**工行桌面集成系统宕机分析**

1. **系统概述**

目前，此系统为基于桌面集成的信息推送，网点端注册后，如果有消息则从总行把消息推送到网点端，应用共分为三部分：网点端应用、一级行应用和总行应用。

1. **桌面集成系统的物理部署架构**

桌面集成系统的部署模式为网点、一级行和总行三层，其通信模式如下：

* 网点应用通过订阅模式的长连接与一级行进行连接，
* 一级行收到网点消息后通过短连接与总行通信，中间经过F5进行负载均衡；
* 总行收到分行消息后建立与分行的长连接进行消息交互。
* 分行经过对消息的处理通过第一步网点与分行建立的长连接进行消息的推送。

其中分行、总行的应用为CTP应用，通过其提供的渠道、通道进行通信。

其部署图如下：

目前，一级行应用共有35家左右，总行应用6个，最多的一级行下有5000左右的客户端与一级行相连。

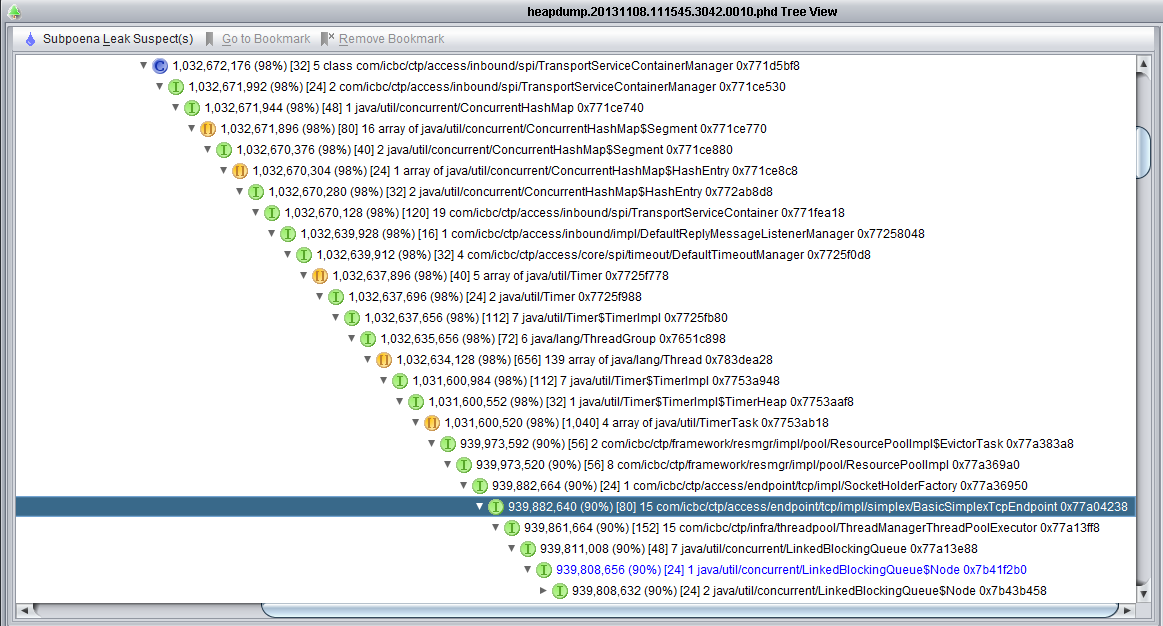
分行应用的操作系统为linux，JDK版本为Java version : JRE 1.6.0 IBM J9 2.4 Linux x86-32 build jvmxi3260sr9-20101209\_70480。

1. **宕机现象**

在总行、分行应用的使用过程中，针对每个应用都有出现过因为ＯＯＭ而导致的系统宕机发生，其发生ＯＯＭ的原因也不太一致，通过对ｄｕｍｐ文件的分析，大致存在三种导致ＯＯＭ的原因。

* 渠道回写wirtebuffer过大导致的溢出
* 订阅渠道中接收数据量过大导致渠道的程池的queue队列中的任务数过大
* 通道线程池的queue队列中的任务数过大

