

## 借助开源工具高效完成 Java 应用的运行分析

不止一次，我们都萌发过想对运行中程序的底层状况一探究竟的念头。产生这种需求的原因可能是运行缓慢的服务、Java 虚拟机（JVM）崩溃、挂起、死锁、频繁的 JVM 暂停、突然或持续的高 CPU 使用率、甚至于可怕的内存溢出(OOME)。好消息是现在已有许多工具能帮你得到 Java 虚拟机运行过程中的不同参数，这些信息有助于你了解其内部状况，从而诊断上述的各种情况。

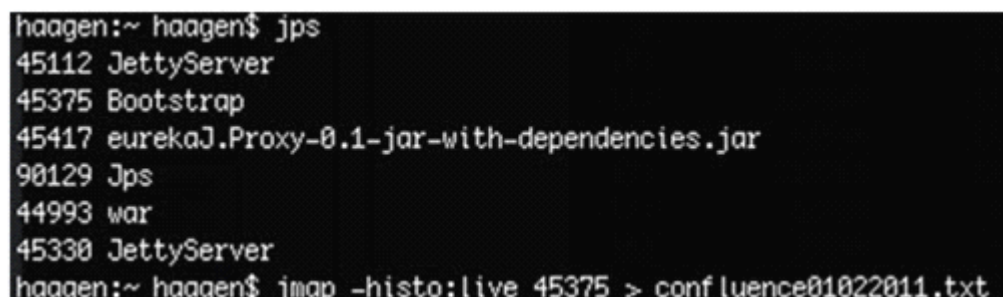
在这篇文章中，我将介绍一些优秀的开源工具。其中一些是 JVM 自带的，另一些则是第三方工具。我将从最简单的工具开始介绍，逐渐过渡到一些比较复杂的工具。本文的目的是帮助你找到合适的调试诊断工具，这样当程序出现执行异常、缓慢或根本不能执行时，手头随时有可用的工具。

如果程序出现不正常的高内存负载、频繁无响应或内存溢出，通常最好的分析切入点是查看内存对象。幸好 JVM 内置了工具“jmap”，让它天生就能完成这种任务

## Jmap（借助 JPM 的一点帮助）

Oracle 将描述为一种“输出进程、核心文件、远程调试服务器的共享对象内存映射和堆内存细节”的程序。本文将使用 jmap 打印一张内存统计图。

为了运行 jmap, 你需要知道被调试程序的 PID(进程标识符)。得到 PID 的简单办法是使用 JVM 提供的 jps, 它能列出机器上每一个 JVM 进程及其 PID。jps 输出结果如下图：



```
haagen:~ haagen$ jps
45112 JettyServer
45375 Bootstrap
45417 eurekaJ.Proxy-0.1-jar-with-dependencies.jar
98129 Jps
44993 war
45330 JettyServer
haagen:~ haagen$ jmap -histo:live 45375 > confluence01022011.txt
```

图1: jps 命令的终端输出

为了打印内存统计图，我们需要打开 jmap 控制台程序，并输入程序的 PID 和“-histo:live”选项。如果不添加这个选项，jmap 将完整导出该程序的堆内存,这不是我们想要的结果。所以，如果想得到上图中

“eureka.Proxy”程序的内存统计图，我们应该用如下命令来运行 jmap:

```
jmap -histo:live 45417
```

上述命令输出如下:

| num | #instances | #bytes   | class name                                 |
|-----|------------|----------|--|
| 1:  | 296538     | 35580576 | [D   |
| 2:  | 381465     | 14478320 | java.util.Hashtable\$Entry                 |
| 3:  | 296457     | 9486624  | com.akvaplan.data.input.MaanedsBestandData |
| 4:  | 298263     | 7158312  | java.lang.Integer                          |
| 5:  | 38183      | 5186248  | <constMethodKlass>                         |
| 6:  | 1830       | 4755416  | [Ljava.util.Hashtable\$Entry;              |
| 7:  | 38183      | 4587128  | <methodKlass>                              |
| 8:  | 99922      | 3996880  | com.akvaplan.data.input.CelleData          |
| 9:  | 3213       | 3747880  | <constantPoolKlass>                        |
| 10: | 3213       | 2805184  | <instanceKlassKlass>                       |
| 11: | 58179      | 2772864  | <symbolKlass>                              |
| 12: | 3838       | 2634480  | <constantPoolCacheKlass>                   |
| 13: | 99928      | 2398272  | java.lang.Double                           |
| 14: | 3527       | 1332464  | [Ljava.lang.Object;                        |

图2: 命令 *jmap -histo:live* 的输出结果显示了堆中现有对象的个数

结果中每行显示了当前堆中每种类类型的信息，包含被分配的实例个数及其消耗的字节数。

本例中，我请同事有意给程序增加了一处明显的内存泄露。请特别注意位于第 8 行的类， *CelleData*。将它与下图显示的4分钟后截屏进行比较

| num | #instances | #bytes   | class name                                 |
|-----|------------|----------|--|
| 1:  | 296540     | 35580720 | [D   |
| 2:  | 731623     | 29264920 | com.akvaplan.data.input.CelleData          |
| 3:  | 731629     | 17559096 | java.lang.Double                           |
| 4:  | 301431     | 14468688 | java.util.Hashtable\$Entry                 |
| 5:  | 296457     | 9486624  | com.akvaplan.data.input.MaanedsBestandData |
| 6:  | 3493       | 7615680  | [Ljava.lang.Object;                        |
| 7:  | 298240     | 7157760  | java.lang.Integer                          |
| 8:  | 38129      | 5189712  | <constMethodKlass>                         |
| 9:  | 1794       | 4751384  | [Ljava.util.Hashtable\$Entry;              |
| 10: | 38129      | 4590248  | <methodKlass>                              |
| 11: | 3217       | 3751824  | <constantPoolKlass>                        |
| 12: | 3217       | 2808136  | <instanceKlassKlass>                       |
| 13: | 58244      | 2776088  | <symbolKlass>                              |
| 14: | 3043       | 2637592  | <constantPoolCacheKlass>                   |

