

# Memory Analyzer Tools

## 使用说明

zhyea.com robin

# 简介

Eclipse Memory Analyzer 是一个功能丰富且轻量的 Java 堆内存分析工具，可以用来辅助发现内存泄漏减少内存占用。

使用 Memory Analyzer 来分析生产环境的 Java 堆转储文件，可以从数以百万计的对象中快速计算出对象的 Retained Size，查看是谁在阻止垃圾回收，并自动生成一个 Leak Suspect（内存泄露可疑点）报表。

Memory Analyzer 有两种使用方式：一种是下载独立版本的 MAT，一种是使用嵌入到 Eclipse 中的 MAT 插件。我这里是用的 eclipse 插件。如果平时用的是其他 IDE，可以尝试使用独立版 MAT。

# 概念

## 1. Heap Dump

Heap Dump 是一个 Java 进程在某个时间点上的内存快照。Heap Dump 是有着多种格式的。不过总体上 Heap Dump 在触发快照的时候都保存了 java 对象和类的信息。通常在写 Heap Dump 文件前会触发一次 FullGC，所以 Heap Dump 文件中保存的是 FullGC 后留下的对象信息。

Memory Analyzer 可以用来处理 HPROF 二进制 Heap Dump 文件、IBM 系统 dump 文件（经过处理后）、以及来自各个平台上的 IBM portable Heap Dumps (PHD)文件。

一般在 Heap Dump 文件中可以获取到（这仍然取决于 Heap Dump 文件的类型）如下信息：

- 对象信息：类、成员变量、直接量以及引用值；
- 类信息：类加载器、名称、超类、静态成员；
- Garbage Collections Roots: JVM 可达的对象；
- 线程栈以及本地变量：获取快照时的线程栈信息，以及局部变量的详细信息。

Heap Dump 文件中并不包含内存分配信息，所以通常无法通过 Heap Dump 文件解决是谁以及在哪儿创建了哪些对象这样的问题。

## 2. Shallow or Retained Heap

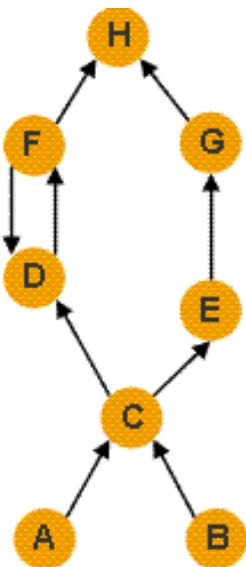
Shallow Heap 表示一个对象消费的内存的总量。对象的每个引用变量会占用 32 或 64bit（取决于操作系统），每个 Integer 需要占用 4byte，每个 Long 需要占用 8byte，诸如此类的其他信息可以自行查询。Shallow heap 的值可能是经过了调整的（比如对齐到 8，具体取决于 HeapDump 文件的格式），以便更好地模拟虚拟机的真实消费情况。

对象 X 的 Retained Set 指的是一旦 X 被垃圾回收后也会随之被 GC 回收掉的对象集合。

对象 X 的 retained heap 指的是 X 的 retained set 中所有对象的 shallow heap 之和，或者说是因为对象 X 而保持 alive 的内存的大小。

通常来说，shallow heap 就是对象自身在堆内存中的大小，而同一个对象的 retained heap 指的是该对象被垃圾回收后释放的堆内存的大小。

一组 leading 对象的 retained set（如一个特定类的全部对象、一个特定类加载器加载的所有类的全部对象、又或者一串任意的对象）在 leading 对象集中的对象全部不可达时被释放掉。Leading 对象集的 retained set 包括这些对象本身和其他的只能通过这对象访问到的对象。而 leading 对象集的 retained size 指的就是 retained set 中的全部对象的堆内存之和。如下图：



A和B是GC Roots，比如方法参数，局部对象，用在wait()、notify()和synchronized()方法上的对象等等

Leading Set	Retained Set
E	E, G
C	C, D, E, F, G, H
A, B	A, B, C, D, E, F, G, H

Minimum RetainedSize 提供了一种很好的估算 retained size 的方案。这种方案的计算速度远比获取精确的 retained size 快得多。因为这种计算方式只依赖于要查看的集合中的对象的数量，而非 Heap Dump 中对象的数量。

