什么时候会用到MAT?为什么需要分析GC?

1）OutOfMemoryError的时候，触发full gc，但空间却回收不了，引发内存泄露

2）Java服务器系统异常，比如load飙高，io异常，或者线程死锁等，都可能通过分析堆中的内存对象来定位原因

JVM 相关JDK命令操作

查看 Java 进程信息

1.查看当前机器上所有运行的java进程名称与pid(进程编号)

JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程

```
jps -l
//或
ps -ef | grep java
```

2.显示指定的jvm进程所有的属性设置和配置参数

```
jinfo pid
```

生成heap dump文件

1.方法一：使用jdk的jmap命令

```
jmap -dump:format=b,file=dump.heap  pid
//如果只dump heap中的存活对象，则加上选项-live，如下：
jmap  -dump:live,format=b,file=/path/dump.heap pid
```

2.方法二：让JVM在遇到OutOfMemoryError时生成Dump文件

-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/path

使用jstat监视虚拟机运行时状态

jstat(JVM statistics Monitoring)是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。

实时监控old区的大小，快要fullgc时执行heap

```
jstat -gc pid
```

常用参数说明：

```
> jstat -options
-class 显示ClassLoad的相关信息；
-gc 显示和gc相关的堆信息；
-gcnew 显示新生代信息；
-gcnewcapacity 显示新生代大小和使用情况；
-gcold 显示老年代和永久代的信息；
-gcoldcapacity 显示老年代的大小；
-gcutil     显示垃圾收集信息；
```

例子：

```
PS C:\> jstat -gc 6368
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC
5120.0 5120.0  0.0    0.0   32768.0  29858.7   86016.0      0.0     4480.0 77
PS C:\>
```

重点关注oc和ou

OC：老年代大小

OU：老年代使用大小

MAT介绍

MAT 全称 Eclipse Memory Analysis Tools 是一个分析 Java堆数据的专业工具，可以计算出内存中对象的实例数量、占用空间大小、引用关系等，看看是谁阻止了垃圾收集器的回收工作，从而定位内存泄漏的原因。

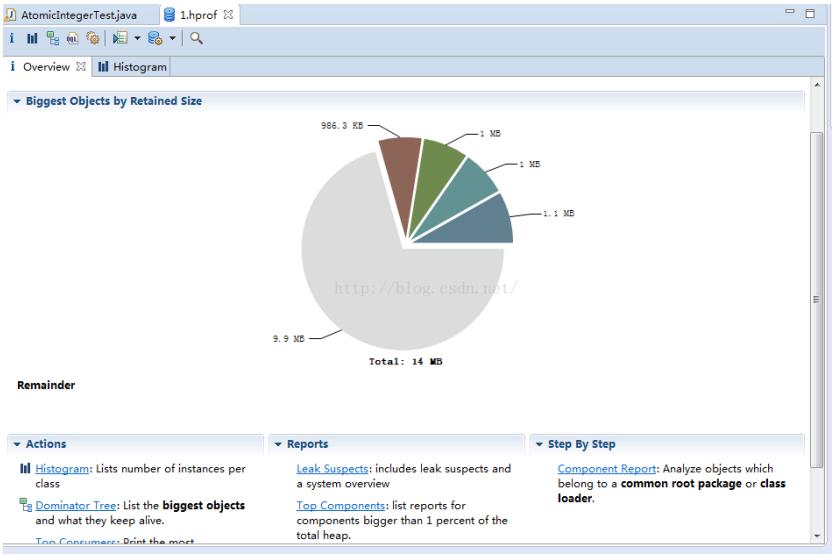
MAT是一个基于Eclipse 的内存分析工具，是一个基础插件。

下载&安装

MAT工具的下载地址为： http://www.eclipse.org/mat/downloads.php

MAT插件的下载地址为： http://download.eclipse.org/mat/1.3/update-site

Eclipse 安装方法：http://jingyan.baidu.com/article/cb5d61053562ed005c2fe022.html



MAC打开报错设置

1.open app即报错

解决方法一：

把下载解压后的mat.app 文件放到/Applications目录下

解决方法二：

在在配置文件中添加如下配置；配置文件路径：/Applications/mat.app/Contents/Eclipse/MemoryAnalyzer.ini

```
-vm
/Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home
```

