



# 实战 Java 虚拟机

## ——JVM 故障诊断与性能优化

葛一鸣 著

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

## 内 容 简 介

随着越来越多的第三方语言（Groovy、Scala、JRuby 等）在 Java 虚拟机上运行，Java 也俨然成为了一个充满活力的生态圈。本书将通过 200 余示例详细介绍 Java 虚拟机中的各种参数配置、故障排查、性能监控以及性能优化。

本书共 11 章。第 1~3 章介绍了 Java 虚拟机的定义、总体架构、常用配置参数。第 4~5 章介绍了垃圾回收的算法和各种垃圾回收器。第 6 章介绍了 Java 虚拟机的性能监控和故障诊断工具。第 7 章详细介绍了对 Java 堆的分析方法和案例。第 8 章介绍了 Java 虚拟机对多线程，尤其是对锁的支持。第 9~10 章介绍了 Java 虚拟机的核心——Class 文件结构，以及 Java 虚拟机中类的装载系统。第 11 章介绍了 Java 虚拟机的执行系统和字节码，并给出了通过 ASM 框架进行字节码注入的案例。

本书不仅适合 Java 程序员，还适合任何一名工作于 Java 虚拟机之上的研发人员、软件设计师、架构师。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有，侵权必究。

### 图书在版编目（CIP）数据

实战 Java 虚拟机：JVM 故障诊断与性能优化 / 葛一鸣著. —北京：电子工业出版社，2015.3  
（51CTO 学院系列丛书）  
ISBN 978-7-121-25612-7

I. ①实… II. ①葛… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字（2015）第 040500 号

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 [zltz@phei.com.cn](mailto:zltz@phei.com.cn)，盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

服务热线：（010）88258888。

# 前言

## 关于 Java 生态圈

Java 是目前应用最为广泛的软件开发平台之一。随着 Java 以及 Java 社区的不断壮大，Java 也早已不再是简简单单的一门计算机语言了，它更是一个平台、一种文化、一个社区。

作为一个平台，Java 虚拟机扮演着举足轻重的作用。除了 Java 语言，任何一种能够被编译成字节码的计算机语言都属于 Java 这个平台。Groovy、Scala、JRuby 等都是 Java 平台的一个部分，它们依赖于 Java 虚拟机，同时，Java 平台也因为它们变得更加丰富多彩。

作为一种文化，Java 几乎成为了“开源”的代名词。在 Java 程序中，有着数不清的开源软件和框架，如 Tomcat、Struts、Hibernate、Spring 等。就连 JDK 和 JVM 自身也有不少开源的实现，如 OpenJDK、Harmony。可以说，“共享”的精神在 Java 世界里体现得淋漓尽致。

作为一个社区，Java 拥有无数的开发人员，有数不清的论坛和资料。从桌面应用软件、嵌入式开发到企业级应用、后台服务器、中间件，都可以看到 Java 的身影。其应用形式之复杂、参与人数之众多也令人咋舌。可以说，Java 社区已经俨然成为了一个良好而庞大的生态系统。

而本书，将主要介绍这个生态系统的核心——**Java 虚拟机**。

## 本书的体系结构

本书立足于实际开发，又不缺乏理论介绍，力求通俗易懂、循序渐进。本书共分为 11 章：

第1章主要为综述，介绍了 Java 虚拟机的概念、定义，讲解了 Java 语言规范和 Java 虚拟机规范，最后，还介绍了 OpenJDK 的调试方法。

第2章介绍了 Java 虚拟机的总体架构，说明了堆、栈、方法区等内存空间的作用和彼此之间的联系。

第3章介绍了 Java 虚拟机的常用配置参数，重点对垃圾回收跟踪参数、内存配置参数做了详细的介绍，并给出了案例说明。

第4章从理论层面介绍了垃圾回收的算法，如引用计数、标记清除、标记压缩、复制算法等。本章是第5章的理论基础。

第5章讲解了基于垃圾回收的理论知识，进一步详细介绍了 Java 虚拟机中实际使用的各种垃圾回收器，包括串行回收器、并行回收器、CMS、G1 等。

第6章介绍了 Java 虚拟机的性能监控和故障诊断工具，考虑到实用性，也介绍了系统级性能监控工具的使用，两者结合，可以更好地帮助读者处理实际问题。

第7章详细介绍了对 Java 堆的分析方法和案例，主要讲解了 MAT 和 Visual VM 两款工具的使用，以及各自 OQL 的编写方式。

第8章介绍了 Java 虚拟机对多线程，尤其是对锁的支持，本章不仅介绍了虚拟机内部锁的实现、优化机制，也给出了一些 Java 语言层面的锁优化思路，最后，还介绍了无锁的并行控制方法。

第9章介绍了 Java 虚拟机的核心——Class 文件结构，Class 文件作为 Java 虚拟机的基石，有着举足轻重的作用，对深入理解 Java 虚拟机有着不可忽视的作用。

第10章介绍了 Java 虚拟机中类的装载系统，其中，着重介绍了 Java 虚拟机中 ClassLoader 的实现以及设计模式。

第11章介绍了 Java 虚拟机的执行系统和字节码，为了帮助读者更快更好地理解 Java 字节码，本章对字节码进行了分类讲解，并且理论联系实际，给出了通过 ASM 框架进行字节码注入的案例。

## 本书特色

本书的主要特点有：

1. **结构清晰。**本书采用从整体到局部的视角，首先第1、2章介绍了 Java 虚拟机的整体概

况和结构。接着步步为营，每一章节对应一个单独的知识点，力求展示虚拟机的全貌。

2. **理论结合实战。**本书不甘心于简单地枚举理论知识，在每一个理论背后，都给出了演示示例供读者参考，帮助读者更好地消化这些理论。比如，在对 Class 文件结构和字节码的介绍中，不仅仅简单地给出了理论说明，更是使用 ASM 框架将这些理论应用于实践，尽可能地做到理论和实践结合。
3. **专注专业。**本书着眼于 Java 虚拟机，对 Java 虚拟机的原理和实践做了丰富的介绍，包括但不限于体系结构、虚拟机的调试方式、常用参数、垃圾回收系统、Class 文件结构、执行系统等，力求从多角度更专业地对 Java 虚拟机进行探讨。
4. **通俗易懂。**本书依然服务于广大虚拟机初学者，尽量避免采用过于理论的描述方式，简单的白话文风格贯穿全书，尽量做到读者在阅读过程中少盲点、无盲点。
5. **技术全面。**纵横 Windows 和 Linux 双系统下的性能诊断、涉及 32 位系统和 64 位系统的优化比较、贯穿从 JDK 1.5 到 JDK 1.8 的优化演进。

## 适合阅读人群

虽然本书力求通俗，但要通读本书并取得良好的学习效果，要求读者需要具备基本的 Java 知识或者一定的编程经验。因此，本书适合以下读者：

- 拥有一定开发经验的 Java 平台开发人员（Java、Scala、JRuby 等）
- 软件设计师、架构师
- 系统调优人员
- 有一定的 Java 编程基础并希望进一步理解 Java 的程序员
- 虚拟机爱好者，JVM 实践者

## 本书的约定

本书在叙述过程中，有如下约定：

- 本书中所述的 JDK 1.5、JDK 1.6、JDK 1.7、JDK 1.8 等同于 JDK 5、JDK 6、JDK 7、JDK 8。
- 如无特殊说明，Java 虚拟机均指 HotSpot 虚拟机。
- 如无特殊说明，本书的程序、示例均在 JDK 1.7 环境中运行。

- 本书赠送的课程优惠券,可以观看笔者在 51CTO 学院的 JVM 课程,地址是 [http://edu.51cto.com/course/course\\_id-1952.html](http://edu.51cto.com/course/course_id-1952.html)。

## 联系作者

本书的写作过程远比我想象的更艰辛,为了让全书能够更清楚、更正确地表达和论述,我经历了好多个不眠之夜,即使现在回想起来,也忍不住让我打个寒战。由于写作水平的限制,书中难免会有不妥之处,望读者谅解。

为此,如果读者有任何疑问或者建议,非常欢迎大家加入 QQ 群 397196583,一起探讨学习中的困难、分享学习的经验,我期待与大家一起交流、共同进步。同时,也希望大家可以关注我的博客 <http://www.uucode.net/>。

## 感谢

这本书能够面世,是因为得到了众人的支持。首先,要感谢我的妻子,她始终不辞辛劳,毫无怨言地对我照顾有加,才让我得以腾出大量时间,并可以安心工作。其次,要感谢小编为我一次又一次地审稿改错,批评指正,才能让本书逐步完善。最后,感谢我的母亲 30 年如一日对我的体贴和关心。

葛一鸣

# 目 录

第 1 章 初探 Java 虚拟机 .....	1
1.1 知根知底：追溯 Java 的发展历程 2	
1.1.1 那些依托 Java 虚拟机的语言大咖们 2	
1.1.2 Java 发展史上的里程碑 2	
1.2 跨平台的真相：Java 虚拟机来做中介 4	
1.2.1 理解 Java 虚拟机的原理 4	
1.2.2 看清 Java 虚拟机的种类 5	
1.3 一切看我的：Java 语言规范 6	
1.3.1 词法的定义 6	
1.3.2 语法的定义 7	
1.3.3 数据类型的定义 8	
1.3.4 Java 语言规范总结 9	
1.4 一切听我的：Java 虚拟机规范 9	
1.5 数字编码就是计算机世界的水和电 10	
1.5.1 整数在 Java 虚拟机中的表示 10	
1.5.2 浮点数在 Java 虚拟机中的表示 12	

1.6 抛砖引玉：编译和调试虚拟机	14
1.7 小结	19
第 2 章 认识 Java 虚拟机的基本结构	20
2.1 谋全局者才能成大器：看穿 Java 虚拟机的架构	20
2.2 小参数能解决大问题：学会设置 Java 虚拟机的参数	22
2.3 对象去哪儿：辨清 Java 堆	23
2.4 函数如何调用：出入 Java 栈	25
2.4.1 局部变量表	27
2.4.2 操作数栈	32
2.4.3 帧数据区	32
2.4.4 栈上分配	33
2.5 类去哪儿了：识别方法区	35
2.6 小结	37
第 3 章 常用 Java 虚拟机参数	38
3.1 一切运行都有迹可循：掌握跟踪调试参数	38
3.1.1 跟踪垃圾回收——读懂虚拟机日志	39
3.1.2 类加载/卸载的跟踪	42
3.1.3 系统参数查看	44
3.2 让性能飞起来：学习堆的配置参数	45
3.2.1 最大堆和初始堆的设置	45
3.2.2 新生代的配置	49
3.2.3 堆溢出处理	52
3.3 别让性能有缺口：了解非堆内存的参数配置	54



3.3.1	方法区配置	55
3.3.2	栈配置	55
3.3.3	直接内存配置	55
3.4	Client 和 Server 二选一：虚拟机的工作模式	58
3.5	小结	59
第 4 章	垃圾回收概念与算法	60
4.1	内存管理清洁工：认识垃圾回收	60
4.2	清洁工具大 PK：讨论常用的垃圾回收算法	61
4.2.1	引用计数法（Reference Counting）	62
4.2.2	标记清除法（Mark-Sweep）	63
4.2.3	复制算法（Copying）	64
4.2.4	标记压缩法（Mark-Compact）	66
4.2.5	分代算法（Generational Collecting）	67
4.2.6	分区算法（Region）	68
4.3	谁才是真正的垃圾：判断可触及性	69
4.3.1	对象的复活	69
4.3.2	引用和可触及性的强度	71
4.3.3	软引用——可被回收的引用	72
4.3.4	弱引用——发现即回收	76
4.3.5	虚引用——对象回收跟踪	77
4.4	垃圾回收时的停顿现象：Stop-The-World 案例实战	79
4.5	小结	83
第 5 章	垃圾收集器和内存分配	84

- 5.1 一心一意一件事：串行回收器 85
  - 5.1.1 新生代串行回收器 85
  - 5.1.2 老年代串行回收器 86
- 5.2 人多力量大：并行回收器 86
  - 5.2.1 新生代 ParNew 回收器 87
  - 5.2.2 新生代 ParallelGC 回收器 88
  - 5.2.3 老年代 ParallelOldGC 回收器 89
- 5.3 一心多用都不落下：CMS 回收器 90
  - 5.3.1 CMS 主要工作步骤 90
  - 5.3.2 CMS 主要的设置参数 91
  - 5.3.3 CMS 的日志分析 92
  - 5.3.4 有关 Class 的回收 94
- 5.4 未来我做主：G1 回收器 95
  - 5.4.1 G1 的内存划分和主要收集过程 95
  - 5.4.2 G1 的新生代 GC 96
  - 5.4.3 G1 的并发标记周期 97
  - 5.4.4 混合回收 100
  - 5.4.5 必要时的 Full GC 102
  - 5.4.6 G1 日志 102
  - 5.4.7 G1 相关的参数 106
- 5.5 回眸：有关对象内存分配和回收的一些细节问题 107
  - 5.5.1 禁用 System.gc() 107
  - 5.5.2 System.gc()使用并发回收 107

5.5.3	并行 GC 前额外触发的新生代 GC	109
5.5.4	对象何时进入老年代	110
5.5.5	在 TLAB 上分配对象	117
5.5.6	方法 finalize()对垃圾回收的影响	120
5.6	温故又知新：常用的 GC 参数	125
5.7	动手才是真英雄：垃圾回收器对 Tomcat 性能影响的实验	127
5.7.1	配置实验环境	127
5.7.2	配置进行性能测试的工具 JMeter	128
5.7.3	配置 Web 应用服务器 Tomcat	131
5.7.4	实战案例 1——初试串行回收器	133
5.7.5	实战案例 2——扩大堆以提升系统性能	133
5.7.6	实战案例 3——调整初始堆大小	134
5.7.7	实战案例 4——使用 ParrellOldGC 回收器	135
5.7.8	实战案例 5——使用较小堆提高 GC 压力	135
5.7.9	实战案例 6——测试 ParallelOldGC 的表现	135
5.7.10	实战案例 7——测试 ParNew 回收器的表现	136
5.7.11	实战案例 8——测试 JDK 1.6 的表现	136
5.7.12	实战案例 9——使用高版本虚拟机提升性能	137
5.8	小结	137
第 6 章	性能监控工具	138
6.1	有我更高效：Linux 下的性能监控工具	139
6.1.1	显示系统整体资源使用情况——top 命令	139
6.1.2	监控内存和 CPU——vmstat 命令	140

- 6.1.3 监控 IO 使用——iostat 命令 142
- 6.1.4 多功能诊断器——pidstat 工具 143
- 6.2 用我更高效：Windows 下的性能监控工具 148
  - 6.2.1 任务管理器 148
  - 6.2.2 perfmon 性能监控工具 150
  - 6.2.3 Process Explorer 进程管理工具 153
  - 6.2.4 pslist 命令——Windows 下也有命令行工具 155
- 6.3 外科手术刀：JDK 性能监控工具 157
  - 6.3.1 查看 Java 进程——jps 命令 158
  - 6.3.2 查看虚拟机运行时信息——jstat 命令 159
  - 6.3.3 查看虚拟机参数——jinfo 命令 162
  - 6.3.4 导出堆到文件——jmap 命令 163
  - 6.3.5 JDK 自带的堆分析工具——jhat 命令 165
  - 6.3.6 查看线程堆栈——jstack 命令 167
  - 6.3.7 远程主机信息收集——jstatd 命令 170
  - 6.3.8 多功能命令行——jcmd 命令 172
  - 6.3.9 性能统计工具——hprof 175
  - 6.3.10 扩展 jps 命令 177
- 6.4 我是你的眼：图形化虚拟机监控工具 JConsole 178
  - 6.4.1 JConsole 连接 Java 程序 178
  - 6.4.2 Java 程序概况 179
  - 6.4.3 内存监控 180
  - 6.4.4 线程监控 180

6.4.5	类加载情况	182
6.4.6	虚拟机信息	182
6.5	一目了然：可视化性能监控工具 Visual VM	183
6.5.1	Visual VM 连接应用程序	184
6.5.2	监控应用程序概况	185
6.5.3	Thread Dump 和分析	186
6.5.4	性能分析	187
6.5.5	内存快照分析	189
6.5.6	BTrace 介绍	190
6.6	来自 JRockit 的礼物：虚拟机诊断工具 Mission Control	198
6.6.1	MBean 服务器	198
6.6.2	飞机记录器 (Flight Recorder)	200
6.7	小结	203
第 7 章	分析 Java 堆	204
7.1	对症下药：找到内存溢出的原因	205
7.1.1	堆溢出	205
7.1.2	直接内存溢出	205
7.1.3	过多线程导致 OOM	207
7.1.4	永久区溢出	209
7.1.5	GC 效率低下引起的 OOM	210
7.2	无处不在的字符串：String 在虚拟机中的实现	210
7.2.1	String 对象的特点	210
7.2.2	有关 String 的内存泄漏	212

