

MAT实战分析内存溢出

1. 背景

四个闪贷要上贷中需求，测试发现用户段某某在沅澧闪贷进行贷中跑批时，沅澧闪贷超脑多次停止服务，初步定位是内存溢出导致。

之前测试阶段，为了定位该内存溢出的问题，测试和开发用了不少时间，所以特地拿出这个案例来分享，希望大家以后遇到同样的问题，可以有值得借鉴的地方。

2. 目标

掌握本文章定位内存溢出的技巧

实现关键思路：

1. 掌握在hrxj-bigbrain超脑应用中，增加输出内存溢出日志文件配置的方法
2. 掌握使用Mat工具分析内存溢出位置点的方法
3. 掌握使用DBeaver工具，快速执行带有参数的kettle脚本的方法

3. 实现过程

3.1 先复现内存溢出的场景和现象

复现方法：重新同步行方用信文件，并让其中尾号0040的用户重新采集贷中政务数据

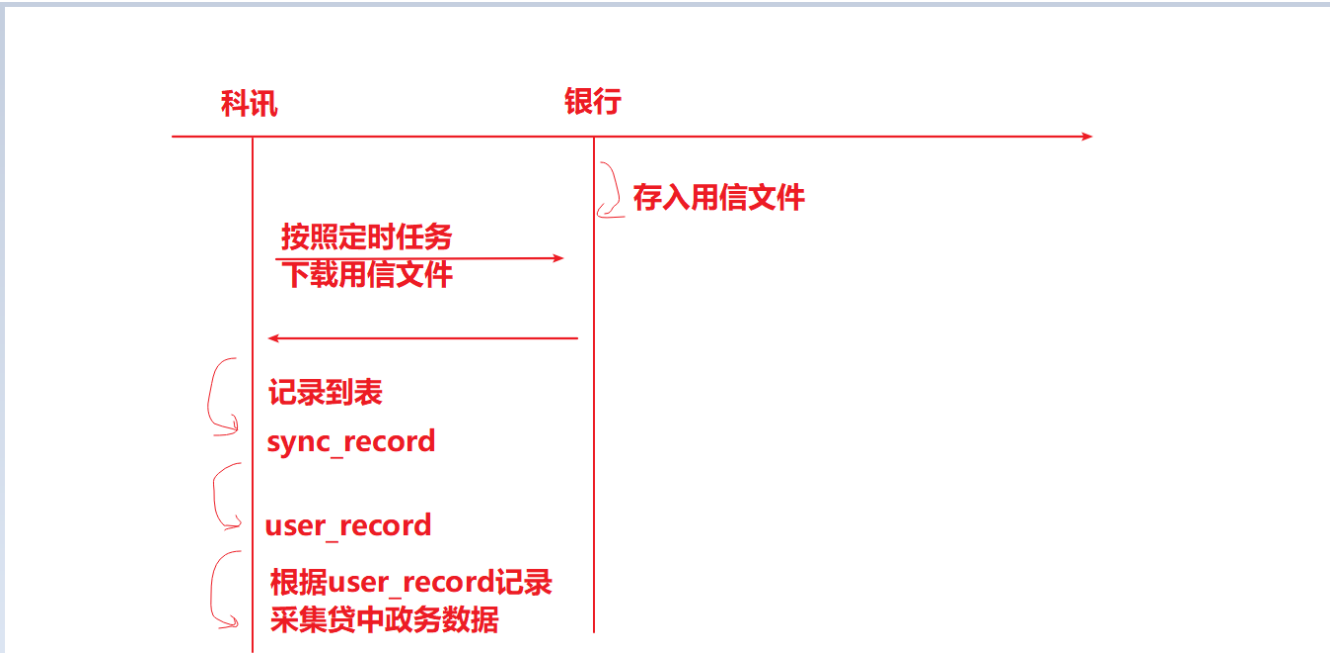
尾号0040的用户同步状态改成1后，会触发自动采集贷中政务数据

同步的定时任务配置：

```
#??????????
DATA_BANK_LOAN_USER_SYNC_TIME=30 0/1 * * * ?

#????????????
DATA_BANK_LOAN_HANDLE_TIME=30 0/1 * * * ?
```

示意图：



SELECT * from data_bank_loan_user_record where ID_CARD = '432502198906120040'; -- 行方用户贷款记录 2401189659900016