2.打开dump.heap文件报错

解决方法一：

由于一般fullgc产生的dump.head文件非常大，导致mat工具打不开，在MemoryAnalyzer.ini配置文件，添加如下配置即可：

```
-Xms8192m
-Xmx8192m
```

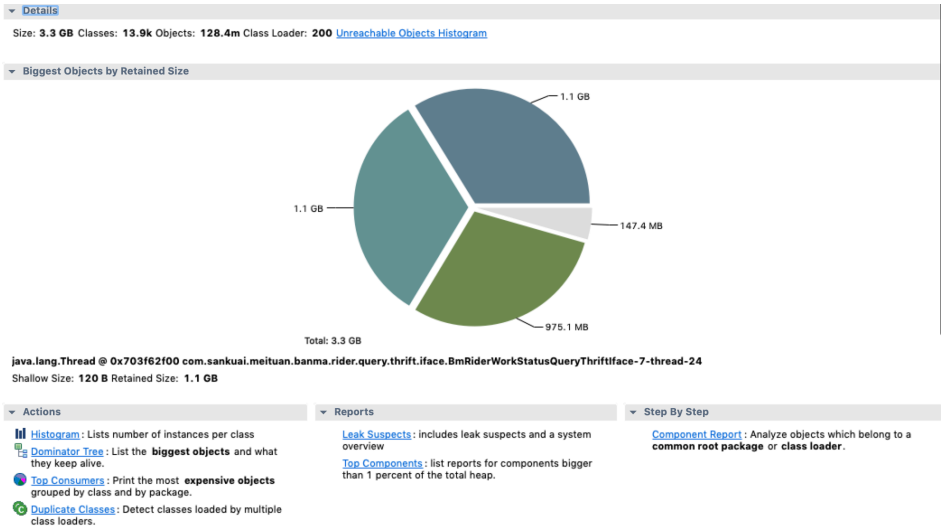
解决方法二：

修改启动参数：MemoryAnalyzer.exe -vmargs -Xmx4g

MAT使用

overview(概观)

用MAT打开一个heap文件后一般会进入如下的overview界面，overview界面会以饼图的方式显示当前消耗内存最多的几类对象。但是，除非你的程序内存泄漏特别明显或者你正好在生成heap文件之前复现了程序的内存泄漏场景，你才可能通过这个界面猜到程序出问题的地方。



