



链滴

Tomcat 源码解析 (1)-HTTPRequestResponse

作者: [xiantang](#)

原文链接: <https://ld246.com/article/1568725278860>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

<p></p>

HTTP

Web 服务器也称为超文本传输协议服务器，因为他使用HTTP 与其客户端进行通讯。

HTTP 允许Web服务器和浏览器通过Internet 发送请求，他是一种基于“请求 - 响应”的协议。客户端请求一个文件，服务端对于该请求进行响应。

HTTP 请求

一个HTTP 请求包括三部分：

- 请求方法 - - - 统一资源标识符 URI 协议 / 版本
- 请求头
- 实体

```
<code>POST /ajax/ShowCaptcha HTTP/1.1\r\nContent-Type: application/x-www-form-urlencoded\r\nHost: www.renren.com\r\nContent-Length: 36\r\n\r\nemail=%E5%B7%A5&password=asdasdsadas</code>
```

请求方法 - - URI - - 协议 / 版本

POST /ajax/ShowCaptcha HTTP/1.1\r\n

会出现在第一行

每个请求头之前都会用回车 / 换行符隔开 (CRLF)

并且请求头和请求实体之间会有一个空行，空行只有 CRLF 符号。CRLF告诉HTTP服务器请求的文从哪里开始。

HTTP 响应

与HTTP 请求相似，HTTP 响应也分三部分：

- 协议 - - 状态码
- 响应头
- 响应实体段

```
<code>HTTP/1.1 200 OK\r\nDate: Sat, 31 Dec 2005 23:59:59 GMT\r\nContent-Type: text/html; charset=ISO-8859-1\r\nContent-Length: 122\r\n\r\n<html>\r\n<head>\r\n<title>Wrox Homepage</title>
```

```
&lt;/head&gt;
|
&lt;body&gt;
&lt;!-- body goes here --&gt;
&lt;/body&gt;
&lt;/html&gt;
</code></pre>
<p>HTTP/1.1 200 OK Date: Sat, 31 Dec 2005 23:59:59 GMT Content-Type: text/html;charset ISO-8859-1 Content-Length: 122</p>
```

Wrox Homepage

```
<p>第一行类似使用的协议以及状态码（200 表示请求成功）</p>
<h2 id="toc_h2_3">StandardServer</h2>
<p>此类是Server 标准实现类， Server 仅此一个实现类。是Tomcat 顶级容器。Server是Tomcat中顶层的组件，它可以包含多个Service组件。这一节主要给大家讲解Tomcat 是如何关闭的。之后的章会给大家带来addService() 和findService (String) 方法的解析。</p>
<p>这个<code>StandardServer</code> 继承了 <code>Server</code></p>
<p>并且实现了其中比较关键的一个方法:</p>
<pre><code> /**
 * Wait until a proper shutdown command is received, then return.
 */
public void await();
</code></pre>
<p>z</p>
<pre><code>try {
InputStream stream;
try {
socket = serverSocket.accept();
socket.setSoTimeout(10 * 1000); // Ten seconds
stream = socket.getInputStream();
} catch (AccessControlException ace) {
log.warn("StandardServer.accept security exception: "
+ ace.getMessage(), ace);
continue;
} catch (IOException e) {
if (stopAwait) {
// Wait was aborted with socket.close()
break;
}
log.error("StandardServer.await: accept: ", e);
break;
}
while (expected &gt; 0) {
int ch = -1;
try {
ch = stream.read();
} catch (IOException e) {
log.warn("StandardServer.await: read: ", e);
ch = -1;
}
if (ch &lt; 32) // Control character or EOF terminates loop
break;
command.append((char) ch);
</code></pre>
```

```

expected--;
}finally {
// Close the socket now that we are done with it
try {
if (socket != null) {
socket.close();
}
} catch (IOException e) {
// Ignore
}
}
// Match against our command string
boolean match = command.toString().equals(shutdown);
if (match) {
log.info(sm.getString("standardServer.shutdownViaPort"));
break;
} else
log.warn("StandardServer.await: Invalid command "
+ command.toString() + " received");

