HowTomcatWorks问题整理

# 问题1 Tomcat不支持重复访问

|  |  |  |  |
| --- | --- | --- | --- |
| 标题 | Tomcat不支持重复访问 | | |
| 提交时间 | 20190915 | 章节 | Chapter4 Tomcat Default Connector |
| **问题描述**  chap4/BootStrap.java启动后，httpclient第一次调用没问题，但是第二次调用报错：  org.apache.http.NoHttpResponseException: 127.0.0.1:8080 failed to respond  应该是服务端没有响应  用浏览器访问也是一样，也是第一次调用没问题，但是第二次调用没啥反应  我们观察一下服务端的日志 | | | |
| **异常日志**  HttpClient异常日志：  信息: I/O exception (org.apache.http.NoHttpResponseException) caught when processing request to {}->http://127.0.0.1:8080: The target server failed to respond  九月 21, 2019 6:32:49 上午 org.apache.http.impl.execchain.RetryExec execute  信息: Retrying request to {}->http://127.0.0.1:8080  org.apache.http.NoHttpResponseException: 127.0.0.1:8080 failed to respond  at org.apache.http.impl.conn.DefaultHttpResponseParser.parseHead(DefaultHttpResponseParser.java:141)  at org.apache.http.impl.conn.DefaultHttpResponseParser.parseHead(DefaultHttpResponseParser.java:56)  at org.apache.http.impl.io.AbstractMessageParser.parse(AbstractMessageParser.java:259)  at org.apache.http.impl.DefaultBHttpClientConnection.receiveResponseHeader(DefaultBHttpClientConnection.java:163)  at org.apache.http.impl.conn.CPoolProxy.receiveResponseHeader(CPoolProxy.java:157)  at org.apache.http.protocol.HttpRequestExecutor.doReceiveResponse(HttpRequestExecutor.java:273)  at org.apache.http.protocol.HttpRequestExecutor.execute(HttpRequestExecutor.java:125)  at org.apache.http.impl.execchain.MainClientExec.execute(MainClientExec.java:272)  at org.apache.http.impl.execchain.ProtocolExec.execute(ProtocolExec.java:186)  at org.apache.http.impl.execchain.RetryExec.execute(RetryExec.java:89)  at org.apache.http.impl.execchain.RedirectExec.execute(RedirectExec.java:110)  at org.apache.http.impl.client.InternalHttpClient.doExecute(InternalHttpClient.java:185)  at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:83)  at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:108)  at com.nbcb.mytomcat.chap4.client.HttpClientTest.main(HttpClientTest.java:59) | | | |
| **问题分析**  所谓的” NoHttpResponseException”，其实就是服务端没有正常返回，再看一下HttpClient异常堆栈：  org.apache.http.impl.conn.DefaultHttpResponseParser.parseHead(DefaultHttpResponseParser.java:141)  可以看到是解析Http Header的时候报错。那么，我们怀疑可能是服务端没有返回合适的Http header。观察一下服务端返回http header的代码：  HttpResponseBase.sendHeaders()  通过Debug发现，这个sendHeader()方法中：  if (isCommitted())  return;  这个步骤就直接返回了。后续没有继续发送Header。那么，我们之前是在哪里commit的呢？要先找找到哪个方法会把commit设置为true，我们看到是这个方法：  ResponseBase.flushBuffer()  那么，看哪些地方会调用flushBuffer()方法呢：  ResponseBase.write()  ResponseStream.close();  ResponseStream.flush();  都打上断点。  我们看到，第一次正常访问，以下代码会被调用到，用这两个地方的代码来commit  ResponseStream.close();  ResponseBase.write()  可是奇怪的是，第二次访问的时候，这三处代码都没有定位到，那么到底是在哪里commit的呢？之前肯定有地方出现异常，导致Response提前commit  为了找出在哪里commit，我们只能一个个地方去试  发现第二次异常调用的时候，SimpleContainer.invoke()之前，就已经commit了，而且  ResponseBase.flushBuffer()居然没有被调用到。  因此我们怀疑Response实例创建有问题。  我知道了，是我代码写的有问题！  因为我这边Response实例是在HttpProcessor的contstructor中创建的。  HttpProcessor线程被从线程池取出来之后，使用完Response实例之后就commit了。  后续这个HttpProcessor线程被从线程池再次取出来之后，显然commit状态是异常的。 | | | |
| **问题原因总结**  通过问题分析，我们也看到了，原因是我们之前没有对Response对象进行回收。导致Response的参数(commited为true)，导致后续的客户端请求，服务端都没有响应。 | | | |
| **解决方案**  既然知道了问题原因，也就好解决了。  HttpProcessor线程使用完Request/Response对象之后，  也就是HttpProcessor.process()在执行完servlet之后，需要对Request/Response对象进行“回收”：  request.recycle();  response.recycle();  所谓的“回收”，其实就是将Request/Response对象中的变量初始化一下(比如committed 设置为 false) | | | |
|  | | | |

