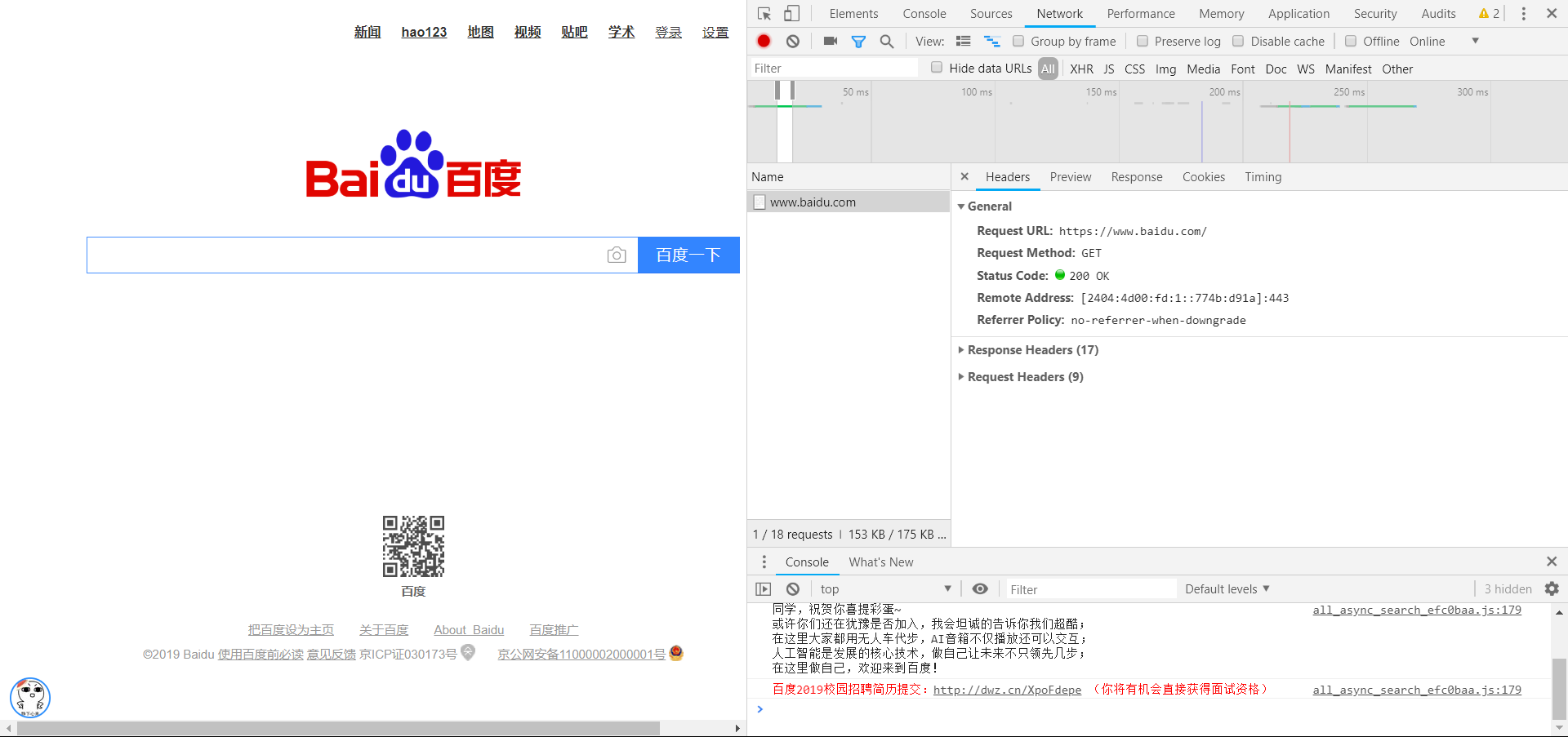
protected String getReason(final int code) {//这是一个内部方法，用于从编码字段中获取原因字段  
 return this.reasonCatalog != null ? this.reasonCatalog.getReason(code,  
 this.locale != null ? this.locale : Locale.*getDefault*()) : null;