7.2.3	有关 String 常量池的位置	215
7.3	虚拟机也有内窥镜：使用 MAT 分析 Java 堆	217
7.3.1	初识 MAT	217
7.3.2	浅堆和深堆	220
7.3.3	例解 MAT 堆分析	221
7.3.4	支配树（Dominator Tree）	225
7.3.5	Tomcat 堆溢出分析	226
7.4	筛选堆对象：MAT 对 OQL 的支持	230
7.4.1	Select 子句	230
7.4.2	From 子句	232
7.4.3	Where 子句	234
7.4.4	内置对象与方法	234
7.5	更精彩的查找：Visual VM 对 OQL 的支持	239
7.5.1	Visual VM 的 OQL 基本语法	239
7.5.2	内置 heap 对象	240
7.5.3	对象函数	242
7.5.4	集合/统计函数	247
7.5.5	程序化 OQL 分析 Tomcat 堆	252
7.6	小结	255
第 8 章	锁与并发.....	256
8.1	安全就是锁存在的理由：锁的基本概念和实现	257
8.1.1	理解线程安全	257
8.1.2	对象头和锁	259

8.2	避免残酷的竞争：锁在 Java 虚拟机中的实现和优化	260
8.2.1	偏向锁	260
8.2.2	轻量级锁	262
8.2.3	锁膨胀	263
8.2.4	自旋锁	264
8.2.5	锁消除	264
8.3	应对残酷的竞争：锁在应用层的优化思路	266
8.3.1	减少锁持有时间	266
8.3.2	减小锁粒度	267
8.3.3	锁分离	269
8.3.4	锁粗化	271
8.4	无招胜有招：无锁	273
8.4.1	理解 CAS	273
8.4.2	原子操作	274
8.4.3	新宠儿 LongAddr	277
8.5	将随机变为可控：理解 Java 内存模型	280
8.5.1	原子性	280
8.5.2	有序性	282
8.5.3	可见性	284
8.5.4	Happens-Before 原则	286
8.6	小结	286
第 9 章	Class 文件结构	287
9.1	不仅跨平台，还能跨语言：语言无关性	287

- 9.2 虚拟机的基石：Class 文件 289
  - 9.2.1 Class 文件的标志——魔数 290
  - 9.2.2 Class 文件的版本 292
  - 9.2.3 存放所有常数——常量池 293
  - 9.2.4 Class 的访问标记（Access Flag） 300
  - 9.2.5 当前类、父类和接口 301
  - 9.2.6 Class 文件的字段 302
  - 9.2.7 Class 文件的方法基本结构 304
  - 9.2.8 方法的执行主体——Code 属性 306
  - 9.2.9 记录行号——LineNumberTable 属性 307
  - 9.2.10 保存局部变量和参数——LocalVariableTable 属性 308
  - 9.2.11 加快字节码校验——StackMapTable 属性 308
  - 9.2.12 Code 属性总结 313
  - 9.2.13 抛出异常——Exceptions 属性 314
  - 9.2.14 用实例分析 Class 的方法结构 315
  - 9.2.15 我来自哪里——SourceFile 属性 318
  - 9.2.16 强大的动态调用——BootstrapMethods 属性 319
  - 9.2.17 内部类——InnerClasses 属性 320
  - 9.2.18 将要废弃的通知——Deprecated 属性 321
  - 9.2.19 Class 文件总结 322
- 9.3 操作字节码：走进 ASM 322
  - 9.3.1 ASM 体系结构 322
  - 9.3.2 ASM 之 Hello World 324



9.4 小结	325
第 10 章 Class 装载系统 .....	326
10.1 来去都有序：看懂 Class 文件的装载流程	326
10.1.1 类装载的条件	327
10.1.2 加载类	330
10.1.3 验证类	332
10.1.4 准备	333
10.1.5 解析类	334
10.1.6 初始化	336
10.2 一切 Class 从这里开始：掌握 ClassLoader	340
10.2.1 认识 ClassLoader，看懂类加载	341
10.2.2 ClassLoader 的分类	341
10.2.3 ClassLoader 的双亲委托模式	343
10.2.4 双亲委托模式的弊端	347
10.2.5 双亲委托模式的补充	348
10.2.6 突破双亲模式	350
10.2.7 热替换的实现	353
10.3 小结	357
第 11 章 字节码执行 .....	358
11.1 代码如何执行：字节码执行案例	359
11.2 执行的基础：Java 虚拟机常用指令介绍	369
11.2.1 常量入栈指令	369
11.2.2 局部变量压栈指令	370

- 11.2.3 出栈装入局部变量表指令 371
- 11.2.4 通用型操作 372
- 11.2.5 类型转换指令 373
- 11.2.6 运算指令 375
- 11.2.7 对象/数组操作指令 377
- 11.2.8 比较控制指令 379
- 11.2.9 函数调用与返回指令 386
- 11.2.10 同步控制 389
- 11.2.11 再看 Class 的方法结构 391
- 11.3 更上一层楼：再看 ASM 393
  - 11.3.1 为类增加安全控制 393
  - 11.3.2 统计函数执行时间 396
- 11.4 谁说 Java 太刻板：Java Agent 运行时修改类 399
  - 11.4.1 使用 -javaagent 参数启动 Java 虚拟机 400
  - 11.4.2 使用 Java Agent 为函数增加计时功能 402
  - 11.4.3 动态重转换类 404
  - 11.4.4 有关 Java Agent 的总结 407
- 11.5 与时俱进：动态函数调用 407
  - 11.5.1 方法句柄使用实例 407
  - 11.5.2 调用点使用实例 411
  - 11.5.3 反射和方法句柄 412
  - 11.5.4 指令 invokedynamic 使用实例 414
- 11.6 跑得再快点：静态编译优化 418

11.6.1	编译时计算	419
11.6.2	变量字符串的连接	421
11.6.3	基于常量的条件语句裁剪	422
11.6.4	switch 语句的优化	423
11.7	提高虚拟机的执行效率：JIT 及其相关参数	424
11.7.1	开启 JIT 编译	425
11.7.2	JIT 编译阈值	426
11.7.3	多级编译器	427
11.7.4	OSR 栈上替换	430
11.7.5	方法内联	431
11.7.6	设置代码缓存大小	432
11.8	小结	436



# 7

## 第 7 章

---

### 分析 Java 堆

内存一直是应用系统中最为重要的组成部分，在 Java 应用中，系统内存通常会被分为几块不同的空间，了解这些不同内存区域的作用有助于更好地编写 Java 应用，构建更加稳定的系统。而堆空间更是 Java 内存中最为重要的区域，几乎所有的应用程序对象都在堆中分配，当系统出现故障时，具备 Java 堆的内存分析能力，也可以更加方便地诊断系统的故障，而本章最主要的就是介绍有关 Java 堆的分析技术。

本章涉及的主要知识点有：

- 常见的内存溢出原因及其解决思路。
- 有关 `java.lang.String` 的探讨。
- 使用 Visual VM 分析堆。
- 使用 MAT 分析堆。

## 7.1 对症下药：找到内存溢出的原因

内存溢出（OutOfMemory，简称 OOM）是一个令人头痛的问题，它通常出现在某一块内存空间块耗尽的时候。在 Java 程序中，导致内存溢出的原因有很多，本节将主要讨论最常见的几种内存溢出问题，包括堆溢出、直接内存溢出、永久区溢出等。

### 7.1.1 堆溢出

堆是 Java 程序中最为重要的内存空间，由于大量的对象都直接分配在堆上，因此它也成为最有可能发生溢出的区间。一般来说，绝大部分 Java 的内存溢出都属于这种情况。其原因是因为大量对象占据了堆空间，而这些对象都持有强引用，导致无法回收，当对象大小之和大于由 Xmx 参数指定的堆空间大小时，溢出错误就自然而然地发生了。

【示例 7-1】下面这段代码就是堆溢出的典型，一个 ArrayList 对象总是持有 byte 数组的强引用，导致 byte 数据无法回收。

```
public class SimpleHeapOOM {
    public static void main(String args[]){
        ArrayList<byte[]> list=new ArrayList<byte[]>();
        for(int i=0;i<1024;i++){
            list.add(new byte[1024*1024]);
        }
    }
}
```

运行以上代码，应该会立即抛出错误：

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at geym.zbase.ch7.oom.SimpleHeapOOM.main(SimpleHeapOOM.java:14)
```

可以看到，在错误信息中注明了“java heap space”，表示这是一次堆空间的溢出。

为了缓解堆溢出错误，一方面可以使用-Xmx 参数指定一个更大的堆空间，另一方面，由于堆空间不可能无限增长，通过下文提到的 MAT 或者 Visual VM 等工具，分析找到大量占用堆空间的对象，并在应用程序上做出合理的优化也是十分必要的。

### 7.1.2 直接内存溢出

在 Java 的 NIO（New IO）中，支持直接内存的使用，也就是通过 Java 代码，获得一块堆外的内存空间，这块空间是直接向操作系统申请的。直接内存的申请速度一般要比堆内存慢，

但是其访问速度要快于堆内存。因此，对于那些可复用的，并且会被经常访问的空间，使用直接内存是可以提高系统性能的。但由于直接内存没有被 Java 虚拟机完全托管，若使用不当，也容易触发直接内存溢出，导致宕机。

**【示例 7-2】** 下面的代码不断地申请直接内存，并最终可能导致内存溢出。

```
01 public class DirectBufferOOM {
02     public static void main(String args[]){
03         for(int i=0;i<1024;i++){
04             ByteBuffer.allocateDirect(1024*1024);
05             System.out.println(i);
06 //         System.gc();
07         }
08     }
09 }
```

注意代码第 6 行，`System.gc()` 暂时被注释掉，也就是不会显式触发 GC。接着，在 Windows 平台上，使用 32 位 Java 虚拟机，根据以下参数运行上述代码：

```
-Xmx1g -XX:+PrintGCDetails
```

不用多久，程序就会因为内存溢出而退出，部分打印信息如下：

```
732
733
Exception in thread "main" java.lang.OutOfMemoryError
at sun.misc.Unsafe.allocateMemory(Native Method)
at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:127)
at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:306)
at geym.zbase.ch7.oom.DirectBufferOOM.main(DirectBufferOOM.java:14)
```

可以看到，在大约 733 次循环时，发生 `OutOfMemoryError` 错误。从堆栈可以看到，发生 OOM 时，正在进行 `DirectByteBuffer` 的分配。

**提醒：** 笔者在此使用的是 JDK 1.7u40 32 位 Java 虚拟机，如果使用 JDK 1.7 64 位虚拟机，程序是可以正常执行的，且不会出现 OOM，这是因为 32 位计算机系统对应用程序的可用最大内存有限制。以 Windows 平台为例，在 32 位系统中，进程的寻址空间为 4GB，其中 2GB 为用户空间，2GB 为系统空间，故实际可用的系统内存只有 2GB，当 Java 进程的所有内存之和（堆空间、栈空间、直接内存以及虚拟机自身所用的内存）大于 2GB 时，就会出现 OOM 的错误。

读者也许还会有一个疑问，就是在这里为什么 Java 的垃圾回收机制没有发挥作用？程序第

4 行分配的直接内存并没有被任何对象所引用，为何没有被回收呢？从程序的输入日志中也可以看到，虽然打开了`-XX:+PrintGCDetails` 开关，但是并没有一次 GC 日志，这说明在整个执行过程中，GC 并没有进行。事实上，直接内存不一定能够触发 GC（除非直接内存使用量达到了`-XX:MaxDirectMemorySize` 的设置），所以保证直接内存不溢出的方法是合理地进行 Full GC 的执行，或者设定一个系统实际可达的`-XX:MaxDirectMemorySize` 值（默认情况下等于`-Xmx` 的设置）。因此，如果系统的堆内存少有 GC 发生，而直接内存申请频繁，会比较容易导致直接内存溢出（这个问题在 32 位虚拟机上尤为明显）。

如果将上述代码中第 6 行的 `System.gc()` 的注释去掉，使显式 GC 生效，那么程序将可以正常结束，这说明 GC 可以回收直接内存。

另一个让该程序正常执行的方法是设置一个较小的堆，在不指定`-XX:MaxDirectMemorySize` 的情况下，最大可用直接内存等于`-Xmx` 的值。

```
-Xmx512m -XX:+PrintGCDetails
```

这里将最大堆限制在 512MB，而非 1GB，这种情况下，最大可用直接内存也为 512MB，操作系统可以同时为堆和直接内存提供足够的空间，当直接内存使用量达到 512MB 时，也会进行 GC 释放无用内存空间。

此外，显式设置`-XX:MaxDirectMemorySize` 也是解决这一问题的方法。只要设置一个系统实际可达的最大直接内存值，那么像这种实际上不应该触发的内存溢出就不会发生了。

综上所述，为避免直接内存溢出，在确保空间不浪费的基础上，合理得执行显式 GC，可以降低直接内存溢出的概率，设置合理的`-XX:MaxDirectMemorySize` 也可以避免意外的内存溢出发生，而设置一个较小的堆在 32 位虚拟机上可以使得更多的内存用于直接内存。

### 7.1.3 过多线程导致 OOM

由于每一个线程的开启都要占用系统内存，因此当线程数量太多时，也有可能导致 OOM。由于线程的栈空间也是在堆外分配的，因此和直接内存非常相似，如果想让系统支持更多的线程，那么应该使用一个较小的堆空间。

【示例 7-3】下面的代码对这种情况作了演示，这里使用的依然是 Windows 平台 32 位 Java 虚拟机 JDK 1.7u40。

```
public class MultiThreadOOM {
    public static class SleepThread implements Runnable{
        public void run(){
```



```
        try {
            Thread.sleep(10000000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public static void main(String args[]){
    for(int i=0;i<1500;i++){
        new Thread(new SleepThread(),"Thread"+i).start();
        System.out.println("Thread"+i+" created");
    }
}
```

上述代码试图创建 1500 个 Java 线程，使用以下参数执行这个程序：

```
-Xmx1g
```

运行结果如下：

```
Thread1125 created
Thread1126 created
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new
native thread
    at java.lang.Thread.start0(Native Method)
    at java.lang.Thread.start(Thread.java:693)
    at geym.zbase.ch7.oom.MultiThreadOOM.main(MultiThreadOOM.java:23)
```

可以看到，在线程 1126 处，系统抛出了 OOM，并且打印出了“unable to create new native thread”，表示系统创建线程的数量已经饱和，其原因是 Java 进程已经达到了可使用的内存上限。要解决这个问题，也可以从以下两方面下手：

（1）一个方法是可以尝试减少堆空间，如使用以下参数运行程序：

```
-Xmx512m
```

使用 512MB 堆空间后，操作系统就可以预留更多内存用于线程创建，因此程序可以正常执行。

（2）另一个方法是减少每一个线程所占的内存空间，使用-Xss 参数可以指定线程的栈空间。尝试以下参数：

```
-Xmx1g -Xss128k
```

这里依然使用 1GB 的堆空间，但是将线程的栈空间减少到 128KB，剩余可用的内存理应可以容纳更多的线程，因此程序也可以正常执行。

**注意：**如果减少了线程的栈空间大小，那么栈溢出的风险会相应地上升。

因此，处理这类 OOM 的思路，除了合理的减少线程总数外，减少最大堆空间、减少线程的栈空间也是可行的。

### 7.1.4 永久区溢出

永久区（Perm）是存放类元数据的区域。如果一个系统定了太多的类型，那么永久区是有可能溢出的。在 JDK 1.8 中，永久区被一块称为元数据的区域替代，但是它们的功能是类似的，都是为了保存类的元信息。

**【示例 7-4】**如果一个系统不断地产生新的类，而没有回收，那最终非常有可能导致永久区溢出。下面这段代码每次循环都生成一个新的类（注意是类，而不是对象实例）。

```
public class PermOOM {
    public static void main(String[] args) {
        try{
            for(int i=0;i<100000;i++){
                CglibBean bean = new CglibBean("geym.jvm.ch3.perm.bean"+i,new
HashMap());
            }
        }catch(Error e){
            e.printStackTrace();
        }
    }
}
```

这里使用 JDK 1.6，并使用下述参数执行上述代码：

```
-XX:MaxPermSize=5m
```

程序运行一段时间后，抛出以下异常：

```
Caused by: java.lang.OutOfMemoryError: PermGen space
at java.lang.ClassLoader.defineClass1(Native Method)
at java.lang.ClassLoader.defineClassCond(ClassLoader.java:631)
at java.lang.ClassLoader.defineClass(ClassLoader.java:615)
... 9 more
```

一般来说，要解决永久区溢出问题，可以从以下几个方面考虑：

- 增加 MaxPermSize 的值。
- 减少系统需要的类的数量。
- 使用 ClassLoader 合理地装载各个类，并定期进行回收。

### 7.1.5 GC 效率低下引起的 OOM

GC 是内存回收的关键，如果 GC 效率低下，那么系统的性能会受到严重的影响。如果系统的堆空间太小，那么 GC 所占的时间就会较多，并且回收所释放的内存就会较少。根据 GC 占用的系统时间，以及释放内存的大小，虚拟机会评估 GC 的效率，一旦虚拟机认为 GC 的效率过低，就有可能直接抛出 OOM 异常。但是，虚拟机不会对这个判定太随意，因为即使 GC 效率不高，强制中止程序还是显得有些太野蛮。一般情况下，虚拟机会检查以下几种情况：

- 花在 GC 上的时间是否超过了 98%。
- 老年代释放的内存是否小于 2%。
- eden 区释放的内存是否小于 2%。
- 是否连续最近 5 次 GC 都出现了上述几种情况（注意是同时出现，不是出现一个）。

只有满足所有条件，虚拟机才有可能抛出如下 OOM：