# 问题2 Httpclient并发访问Tomcat失败-Permission denied

|  |  |  |  |
| --- | --- | --- | --- |
| 标题 | Httpclient并发访问Tomcat失败的问题- Permission denied | | |
| 提交时间 | 20190922 | 章节 | Chapter4 Tomcat Default Connector |
| **问题描述**  我们通过Httpclient访问Tomcat，在并发数调高的时候，会报错。  Httpclient并发线程数提高到50/每个线程循环调用100次的时候会报错。  之前10并发/20并发/30并发/40并发都问题。  只有到50并发的时候，部分线程没法完成任务，导致报错。 | | | |
| **异常日志**  信息: I/O exception (java.net.SocketException) caught when processing request to {}->http://127.0.0.1:8080: Permission denied: connect  九月 22, 2019 11:33:49 上午 org.apache.http.impl.execchain.RetryExec execute  信息: Retrying request to {}-><http://127.0.0.1:8080>  java.net.SocketException: Permission denied: connect  at java.net.DualStackPlainSocketImpl.connect0(Native Method)  at java.net.DualStackPlainSocketImpl.socketConnect(DualStackPlainSocketImpl.java:79)  at java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:350)  at java.net.AbstractPlainSocketImpl.connectToAddress(AbstractPlainSocketImpl.java:206)  at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:188)  at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:172)  at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392)  at java.net.Socket.connect(Socket.java:589)  at org.apache.http.conn.socket.PlainConnectionSocketFactory.connectSocket(PlainConnectionSocketFactory.java:75)  at org.apache.http.impl.conn.DefaultHttpClientConnectionOperator.connect(DefaultHttpClientConnectionOperator.java:142)  at org.apache.http.impl.conn.PoolingHttpClientConnectionManager.connect(PoolingHttpClientConnectionManager.java:374)  at org.apache.http.impl.execchain.MainClientExec.establishRoute(MainClientExec.java:393)  at org.apache.http.impl.execchain.MainClientExec.execute(MainClientExec.java:236)  at org.apache.http.impl.execchain.ProtocolExec.execute(ProtocolExec.java:186)  at org.apache.http.impl.execchain.RetryExec.execute(RetryExec.java:89)  at org.apache.http.impl.execchain.RedirectExec.execute(RedirectExec.java:110)  at org.apache.http.impl.client.InternalHttpClient.doExecute(InternalHttpClient.java:185)  at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:83)  at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:108)  at com.nbcb.mytomcat.chap4.client.HttpClientTest.startAccessTomcat(HttpClientTest.java:60)  at com.nbcb.mytomcat.chap4.client.HttpClientThread.run(HttpClientThread.java:28)  at com.nbcb.mytomcat.chap4.client.StartBatch.main(StartBatch.java:25) | | | |
| **问题分析**  为啥并发小的时候没问题，并发慢慢上去了会有问题呢？  1.一方面我们当然可以调整Server端的并发数试试。目前线程池是5-20。在这样的线程池配置下，目前只能支持40左右的并发。这个经过验证，是可行的。  2.另一方面，我觉得服务端代码应该也应该优化一下，如果线程池资源不够，就有一个等待超时设置，而不是马上关闭socket连接，导致客户端报错。这个等待后续优化，或者看看Tomcat是怎么处理的。我们看了一下How Tomcat Works官方代码，发现ServerSocket accept到客户端socket之后，设置了socket属性：  socket.setSoTimeout(Constants.DEFAULT\_CONNECTION\_TIMEOUT); // socket 连接超时  socket.setTcpNoDelay(true); // 是否关闭Naggle算法  设置了之后，貌似还是会报错。  3.我们看看线程池的资源分配是否合理。我觉得应该不是很合理。可以打印一下，各个线程被调用的次数。这个我们看一下线程池的实现机制就能知道，因为采用stack的方式保存线程资源，放在stack顶端的线程资源显然被调用的次数会远远大于底部的线程。后续我们打印一下线程池的使用情况。通过打印线程池的数量  4.可以借鉴http1.1 persistent connection的机制，减少socket资源的消耗，提升单次服务的处理效率。当然，现在的程序也要做出调整，服务端要支持persistent connection，客户端也要通过调整request http header支持persistent connection，建议客户端一个线程就占用一个socket资源。  5.调整服务端，提升服务端单个线程处理客户端请求的效率。比如减少SystemOut打印的次数。实际测试下来，效果是有的：服务端线程池是5-20，客户端50并发循环100次，只是偶尔报错了。后续我们继续提升客户端并发数，提升到80，就报错了。当然，提升服务端的效率是没低的，需要不断努力。  6.优化服务端线程池代码。从实际测试情况来看，服务端线程池代码确实有问题：HttpConnector.run()每次接收到客户端socket请求，分配一个HttpProcessor线程资源进行处理之后，都打印一下当前线程池当前分配数。我们发现，这个当前线程连接数一直维持在初始数量上，这个要么是我们打印的当前线程池分配数有问题，要么就是我们线程池有问题。我们换一种打印方式试试，直接打印线程池stack大小试试，也是一样stack.size()一直维持在初始水平。到底哪里有问题呢？一种情况可能是HttpClient进行了优化，对多个Http请求进行了处理，复用同一个socket请求。为了验证我们的猜想，netstat –an|grep 8080看看socket请求有多少。  7.直接百度这个报错信息  “java.net.SocketException: Permission denied: connect”  网上给出的方案是设置JAVA启动参数： -Djava.net.preferIPv4Stack=true  实践证明是有效的！  服务端并发数： 5-20  HttpClient并发80，循环100次的条件下，已经不会报错了！ | | | |
| **解决方案**  **方案1** 扩大服务端线程池数量  HttpConnector.maxProcessors  这当然是可以做到的。但是服务端线程数也不能无限扩展。因为显然线程数配置的越多，占用内存资源也越大。我们用实际测试也能验证这点：  并发数： 5-20  初始化内存占用： 14000  高峰期内存占用\*：16000  并发数： 5-30  初始化内存占用： 14000  高峰期内存占用：23000  从这个数据来看，30并发下的内存占用确实要比20并发的内存多50%。单个线程占用内存大小大概在800KB左右。  \*并发数为5-20的时候，客户端部分线程报错。原因也很明白，就是服务端的线程池已经占满了，没发提供更多的线程资源来处理客户端请求了。  **方案2** 设置JAVA启动参数： -Djava.net.preferIPv4Stack=true | | | |
| **问题原因**  经过上面的详细分析，其实这个报错是由于JDK的一个BUG引起的。至于线程池大小为啥一直不上去，后续我们对代码有改进，是我们客户端程序StartBatch有问题，已经优化了。 | | | |
| **Thinking**  1.线程池管理  从这个问题我们可以发散一下，我们知道线程资源是比较耗费内存的，那么是否需要对空闲内存进行回收？这个可以参考dbcp方案，采用定期回收的机制，或者简单点用完直接回收。这个线程池的管理，建议参考dbcp，通过commons-pool组件进行专业管理。  2.性能对比  有空还可以拿我们这个版本的TomcatWin和Tomcat4进行比对，看并发性能相差多少。 | | | |