```

</code></pre>

<p>根据源码上的注释 我们可以大致了解，在启动Tomcat 的时候，会开启一个8005的端口，这个服务负责监听到来的 telnet 连接，当受到为SHUTDOWN 的命令时候，销毁Tomcat 的所有服务并且关闭Tomcat。</p>

<h2 id="toc_h2_4">Request & Response</h2>

<p>在阅读Tomcat Request 源码的时候，我发现了一个比较有趣的东西:</p>

```

<pre><code>private MessageBytes schemeMB = MessageBytes.newInstance();
private MessageBytes methodMB = MessageBytes.newInstance();
private MessageBytes unparsedURIMB = MessageBytes.newInstance();
private MessageBytes uriMB = MessageBytes.newInstance();
private MessageBytes decodedUriMB = MessageBytes.newInstance();
private MessageBytes queryMB = MessageBytes.newInstance();
private MessageBytes protoMB = MessageBytes.newInstance();
// remote address/host
private MessageBytes remoteAddrMB = MessageBytes.newInstance();
private MessageBytes localNameMB = MessageBytes.newInstance();
private MessageBytes remoteHostMB = MessageBytes.newInstance();
private MessageBytes localAddrMB = MessageBytes.newInstance();
</code></pre>

```

<p>他的大多数成员变量都是<code>MessageBytes</code> 的实例，这让我产生了兴趣，这个<code>MessageBytes</code>到底是什么东西?</p>

<p>后来通过查阅资料发现Tomcat 为了提升性能，用了一些很有趣的 Tricks</p>

<p>Tomcat 对于读取来的字节流不会立马解析，而是将它进行打标 + 延时提取的方式来实现 按需使用。</p>

<p>下面我来跑一个小 demo 来了解一下MessageBytes 是个什么样的东西?</p>

```

<pre><code>public class MessageBytesTest {
public static void main(String[] args) {
MessageBytes mb = MessageBytes.newInstance();
// 等待测试的byte 对象
byte[] bytes = "abcdefg".getBytes(Charset.defaultCharset());
// 调用`setBytes`对bytes 进行标记
mb.setBytes(bytes, 2, 3);
System.out.println(mb.toString());
</code></pre>

