HTTPCORE源码分析

2016K8009929009

秦 宏

2018

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

[写在前面 1](#_Toc531890479)

[一、HTTP初步了解 3](#_Toc531890480)

[（一）概念 3](#_Toc531890481)

[（二）目的 3](#_Toc531890482)

[（三）详细内容 3](#_Toc531890483)

[（四）消息(报文)格式 3](#_Toc531890484)

[（五）HTTP实体 3](#_Toc531890485)

[（六）八种方法 4](#_Toc531890486)

[1.GET 4](#_Toc531890487)

[2.HEAD 4](#_Toc531890488)

[3.POST 4](#_Toc531890489)

[4.PUT 4](#_Toc531890490)

[5.DELETE 5](#_Toc531890491)

[6.TRACE 5](#_Toc531890492)

[7.OPTIONS 5](#_Toc531890493)

[8.CONNECT 5](#_Toc531890494)

[（七）安全方法 5](#_Toc531890495)

[（八）关于HTTPcore（摘自HTTPcore官网） 5](#_Toc531890496)

[1. HTTPcore的范围 5](#_Toc531890497)

[2. HTTP core的目标 5](#_Toc531890498)

[3. HTTPcore不是什么 5](#_Toc531890499)

[二、源代码文件结构解析 6](#_Toc531890500)

[（一） httpcore 6](#_Toc531890501)

[（二） httpcore-ab 6](#_Toc531890502)

[（三） httpcore-nio 6](#_Toc531890503)

[（四） httpcore-osgi 6](#_Toc531890504)

[三、代码具体功能解析 7](#_Toc531890505)

[（一）以request和response message为例分析源码中的数据结构。 7](#_Toc531890506)

[1.request message 7](#_Toc531890507)

[2. response message 8](#_Toc531890508)

[（二）分析三种实体的创建和数据结构 9](#_Toc531890509)

[1. BasicHttpEntity 9](#_Toc531890510)

[2. ByteArrayEntity 9](#_Toc531890511)

[3.StringEntity 10](#_Toc531890512)

[4.InputStreamEntity 10](#_Toc531890513)

[5.FileEntity 11](#_Toc531890514)

[6.HttpEntityWrapper 11](#_Toc531890515)

[7. BufferedHttpEntity 12](#_Toc531890516)

[（三）HTTP协议处理器 12](#_Toc531890517)

[1. 标准协议拦截器 12](#_Toc531890518)

# 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

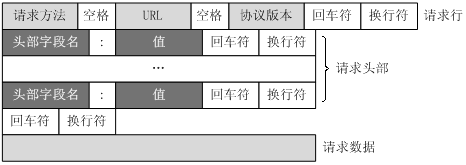
设计HTTP最初的目的是为了提供一种发布和接收HTML页面的方法。通过HTTP或者HTTPS协议请求的资源由统一资源标识符（Uniform Resource Identifiers，URI）来标识。