SETTLE_TIME	LOAN_TIME	CREATE_TIME	123 FLAG	ABC SYNC_DATE
[NULL]	[NULL]	2024-01-16 11:53:30.0	1	202401

数值查看器 2024-01-16 11:53:30.0

卡死的日志如下：

```

- | 2024-02-20 10:46:33.092 | DEBUG | localhost | DubboServerHandler-10.4.190.69:20891-thread-107 | o.s.jdbc.datasource.DataSourceUtils > 327 | | | Returning JDBC Connection to DataSource
- | 2024-02-20 10:46:33.092 | INFO | localhost | DubboServerHandler-10.4.190.69:20891-thread-107 | c.i.bigbrain.bs.FinsIntegrateSvrBs > 196 | | | 实时授信政务数据整合开始:applyId=2309209659900017,member=0,idCard=432502198906120040,bankCode=96599
- | 2024-02-20 10:46:33.092 | INFO | localhost | DubboServerHandler-10.4.190.69:20891-thread-107 | c.i.bigbrain.bs.FinsIntegrateSvrBs > 230 | | | 脚本入参: {filepath=/home/bigbrain/batch/kettle/ETLitem, SEQ_NO=230920965990001704630, APPLY_ID=2309209659900017, CUST GROUP=qtgsh, AREA CODE=96599, ZW SPOUSE JOB=cxjm, TYPE=0, PRIPID=, ID CARD='432502198906120040'}

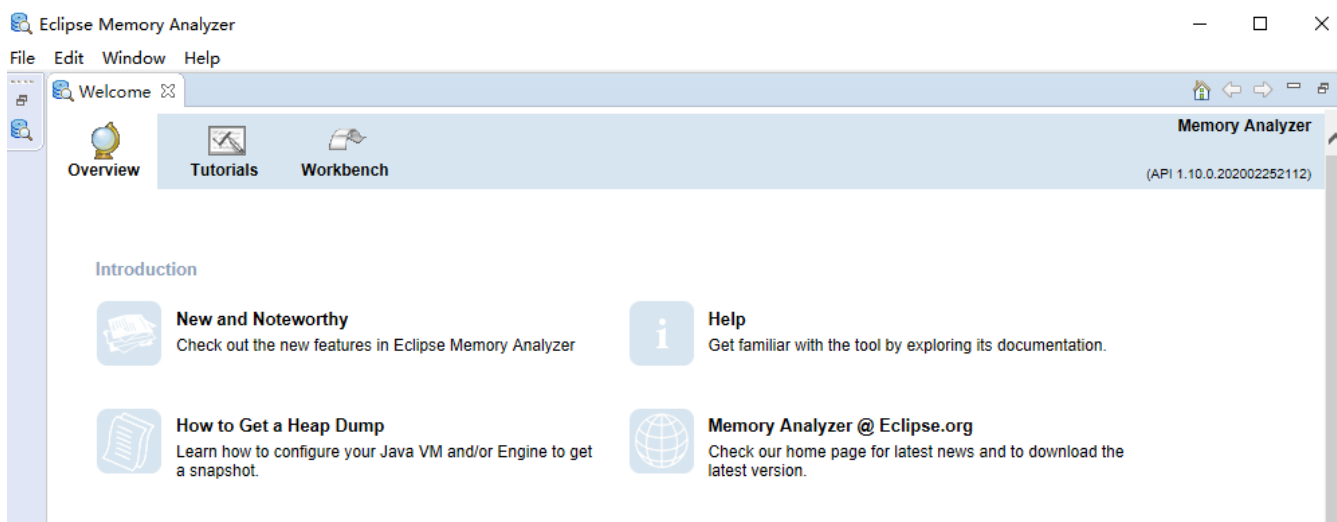
```

卡死后，超脑不再有响应。

现在卡死了，怎么知道他是响应超时，还是内存溢出？

这里由于没有典型的内存溢出日志: OutOfMemmmory, 所以只能上工具。

我们可以使用jvisualvm工具动态监控，开启监控后，再将问题场景操作一遍就监控到了。这个过程就不做演示了。总之，这里经过分析，是内存溢出导致的。但是为了定位内存溢出的原因，我们需要使用工具**MAT**



而使用MAT分析时，需要先拿到内存溢出的hprof文件

3.2 生成内存溢出的hprof文件

在内存溢出的服务中，增加下载溢出的hprof文件的配置

```

JAVA_MEM_OPTS="-server -Xmx512m -Xms512m -Xmn256m -XX:PermSize=128m -Xss256k -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactFullCollection -XX:LargePageSizeInBytes=128m -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -XX:CMSInitiatingOccupancyFraction=70 -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/home/bigdata/brain/logs"

echo -e "Starting the $SERVER_NAME ...c"
nohup java $JAVA_OPTS $JAVA_MEM_OPTS $JAVA_DEBUG_OPTS $JAVA_JMX_OPTS $JACOCO_OPTS -classpath $CONF_DIR:$LIB_JARS com.compas.core.container.Bootstrapper > $STDOUT_FILE 2>&1 &

```

```
-XX:+HeapDumpOnOutOfMemoryError  
-XX:HeapDumpPath=/home/bigbrain/logs
```

添加策略：哪个JAVA服务内存溢出，就在那个服务的启动命令上添加配置

例如：这里是超脑内存溢出，所以在超脑的启动命令上添加配置

然后重启服务，再重现刚才卡死的场景，就能在配置的/home/bigbrain/logs目录下看到hprof文件

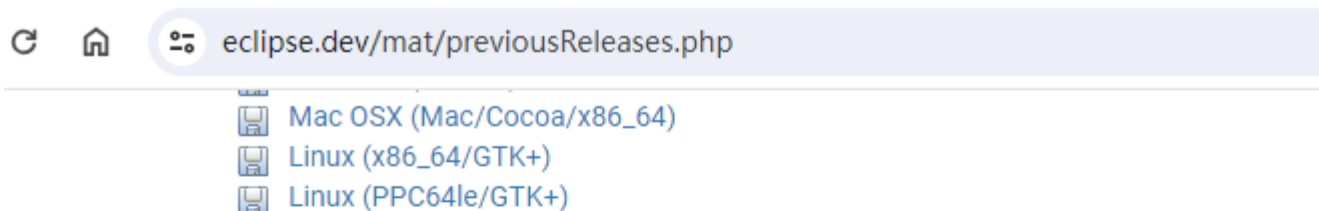
```
hrxj-bigbrain-service_all.log    java_pid21930.hprof  
hrxj-bigbrain-service_err.log   java_pid29467.hprof  
hrxj-bigbrain-service_info.log  java_pid30901.hprof  
hrxj-bigbrain-service_trace.log  
hrxj-bigbrain-service_warn.log
```

然后把最新生成的文件下载下来

java_pid29467.hprof	2024/1/18 17:43	HPROF 文件	878,122 KB
---------------------	-----------------	----------	------------