```
java.lang.OutOfMemoryError: GC overhead limit exceeded
```

尽管虚拟机限制的条件如此严格，但是在绝大部分场合，还是会抛出堆溢出错误。由于这个 OOM 只是起到辅助作用，帮助提示系统分配的堆可能太小，因此虚拟机并不强制一定要开启这个错误提示，可以通过关闭开关-XX:-UseGCOverheadLimit 来禁止这种 OOM 的产生。

## 7.2 无处不在的字符串：String 在虚拟机中的实现

String 字符串一直是各种编程语言的核心。字符串应用之广泛，使得每一种计算机语言都必须对其做特殊的优化和实现。在 Java 中，String 虽然不是基本数据类型，但是也享有了和基本数据类型一样的待遇。本节将主要讨论字符串在虚拟机中的实现。

### 7.2.1 String 对象的特点

在 Java 语言中，Java 的设计者对 String 对象进行了大量的优化，其主要表现在以下 3 个方面，同时这也是 String 对象的 3 个基本特点：

- 不变性。
- 针对常量池的优化。
- 类的 final 定义。

### 1. 不变性

不变性是指 String 对象一旦生成，则不能再对它进行改变。String 的这个特性可以泛化成不变（immutable）模式，即一个对象的状态在对象被创建之后就不再发生变化。不变模式的主要作用在于，当一个对象需要被多线程共享，并且访问频繁时，可以省略同步和锁等待的时间，从而大幅提高系统性能。

**注意：**不变性可以提高多线程访问的性能。因为对象不可变，因此对于所有线程都是只读的，多线程访问时，即使不加同步也不会产生数据的不一致，故减小了系统开销。

由于不变性，一些看起来像是修改的操作，实际上都是依靠产生新的字符串实现的。比如 String.substring()、String.concat() 方法，它们都没有修改原始字符串，而是产生了一个新的字符串，这一点是非常值得注意的。如果需要一个可以修改的字符串，那么需要使用 StringBuffer 或者 StringBuilder 对象。

### 2. 针对常量池的优化

针对常量池的优化指当两个 String 对象拥有相同的值时，它们只引用常量池中的同一个拷贝。当同一个字符串反复出现时，这个技术可以大幅度节省内存空间。

```
String str1=new String("abc");
String str2=new String("abc");
System.out.println(str1==str2);           //返回 false
System.out.println(str1==str2.intern());   //返回 false
System.out.println("abc"==str2.intern()); //返回 true
```

以上代码 str1 和 str2 都开辟了一块堆空间存放 String 实例，如图 7.1 所示。虽然 str1 和 str2 内容相同，但是在堆中的引用是不同的。String.intern() 返回字符串在常量池中的引用，显然它和 str1 也是不同的，但是，根据最后一行代码可以看到，String.intern() 始终和常量字符串相等，读者可以思考一下，str1.intern() 与 str2.intern() 是否相等呢？

### 3. 类的 final 定义

除以上两点外，final 类型定义也是 String 对象的重要特点。作为 final 类的 String 对象在系统中不可能有任何子类，这是对系统安全性的保护。同时，在 JDK 1.5 版本之前的环境中，使

用 `final` 定义有助于帮助虚拟机寻找机会，内联所有的 `final` 方法，从而提高系统效率。但这种优化方法在 `JDK 1.5` 以后，效果并不明显。

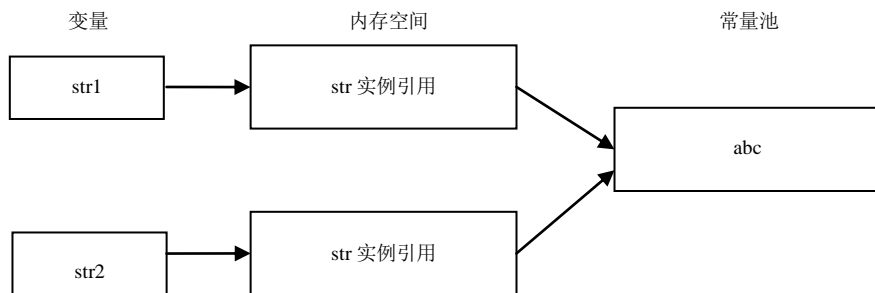


图 7.1 String 内存分配方式

## 7.2.2 有关 String 的内存泄漏

什么是内存泄漏？所谓内存泄漏，简单地说，就是由于疏忽或错误造成程序未能释放已经不再使用的内存的情况，它并不是说物理内存消失了，而是指由于不再使用的对象占据内存不被释放，而导致可用内存不断减小，最终有可能导致内存溢出。

由于垃圾回收器的出现，与传统的 C/C++ 相比，Java 已经把内存泄漏的概率大大降低了，所有不再使用的对象会由系统自动收集，但这并不意味着已经没有内存泄漏的可能。内存泄漏实际上更是一个应用问题，这里以 `String.substring()` 方法为例，说明这种内存泄漏的问题。

在 `JDK 1.6` 中，`java.lang.String` 主要由 3 部分组成：代表字符数组的 `value`、偏移量 `offset` 和长度 `count`，如图 7.2 所示。

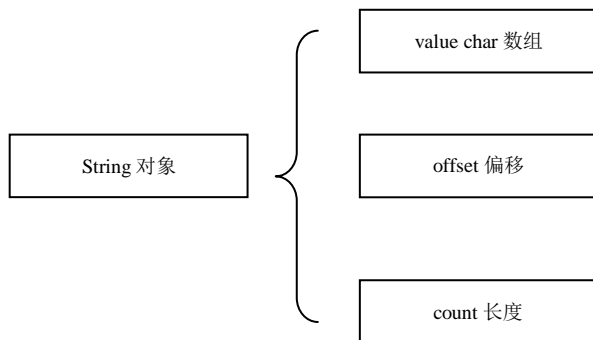


图 7.2 JDK 1.6 中 String 对象内部结构

这个结构为内存泄漏埋下了伏笔，字符串的实际内容由 `value`、`offset` 和 `count` 三者共同决定，而非 `value` 一项。试想，如果字符串 `value` 数组包含 100 个字节，而 `count` 长度只有 1 个字节，那么这个 `String` 实际上只有 1 个字符，却占据了至少 100 个字节，那剩余的 99 个就属于泄漏的部分，它们不会被使用，不会被释放，却长期占用内存，直到字符串本身被回收。如图 7.3 所示，显示了这种糟糕的情况。可以看到，`str` 的 `count` 为 1，而它的实际取值为字符串“0”，但是在 `value` 的部分，却包含了上万个字节，在这个极端情况中，原本只应该占用 1 个字节的 `String`，却占用了上万个字节，因此，可以判定为内存泄漏。

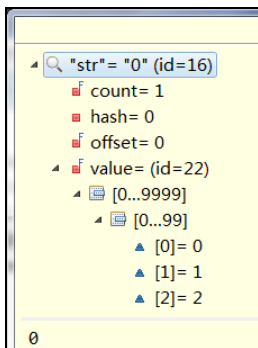


图 7.3 一个泄漏的 String

不幸的是，这种情况在 `JDK 1.6` 中非常容易出现。使用 `String.substring()` 方法就可以很容易地构造这么一个字符串。下面简单解读一下 `JDK 1.6` 中 `String.substring()` 的实现。

```
public String substring(int beginIndex, int endIndex) {
    if (beginIndex < 0) {
        throw new StringIndexOutOfBoundsException(beginIndex);
    }
    if (endIndex > count) {
        throw new StringIndexOutOfBoundsException(endIndex);
    }
    if (beginIndex > endIndex) {
        throw new StringIndexOutOfBoundsException(endIndex - beginIndex);
    }
    return ((beginIndex == 0) && (endIndex == count)) ? this :
        new String(offset + beginIndex, endIndex - beginIndex, value);
}
```

可以看到，在 `substring()` 的实现中，最终是使用了 `String` 的构造函数，生成了一个新的 `String`。该构造函数的实现如下：

```
// Package private constructor which shares value array for speed.
String(int offset, int count, char value[]) {
    this.value = value;
    this.offset = offset;
    this.count = count;
}
```

该构造函数并非公有构造函数，这点应该万幸，因为正是这个构造函数引起了内存泄漏问题。新生成的 `String` 并没有从 `value` 中获取自己需要的那部分，而是简单地使用了相同的 `value` 引用，只是修改了 `offset` 和 `count`，以此来确定新的 `String` 对象的值。当原始字符串没有被回收时，这种情况是没有问题的，并且通过共用 `value`，还可以节省一部分内存，但是一旦原始字符串被回收，`value` 中多余的部分就造成了空间浪费。

综上所述，如果使用了 `String.substring()` 将一个大字符串切割为小字符串，当大字符串被回收时，小字符串的存在就会引起内存泄漏。

所幸，这个问题已经引起官方的重视，在 `JDK 1.7` 中，对 `String` 的实现有了大幅度的调整。在新版本的 `String` 中，去掉了 `offset` 和 `count` 两项，而 `String` 的实质性内容仅仅由 `value` 决定，而 `value` 数组本身也就代表了这个 `String` 实际的取值。下面，简单地对比 `String.length()` 方法来说明这个问题，代码如下：

```
//JDK 1.7 的实现
public int length() {
    return value.length;
}

//JDK 1.6 的实现
public int length() {
    return count;
}
```

可以看到，在 `JDK 1.6` 中，`String` 的长度和 `value` 无关。基于这种改进的实现，`substring()` 方法的内存泄漏问题也得以解决，如下代码所示，展示了 `JDK 1.7` 中的 `String.substring()` 实现。

```
public String substring(int beginIndex, int endIndex) {
    //省略部分无关内容，读者自行查看代码
    int subLen = endIndex - beginIndex;
    //省略部分无关内容，读者自行查看代码
    return ((beginIndex == 0) && (endIndex == value.length)) ? this
        : new String(value, beginIndex, subLen);
}
```

```

public String(char value[], int offset, int count) {
    //省略部分无关内容, 读者自行查看代码
    // Note: offset or count might be near -1>>>1.
    if (offset > value.length - count) {
        throw new StringIndexOutOfBoundsException(offset + count);
    }
    this.value = Arrays.copyOfRange(value, offset, offset+count);
}

```

从上述代码可以看到, 在新版本的 `substring()` 中, 不再复用原 `String` 的 `value`, 而是将实际需要的部分做了复制, 该问题也得到了完全的修复。

### 7.2.3 有关 String 常量池的位置

在虚拟机中, 有一块称为常量池的区间专门用于存放字符串常量。在 `JDK 1.6` 之前, 这块区间属于永久区的一部分, 但是在 `JDK 1.7` 以后, 它就被移到了堆中进行管理。

【示例 7-5】请看下面的例子。

```

public class StringInternOOM {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        int i = 0;
        while(true){
            list.add(String.valueOf(i++).intern());
        }
    }
}

```

上述代码使用 `String.intern()` 方法获得在常量池中的字符串引用, 如果常量池中没有该常量字符串, 该方法会将字符串加入常量池。然后, 将该引用放入 `list` 进行持有, 确保不被回收。使用如下参数运行这段程序:

```
-Xmx5m -XX:MaxPermSize=5m
```

在 `JDK 1.6` 中抛出错误如下:

```

Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
    at java.lang.String.intern(Native Method)
    at geym.zbase.ch7.string.StringInternOOM.main(StringInternOOM.java:16)

```

在 `JDK 1.7` 中抛出错误如下:



```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at java.util.Arrays.copyOf(Arrays.java:2245)
at java.util.Arrays.copyOf(Arrays.java:2219)
at java.util.ArrayList.grow(ArrayList.java:242)
at java.util.ArrayList.ensureExplicitCapacity(ArrayList.java:216)
at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:208)
at java.util.ArrayList.add(ArrayList.java:440)
at geym.zbase.ch7.string.StringInternOOM.main(StringInternOOM.java:16)
```

溢出的区域已经不同，JDK 1.6 中发生在永久区，而 JDK 1.7 则发生在堆中。这间接表明了常量池的位置变化。

另外一点值得注意的是，虽然 `String.intern()` 的返回值永远等于字符串常量。但这并不代表在系统的每时每刻，相同的字符串的 `intern()` 返回都会是一样的（虽然在 95% 以上的情况下，都是相同的）。因为存在这么一种可能：在一次 `intern()` 调用之后，该字符串在某一个时刻被回收，之后，再进行一次 `intern()` 调用，那么字面量相同的字符串重新被加入常量池，但是引用位置已经不同。

```
public class ConstantPool {
    public static void main(String[] args) {
        if (args.length == 0) return;
        System.out.println(System.identityHashCode((args[0] + Integer.
toString(0))));
        System.out.println(System.identityHashCode((args[0] + Integer.
toString(0)).intern()));
        System.gc();
        System.out.println(System.identityHashCode((args[0] + Integer.
toString(0)).intern()));
    }
}
```

上述代码接收一个参数，用于构造字符串，构造的字符串都是在原有字符串后加上字符串“0”。一共输出 3 次字符串的 Hash 值：第一次为字符串本身，第二次为常量池引用，第三次为进行了常量池回收后的相同字符串的常量池引用。程序的一种可能输出如下：

