解析

httpcomponents-core-master

李欢洋
2016K8009929042

目录

1 简介

超文本传输协议(http)是目前互联网上极其普遍的传输协议,它为构建功能丰富，绚丽多彩的网页提供了强大的支持。构建一个网站，通常无需直接操作http 协议，目前流行的 WEB 框架已经透明的将这些底层功能封装的很好了，如常见的 J2EE，.NET，php 等框架或语言。

除了作为网站系统的底层支撑，http 同样可以在其它的一些场景中使用，如游戏服务器和客户端的传输协议、web service、 网络爬虫、HTTP 代理、网站后台数据接口等。

HttpComponents Core 简称 HttpCore，包括用于构建客户机/代理/服务器端 HTTP 服务的一致 API，用于构建同步和异步 HTTP 服务的一致 API 和一组基于阻塞(经典)和非阻塞(NIO) I/O 模型的低级别组件

，可用于以最小的占用空间构建客户端和服务器端 HTTP 服务，支持两种 I/O 模型：阻塞 I/O 模型和和非阻塞 I/O 模型。上层组件(HttpComponents Client，HttpComponents AsyncClient)依赖此组件实现数据传输。

阻塞 I/O 模型基于基本的 JAVA I/O 实现，非阻塞模型基于 JAVA NIO 实现。阻塞 I/O 模型可能更适合于数据密集、低延迟的场景，而非阻塞模型可能更适合于高延迟的场景，在这些场景中，原始数据吞吐量不如以资源高效的方式处理数千个并发 HTTP 连接的能力重要。HttpCore 目标在于，实现最基本的 HTTP 传输方面功能，在良好的性能和 API 的清晰表达之间做到平衡，满足小的(可预测的)内存占用和自包含库(JRE 之外没有外部依赖项).

1.1 HTTP 报文

1.1.1 结构

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

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

1.1.2 基本操作

1.1.2.1 HTTP 请求报文

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

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

输出内容为：

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

1.1.2.2 HTTP 响应报文

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

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

输出内容为：

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

1.1.2.3 HTTP 报文通用的属性和方法

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

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

输出内容为：

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

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

1.1.3 HTTP 实体

HTTP 报文可以携带和请求或响应相关的内容实体。实体可以在一些请求和

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

HttpCore 区分三种类型的实体，这是基于它们的内容是在哪里生成的：

streamed 流式：内容从流中获得，或者在运行中产生。特别是这种分类包含从 HTTP 响应中获取的实体。流式实体是不可重复生成的。

self-contained 自我包含式：内容在内存中或通过独立的连接或其它实体中获得。自我包含式的实体是可以重复生成的。这种类型的实体会经常用于封闭 HTTP 请求的实体。

wrapping 包装式：内容从另外一个实体中获得。

2. Get 请求

Get、Post 是最常见的获取网页内容的请求形式，当然，返回内容并非必须是 html 代码，任何的 xml、json 或文字字符串都可以作为返回内容。

下面是用 Get 请求获取一个 html 网页内容的代码

```
// (1) 创建 HttpGet 实例
HttpGet get = new HttpGet("http://www.126.com");
// (2) 使用 HttpClient 发送 get 请求，获得返回结果 HttpResponse
HttpClient http = new DefaultHttpClient();
HttpResponse response = http.execute(get);
// (3) 读取返回结果
if (response.getStatusLine().getStatusCode() == 200) {
    HttpEntity entity = response.getEntity();
    InputStream in = entity.getContent();
    readResponse(in);
}
```

（1）HttpGet 的实例就是一个 get 请求，构造函数只有一个字符串参数，即要获取的网页地址。另外一种构造形式是使用 URI 实例作为 HttpGet 的参数。HttpComponents 提供了 URIUtils 类，它的 createURI()返回一个 URI 实例。

（2）请求最后被 HttpClient 发送出去，new DefaultHttpClient()创建一个基本的 HttpClient 实例。由于底层是基于阻塞的 JAVA I/O 模型，执行 execute()的时间与具体请求的远程服务器和网络速度有关，在实际运行场景中应特别注意此问题。如果是在 tomcat 等环境中执行可能会造成线程等待，浪费服务器资源，或