# 问题3 Httpclient并发访问Tomcat失败- Connection reset by peer

|  |  |  |  |
| --- | --- | --- | --- |
| 标题 | Httpclient并发访问Tomcat失败的问题- Connection reset by peer | | |
| 提交时间 | 20190922 | 章节 | Chapter4 Tomcat Default Connector |
| **问题描述**  我们调整HttpClient并发程序：StartBatch之后，发现线程池的数量已经会根据实际需要上升了。但是随之而来的客户端HttpClient请求还是会报错。  测试条件：  服务端线程：5-20  客户端：50并发，循环100次 | | | |
| **异常日志**  信息: I/O exception (java.net.SocketException) caught when processing request to {}->http://127.0.0.1:8080: Connection reset by peer: socket write error  下午 org.apache.http.impl.execchain.RetryExec execute  信息: Retrying request to {}->http://127.0.0.1:8080  org.apache.http.conn.HttpHostConnectException: Connect to 127.0.0.1:8080 [/127.0.0.1] failed: Connection refused: connect  at org.apache.http.impl.conn.DefaultHttpClientConnectionOperator.connect(DefaultHttpClientConnectionOperator.java:156)  at org.apache.http.impl.conn.PoolingHttpClientConnectionManager.connect(PoolingHttpClientConnectionManager.java:374)  at org.apache.http.impl.execchain.MainClientExec.establishRoute(MainClientExec.java:393)  at org.apache.http.impl.execchain.MainClientExec.execute(MainClientExec.java:236)  at org.apache.http.impl.execchain.ProtocolExec.execute(ProtocolExec.java:186)  at org.apache.http.impl.execchain.RetryExec.execute(RetryExec.java:89)  at org.apache.http.impl.execchain.RedirectExec.execute(RedirectExec.java:110)  at org.apache.http.impl.client.InternalHttpClient.doExecute(InternalHttpClient.java:185)  at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:83)  at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:108)  at com.nbcb.mytomcat.chap4.client.HttpClientTest.startAccessTomcat(HttpClientTest.java:60)  at com.nbcb.mytomcat.chap4.client.HttpClientThread.run(HttpClientThread.java:31)  at java.lang.Thread.run(Thread.java:745)  Caused by: java.net.ConnectException: Connection refused: connect  at java.net.TwoStacksPlainSocketImpl.socketConnect(Native Method)  at java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:350)  at java.net.AbstractPlainSocketImpl.connectToAddress(AbstractPlainSocketImpl.java:206)  at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:188)  at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:172)  at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392)  at java.net.Socket.connect(Socket.java:589)  at org.apache.http.conn.socket.PlainConnectionSocketFactory.connectSocket(PlainConnectionSocketFactory.java:75)  at org.apache.http.impl.conn.DefaultHttpClientConnectionOperator.connect(DefaultHttpClientConnectionOperator.java:142)  ... 12 more | | | |
| **问题分析**  我们查看服务端日志：  其中” current HttpProccessor no”是当前有多少HttpProcessor线程对象被创建出来，” current Proccessor stack no”是当前线程池大小(也就是当前可用的线程资源)，一般来说，currProccessors肯定是大于等于processors.size()的，多出来的数量其实就是目前正在处理客户端socket servlet请求的。  从以下服务端可以清晰看到，” current Proccessor stack no”为0，说明线程池已经空了；而current HttpProccessor no为20(达到了线程池最大值)，没法创建更多的线程了。  current Proccessor stack no: 0  current HttpProccessor no: 20  从线程池获取可用的HttpProcessor失败！  这段服务端的日志清晰告诉我们，服务端线程池已经到达极限，没法提供更多的线程资源来处理客户端请求了。  怎么处理呢？  方案1 当然是扩大服务端的线程资源，  方案2 优化客户端代码  优化方式为，之前每个httpclient线程(HttpClientTest)，都会new一个httpclient对象，这也就意味着每调用一次服务端，都会new 一个httpclient对象。后续优化为，HttpClientThread线程只new 一个httpclient对象，HttpClientThread循环调用HttpClientTest的时候，只用一个httpclient对象。这样就减少了反复new httpclient对象的耗时。实际测试效果来看，在50并发下，平均耗时从200ms+减少到80ms的水平。这相当于提升了客户端调用的效率。但是报错还是依然会有。  另外，我们可以看一下httpclient返回的结果，有可能虽然报错了，但通过重连机制，结果还是成功的。当然，解析httpclient返回结果是需要损失性能的，50并发下，httpclient平均耗时从300ms上升到400ms。  方案3 httpclient组件自带的重传机制  我们在这里大言不惭的说“加入重传机制”，说得好像是自己优化似的，其实重传机制是httpclient组件自带的(想得很是周到)。  我们看报错信息，” Connection refused: connect”，客户端连接被拒绝了。那么我们知道，httpclient是有重连机制的，那么httpclient是否能够通过重连机制，保证客户端的访问还是能够成功呢？我们要改造一下httpclienttest，加入重连机制，然后校验客户端返回结果，看能否通过重连机制，使得httpclienttest就算报错，也还是能够从服务端取到数据。通过研究Httpclient的重传机制，我们发现，httpclient本身就是有重传机制的：默认3次重传。