下图为第3种情况通道线程池的queue队列中的任务数过大的dump分析图，从图中可以看到在我们的单工通道的线程池中堆积了大量的待处理的任务，占到了内存占用总量的90%。



1. **问题定位**

* 针对第一个OOM原因：渠道回写wirtebuffer过大导致的溢出

查看Netty代码主要出现在类NIOWorker中，执行方法writeFromUserCode进而执行scheduleWriteIfNecessary

|  |
| --- |
| **protected** **void** writeFromUserCode(**final** NioSocketChannel channel) {  ………  **if** (**scheduleWriteIfNecessary**(channel)) {  **return**;  }  ………  write0(channel);  } |

|  |
| --- |
| **private** **boolean** scheduleWriteIfNecessary(**final** NioSocketChannel channel) {  **final** Thread currentThread = Thread.*currentThread*();  **final** Thread workerThread = thread;  **if** (currentThread != workerThread) {  **if** (channel.writeTaskInTaskQueue.compareAndSet(**false**, **true**)) {  **boolean** offered = writeTaskQueue.offer(channel.writeTask);  **assert** offered;  }  ………  } |

可以看到，写的任务放到了队列writeTaskInTaskQueue当中，如果写的速度慢过消息的发送速度，则会造成写队列的堆积。

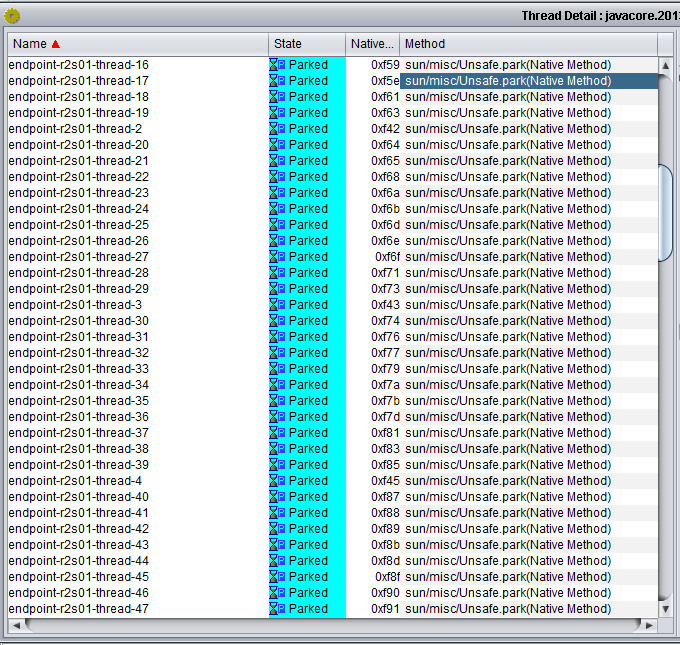
此案例在杭州FMTS项目也发生过，原因为客户端接收数据的速度太慢，后来采用客户端消息接收与消息处理分离的方式进行了解决。在工行桌面集成中因后面进行了修改我去时已没有此现象了，仅听说以前出现过，没见到具体的dump文件。