3.3 MAT工具分析hrpof文件

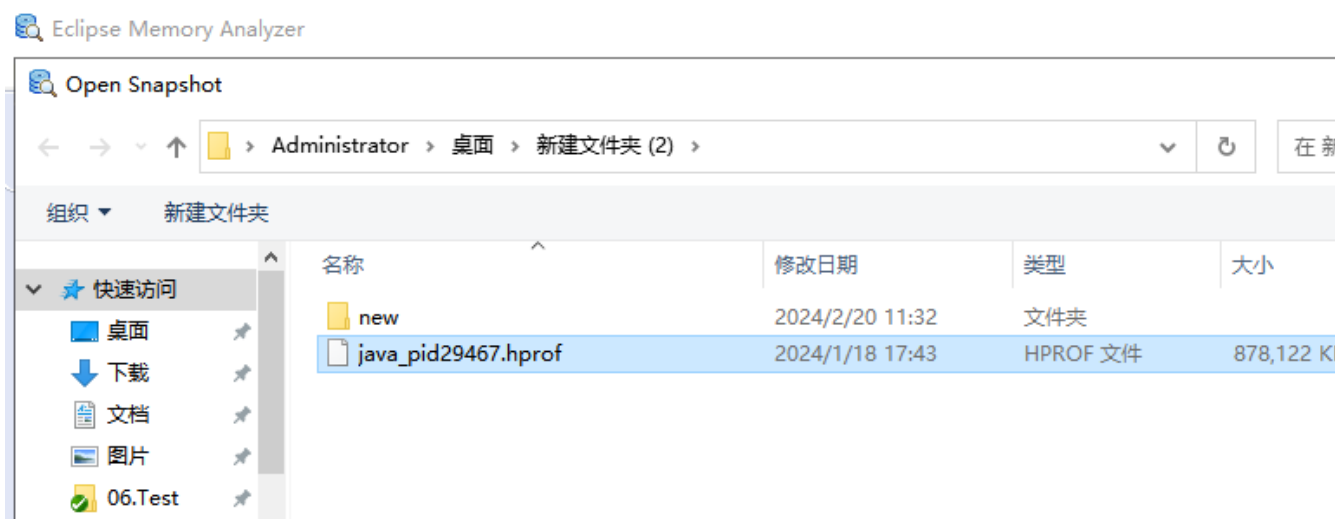
下载MAT工具：<https://eclipse.dev/mat/previousReleases.php>



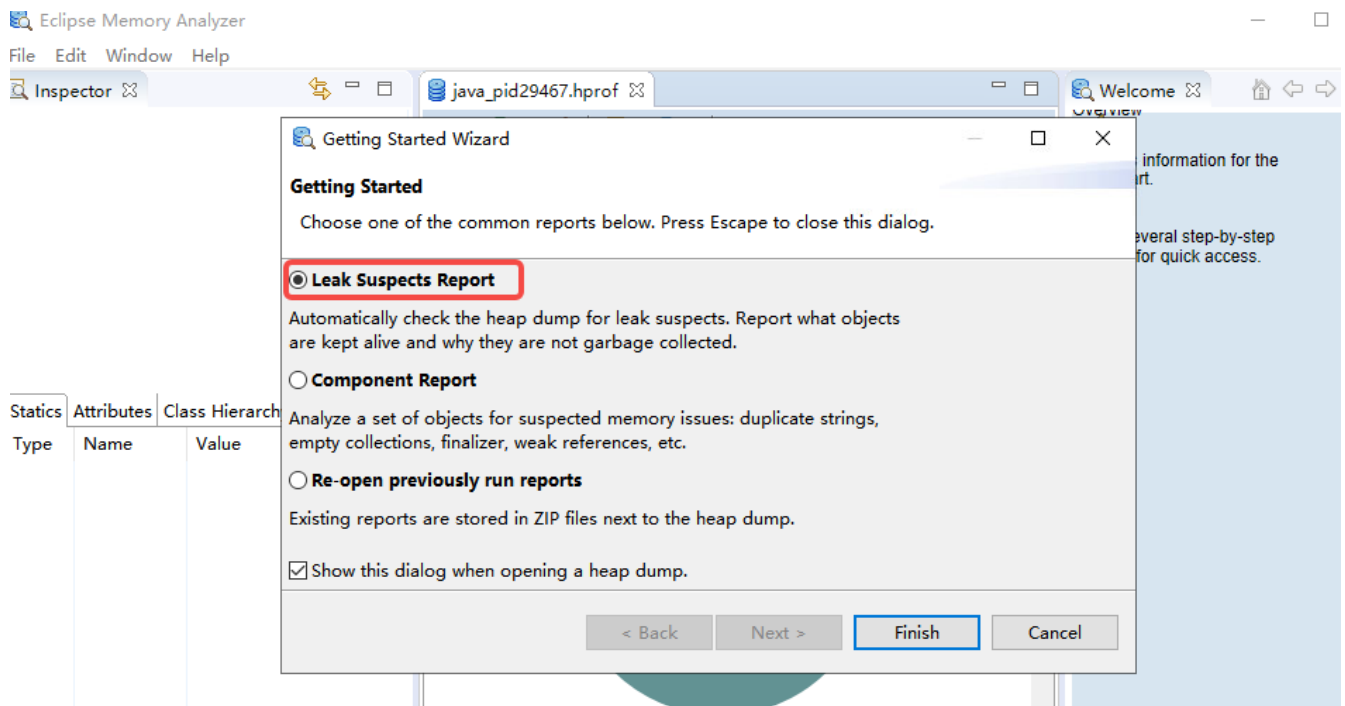
Memory Analyzer 1.10.0 Release

- **Version:** 1.10.0.20200225 | **Date:** 18 March 2020 | **Type:** Released
 - **Update Site:** <https://download.eclipse.org/mat/1.10.0/update-site/>
 - **Archived Update Site:** [MemoryAnalyzer-1.10.0.202002252112.zip](#)
 - **Stand-alone Eclipse RCP Applications**
 - Windows (x86)
 - Windows (x86_64)**
 - Mac OSX (Mac/Cocoa/x86_64)
 - Linux (x86/GTK+)
 - Linux (x86_64/GTK+)
 - Linux (PPC64/GTK+)
 - Linux (PPC64le/GTK+)