图3: *jmap* 的输出表明 *CelleData* 类的对象数目增加了

请注意 *CelleData* 类现在已经变为系统中第二多的类，短短4分钟内已经增加了631,701个额外实例。等待约一小时后，我们观察到如下结果：

| num | #instances | #bytes     | class name  |
|-----|------------|------------|---|
| 1:  | 25498787   | 1019951480 | com.akvaplan.data.input.CelleData                 |
| 2:  | 25672862   | 616148688  | java.lang.Double                                  |
| 3:  | 29140      | 254399608  | [Ljava.lang.Object;                               |
| 4:  | 3509460    | 196529760  | com.akvaplan.data.input.SimuleringArtResultatData |
| 5:  | 3836672    | 184160256  | java.util.Hashtable\$Entry                        |
| 6:  | 27456      | 45836312   | [Ljava.util.Hashtable\$Entry;                     |
| 7:  | 296540     | 35580720   | [D  |
| 8:  | 188418     | 16439080   | [C  |
| 9:  | 296457     | 9486624    | com.akvaplan.data.input.MaanedsBestandData        |
| 10: | 321439     | 7714536    | java.lang.Integer                                 |
| 11: | 189855     | 7594200    | java.lang.String                                  |
| 12: | 173664     | 6946560    | com.akvaplan.data.resultat.ResultatSimulering     |
| 13: | 38218      | 5200024    | <constMethodKlass>                                |
| 14: | 38218      | 4600928    | <methodKlass>                                     |

图4: 程序执行1小时后 *jmap* 的输出结果，显示超过2千5百万个 *CelleData* 类实例

现在有超过2千5百万个 *CelleData* 类实例，占用了超过1GB 内存！我们可以确认这是一个内存泄露。

这类数据信息的好处是，不仅非常有用而且对于很大的 JVM 堆也能快速反馈结果。我曾经试过检测一个运行频繁并且占用17GB 堆内存的程序，使用 *jmap* 能够在1分钟内生成程序的性能统计图。

需要注意的是，*jmap* 不是运行分析工具，在生成统计图时 JVM 可能会暂停，因此当生成统计图时需要确认这种暂停对程序是可接受的。以我的经验，通常在调试一个严重 bug 时需要生成这种统计图，这种情况

下，这些1分钟的暂停对程序来说是可接受的。这里，我们引出了下一个话题 - 半自动的运行分析工具 *VisualVM*。

## VisualVM

另一个包含于 JVM 中的工具是 **VisualVM**，它的开发者将它描述为“一种集成了多个 JDK 命令行工具的可视化工具，它能为您提供轻量级的运行分析能力”。这样看来，VisualVM 是另一种你最有可能用到的事后分析工具，一般是错误已出现或性能问题已经用传统方法（客户抱怨大多属于此类）发现。

继续之前的示例程序和它严重的内存泄露问题，在程序执行30分钟后，VisualVM 帮我们绘制了如下图表：

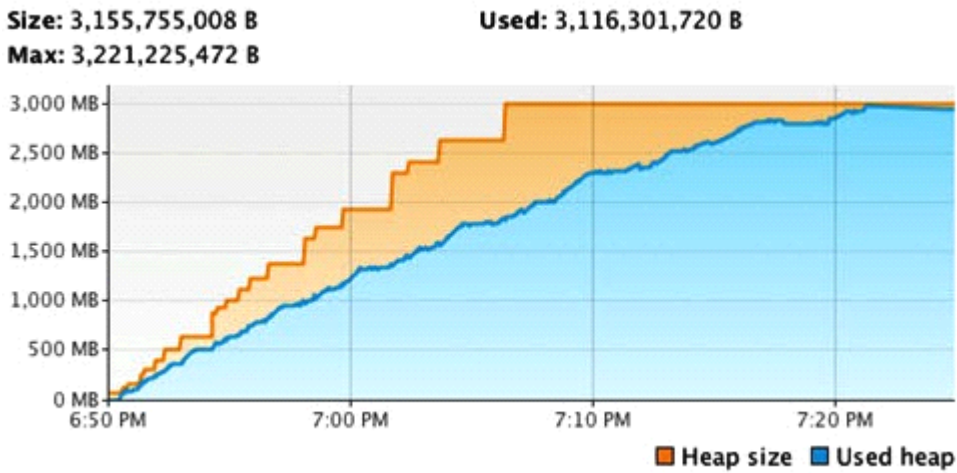


图5：程序初始运行的 VisualVM 内存图

从这个图表，我们可以清晰地看到截止到7:00pm，运行仅仅10分钟后，程序已经消耗掉超过1GB 的堆空间。又过了23分钟，JVM 已经到了它启动参数-Xmx3g 最大值，导致程序响应缓慢，系统响应缓慢（持续的垃圾回收）和数量惊人的内存溢出错误。