//返回原因字段，如果原因目录为空，那么就调用编码和国际化地区编码构建一个原因字段，其中国际化地区编码字段如果为空则采用系统默认编码  
}  
  
@Override  
public String toString() {//这个就比较厉害了，重写了java.lang的Object类，按照HttpRequestMessage的格式构建了这个方法  
 final StringBuilder sb = new StringBuilder();//创建一个字符串  
 sb.append(getStatusLine());//添加状态行  
 sb.append(' ');  
 sb.append(this.headergroup);//添加消息头部的包  
 if (this.entity != null) {//如果有实体的话添加实体  
 sb.append(' ');  
 sb.append(this.entity);  
 }  
 return sb.toString();//返回这整个字符串  
}

以上是对HTTP Message的主要框架的解析。实际上这个框架中还有大量的细节值得去注意，除了以上提到的对于客户端请求的包装处理，对服务器端响应的包装处理以外，还有对于header（请求头数据包）、entity（请求和响应实体）的构造和搭建等。Entity方面，httpcore针对不同的应用场景给出了以下几种实体：InputStreamEntity, SerializableEntity, FileEntity, BasicHttpEntity, StringEntity 和ByteArrayEntity。这些消息实体都继承自抽象类AbstractHttpEntity，有着严格的层次关系。受制于篇幅，对HttpEntity的具体实现不多做阐述。

在一条HTTP Message中，除了请求行的构建，消息头部的组成也是值得认真研读的，因为消息头部的内容决定了客户端和服务器之间的交互行为和交互模式，即客户端和服务器之间有哪些交互，这些交互以何种顺序组织，最终的呈现又是什么样的。下面我们就通过一个生活中的例子来看一下消息头在日常的人机交互过程中是如何运作的。

下图是一张chrome控制台的截图。打开的网页是众人熟知的国产广告引擎百度。除了控制台的广告以外，不难发现在network一栏下对于一个特定的消息名称“www.baidu.com”有相应的多个headers。在Http里面有对应的HeaderGroup用于储存和管理这些Headers，其中还有部分设计模式的体现。



**Figure 9. Chrome network record of ‘www.baidu.com’ (Headers)**

头部分为三块，不难发现通用头部（General）就是之前我们分析过的请求行的相关内容，所以研究消息头部最重要的内容就是分析在源代码中Response Headers和Request Headers在我们message中是怎么管理的。

在package http下，Header是一个接口，继承了接口NameValuePair，这也决定了Header最基本的形式就是Name-Value对。以下是对Header中可能包含的全部成分的官方注释：

*/\*\*  
 \* One element of an HTTP {****@link*** *Header header} value consisting of  
 \* a name / value pair and a number of optional name / value parameters.  
 \* <p>  
 \* Some HTTP headers (such as the set-cookie header) have values that  
 \* can be decomposed into multiple elements. Such headers must be in the  
 \* following form:  
 \* </p>  
 \* <pre>  
 \* header = [ element ] \*( "," [ element ] )  
 \* element = name [ "=" [ value ] ] \*( ";" [ param ] )  
 \* param = name [ "=" [ value ] ]  
 \*  
 \* name = token  
 \* value = ( token | quoted-string )  
 \*  
 \* token = 1\*&lt;any char except "=", ",", ";", &lt;"&gt; and  
 \* white space&gt;  
 \* quoted-string = &lt;"&gt; \*( text | quoted-char ) &lt;"&gt;  
 \* text = any char except &lt;"&gt;  
 \* quoted-char = "\" char  
 \* </pre>  
 \* <p>  
 \* Any amount of white space is allowed between any part of the  
 \* header, element or param and is ignored. A missing value in any  
 \* element or param will be stored as the empty {****@link*** *String};  
 \* if the "=" is also missing <var>null</var> will be stored instead.  
 \*  
 \** ***@since*** *4.0  
 \*/*