## （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。

### 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。

### 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)

### 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。

## 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。

## httpcore-ab

该文件下存储的是用于测试httpcore执行的benchmark。

## 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。

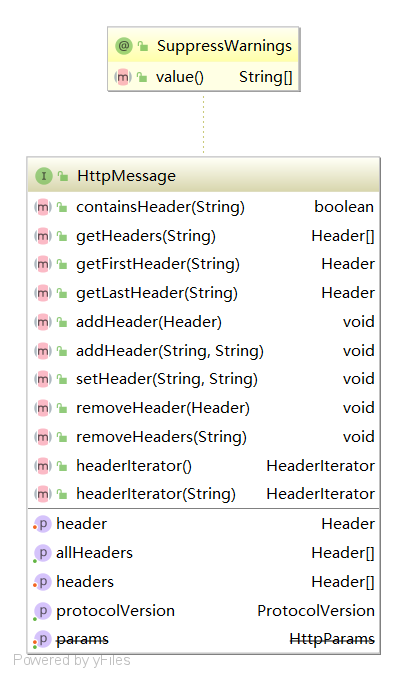
## httpcore-osgi

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

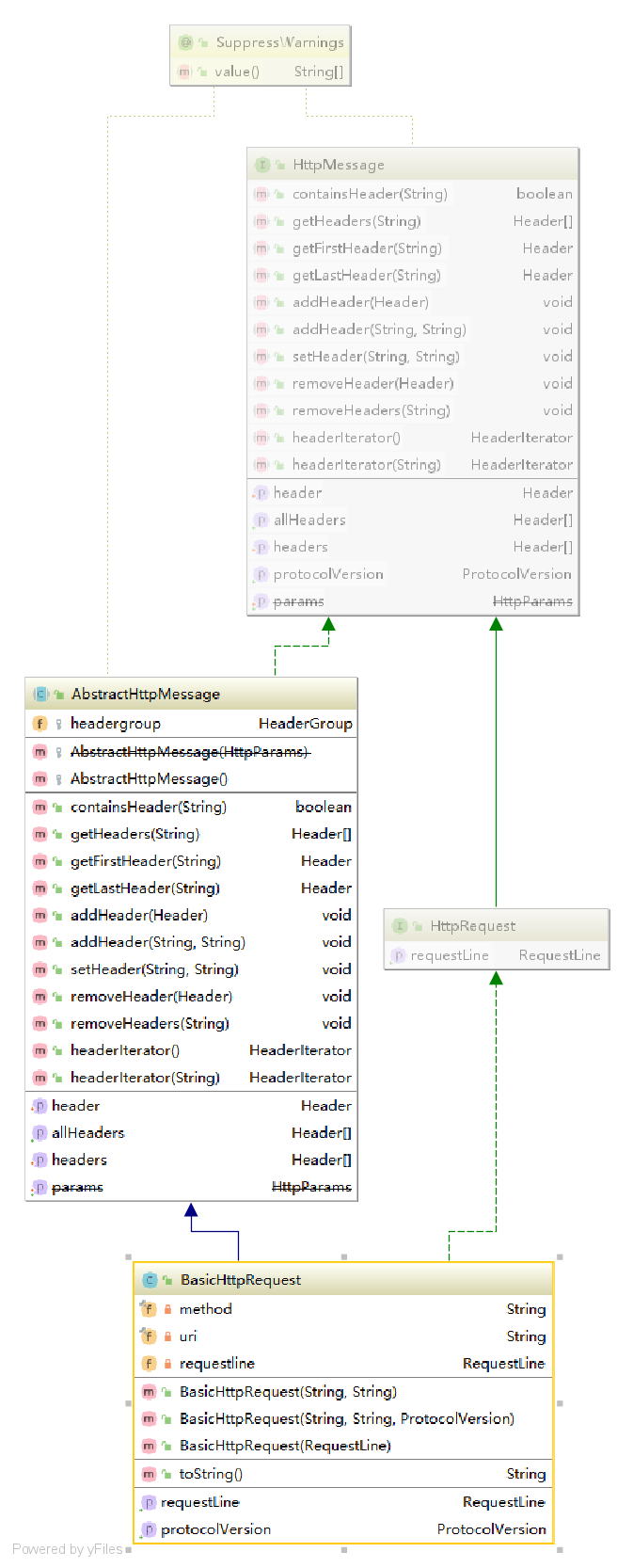
/

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。

BasicRequest中类BasicRequest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、

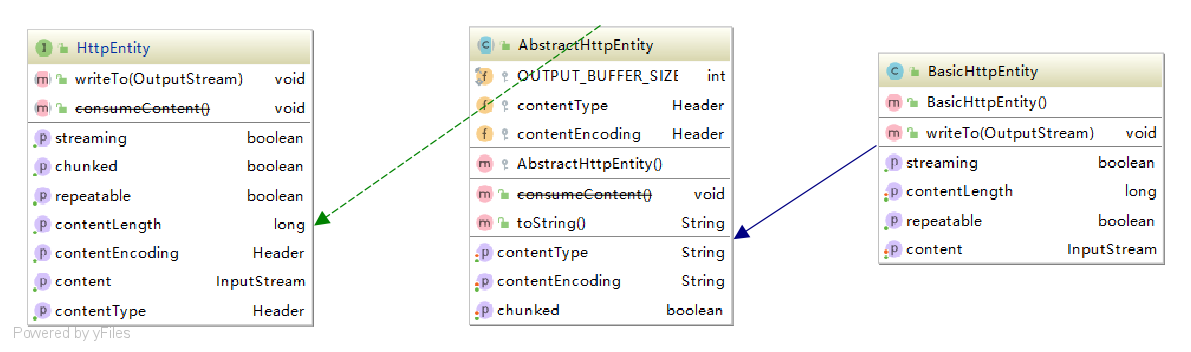
不同于请求消息时输入的参数是满的，这里只输入了ProtocolVersion,code,reasonPhrase.