### 3. Dominator Tree

Memory Analyzer 提供了一个 dominator tree（支配树）的概念来描述对象关系图。将对象引用图转为 dominator tree 可以使你很容易地找到占用内存最大的一块以及对象之间的依赖关系。如下是对 dominator tree 的一些正式定义：

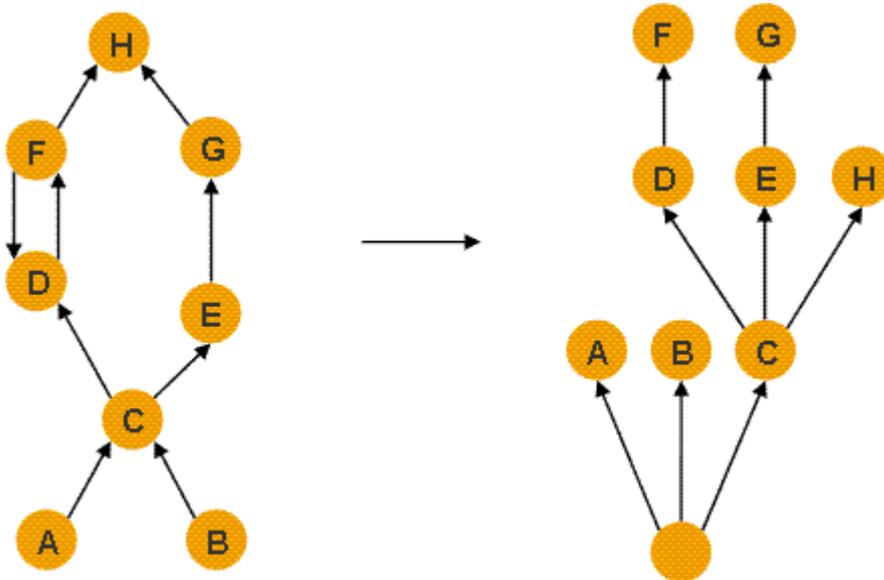
- 如果说对象 X 支配(dominate)了对象 Y，那么在对象关系图中，从起始节点（或者说是根节点）到 Y 的每一条路径都必须经过 X。
- 对象 Y 的直接支配者(immediate dominator)X 是距离对象 Y 最近的一个支配者。

Dominator Tree 基于对象关系图构建。在 Dominator Tree 中，每个对象都是它的子节点的直接支配者，所以对象间的依赖关系很容易确认。

Dominator tree 有如下几个重要的特征：

- X 的子树中的对象（也就是由 x 支配的对象）代表了 X 的 retained set；
- 如果 x 是 y 的直接支配者，那么 x 的直接支配者同样支配着 y，以此类推；
- Dominator tree 中的边并不直接对应着对象引用图中的对象引用关系。

下图是一张对象引用关系图与 dominator tree 的对照图：



### 4. Garbage Collection Roots

一个 garbage collection root（简称 GCRoot）就是一个可以从堆内存以外访问的对象。以下原因使一个对象成为 GCRoot：

## System Class

由引导类加载器（bootstrap classloader）或系统类加载器（system classloader）加载的类。比如由 rt.jar 加载的全部类，如 java.util.\*。

## JNI Local（局部 JNI 对象）

native 代码编写的局部变量，比如用户自定义的 JNI 代码或者 JVM 内部代码。

## JNI Global（全局 JNI 对象）

native 代码编写的全局变量，比如用户自定义的 JNI 代码或者 JVM 内部代码。

## Thread Block

当前活跃的 thread block 中引用的对象。

## Thread

已经启动的，仍未终止的线程。

## Busy Monitor（活跃的监控器）

所有调用了 wait()、notify()或者进入同步的对象或类（静态方法指的是类，非静态方法时是对象）。举例说如调用了 synchronized(Object)或者进入了一个同步的方法。

## Java Local

局部变量。指的是仍在线程栈中的方法的输入参数或者创建的局部对象变量。

## Native Stack

native 代码中的输入输出参数，如用户自定义的 JNI 代码，或者 JVM 内置代码。通常是在这样的情况下：一个方法中有 native 代码参与，作为这个方法的参数的对象就成了 GCRoot。比如说用在文件操作或者网络传输中的 IO 方法的参数、或者用在反射中的参数。