拒绝其它的连接。

（3）请求返回后就可以读取返回内容了，但服务器地址错误，或请求的页面不存在等问题都会让请求失败。为了确保得到了正确的响应首先应判断返回码是否正确。调用 response.getStatusLine()返回一个 StatusLine 的实例，此实例描述了一次请求的响应信息。一个成功响应的 StatusLine 实例本身包含如下信息：

HTTP/1.0 200 OK

HTTP/1.0：是请求协议和版本号

200：是响应码

StatusLine 的下面 2 个方法分别用于获取响应信息的各部分内容

getProtocolVersion()： 得到请求协议和协议版本号，如 HTTP/1.0

getStatusCode()：得到响应码，如 200

HttpEntity entity = response.getEntity()返回一个 HttpEntity 实例，进而调用

getContent()就得到了一个输入流。后面的事情应该很明确了。readResponse()是一个自己写的读取输入流中字符串的方法，代码如下：

```
public static void readResponse(InputStream in) throws Exception{
    BufferedReader reader = new BufferedReader(new InputStreamReader(in));
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}
```

3. Post 请求

Post 请求在代码上与 Get 请求的主要区别是将 HttpGet 换成了 HttpPost，其余部分代码基本一致。

```
// (1) 创建 HttpGet 实例
<span style="color: #ff0000;">HttpPost post = new HttpPost("http://www.126.com");</span>
// (2) 使用 HttpClient 发送 get 请求，获得返回结果 HttpResponse
HttpClient http = new DefaultHttpClient();
HttpResponse response = http.execute(<span style="color: #ff0000;">post</span>);
// (3) 读取返回结果
if (response.getStatusLine().getStatusCode() == 200) {
    HttpEntity entity = response.getEntity();
    InputStream in = entity.getContent();
    readResponse(in);
}
```

4.HttpRequest 执行流程

HttpProcessor 是 HTTP protocol processor 的缩写，它的作用就是给我们写的 request 设置默认参数（比如协议版本，是否保持连接等），一般我们写的 HttpRequest 只有 url 和数据，事实上，一个完整的请求不只有这两个数据，查看一下 HTTP 协议就知道一个请求需要很多数据了，而之所以我们不用写这些数据，是因为 HttpCore 或 HttpClient 帮我们完成了，而负责这部分功能的就是

HttpProcessor。HttpRequestExecutor 的作用就是调用 HttpProcessor 完善原始 HttpRequest，根据请求参数使用 HttpClientConnection 执行 HttpRequest 并接受 HttpResponse，以及设置 HttpCoreContext 参数。HttpClientConnection 获取方法有两种：1、自己新建 conn 绑定根据 host 新建的 socket；2、新建 connpool，根据 host 从 pool 中获取 conn。HttpCoreContext：是为了完成会话功能。

5.对 Get 和 Push 功能的分析

5.1 基于 ClassicGetExecutionExample 的分析

在 ClassicGetExecutionExample 这个 public 类中，

```
public static void main(String[] args) throws Exception {
```

告诉编译器这个方法中间可能有些地方要抛出异常，相应的，在代码最后对异常做了处理。

接下来，调用 RequesterBootstrap.bootstrap()方法创建了一个网站请求程序对象 httpRequester ，并对 ClassicGetExecutionExample 继承自其父类 examples 的 onRequestHead、onResponseHead、onExchangeComplete 三个方法进行重写.

```
HttpRequester httpRequester = RequesterBootstrap.bootstrap()
.setStreamListener(new Http1StreamListener() {

@Override
public void onRequestHead(final HttpConnection connection, final HttpRequest request) {
System.out.println(connection + " " + new RequestLine(request));
}

@Override
public void onResponseHead(final HttpConnection connection, final HttpResponse response) {
System.out.println(connection + " " + new StatusLine(response));
}

@Override
public void onExchangeComplete(final HttpConnection connection, final boolean keepAlive) {
if (keepAlive) {
System.out.println(connection + " exchange completed (connection kept alive)");
} else {
System.out.println(connection + " exchange completed (connection closed)");
}
}
})
```

接下来调用配置端口号方法，包括自定义设置、设置超时时间。

```
.setSocketConfig(SocketConfig.custom()
.setSoTimeout(5, TimeUnit.SECONDS)
.build())
```