```
3916375
22279806
3154093
```

可以看到，3 次 Hash 值都是不同的。但是如果不进行程序当中的显式 GC 操作，那么后两次 Hash 值理应是相同的，读者可以自行尝试。

### 7.3 虚拟机也有内窥镜：使用 MAT 分析 Java 堆

MAT 是 Memory Analyzer 的简称，它是一款功能强大的 Java 堆内存分析器。可以用于查找内存泄露以及查看内存消耗情况。MAT 是基于 Eclipse 开发的，是一款免费的性能分析工具。读者可以在 <http://www.eclipse.org/mat/> 下载并使用 MAT。

#### 7.3.1 初识 MAT

在分析堆快照前，首先需要导出应用程序的堆快照。在本书前文中提到的 jmap、JConsole 和 Visual VM 等工具都可用于获得 Java 应用程序的堆快照文件。此外，MAT 本身也具有这个功能。

如图 7.4 所示，单击“Acquire Heap Dump”菜单后，会弹出当前 Java 应用程序列表，选择要分析的应用程序即可，如图 7.5 所示。

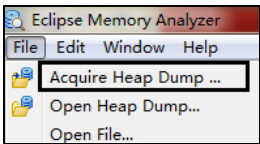


图 7.4 MAT 获取堆快照

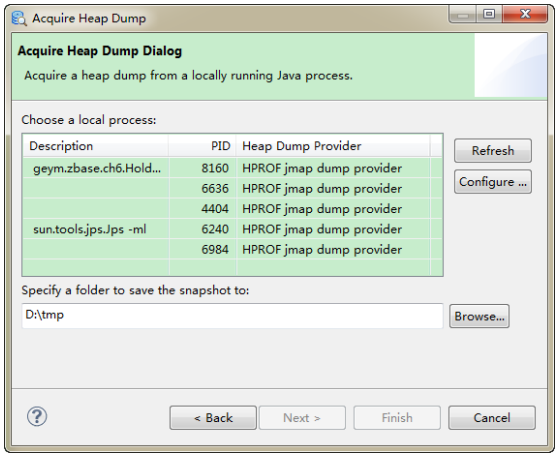


图 7.5 导出指定程序堆快照

除了直接在 MAT 中导出正在运行的应用程序堆快照外，也可以通过“Open Heap Dump”来打开一个既存的堆快照文件。

**注意：**使用 MAT 既可以打开一个已有的堆快照，也可以通过 MAT 直接从活动 Java 程序中导出堆快照。

如图 7.6 所示，显示了正常打开堆快照文件后的 MAT 的界面。

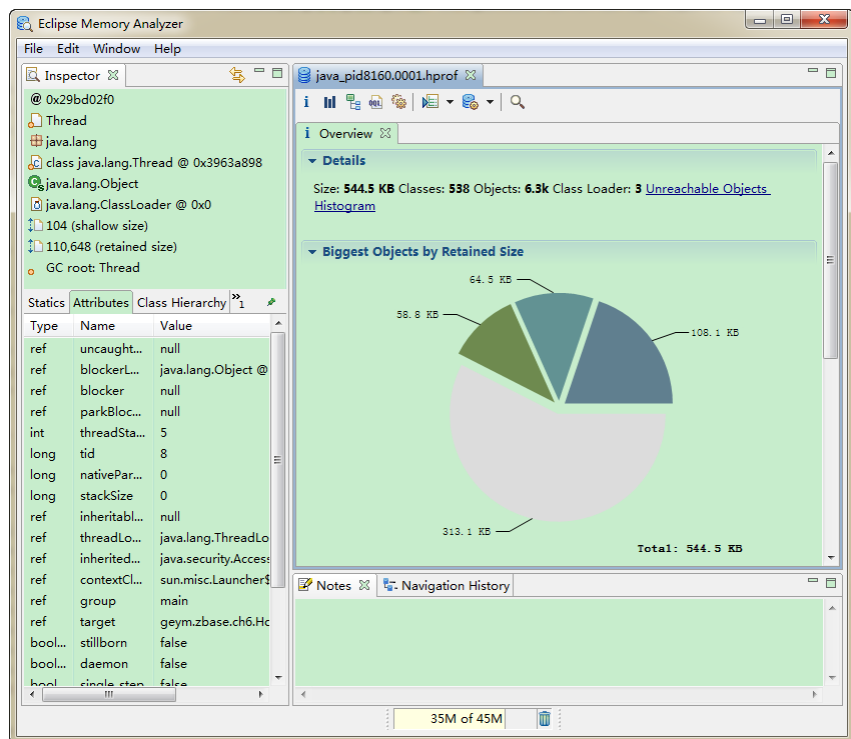


图 7.6 MAT 运行界面

右侧界面中，显示了堆快照文件的大小、类、实例和 `ClassLoader` 的总数。在右侧的饼图中，显示了当前堆快照中最大的对象。将鼠标悬停在饼图中，可以在左侧的 `Inspector` 界面中，查看该对象的相应信息。在饼图中单击某对象，可以对选中的对象进行更多的操作。

如图 7.7 所示，在工具栏上单击柱状图，可以显示系统中所有类的内存使用情况。



图 7.7 通过 MAT 工具栏查看内存使用情况

图 7.8 为系统内所有类的统计信息，包含类的实例数量和占用的空间。

另外一个实用的功能是，可以通过 MAT 查看系统中的 Java 线程，如图 7.9 所示。

当然，这里查看 Java 层面的应用线程，对于虚拟机的系统线程是无法显示的。通过线程的堆栈，还可以查看局部变量的信息。如图 7.10 所示，带有“`<local>`”标记的，就为当前帧栈的局部变量，这部分信息可能存在缺失。

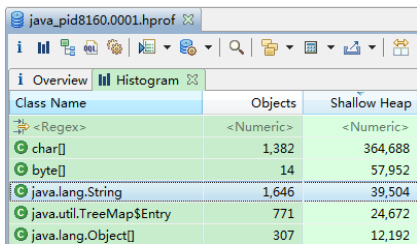


图 7.8 MAT 查看类的柱状图



图 7.9 通过 MAT 工具栏查看 Java 线程

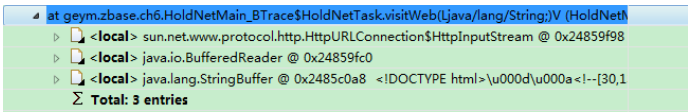


图 7.10 局部变量信息

MAT 的另外一个常用功能，是在各个对象的引用列表中穿梭查看。对于给定一个对象，通过 MAT 可以找到引用当前对象的对象，即入引用（Incomming References），以及当前对象引用的对象，即出引用（Outgoing References），如图 7.11 所示。

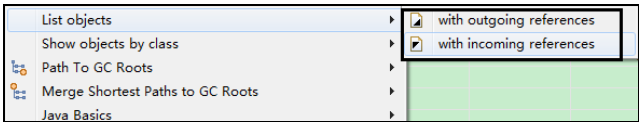


图 7.11 显示对象引用信息

图 7.12 显示了由 HttpURLConnection\$HttpInputStream 对象开始的 outgoing 引用链，可以看到，顺着该对象查找，可以依次找到 HttpURLConnection 对象和 Java.net.URL 对象，这说明在 HttpURLConnection\$HttpInputStream 对象内部引用了 HttpURLConnection，而在 HttpURLConnection 内部则引用了 Java.net.URL。

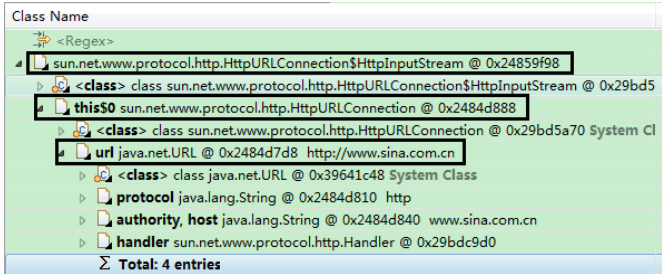


图 7.12 显示 outgoing 对象引用

7.3.2 浅堆和深堆

浅堆（Shallow Heap）和深堆（Retained Heap）是两个非常重要的概念，它们分别表示一个对象结构所占用的内存大小和一个对象被 GC 回收后，可以真实释放的内存大小。

浅堆（Shallow Heap）是指一个对象所消耗的内存。在 32 位系统中，一个对象引用会占据 4 个字节，一个 int 类型会占据 4 个字节，long 型变量会占据 8 个字节，每个对象头需要占用 8 个字节。

根据堆快照格式不同，对象的大小可能会向 8 字节进行对齐。以 String 对象为例，如图 7.13 所示，显示了 String 对象的几个属性（JDK 1.7，与 JDK 1.6 有差异）。

int	hash32	0
int	hash	0
ref	value	C:\Users\Administrati

图 7.13 JDK 1.7 中 String 结构

2 个 int 值共占 8 字节，对象引用占用 4 字节，对象头 8 字节，合计 20 字节，向 8 字节对齐，故占 24 字节。

这 24 字节为 String 对象的浅堆大小。它与 String 的 value 实际取值无关，无论字符串长度如何，浅堆大小始终是 24 字节。

深堆（Retained Heap）的概念略微复杂。要理解深堆，首先需要了解保留集（Retained Set）。对象 A 的保留集指当对象 A 被垃圾回收后，可以被释放的所有的对象集合（包括对象 A 本身），即对象 A 的保留集可以被认为是只能通过对象 A 被直接或间接访问到的所有对象的集合。通俗地说，就是指仅被对象 A 所持有的对象的集合。深堆是指对象的保留集中所有的对象的浅堆大小之和。

**注意：**浅堆指对象本身占用的内存，不包括其内部引用对象的大小。一个对象的深堆指只能通过该对象访问到的（直接或间接）所有对象的浅堆之和，即对象被回收后，可以释放的真实空间。

另外一个常用的概念是对象的实际大小。这里，对象的实际大小定义为一个对象所能触及的所有对象的浅堆大小之和，也就是通常意义上我们说的对象大小。与深堆相比，似乎这个在日常开发中更为直观和被人接受，但实际上，这个概念和垃圾回收无关。

如图 7.14 所示，显示了一个简单的对象引用关系图，对象 A 引用了 C 和 D，对象 B 引用了 C 和 E。那么对象 A 的浅堆大小只是 A 本身，不含 C 和 D，而 A 的实际大小为 A、C、D 三者之和。而 A 的深堆大小为 A 与 D 之和，由于对象 C 还可以通过对象 B 访问到，因此不在对

象 A 的深堆范围内。

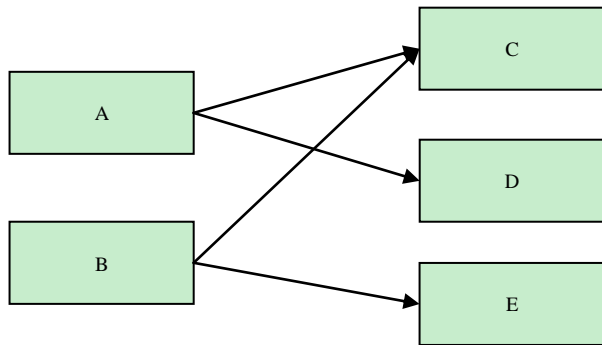


图 7.14 对象引用和深堆大小

### 7.3.3 例解 MAT 堆分析

在了解了浅堆、深堆和 MAT 的基本使用方法后，本节将通过一个简单的小案例，展示堆文件的分析方法。

**【示例 7-6】**在本案例中，设想这样一个场景：有一个学生浏览网页的记录程序，它将记录每个学生访问过的网站地址。它由三个部分组成：**Student**、**WebPage** 和 **TraceStudent** 三个类。它们的实现如下（本书使用 32 位 JDK 演示，64 位 JDK 的对象头大于 32 位 JDK，数据上存在出入，望读者留意）。

**Student 类：**

```

public class Student {
    private int id;
    private String name;
    private List<WebPage> history=new Vector<WebPage>();

    public Student(int id, String name) {
        super();
        this.id = id;
        this.name = name;
    }
    //省略 setter 和 getter 方法
}

```

**WebPage 类：**

```

public class WebPage {
    private String url;
}

```

```
private String content;
//省略 getter 和 setter
}
```

TraceStudent 类:

```
01 public class TraceStudent {
02     static List<WebPage> webpages = new Vector<WebPage>();
03     public static void createWebPages() {
04         for (int i = 0; i < 100; i++) {
05             WebPage wp = new WebPage();
06             wp.setUrl("http://www." + Integer.toString(i) + ".com");
07             wp.setContent(Integer.toString(i));
08             webpages.add(wp);
09         }
10     }
11     public static void main(String[] args) {
12         createWebPages();
13         Student st3 = new Student(3, "billy");
14         Student st5 = new Student(5, "alice");
15         Student st7 = new Student(7, "taotao");
16         for (int i = 0; i < webpages.size(); i++) {
17             if (i % st3.getId() == 0)
18                 st3.visit(webpages.get(i));
19             if (i % st5.getId() == 0)
20                 st5.visit(webpages.get(i));
21             if (i % st7.getId() == 0)
22                 st7.visit(webpages.get(i));
23         }
24         webpages.clear();
25         System.gc();
26     }
27 }
```

可以看到，在 `TraceStudent` 类中，首先创建了 100 个网址，为阅读方便，这里的网址均以数字作为域名，分别为 0~99。之后，程序创建了 3 名学生：billy、alice 和 taotao。他们分别浏览了能被 3、5、7 整除的网页。在程序运行后，3 名学生的 `history` 中应该保护他们各自访问过的网页。现在，希望在程序退出前，得到系统的堆信息，并加以分析，查看每个学生实际访问的网页地址。

使用如下参数运行程序：