## Finalizable

进入等待队列，等待自己的 Finalizer 执行的对象。

## Unfinalized

一个有 finalize 方法的对象，但是尚未被 finalize 也没有进入等待执行 finalize 的队列时。

## Unreachable

其他 GCRoot 皆不可达的对象，但是也不会被 MAT 标记为 GCRoot。这种对象不会被纳入分析的范围。

## Java Stack Frame (Java 栈帧)

Stackframe (堆栈帧) 是一个为函数保留的区域，用来存储关于参数、局部变量和返回地址的信息。只有在解析 dump 文件时将之视为一个对象来处理，才会认为其是一个 GCRoot。

## Unknown

未知类型的 root 对象。一些 dump 文件，如 IBM Portable Heap Dump 文件，其中并不包含 root 信息。MAT 在解析这一类文件时，如果发现某些对象不存在任何引用信息或者对任何 GCRoot 均不可达，那么这些对象就会被认为是未知类型的 GCRoot。这样可以保证 GCRoot 持有 dump 中的所有对象信息。

## 5. Incoming & Outgoing Reference

简言之，Incoming Reference 指的是引用当前对象的外部对象；Outgoing Reference 指的是当前对象引

用的外部对象。

对象的 incoming reference 保证对象处于 alive 从而免于被垃圾回收掉。

Outgoing reference 则展示了对象的具体内容，有助于我们发现对象的用处。

## 基础教程

这里的基础教程可以用来作为熟悉 Eclipse Memory Analyzer 的一个出发点。

### 第一步：获取 Heap Dump 文件

Memory Analyzer 主要基于 Heap Dump 文件工作。Heap Dump 文件中包含了在某个时间点上所有 alive 状态的 java 对象的信息。现在所有 Java 虚拟机都有写 Heap Dump 文件的功能，但是具体的步骤随着提供商、版本以及操作系统的不同而有些差别。

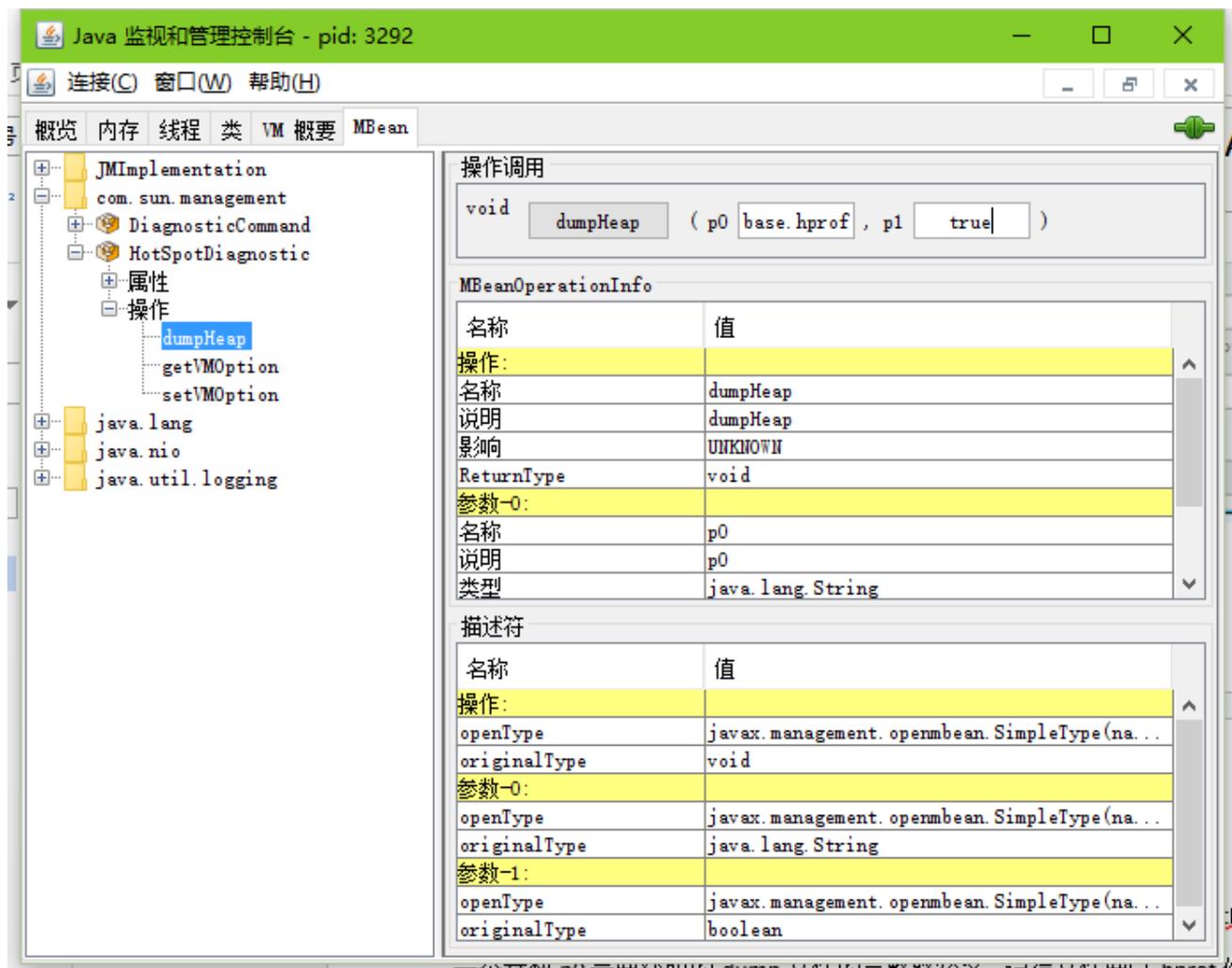
在这个教程中会尝试使用 windows 下的 java8 和 jconsole 来下载 Heap Dump 文件。使用 java8 启动一个应用，打开<JAVA\_HOME>/bin/jconsole.exe，在 jconsole 中选择正在运行的应用