观察日志：  com.nbcb.mytomcat.chap4.client.HttpClientTest  九月 25, 2019 10:13:13 下午 org.apache.http.impl.execchain.RetryExec execute  // 第1次访问失败  caught when processing request to {}->http://127.0.0.1:8080: Software caused connection abort: recv failed  九月 25, 2019 10:13:13 下午 org.apache.http.impl.execchain.RetryExec execute  信息: Retrying request to {}->http://127.0.0.1:8080  九月 25, 2019 10:13:13 下午 org.apache.http.impl.execchain.RetryExec execute  信息: I/O exception (org.apache.http.NoHttpResponseException)  // 第2次访问失败  caught when processing request to {}->http://127.0.0.1:8080: The target server failed to respond  九月 25, 2019 10:13:13 下午 org.apache.http.impl.execchain.RetryExec execute  信息: Retrying request to {}->http://127.0.0.1:8080  九月 25, 2019 10:13:13 下午 org.apache.http.impl.execchain.RetryExec execute  信息: I/O exception (org.apache.http.NoHttpResponseException)  // 第3次访问失败  caught when processing request to {}->http://127.0.0.1:8080: The target server failed to respond  九月 25, 2019 10:13:13 下午 org.apache.http.impl.execchain.RetryExec execute  信息: Retrying request to {}->http://127.0.0.1:8080  org.apache.http.NoHttpResponseException: 127.0.0.1:8080 failed to respond  at org.apache.http.impl.conn.DefaultHttpResponseParser.parseHead(DefaultHttpResponseParser.java:141)  at org.apache.http.impl.conn.DefaultHttpResponseParser.parseHead(DefaultHttpResponseParser.java:56)  at org.apache.http.impl.io.AbstractMessageParser.parse(AbstractMessageParser.java:259)  at org.apache.http.impl.DefaultBHttpClientConnection.receiveResponseHeader(DefaultBHttpClientConnection.java:163)  at org.apache.http.impl.conn.CPoolProxy.receiveResponseHeader(CPoolProxy.java:157)  at org.apache.http.protocol.HttpRequestExecutor.doReceiveResponse(HttpRequestExecutor.java:273)  at org.apache.http.protocol.HttpRequestExecutor.execute(HttpRequestExecutor.java:125)  at org.apache.http.impl.execchain.MainClientExec.execute(MainClientExec.java:272)  at org.apache.http.impl.execchain.ProtocolExec.execute(ProtocolExec.java:186)  at org.apache.http.impl.execchain.RetryExec.execute(RetryExec.java:89)  at org.apache.http.impl.execchain.RedirectExec.execute(RedirectExec.java:110)  at org.apache.http.impl.client.InternalHttpClient.doExecute(InternalHttpClient.java:185)  at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:83)  at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:108)  at com.nbcb.mytomcat.chap4.client.HttpClientTest.startAccessTomcat(HttpClientTest.java:82)  at com.nbcb.mytomcat.chap4.client.HttpClientTest.startAccessTomcat(HttpClientTest.java:27)  at com.nbcb.mytomcat.chap4.client.HttpClientTest.main(HttpClientTest.java:114)  // finally 最终失败  Httpclient response return error===============>  从日志来看，如果3次重传都失败，就会有3次” caught when processing request”异常日志提示失败。  那么我们看看并发条件下，重传失败的日志( caught when processing request)条数是否刚好为最终失败日志(Httpclient response return error)条数的三倍。  服务端启动后，第1次调用httpclient并发程序：StartBatch：  最终失败日志(Httpclient response return error)条数：0  重传失败的日志( caught when processing request)条数 142  第2次调用httpclient并发程序：StartBatch：  最终失败日志(Httpclient response return error)条数：22  重传失败的日志( caught when processing request)条数：61  第3次调用httpclient并发程序：StartBatch：  最终失败日志(Httpclient response return error)条数：1  重传失败的日志( caught when processing request)条数：50  第4次调用httpclient并发程序：StartBatch：  最终失败日志(Httpclient response return error)条数：9  重传失败的日志( caught when processing request)条数：138  重启一下BootStrap:  第1次调用httpclient并发程序：StartBatch：  最终失败日志(Httpclient response return error)条数：37  重传失败的日志( caught when processing request)条数：77  第2次调用httpclient并发程序：StartBatch：  最终失败日志(Httpclient response return error)条数：7  重传失败的日志( caught when processing request)条数：82  这些重传次数，和最终客户端调用失败次数的数据，说明什么问题呢？说明通过httpclient的自动重传3次，是能够解决一部分问题的：就是有些调用成功了。  但是我们看到，还有很多最终失败的日志。最终失败的原因可能是重传3次也没发成功。怎么办呢？除了httpclient自动的3次重传，我们自己发现最终失败后，能否再试试重传呢？我猜想应该能够进一步减少最终失败的次数，但是当然会增加客户端平均访问时间。  方案4 应用层加入重传机制  我们知道，在方案3中，会出现httpclient重传3次还是会失败的情况。那么，我们在应用层，是否能够再加入重传机制呢？  应用成重传机制实现后，效果还是非常好的。  服务端线程池在5-20，客户端50并发，每个并发循环100次。  最终全部访问成功，有两大量的httpclient内部的重传(66次)，有少量应用层重传(8次)。服务平均耗时也比较少，维持在30-90ms之间。  当然，如果并发数不断上升，比如60并发，每个并发循环500次，应就会有经过3次重传还是失败的记录(一般只有一条这样的记录)  至此，httpclient并发访问Tomcat的调优算是圆满完成了。 | | | |
| **解决方案** | | | |
| **总结**  这个问题要重点定位，和我们生产环境AIBC connect reset非常像！生产环境无论是服务端(DMS socket server)，还是客户端(AIBC socket client)，连接方式都差不多。 | | | |

