原 [OOM]记一次线上OOM的问题

目录(?) [-]

背景

现象：

收到某台服务器无法正常提供服务的报警，应用日志里没有新内容，再看catalina.out日志，发现报出OOM异常。

Tomcat日志

分析问题

分析问题过程中的想法和方法是最有价值的经验

因为OOM属于系统抛出的错误，因此没有写入到应用日志里，只在catalina.out日志里面有。

通过命令jmap -heap pid查看堆内存信息，发现空间确实满了。

考虑到有3G的内存，把内存dump出来查看本机受不了，故使用jmap -histo:live pid命令，查看对象的统计信息。如下图：

```
 num     #instances         #bytes  class name
----------------------------------------------
   1:       1291920     2170067936  [B
   2:       3831489      321322296  [C
   3:       3751720      120055040  java.util.concurrent.ConcurrentHashMap$HashEntry
   4:       2463371       78827872  java.util.concurrent.CountDownLatch$Sync
   5:       2597263       62334312  java.lang.String
   6:       1231685       49267400  org.apache.http.entity.BasicHttpEntity
   7:       1231685       49267400  org.apache.http.impl.io.ContentLengthInputStream
   8:       2463371       39413936  java.util.concurrent.CountDownLatch
   9:       1231685       39413920  com[redacted]gent.support.httpclient.CachedBufferedInputStream
  10:       1231687       29560488  java.io.ByteArrayOutputStream
  11:       1231685       29560440  org.apache.http.message.BufferedHeader
  12:       1231685       29560440  org.apache.http.util.CharArrayBuffer
  13:       1231685       29560440  org.apache.http.conn.BasicManagedEntity
  14:       1231685       29560440  org.apache.http.conn.EofSensorInputStream
  15:          7670       25767744  [Ljava.util.concurrent.ConcurrentHashMap$HashEntry;
```

重点关注排名第九的类，因为它是我们架构组写的代码。

找到该类查看后，发现里面确实有静态的ConcurrentHashMap对象。初步可以断定是该类内存溢出导致。

粗略查看了下代码，发现静态的Map对象存在没有被释放的可能。而且代码里有些并发的逻辑写的也有些问题。

Eclipse的MAT工具

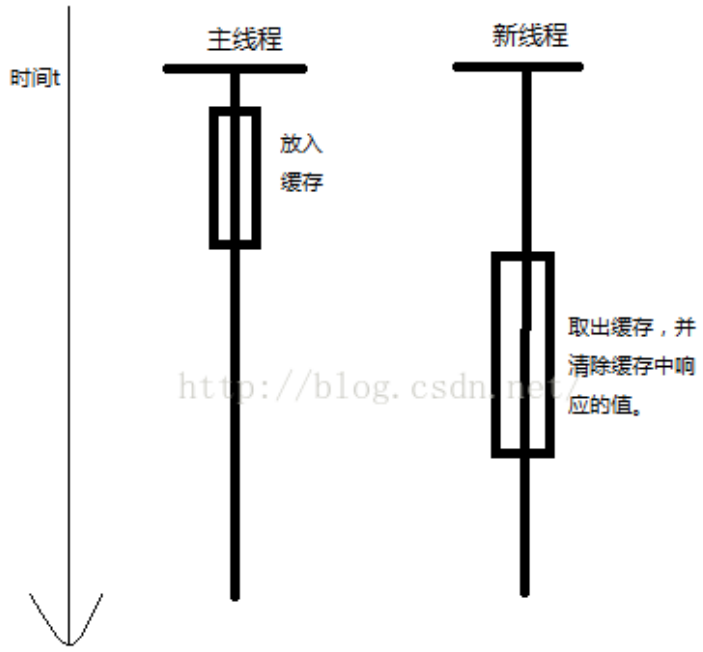
另外，将堆内存信息dump出来（jmap -dump:live,format=b,file=1.dump），然后让同事下载到本机用工具进行查看，发现这个类中Map对象数量惊人，达到了2-3G。

问题原因

将问题定位后，找架构组同事了解代码的功能：主线程拦截HttpClient的execute方法，将输出信息流CachedBufferedInputStream保存到Map里缓存起来；另外开启一个线程，在请求执行成功时，获取到缓存里的信息流，同时清除缓存。

新开启的线程为了保证缓存中有数据，使用了CountDownLatch的等待机制，当主线程将数据放入缓存后，调用countDown()方法；新线程会等待3s，除非主线程已将数据放入缓存。

时序图如下：



举一反三

但是经过分析代码、调试，发现出现内存溢出的几种可能性：

1、虽然新线程里有通过CountDownLatch机制等待3s，如果在特殊情况下，新线程等待3s后，数据依旧没有放入，那么这条数据就会一直在缓存中不被删除。

2、代码中逻辑不完善，如果出现异常或者状态码不对，也无法删除缓存数据。核心代码如下：

```
try {
    if (innerLog.t != null) {
        retCode = innerLog.t.getClass().getSimpleName();
        retMsg = innerLog.t.getLocalizedMessage();
    } else {
        if (response != null) {
            Class cls = response.getClass().getInterfaces()[0];
            Method getStatusLine = findMethod(cls, "getStatusLine");
            Object statusLine = getStatusLine.invoke(response);
            Method getStatusCode = findMethod(statusLine.getClass(), "getStatusCode");
            Method getReasonPhrase = findMethod(statusLine.getClass(), "getReasonPhrase");
            int statusCode = (Integer) getStatusCode.invoke(statusLine);
            if (statusCode == 200) {
                isSuccess = true;
                retCode = "200";
                String content = null;
                CachedBufferedOutputStream outputStream = CachedBufferedOutputStream.getCache(innerLog.traceId + TraceContext.SEPARATOR + innerLog.rpcId);
                if (outputStream != null) {
                    content = outputStream.getBuffer();
                    outputStream.closeCache();
                } else {
                    CachedBufferedInputStream input = CachedBufferedInputStream.getCache(innerLog.traceId + TraceContext.SEPARATOR + innerLog.rpcId);
                    if (input != null) {
                        content = input.getBuffer();
                        input.closeCache();
                    }
                }
            } else {
                retCode = statusCode + "";
                retMsg = (String) getReasonPhrase.invoke(statusLine);
            }
        }
    }
    return traceLog;
} catch (Throwable e) {
    LOGGER.log(Level.SEVERE, "[katarina-agent] handleHttpClientLog error (" + innerLog.traceId + ")(" + url + ") :", e);
}
```