从上图可以看到它提供的一些功能，后面会重点说一下：

1） Histogram可以查看内存中的对象，对象的个数以及大小。

2）Dominator Tree可以列出线程，以及线程下面的那些对象占用的空间。

3）Top consumers通过图形列出最大的object。

4）Leak Suspects通过MA自动分析泄漏的原因。

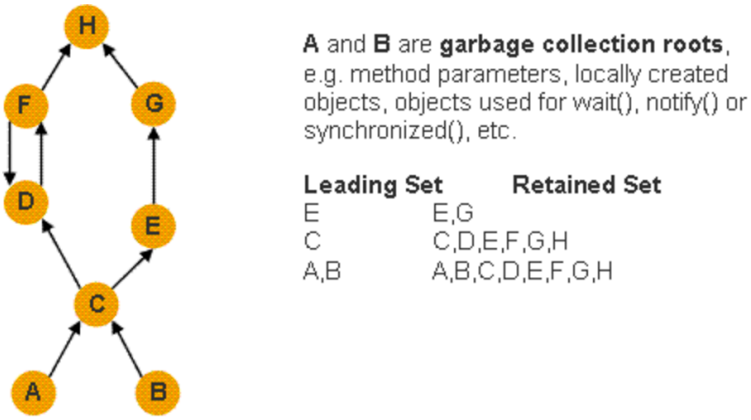
1.Dorminator Tree（支配树）

支配树可以直观地反映一个对象的retained heap，这里我们首先要了解下面的两个概念：

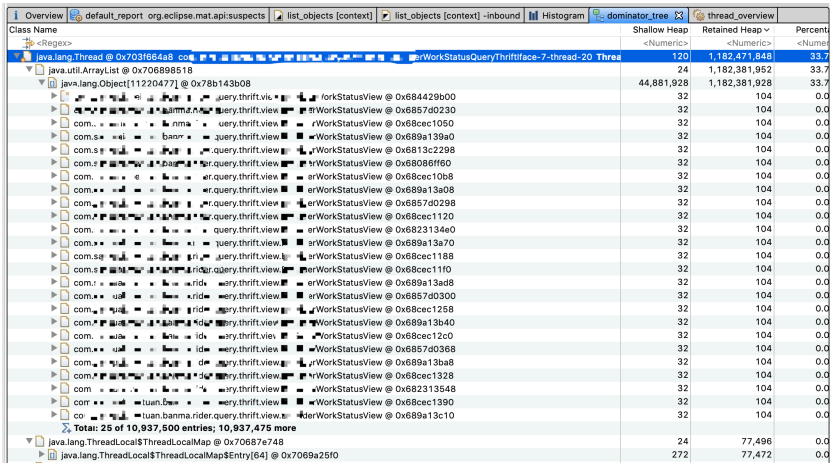
shallow heap：指的是某一个对象所占内存大小。

retained heap：指的是一个对象的retained set所包含对象所占内存的总大小。

retained set指的是这个对象本身和他持有引用的对象和这些对象的retained set所占内存大小的总和，本质是反映一个对象如果被回收，有多少内存可以同时得到释放，用官方的一张图来解释如下：



使用dorminator tree分析的时候，忽略jdk相关的类引用，重点找跟业务属性相关的类和对象，以及对象的引用函数等。

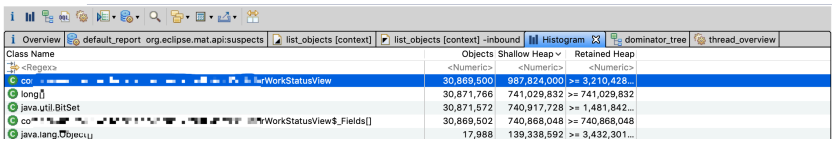


分析：

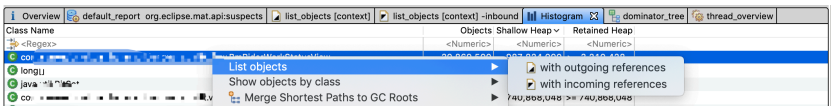
首先就可以看到上图对象（蓝色选中的）的shallow heap和retained heap是非常不成比例的，这说明这个对象持有了大量对象的引用，正常情况下不会出现此现象。于是我们根据retained heap的大小逐步对该对象的支配树进行展开可以发现，ArrayList下面持有大量的...StatusView对象的引用，数量高达1100多万个（非常恐怖），而在正常情况下，不可能如此庞大，因此，从这里就可以断定此对象使用不当出现了内存泄漏。

2.Histogram

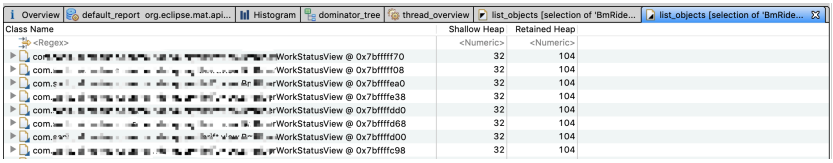
以直方图的方式来显示当前内存使用情况可能更加适合较为复杂的内存泄漏分析，它默认直接显示当前内存中各种类型对象的数量及这些对象的shallow heap和retained heap。结合MAT提供的不同显示方式，往往能够直接定位问题，还是以上面的代码为例，当我们切换到histogram视图下时，可以看到：