# 问题4 Tomcat加载servlet class问题

|  |  |  |  |
| --- | --- | --- | --- |
| 标题 | Tomcat加载servlet class问题 | | |
| 提交时间 | 20200801 | 章节 | Chapter8 Loader |
| **问题描述**  这个问题怎么说呢，和Tomcat 的Loader机制有关。之前的章节，没有深入了解过Loader的机制，只是通过SimpleLoader，简单把servlet class加载进来。  这次我们深入探讨Loader的实现。在实际调试过程中，发现我们的Tomcat调用的servlet class，并不是我们WEB-INF/classes指定的class，而是我们工程编译出来的class：  file:/Users/zhoushuo/Documents/workspace/TomcatWin/out/production/TomcatWin/  这个显然是和我们的预期不相符合，我们预期要加载的class要从这个目录去找：  file:/Users/zhoushuo/Documents/workspace/TomcatWin/WEB-INF/classes/ | | | |
| **异常日志**  这个也没啥异常日志 | | | |
| **问题分析**  我们调试了很久，无论我们怎么改，Loader都是去out/production目录下取加载servlet类。后来我们偶尔尝试把out/production下的文件删除，发现终于加载WEB-INF下的class类了。结合第8章开篇的java loader原理，我们才醒悟，这个和Java Loader顺序有关，一定是Java Loader优先检索out/production目录下的servlet class类，才会出现这样的问题。 | | | |
| **解决方案**  为了解决这个问题，我们只要保证out/production目录下的class不存在就行了：  mv /Users/zhoushuo/Documents/workspace/TomcatWin/out/production/TomcatWin/ModernServlet.class /Users/zhoushuo/Documents/workspace/TomcatWin/WEB-INF/classes | | | |
| **总结**  这个问题，折射出java load的机制，具体参考《How Tomcat works》第8章开篇的”Java Class Loader”，介绍得非常清楚，加载顺序为：   1. Bootstrap 2. Extension 3. System Class loader   我们具体说明一下：   1. Bootstrp   所谓的bootstrap，其实就是我们启动JVM运行Java程序要用到的核心类，具体代码可以参考我们本地JDK1.8环境的Java源码：  /Library/Java/JavaVirtualMachines/jdk1.8.0\_71.jdk/Contents/Home/src/java   1. Extension   所谓的extension，就是一些扩展的类，一般是Java实现厂商扩展的一些方便开发的类，参考本地的扩展类：  /Library/Java/JavaVirtualMachines/jdk1.8.0\_71.jdk/Contents/Home/src/javax   1. System Class loader   就是我们当前Java程序指定的classpath  比如对于IDEA工程来说，就是out/production目录  对于MyEclipse来书，就是bin目录  还有就是我们手工指定的classpath，比如  export CLASSPATH=.:$JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar   1. User specified class   最后才去用户指定的loader目录找要执行的class类  比如我们tongguo URLClassLoader初始化loader类的时候，指定的加载路径：  file:/Users/zhoushuo/Documents/workspace/TomcatWin/WEB-INF/classes/  以上的加载顺序是从1到4。  同时参考我们在工作中遇到的http client jar包加载的问题。  其实说白了也是java class load顺序的问题。 | | | |

# 问题5 Null Pointer

|  |  |  |  |
| --- | --- | --- | --- |
| 标题 | 空指针异常 | | |
| 提交时间 | 20200801 | 章节 | Chapter Session |
| 问题描述 我们调试servlet的时候，发现第一次访问servlet没问题，但是第二次开始就会报错： | | | |
| 异常日志 java.lang.NullPointerException  at MySessionServlet.doGet(MySessionServlet.java:58)  at javax.servlet.http.HttpServlet.service(HttpServlet.java:740)  at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)  at com.nbcb.mytomcat.catalina.core.SimpleWrapperValve.invoke(SimpleWrapperValve.java:77)  at com.nbcb.mytomcat.catalina.core.SimplePipeline$SimplePipelineValveContext.invokeNext(SimplePipeline.java:137)  at com.nbcb.mytomcat.catalina.core.SimplePipeline.invoke(SimplePipeline.java:78)  at com.nbcb.mytomcat.catalina.core.SimpleWrapper.invoke(SimpleWrapper.java:201)  at com.nbcb.mytomcat.catalina.core.SimpleContextValve.invoke(SimpleContextValve.java:69)  at com.nbcb.mytomcat.catalina.core.SimplePipeline$SimplePipelineValveContext.invokeNext(SimplePipeline.java:137)  at com.nbcb.mytomcat.catalina.core.SimplePipeline.invoke(SimplePipeline.java:78)  at com.nbcb.mytomcat.catalina.core.StandardContext.invoke(StandardContext.java:77)  at com.nbcb.mytomcat.catalina.connector.HttpProcessor.process(HttpProcessor.java:292)  at com.nbcb.mytomcat.catalina.connector.HttpProcessor.run(HttpProcessor.java:191)  at java.lang.Thread.run(Thread.java:745) | | | |
| **问题分析**  问题原因是Servlet(MySessionServlet)中，通过Request(HttpRequestBase)获取session对象的时候，Request会需要通过Context获取Manager，并最终通过Manager创建session对象。但是我们通过定位跟踪发现，Request获取到的Context对象竟然是空的。这是怎么回事呢？  我们调研一下Request对象实例中的Context对象是什么时候设置的。  Request对象第一次设置Context，是在HttpProcessor创建的时候，参考：   |  | | --- | | 1.HttpProcessor.HttpProcessor() // constructor  this.request = (HttpRequestImpl) connector.createRequest();  2.HttpConnector.createRequest()  request.setContext((Context) this.container); |   但是，HttpProcessor处理完客户端第一次http servlet请求之后，就回收了request对象：   |  | | --- | | 1.HttpProcessor.process()  request.recycle();  2.HttpRequestBase.recycle()  super.recycle()  3.RequestBase.recycle()  context = null; |   这就导致，我们第二次访问servlet的时候，Request对象中的context对象为空。 | | | |
| 解决方案 我们参考官方是怎么做的。我们通过查看RequestBase.setContext()方法，被哪些方法调用，来定位官方tomcat是怎么实现Request对象在处理每次客户端请求的时候，都有context对象。参考代码：   |  | | --- | | StandardHostMapper.map()  if (update) {  request.setContext(context);  } |   虽然StandardHost我们还没有涉及，但是总体的思路是清楚的，就是在Mapper端，将客户端servlet请求路径映射到实际的servlet类的时候，设置request请求。  有了这个思路，我们在现有条件下，尝试解决这个问题。参考：   |  | | --- | | SimpleContextMapper.map(){  request.setContext(this.context);  } |   有了这段逻辑，后续的servlet请求就不会报错了。 | | | |
| 总结 | | | |