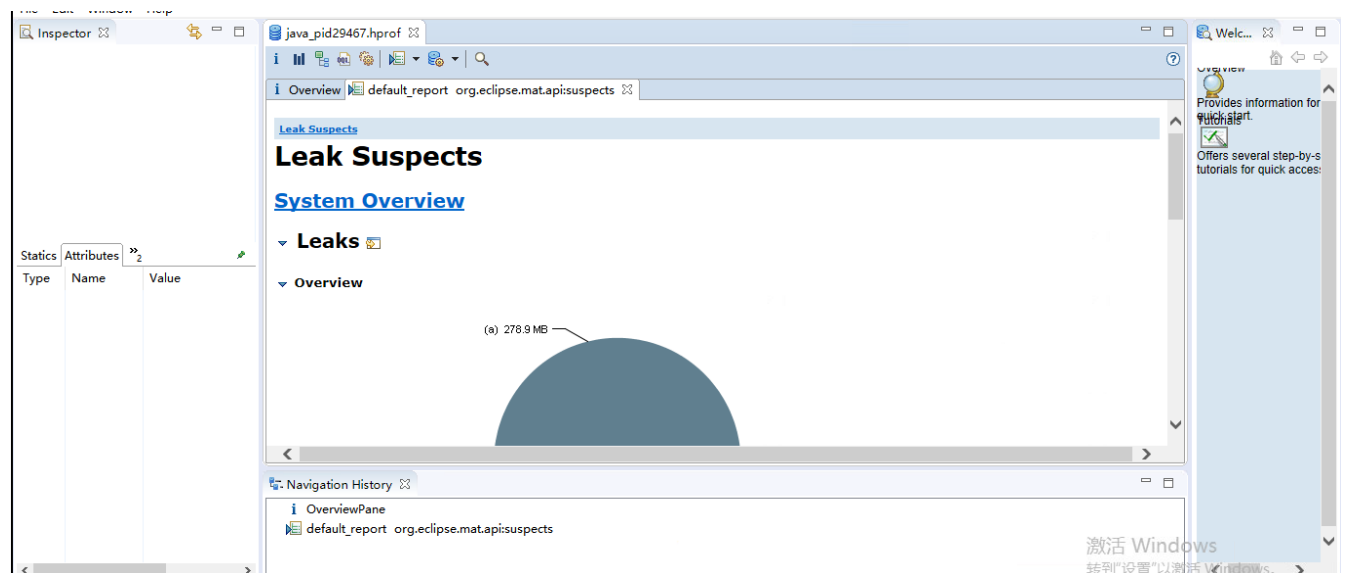
下载后安装好，然后使用MAT工具，打开hprof文件



打开后，会弹出一个分析选项，直接选择第一个，第一个功能会自动分析最有可能存在内存溢出的位置，一目了然



打开后效果如下：



往下翻会看到几个溢出的原因：

Total: 483.8 MB

▼ Problem Suspect 1

One instance of "**org.pentaho.di.trans.steps.mergejoin.MergeJoinData**" loaded by "**sun.misc.Launcher\$AppClassLoader @ 0xf1ccbfa8**" occupies **292,455,184 (57.64%)** bytes. The memory is accumulated in one instance of "**java.lang.Object[]**" loaded by "<system class loader>".

Keywords

org.pentaho.di.trans.steps.mergejoin.MergeJoinData
java.lang.Object[]
sun.misc.Launcher\$AppClassLoader @ 0xf1ccbfa8

[Details »](#)

▼ Problem Suspect 2




One instance of "**org.pentaho.di.trans.steps.mergejoin.MergeJoinData**" loaded by "**sun.misc.Launcher\$AppClassLoader @ 0xf1ccbfa8**" occupies **174,457,768 (34.39%)** bytes. The memory is accumulated in one instance of "**java.lang.Object[]**" loaded by "<system class loader>".

Keywords

org.pentaho.di.trans.steps.mergejoin.MergeJoinData
java.lang.Object[]
sun.misc.Launcher\$AppClassLoader @ 0xf1ccbfa8

[Details »](#)

然后逐一展开分析，先点击第一个的Details


 Overview  default_report org.eclipse.mat.api:suspects 

[Leak Suspects](#) » [Leaks](#) » [Problem Suspect 1](#)

▼ **Description**

One instance of "**org.pentaho.di.trans.steps.mergejoin.MergeJoinData**" loaded by "**sun.misc.Launcher\$AppClassLoader @ 0xf1ccbfa8**" occupies **292,455,184 (57.64%)** bytes. The memory is accumulated in one instance of "**java.lang.Object[]**" loaded by "<system class loader>".