借助 jmap，我们定位了这种内存消耗攀升的原因。修复后，我们让程序重新运行于 VisualVM 的严格监测之下，观察到下面的情况：

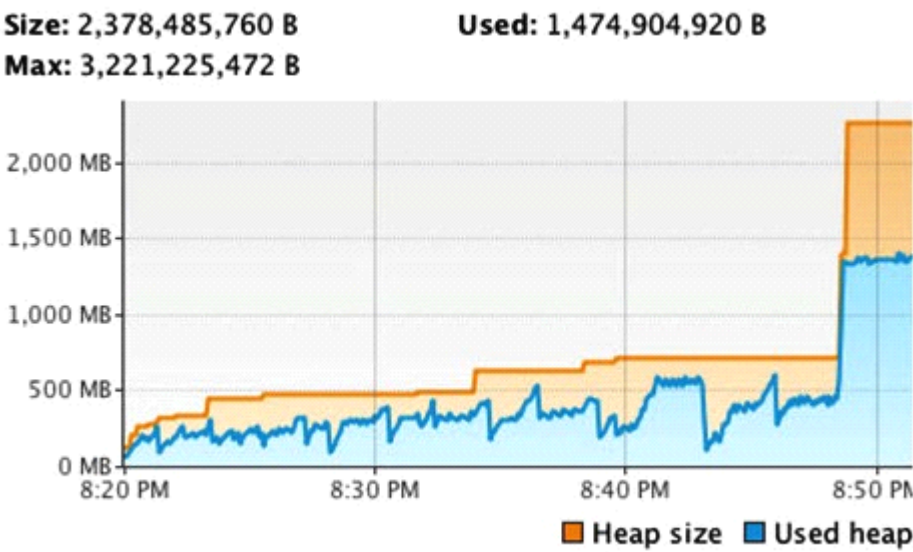


图6：修复内存泄露问题后的 *VisualVM* 内存图

如你所见，程序的内存曲线（启动参数仍然为-Xmx3g）有了明显改善。

除了内存图像工具，VisualVM 还提供了一个采样器和一个轻量级的剖析器（Profiler）。

VisualVM 采样器能周期采样程序 CPU 和内存的使用情况。得到的统计数据类似 jmap 的反馈，此外，你还可以通过采样得到方法调用对 CPU 的占用情况。它让你能快速了解周期采样过程中的方法执行次数：

（点击图片可以放大）

| Hot Spots - Method   | Self time [%] | Self time        | Self time (CPU) |
|--|---------------|------------------|-----------------|
| com.akvaplan.exec.StartBeregning.simuler ()                |               | 38112... (44.9%) | 38112 ms        |
| org.jfree.chart.encoders.SunPNGEncoderAdapter.encode ()    |               | 19811... (23.4%) | 19811 ms        |
| com.akvaplan.fil.ParseFiler.trimLinje ()                   |               | 16406... (19.3%) | 16406 ms        |
| com.akvaplan.fil.ParseFiler.lesInnCelleFil ()              |               | 6283... (7.4%)   | 6283 ms         |
| com.akvaplan.fil.ParseFiler.lesInnCelleFilLinje ()         |               | 1806... (2.1%)   | 1806 ms         |
| org.jfree.chart.renderer.category.AbstractCategoryItemRenc |               | 614 ms (0.7%)    | 614 ms          |
| com.akvaplan.fil.ParseFiler.parseResultatSimulering ()     |               | 399 ms (0.5%)    | 399 ms          |

图7： *VisualVM* 方法执行时间表

VisualVM 剖析器无需对程序周期采样就可以提供类似采样器的反馈信息，它还可以收集程序在整个正常执行过程中的统计数据（通过操纵程序源代码的字节码）。从剖析器得到的这种统计数据比从采样器而来的更精确和实时。

（点击图片可以放大）

| Class Name - Live Allocated Objects       | Live Bytes ▾ | Live Bytes          | Live Objects    | Generatio... |
|---|--------------|---------------------|-----------------|--------------|
| java.lang.Integer                         | <div></div>  | 8,125,456 B (27.3%) | 507,841 (47.2%) | 19           |
| java.util.TreeMap\$Entry                  | <div></div>  | 6,717,600 B (22.5%) | 167,940 (15.6%) | 28           |
| com.akvaplan.data.input.CelleData         | <div></div>  | 2,727,552 B (9.2%)  | 85,236 (7.9%)   | 137          |
| java.lang.Double                          | <div></div>  | 2,043,936 B (6.9%)  | 85,164 (7.9%)   | 137          |
| java.lang.Object[]                        | <div></div>  | 1,948,736 B (6.5%)  | 42,738 (4%)     | 107          |
| int[]                                     | <div></div>  | 1,478,840 B (5%)    | 2,610 (0.2%)    | 42           |
| java.util.TreeMap                         | <div></div>  | 1,292,784 B (4.3%)  | 26,933 (2.5%)   | 25           |
| char[]                                    | <div></div>  | 1,127,568 B (3.8%)  | 13,216 (1.2%)   | 53           |
| java.io.ObjectStreamClass\$WeakClassKey   | <div></div>  | 983,968 B (3.3%)    | 30,749 (2.9%)   | 18           |
| java.management.remote.jmxrmi.CombinedOut | <div></div>  | 644,864 B (2.2%)    | 16,860 (1.5%)   | 36           |