```
-XX:+HeapDumpBeforeFullGC -XX:HeapDumpPath=D:/stu.hprof
```

使用 MAT 打开产生的 stu.hrof 文件。在线程视图中可以通过主线程，找到 3 名学生的引用，如图 7.15 所示，为读者阅读方便，这里已经标出了每个实例的学生名。除了对象名称外，MAT 还给出了浅堆大小和深堆大小。可以看到，所有 Student 类的浅堆统一为 24 字节，和它们持有的内容无关，而深堆大小各不相同，这和每名学生访问的网页有关。

Object / Stack Frame	Name	Shall...	Retain...
<Regex>	<Rege...	<Num...	<Num...
java.lang.Thread @ 0x24680a90	main	104	8,512
at java.lang.Runtime.gc()V (Native Method)			
at java.lang.System.gc()V (System.java:983)			
at geym.zbase.ch7.heap.TraceStudent.main([Ljava/lang/String;)V (Tr			
> <local> java.lang.String[0] @ 0x24704118		16	16
> <local> geym.zbase.ch7.heap.Student @ 0x2470dd50 billy		24	3,216
> <local> geym.zbase.ch7.heap.Student @ 0x2470dde8 alice		24	1,600
> <local> geym.zbase.ch7.heap.Student @ 0x2470de80 taotao		24	1,216
Σ Total: 4 entries			
Σ Total: 3 entries			

图 7.15 在堆中显示 3 名学生

为了获得 taotao 同学访问过的网页，可以在 taotao 的记录中通过“出引用”（Outgoing References）查找，就可以找到由 taotao 可以触及的对象，也就是他访问过的网页，如图 7.16 所示。

<Regex>	<Numeric>	<Numeric>
geym.zbase.ch7.heap.Student @ 0x2470de80	24	1,216
> <class> class geym.zbase.ch7.heap.Student @ 0x2470dce0	0	0
> name java.lang.String @ 0x2470de98 taotao	24	48
> history java.util.Vector @ 0x2470dec8	24	1,144
> <class> class java.util.Vector @ 0x3963c798 System Class	16	16
> elementData java.lang.Object[20] @ 0x2470e088	96	1,120
> <class> class java.lang.Object[] @ 0x3963c150	0	0
> [0] geym.zbase.ch7.heap.WebPage @ 0x247055e8	16	128
> <class> class geym.zbase.ch7.heap.WebPage @ 0x247	0	0
> url java.lang.String @ 0x24705a68 http://www.0.com	24	72
> content java.lang.String @ 0x24705ac0 0	24	40
Σ Total: 3 entries		
> [1] geym.zbase.ch7.heap.WebPage @ 0x247060d8	16	128
> <class> class geym.zbase.ch7.heap.WebPage @ 0x247	0	0
> url java.lang.String @ 0x24706168 http://www.7.com	24	72
> content java.lang.String @ 0x247061c0 7	24	40
Σ Total: 3 entries		
> [2] geym.zbase.ch7.heap.WebPage @ 0x24706838	16	128
> [3] geym.zbase.ch7.heap.WebPage @ 0x24706fe8	16	128
> [4] geym.zbase.ch7.heap.WebPage @ 0x247076e8	16	128
> [5] geym.zbase.ch7.heap.WebPage @ 0x24707de8	16	128
> [6] geym.zbase.ch7.heap.WebPage @ 0x24708638	16	128

图 7.16 查找 taotao 访问过的网址

可以看到，堆中完整显示了所有 taotao 同学的 history 中的网址页面(都是可以被 7 整除的网址)。

如果现在希望查看哪些同学访问了“http://www.0.com”，则可以在对应的 WebPage 对象中通过“入引用”（Incoming References）查找。如图 7.17 所示，显然这个网址被 3 名学生都访问过了。



<Regex>	<Numeric>	<Numeric>
geym.zbase.ch7.heap.WebPage @ 0x247055e8	16	128
[0] java.lang.Object[20] @ 0x2470df78	96	1,504
elementData java.util.Vector @ 0x2470de30	24	1,528
history geym.zbase.ch7.heap.Student @ 0x2470dde8	24	1,600
[0] java.lang.Object[40] @ 0x2470dfd8	176	3,120
elementData java.util.Vector @ 0x2470dd98	24	3,144
history geym.zbase.ch7.heap.Student @ 0x2470dd50	24	3,216
[0] java.lang.Object[20] @ 0x2470e088	96	1,120
elementData java.util.Vector @ 0x2470dec8	24	1,144
history geym.zbase.ch7.heap.Student @ 0x2470de80	24	1,216
Σ Total: 3 entries		

图 7.17 通过入引用查找浏览过 www.0.com 的学生

下面，在这个实例中，再来理解一下深堆的概念，如图 7.18 所示，在 taotao 同学的访问历史中，一共有 15 条数据，每一条 WebPage 占用 128 字节的空间（深堆），而 15 条数据合计共占用 1920 字节。而 history 中的 elementData 数组实际深堆大小为 1120 字节。这是因为部分网址 WebPage 既被 taotao 访问，又被其他学生访问，因此 taotao 并不是唯一可以引用到它们的对象，对于这些对象的大小，自然不应该算在 taotao 同学的深堆中。根据程序的规律，只要被 3 或者 5 整除的网址，都不应该计算在内，满足条件的网址（能被 3 和 7 整除，或者能被 5 和 7 整除）有 0、21、35、42、63、70、84 等 7 个。它们合计大小为  $7 \times 128 = 896$  字节，故 taotao 的 history 对象中的 elementData 数组的深堆大小为  $1920 - 896 + 96 = 1120$  字节。这里的 96 字节表示 elementData 数组的浅堆大小，由于 elementData 数组长度为 20（第 15~19 项为 null），每个引用 4 字节，合计  $4 \times 20 = 80$  字节，数组对象头 8 字节，数组长度占 4 字节，合计  $80 + 8 + 4 = 92$  字节，向 8 字节对齐填充后，为 96 字节。

<class> class geym.zbase.ch7.heap.Student @ 0x2470dce0	0	0
name java.lang.String @ 0x2470de98 taotao	24	48
history java.util.Vector @ 0x2470dec8	24	1,144
<class> class java.util.Vector @ 0x3963c798 System class	16	16
elementData java.lang.Object[20] @ 0x2470e088	96	1,120
<class> class java.lang.Object[] @ 0x3963c150	0	0
[0] geym.zbase.ch7.heap.WebPage @ 0x247055e8	16	128
[1] geym.zbase.ch7.heap.WebPage @ 0x247060d8	16	128
[2] geym.zbase.ch7.heap.WebPage @ 0x24706838	16	128
[3] geym.zbase.ch7.heap.WebPage @ 0x24706fe8	16	128
[4] geym.zbase.ch7.heap.WebPage @ 0x247076e8	16	128
[5] geym.zbase.ch7.heap.WebPage @ 0x24707de8	16	128
[6] geym.zbase.ch7.heap.WebPage @ 0x24708638	16	128
[7] geym.zbase.ch7.heap.WebPage @ 0x24708d38	16	128
[8] geym.zbase.ch7.heap.WebPage @ 0x24709438	16	128
[9] geym.zbase.ch7.heap.WebPage @ 0x24709b38	16	128
[10] geym.zbase.ch7.heap.WebPage @ 0x2470a238	16	128
[11] geym.zbase.ch7.heap.WebPage @ 0x2470a938	16	128
[12] geym.zbase.ch7.heap.WebPage @ 0x2470b2c8	16	128
[13] geym.zbase.ch7.heap.WebPage @ 0x2470b9c8	16	128
[14] geym.zbase.ch7.heap.WebPage @ 0x2470c0c8	16	128
Σ Total: 16 entries		

图 7.18 对象数据的深堆大小

### 7.3.4 支配树 (Dominator Tree)

MAT 提供了一个称为支配树 (Dominator Tree) 的对象图。支配树体现了对象实例间的支配关系。在对象引用图中, 所有指向对象 B 的路径都经过对象 A, 则认为对象 A 支配对象 B。如果对象 A 是离对象 B 最近的一个支配对象, 则认为对象 A 为对象 B 的直接支配者。支配树是基于对象间的引用图所建立的, 它有以下基本性质:

- 对象 A 的子树 (所有被对象 A 支配的对象集合) 表示对象 A 的保留集 (retained set), 即深堆。
- 如果对象 A 支配对象 B, 那么对象 A 的直接支配者也支配对象 B。
- 支配树的边与对象引用图的边不直接对应。

如图 7.19 所示, 左图表示对象引用图, 右图表示左图所对应的支配树。对象 A 和 B 由根对象直接支配, 由于在到对象 C 的路径中, 可以经过 A, 也可以经过 B, 因此对象 C 的直接支配者也是根对象。对象 F 与对象 D 相互引用, 因为到对象 F 的所有路径必然经过对象 D, 因此, 对象 D 是对象 F 的直接支配者。而到对象 D 的所有路径中, 必然经过对象 C, 即使是从对象 F 到对象 D 的引用, 从根节点出发, 也是经过对象 C 的, 所以, 对象 D 的直接支配者为对象 C。

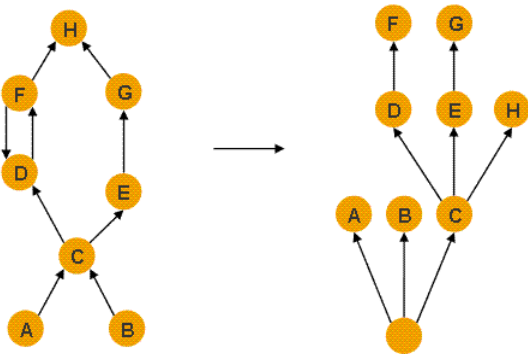


图 7.19 引用关系与支配树

同理, 对象 E 支配对象 G。到达对象 H 的可以通过对象 D, 也可以通过对象 E, 因此对象 D 和 E 都不能支配对象 H, 而经过对象 C 既可以到达 D 也可以到达 E, 因此对象 C 为对象 H 的直接支配者。

在 MAT 中, 单击工具栏上的对象支配树按钮, 可以打开对象支配树视图, 如图 7.20 所示。



图 7.20 从工具栏打开支配树

图 7.21 显示了对对象支配树视图的一部分。该截图显示部分 billy 学生的 history 队列的直接支配对象。即当 billy 对象被回收，也会一并回收的所有对象。显然能被 5 或者 7 整除的网页不会出现在该列表中，因为它们同时被另外两名学生对象引用。

geym.zbase.ch7.heap.Student @ 0x2470dd50	24	3,216	0.84%
java.util.Vector @ 0x2470dd98	24	3,144	0.82%
java.lang.Object[40] @ 0x2470dfd8	176	3,120	0.81%
geym.zbase.ch7.heap.WebPage @ 0x24705cd8	16	128	0.03%
geym.zbase.ch7.heap.WebPage @ 0x24705fd8	16	128	0.03%
geym.zbase.ch7.heap.WebPage @ 0x247062d8	16	128	0.03%
geym.zbase.ch7.heap.WebPage @ 0x24706638	16	128	0.03%
geym.zbase.ch7.heap.WebPage @ 0x24706c38	16	128	0.03%
geym.zbase.ch7.heap.WebPage @ 0x247072e8	16	128	0.03%

图 7.21 支配树显示结果

注意：对象支配树中，某一个对象的子树，表示在该对象被回收后，也将被回收的对象的集合。

### 7.3.5 Tomcat 堆溢出分析

Tomcat 是最常用的 Java Servlet 容器之一，同时也可以当做单独的 Web 服务器使用。Tomcat 本身使用 Java 实现，并运行于 Java 虚拟机之上。在大规模请求时，Tomcat 有可能会因为无法承受压力而发生内存溢出错误。本节根据一个被压垮的 Tomcat 的堆快照文件，来分析 Tomcat 在崩溃时的内部情况。

图 7.22 显示了 Tomcat 溢出时的总体信息，可以看到堆的大小为 29.7MB。从统计饼图中得知，当前深堆最大的对象为 StandardManager，它持有大约 16.4MB 的对象。

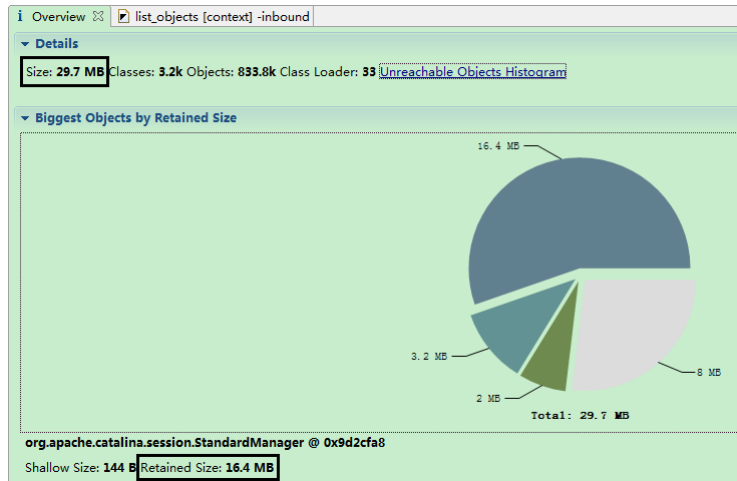


图 7.22 Tomcat 堆溢出总体信息

一般来说，我们总是会对占用空间最大的对象特别感兴趣，如果可以查看 StandardManager 内部究竟引用了哪些对象，对于分析问题可能会起到很大的帮助。因此，在饼图中单击 StandardManager 所在区域，在弹出菜单中选择“with outgoing references”命令，如图 7.23 所示。这样将会列出被 StandardManager 引用的所有对象。

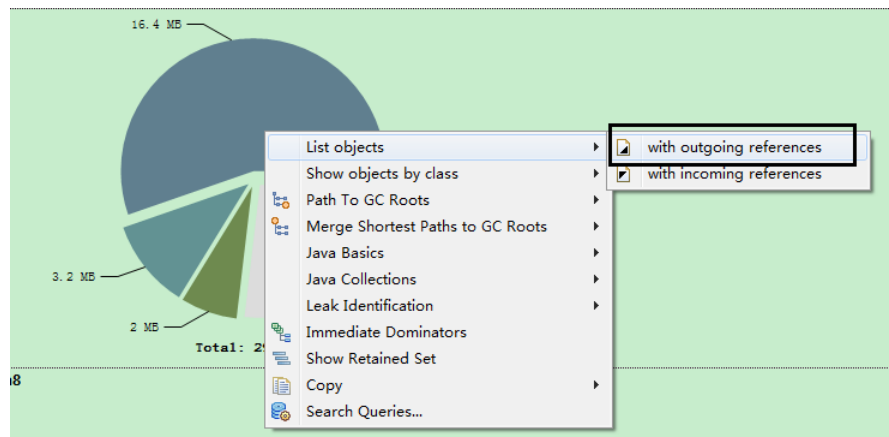


图 7.23 显示 StandardManager 的引用对象

图 7.24 显示了被 StandardManager 引用的对象，其中特别显眼的就是 sessions 对象，它占用了约 17MB 空间。

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
org.apache.catalina.session.StandardManager @ 0x9d2cfa8	144	17,211,600
sessions java.util.concurrent.ConcurrentHashMap @ 0x9d2d0f0	40	17,201,792
container org.apache.catalina.core.StandardContext @ 0x98023b0	520	311,592
mserver com.sun.jmx.mbeanserver.JmxMBeanServer @ 0x95cc6e8	32	29,832
sessionCreationTiming java.util.LinkedList @ 0x9d2d080	24	4,848
sessionExpirationTiming java.util.LinkedList @ 0x9d2d0b0	24	2,448

图 7.24 被 StandardManager 引用的 sessions

继续查找，打开 sessions 对象，查看被它引用的对象，如图 7.25 所示。可以看到 sessions 对象为 ConcurrentHashMap，其内部分为 16 个 Segment。从深堆大小看，每个 Segment 都比较平均，大约为 1MB，合计 17MB。

继续打开 Segment，查看存储在 sessions 中的真实对象。如图 7.26 所示，可以找到内部存放的为 StandardSession 对象。

<Regex>	<Numeric>	<Numeric>
org.apache.catalina.session.StandardManager @ 0x9d2cfa8	144	17,211,600
sessions java.util.concurrent.ConcurrentHashMap @ 0x9d2d0f0	40	17,201,792
segments java.util.concurrent.ConcurrentHashMap\$Segment[16] @	80	17,201,752
[3] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2d	32	1,151,440
[4] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2d	32	1,117,008
[11] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2	32	1,115,320
[8] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2d	32	1,110,160
[15] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2	32	1,099,808
[0] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2d	32	1,094,672
[7] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2d	32	1,087,800
[5] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2d	32	1,086,048
[9] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2d	32	1,080,920
[10] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2	32	1,079,200
[6] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2d	32	1,074,040
[2] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2d	32	1,070,600
[14] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2	32	1,046,488
[1] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2d	32	1,017,248
[12] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2	32	989,760
[13] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2	32	981,160
<class> class java.util.concurrent.ConcurrentHashMap\$Segment	0	0
Σ Total: 17 entries		
<class> class java.util.concurrent.ConcurrentHashMap @ 0x5756a4	32	32

图 7.25 sessions 对象的内部引用

sessions java.util.concurrent.ConcurrentHashMap @ 0x9d2d0f0	40	17,201,792
segments java.util.concurrent.ConcurrentHashMap\$Segment[16] @ 0x9d2d118	80	17,201,752
[3] java.util.concurrent.ConcurrentHashMap\$Segment @ 0x9d2d408	32	1,151,440
table java.util.concurrent.ConcurrentHashMap\$HashEntry[1024] @ 0xaadbbc8	4,112	1,151,352
[112] java.util.concurrent.ConcurrentHashMap\$HashEntry @ 0xaad9318	24	8,600
next java.util.concurrent.ConcurrentHashMap\$HashEntry @ 0xaa6d370	24	6,880
value org.apache.catalina.session.StandardSession @ 0xaad8ca0	80	1,592
key java.lang.String @ 0xaad92b0 D54FB440CBF6A221493DD7AF99767	24	104
<class> class java.util.concurrent.ConcurrentHashMap\$HashEntry @ 0x!	0	0
Σ Total: 4 entries		
[536] java.util.concurrent.ConcurrentHashMap\$HashEntry @ 0xab23a28	24	6,880
next java.util.concurrent.ConcurrentHashMap\$HashEntry @ 0xaa3dda8	24	5,160
value org.apache.catalina.session.StandardSession @ 0xab233b0	80	1,592
key java.lang.String @ 0xab239c0 019B9DA95527925E61DDE9DF9AA55	24	104
<class> class java.util.concurrent.ConcurrentHashMap\$HashEntry @ 0x!	0	0
Σ Total: 4 entries		
[953] java.util.concurrent.ConcurrentHashMap\$HashEntry @ 0xaf86958	24	6,880
next java.util.concurrent.ConcurrentHashMap\$HashEntry @ 0xadfcd0	24	5,160
value org.apache.catalina.session.StandardSession @ 0xaf862e0	80	1,592
key java.lang.String @ 0xaf868f0 2520207DD1BAD988FA48BCFD207F2f	24	104
<class> class java.util.concurrent.ConcurrentHashMap\$HashEntry @ 0x!	0	0
Σ Total: 4 entries		

图 7.26 通过 MAT 找到 StandardSession 对象

通过 OQL 命令，查找所有的 StandardSession，如图 7.27 所示（有关 OQL，请参阅 7.4 节）。可以看到当前堆中含有 9941 个 session，并且每一个 session 的深堆为 1592 字节，合计约 15MB，达到当前堆大小的 50%。由此，可以知道，当前 Tomcat 发生内存溢出的原因，极可能是由于在短期内接收大量不同客户端的请求，从而创建大量 session 导致。

The screenshot shows the Eclipse Memory Analyzer interface with the OQL query: `SELECT OBJECTS s from org.apache.catalina.session.StandardSession s`. The results are displayed in a table with three columns: Class Name, Shallow Heap, and Retained Heap.

Class Name	Shallow Heap	Retained Heap
org.apache.catalina.session.StandardSession @ 0xb28a530	80	1,592
org.apache.catalina.session.StandardSession @ 0xb289e48	80	1,592
org.apache.catalina.session.StandardSession @ 0xb289760	80	1,592
org.apache.catalina.session.StandardSession @ 0xb289078	80	1,592
org.apache.catalina.session.StandardSession @ 0xb288990	80	1,592
org.apache.catalina.session.StandardSession @ 0xb288248	80	1,592
org.apache.catalina.session.StandardSession @ 0xb287b60	80	1,592
org.apache.catalina.session.StandardSession @ 0xb287478	80	1,592
org.apache.catalina.session.StandardSession @ 0xb286bc8	80	1,592
org.apache.catalina.session.StandardSession @ 0xb2864e0	80	1,592
org.apache.catalina.session.StandardSession @ 0xb285df8	80	1,592
org.apache.catalina.session.StandardSession @ 0xb2856b0	80	1,592
org.apache.catalina.session.StandardSession @ 0xb284ff0	80	1,592
org.apache.catalina.session.StandardSession @ 0xb2848e0	80	1,592
org.apache.catalina.session.StandardSession @ 0xb2841f8	80	1,592
org.apache.catalina.session.StandardSession @ 0xb283b10	80	1,592
org.apache.catalina.session.StandardSession @ 0xb283428	80	1,592
Total: 22 of 9,963 entries 9,941 more		

图 7.27 通过 OQL 查找所有的 session 对象

为了获得更为精确的信息，可以查看每一个 session 的内部数据，如图 7.28 所示，在左侧的对象属性表中，可以看到所选中的 session 的最后访问时间和创建时间。

The screenshot shows the Eclipse Memory Analyzer interface with the Inspector view selected. The selected object is a `StandardSession` object at address `0xaad8ca0`. The left pane shows the object's class hierarchy and attributes. The right pane shows the object's internal structure, including a `segments` array and a `table` of `ConcurrentHashMap$Segment` objects.

Type	Name	Value
long	accessCount	1403324653999
ref	support	java.beans.PropertyChangeSupport @ ...
ref	principal	null
ref	notes	java.util.Hashtable @ 0xaad9238
bool...	isValid	true
bool...	isNew	false
int	maxInactiveIn...	1200
ref	manager	org.apache.catalina.session.StandardM...
ref	listeners	java.util.ArrayList @ 0xaad91e8
long	lastAccessedT...	1403324653999
ref	id	D54FB440CBF6A221493DD7AF99767182
ref	facade	org.apache.catalina.session.StandardS...
bool...	expiring	false
long	creationTime	1403324653998
ref	authType	null
ref	attributes	java.util.concurrent.ConcurrentHashMa...

图 7.28 session 的内部数据

通过 OQL 命令和 MAT 的排序功能,如图 7.29 所示,可以找到当前系统中最早创建的 session 和最后创建的 session。再根据当前的 session 总数,可以计算每秒的平均压力为:  $9941/(1403324677648-1403324645728)*1000=311$  次/秒。

由此推断,在发生 Tomcat 堆溢出时, Tomcat 在连续 30 秒的时间内,平均每秒接收了约 311 次不同客户端的请求,创建了合计 9941 个 session。

s.creationTime	s.creationTime
1,403,324,645,728	1,403,324,677,648
1,403,324,645,755	1,403,324,677,129
1,403,324,645,759	1,403,324,676,355
1,403,324,645,763	

图 7.29 查找最早和最晚创建的 session

## 7.4 筛选堆对象：MAT 对 OQL 的支持

MAT 支持一种类似于 SQL 的查询语言 OQL (Object Query Language)。OQL 使用类 SQL 语法,可以在堆中进行对象的查找和筛选。本节将主要介绍 OQL 的基本使用方法,帮助读者尽快掌握这种堆文件的查看方式。

### 7.4.1 Select 子句

在 MAT 中, Select 子句的格式与 SQL 基本一致,用于指定要显示的列。Select 子句可以使用 “\*” , 查看结果对象的引用实例 (相当于 outgoing references)。

```
SELECT * FROM java.util.Vector v
```

以上查询的输出如图 7.30 所示,在输出结果中,结果集中的每条记录都可以展开,查看各自的引用对象。

OQL 还可以指定对象的属性进行输出,下例输出所有 Vector 对象的内部数组,输出结果如图 7.31 所示。使用 “OBJECTS” 关键字,可以将返回结果集中的项以对象的形式显示。