Keywords
org.pentaho.di.trans.steps.mergejoin.MergeJoinData
java.lang.Object[]
sun.misc.Launcher\$AppClassLoader @ 0xf1ccbfa8

▼ **Shortest Paths To the Accumulation Point** 

如果英文好可以翻译一下，大概意思是：一个实例对象MergeJoinData被ClassLoader加载时，分配了292455184字节大小的内存空间。

再往下翻可以看到更具体的蓝色标注：**蓝色标注就是可能存在内存溢出的位置**

▼ Shortest Paths To the Accumulation Point 📄

Class Name	Shallow Heap	Retained Heap
java.lang.Object[f12154871] @ 0xea9ac170	4,861,968	292,441,440
elementData java.util.ArrayList @ 0xf8948490	24	292,441,464
ones org.pentaho.di.trans.steps.mergejoin.MergeJoinData @ 0xf411a908	80	292,455,184
data, stepDataInterface org.pentaho.di.trans.steps.mergejoin.MergeJoin @ 0xf0954160	336	5,536
<Java Local> java.lang.Thread @ 0xf0247298 DATA_SOCIAL_APPLY_AFTER_GRADE - Merge Join 7 Thread	120	1,720
data org.pentaho.di.trans.step.RunThread @ 0xf18e9c28 »	32	32
data org.pentaho.di.trans.step.StepMetaDataCombi @ 0xf2c7b028 »	40	40
Σ Total: 3 entries		

那么看到这里，我们就能知道是哪个包，哪个类，哪个线程可能存在内存溢出了

包：org.pentaho.di.trans.steps.mergejoin

类：MergeJoin类

线程：java.lang.Thread @0xf0247298

DATA_SOCIAL_APPLY_AFTER_GRADE - Merge Join 7 Thread

对于普通测试来讲，能定位到这里就可以告一段落了。我们可以把这些信息发给开发，让他们再继续进行定位分析，但是我们不能止步于此，我们可以直接找到根本原因。

接下来就要结合项目进一步来分析问题产生的根本原因是什么了

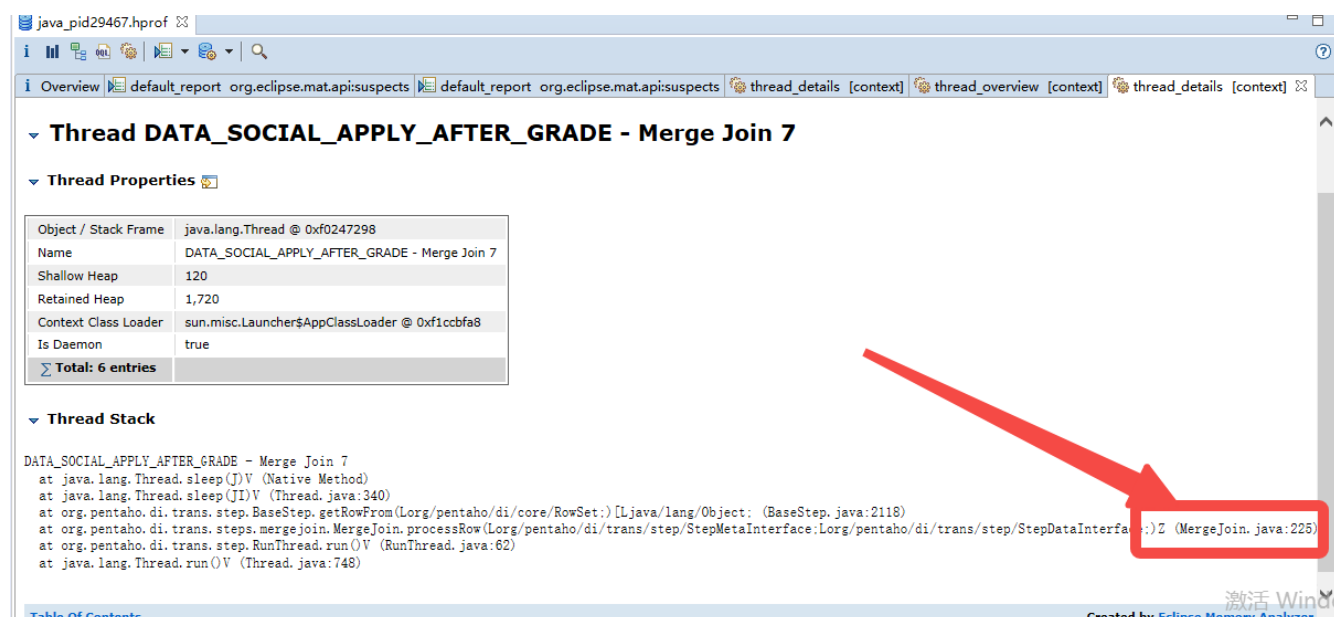
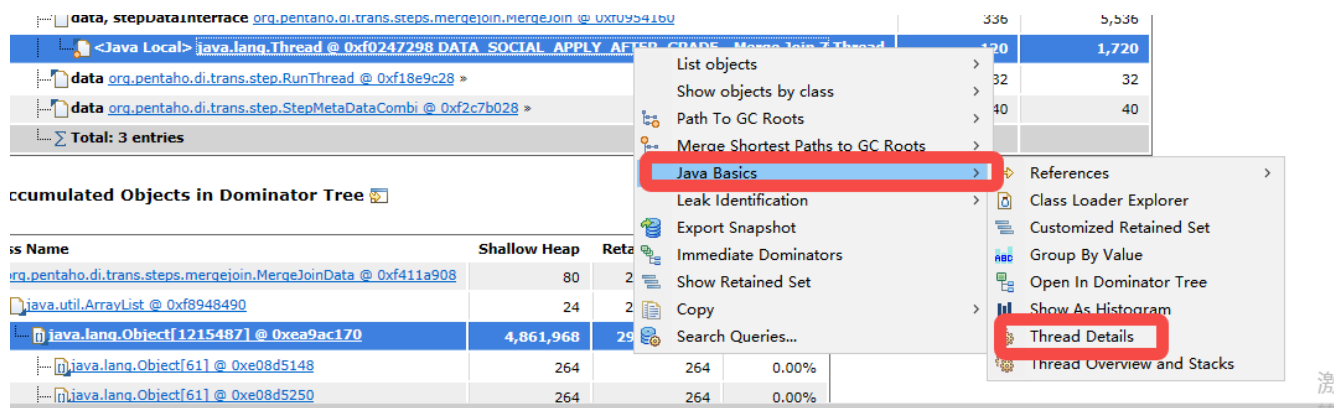
3.4 结合项目进一步分析（高能）