图8: VisualVM 剖析器的输出

但是，你必须考虑的另一方面是该剖析器属于一种“暴力”分析工具。它的检测方法本质上是重新定义程序执行中的大多数类和方法，结果必然会明显减缓程序执行速度。例如，上述程序运行部分的常规分析，大约要35秒。开启 VisualVM 的内存剖析器后，导致程序完成相同分析要31分钟。

我们需要清楚的是 VisualVM 并非功能齐全的剖析器。它无法在你的产品 JVM 上持续运行，不会保存分析数据，无法指定阈值，也不会在超过阈值时发出警报。要想更多的了解功能齐全的剖析器的目标。下面，让我们看看 BTrace，这个功能齐全的开源 java 代理程序。

## BTrace

想象一下，如果能收集 JVM 当前的任何信息，那么你感兴趣的信息有哪些？我猜想问题列表会将因人而异，因情形而异。就个人来说，我通常感兴趣的是以下的问题：

程序对堆、非堆、永久保存区（Permanent Generation），以及 JVM 包含的不同内存池（新生对象区、长期对象区、存活空间等）的内存使用情况

当前程序的线程数量，以及哪种类型线程正在被使用（单独计数）

JVM 的 CUP 负载

系统平均负载/系统 CPU 使用总和

对程序中的某些类和方法，我需要了解它们被调用次数，各自平均执行时间和整体平均时间

对 SQL 调用的调用计数及执行次数

对硬盘和网络操作的调用计数及执行次数

利用 BTrace 可以采集到所有以上信息，你可以使用 BTrace 脚本定义需要采集的数据。方便的是，BTrace 脚本就是普通 Java 类，包含一些特殊注解来定义 BTrace 在什么地方及如何跟踪你的程序。BTrace 脚本会被 BTrace 编译器-btracec 编译成标准的.class 文件。

BTrace 脚本包含许多部分，正如下图所示。如果需要了解下图脚本的详细内容，请[点击该链接](#)或访问 [BTrace 项目网站](#)。

由于 BTrace 仅仅是一个代理，记录结果后，它的任务就算完成了。除了文本输出，BTrace 并不具备动态展现被收集信息的功能。缺省情况下，BTrace 脚本输出结果将在 btrace.class 文件所在位置生成一个名为 *BTrace* 脚本名.class.btrace 的 text 文件。

我们可以通过给 BTrace 设置一个额外参数，让它按某时间间隔循环记录日志。切记，它最多能在100个日志文件间循环，当达到\*.class.btrace.99，它将覆盖\*.class.btrace.00文件。若让循环间隔在一个合理数字（如，每7.5秒）内，你就有充足时间来处理这些输出。只要在 java 代理的输入参数中加上 *fileRollMilliseconds=7500*，就可以实现日志循环。

BTrace 一大缺点是它比较原始，难以定义它的输出格式。你也许非常希望有一种更好的方式来处理 BTrace 的输出和数据，比如可以用一种一致的图形用户界面来展示。你可能还需要比较不同时间点的数据和超出阈值能发送警告。一个新的开源工具 **EurekaJ**，就此应运而生。

（点击图片可以放大）

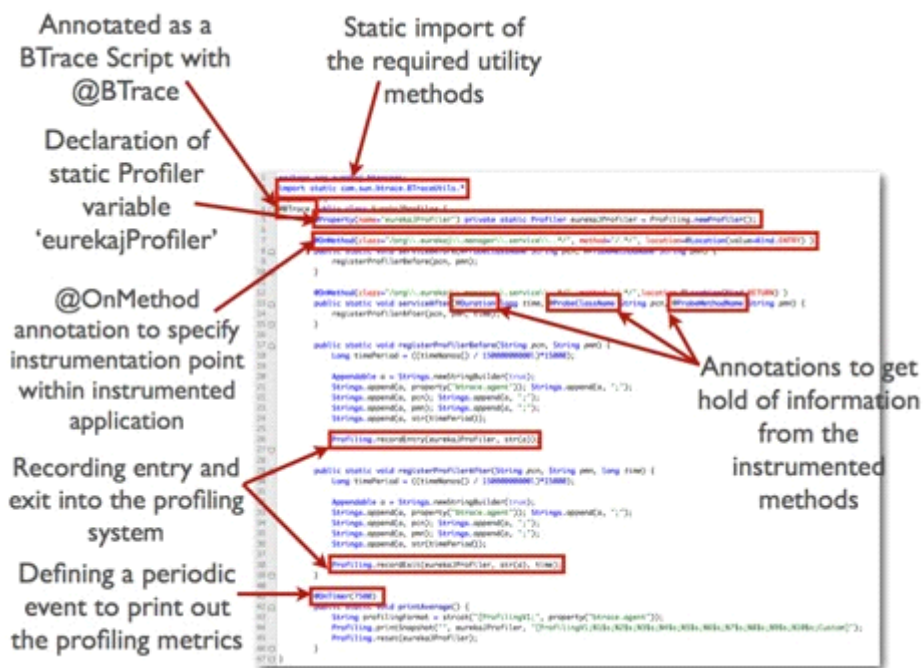


图9：激活方法分析时必需的 *BTrace* 脚本

## EurekaJ

我最初开发 **EurekaJ** 是在2008年。那时，我正在寻找一种具有我需要功能的开源剖析器，但没有找到。于是，我开始开发自己的工具。开发过程中，我涉猎了大量不同的技术并参考了许多架构模型，直到 **EurekaJ** 第一个版本发布。你可以从[项目网站](#)上了解更多的 **EurekaJ** 历史，查看源代码或下载并试着安装自己的版本。