```

```
    }
}
</code></pre>
<p>这个例子用来提取字节流中的子子节，并将它转换为String</p>
<p>下面我们继续来阅读这个 MessageBytes 到底是何方神圣?</p>
<p>MessageByte 主要有四种类型:</p>
<pre><code>public static final int T_NULL = 0;
/** getType() is T_STR if the object used to create the MessageBytes
was a String */
// 表示消息为字符串
public static final int T_STR &nbsp;= 1;
/** getType() is T_STR if the object used to create the MessageBytes
was a byte[] */
// 表示消息为字节数组类型
public static final int T_BYTES = 2;
/** getType() is T_STR if the object used to create the MessageBytes
was a char[] */
// 表示消息为字符数组
public static final int T_CHARS = 3;
</code></pre>
<ol>
<li>
<p><code>T_NULL</code>表示空消息，即消息为<code>null</code></p>
</li>
<li>
<p><code>T_STR</code>表示消息为字符串类型</p>
</li>
<li>
<p><code>T_BYTES</code>表示消息为字节数组类型</p>
</li>
<li>
<p><code>T_CHARS</code>表示消息为字符数组类型</p>
</li>
</ol>
<p>&nbsp;接着我们查看一下构造方法:</p>
<pre><code>/**
 * Creates a new, uninitialized MessageBytes object.
 * Use static newInstance() in order to allow
 * &nbsp; future hooks.
 */</code></pre>
// 使用工厂方法来创建实例
private MessageBytes() {
}
</code></pre>
<p>它的构造方法是私有的，我们只能通过工厂方法来获取实例</p>
<p>接着我们查看我们demo中使用的方法<code>setBytes</code> 这个是一个关键方法，它负责
bytes 打标。</p>
<pre><code> /**
 * Sets the content to the specified subarray of bytes.
 *
 * @param b the bytes
 * @param off the start offset of the bytes
*
```

```

* @param len the length of the bytes
*/
]
public void setBytes(byte[] b, int off, int len) {
//private final ByteChunk byteC=new ByteChunk();
//private final CharChunk charC=new CharChunk();
byteC.setBytes( b, off, len );
type=T_BYTES;
hasStrValue=false;
hasHashCode=false;
hasIntValue=false;
hasLongValue=false;
}
</code></pre>
<p>它内部调用了 <code>ByteChunk</code> 的<code>setBytes</code>方法,同时设置了<code>type</code>字段。</p>
<p>我们继续向里面走! </p>
<p>发现内部十分简单只是对数组进行了标识。 </p>
<pre><code> //非常简单, 就是设置一下待标识的字节数组、开始位置、结束位置。
public void setBytes(byte[] b, int off, int len) {
buff = b;
start = off;
end = start+ len;
isSet=true;
}
</code></pre>
<p>同时也印证了我们开头所说, 打标记但是没有转码。 </p>
<p>i</p>
<pre><code>// ----- MessageBytes -----
/** Compute the string value
 * 首先判断是否有缓存的字符串, 有的话就直接返回,
 * 这也是提高性能的一种方式。其次是根据type来选择不同的*Chunk,
 * 然后调用其toString()方法。那么我们这儿选择ByteChunk.toString()来分析。
*/
@Override
public String toString() {
// 先取缓存
if( hasStrValue ) {
return strValue;
}
// 判断缓存类型
// 设置缓存
switch (type) {
case T_CHARS:
strValue=charC.toString();
hasStrValue=true;
return strValue;
case T_BYTES:
strValue=byteC.toString();
hasStrValue=true;
return strValue;
}
return null;
}

```

```

// ----- ByteChunk -----
@Override
public String toString() {
if (null == buff) {
return null;
} else if (end-start == 0) {
return "";
}
return StringCache.toString(this);
}
}

public String toStringInternal() {
if (charset == null) {
charset = DEFAULT_CHARSET;
}
// 如果我们只有少部分要使用
// 通过打标记 + 延时提取的方式
// new String(byte[], int, int, Charset) takes a defensive copy of the
// entire byte array. This is expensive if only a small subset of the
// bytes will be used. The code below is from Apache Harmony.
CharBuffer cb;
cb = charset.decode(ByteBuffer.wrap(buff, start, end-start));
// return new String(buff, start, end - start, charset);
return new String(cb.array(), cb.arrayOffset(), cb.length());
}
</code></pre>
<p>需要关注的主要是这三个方法</p>
<p>MB 调用 toString 方法的时候首先会从当前实例中取出缓存，如果没有缓存就调用 ByteChunk toString 方法，设置缓存并且返回。</p>
<p>ByteChunk 的 toString 方法是使用 StringCache 的 toString 方法 但是其中的主要调用仍然是 StringCache.toStringInternal()</p>
<p>我们来讲解一下这个方法吧！</p>
<p>他使用的是NIO 的 ByteBuffer 根据<code>偏移量</code>和<code>待提取长度</code>进<code>编码提取转换</code>。</p>
<p>需要注意的是该注释已经给出了使用<code>java.nio.charset.CharSet.decode()</code>代替接使用<code>new String(byte[], int, int, Charset)</code>的原因。</p>
<p>如果是用默认的 <code>new String(byte[], int, int, Charset)</code> 会对整个byte 进行拷，对于一个巨大的byte[] 中我们只需要提取一些些数据，就会带来严重的性能损耗。</p>
<h3 id="toc_h3_5">Request 是如何被解析的</h3>
<p>他是如何判断打标的位置的？</p>
<p>下面为以给请求行中的 URI 打标为大家解释</p>
<p>我们要探寻的是:</p>
<pre><code>/**
 * Implementation of InputBuffer which provides HTTP request header parsing as
 * well as transfer decoding.
 *
 * @author <a href="mailto:remm@apache.org">Remy Maucherat</a>;
 * @author Filip Hanik
 */
public class InternalNioInputBuffer extends AbstractInputBuffer<NioChannel> {
@Override
public boolean parseRequestLine(boolean useAvailableDataOnly)
throws IOException {
//----省略前面的解析步骤
}
}
</code></pre>