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

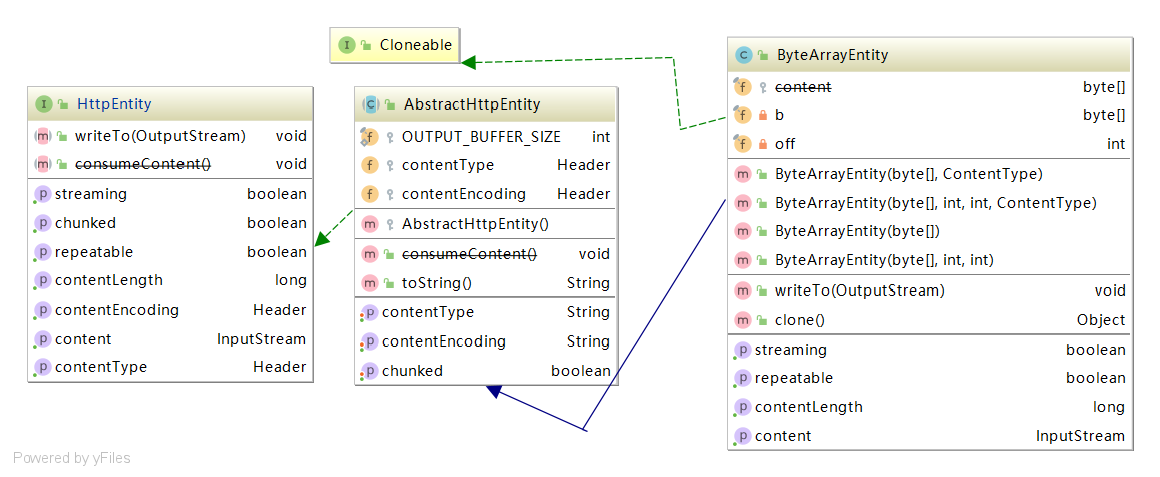
此类用于从HTTP消息接收的实体。该实体具有空构造函数。构建后，它表示没有内容，并且具有负内容长度。



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

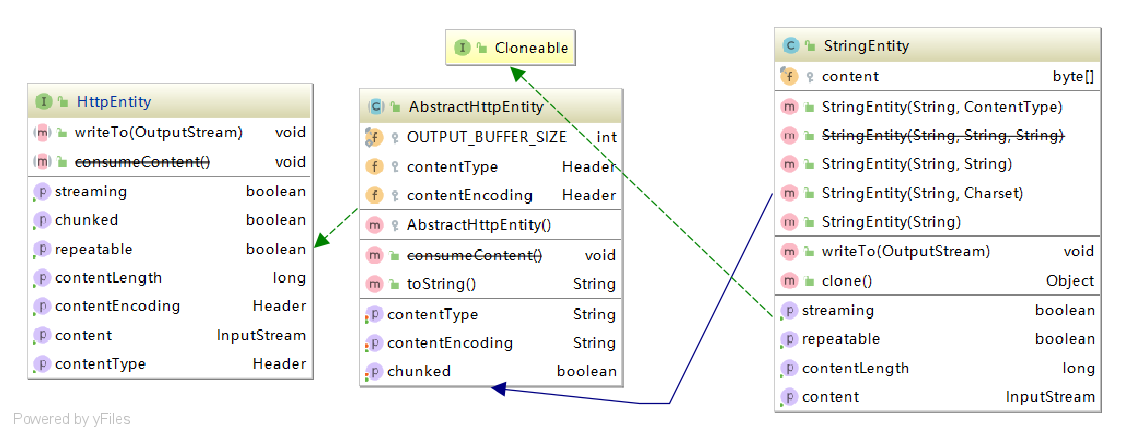
ByteArrayEntity是一个自包含，可重复的实体，从给定的字节数组中获取其内容。将字节数组提供给构造函数。