# 问题6 StoreBase.processExpire()的Pending null session问题

|  |  |  |  |
| --- | --- | --- | --- |
| 标题 | StoreBase.processExpire()处理空闲过长的session造成的pending null session问题 | | |
| 提交时间 | 20210301 | 章节 | Chapter9 Session |
|  | | | |

## 问题描述

我们在调试StoreBase.processExpire()方法的时候碰到了问题。这个方法的作用是这样的：将持久化层中的session文件加载到内存，生成session对象。然后检测这些session，一旦发现这些session的空闲时间过长，就把这些session淘汰掉(recycle()或者expire())。

实际调试的时候，发现StoreBase将这些空闲时间过长的session淘汰掉(expire())之后，发现Manager有很多异常的session对象。

## 问题重现

接下来，我们尝试重现一下这个问题。重现步骤如下：

### 1.启动tomcat

一开始先只打开PersistentManagerBase.processMaxIdleSwaps()，将内存中空闲时间过长的session Swap out到持久层。同时，关闭StoreBase.processExpire()，防止StoreBase把持久层中的session文件搞乱。

### 2.创建session

SessionClient发起10个session创建请求，这些session信息为：

|  |
| --- |
| print session info StandardSession{sessionId='15840fc54dae4afdbb1e359d784070e1', isNew=true, isValid=true, maxInactiveInterval=60, createTime=1614437071311, lastAccessTime=1614437071311, thisAccessTime=1614437071311, expire=false}  print session info StandardSession{sessionId='e76f427a36c04b0f8ed5ad15ec922e5b', isNew=true, isValid=true, maxInactiveInterval=60, createTime=1614437071330, lastAccessTime=1614437071330, thisAccessTime=1614437071330, expire=false}  print session info StandardSession{sessionId='80334cb508554f43895383b8c53fd414', isNew=true, isValid=true, maxInactiveInterval=60, createTime=1614437071340, lastAccessTime=1614437071340, thisAccessTime=1614437071340, expire=false}  print session info StandardSession{sessionId='17b9aef1bf64436daa89a5548365d4ea', isNew=true, isValid=true, maxInactiveInterval=60, createTime=1614437071346, lastAccessTime=1614437071346, thisAccessTime=1614437071346, expire=false}  print session info StandardSession{sessionId='fe20e04eeb22408189a2250c94cf9944', isNew=true, isValid=true, maxInactiveInterval=60, createTime=1614437071343, lastAccessTime=1614437071343, thisAccessTime=1614437071343, expire=false}  print session info StandardSession{sessionId='3d0109cec2f14318b9b622c37eff5d7a', isNew=true, isValid=true, maxInactiveInterval=60, createTime=1614437071355, lastAccessTime=1614437071355, thisAccessTime=1614437071355, expire=false}  print session info StandardSession{sessionId='a43b455f551b458ca27955f8186e4dab', isNew=true, isValid=true, maxInactiveInterval=60, createTime=1614437071357, lastAccessTime=1614437071357, thisAccessTime=1614437071357, expire=false}  print session info StandardSession{sessionId='389533ce69ad405b81553e6c91eaa24c', isNew=true, isValid=true, maxInactiveInterval=60, createTime=1614437071334, lastAccessTime=1614437071334, thisAccessTime=1614437071334, expire=false}  print session info StandardSession{sessionId='ba02630b99cc49aca9b02afe8acc6cab', isNew=true, isValid=true, maxInactiveInterval=60, createTime=1614437071337, lastAccessTime=1614437071337, thisAccessTime=1614437071337, expire=false}  print session info StandardSession{sessionId='7bb84d5301314f11ba4fbd931475bf94', isNew=true, isValid=true, maxInactiveInterval=60, createTime=1614437071360, lastAccessTime=1614437071360, thisAccessTime=1614437071360, expire=false} |

### 3.Session swap out to persistence

通过PersistentManager异步线程，将(内存中)空闲时间过长的session swap out 到持久化层。可以看到，持久化层中的session文件刚好是10个：

|  |
| --- |
| 15840fc54dae4afdbb1e359d784070e1.session  17b9aef1bf64436daa89a5548365d4ea.session  389533ce69ad405b81553e6c91eaa24c.session  3d0109cec2f14318b9b622c37eff5d7a.session  7bb84d5301314f11ba4fbd931475bf94.session  80334cb508554f43895383b8c53fd414.session  a43b455f551b458ca27955f8186e4dab.session  ba02630b99cc49aca9b02afe8acc6cab.session  e76f427a36c04b0f8ed5ad15ec922e5b.session  fe20e04eeb22408189a2250c94cf9944.session |

这时，我们通过日志可以看到，内存中的session为0，recycle的session为10.