一个基本的Header的实现见class **BasicHeader：**

public class BasicHeader implements Header, Cloneable, Serializable {  
//实现了Header接口  
 private static final HeaderElement[] *EMPTY\_HEADER\_ELEMENTS* = new HeaderElement[] {};  
  
 private static final long *serialVersionUID* = -5427236326487562174L;  
  
 private final String name;  
 private final String value;//将name-value存为private有助于保护数据安全  
public BasicHeader(final String name, final String value) {  
 this.name = Args.*notNull*(name, "Name");  
 this.value = value;  
 }//根据所给的NameValuePair提供基础消息头的构造方法  
  
 @Override  
 public Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }//提供复制消息头的方法，继承自JDK提供的Cloneable接口，对特殊情况作出特判  
  
 @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*;  
 }//从外部获取消息头除String name 和 String value 外的其他元素，可能为空，如果无法获取则返回错误信息。  
  
 @Override  
 public String getName() {  
 return name;  
 }//提供接口给外部变量，在保证数据交互的同时保证数据安全。  
  
 @Override  
 public String getValue() {  
 return value;  
 }  
  
 @Override  
 public String toString() {  
 // no need for non-default formatting in toString()  
 return BasicLineFormatter.*INSTANCE*.formatHeader(null, this).toString();  
 }  
  
}

Httpcore提供了Header的所有name字符串，在类HttpHeaders中。这个类是不可实例化的，它的作用就是定义所有消息头的名称。

以上是对单个Header的介绍。但是在一条HTTP Message中，往往含有不止一个header，所以在组织框架中消息头通常以HeaderGroup的形式存在。值得注意的是，HeaderGroup是一个列表（List），元素是按顺序存储的，所以不会乱，同时这个列表中可以有重复变量。列表中的变量类型是Header接口而非具体的Header类。因为除了BasicHeader以外，还有其他种类的Headers，比如说已经被格式化的消息头（**FormattedHeader**）以及继承了这一接口的BufferedHeader。所有的Header都实现了Header这个接口，都属于Header这个类型。从同一类事物中提取出共性然后层次化这是面向对象的程序里面非常常见的方法。下面来看一下HeaderGroup的具体代码实现。

public class HeaderGroup implements Cloneable, Serializable {  
  
 private static final long *serialVersionUID* = 2608834160639271617L;  
  
 private final Header[] EMPTY = new Header[] {};//提供一个空的消息头数组  
private final List<Header> headers;//在HeaderGroup中最主要的存储方式是一个Header表，以维护其有序性  
public HeaderGroup() {//实例化函数，创建一个ArrayList，底层的数据结构使用的是数组结构（数组长度是可变的百分之五十延长）（特点是查询很快，但增删较慢）线程不同步  
 this.headers = new ArrayList<Header>(16);  
 }  
public void clear() {  
 headers.clear();  
 }  
public void addHeader(final Header header) {  
 if (header == null) {  
 return;  
 }  
 headers.add(header);  
 }  
public void removeHeader(final Header header) {  
 if (header == null) {  
 return;  
 }  
 headers.remove(header);  
 }  
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);  
 }public void setHeaders(final Header[] headers) {  
 clear();  
 if (headers == null) {  
 return;  
 }  
 Collections.*addAll*(this.headers, headers);  
 }

以上提供了HeaderGroup对的header的基本操作：addHeader, removeHeader, updateHeader和setHeaders。这些方式为外部提供了管理、调用Headers的接口，同时配合外部的HttpMessage实现了Observer行为模式。Observer模式是一种一对多的对象之间的关系，存在一个对象（Subject）和对应的多个观察者（Observer），类似于日常生活中的订阅，当Subject有更新时，自动通知（update）所有变量或者相关的变量（拥有相同“兴趣标签”的变量），确保所有相关成员变量保持一致。虽然只是实现了基本的Observer模式，但是这样的集中管理模式让层次更加清楚。这些方法为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。