(在这个例子中选择的是 eclipse，虽然进程名称为空，但是通过重启 eclipse 可以确认)。

建立连接后，选择页签 MBean，执行 com.sun.management.HotSpotDiagnostic 下的操作 dumpHeap。

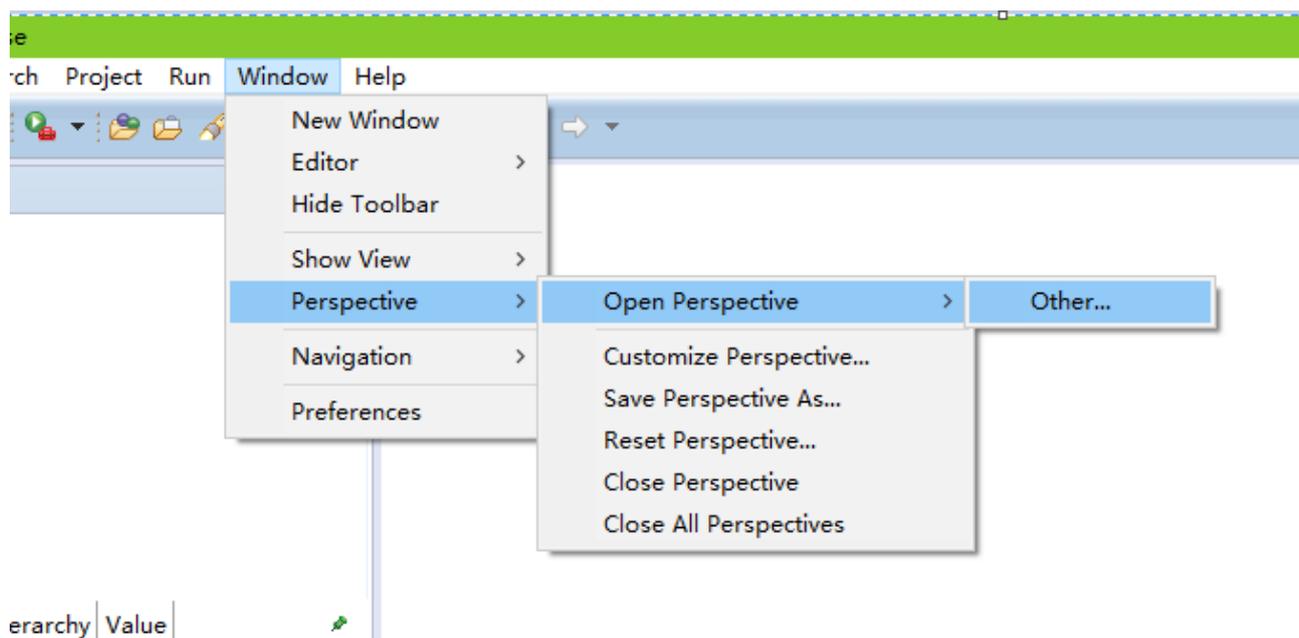
第一个参数 `p0` 是要获取的 dump 文件的完整路径名，记得文件要以 `.hprof` 作为扩展名（要在 Memory AnalysisPerspective 下打开扩展名必须是这个）。如果我们只想获取 live 的对象，第二个参数 `p1` 需要保持为 `true`。