```

```
if (parsingRequestLinePhase == 4) {
// Mark the current buffer position

int end = 0;
//
// Reading the URI
//
boolean space = false;
while (!space) {
// Read new bytes if needed
if (pos >= lastValid) {
if (!fill(true, false)) //request line parsing
return false;
}
if (buf[pos] == Constants.SP || buf[pos] == Constants.HT) {
space = true;
end = pos;
} else if ((buf[pos] == Constants.CR)
|| (buf[pos] == Constants.LF)) {
// HTTP/0.9 style request
parsingRequestLineEol = true;
space = true;
end = pos;
} else if ((buf[pos] == Constants.QUESTION)
&& (parsingRequestLineQPos == -1)) {
parsingRequestLineQPos = pos;
}
pos++;
}
request.unparsedURI().setBytes(buf, parsingRequestLineStart, end - parsingRequestLineStart);
if (parsingRequestLineQPos >= 0) {
request.queryString().setBytes(buf, parsingRequestLineQPos + 1,
end - parsingRequestLineQPos - 1);
request.requestURI().setBytes(buf, parsingRequestLineStart, parsingRequestLineQPos - parsin
RequestLineStart);
} else {
// URL 当解析的时候之前个请求方法执行完之后会找到对应的空格
// 请求行的开始就就是parseRequestLineStart 开始位置
```

```
// 之后向下寻找空格 并将他标记为end
// setBytes 的时候只要把开始的位置和长度设置进去就行了
request.requestURI().setBytes(buf, parsingRequestLineStart, end - parsingRequestLineStart);
}
System.out.println("解析出来的URI为: " + request.requestURI().toString());
parsingRequestLinePhase = 5;
}
}
}

</code></pre>
```

<p>这里主要要了解的是几个变量</p>

```
<ul>
<li>
<p>buf 整条请求头的byte[]</p>
</li>
<li>
<p>parsingRequestLineStart URI 开始位置</p>
</li>
<li>
<p>end URI 结束位置</p>
</li>
</ul>
```

<p>上面代码的大致意识是 将parsingRequestLineStart的位置设置为上次解析（解析请求方法）的位置 + 1</p>

<p>然后通过遍历buf 寻找从 parsingRequestLineStart 开始的第一个空格。</p>
<p>并且为了避免多余的编码，tomcat 将 <code>空格</code> <code>CR</code> <code>LF</code> 也转换为字节，只要比较字节就能判断是否相同，期间没有任何编码。</p>

```
<pre><code>/**
 * CR.
 */
public static final byte CR = (byte) '\r';
/**
 * LF.
 */
public static final byte LF = (byte) '\n';
/**
 * SP.
 */
public static final byte SP = (byte) ' ';</code></pre>
```

<p>将这些字节流通过setBytes 打标，记住是offset/offset+长度。</p>

<h3 id="toc_h3_6">总结</h3>

<p>还是开头那句话:</p>

<p>Tomcat 采用延时编码的方式来提升性能，解析完一个Request后，如果没有被利用，变量存储只是这个字节流的打标，只有在使用的时候才会去编码或者去取缓存。这样有个好处，就是Request的信息不是全部要使用的，有时候我们只需要取一部分就行了，所以就可以降低编码的性能消耗。</p>

<p>参考:</p>

<p><a href="https://ld246.com/forward?goto=https%3A%2F%2Fwww.jianshu.com%2Fp%2F

b27c8da1543" target="_blank" rel="nofollow ugc">>深入理解Tomcat (12) 拾遗-MessageByte
</p>

消息字节—MessageBytes</p>