下面是对于Header获取的一些接口，对应于不同的应用场景有不同的消息头获取方法：getCondensedHeader、getHeaders、getFirstHeader、getLastHeader，具体功能见注释。

public Header getCondensedHeader(final String name) {

//取出一个具有代表性的消息头，如果有多个同名的消息头，将其中的内容提取出来，中间用逗号间隔，以字符串的形式返回。  
 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());  
 }  
 }  
public Header[] getHeaders(final String name) {//获取对应名字的消息头  
 List<Header> headersFound = null;  
 // 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 header = this.headers.get(i);  
 if (header.getName().equalsIgnoreCase(name)) {  
 if (headersFound == null) {  
 headersFound = new ArrayList<Header>();  
 }  
 headersFound.add(header);  
 }  
 }  
 return headersFound != null ? headersFound.toArray(new Header[headersFound.size()]) : EMPTY;  
 }  
public Header getFirstHeader(final String name) {

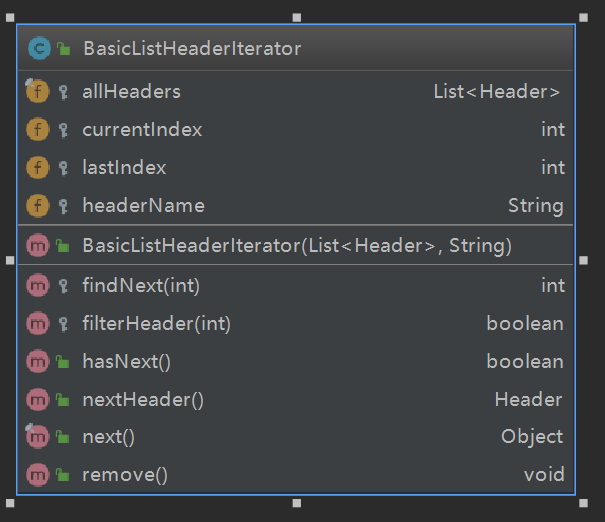
//获取队列中第一个同名消息头  
 // HTTPCORE-361 : we don't use the for-each syntax, i.e.  
 // for (Header header : headers)  
 // as that creates an Iterator that needs to be garbage-collected  
 for (int i = 0; i < this.headers.size(); i++) {  
 final Header header = this.headers.get(i);  
 if (header.getName().equalsIgnoreCase(name)) {  
 return header;  
 }  
 }  
 return null;  
 }  
public Header getLastHeader(final String name) {

//获取表中最后一个同名消息头  
 // start at the end of the list and work backwards  
 for (int i = headers.size() - 1; i >= 0; i--) {  
 final Header header = headers.get(i);  
 if (header.getName().equalsIgnoreCase(name)) {  
 return header;  
 }  
 }  
  
 return null;  
 }  
 @Override  
 public Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
  
 @Override  
 public String toString() {  
 return this.headers.toString();  
 }  
  
}

以上是HeaderGroup中所有的方法。HeaderGroup作为一个容器，其中包含有所有Headers，如果外界需要在容器中寻找对应的Header，需要对容器进行遍历。一般来说有两种方式实现这个遍历：外界方法直接对list进行一个遍历，但是这样会影响到数据安全，有可能在遍历过程中出现修改数据的行为，这是我们不希望看到的；容器自身带有遍历方法，这样会让容器在提供对外操作接口、维护列表完整性和顺序性的同时还需要承担提供遍历方法的职责，几个职责可能会冲突，而且管理难度增加。这个时候我们需要第二个设计模式：Iterator设计模式。迭代器（Iterator）的作用是为外界的访问需求提供一个光标，以此来实现对于容器内部变量的一个遍历。Iterator本身只需要提供一个能够遍历容器的方法即可。在容器（HeaderGroup）中提供了两种建立Iterator的方法，分别用于遍历所有headers和遍历所有同名headers。如下所示。

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

我们接下来来看一下一个Iterator实现类需要哪些构成：



**Figure 10. UML plot of Class BasicListHeaderIterator**

一个Iterator至少需要hasNext()和Next()两个方法来满足其完成遍历的需求。值得一提的是Iterator模式为同时多组无冲突遍历提供了可能性，因为不同的遍历需要不同的光标，由于遍历是外部单独实现，所以对列表本身来说并没有任何影响。在这个例子中系统给光标较大的权利，使其可以removeHeader。