调用 create()方法创建上下文。创建一个网站主机对象：目标（target）。创建一个串对象：路径（requestUris）。

```
HttpCoreContext coreContext = HttpCoreContext.create();
HttpHost target = new HttpHost("httpbin.org");
String[] requestUris = new String[] {"/", "/ip", "/user-agent", "/headers"};
```

最后，对路径长度建立循环，使用带参数的 try(){}语法，打印除去 host（域名或者 ip）部分的路径->页请求状态码，打印从响应中获得的消息实体。

```
for (int i = 0; i < requestUris.length; i++) {
String requestUri = requestUris[i];
ClassicHttpRequest request = new BasicClassicHttpRequest("GET", target, requestUri);
try (ClassicHttpResponse response = httpRequester.execute(target, request, Timeout.ofSeconds(5),
coreContext)) {
System.out.println(requestUri + "->" + response.getCode());
System.out.println(EntityUtils.toString(response.getEntity()));
System.out.println("==============");
}
}
```

5.2 基于 ClassicPostExecutionExample 的分析

抛出异常，调用 RequesterBootstrap.bootstrap()方法创建对象 ，对继承自其父类的三个方法进行重写等与 Get 实例类似之处就不过多赘述了

```
public class ClassicPostExecutionExample {
public static void main(String[] args) throws Exception {
HttpRequester httpRequester = RequesterBootstrap.bootstrap()
.setStreamListener(new Http1StreamListener() {...
}
})
.setSocketConfig(SocketConfig.custom()
.setSoTimeout(5, TimeUnit.SECONDS)
.build())
.create();
HttpCoreContext coreContext = HttpCoreContext.create();
HttpHost target = new HttpHost("httpbin.org");
```

自然地，更多的关注不同之处。这里创建了一个对象：请求体。属于 http 实体一类。其中包含新创建的串实体，比特队列实体，输入流实体，比特列输入流实体四个对象。

```
HttpEntity[] requestBodies = {
new StringEntity(
"This is the first test request",
ContentType.create("text/plain", StandardCharsets.UTF_8)),
new ByteArrayEntity(
"This is the second test request".getBytes(StandardCharsets.UTF_8),
ContentType.APPLICATION_OCTET_STREAM),
new InputStreamEntity(
new ByteArrayInputStream(
"This is the third test request (will be chunked)"
.getBytes(StandardCharsets.UTF_8)),
ContentType.APPLICATION_OCTET_STREAM)
};
```

5.3 基于 ClassicPostWithTrailersExecutionExample 的分析

与 ClassicPostExecutionExample 相比，ClassicPostWithTrailersExecutionExample 并没有对继承自其父类的三个方法进行重写：

```
public static void main(String[] args) throws Exception {
HttpRequester httpRequester = RequesterBootstrap.bootstrap()
.setSocketConfig(SocketConfig.custom()
.setSoTimeout(5, TimeUnit.SECONDS)
.build())
.create();
```

此外，另一个不同之处是在创建请求体这个对象时，只包含串实体和基态头。

```
HttpEntity requestBody = new HttpEntityWithTrailers(
new StringEntity("Chunked message with trailers", ContentType.TEXT_PLAIN),
new BasicHeader("t1","Hello world"));
```