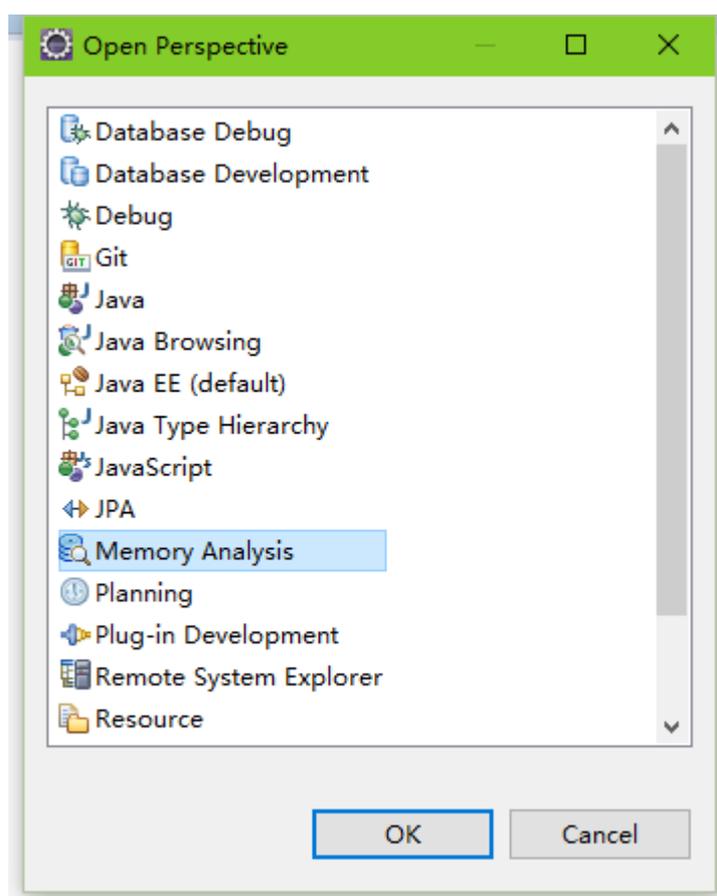
建议将导出的 dump 文件保存到一个独立的文件夹，在接下来的分析中会通过这个文件创建很多图表文件。