```
SELECT OBJECTS v.elementData FROM java.util.Vector v
```



OverviewOQL

SELECT \* FROM java.util.Vector v

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.Vector @ 0x2470dec8	24	1,144
<class> class java.util.Vector @ 0x3963	16	16
elementData java.lang.Object[20] @ 0x...	96	1,120
Σ Total: 2 entries		
java.util.Vector @ 0x2470de30	24	1,528
java.util.Vector @ 0x2470dd98	24	3,144
java.util.Vector @ 0x247040c8	24	680
java.util.Vector @ 0x246f95e8	24	80
java.util.Vector @ 0x246f9508	24	768
java.util.Vector @ 0x246f6768	24	80
java.util.Vector @ 0x246f63d8	24	80
java.util.Vector @ 0x24680ce0	24	80
java.util.Vector @ 0x24680c90	24	80
Σ Total: 10 entries		

图 7.30 Select 查询返回结构

OverviewOQL

SELECT objects v.elementData FROM java.util.Vector v |

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.lang.Object[20] @ 0x2470e088	96	1,120
java.lang.Object[20] @ 0x2470df78	96	1,504
java.lang.Object[40] @ 0x2470dfd8	176	3,120
java.lang.Object[160] @ 0x2470ad38	656	656
java.lang.Object[10] @ 0x246f9600	56	56
java.lang.Object[10] @ 0x246f9520	56	744
java.lang.Object[10] @ 0x246f6780	56	56
java.lang.Object[10] @ 0x246f63f0	56	56
java.lang.Object[10] @ 0x24680cf8	56	56
java.lang.Object[10] @ 0x24680ca8	56	56
Σ Total: 10 entries		

图 7.31 指定查询属性

下例显示 String 对象的 char 数组（用于 JDK 1.7 的堆）：

```
SELECT OBJECTS s.value FROM java.lang.String s
```

在 Select 子句中，使用“AS RETAINED SET”关键字可以得到所得对象的保留集。下例得到 geym.zbase.ch7.heap.Student 对象的保留集，其结果如图 7.32 所示。

```
SELECT AS RETAINED SET * FROM geym.zbase.ch7.heap.Student
```

OverviewOQLOQL

SELECT AS RETAINED SET \* FROM geym.zbase.ch7.heap.Student

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
geym.zbase.ch7.heap.WebPage @ 0x247055e8	16	128
java.lang.String @ 0x24705a68 http://www.0.com	24	72
char[16] @ 0x24705a80 http://www.0.com	48	48
char[1] @ 0x24705ab0 0	16	16
java.lang.String @ 0x24705ac0 0	24	40
geym.zbase.ch7.heap.WebPage @ 0x24705cd8	16	128
java.lang.String @ 0x24705d68 http://www.3.com	24	72
char[16] @ 0x24705d80 http://www.3.com	48	48
char[1] @ 0x24705db0 3	16	16
java.lang.String @ 0x24705dc0 3	24	40
geym.zbase.ch7.heap.WebPage @ 0x24705ed8	16	128
java.lang.String @ 0x24705f68 http://www.5.com	24	72
char[16] @ 0x24705f80 http://www.5.com	48	48
char[1] @ 0x24705fb0 5	16	16
java.lang.String @ 0x24705fc0 5	24	40
geym.zbase.ch7.heap.WebPage @ 0x24705fd8	16	128
java.lang.String @ 0x24706068 http://www.6.com	24	72
char[16] @ 0x24706080 http://www.6.com	48	48
char[1] @ 0x247060b0 6	16	16

图 7.32 查询对象保留集



“DISTINCT”关键字用于在结果集中去除重复对象。下例的输出如图 7.33 所示，输出结果中只有一条“class java.lang.String”记录。如果没有“DISTINCT”，那么查询将为每个 String 实例输出其对应的 Class 信息。

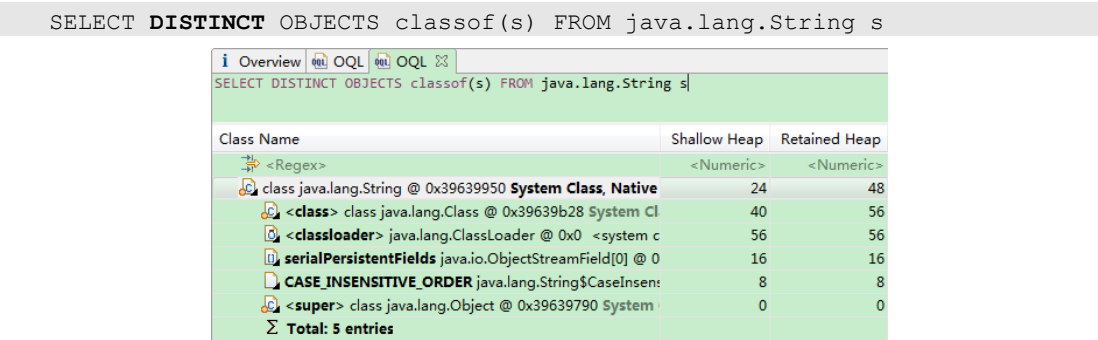
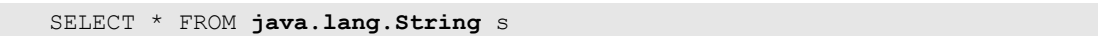


图 7.33 DISTINCT 关键字的使用

7.4.2 From 子句

From 子句用于指定查询范围，它可以指定类名、正则表达式或者对象地址。

下例使用 From 子句，指定类名进行搜索，并输出所有的 java.lang.String 实例。



下例使用正则表达式，限定搜索范围，输出所有 geym.zbase 包下所有类的实例，如图 7.34 所示。

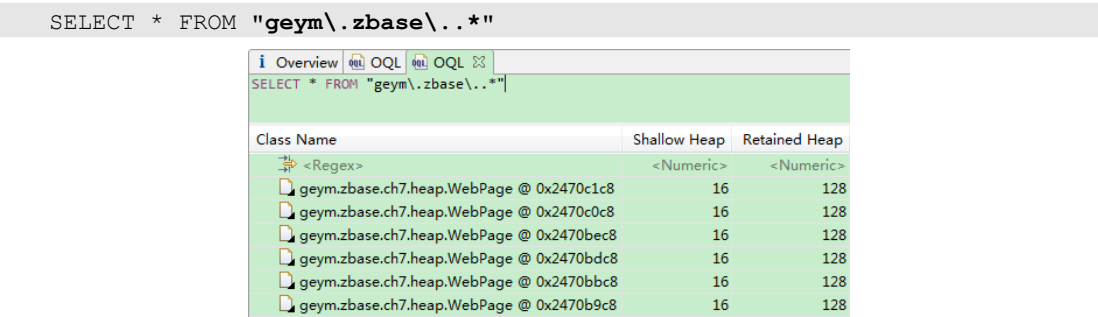


图 7.34 正则表达式查询

也可以直接使用类的地址进行搜索。使用类的地址的好处是可以区分被不同 ClassLoader 加载的同一种类型。下例中“0x37a014d8”为类的地址。

```
select * from 0x37a014d8
```

有多种方法可以获得类的地址，在 MAT 中，一种最为简单的方法如图 7.35 所示。

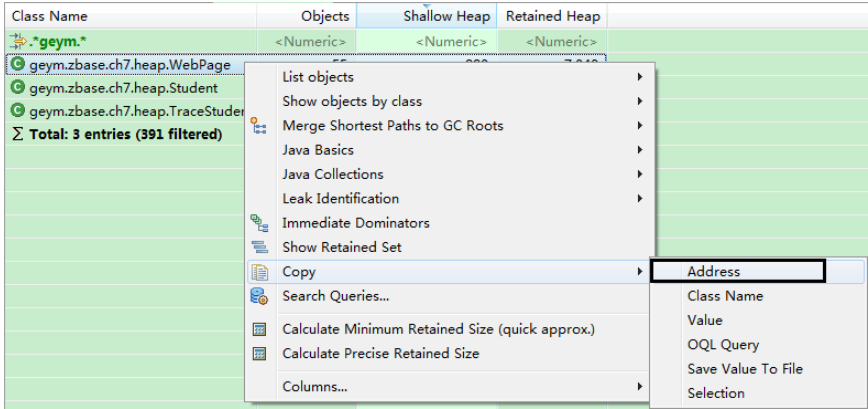


图 7.35 复制对象地址

在 From 子句中，还可以使用“INSTANCEOF”关键字，返回指定类的所有子类实例。下例的查询返回了当前堆快照中所有的抽象集合实例，包括 java.util.Vector、java.util.ArrayList 和 java.util.HashSet 等。

```
SELECT * FROM INSTANCEOF java.util.AbstractCollection
```

在 From 子句中，还可以使用“OBJECTS”关键字。使用“OBJECTS”关键字后，那么原本应该返回类的实例的查询，将返回类的信息。

```
SELECT * FROM OBJECTS java.lang.String
```

以上查询的返回结果如图 7.36 所示。它仅返回一条记录，表示 java.lang.String 的类的信息。如果不使用“OBJECTS”关键字，这个查询将返回所有的 java.lang.String 实例。

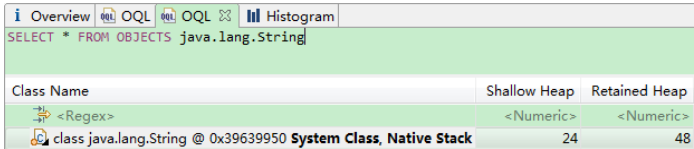
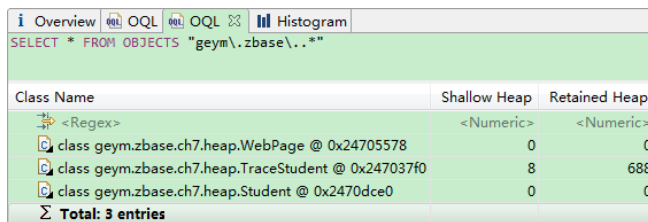


图 7.36 OBJECTS 关键字用于 FROM 子句

“OBJECTS”关键字也支持与正则表达式一起使用。下面的查询，返回了所有满足给定正则表达式的所有类，其结果如图 7.37 所示。

```
SELECT * FROM OBJECTS "geym\.zbase\.."
```



Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
class geym.zbase.ch7.heap.WebPage @ 0x24705578	0	0
class geym.zbase.ch7.heap.TraceStudent @ 0x247037f0	8	688
class geym.zbase.ch7.heap.Student @ 0x2470dce0	0	0
<b>Σ Total: 3 entries</b>		

图 7.37 OBJECTS 关键字与正则表达式结合

注意：在 From 子句中使用 OBJECTS 关键字，将返回符合条件的类信息，而非实例信息。这与 Select 子句中的 OBJECTS 关键字是完全不同的。

### 7.4.3 Where 子句

Where 子句用于指定 OQL 的查询条件。OQL 查询将只返回满足 Where 子句指定条件的对象。Where 子句的格式与传统 SQL 极为相似。

下例返回长度大于 10 的 char 数组。

```
SELECT * FROM char[] s WHERE s.@length>10
```

下例返回包含“java”子字符串的所有字符串，使用“LIKE”操作符，“LIKE”操作符的操作参数为正则表达式。

```
SELECT * FROM java.lang.String s WHERE toString(s) LIKE ".*java.*"
```

下例返回所有 value 域不为 null 的字符串，使用“=”操作符。

```
SELECT * FROM java.lang.String s where s.value!=null
```

Where 子句支持多个条件的 AND、OR 运算。下例返回数组长度大于 15，并且深堆大于 1000 字节的所有 Vector 对象。

```
SELECT * FROM java.util.Vector v WHERE v.elementData.@length>15 AND v.@retainedHeapSize>1000
```

### 7.4.4 内置对象与方法

OQL 中可以访问堆内对象的属性，也可以访问堆内代理对象的属性。访问堆内对象的属性时，格式如下：

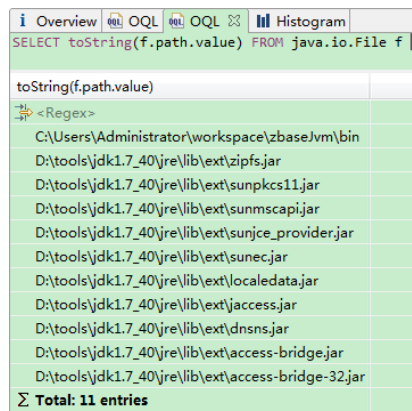
```
[ <alias>. ] <field> . <field> . <field>
```

其中 alias 为对象名称。

下例访问 java.io.File 对象的 path 属性，并进一步访问 path 的 value 属性。

```
SELECT toString(f.path.value) FROM java.io.File f
```

以上查询得到的结果如图 7.38 所示。



toString(f.path.value)
<Regex>
C:\Users\Administrator\workspace\zbase\vm\bin
D:\tools\jdk1.7_40\jre\lib\ext\zipfs.jar
D:\tools\jdk1.7_40\jre\lib\ext\sunpkcs11.jar
D:\tools\jdk1.7_40\jre\lib\ext\sunmscapi.jar
D:\tools\jdk1.7_40\jre\lib\ext\sunjce_provider.jar
D:\tools\jdk1.7_40\jre\lib\ext\sunec.jar
D:\tools\jdk1.7_40\jre\lib\ext\localedata.jar
D:\tools\jdk1.7_40\jre\lib\ext\jaccess.jar
D:\tools\jdk1.7_40\jre\lib\ext\dnsns.jar
D:\tools\jdk1.7_40\jre\lib\ext\access-bridge.jar
D:\tools\jdk1.7_40\jre\lib\ext\access-bridge-32.jar
Σ Total: 11 entries

图 7.38 显示文件信息

这些堆内对象的属性与 Java 对象一致，拥有与 Java 对象相同的结果。

MAT 为了能快速地从堆内对象的额外属性（比如对象占用的堆大小、对象地址等），为每种元类型的堆内对象建立了相对应的代理对象，以增强原有的对象功能。访问代理对象的属性时，使用如下格式：

```
[ <alias>. ] @<attribute>
```

其中，alias 为对象名称，attribute 为属性名。

下例显示了 String 对象的内容、objectid 和 objectAddress。

```
SELECT s.toString(), s.@objectId, s.@objectAddress FROM java.lang.String s
```

下例显示了 File 对象的对象 ID、对象地址、代理对象的类型、类的类型、对象的浅堆大小以及对象的显示名称。

```
SELECT f.@objectId, f.@objectAddress, f.@class, f.@clazz, f.@usedHeapSize, f.@displayName FROM java.io.File f
```

下例显示 java.util.Vector 内部数组的长度。

```
SELECT v.elementData.@length FROM java.util.Vector v
```

表 7.1 整理了 MAT 代理对象的基本属性。

表 7.1 MAT代理对象的基本属性

对象说明	对象名	对象方法/字段	对象方法/字段说明
基对象	IObejct	objectId	对象ID
		objectAddress	对象地址
		class	代理对象类型
		clazz	对象类类型
		usedHeapSize	浅堆大小
		retainedHeapSize	深堆大小
		displayName	显示名称
Class对象	IClass	classLoaderId	ClassLoad的ID
数组	IArray	length	数组长度
元类型数组	IPrimitiveArray	valueArray	数组内容
对象数组	IObjectArray	referenceArray	数组内容

除了使用代理对象的属性，OQL 中还可以使用代理对象的方法，使用格式如下：

```
[ <alias> . ] @<method>( [ <expression>, <expression> ] )
```

下例显示 int 数组中索引下标为 2 的数据内容。

```
SELECT s.getValueAt(2) FROM int[] s WHERE (s.@length > 2)
```

下例显示对象数组中索引下标为 2 的对象。

```
SELECT OBJECTS s.@referenceArray.get(2) FROM java.lang.Object[] s WHERE (s.@length > 2)
```

下例显示了当前堆中所有的类型。

```
select * from ${snapshot}.getClasses()
```

下例显示了所有的 java.util.Vector 对象及其子类型，它的输出如图 7.39 所示。