6. 设计模式分析

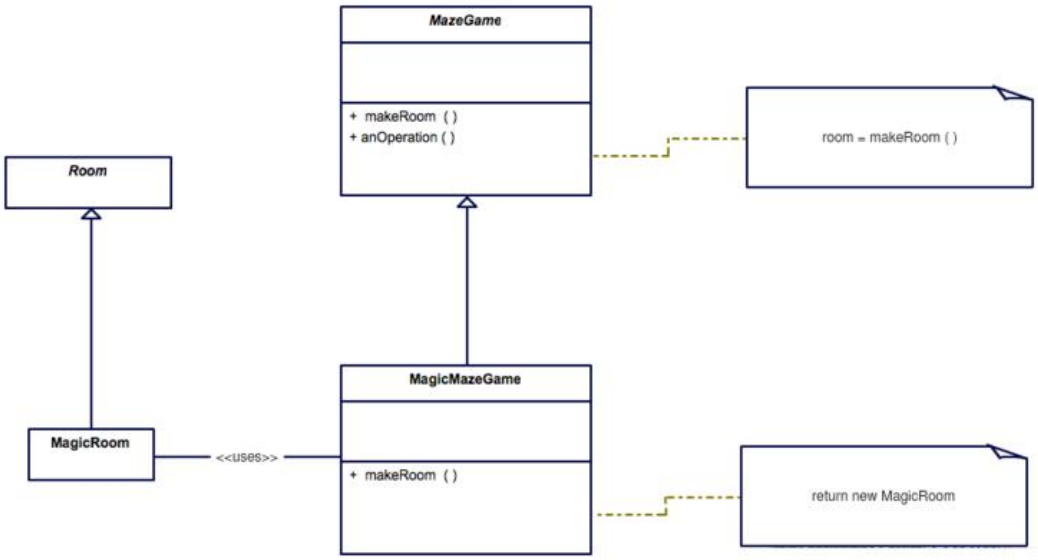
6.1 工厂模式

在基于类的编程中，工厂方法模式是一种创建模式，它使用工厂方法处理创建对象的问题，而不需要指定将要创建的对象的确切类。这是通过调用工厂方法(要么在接口中指定并由子类实现，要么在基类中实现并可由派生类覆盖)来创建对象，而不是调用构造函数。

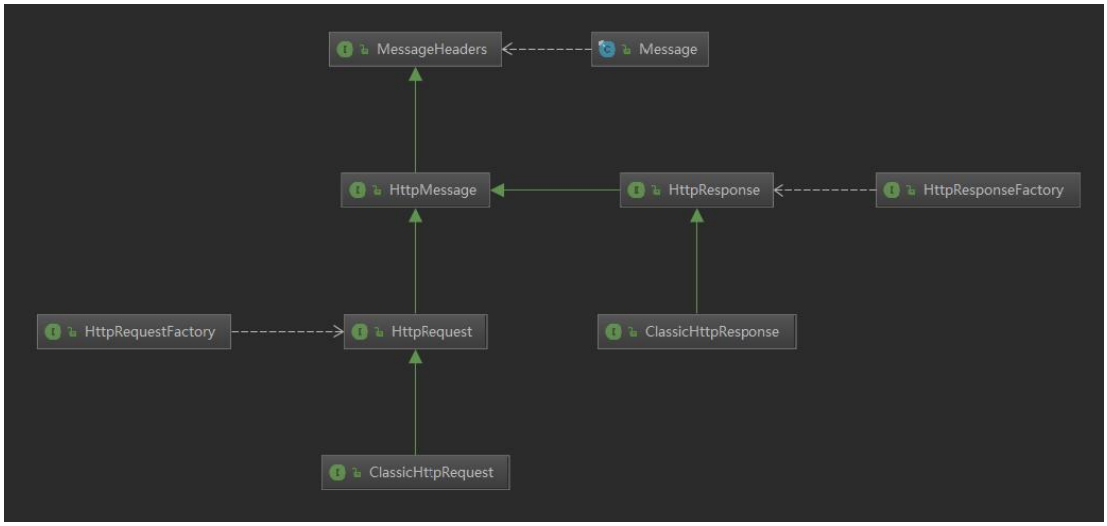
创建对象通常需要复杂的流程，而这些流程不适合包含在组合对象中。对象的创建可能导致代码的大量重复，可能需要组成对象无法访问的信息，可能没有提供足够的抽象级别，或者可能不属于组成对象的关注点。工厂方法设计模式通过定义用于创建对象的单独方法来处理这些问题，然后子类可以覆盖这些方法来指定将要创建的产品的派生类型。

工厂方法模式依赖于继承，因为对象创建被委托给实现工厂方法来创建对象的子类。

结构如下：



httprequest 和 httpresponse 中的工厂模式：



抽象工厂接口 HttpRequestFactory 定义了创建 http 请求的公共接口：newHttpRequest(String method, String uri) 和 newHttpRequest(String method, URI uri) 。

```
public interface HttpRequestFactory<T extends HttpRequest> {
    T newHttpRequest(String method, String uri) throws MethodNotSupportedException;
    T newHttpRequest(String method, URI uri) throws MethodNotSupportedException;
}
```

