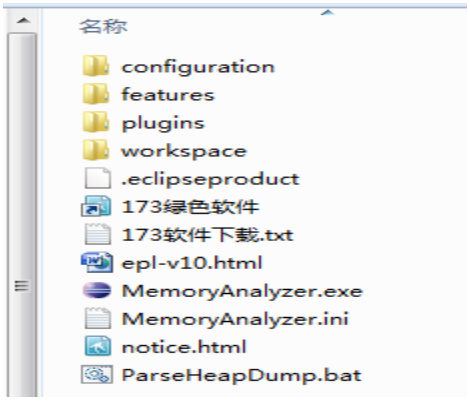
对于大型 JAVA 应用程序来说，再严格精细的测试也很难堵住所有的漏洞，即便我们在测试阶段进行了大量卓有成效的工作，很多问题还是会在生产环境下暴露出来，并且很难在测试环境中进行重现。JVM 能够记录下问题发生时系统的部分运行状态，并将其存储在堆转储(Heap Dump)文件中，从而为我们分析和诊断问题提供了重要的依据。

通常内存泄露分析被认为是一件很有难度的工作，一般由团队中的资深人士进行。不过，今天我们要介绍的 MAT（Eclipse Memory Analyzer）被认为是一个“傻瓜式“的堆转储文件分析工具，你只需要轻轻点击一下鼠标就可以生成一个专业的分析报告。和其他内存泄露分析工具相比，MAT 的使用非常容易，基本可以实现一键到位，即使是新手也能够很快上手使用。

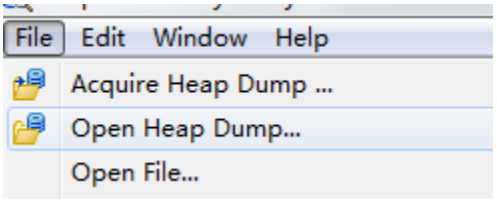
MAT 内存泄露分析工具使用步骤：

1、下载(MemoryAnalyzer-1.0.1.20100809-win32.win32.x86.zip)并解压到本地。注意：下载的 MAT 版本要和你的电脑位数兼容。