```
select * from INSTANCEOF java.util.Vector
```

下例显示当前对象是否是数组。

```
SELECT c, classof(c).isArrayType() FROM ${snapshot}.getClasses() c
```

代理对象的方法整理如表 7.2 所示。

i Overview default_report org.eclipse.mat.api:suspects OQL		
select * from INSTANCEOF java.util.Vector		
Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.Vector @ 0x2470dec8	24	1,144
java.util.Vector @ 0x2470de30	24	1,528
java.util.Vector @ 0x2470dd98	24	3,144
java.util.Vector @ 0x247040c8	24	680
java.util.Vector @ 0x246f95e8	24	80
java.util.Vector @ 0x246f9508	24	768
java.util.Vector @ 0x246f6768	24	80
java.util.Vector @ 0x246f63d8	24	80
java.util.Vector @ 0x24680ce0	24	80
java.util.Vector @ 0x24680c90	24	80
java.util.Stack @ 0x246f9940	24	80
java.util.Stack @ 0x246f6af8	24	80
java.util.Stack @ 0x24680d30	24	80
Σ Total: 13 entries		

图 7.39 getClassByName()函数使用

表 7.2 MAT代理对象的方法

对象说明	对象名	对象方法	对象方法说明
全局快照	ISnapshot	getClasses()	所有实例的集合
		getClassesByName(String name, boolean includeSubClasses)	根据名称选取符合条件的实例
类对象	IClass	hasSuperClass()	是否有超类
		isArrayType()	是否是数组
基对象	IObject	getObjectAddress()	取得对象地址
元类型数组	IPrimitiveArray	getValueAt(int index)	取得数组中给定索引的数据
元类型数组，对象数组	[] or List	get(int index)	取得数组中给定索引的数据

MAT 的 OQL 中还内置一些有用的函数，如表 7.3 所示。

表 7.3 OQL中的内置函数

函数	说明
toHex( number )	转为16进制
toString( object )	转为字符串
dominators( object )	取得直接支配对象
outbounds( object )	取得给定对象引用的对象
inbounds( object )	取得引用给定对象的对象

续表

函数	说明
<code>classof( object )</code>	取得当前对象的类
<code>dominatorof( object )</code>	取得给定对象的直接支配者

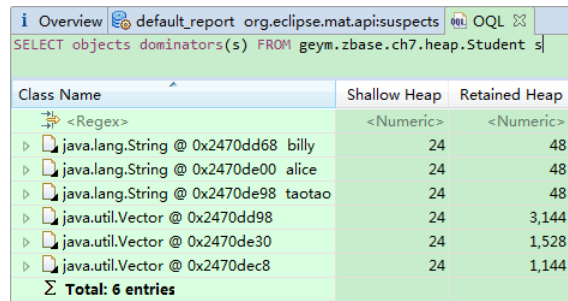
下例显示所有长度为 15 的字符串内容（JDK 1.7 导出的堆）。

```
SELECT toString(s) FROM java.lang.String s WHERE ((s.value.@length = 15) and
(s.value != null))
```

下例显示所有 `geym.zbase.ch7.heap.Student` 对象的直接支配对象。即给定对象回收后，将释放的对象集合。

```
SELECT objects dominators(s) FROM geym.zbase.ch7.heap.Student s
```

以上查询的输出如图 7.40 所示，显示 `Student` 对象支配了 3 个字符串和 3 个 `Vector` 对象。



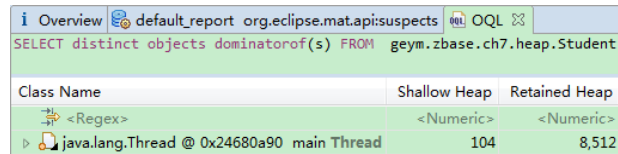
Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.lang.String @ 0x2470dd68 billy	24	48
java.lang.String @ 0x2470de00 alice	24	48
java.lang.String @ 0x2470de98 taotao	24	48
java.util.Vector @ 0x2470dd98	24	3,144
java.util.Vector @ 0x2470de30	24	1,528
java.util.Vector @ 0x2470dec8	24	1,144
Σ Total: 6 entries		

图 7.40 dominators()函数输出

函数 `dominatorof()` 与 `dominators()` 的功能相反，它获取直接支配当前对象的对象。

```
SELECT distinct objects dominatorof(s) FROM geym.zbase.ch7.heap.Student s
```

以上查询的输出如图 7.41 所示，显示所有的 `Student` 对象直接被主线程支配。



Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.lang.Thread @ 0x24680a90 main Thread	104	8,512

图 7.41 dominatorof()函数输出

注意：函数 `dominatorof()` 与 `dominators()` 的功能正好相反。`dominatorof()` 用于获得直接支配当前对象的对象，而 `dominators()` 用于获取直接支配对象。

下例取得引用 `WebPage` 的对象。

```
SELECT objects inbounds(w) FROM geym.zbase.ch7.heap.WebPage w
```

下例取得堆快照中所有在 `geym.zbase` 包中的存在对象实例的类型，其输出如图 7.42 所示。

```
SELECT distinct objects classof(obj) FROM "geym\.zbase\..*" obj
```

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
class geym.zbase.ch7.heap.Student @ 0x2470dce0	0	0
class geym.zbase.ch7.heap.WebPage @ 0x24705578	0	0
<b>Total: 2 entries</b>		

图 7.42 classof()函数输出

## 7.5 更精彩的查找：Visual VM 对 OQL 的支持

在第 6 章中，已经简单地介绍了如何通过 Visual VM 查看堆内存快照中的对象信息。但通常堆内存快照十分庞大，快照中的类数量也很多，很难通过浏览的方式找到所需的内容。为此，Visual VM 也和 MAT 一样，提供了对 OQL（对象查询语言）的支持，以方便开发人员在庞大的堆内存数据中，快速定位所需的资源。但不幸的是，MAT 的 OQL 和 Visual VM 的 OQL 在语法上很不一样，故需要对两者做独立的介绍。

### 7.5.1 Visual VM 的 OQL 基本语法

Visual VM 的 OQL 语言是一种类似于 SQL 的查询语言，它的基本语法如下：

```
select <JavaScript expression to select>
[ from [instanceof] <class name> <identifier>
[ where <JavaScript boolean expression to filter> ] ]
```

OQL 由 3 个部分组成：select 子句、from 子句和 where 子句。select 子句指定查询结果要显示的内容。from 子句指定查询范围，可指定类名，如 `java.lang.String`、`char[]`、`[Ljava.io.File;`（File 数组）。where 子句用于指定查询条件。

**注意：**对于 MAT 来说，OQL 的关键字，如 `select`、`from` 等可以使用大写，也可以使用小写，但对于 Visual VM 而言，必须统一使用小写。

select 子句和 where 子句支持使用 JavaScript 语法处理较为复杂的查询逻辑，select 子句可



以使用类似 json 的语法输出多个列。from 子句中可以使用 instanceof 关键字，将给定类的子类也包括到输出列表中。

在 Visual VM 的 OQL 中，可以直接访问对象的属性和部分方法。如下例中，直接使用了 String 对象的 count 属性，筛选出长度大于等于 100 的字符串。

```
select s from java.lang.String s where s.count >= 100 (JDK 1.6)
select s from java.lang.String s where s.value.length >= 100 (JDK 1.7)
```

选取长度大于等于 256 的 int 数组。

```
select a from int[] a where a.length >= 256
```

筛选出以 “geym” 开头的字符串。

```
select {instance: s, content: s.toString()} from java.lang.String s where
/^geym.*$/ (s.toString())
```

上例中，select 子句使用了 json 语法，指定输出两列为 String 对象以及 String.toString() 的输出。where 子句使用正则表达式，指定了符合 /^geym.\*\$/ 条件的字符串。本例的部分输出数据如下所示：

```
{
  content = geym.zbase.ch7.heap.TraceStudent,
  instance = java.lang.String#924
}

{
  content = geym.zbase.ch7.heap.TraceStudent,
  instance = java.lang.String#1280
}
```

下例筛选出所有的文件路径及文件对象，其中调用了类的 toString() 方法。

```
select {content:file.path.toString(),instance:file} from java.io.File file
```

下例使用 instanceof 关键字选取所有的 ClassLoader，包括子类。

```
select cl from instanceof java.lang.ClassLoader cl
```

## 7.5.2 内置 heap 对象

heap 对象是 Visual VM OQL 的内置对象。通过 heap 对象可以实现一些强大的 OQL 功能。heap 对象的主要方法如下。

- `forEachClass()`: 对每一个 `Class` 对象执行一个回调操作。它的使用方法类似于 `heap.forEachClass(callback)`, 其中 `callback` 为 `JavaScript` 函数。
- `findClass()`: 查找给定名称的类对象, 返回类的方法和属性如表 7.4 所示。它的调用方法类似 `heap.findClass(className)`。
- `classes()`: 返回堆快照中所有的类集合。使用方法如: `heap.classes()`。
- `objects()`: 返回堆快照中所有的对象集合。使用方法如 `heap.objects(clazz, [includeSubtypes], [filter])`, 其中 `clazz` 指定类名称, `includeSubtypes` 指定是否选出子类, `filter` 为过滤器, 指定筛选规则。 `includeSubtypes` 和 `filter` 可以省略。
- `livepaths()`: 返回指定对象的存活路径。即, 显示哪些对象直接或者间接引用了给定对象。它的使用方法如 `heap.livepaths(obj)`。
- `roots()`: 返回这个堆的根对象。使用方法如 `heap.roots()`。

表 7.4 使用findClass()返回的Class对象拥有的属性和方法

属性	方法
name: 类名称	isSubclassOf(): 是否是指定类的子类
superclass: 父类	isSuperclassOf(): 是否是指定类的父类
statics: 类的静态变量的名称和值	subclasses(): 返回所有子类
fields: 类的域信息	superclasses(): 返回所有父类

下例查找 `java.util.Vector` 类:

```
select heap.findClass("java.util.Vector")
```

查找 `java.util.Vector` 的所有父类:

```
select heap.findClass("java.util.Vector").superclasses()
```

输出结果如下:

```
java.util.AbstractList
java.util.AbstractCollection
java.lang.Object
```

查找所有在 `java.io` 包下的对象:

```
select filter(heap.classes(), "/java.io./(it.name)")
```

查找字符串“56”的引用链:

```
select heap.livepaths(s) from java.lang.String s where s.toString()='56'
```

如下是一种可能的输出结果, 其中 `java.lang.String#1600` 即字符串“56”。它显示了该字符

串被一个 `WebPage` 对象持有。

```
java.lang.String#1600->geym.zbase.ch7.heap.WebPage#57->java.lang.Object[
]#341->java.util.Vector#11->geym.zbase.ch7.heap.Student#3
```

查找这个堆的根对象：

```
select heap.roots()
```

下例查找当前堆中所有 `java.io.File` 对象实例，参数 `true` 表示 `java.io.File` 的子类也需要被显示：

```
select heap.objects("java.io.File",true)
```

下例访问了 `TraceStudent` 类的静态成员 `webpages` 对象：

```
select heap.findClass("geym.zbase.ch7.heap.TraceStudent").webpages
```

说明：本节中部分查询语句使用了 7.3.4 节中产生的堆文件，读者可以先阅读该章节。

### 7.5.3 对象函数

在 Visual VM 中，为 OQL 语言还提供了一组以对象为操作目标的内置函数。通过这些函数，可以获取目标对象的更多信息。本节主要介绍一些常用的对象函数。

#### 1. `classof()` 函数

返回给定 Java 对象的类。调用方法形如 `classof(objname)`。返回的类对象有以下属性。

- `name`：类名称。
- `superclass`：父类。
- `statics`：类的静态变量的名称和值。
- `fields`：类的域信息。

`Class` 对象拥有以下方法。

- `isSubclassOf()`：是否是指定类的子类。
- `isSuperclassOf()`：是否是指定类的父类。
- `subclasses()`：返回所有子类。
- `superclasses()`：返回所有父类。

下例将返回所有 `Vector` 类以及子类的类型：

```
select classof(v) from instanceof java.util.Vector v
```

一种可能的输出如下：

```
java.util.Vector
java.util.Vector
java.util.Stack
```

## 2. objectid()函数

objectid()函数返回对象的 ID。使用方法如 objectid(objname)。

返回所有 Vector 对象（不包含子类）的 ID：

```
select objectid(v) from java.util.Vector v
```

## 3. reachables()函数

reachables()函数返回给定对象的可达对象集合。使用方法如 reachables(obj,[exclude])。obj 为给定对象，exclude 指定忽略给定对象中的某一字段的可达引用。

下例返回 WebPage 的可达对象：

```
select {r:toHtml(reachables(s)),url:s.url.toString()} from geym.zbase.ch7.
heap.WebPage s
```

它的部分输出如下：

```
{
r = [ java.lang.String#1432, java.lang.String#1431, char[]#2026, char[]#2027, ],
url = http://www.0.com
}
{
r = [ java.lang.String#1435, java.lang.String#1434, char[]#2030, char[]#2031, ],
url = http://www.1.com
}
```

这里的返回结果是 WebPage.url 和 WebPage.content 两个字段的引用对象。如果使用过滤，要求输出结果中不包含 WebPage.content 字段的引用对象。代码如下：

```
select {r:toHtml(reachables(s,'geym.zbase.ch7.heap.WebPage.content')),url:
s.url.toString()} from geym.zbase.ch7.heap.WebPage s
```

以上查询输出如下：

```
{
r = [ java.lang.String#1431, char[]#2026, ],
url = http://www.0.com
}
{
```

```
r = [ java.lang.String#1434, char[]#2030, ],
url = http://www.1.com
}
```

可以看到，引用对象减少了一半，目前显示的对象都是通过 `WebPage.url` 字段引用得到的。

#### 4. `referrers()` 函数

`referrers()` 函数返回引用给定对象的对象集合。使用方法如：`referrers(obj)`。

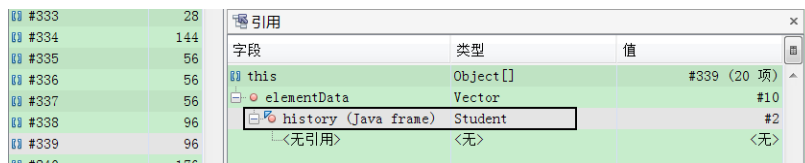
下例返回了引用表示“`http://www.15.com`”域名的 `WebPage` 对象，并且该对象本身也被其他对象引用。

```
select filter(referrers(s), 'count(referrers(it))>0') from geym.zbase.ch7.
heap.WebPage s where s.url.toString() == "http://www.15.com"
```

它的输出可能如下：

```
java.lang.Object[]#339
java.lang.Object[]#340
```

可以看到有两个数组保存着这个 `WebPage` 的引用，根据前文描述的该程序的用意，`http://www.15.com` 也确实应该被 2 名学生访问。在实例页面找到 `java.lang.Object[]#339` 对象，如图 7.43 所示。



地址	偏移量	引用
#333	28	
#334	144	
#335	56	
#336	56	
#337	56	
#338	96	
#339	96	
#340	176	

字段	类型	值
this	Object[]	#339 (20 项)
elementData	Vector	#10
history (Java frame)	Student	#2
	<无引用>	<无>

图 7.43 `WebPage` 最终被 `Student` 对象引用

下例找出长度为 2，并且至少被两个对象引用的字符串：

```
JDK 1.6 产生的堆
select s.toString() from java.lang.String s where (s.count==2 && count
(referrers(s)) >=2)

JDK 1.7 产生的堆
select s.toString() from java.lang.String s where (s.value != null && s.value.
length==2 && count(referrers(s)) >=2)
```

**注意：**`where` 子句中使用的逻辑运算符是 `&&`。这是 JavaScript 语法，不能像 SQL 一样使用 `AND` 操作符。

## 5. referees()函数

referees()函数返回给定对象的直接引用对象集合，用法形如：referees(obj)。

下例返回了 File 对象的静态成员引用：

```
select referees(heap.findClass("java.io.File"))
```

下例返回 Student 类直接引用的对象：

```
select referees(s) from geym.zbase.ch7.heap.Student s
```

上述查询的返回为：

```
java.util.Vector#9
java.lang.String#1747
java.util.Vector#10
java.lang.String#1748
java.util.Vector#11
java.lang.String#1749
```

可以看到3个 Student 对象分别持有一个 Vector 和 String 对象。其中 Vector 对象就是由 hisotry 字段持有，String 对象就是由 name 字段持有。

## 6. sizeof()函数

sizeof()函数返回指定对象的大小（不包括它的引用对象），即浅堆（Shallow Size）。

**注意：**sizeof()函数返回对象的大小不包括对象的引用对象。因此，sizeof()的返回值由对象的类型决定，和对象的具体内容无关。

下例返回所有 int 数组的大小以及对象：

```
select {size:sizeof(o),Object:o} from int[] o
```

下例返回所有 Vector 的大小以及对象：

```
select {size:sizeof(o),Object:o} from java.util.Vector o
```

它的输出可能为如下形式：

```
{
Object = java.util.Vector#1,
size = 24.0
}
{
Object = java.util.Vector#2,
```

```
size = 24.0
}
```

可以看到，不论 `Vector` 集合包含多少对象。`Vector` 对象所占用的内存大小始终为 24 字节。这是由 `Vector` 本身的结构决定的，与其内容无关。`sizeof()` 函数就是返回对象的固有大小。

## 7. `rsizeof()` 函数

`rsizeof()` 函数返回对象以及其引用对象的大小总和，即深堆（Retained Size）。这个数值不仅与类本身的结构有关，还与对象的当前数据内容有关。

下例显示了所有 `Vector` 对象的 Shallow Size 以及 Retained Size：

```
select {size:sizeof(o), rsize:rsizeof(o)} from java.util.Vector o
```

部分输出可能如下所示：

```
{
  rsize = 80.0,
  size = 24.0
}
{
  rsize = 80.0,
  size = 24.0
}
```

**注意：**`rsizeof()` 取得对象以及其引用对象的大小总和。因此，它的返回值与对象的当前数据内容有关。

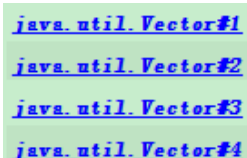
## 8. `toHtml()` 函数

`toHtml()` 函数将对象转为 HTML 显示。

下例将 `Vector` 对象的输出使用 HTML 进行加粗和斜体显示：