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。

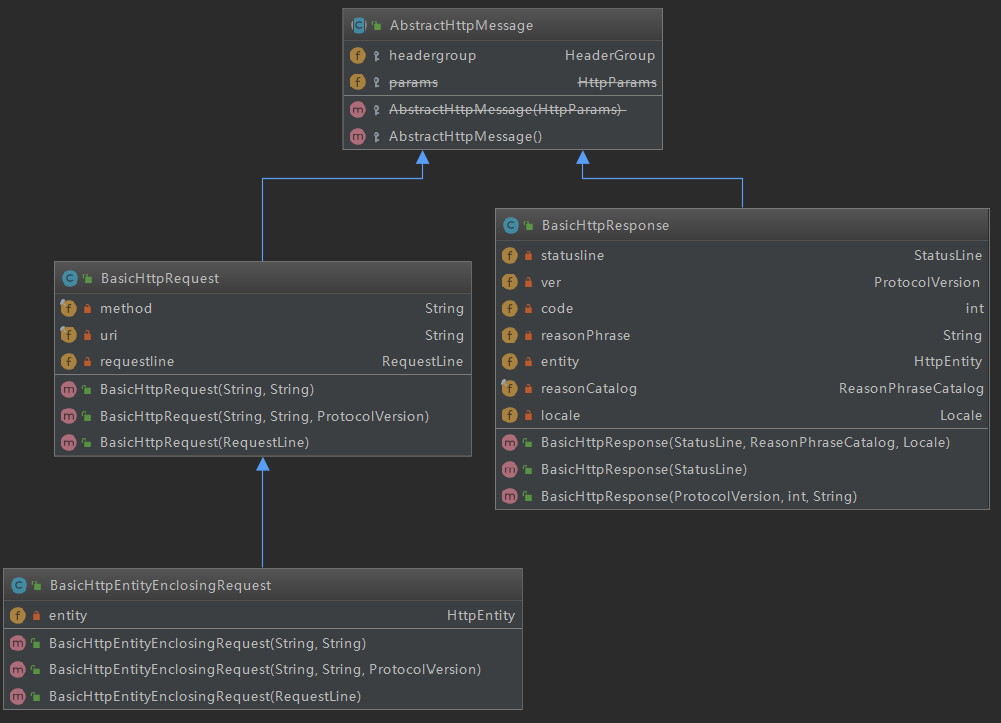
在阅读源代码的过程中，不难发现源代码中有一定数量的对于Oracle提供的Java内核中的方法的重写，比如toString()，hasNext()等，这些方法的重写对于之后构建Http消息有益，可以在不冲突的情况下使代码变得更加容易编写和阅读。