**EurekaJ** 提供了两个主要应用：

- 1 一个基于 java 的管理器程序，可以接收传入的统计数据并一致地以可视化视图展现出来
- 2 一个解析 BTrace 输出的代理程序，将其转化为 JSON 格式并输入到 **EurekaJ** 管理程序的 REST 接口



EurekaJ 接受两种类型的输入数据格式。EurekaJ 代理期望 BTrace 脚本的输出被格式化为逗号分隔的文件（这点在 BTrace 中可很容易做到），而 EurekaJ 管理程序期望它的输入符合它的 JSON REST 接口格式。这意味着你可以通过代理程序或直接经由 REST 接口来传递度量数据。

借助 EurekaJ 管理程序，我们可以在一张图上分组显示多个统计数据、可以定义阈值和给接收者发出警报。我们还可以方便的查看收集到的实时数据或历史数据。

所有收集到的数据排序成一种逻辑树结构，其结构由 BTrace 脚本作者指定。我建议 BTrace 脚本的作者对相关统计数据分组，这样，当它们显示在 EurekaJ 中时会更容易理解和观察。例如，我个人喜欢对统计数据数据进行如下的逻辑分组：

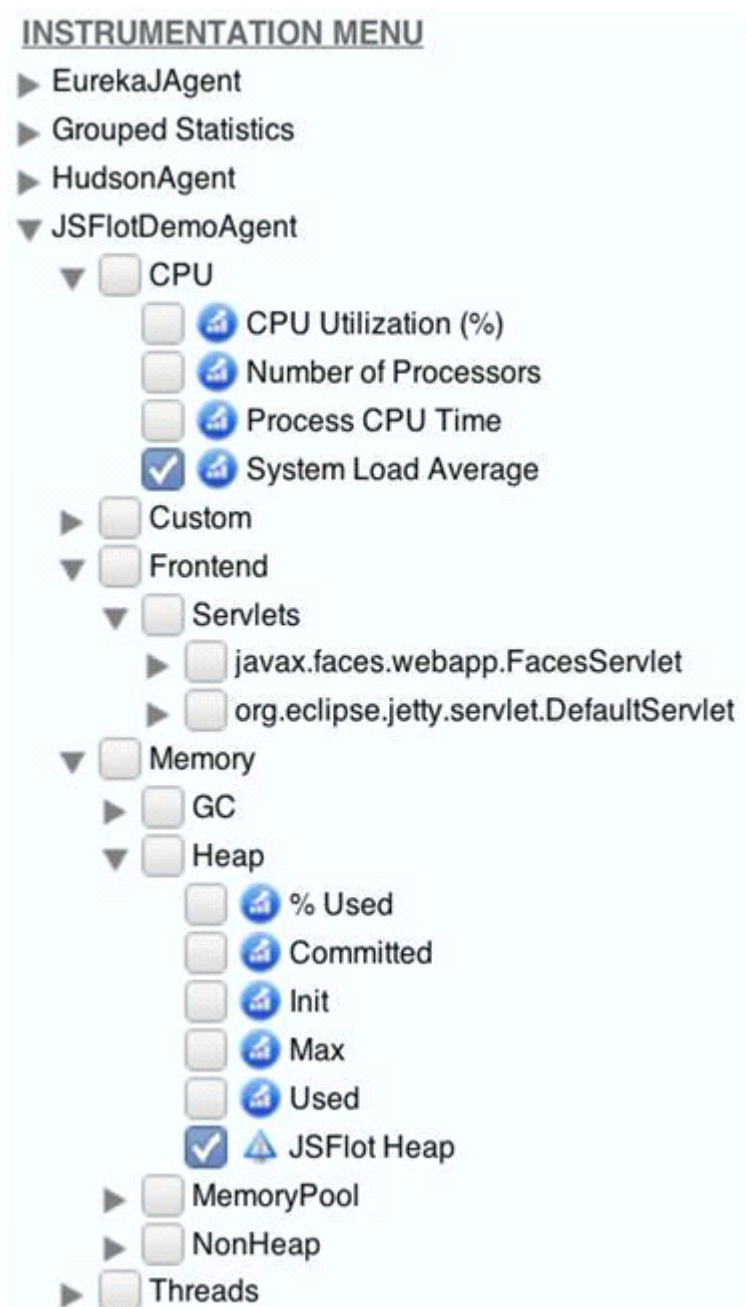




图10: *EurekaJ*演示程序的统计分组示例

图例

一种需要采集的重要信息是程序运行时的平均系统负载。要是你正面对一个运行缓慢的程序，那么缺陷可能并不在程序自身，而是隐藏到应用驻留的主机某处。我曾经在调试运行缓慢的应用时偶尔发现，真正的根源是病毒扫描程序。如果不进行测量分析，这种事情会很难被发现。考虑到这一点，我们需要能够在一张图中显示系统平均负载和进程加载后产生的负载。下图显示了一个运行 *EurekaJ* 演示程序的 Amazon EC2虚拟服务器的2小时平均负载（[该应用](#)登录的用户名和密码都是‘user’）。