### 4.打开StoreBase.processExpire，重启tomcat

打开StoreBase.processExpire后，观察StoreBase是如何将持久化层中的过期Session Expire掉的：

首先，持久化层中的session文件确实清理掉了。这说明StoreBase处理持久化成中过期session的动作是实现了的。

然后，观察PersistentManager管理的session:

|  |
| --- |
| current session count(in memory) : 10  current session count(recycled) : 1 |

可以看到，内存中居然还有10个session，打印这些session详细如下表所示。可以看到这些内存中的session明显是异常session，因为这些session的sid都是为空。

这说明，StoreBase处理持久层session文件是有问题的。这些session按照正常流程，从持久化层加载进来后，一旦发现空闲时间过长，应该被expire掉，而不应该重新放到内存中。至此，BUG重现了。

|  |
| --- |
| print session info StandardSession{sessionId='null', isNew=false, isValid=false, maxInactiveInterval=-1, createTime=0, lastAccessTime=0, thisAccessTime=1614437071334, expire=false}  print session info StandardSession{sessionId='null', isNew=false, isValid=false, maxInactiveInterval=-1, createTime=0, lastAccessTime=0, thisAccessTime=1614437071334, expire=false}  print session info StandardSession{sessionId='null', isNew=false, isValid=false, maxInactiveInterval=-1, createTime=0, lastAccessTime=0, thisAccessTime=1614437071334, expire=false}  print session info StandardSession{sessionId='null', isNew=false, isValid=false, maxInactiveInterval=-1, createTime=0, lastAccessTime=0, thisAccessTime=1614437071334, expire=false}  print session info StandardSession{sessionId='null', isNew=false, isValid=false, maxInactiveInterval=-1, createTime=0, lastAccessTime=0, thisAccessTime=1614437071334, expire=false}  print session info StandardSession{sessionId='null', isNew=false, isValid=false, maxInactiveInterval=-1, createTime=0, lastAccessTime=0, thisAccessTime=1614437071334, expire=false}  print session info StandardSession{sessionId='null', isNew=false, isValid=false, maxInactiveInterval=-1, createTime=0, lastAccessTime=0, thisAccessTime=1614437071334, expire=false}  print session info StandardSession{sessionId='null', isNew=false, isValid=false, maxInactiveInterval=-1, createTime=0, lastAccessTime=0, thisAccessTime=1614437071334, expire=false}  print session info StandardSession{sessionId='null', isNew=false, isValid=false, maxInactiveInterval=-1, createTime=0, lastAccessTime=0, thisAccessTime=1614437071334, expire=false}  print session info StandardSession{sessionId='null', isNew=false, isValid=false, maxInactiveInterval=-1, createTime=0, lastAccessTime=0, thisAccessTime=1614437071334, expire=false} |

## 问题定位

问题也重现了，那么问题可能出现在哪里呢？我们分析一下，问题最有可能出现在哪里？这些内存中的异常session信息，是怎么来的呢？当然是StoreBase从持久化文件中带过来的。那么StoreBase怎么处理那些持久化文件的呢：

1. 将持久化文件从文件系统load到内存；
2. 对session访问时间进行校验；
3. 一旦发现session空闲时间过久，就直接将这个session expire掉；

从日志来看很明显，内存中那些异常的null session，应该是从持久化层中加载进来之后，没有及时清理导致的。

加载的动作是FileStore.load()方法执行的，load进来之后，发现session空闲时间过久，又马上expire掉。

我们通过DEBUG，定位一下StoreBase处理过程中，Session信息的状态。通过DEBUG，我们发现，StoreBase在处理持久层的session信息的时候，将session从文件系统load到内存之后，PersistentManager正好也会访问这个session，然后发现这个session的空闲时间很长，所以又会把这个session swap out to persistent，同时回收这个session。这就导致了StoreBase在处理的时候，这个session已经被回收了，session各个字段也已经被置为初始值。

因此，为避免受到PersistentManager的影响，我们暂时关闭异步线程中的PersistentManager.processExpire()方法。

但是我们关闭PersistentManager.processExpire()方法之后，发现经过StoreBase处理之后的session，还是有问题：

|  |
| --- |
| current session count(in memory) : 10  current session count(recycled) : 1  print session info StandardSession{sessionId='null', isNew=false, isValid=false, maxInactiveInterval=-1, createTime=0, lastAccessTime=0, thisAccessTime=1614498887184, expire=false} |

可以看到，在排除了PersistentManager干扰之后，StoreBase处理的session还是进入到Manager管理的session列表中，而且这些session的状态是异常的。

我们再重新DEBUG。看看StoreBase是怎么处理持久层session的。我们在DEBUG过程中，终于发现了问题所在。

问题是这样的，StoreBase检测持久层session之前，首先要把session文件load到内存中。load步骤的实现是这样的：

1. manager创建一个新的session；
2. 从持久化文件中，读取session各个信息；
3. 将持久化文件中解析出来的session各个字段，赋给新创建的session对象；

以上三个步骤看起来没问题，但是请注意，manager中保留的session对象列表，是key/value格式的：

<key,value>:<sessionId,Session对象>

我们更新了其中Session对象中的sessionId，但是这个key还是manager创建session时所用的随机ID！这就导致了key/value中的key(sessionId)和value中的Session对象中的sessionId不一致！这会导致后续我们将session expire的时候，没法真正把manager中我们想要删除的session删除！

## 解决方案讨论1

既然知道了问题原因，怎么解决呢？其实官方的实现是没问题的，只是我自己写代码的时候没有注意到这个细节：

|  |
| --- |
| 参考代码：  StandardSession.setId()  原来的代码：  public void setId(String id) {  // 设置session id  this.sessionId = id;  if(null != id && null != this.manager){  manager.remove(this);  }    if(null != manager){  manager.add(this);  } }  优化后的代码：  public void setId(String id) {  if(null != id && null != this.manager){  manager.remove(this);  }  // 设置session id  this.sessionId = id;  if(null != manager){  manager.add(this);  } } |

具体参考StandardSession.setId()方法的注释。

修改之后，发现问题还是依旧存在：

|  |
| --- |
| current session count(in memory) : 9  current session count(recycled) : 1 |

我们注意到，已经有一个session放到recycle队列了，已经之前好了一点。但还是有9个异常session放到内存了。

这是怎么回事呢？我们再看StoreBase.processExpire()的逻辑。我们看到StoreBase从持久化层load一个session文件，这个load是由FileStore实现的，实现逻辑就是先通过manager(ManagerBase)创建一个session。