1. **Httpcore高级设计意图分析**

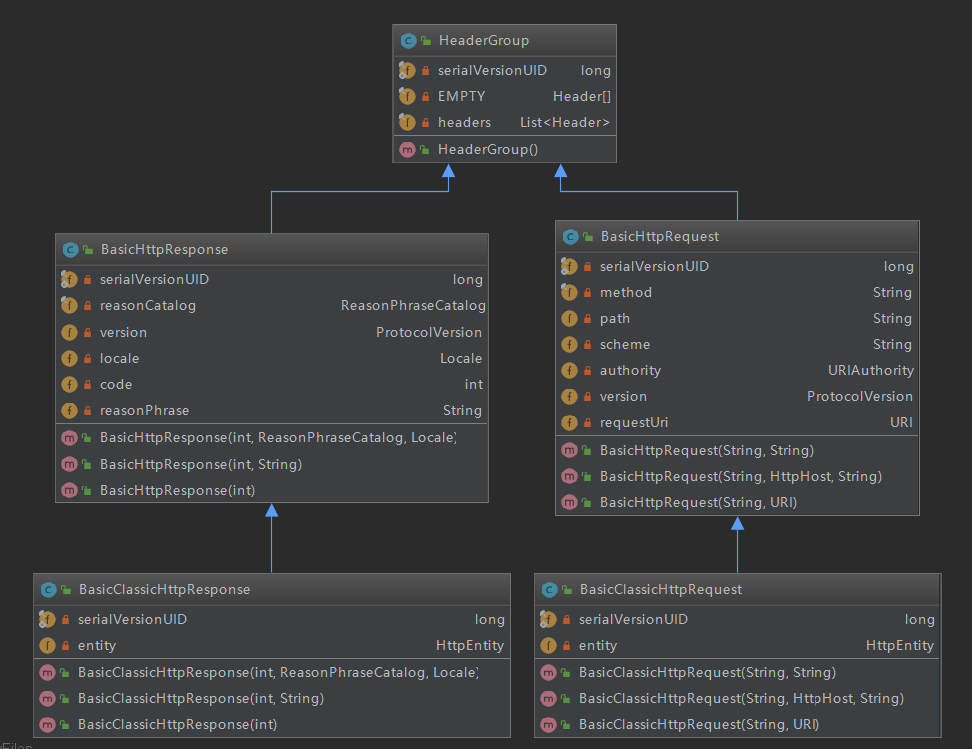
之前我们分析了httpcore中较为重要的HttpMessage的主要实现过程。下面从面向对象的程序设计原则角度观察，希望从中分析出一些面向对象程序设计的基本原则：SOLID五原则，分别是单一职责、开放封闭、里氏替换、接口隔离和依赖倒转。

之前我们在分析源代码的时候所提到的Iterator模式，就很好的体现了单一职责原则。Iterator本身只提供遍历服务，并且兼容性很强，即使外部环境（包括容器内部对象数量、种类等）发生变化，迭代器经过很小的改动甚至不改动就能满足使用需要。

在httpcore中存在大量接口和抽象类，这些接口和抽象类的存在不仅是对基础的一种抽象，更是为之后的扩展提供了可能性。我们可以对比一下前后两个发行版本的代码对于同一个功能的实现。采用的版本分别是httpcomponents-core5和httpcomponents-core-4.4.x，对比的功能均为HttpMessage的实现。



**Figure 11. HttpMessage in core-4.4.x**



**Figure 12. HttpMessage in core-5**

不难发现BasicHttpResponse的成员变量和构造方法基本上没有变化，但是多了一个继承它的实现类BasicClassicHttpResponse，体现了开放封闭的特点。但是这两个版本中的设计模式完全改变了，取消了AbstractHttpMessage这个抽象类，而是用AbstractMessageWrapper实现原来的AbstractHttpMessage的工作，精简了结构。这是设计层面上的改动，与开放封闭这个原则并不违背，因为具体的实现类在不同的环境下发生变动都是正常的。毕竟core5相比于core4.4.x，它对于接口的抽象都不一样，所以会有实现类的改变。

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。

在HttpMessage的源代码中不难发现，绝大部分类都实现了自己的专用接口。最明显的特点是，HttpResponse接口和HttpRequest接口继承自HttpMessage接口，与之相对应，BasicHttpRequest类和BasicHttpResponse类继承自抽象类AbstractHttpMessage，且分别实现了对应的接口。各个类的专用接口出现使得类对于接口的依赖变得有针对性。接口的改变只会影响对应的一个或极少数的类，对整体几乎没有影响。这在代码维护和更新方面是非常重要的，这大大降低了代码维护的成本。同时通过接口之间的关系也能够找到对应的类之间的层次结构，使得代码变得容易理解。

从UML类图（Figure 1）中可以看出，模块（package包）之间的关系基本上是对于接口的继承和实现，只有极少数在特殊情况下是对实际类的直接继承。这样的设计能够最大程度降低系统的耦合性。同时，所有实际类都是对于接口的实现，所以即使在细节上出现改动，接口基本上维持不变，只有实际类的方法部分根据需要改动。这样能够降低维护代码的成本，有效提高代码运维的效率。