（点击图片可以放大）

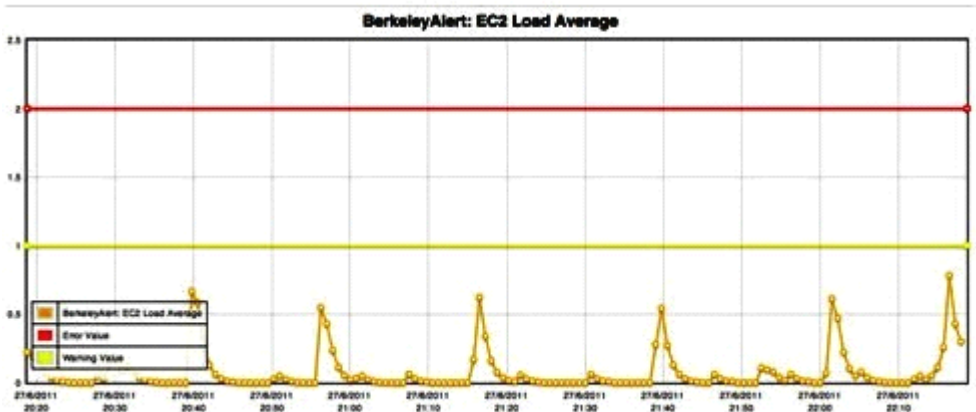


图11: 显示平均系统负载的 *EurekaJ* 图表

图中，黄色和红色的线条表示警戒阈值。一旦图形超过黄线的次数超过预设的最小警戒次数时，则测量结果到达“警告”状态。类似，若突破红线，测量结果就到达“危险”或“错误”状态。每当发生状态转换，*EurekaJ* 都会发送一封邮件给之前注册的收件人。

在上面的情形中，好像有周期性的事件每20分钟发生一次，从平均负载图上显示的波峰可以看到这一点。首先你要确定的是这个波峰确实由你的程序产生，而非其他原因。我们也可以通过测量进程的 CPU 负载来确认这点。在 *EurekaJ* 树菜单中，选择两个测量点后，两个图表结果会一起快速成像显示出来，其中一个位于另一个下面。

（点击图片可以放大）

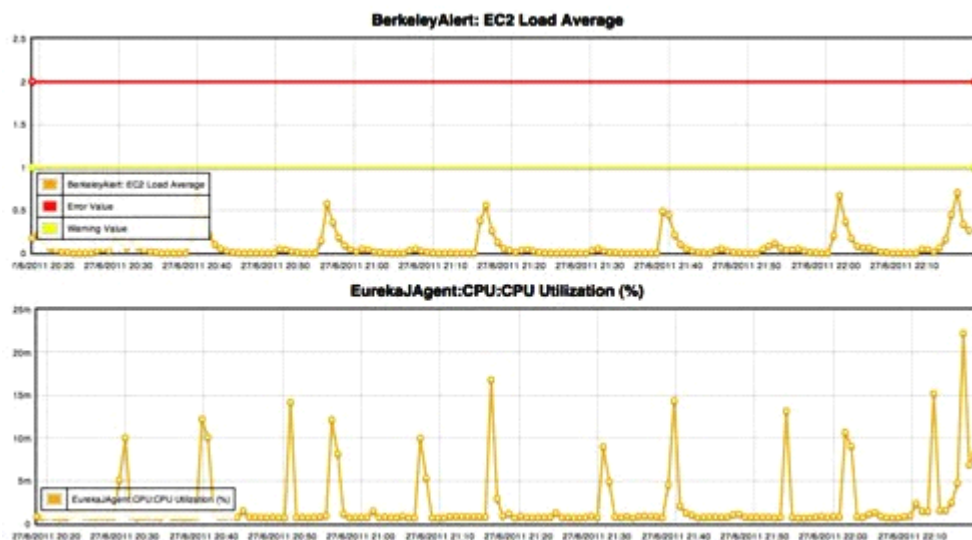


图12：同时显示多个图表

在上面的例子中，我们清楚地看到进程 CUP 占用和系统负载存在必然的联系。

许多应用需要在程序无响应或不可用时及时发出警告。下图是一个 Confluence（Atlassian 的企业级 Wiki 软件）的例子。这个例子中，程序内存占用快速上升，直到产生程序内存溢出。这时，Confluence 无法处理接收到的请求，同时日志文件记录了各种奇怪的错误。

你可能希望当程序运行导致内存溢出时，程序能立刻抛出一个 OOME（内存溢出错误），然而，事实上 JVM 不会抛出 OOME 直到它发觉垃圾回收过于缓慢。结果，程序没有完全崩溃，又过了2小时，Java 仍然没有抛出 OutOfMemoryError，甚至两小时后程序依然在“运行”（意味着 JVM 进程仍然在运行）。显然，这时任何进程监测工具都不能发现程序已经“停止”。

（点击图片可以放大）

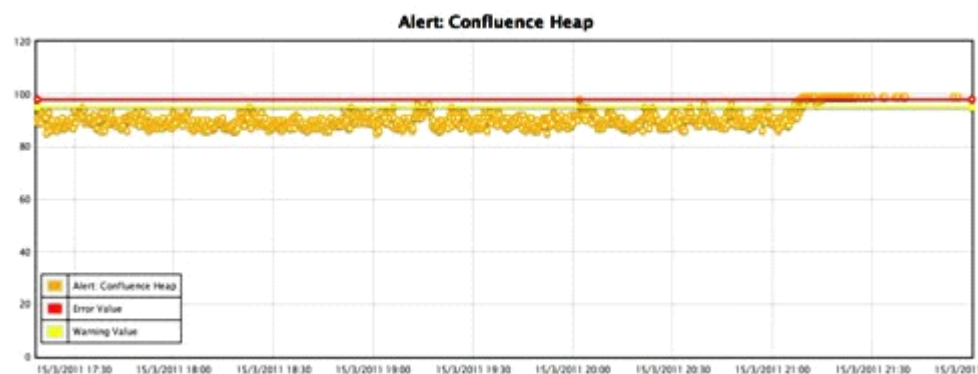


图13：EurekaJ堆图显示内存溢出错误的一种可能情形

注意最后几个小时的执行情况，图表揭示了下面的度量指标。

(点击图片可以放大)