根据包名(org.pentaho.di.trans.steps.mergejoin)，我们在网上百度一搜就知道，是kettle的包，所以可以确认是"kettle"出了问题。

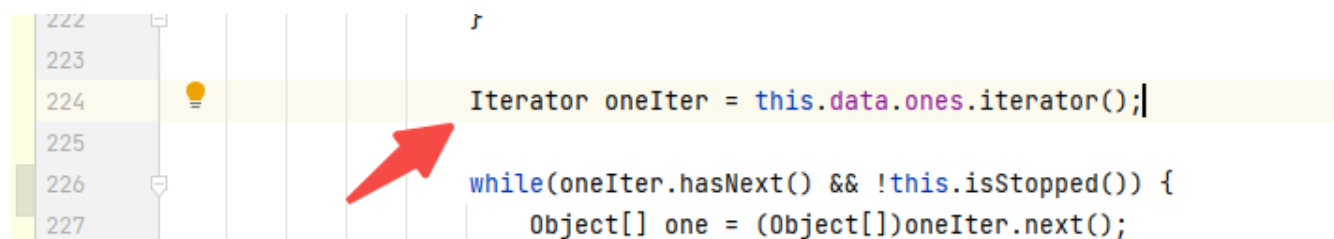
那么kettle是哪个环节出了问题呢？聪明的小伙伴这个时候可能已经能够基于之前的信息联想到答案了，但是我们先不着急，继续分析。

根据类名可以知道出问题的类就是MergeJoin类

然后可以在MAT中进一步查看详细信息知道类中的哪一行代码出了问题



然后根据代码行号查看源码，就能知道具体报错代码是什么



根据这个代码，大概能猜到肯定是this.data.ones这个列表在转化为迭代器时，发生了堆溢出。如果继续分析，那么就要逆推this.data.ones是怎么添加那么多数据的。

这个过程就非常繁琐了，所以我们这里直接祭出大招：结合日志+kettle脚本分析，请看下面的日志

```
| 2024-02-20 10:48:33.092 | DEBUG | localhost | DubboServerHandler-10.4.190.69:20891-thread-107 | o.s.jdbc.datasource.DataSourceUtils > 327 | | Returning JDBC Connection to DataSource
| 2024-02-20 10:48:33.092 | INFO | localhost | DubboServerHandler-10.4.190.69:20891-thread-107 | c.i.bigbrain.bs.FinsIntegrateSvrBs > 196 | | 实时授信政务数据整合开始:applyId=2309209659900017,member=0,idCard=432502198906120940,bankCode=96599 |
| 2024-02-20 10:48:33.092 | INFO | localhost | DubboServerHandler-10.4.190.69:20891-thread-107 | c.i.bigbrain.bs.FinsIntegrateSvrBs > 230 | | 脚本入参: {filepath=/home/bigbrain/batch/kettle/ETLitem, SEQ_NO=2309209659900017104630, APPLY_ID=2309209659900017, CUST_GROUP='gtqsh', AREA_CODE=96599, ZW_SPOUSE_JOB='cxjm', TYPE=0, PRIPIID=, ID_CARD='432502198906120940'} |
```

日志中告诉我们，超脑是在执行"实时授信政务数据整合开始"卡住后溢出的。

那么直接在源码中搜关键字："实时授信政务数据整合开始"，就能找到对应的类和代码

```
/**/
private void applyDataInterGov(String applyId,Integer member,String idCard,String bankCode,String queryTp,String
    logger.info("实时授信政务数据整合开始:applyId={},member={},idCard={},bankCode={}", applyId,member,idCard,bankCode)
    long start = new Date().getTime();
```

然后顺着代码往下读，就能知道什么时候执行的kettle脚本

```
if (!KettleUtil.runTransfer(paramMap, scriptPath)) {
```

然后分析执行过程就能知道，kettle脚本先启动了job

```
job.start();
```

然后找到执行入口start

```
startpoint = this.jobMeta.findJobEntry( name: "START", nr: 0, searchHiddenToo: false);
```

接着根据入口去执行Job

```
if (!startpoint.isStart()) {
    res = this.execute(nr: 0, res, startpoint, (JobEntryCopy)null, BaseMessages.getString(PKG, key: "Job.Reason.Started", new String[0]),
    jerEnd = new JobEntryResult(res, startpoint.getEntry().getLogChannel().getLogChannelId(), BaseMessages.getString(PKG, key: "Job.Com
} else {
    boolean isFirst = true;

    JobEntrySpecial jes;
    for(jes = (JobEntrySpecial)startpoint.getEntry(); (jes.isRepeat() || isFirst) && !this.isStopped(); res = this.execute(nr: 0, (Resu
        isFirst = false;
    }

    jerEnd = new JobEntryResult(res, jes.getLogChannelId(), BaseMessages.getString(PKG, key: "Job.Comment.JobFinished", new String[0]),
}
```