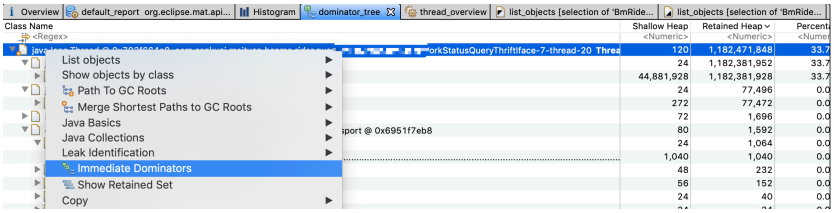
根据retained heap进行排序后可见，上述对象存在3000多万个对象，消耗的内存也比较靠前，因此这其中必定存在问题。那么又如何来确定到底是谁在持有这些对象而导致内存没有被回收呢？有两种方式，第一就是右键选择List Objects -> with incoming reference，这可以列出所有持有这些对象的引用路径，如图



从中可以看出几乎所有的...StatusView对象都被ArrayList持有，因此基本可以断定出现一次性放入List的对象过多，有无法及时回收的问题；



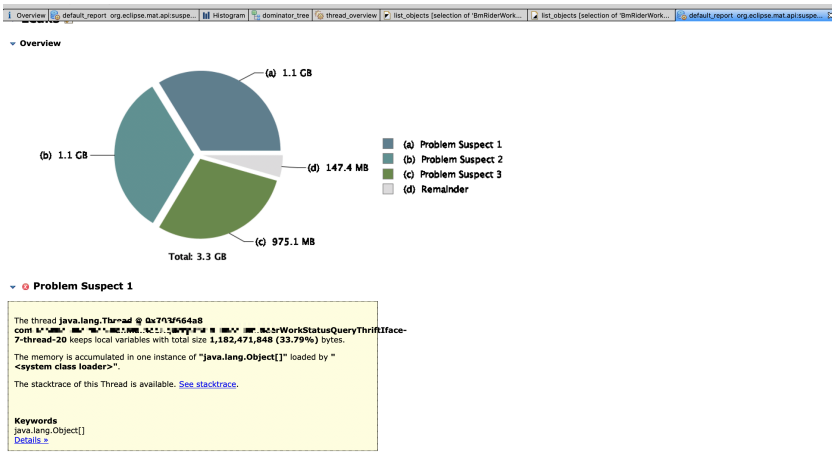
第二种方式就更直接粗暴，但是非常有效果，那就是借助我们刚才说的dorminator tree，右键选择Immediate Dorminator后就可以看到如下结果：



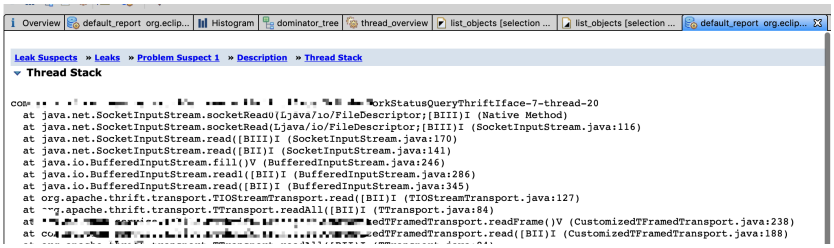
3.Leak Suspects（自动分析结果）

傻瓜式分析，MAT自动分析大盘显示problem结果；如果full gc特别明显和验证的话，这个工具效率会很高，因为能自动分析出问

题，以及可以查看问题的堆栈详情。



每个problem中都可以查看异常堆栈，点开see stacktrace可以看到具体堆栈信息，定位到错误代码的具体行。



好文推荐

https://blog.csdn.net/ZYC88888/article/details/79975194