```
select "<b><em>" + toHtml(o) + "</em></b>" from java.util.Vector o
```

输出部分结果如图 7.44 所示。直接点击输出对象，可以展示实例页面中的对应对象。



```
java.util.Vector#1
java.util.Vector#2
java.util.Vector#3
java.util.Vector#4
```

图 7.44 `toHtml()` 函数输出

### 7.5.4 集合/统计函数

Visual VM 中还有一组用于集合操作和统计的函数。可以方便地对结果集进行后处理或者统计操作。集合/统计函数主要有 `contains()`、`count()`、`filter()`、`length()`、`map()`、`max()`、`min()`、`sort()`、`top()`等。

#### 1. `contains()`函数

`contains()` 函数判断给定集合是否包含满足给定表达式的对象。它的使用方法形如 `contains(set,boolexpression)`。其中 `set` 为给定集合，`boolexpression` 为表达式。在 `boolexpression` 中，可以使用如下 `contains()`函数的内置对象。

- `it`: 当前访问对象。
- `index`: 当前对象索引。
- `array`: 当前迭代的数组/集合。

下例返回被 `File` 对象引用的 `String` 对象集合。首先通过 `referrers(s)`得到所有引用 `String` 对象的对象集合。使用 `contains()`函数及其参数布尔等式表达式 `classof(it).name == 'java.io.File'`，将 `contains()`的筛选条件设置为类名是 `java.io.File` 的对象。

```
select s.toString() from java.lang.String s where contains(referrers(s),
"classof(it).name == 'java.io.File'")
```

以上查询的部分输出结果如下：

```
D:\tools\jdk1.7_40\jre\bin\zip.dll
D:\tools\jdk1.7_40\jre\bin\zip.dll
D:\tools\jdk1.7_40\jre\lib\ext
C:\Windows\Sun\Java\lib\ext
D:\tools\jdk1.7_40\jre\lib\ext\meta-index
```

通过该 OQL，得到了当前堆中所有的 `File` 对象的文件名称。可以理解为当前 Java 程序通过 `java.io.File` 获得已打开或持有的所有文件。

#### 2. `count()`函数

`count()`函数返回指定集合内满足给定布尔表达式的对象数量。它的基本使用方法如：`count(set, [boolexpression])`。参数 `set` 指定要统计总数的集合，`boolexpression` 为布尔条件表达式，可以省略，但如果指定，`count()`函数只计算满足表达式的对象个数。在 `boolexpression` 表达式中，可以使用以下内置对象。

- `it`: 当前访问对象。



- **index**: 当前对象索引。
- **array**: 当前迭代的数组/集合。

下例返回堆中所有 `java.io` 包中的类的数量，布尔表达式使用正则表达式表示。

```
select count(heap.classes(), "/java.io./(it.name)")
```

下列返回堆中所有类的数量。

```
select count(heap.classes())
```

### 3. filter()函数

`filter()`函数返回给定集合中，满足某一个布尔表达式的对象子集合。使用方法形如 `filter(set, boolexpression)`。在 `boolexpression` 中，可以使用以下内置对象。

- **it**: 当前访问对象。
- **index**: 当前对象索引。
- **array**: 当前迭代的数组/集合。

下例返回所有 `java.io` 包中的类。

```
select filter(heap.classes(), "/java.io./(it.name)")
```

下例返回了当前堆中，引用了 `java.io.File` 对象并且不在 `java.io` 包中的所有对象实例。首先使用 `referrers()`函数得到所有引用 `java.io.File` 对象的实例，接着使用 `filter()`函数进行过滤，只选取不在 `java.io` 包中的对象。

```
select filter(referrers(f), "! /java.io./(classof(it).name)") from java.io.File f
```

### 4. length()函数

`length()`函数返回给定集合的数量，使用方法形如 `length(set)`。

下例返回当前堆中所有类的数量。

```
select length(heap.classes())
```

### 5. map()函数

`map()`函数将结果集中的每一个元素按照特定的规则进行转换，以方便输出显示。使用方法形如：`map(set, transferCode)`。`set` 为目标集合，`transferCode` 为转换代码。在 `transferCode` 中可以使用以下内置对象。

- **it**: 当前访问对象。
- **index**: 当前对象索引。

- **array**: 当前迭代的数组/集合。

下例将当前堆中的所有 **File** 对象进行格式化输出:

```
select map(heap.objects("java.io.File"), "index + '=' + it.path.toString()")
```

输出结果为:

```
0=D:\tools\jdk1.7_40\jre\bin\zip.dll
1=D:\tools\jdk1.7_40\jre\bin\zip.dll
2=D:\tools\jdk1.7_40\jre\lib\ext
3=C:\Windows\Sun\Java\lib\ext
4=D:\tools\jdk1.7_40\jre\lib\ext\meta-index
5=D:\tools\jdk1.7_40\jre\lib\ext
```

**注意:** **map()**函数可以用于输出结果的数据格式化。它可以将集合中每一个对象转成特定的输出格式。

## 6. **max()**函数

**max()**函数计算并得到给定集合的最大元素。使用方法为: **max(set, [express])**。其中 **set** 为给定集合, **express** 为比较表达式, 指定元素间的比较逻辑。参数 **express** 可以省略, 若省略, 则执行数值比较。参数 **express** 可以使用以下内置对象。

- **lhs**: 用于比较的左侧元素。
- **rhs**: 用于比较的右侧元素。

下例显示了当前堆中最长的 **String** 长度。对于 **JDK 1.6** 得到的堆, 首先使用 **heap.objects()** 函数得到所有 **String** 对象, 接着, 使用 **map()**函数将 **String** 对象集合转为 **String** 对象的长度集合, 最后, 使用 **max()**函数得到集合中的最大元素。对于 **JDK 1.7** 得到的堆, 由于 **String** 结构发生变化, 故通过 **String.value** 得到字符串长度。

JDK 1.6 导出的堆

```
select max(map(heap.objects('java.lang.String', false), 'it.count'))
```

JDK 1.7 导出的堆

```
select max(map(filter(heap.objects('java.lang.String', false), 'it.value!=null'), 'it.value.length'))
```

以上 **OQL** 的输出为最大字符串长度, 输出如下:

```
734.0
```

下例取得当前堆的最长字符串。它在 **max()**函数中设置了比较表达式, 指定了集合中对象

的比较逻辑。

```
JDK 1.6 导出的堆
select max(heap.objects('java.lang.String'), 'lhs.count > rhs.count')

JDK 1.7 导出的堆
select max(filter(heap.objects('java.lang.String'),'it.value!=null'), 'lhs.
value.length > rhs.value.length')
```

与上例相比，它得到的是最大字符串对象，而非对象的长度：

```
java.lang.String#908
```

## 7. min()函数

min()函数计算并得到给定集合的最小元素。使用方法为：min(set, [expression])。其中 set 为给定集合，expression 为比较表达式，指定元素间的比较逻辑。参数 expression 可以省略，若省略，则执行数值比较。参数 expression 可以使用以下内置对象：

- lhs：用于比较的左侧元素
- rhs：用于比较的右侧元素

下例返回当前堆中数组长度最小的 Vector 对象的长度：

```
select min(map(heap.objects('java.util.Vector', false), 'it.elementData.
length'))
```

下例得到数组元素长度最长的一个 Vector 对象：

```
select min(heap.objects('java.util.Vector'), 'lhs.elementData.length >
rhs.elementData.length')
```

## 8. sort()函数

sort()函数对指定的集合进行排序。它的一般使用方法为：sort(set, expression)。其中，set 为给定集合，expression 为集合中对象的排序逻辑。在 expression 中可以使用以下内置对象：

- lhs：用于比较的左侧元素
- rhs：用于比较的右侧元素

下例将当前堆中的所有 Vector 按照内部数组的大小进行排序：

```
select sort(heap.objects('java.util.Vector'), 'lhs.elementData.length -
rhs.elementData.length')
```

下例将当前堆中的所有 Vector 类（包括子类），按照内部数据长度大小，从小到大排序，

并输出 `Vector` 对象的实际大小以及对象本身。

```
select map(
  sort(
    heap.objects('java.util.Vector'),
    'lhs.elementData.length - rhs.elementData.length'
  ),
  '{ size: rsizetof(it), obj: it }'
)
```

上述查询中，首先通过 `heap.objects()` 方法得到所有 `Vector` 及其子类的实例，接着，使用 `sort()` 函数，通过 `Vector` 内部数组长度进行排序，最后使用 `map()` 函数对排序后的集合进行格式化输出。

## 9. `top()` 函数

`top()` 函数返回在给定集合中，按照特定顺序排序的前几个对象。一般使用方法为：`top(set, expression, num)`。其中 `set` 为给定集合，`expression` 为排序逻辑，`num` 指定输出前几个对象。在 `expression` 中，可以使用以下内置对象。

- `lhs`：用于比较的左侧元素。
- `rhs`：用于比较的右侧元素。

下例显示了长度最长的前 5 个字符串：

JDK 1.6 的堆

```
select top(heap.objects('java.lang.String'), 'rhs.count - lhs.count', 5)
```

JDK 1.7 的堆

```
select top(filter(heap.objects('java.lang.String'),'it.value!=null'), 'rhs.
value.length - lhs.value.length', 5)
```

下例显示长度最长的 5 个字符串，输出它们的长度与对象：

JDK 1.6 的堆

```
select map(top(heap.objects('java.lang.String'), 'rhs.count - lhs.count',
5), '{ length: it.count, obj: it }')
```

JDK 1.7 的堆

```
select map(top(filter(heap.objects('java.lang.String'),'it.value!=null'),
'rhs.value.length - lhs.value.length', 5), '{ length: it.value.length, obj:
it }')
```

上述查询的部分输出可能如下所示：

```
{
length = 734.0,
obj = java.lang.String#908
}
{
length = 293.0,
obj = java.lang.String#914
}
```

## 10 . sum()函数

sum()函数用于计算集合的累计值。它的一般使用方法为：sum(set,[expression])。其中第一个参数 set 为给定集合，参数 expression 用于将当前对象映射到一个整数，以便用于求和。参数 expression 可以省略，如果省略，则可以使用 map()函数作为替代。

下例计算所有 Student 对象的可达对象的总大小：

```
select sum(map(reachables(p), 'sizeof(it)')) from geym.zbase.ch7.heap.
Student p
```

将使用 sum()函数的第 2 个参数 expression 代替 map()函数，实现相同的功能：

```
select sum(reachables(p), 'sizeof(it)') from geym.zbase.ch7.heap.Student p
```

## 11 . unique()函数

unique()函数将除去指定集合中的重复元素，返回不包含重复元素的集合。它的一般使用方法形如 unique(set)。

下例返回当前堆中，有多个不同的字符串：

```
select count(unique(map(heap.objects('java.lang.String'), 'it.value')))
```

## 7.5.5 程序化 OQL 分析 Tomcat 堆

Visual VM 不仅支持在 OQL 控制台上执行 OQL 查询语言，也可以通过其 OQL 相关的 JAR 包，将 OQL 查询程序化，从而获得更加灵活的对象查询功能，实现堆快照分析的自动化。

【示例 7-7】这里以分析 Tomcat 堆溢出文件为例，展示程序化 OQL 带来的便利。

在进行 OQL 开发前，工程需要引用 Visual VM 安装目录下 JAR 包，如图 7.45 所示。

在本示例中，加入如图 7.46 所示的 JAR 包。

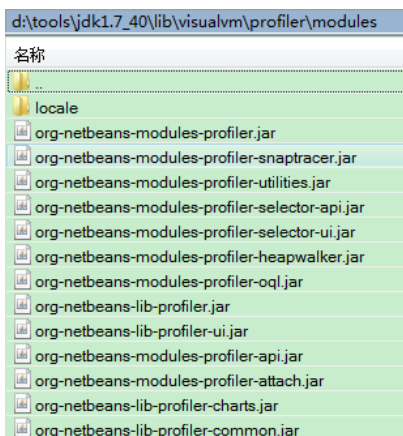


图 7.45 Visual VM 中 OQL 相关 JAR 包

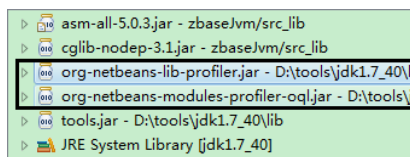


图 7.46 加入工程的 Visual VM 的 JAR 包

对于给定的 Tomcat 堆溢出 Dump 文件，这里将展示如何通过程序，计算 Tomcat 平均每秒产生的 session 数量，代码如下：

```
01 public class AveLoadTomcatOOM {
02     public static final String dumpFilePath="d:/tmp/tomcat_oom/tomcat.hprof";
03
04     public static void main(String args[]) throws Exception{
05         OQLEngine engine;
06         final List<Long> creationTimes=new ArrayList<Long>(10000);
07         engine=new OQLEngine(HeapFactory.createHeap(new File(dumpFilePath)));
08         String query="select s.creationTime from org.apache.catalina.
session.StandardSession s";
09         engine.executeQuery(query, new OQLEngine.ObjectVisitor(){
10             public boolean visit(Object obj){
11                 creationTimes.add((Long)obj);
12                 return false;
13             }
14         });
15
16         Collections.sort(creationTimes);
17
18         long min=creationTimes.get(0)/1000;
19         long max=creationTimes.get(creationTimes.size()-1)/1000;
20
21         System.out.println(" 平均压力: "+creationTimes.size()*1.0/
(max-min)+"次/秒");
```

```
22     }
23 }
```

上述代码第 8 行,通过 OQL 语句得到所有 session 的创建时间,在第 18、19 行获得所有 session 中最早创建和最晚创建的 session 时间,在第 21 行计算整个时间段内的平均 session 创建速度。

运行上述代码,得到输出如下:

```
平均压力: 311.34375 次/秒
```

使用这种方式可以做到堆转存文件的全自动化分析,并将结果导出到给定文件,当有多个堆转存文件需要分析时,有着重要的作用。

除了使用以上方式外,Visual VM 的 OQL 控制台也支持直接使用 JavaScript 代码进行编程,如下代码实现了相同功能:

```
var sessions=toArray(heap.objects("org.apache.catalina.session.StandardSession"));
var count=sessions.length;
var createtimes=new Array();
for(var i=0;i<count;i++){
    createtimes[i]=sessions[i].creationTime;
}
createtimes.sort();
var min=createtimes[0]/1000;
var max=createtimes[count-1]/1000;
count/(max-min)+"次/秒"
```

图 7.47 显示了在 OQL 控制台中,执行上述脚本以及输出结果。

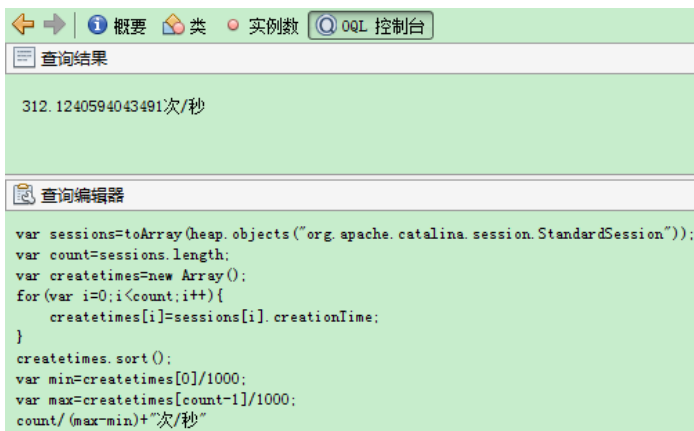


图 7.47 通过 Javascript 计算 Tomcat 中 session 的平均创建速度

细心的读者可能会发现，这个结果和使用 Java 访问 Dump 文件时的结果有所差异，这是因为 JavaScript 是弱类型语言，在处理整数除法时和 Java 有所不同，读者可以自行研究，在此不予展开讨论。

Visual VM 的 OQL 是非常灵活的，除了上述使用 JavaScript 风格外，也可以使用如下函数式编程风格计算：

```
count(heap.objects('org.apache.catalina.session.StandardSession'))/  
(  
    max(map(heap.objects('org.apache.catalina.session.StandardSession'),'it.  
creationTime'))/1000-  
    min(map(heap.objects('org.apache.catalina.session.StandardSession'),'it.  
creationTime'))/1000  
)
```

上述代码使用了 count()、min()、max()、map()等函数，共同完成了平均值的计算。执行上述代码，输出如下：

```
312.1240594043491
```

## 7.6 小结

本章主要介绍了 Java 堆的分析方法。首先，介绍了几种常见的 Java 内存溢出现象及解决思路。其次，探讨了 java.lang.String 类的特点，以及在 JDK 1.6 和 JDK 1.7 中的改进与区别。本章最重要的部分是使用 MAT 和 Visual VM 对 Java 堆进行解读和分析，对于使用 Visual VM 分析 Java 堆，本章不仅给出了基于 OQL 查询的分析策略，还提出了一种程序化的分析方法。