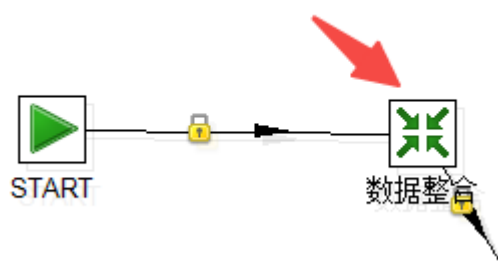
执行后，又找到下一个入口

```
int nrNext = this.jobMeta.findNrNextJobEntries(jobEntryCopy);
```

接着继续执行下一个入口：

```
try {  
    res = this.execute( nr: nr + 1, newResult, nextEntry, jobEntryCopy, nextComment);  
} catch (Throwable var24) {
```

而下一个入口在kettle脚本中是"数据整合"



数据整合的ktr名是：

● 转换文件名:

\$(etl_pat /DATA_SOCIAL_APPLY_AFTER_GRADE.ktr

在代码中，对应的代码是：

```
newResult = cloneJei.execute(prevResult, nr);
```

数据整合，在kettle中，与transformation——对应，所以
cloneJei.execute(prevResult,nr) 调用trans.class的execute代码：

```
public void execute(String[] arguments) throws KettleException {  
    this.prepareExecution(arguments);  
    this.startThreads();  
}
```

在这个代码中，this.startThreads(); 调用后，会设置执行的线程名为当前ktr的文件名：

```
thread.setName(this.getName() + " - " + combi.stepname);
```

所以执行线程的名字就是数据整合的ktr名，也就是
DATA_SOCIAL_APPLY_AFTER_GRADE，

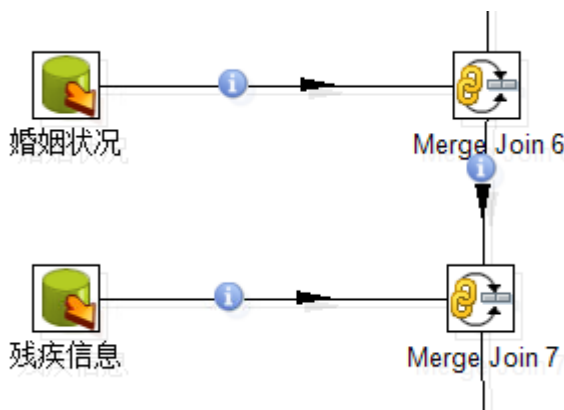
这个ktr脚本内容比较多，我们需要进一步定位到具体是哪个位置发生的
溢出，

可以结合MAT的提示来看：



注意看上图，线程名有一个Merge Join7

对应的就是ktr中的Merge Join7



所以他是执行Merge Join7时，发生的内存溢出

为什么可以认为Merge Join7就是准确位置？主要是这行代码告诉我们的：

```
thread.setName(this.getName() + " - " + combi.stepname)
```

this.getName会获取到DATA_SOCIAL_APPLY_AFTER_GRADE

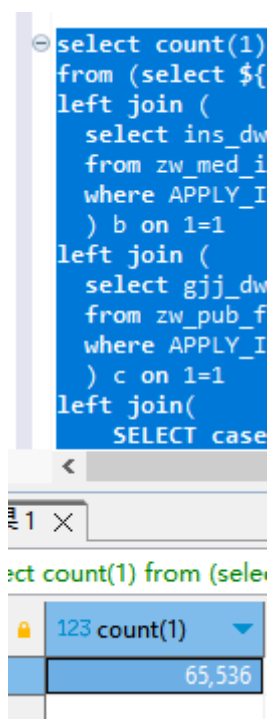
combi.stepname会获取到Merge Join7

分析到这里后，我们就可以分析Merge Join7干了啥，导致了内存溢出

3.5 分析kettle的SQL

使用data-integration工具打开DATA_SOCIAL_APPLY_AFTER_GRADE.ktr

然后用DBeaver逐个执行里面的SQL，可以发现，执行到公积金的kettle脚本时，数据量异常增大：



异常原因：这个SQL是根据身份证号码、APPLY_ID去筛选的，不可能筛选出这么多条

而这里大量的merge操作又需要考虑笛卡尔积的问题。

如果只有1条数据，那么自然不存在笛卡尔积，但是如果有几万条，那么就不正常了

3.6 DBeaver验证

我们继续按照之前的入参去执行SQL：

```
2024-02-20 10:46:33.092 | DEBUG | localhost | DubboServerHandler-10.4.190.69:20891-thread-107 | o.s.jdbc.datasource.DataSourceUtils > 327 | | Returning JDBC Connection to DataSource
2024-02-20 10:46:33.092 | INFO | localhost | DubboServerHandler-10.4.190.69:20891-thread-107 | c.i.bigbrain.bs.FinsIntegrateSvrBs > 196 | | 实时授信或务数据整合开始:applyId=2309209659900017,member=0,idCard=432502198906120040,bankCode=96599 |
2024-02-20 10:46:33.092 | INFO | localhost | DubboServerHandler-10.4.190.69:20891-thread-107 | c.i.bigbrain.bs.FinsIntegrateSvrBs > 230 | | 脚本入参: {filepath=/home/bigbrain/batch/ettle/ETLitem, SEQ_NO=2309209659900017104630, APPLY_ID=2309209659900017, CUST_GROUP='gtqsh', AREA_CODE=96599, ZW_SPOUSE_108='cxjm', TYPE=0, PRIPIID=, ID_CARD='432502198906120040'} |
```

只执行这一个子SQL即可：

```
FROM
(select 1 from dual) t
left join
(select GJJ_JNDWMC, GJJ_JZNY from zw_pub_fund where APPLY_ID=${SEQ_NO} AND USER_IDCARD=${ID_CARD} AND TYPE=${TYPE} AND AREA_CODE=${AREA_CODE}) a
on 1=1
left join
```