抽象工厂 HttpRequestFactory 接口的方法在 public 类 DefaultHttpRequestFactory 、DefaultClassicHttpRequestFactory 中分别通过返回 BasicHttpRequest(method, uri)和 BasicClassicHttpRequest(method, uri)对象得以实现：

```
public class DefaultHttpRequestFactory implements HttpRequestFactory<HttpRequest> {
    public static final DefaultHttpRequestFactory INSTANCE = new DefaultHttpRequestFactory();
    @Override
    public HttpRequest newHttpRequest(final String method, final URI uri) {
        return new BasicHttpRequest(method, uri);
    }
    @Override
    public HttpRequest newHttpRequest(final String method, final String uri) {
        return new BasicHttpRequest(method, uri);
    }
```

```
}
```

```
public class DefaultClassicHttpRequestFactory implements HttpRequestFactory<ClassicHttpRequest>
{
    public static final DefaultClassicHttpRequestFactory INSTANCE = new
DefaultClassicHttpRequestFactory();
    @Override
    public ClassicHttpRequest newHttpRequest(final String method, final URI uri) {
        return new BasicClassicHttpRequest(method, uri);
    }
    @Override
    public ClassicHttpRequest newHttpRequest(final String method, final String uri) {
        return new BasicClassicHttpRequest(method, uri);
    }
}
```

与之相关的还有 http 报文语法分析器工厂和报文写入工厂：HttpMessageParserFactory、HttpMessageWriterFactory。

HttpMessageParserFactory 接口定义：

```
public interface HttpMessageParserFactory<T extends MessageHeaders> {
    HttpMessageParser<T> create(H1Config h1Config);
}
```

在 public 类 DefaultHttpRequestParserFactory 中，具体实现了一系列关于 HttpRequestParserFactory 的操作，包含了对于 line 的语法分析结果是否为空的一系列操作，以及对于 requestFactory 产品是否为空的操作：

```
public class DefaultHttpRequestParserFactory implements HttpMessageParserFactory<ClassicHttpRequest> {
    public static final DefaultHttpRequestParserFactory INSTANCE = new DefaultHttpRequestParserFactory();
    private final LineParser lineParser;
    private final HttpRequestFactory<ClassicHttpRequest> requestFactory;
    public DefaultHttpRequestParserFactory(final LineParser lineParser,
            final HttpRequestFactory<ClassicHttpRequest> requestFactory) {
        super();
        this.lineParser = lineParser != null ? lineParser : LazyLineParser.INSTANCE;
        this.requestFactory = requestFactory != null ? requestFactory : DefaultClassicHttpRequestFactory.INSTANCE;
    }
    public DefaultHttpRequestParserFactory() {
        this(null, null);
    }
    @Override
```

```
    public HttpMessageParser<ClassicHttpRequest> create(final H1Config h1Config) {
        return new DefaultHttpRequestParser(this.lineParser, this.requestFactory, h1Config);}}
```

HttpMessageWriterFactory 接口定义：

```
public interface HttpMessageWriterFactory<T extends MessageHeaders> {
    HttpMessageWriter<T> create();
}
```

与报文语法分析器工厂接口的实现与运用类似，HttpRequestWriterFactory 的操作在 public 类 DefaultHttpRequestWriterFactory 中得到实现，包括了对于 lint 格式化的相关操作：

```
public class DefaultHttpRequestWriterFactory implements HttpMessageWriterFactory<ClassicHttpRequest> {

    public static final DefaultHttpRequestWriterFactory INSTANCE = new DefaultHttpRequestWriterFactory();

    private final LineFormatter lineFormatter;

    public DefaultHttpRequestWriterFactory(final LineFormatter lineFormatter) {
        super();
        this.lineFormatter = lineFormatter != null ? lineFormatter : BasicLineFormatter.INSTANCE;
    }

    public DefaultHttpRequestWriterFactory() {
        this(null);
    }

    @Override
    public HttpMessageWriter<ClassicHttpRequest> create() {
        return new DefaultHttpRequestWriter(this.lineFormatter);
    }

}
```

HttpCore 在实现对 http 报文、请求、相应的操作的同时，还被定义为一种基础框架，以方便其他模块对功能进行扩展。大量使用的工厂模式便是为了使开发者扩展起来方便。Httpcore 本身提供的实现是很少的，这也是为什么大量的类和接口的名字中含有“Default”而并没有没有其它对应非“Default”的内容的原因。开发者在实现 httprequest 时，只需要定义好和 http 请求相关的接口，