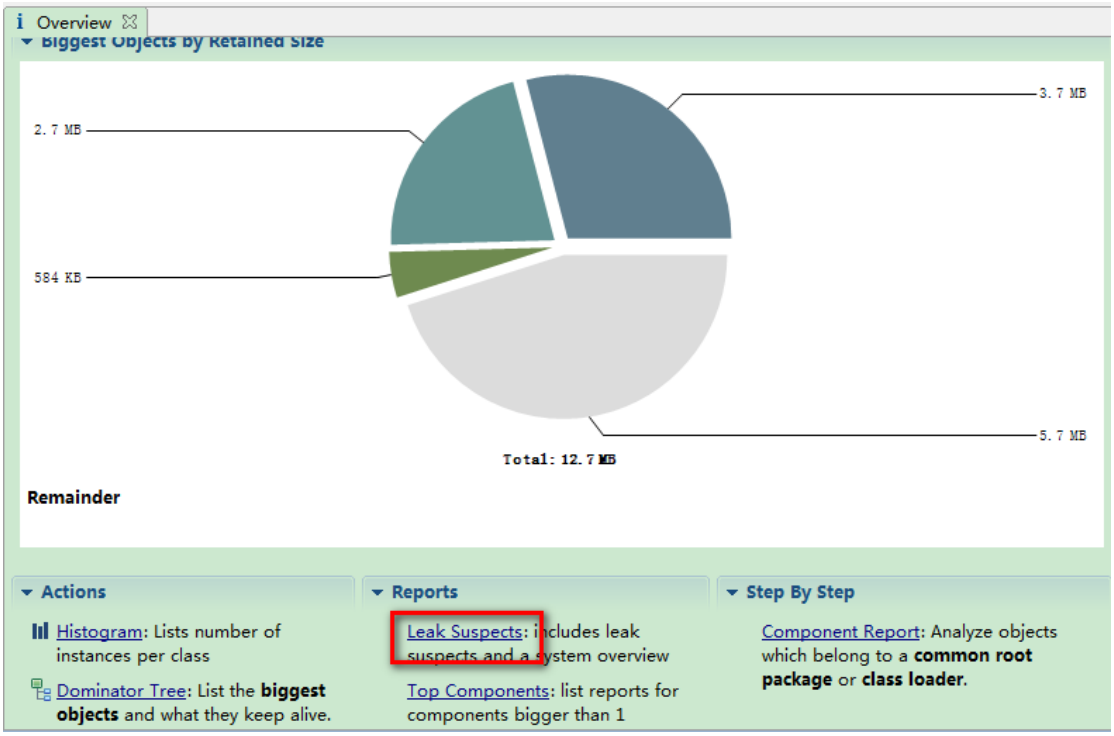


2、MAT(Memory Analyzer Tool) 是基于 heap dumps 来进行分析的，所以首先必须通过一定的手段得到 JAVA 堆的 DUMP 文件，该文件的后缀是.hprof，JDK 自带的 JConsole 或者 visualVM 都是不错的工具。