可以看到，只有10条数据

```
select GJJ_JNDWMC, GJJ_JZNY from zw_pub_fund where APPLY_ID =2401229659900016103430 and USER_IDCARD=430721198902084351;
```

pub_fund 1

select GJJ_JNDWMC, GJJ_JZNY from zw_ 输入一个 SQL 表达式来过滤结果 (使用 Ctrl+Space)

ABC GJJ_JNDWMC	ABC GJJ_JZNY
湖南恒安纸业债券有限公司	201906
湖南恒安纸业债券有限公司	201906
湖南恒安纸业债券有限公司	201906
湖南恒安纸业债券有限公司	201906
湖南恒安纸业债券有限公司	201906
湖南恒安纸业债券有限公司	201906
湖南恒安纸业债券有限公司	201906
湖南恒安纸业债券有限公司	201906
湖南恒安纸业债券有限公司	201906
湖南恒安纸业债券有限公司	201906

数值查看器

湖南恒安纸业债券有限公司

那么这里会查出10条数据呢？

我们进一步分析，把要查的列用"*"号代替发现，要查的是尾号apply_id为3430的数据，实际出来了5030、0131、0430...等数据，所以问题是apply_id的查询条件越界了

```
select * from zw_pub_fund where APPLY_ID =2401229659900016103430 and US
```

ABG APPLY ID	ABG USER_IDCARD	ABG TYPE	ABG AREA_CODE
2401229659900016095030	430721198902084351	0	96599
2401229659900016100131	430721198902084351	0	96599
2401229659900016100430	430721198902084351	0	96599
2401229659900016102731	430721198902084351	0	96599
2401229659900016102930	430721198902084351	0	96599
2401229659900016103030	430721198902084351	0	96599
2401229659900016103130	430721198902084351	0	96599
2401229659900016103230	430721198902084351	0	96599
2401229659900016103330	430721198902084351	0	96599
2401229659900016103430	430721198902084351	0	96599

加上双引号再查

```
select * from zw_pub_fund where APPLY_ID = '2401229659900016103430' and USER_IDCARD=430721198902084351;
```

ABG APPLY_ID	ABG USER_IDCARD	ABG TYPE	ABG AREA_CODE	ABC USER_NAME	ABI
2401229659900016103430	430721198902084351	0	96599	毛耀萱	12

数值查看器 ×
24012296599000

这下就只有1条了。

3.7 结论

因为查询贷中的政务数据时，进行数据整合传入的入参apply_id传入的是纯数字类型，导致内部执行查询时，出现查询条件失效的问题，进而导致执行查询的结果集巨大，达到了内存溢出的条件，最终导致内存溢出

4 总结与分析

本案例中，导致内存溢出的根本原因是SQL执行的结果集太大。

但是导致SQL执行结果集太大的原因是入参APPLY_ID类型不正确

所以修复这个问题，只需要修改APPLY_ID的类型就可以了。

大家可以思考一下，为什么不修改kettle脚本？

实战经验提炼

配置获取hprof文件

在java启动命令中增加如下两个配置

```
-XX:+HeapDumpOnOutOfMemoryError  
-XX:HeapDumpPath=/home/bigbrain/logs
```

如：

```
java -XX:+HeapDumpOnOutOfMemoryError -  
XX:HeapDumpPath=/home/bigbrain/logs -jar xxx.jar
```

其中/home/bigbrain/logs是可以修改的hprof文件的保存路径

使用MAT分析hprof文件

下载MAT工具: <https://eclipse.dev/mat/previousReleases.php>

注意版本: 1.10

打开方式: 左上角File-> Open HeapDump.. -> 选择hprof文件打开 (提前从服务器上下载) -> 查看结果

然后在Overview结果页, 可以点击"Leak Suspects"来查看可能内存溢出的报告

然后在Leak Suspects报告中查看对应Problem 的Detail详情

这样就能知道哪个代码、哪个线程导致的内存溢出

结合项目分析 (重点)

- 根据Leak Suspects报告中的包名、类名定位到源码
- 搜索对应的代码
- 分析代码
 - 静态阅读代码分析
 - 分析问题代码所在行与其相关的上下文
 - 分析问题代码所在方法与该方法的调用关系
 - 分析问题代码所在类, 与该类的继承、实例、调用关系

静态分析经常遇到困难, 并不一定能100%解决问题

- Debug分析代码

由于问题代码所在位置一定会被执行, 所以可以通过打断点的方式, 来进行Debug分析, 然后逐步根据Debug时发现的执行顺序来增加、减少断点从而帮助理解程序运行, 最终定位到问题。

- 找开发分析

使用DBeaver调试kettle中的SQL

这个没啥可以说的，就是用kettle的客户端工具Spoon打开kettle脚本，获取到SQL，然后把SQL拷贝到DBeaver中运行。

5 练习

1.本文用到了哪些技术？

2.DBeaver工具执行SQL时，哪种参数化的方式可以手动填参数值：

A \$参数名

B \${参数名}

C #参数名#

D \$\$参数名

3.假如内存溢出的服务器是fins-apply，那么应该把生成内存溢出hprof文件的配置放在哪里？

4.有其他分析方法吗，举例说明一下

6 重点难点

需要掌握的部分知识：

1. 掌握并理解在hrxj-bigbrain超脑应用中，增加输出内存溢出日志文件配置的方法和作用
2. 掌握使用Mat工具分析内存溢出的方法
3. 掌握分析kettle脚本的方法

难点：

1. 根据Mat工具定位的代码位置去分析代码