继承 HttpRequest 对象即可扩展应用。工厂设计模式这种解耦模式的框架设计解决了复杂对象的创建过程。对于上层调用来说，不需要了解工厂内部的复杂生产流程，只需要调用工厂进行生产即可。

6.2 建造者模式

建造者模式是将一个复杂的对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。建造者模式与工厂模式的区别在于建造者模式更加关注与零件装配的顺序。工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性，其实建造者模式就是前面抽象工厂模式和最后的 Test 结合起来得到的。

建造者模式通常包括下面几个角色：

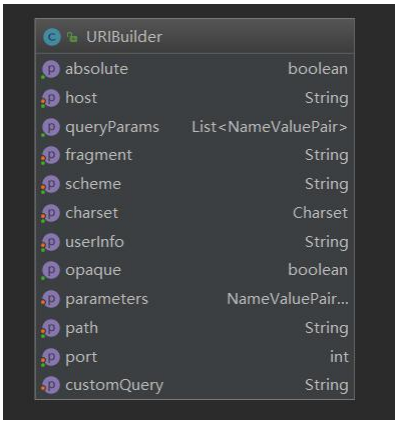
Builder：给出一个抽象接口，以规范产品对象的各个组成成分的建造。这个接口规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建。

ConcreteBuilder：实现 Builder 接口，针对不同的商业逻辑，具体化复杂对象的各部分的创建。 在建造过程完成后，提供产品的实例。

Director：调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建。

Product：要创建的复杂对象。

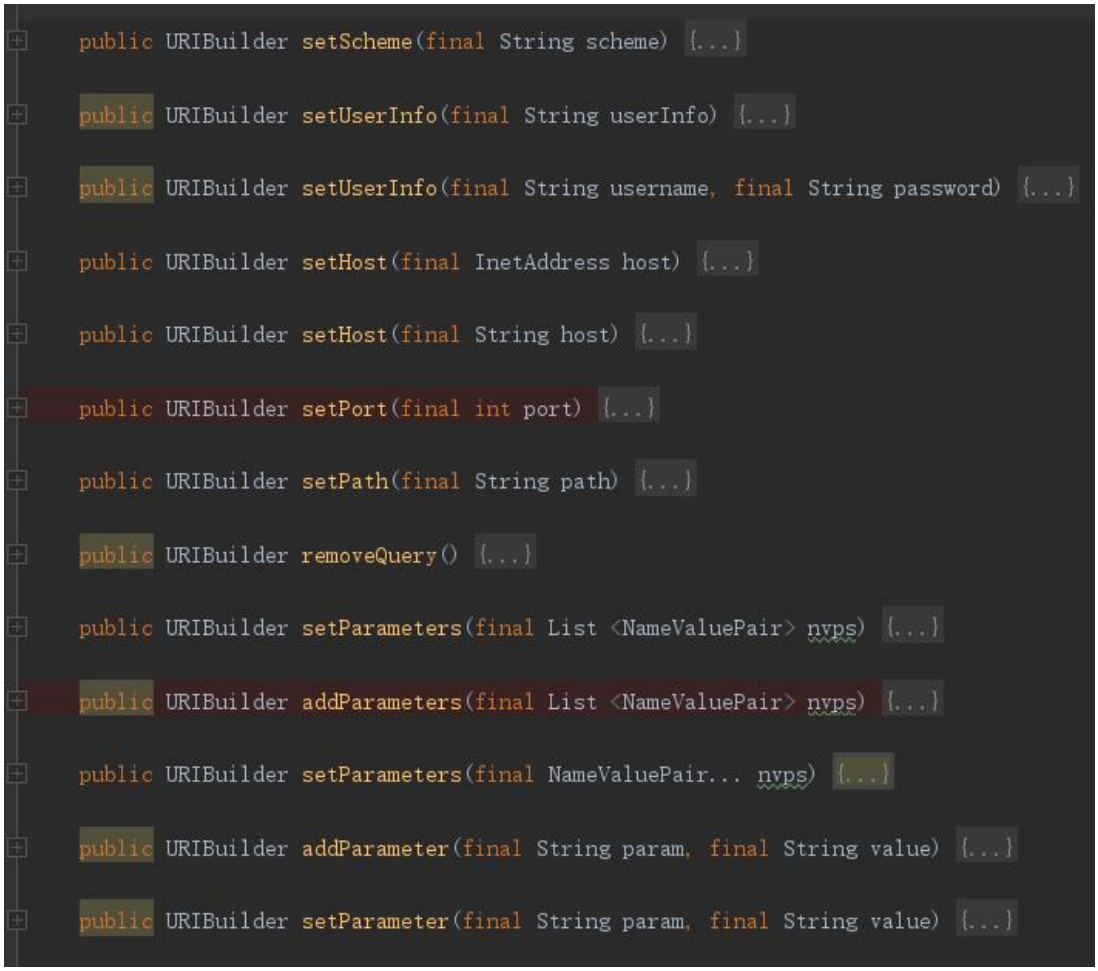
在 httpcore 中一些需要生成的对象具有复杂的内部结构，或者对象内部属性本身相互依赖，这时候建造者模式便派上了用场。下面我们以 URLBuilder 进行分析。