在第四部分对于HttpMessage功能的实现中已经结合源代码详细讲述了Observer操作模式和Iterator操作模式，这两个模式对于整个功能的构建和应用具有至关重要的作用。除了上述提到的两种操作型设计模式以外，Httpcore作为一个巨大的项目，在其他功能的实现过程中还应用了工厂模式（Factory Method）以及相关的抽象工厂模式等创建型设计模式。比如在package io中，实现ConnFactory的BasicConnFactory类，HttpMessageWriterFactory接口等。工厂模式的引入极大的提高了httpcore的功能扩展性。不过值得注意的是，随着发行版本的不断更新，底层设计模式的转变和变量设计的变化很大程度上影响了各个类的实现，最明显的例子是自从httpcore4.3版本之后，HttpParams这个变量停用，所有相关的接口、类、方法都被不同程度上的修改甚至废止使用（@Deprecated），所以不同工厂方法的具体实现都受到了不同程度的影响。

Httpcore-4.4.x虽然文件众多，但是对应关系清晰，对于结构的具体分析也较为容易。大多数接口，尤其是最底层的接口，都在主文件夹org.apache.http下，并且这些接口之间的关系决定了实现这些接口之间的类的的关系。其次，几乎每一个类都有自己专用的接口，所以在类的设计上非常明晰，结构层次严谨，降低代码维护工作难度，解决了“牵一发动全身”的问题。再次，包之间的功能较为清晰，例如负责HttpMessage的包与负责HttpConnection的包之间关联较少，降低整体耦合度，避免由于包内接口变化导致的整体连接错误。

Httpcore5的设计理念，相比于core4.4.x更加注重层次性。对比两个主文件夹下的构造不难发现core5更加注重封装。在主文件夹下取消了所有的接口，而是将所有接口都封装到各个包内（见figure 1），使得包的功能被区分，同时尽可能降低包与包之间的耦合性，做到低耦合。而在同一个包内，重新设计的接口使得包内的接口关系更加紧密。虽然仍然保持每个类都有对应的单一功能接口，但是不至于有冗余的成分，整体看起来也更加紧凑，做到高内聚。

当然，将接口分类放在包内的时候就会出现更下一层的结构，也就是包内还会有小packages来实现包内的接口。值得一提的是，尽管第一层我们将部分接口封装到了不同的包内，在实现接口的过程中还是会跨package实现。举个简单的例子，在core5中，package http下的package主要实现了对于HttpMessage的搭建和处理，但是其中不可避免的会有与连接相关的package io，而这个功能在外部的package io和package reactor中都有部分实现。Httpcore作为一个整体，很多功能无法严格拆分开来，如何找到一个合适的平衡点，让功能之间的关系在层次中体现出来，这是目前设计接口遇到的难题。但是对比前后两个版本的差别，我们不难看到社区的工程师们正在努力朝着高内聚低耦合的方向前进，我们有理由相信，无论之后出现什么样的新需求，互联网对于消息的处理都会变得更加及时快速，维护代码的成本也会越来越低，我们正在朝着面向对象程序设计所设想的程序架构前进。

1. **后记**

受制于个人所掌握的知识和在阅读源码上花费的时间，本文仅对httpcore中的HttpMessage功能的实现进行了分析。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。这些功能无一不为我们日常生活提供快捷、稳定、安全的连接，日后有机会会对这部分代码进行更加细致的研读和讨论。由于对于代码的掌握并不全面，所以对于模块（package包）之间的关系掌握的并不是很清晰，也有可能因此对于某些代码提出了错误的看法，恳请各位批评指正！

对于部分源代码的作用解释和分析在注释里，由于源代码是直接从intelliJ中复制下来的，所以全都保留了黑底彩字的格式，并且注释颜色会随着该行最后一个字符的默认颜色改变，对读者的阅读造成困难，在此深表歉意！

**【参考文献】**

[1]<https://www.w3.org/Protocols/rfc2616/rfc2616.html> , last access 2018.10.31 原文如下：“The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.”

[2] Freeman, E., et al. (2004). Head First Design Patterns, O'Reilly Media, Incorporated.