图中4是正常释放缓存的逻辑；释放资源放在finally块中

图中1：如果http请求返回不为200，无法释放缓存；

图中2：如果中间任何代码段中出现任何异常，也无法释放缓存。事实上，本次OOM问题发生的主要原因就是这个。

图中3：取出缓存中的数据，这块会等待3s后去缓存中的数据。主要代码如下：

```
public static CachedBufferedInputStream getCache(String cacheKey) {
    CountDownLatch wait_downLatch = WAIT_CONSUME_DOWNLATCH.get(cacheKey);
    if (wait_downLatch == null) {
        wait_downLatch = new CountDownLatch(1);
        WAIT_CONSUME_DOWNLATCH.put(cacheKey, wait_downLatch);
    }
    try {
        if (wait_downLatch.await(WAIT_TIMEOUT, TimeUnit.MILLISECONDS)) {
            return CACHED_BUFFERED_INPUT_STREAM_CACHE.get(cacheKey);
        } else {
            return null;
        }
    } catch (InterruptedException e) {
        return null;
    }
}
```

图中2表示会等待3s后再从缓存中获取数据。但如果超过3s后，缓存里依旧没有数据，那也无法清除缓存。

图中3即为内存溢出的Map对象。

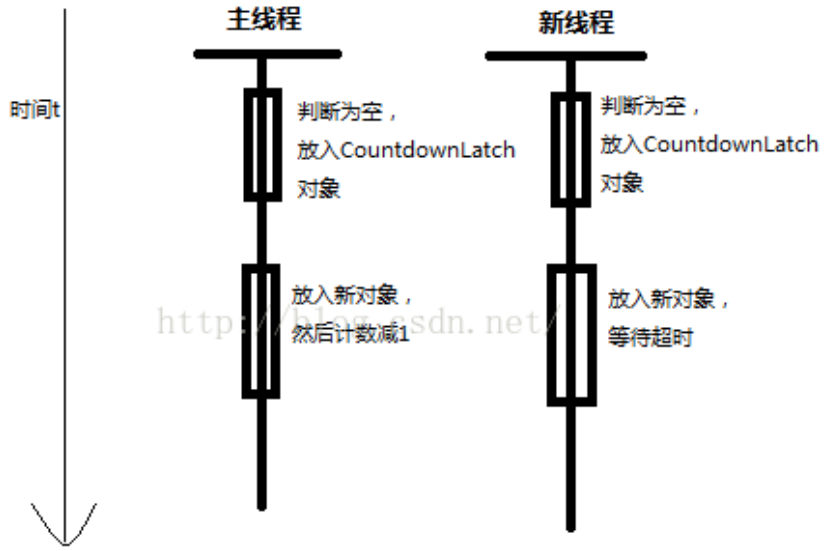
图中1是放入CountDownLatch对象，这块也有问题。请看下面主线程中的主要代码：

```
public static InputStream getInputStream(InputStream in, Object httpEntity) {
    String traceContext = HTTP_ENTITY_CACHE.get(httpEntity);
    if (traceContext != null) {
        if (!CACHED_BUFFERED_OUTPUT_STREAM_CACHE.containsKey(traceContext)) {
            CachedBufferedInputStream old = CACHED_BUFFERED_INPUT_STREAM_CACHE.get(traceContext);
            if (old != null && old.getInputStreamClass() == in.getClass()) {
                return old;
            }
            CachedBufferedInputStream inputStream = new CachedBufferedInputStream(in, traceContext);
            CachedBufferedInputStream override = CACHED_BUFFERED_INPUT_STREAM_CACHE.put(traceContext, inputStream);
            CountDownLatch wait_downLatch = WAIT_CONSUME_DOWNLATCH.get(traceContext);
            if (wait_downLatch == null) {
                wait_downLatch = new CountDownLatch(1);
                WAIT_CONSUME_DOWNLATCH.put(traceContext, wait_downLatch);
            }
            wait_downLatch.countDown();
              LOGGER.info("[katarina agent]  CachedBufferedInputStreamCache put ,inputStream[" + in + "],httpEntity[" + httpEnti
            if (override != null) {
                LOGGER.warning("[katarina agent]  CachedBufferedInputStream was override ,traceContext[" + traceContext + "]");
            }
            return inputStream;
        }
    }
    return in;

}
```

图中2，也会放入一个CountDownLatch对象。该图中2与上图1，因为是两个线程，所以可能会出现上图中1的CountDownLatch对象覆盖下图中2的对象，这样会上图1中永远也不会从缓存中取得数据，因为等待超时了。

时序图如下：



解决方案

既然知道了问题原因，解决方案就好办了，这里就不赘述了。因为这块是其他组的代码，对其内部逻辑不是特别清楚，所以只看了相关的代码。但就在这些片段代码里，并发问题这么多，可见代码质量的好坏非常重要。