URLBuilder 为了建造一个统一资源标识符,创建了数十个类以实现目标。首先通过 private String buildString()来创建一个主机的字串，通过 private static String normalizePath(final String path, final boolean relative)处理空格、单引号和双引号，然后通过 private String encodeUserInfo(final String userInfo)、private String encodePath(final String path)、private String encodeUrlForm(final List<NameValuePair> params)、 private String encodeUric(final String fragment)对用户信息、路径、URL 格式来进行解码:

```
134      /**...*/
137      public URI build() throws URISyntaxException {...}
140
141  @   private String buildString() {...}
187
188      private static String normalizePath(final String path, final boolean relative) {...}
207
208  @   private void digestURI(final URI uri) {...}
223
224  @   private String encodeUserInfo(final String userInfo) {...}
227
228  @   private String encodePath(final String path) {...}
231
232  @   private String encodeUrlForm(final List<NameValuePair> params) {...}
235
236  @   private String encodeUric(final String fragment) {...}
239
```

接着通过 setScheme、 setUserInfo、setHost、setPort、setPath、setParameters 对计划、用户信息、主机、接口、路径、参数进行设置，对于参数设置又有 addParameter、clearParameters 两个类进行具体操作。



采取建造者模式，userInfo、path 等参数之间复杂的关系成了对于 URL 的创建，在《Effective Java》书中第二条，就提到“遇到多个构造器参数时要考虑用构建器”，其实这里的构建器就属于建造者模式。在 httpcore 中，建造者模式解决了面临着 URL 这样一个复杂对象的创建工作,URL 各个部分的子对象(如 useinfo、path、port 等）用一定的算法构成；由于需求的变化，URL 这个复杂对象的各个部分经常面临着剧烈的变化,但是将它们组合在一起的算法却相对稳定。

6.3 观察者模式

观察者模式定义了一个一对多的依赖关系，让一个或多个观察者对象监听一个主题对象。这样一来，当被观察者状态发生改变时，需要通知相应的观察者，使这些观察者对象能够自动更新。例如：GUI 中的事件处理机制采用的就是观察者模式。观察者模式通常包括以下四部分：

Subject（被观察的对象接口）：规定 ConcreteSubject 的统一接口 ； 每个 Subject 可以有多个 Observer；

ConcreteSubject（具体被观察对象）：维护对所有具体观察者的引用的列表 ； - 状态发生变化时会发送通知给所有注册的观察者。

Observer（观察者接口）：规定 ConcreteObserver 的统一接口；定义了一个 update()方法，在被观察对象状态改变时会被调用。

ConcreteObserver（具体观察者）：维护一个对 ConcreteSubject 的引用；特定状态与 ConcreteSubject 同步； 实现 Observer 接口，update()方法的作用：一旦检测到 Subject 有变动，就更新信息。

httpcpre 中对于 I/O 会话的观察者模式如下。安全传输层协议起始、流入流出、连接状态、输入输出就绪状态、超时和例外均做了相应观察。

```
public interface IOSessionListener {

    void tlsStarted(IOSession session);

    void tlsInbound(IOSession session);

    void tlsOutbound(IOSession session);

    void connected(IOSession session);

    void inputReady(IOSession session);
```

```
    void outputReady(IOSession session);

    void timeout(IOSession session);

    void exception(IOSession session, Exception ex);

    void disconnected(IOSession session);

}
```

观察者模式在被观察者和观察者之间建立一个抽象的耦合。被观察者角色所知道的只是一个具体观察者列表。由于被观察者和观察者没有紧密地耦合在一起，因此它们可以属于不同的抽象化层次。如果被观察者和观察者都被扔到一起，那么这个对象必然跨越抽象化和具体化层次。

6.4 策略模式

策略模式对一系列的算法加以封装，为所有算法定义一个抽象的算法接口，并通过继承该抽象算法接口对所有的算法加以封装和实现，具体的算法选择交由客户端决定（策略）。根据不同的需求，产生不同的策略或算法的接口.