## 第二步：Overview（概览）

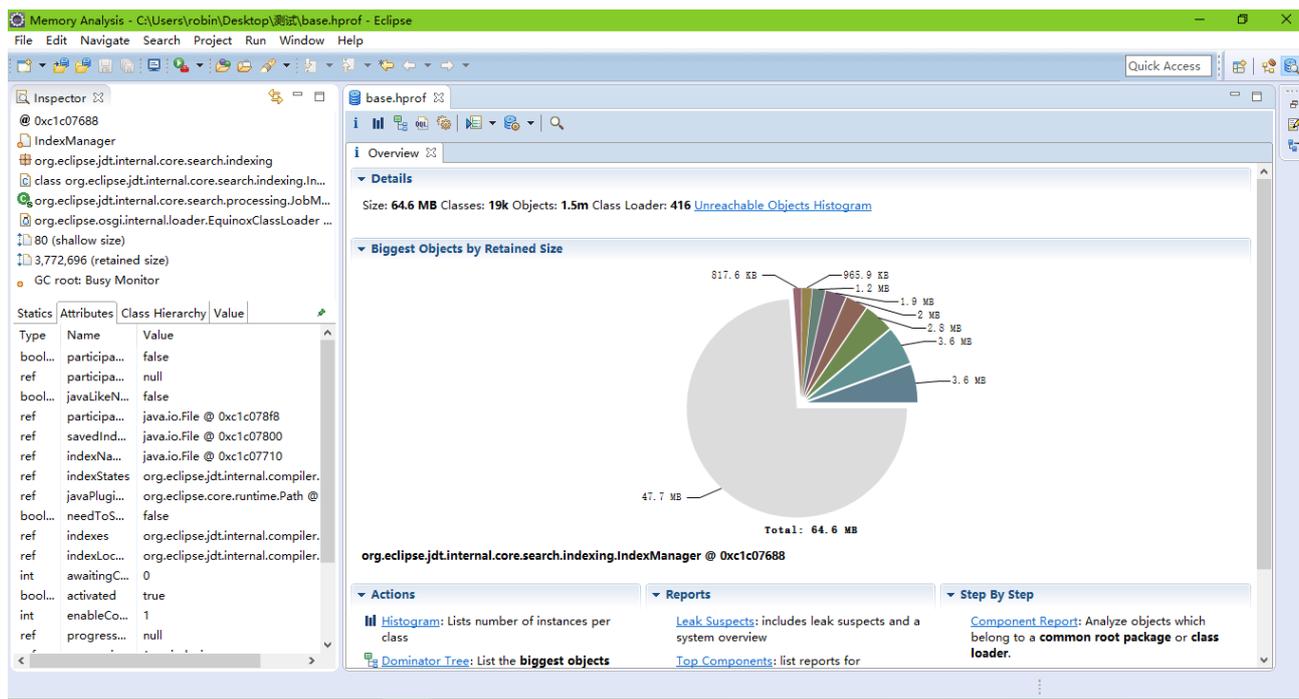
在 eclipse 上通过 File -> OpenFile 来打开 dump 文件，之后会直接进入 Overview 页。当然也可以打开 Memory AnalysisPerspective，这个通常需要手动调整下：



之后在弹出框中选择 Memory Analysis 即可：



在 MemoryAnalysisPerspective 下还可以使用 File -> Open Heap Dump 打开 dump 文件。在 Memory AnalysisPerspective 下 Overview 是这样子的：



在右侧窗口上方的位置可以看到 heapDump 的 size，以及类、对象和类加载器的数量。

右侧窗口中最醒目的饼图直观地显示了 dump 中最大的几个对象。鼠标光标划过饼图中代表某个对象的区块时可以在左侧 Inspector 窗口中看到对象的细节，在区块上点击鼠标左键可以通过菜单项钻取到关于其对应的对象更多的细节。

### 第三步：The Histogram（直方图）

在工具栏中选择 histogram 可以查看每个类的实例的数量，以及 shallow size 和 retained size。

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
char[]	184,380	18,864,024	>= 18,864,024
java.util.HashMap\$Node	208,867	6,683,744	>= 20,584,224
java.lang.String	155,732	3,737,568	>= 17,952,008
java.util.HashMap\$Node[]	57,081	3,643,368	>= 23,309,536
java.util.HashMap	72,077	3,459,696	>= 25,339,616
int[]	29,626	3,426,408	>= 3,426,408
byte[]	79,211	3,409,152	>= 3,409,152
java.lang.Object[]	39,185	2,102,320	>= 16,787,000
java.util.Collections\$UnmodifiableMap	60,776	1,944,832	>= 6,729,000