3、打开 MAT 工具，并导入.hprof 文件



4、生成以下报表。



通过上面的图，我们对内存占用情况有了一个总体的了解。

5、我们来看看生成的报告都包括什么内容，能不能帮我们找到问题所在

MAT 工具内存泄露分析三部曲：

通常我们都会采用下面的“三步曲”来分析内存泄露问题：

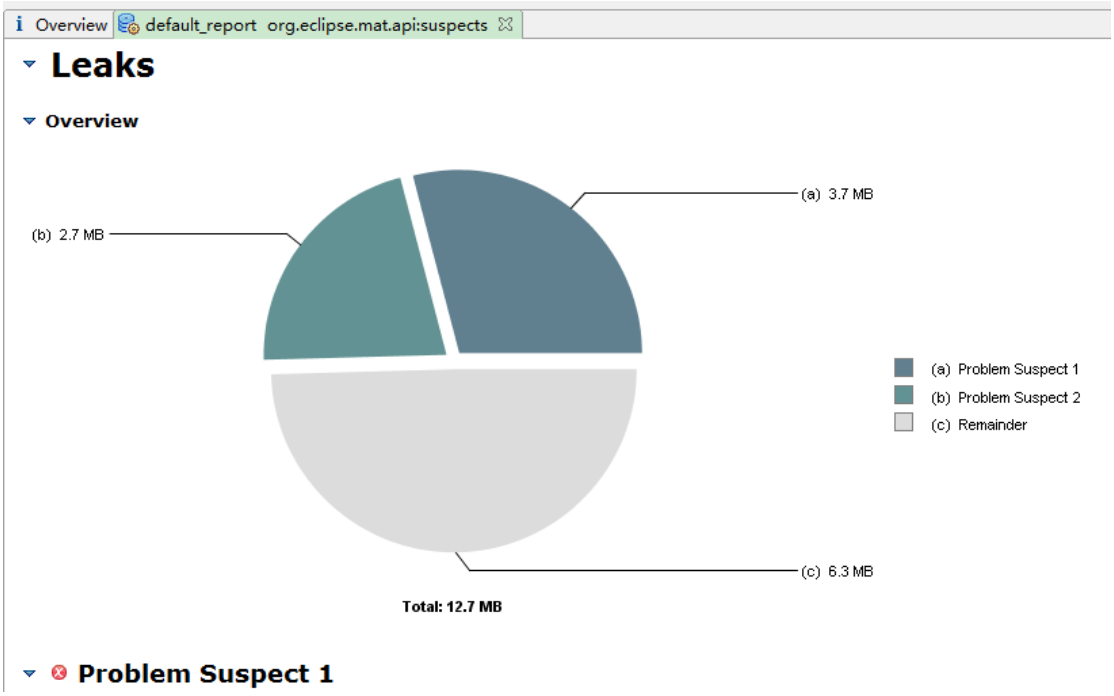
第一步：对问题发生时刻的系统内存状态获取一个整体印象。

第二步：找到最有可能导致内存泄露的元凶，通常也就是消耗内存最多的对象。

第三步：查看这个内存消耗大户的具体情况，看看是否有什么异常的行为。

下面将用一个基本的例子来展示如何采用“三步曲”来查看生产的分析报告。

1）、点击“Leak Suspects”，获取内存泄露总体概况：



在报告上最醒目的就是一张简洁明了的饼图，从图上我们可以清晰地看到哪些对象占用的内存最大。

在图的下方还有对这些对象的进一步描述。这段描述非常短，但我们已经可以从中找到很多线索了，比如是哪个类占用了绝大多数的内存，它属于哪个组件等等。

接下来，我们应该进一步去分析问题，为什么会占据了这么多内存，谁阻止了垃圾回收机制对它的回收。

2）、分析问题的所在

首先我们简单回顾下 JAVA 的内存回收机制，内存空间中垃圾回收的工作由垃圾回收器 (Garbage Collector,GC) 完成的，它的核心思想是：对虚拟机可用内存空间，即堆空间中的对象进行识别，如果对象正在被引用，那么称其为存活对象，反之，如果对象不再被引用，则为垃圾对象，可以回收其占据的空间，用于再分配。

在垃圾回收机制中有一组元素被称为根元素集合，它们是一组被虚拟机直接引用的对象，比如，正在运行的线程对象，系统调用栈里面的对象以及被 system class loader 所加载的那些对象。堆空间中的每个对象都是由一个根元素为起点被层层调用的。因此，一个对象还被某一个存活的根元素所引用，就会被认为是存活对象，不能被回收，进行内存释放。因此，我们可以通过分析一个对象到根元素的引用路径来分析为什么该对象不能被顺利回收。**如果说一个对象已经不被任何程序逻辑所需要，但是还存在被根元素引用的情况，我们可以说这里存在内存泄露。**

现在，让我们开始真正的寻找内存泄露之旅，点击“Details ”链接，可以看到如图所

示对可疑对象 1 的详细分析报告。

Class Name	Shallow Heap	Retained Heap
int[9643961] @ 0x240b1fc8	3,857,600	3,857,600
data sun.awt.image.IntegerInterleavedRaster @ 0x2445ffc8	104	3,857,912
raster java.awt.image.BufferedImage @ 0x2445ff70	40	3,857,952
bufImg sun.awt.image.BufImgSurfaceData @ 0x2445fcf8	56	3,858,008
sdPrevious, sdCurrent, sdBackup sun.java2d.windows.WinVolatileSurfaceManager @ 0x240b1f18	40	3,858,048
volSurfaceManager, surfaceManager sun.awt.image.SunVolatileImage @ 0x240b1ee8	48	48
value java.util.HashMap$Entry @ 0x23e01fc0	24	24
[1] java.util.HashMap$Entry[2] @ 0x23dfd1f0	24	48
table java.util.HashMap @ 0x23d54220	40	88
volatileMap javax.swing.RepaintManager @ 0x23d4a780	64	1,808
value java.util.HashMap$Entry @ 0x23d4f1c0	24	1,832
[16] java.util.HashMap$Entry[64] @ 0x23ef2270	272	9,648
table java.util.HashMap @ 0x23bc7700	40	9,688
table sun.awt.AppContext @ 0x23c0dde0	56	10,000
appContext sun.tools.jconsole.JConsole @ 0x23bf6948 Native Stack	440	1,032
mainAppContext class sun.awt.AppContext @ 0x2c4c8448 **System Class**	32	376
appContext javax.swing.ImageIcon$1 @ 0x23c09c70 »	200	208
appContext sun.tools.jconsole.JConsole$FixedJRootPane @ 0x23d46e00 »	360	1,016
appContext javax.swing.JMenuBar @ 0x23d470b0 »	336	2,680
appContext sun.tools.jconsole.ConnectDialog @ 0x23d71970 »	448	13,664

我们可以很清楚的看到整个引用链，内存聚集点是一个拥有大量对象的集合，如果你对程序代码比较熟悉的话，相信这些信息应该能给你提供一些找到内存泄露的思路了。

对于定位内存泄露，我们做到这一步已经可以了，其他事情，就交给开发处理吧。