* 针对第二种OOM的原因，订阅渠道中接收数据量过大导致渠道的程池的queue队列中的任务数过大。

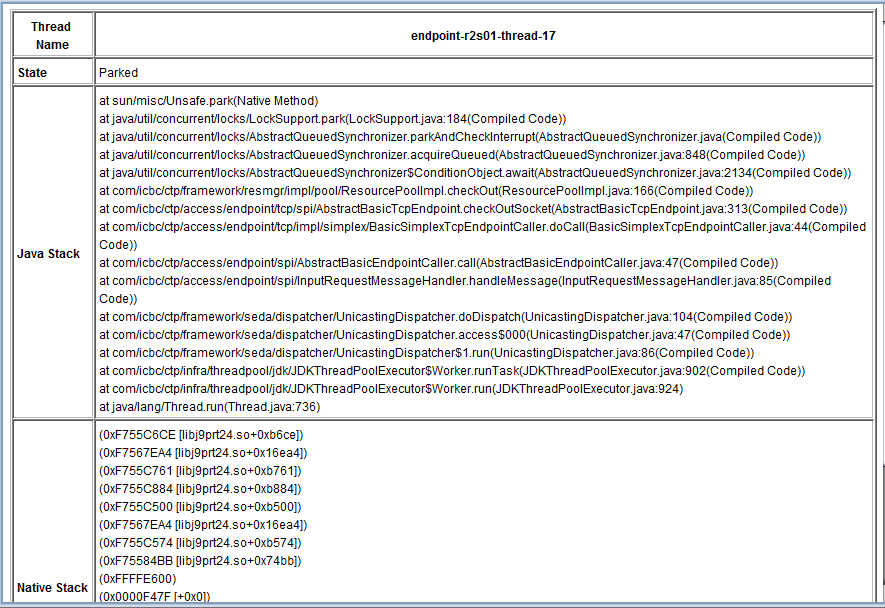
经过分析，在订阅渠道中针对每条链路都绑定了一个数据队列（此功能为以前FMTS项目订制的，保证同一链路数据的有序操作），然后通过渠道的线程池进行数据的处理。分行与总行目前采用一条链路的方式进行连接，通过这一条链路发送的数据量如果过大，将导致数据队列的过大，同时也没有充分利用渠道线程池的处理能力。处理方案为通道中建立多条链路与总行相连，充分利用线程池的效率；同时针对本项目，处理数据时可以采用无序的方式，没必要对链路进行绑定。

* 针对第三种OOM情况，通道线程池的queue队列中的任务数过大。

分析发生OOM时的线程栈，可以看到通道的线程池中的所有线程都处于park状态（分行应用中配置为50），如下图：



每个线程的调用栈如下图：



从上图中可以看到，所有的通道线程都处于从资源池中checkput资源，因所有的资源都有释放，导致接下来请求的数据而不能处理进而堆积到通道的线程池队列中，最后导致OOM发生。

经过初步的代码分析，判断出通道的资源池出现了问题，可能由于某种原因导致资源没有被checkin进来。目前通道（单工）的checkin的代码如下，类AbstractModeSocketNettyHandler：

|  |
| --- |
| **public** **void** writeComplete(ChannelHandlerContext ctx, WriteCompletionEvent e)  **throws** Exception {  **if** (isSimplex()) {  closeSocket(ctx);  } **else** {  **if** ("one-way".equals(ctx.getAttachment())) {  closeSocket(ctx);  checkIn(ctx.getChannel());  }  }  **if** (!isHalfDuplex()){//半双工在messageReceived后checkIn  checkIn(ctx.getChannel());  }  setAllowInput();  } |

而对于其它一些特殊情况而没做checkin操作。其解决方案如下,我们修改其代码，在发生异常exceptionCaught以及channelClosed时增加了checkIn的动作，同时我们把资源池的唤醒机制由signalAll修改为signal。我们把修改后的代码重新部署到了分行（广东行）的应用中，原本一天就出现的OOM没有再次出现。

1. **经验总结**

* 对于通信类程序，渠道与通道的通信模式一定要匹配，不能一边为长连一边为短连
* 渠道、通道的各类参数一定要与生产环境相适应，不能像桌面项目全部采用原默认的
* 系统的异常情况都要处理好

1. **遗留问题**

按正常情况，通道中配置的50个线程全处理park状态，我们在资源池中是有5秒的超时配置，如果经过5秒还没有checkout资源，则会抛出异常，但我们在日志文件中没有看到此异常，感觉好像线程都死掉了。

|  |
| --- |
| try {  if(maxWait <= 0) {  termination.await();  }else{  final long elapsed = (System.currentTimeMillis() - starttime);  final long waitTime = maxWait - elapsed;  if (waitTime > 0)  {  **termination.await(waitTime, TimeUnit.MILLISECONDS);**  }  }  } catch (InterruptedException e) {  throw new RuntimeException("Await be interrupted.");  }  }  **if(maxWait > 0 && ((System.currentTimeMillis() - starttime) >= maxWait)) {**  **throw new RuntimeException("Timeout waiting for idle resource! "+ maxWait +" ms.");**  **}** |