Memory Analyzer 默认展示了每个对象的 retained size。然而同一组对象(在这里指的是同一个类的所有

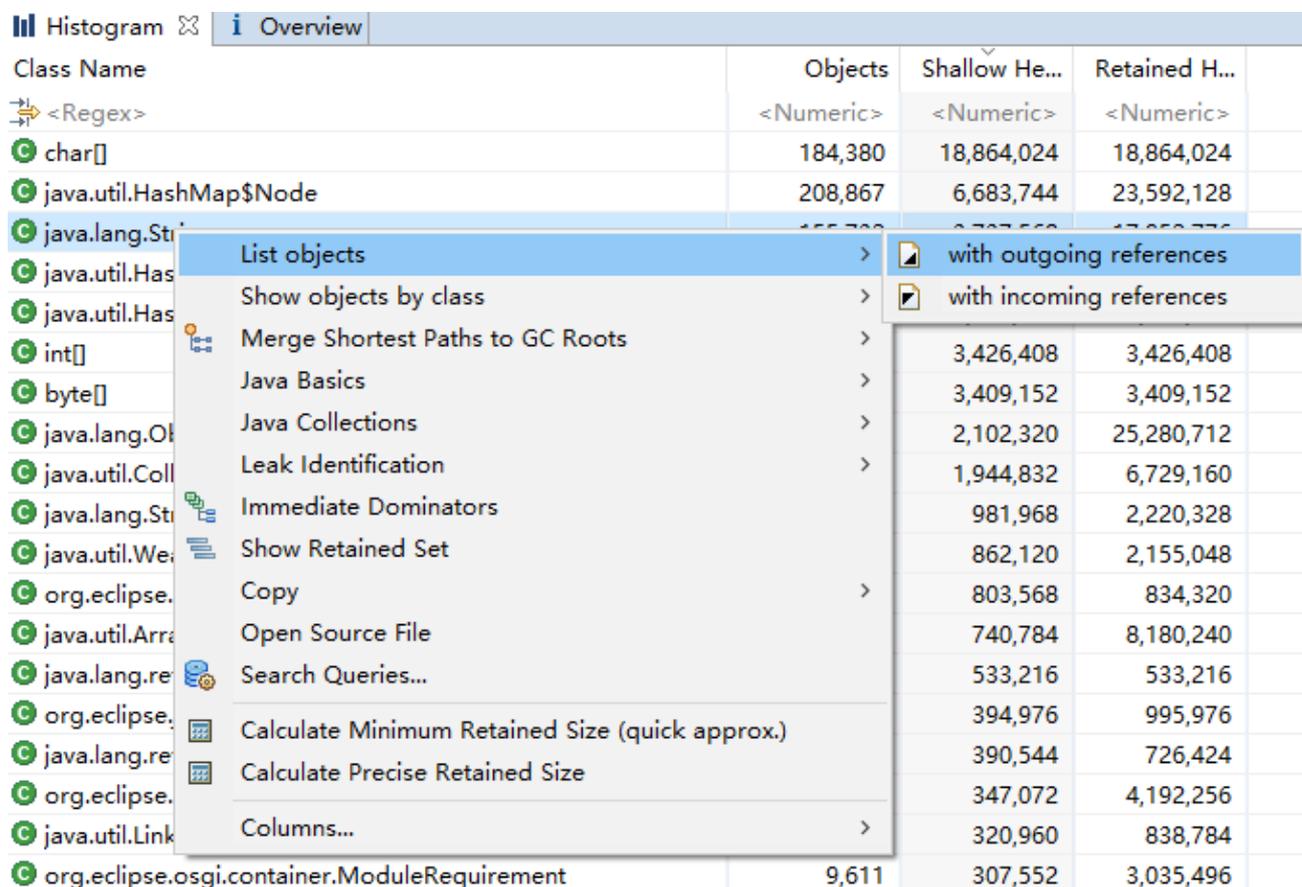
实例)的 retained size 仍然需要计算。

要估算所有行的 retained size，可以使用工具栏中的 。此外，也可以单独选择几行，使用右键菜单中的对应选项来进行计算（如果 retained heap 已经有值的话计算将不会起作用）。

在 Histogram 中使用右键菜单可以深入钻取选定行所代表的对象的信息。比如说，可以使用右键菜单查看对象的 incoming reference 和 Outgoing reference，可以按对象值的属性进行分组，也可按集合的 size 对之进行分组...