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

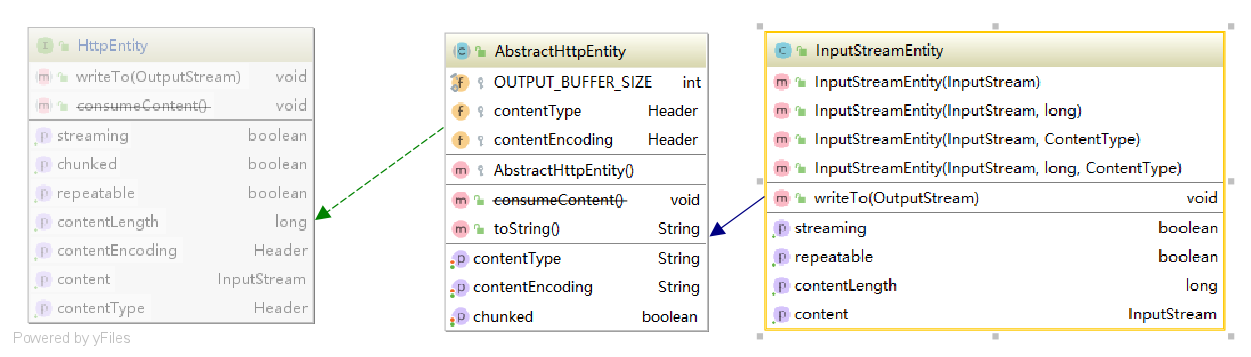
StringEntity是一个自包含，可重复的实体，从java.lang.String对象中获取其内容。它有三个构造方法，一个是用给定的java.lang.String 对象构造; 第二个也对字符串中的数据进行字符编码; 第三个允许指定mime类型。(Multipurpose Internet Mail Extensions)



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

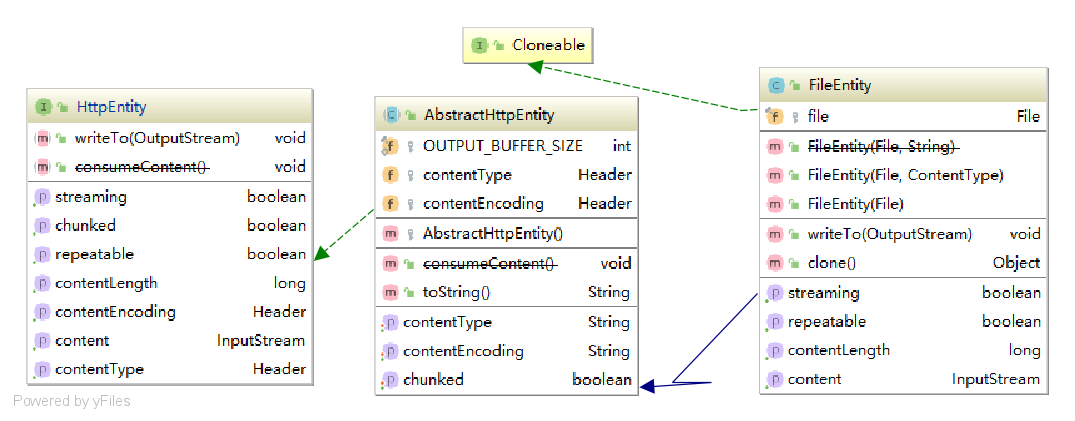
InputStreamEntity是一个流式，不可重复的实体，从输入流中获取其内容。通过提供输入流和内容长度来构造它。使用内容长度来限制从中读取的数据量java.io.InputStream。如果长度与输入流上可用的内容长度匹配，则将发送所有数据。或者，负内容长度将读取输入流中的所有数据，这与提供确切的内容长度相同，因此使用长度来限制要读取的数据量。



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

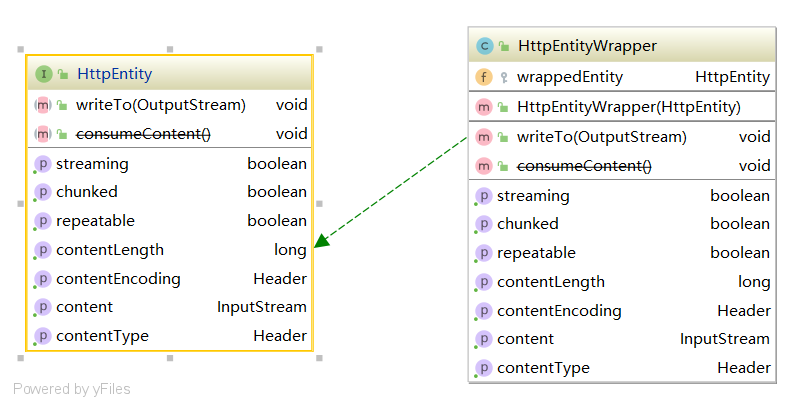
FileEntity是一个自包含，可重复的实体，从文件中获取其内容。主要用于流式传输不同类型的大型文件，需要提供文件的内容类型，例如，发送zip文件需要内容类型 application/zip，用于XML application/xml。