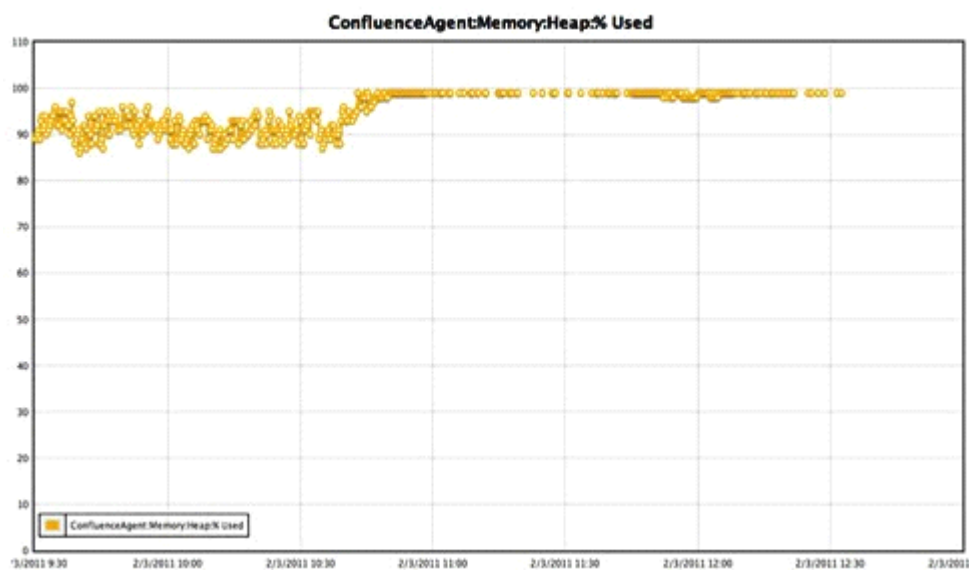


图14：前面图表放大后的效果

EurekaJ 使我们可以设置程序的堆内存警告，个人建议最好如此。若程序持续占用堆内存超过95%-98%（取决于堆的大小），几乎可以肯定，程序存在内存问题，要么用-Xmx 参数为程序分配更多的堆，要么优化程序使其使用更少内存。同时，EurekaJ 未来版本计划增加统计数据不足的警报。

最后的图表示例展示了一个包含4个不同程序内存使用的图表组。这种类型的图表组可方便用来比较一个程序不同部分的、或甚至不同程序之间、服务器之间的数据。下图的这4个程序有不同的内存需求和内存占用模式。

如下图示，不同程序有不同的内存曲线。这些曲线非常依赖一些实际情况，比如使用的架构、缓存数量、用户数、程序负载等。我希望通过下图说明你需要掌握程序在正常和高负载下执行情况的重要性，因为这将直接关系到如何定义报警阈值。

(点击图片可以放大)



图15：EurekaJ 图组会将图像彼此叠加在一起

注意性能干扰 – 让非热点区不受影响！

你使用的每一种测量方法似乎都会引起系统性能干扰。一些数据的测量可以被认为“无干扰”（或“忽略不计”），然而另外一些数据的测量可称得上代价昂贵。非常重要的一点是，要知道你需要 BTrace 测量什么。因此，你要将这种分析工具对程序的性能干扰减少到最小。关于这点，请参考下面的3条原则：

基于“采样”的度量通常可被认为“无影响”。采样 CPU 负载、进程 CPU 负载、内存使用和每5-10秒的线程计数，其带来的额外一两个毫秒的影响可被忽略。在我看来，你应该经常收集这类统计数据，它们对你来说不会有什么损耗。

对长时间运行的任务的测量也可被认为“无影响”。通常，它仅会对每个被测量方法带来1700-2500纳秒的影响。如果你正测量这些对象的执行时间：SQL 查询、网络流量、硬盘读写或一个预期范围在40毫秒（磁盘存取）到1秒（Servlet 处理）之间的 Servlet 处理过程，那么对这些对象每个增加额外的2500纳秒左右的时间也是可接受的。

绝对不要对循环内执行的方法进行测量

寻找程序热点区的一个通用规则是不要影响非热点区域。例如，考虑下面的类：

（点击图片可以放大）

```
1 package example;
2
3 public class MyDaoClass {
4     private List<Statistics> readStatFromDatabase(String statName, long msFrom, long msTo) throws SQLException {
5         String sql = "select * from Statistics s where s.StatisticName = ? and s.StatisticTimestamp between ? and ?";
6
7         PreparedStatement preparedStatement = derbyEnvironment.getConnection().prepareStatement(sql);
8         preparedStatement.setString(1, statName);
9         preparedStatement.setTimestamp(2, new Timestamp(msFrom));
10        preparedStatement.setTimestamp(3, new Timestamp(msTo));
11
12        List<Statistics> statList = new ArrayList<Statistics>();
13        ResultSet rs = preparedStatement.executeQuery();
14        while (rs.next()) {
15            statList.add(buildNewStat(rs));
16        }
17        return statList;
18    }
19
20    public Statistics buildNewStat(ResultSet rs) {
21        Statistics stat = new Statistics();
22        stat.setStatisticID(rs.getLong("StatisticID"));
23        stat.setStatisticName(rs.getString("StatisticName"));
24        stat.setStatisticValue(rs.getDouble("StatisticValue"));
25
26        return stat;
27    }
28 }
29 }
```