我们知道，第一个session load的时候是创建一个完全新的session，然后马上expire掉。后续9个session load的时候是复用了第一个Recycled session。

那么，通过recycle创建的session和全新创建的session有什么不同呢？

我们继续DEBUG。

我们研究StoreBase.processExpires()方法，发现这里有一个细节：

|  |
| --- |
| StandardSession session = (StandardSession) load(keys[i]); |

StoreBase去加载持久化文件系统中的session对象，Manager中管理这个session的时候是这样的：

|  |
| --- |
| Key/value  newSessionId/persistentSessionInstance |

不知道有没有说明白，就是manager维持的这个session实例，实例信息虽然是从持久层加载进来的字段(包括session id)，但是manager中的key是一个新的随机sessionId。你可能会觉得困惑，key和session实例中的sessionId不是应该一致吗？在大多数情况下是这样的，我们只要看一下ManagerMase.createSession()方法就能看出来，无论session实例是新建的还是从recycled队列中复用的，key和session实例的sessionId是一致的。

但是在StoreBase.processExpires()中，load进来的session，就是不一致。虽然不太合理，但是这么做是对的。因为processExpires()后续的逻辑会判断这个Load进来的session，是否纳入了manager管理，意思就是在load之前，这个session已经由Manager在内存中管理了，一般是backup(持久层中的session只是对内存中session对象的备份)的场景，或者由客户端通过managerBase.findSession(sessionId)访问过这个session。

如果我们针对StoreBase.processExpires()中load进来的session，做下调整，那就是自作聪明了。比如我们在StandardSesion.readObjectData()方法中，通过session.setId()方法设置sessionId，调整manager管理这个session的key，那就会有问题。StoreBase.processExpires()会把这个Session当做是之前已经在内存中的Session，从而只做recycle()处理，而不做expire()处理。

Recycle()和expire()的区别是显而易见的。Recycle()只是对session各个字段置为空，并且将Session纳入manager recycle队列。而expire()不仅会recycle()，而且会将这个session从manager管理的session队列中remove掉。这个区别还是挺大的。

在StoreBase.processExpires()中，如果session之前已经在内存中了，那么StoreBase只需要将这个session recycle()即可，expire的动作会由Manager来实现。如果session之前没有在内存中，那么不仅需要recycle，还要把expire的事情给做了。

但是，需要特别注意，我们在StoreBase.processExpires()中，对session做expire()操作的时候，，并不能把我们从持久层load进来的session实例删除。那是因为，在expire()方法逻辑中，将session从manager中remove掉的时候，Manager中对应session的key是新的随机sessionId，并不是我们从持久层中load进来的session的sessionId！因此，StandardSession.exprire()方法中，manage.remove(this)并不能删除session。

这就导致了由大量异常session实例还在由manager管理。这些异常的session就成了manager管理下的pending expired sessions。

那么这些pending sessions会不会有问题呢？后续能否用于回收呢？我们的答案是，能用于回收，但是这些pending sessions会一直伴随着manager，没法删除了。因为在Manager中，这些session对应的key会一直存在。

按照正常来说，manager管理下的session，sessionid和manager中对应的key应该是要一致的。这样在这个session expire的时候，就能够顺利将这个session对象从manager管理的session map中删除。这个问题的本质就是，FileStore从持久化层Load到session之后，通过manager先创建了一个新的session，后来又设置了一下这个session的sessionId，导致sessionId和manager管理的这个session的key不一致。

后续我们要在程序中特别特别注意：通过manager创建完session之后，千万别再(通过调用StandardSession某个方法)修改sessionId了。

## 解决方案讨论2

我们在”解决方案讨论1”中，讨论了这个问题的根源，那么，怎么解决呢？

我觉得可以在StoreBase.processExpire()方法结束前，将Manager管理的这个null pending session从Manager管理中remove掉。因为这个pending虽然经过了recycle()或者expire()处理，但是这个对象实例毕竟还在。我们试试能否根据value从map中把对应的<key/value>给remove掉。这个其实是可以的，Demo参考MapTest.java。StoreBase的实现方案参考：

|  |
| --- |
| StoreBase.processExpire()  /\*\*  \* 全部处理完成后，当前的expire session无论是recycle()还是expire()  \* 都会存在pending null session的问题  \* 即我们从持久层load到当前这个session之后，这个session会纳入manager管理  \* 但是当这个session对象recycle()或者expire()之后，manager中会有一个pending null session  \* 需要我们手工删除  \*/ ((PersistentManager)manager).removePendingSession(session); |

这个方式当然是可以的，解决了StoreBase.processExpire()场景下的问题。但是其他场景呢？比如PersistentManager.findSession()方法中，从持久化层加载的session，也会存在pending session的问题。这个问题的本质原因是，我们通过FileStore.load()把session load进来之后，管理这个session的key总是一个随机id。

我们参考一下新版本的tomcat，发现新版本的tomcat(9.0.30)，就有现成的解决方案。解决思路是这样的：在FileStore.load()方法中，我们创建session对象的时候，不是调用原来的manager.createSession()方法(这个方法返回的sessionId随机，且会自动纳入manager管理)，而是使用manager.createEmpty()方法。通过这个方法返回的session是一个完全空的session，没有sessionId，也不会纳入namager管理)，这个session就是一个独立的对象。这样的话，无论是StoreBase.processExpire()，还是PersistentManager.findSession()，都不会产生pending null session的问题。备注：PersistentManager.findSession()中，从持久层load到session对象之后，需要手动将session add到manager的sessions队列中。

## 最终解决方案

通过两轮的解决方案讨论，我们最终采用的解决方案是，在

## 总结

# 问题X模板

|  |  |  |  |
| --- | --- | --- | --- |
| 标题 |  | | |
| 提交时间 | 20210301 | 章节 | Chapter9 Session |
| 问题描述 | | | |
| 异常日志 | | | |
| 问题分析 | | | |
| 解决方案 | | | |
| 总结 | | | |