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

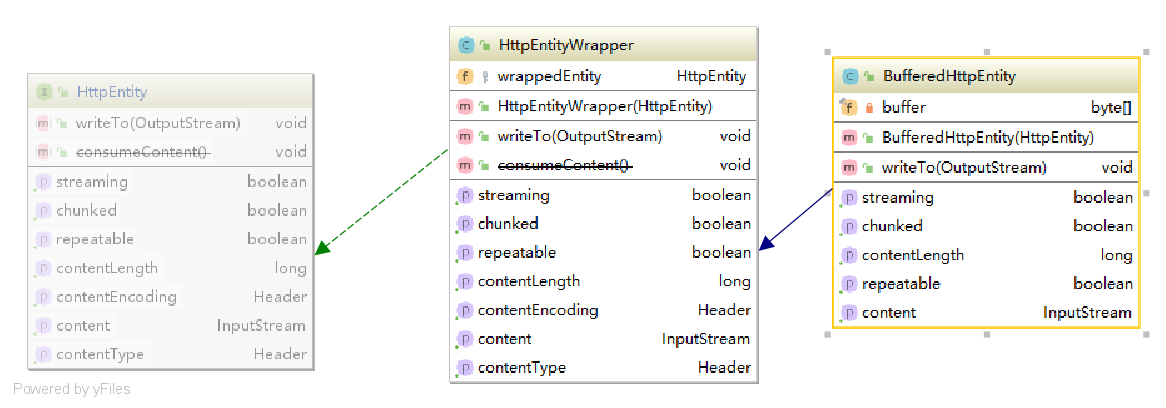
这是用于创建包装实体的基类。包装实体保存对包装实体的引用，并委托对它的所有调用。包装实体的实现可以从此类派生，并且只需要覆盖那些不应该委托给包装实体的方法。



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）。

### 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。

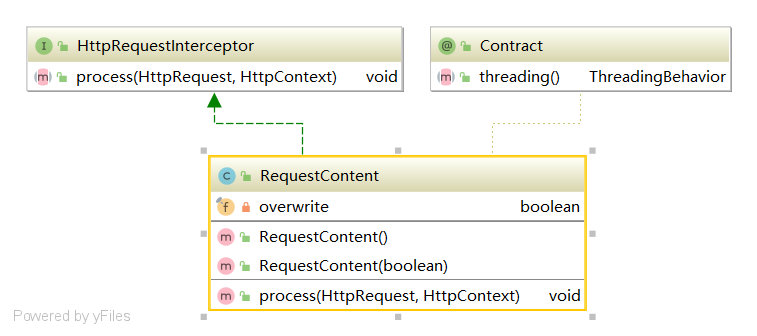
**public interface** HttpRequestInterceptor {  
  
 */\*\*  
 \* Processes a request.  
 \* On the client side, this step is performed before the request is  
 \* sent to the server. On the server side, this step is performed  
 \* on incoming messages before the message body is evaluated.  
 \*  
 \** ***@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。

#### 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* }  
 }  
}

#### ResponseContent

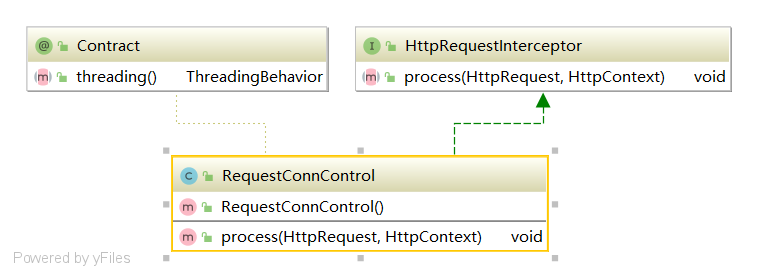
Response的处理位置在服务器端，其主要内容与请求端的操作近似，最后添加了对所请求内容的状态判定，

**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。

#### 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* }  
 }  
  
}

#### 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* }  
 }  
}

#### 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。

#### 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。

#### 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***);  
 }  
 }  
 }  
 }  
}

#### 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。