图16：需要测量的简单的数据访问接口类

第5行的 SQL 执行时间可能使 *readStatFromDatabase* 方法可能成为一个热点。当查询返回相当多的数据行时，它无疑会成为一个热点，这对13行（程序和数据库服务器之间的网络流量）和14-16行（结果集中每行所需处理）会造成负面影响。如果从数据库返回结果时间过长，该方法也会成为一个热点（在13行）。

方法 *buildNewStat* 就其本身来说似乎绝不会成为一个热点。即使被多次执行，每次调用都会在几纳秒内完成。另一方面，若给每次调用增加了2500纳秒的测量采集干扰，则无论 SQL 何时被执行，都势必会让该方法看起来像个热点。因此，我们要避免测量它。

（点击图片可以放大）

Depending on your needs, adding a timer-instrumentation to this method is OK.

Adding a timer-instrumentation to this method will most likely convert it to a hot-spot!

```

1 package example;
2
3 public class MyObjClass {
4     private List<Statistics> readStatFromDatabase(String statName, long nsFrom, long nsTo) throws SQLException {
5         String sql = "SELECT * FROM STATISTICS WHERE S.StatisticName = ? AND S.StatisticTimestamp BETWEEN ? AND ?";
6
7         PreparedStatement preparedStatement = derbyEnvironment.getConnection().prepareStatement(sql);
8         preparedStatement.setString(1, statName);
9         preparedStatement.setTimestamp(2, new Timestamp(nsFrom));
10        preparedStatement.setTimestamp(3, new Timestamp(nsTo));
11
12        List<Statistics> statList = new ArrayList<Statistics>();
13        ResultSet rs = preparedStatement.executeQuery();
14        while (rs.next()) {
15            statList.add(buildNewStat(rs));
16        }
17
18        return statList;
19    }
20
21    public Statistics buildNewStat(ResultSet rs) {
22        Statistics stat = new Statistics();
23        stat.setStatisticID(rs.getLong("StatisticID"));
24        stat.setStatisticName(rs.getString("StatisticName"));
25        stat.setStatisticValue(rs.getDouble("StatisticValue"));
26
27        return stat;
28    }
29 }

```

图17：显示上面类哪些部分可以被测量，哪些需要避免

## 建立完整的运行分析

使用 EurekaJ 建立一个完整的运行分析，需要以下几个主要部分：

准备需要监测/操纵的程序

BTrace - java 代理

告知 BTrace 如何测量的 BTrace 脚本

存储 BTrace 输出的文件系统

将 BTrace 输出传输到 EurekaJ 管理器的 EurekaJ 代理

安装好的 EurekaJ 管理器（本地安装或可通过互联网访问的远程安装）

（点击图片可以放大）

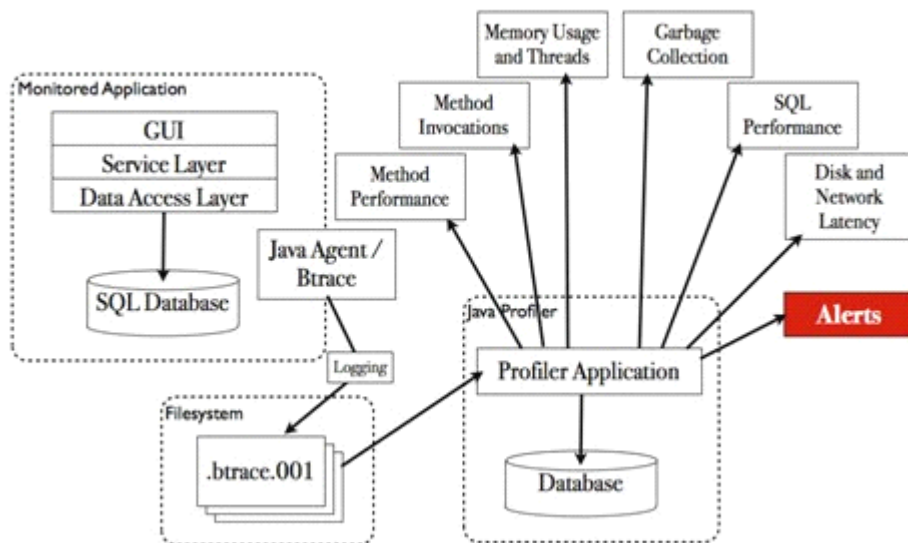


图18：使用本文所描述工具对程序进行运行分析的概览图

关于这些产品的完整安装手册，请访问 [EurekaJ 项目网站](#)。

## 总结

这篇文章给我们介绍了一些用于程序运行分析的开源工具，它们不仅能帮我们完成对运行中 JVM 的深度分析，而且可以帮助我们对开发、测试和程序部署进行多方位的持续监测。

希望你已经开始了了解不断收集度量信息的好处和超过阈值后及时报警能力的重要性。