ContentLengthStrategy 实现了对内容长度处理算法的平滑切换 。

httpcore 默认的策略：

```
public interface ContentLengthStrategy {
    long CHUNKED = -1;
    long UNDEFINED = -Long.MAX_VALUE;
    long determineLength(HttpMessage message) throws HttpException;

}
```

DefaultContentLengthStrategy 在对父类方法的重写中，首先对报 http 报文头进行译码得到 transferEncodingHeader，并对分块情况进行判断。接着对 countHeaders 长度是否大于 1 进行判断。然后对 contentLengthHeader 长度进行判断。不同情况下丢出不同的输出。

```
public class DefaultContentLengthStrategy implements ContentLengthStrategy {
    public static final DefaultContentLengthStrategy INSTANCE = new DefaultContentLengthStrategy();
    public DefaultContentLengthStrategy() {
    }
    @Override
    public long determineLength(final HttpMessage message) throws HttpException {
        Args.notNull(message, "HTTP message");
        final Header transferEncodingHeader = message.getFirstHeader(HttpHeaders.TRANSFER_ENCODING);
        if (transferEncodingHeader != null) {
            final String headerValue = transferEncodingHeader.getValue();
            if (HeaderElements.CHUNKED_ENCODING.equalsIgnoreCase(headerValue)) {
                return CHUNKED;
            }
            throw new NotImplementedException("Unsupported transfer encoding: " + headerValue);
        }
        if (message.countHeaders(HttpHeaders.CONTENT_LENGTH) > 1) {
            throw new ProtocolException("Multiple Content-Length headers");
      ......
```

httpcore 采取策略模式实现了算法可以自由切换，避免使用多重条件判断，同时又保持了良好的扩展性。策略模式缺点在于客户端必须知道所有的策略类，

并自行决定使用哪一个策略类，并且策略模式可能造成产生很多策略类，在 httpcore 中，策略较少，不会出现策略类膨胀。

6.5 其他模式

除了上文提到的工厂模式、建造者模式、观察者模式和策略模式四种模式外。httpcore 还采用了拦截器模式（Interceptor）、状态模式（Status）、装饰器模式（Decorator）、回调模式（Callback）、未来模式（Future）等设计模式。鉴于时间有限，暂时不做展开分析。

拦截器采用动态代理模式，每一个 action 具体类都是一个继承动态代理的类。所以这就不必为每一个类都加相应的逻辑，只需继承就行了。由于动态代理一般都比较难理解，httpcore 设计拦截器接口供开发者使用，开发者只要知道拦截器接口的方法、含义和作用即可，无须知道动态代理是怎么实现的。

状态模式封装了转换规则，枚举可能的状态，在枚举状态之前需要确定状态种类，所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。状态模式允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。

装饰器模式与继承关系的目的都是要扩展对象的功能，但是装饰器模式比继承更灵活。装饰器模式允许系统动态决定贴上一个需要的装饰，或者除掉一个不需要的装饰。通过使用不同的具体装饰器以及这些装饰类的排列组合，设计师可以创造出很多不同的行为组合、回调模式等设计模式。

这些设计模式从不同方面体现了开闭原则、里氏代换原则、依赖倒转原则、接口隔离原则、迪米特法则（最少知道原则）、合成复用原则。

7.结语

httpcore是作者学习的第一份大型开源工程，无论是代码量、项目功能，还是设计意图、设计模式，都值得深入钻研。后续会进一步深入分析，汲取营养，提升水平。

参考文献：

《httpcore教程》 南磊 译

《Java中23种设计模式》

《Effective Java》

https://www.cnblogs.com/malihe/p/6891920.html

https://www.wikipedia.org/