Memory Analyzer 的一个强大之处就在于使用者可以从多个角度钻取对象信息进行分析。所以在需要的时候，可以尝试对象进行切片和钻取。

查看对象的 Outgoing reference:



Class Name	Objects	Shallow He...	Retained H...
<Regex>	<Numeric>	<Numeric>	<Numeric>
char[]	184,380	18,864,024	18,864,024
java.util.HashMap\$Node	208,867	6,683,744	23,592,128
java.lang.Str...	155,700	2,707,568	17,050,376
java.util.Has...			
java.util.Has...			
int[]		3,426,408	3,426,408
byte[]		3,409,152	3,409,152
java.lang.Obj...		2,102,320	25,280,712
java.util.Coll...		1,944,832	6,729,160
java.lang.Str...		981,968	2,220,328
java.util.We...		862,120	2,155,048
org.eclipse...		803,568	834,320
java.util.Arr...		740,784	8,180,240
java.lang.re...		533,216	533,216
org.eclipse...		394,976	995,976
java.lang.re...		390,544	726,424
org.eclipse...		347,072	4,192,256
java.util.Link...		320,960	838,784
org.eclipse.osgi.container.ModuleRequirement	9,611	307,552	3,035,496

Context Menu for 'List objects':

- with outgoing references
- with incoming references

另一个重要的功能是按 classloader、package 或者 superclass 对 histograme 进行分组:

Class Name	Objects	Shallow He...	Retained H...
<Regex>	<Numeric>	<Numeric>	<Numeric>
char[]	80	18,864,024	18,864,024
java.util.HashMap\$Node	208,867	6,683,744	23,592,128
java.lang.String	155,732	3,737,568	17,952,776
java.util.HashMap\$Node[]	57,081	3,643,368	26,004,624

很多不错的应用都采用了由不同的类加载器加载不同组件的方案。Memory Analyzer 会为每个 Classloader 绑定一个有意义的标签，如果使用的是 OSGI bundles，那么这个标签就是 bundle id。通过这个，我们可以更容易将 HeapDump 做更细化的拆分。

Class Loader / Class	Objects	Shallow Heap	Retained Heap
> org.eclipse.equinox.registry	21,095	1,048,016	
> org.eclipse.emf.common	17,593	683,632	
> org.eclipse.ui.workbench	13,516	485,616	
> org.eclipse.jdt.ui	25,979	448,112	
> org.eclipse.e4.ui.model.workbench	7,855	440,344	
> org.eclipse.swt	4,942	226,824	
> org.eclipse.e4.core.contexts	6,023	205,456	
> org.eclipse.equinox.common	7,677	203,744	
> org.eclipse.jdt.core	3,642	170,512	
> org.eclipse.jface	6,150	157,592	
> org.eclipse.core.commands	3,497	153,072	
> org.eclipse.e4.core.di	2,998	95,920	
> org.eclipse.e4.ui.css.swt	2,110	89,432	
> org.eclipse.equinox.preferences	1,229	80,128	
> org.eclipse.core.expressions	2,954	80,112	
> org.eclipse.m2e.maven.indexer	1,544	71,448	
> org.eclipse.e4.ui.css.core	2,357	56,072	
> org.eclipse.core.resources	912	31,152	
> javax.inject	1,913	30,608	
> org.eclipse.e4.ui.workbench.renderers.swt	659	23,688	
> org.eclipse.wst.common.project.facet.core	715	23,176	
> org.eclipse.equinox.p2.metadata	864	21,904	
> org.eclipse.ui.navigator	524	20,408	
> org.eclipse.m2e.archetype.common	443	19,576	
<b>Σ Total: 28 of 387 entries; 359 more</b>	<b>1,459,436</b>	<b>67,709,208</b>	

按 package 对 histogram 分组可以有层次的分析 java 的 package:

Package / Class	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
> java	833,086	29,869,952	
char[]	184,380	18,864,024	18,864,024
v org	260,765	9,618,376	
> eclipse	252,454	9,279,776	
> apache	5,899	249,784	
> codehaus	1,276	44,312	
v osgi	988	40,096	
> framework	720	28,800	
v util	241	10,560	
> tracker	241	10,560	
> service	27	736	
> resource	0	0	
> dto	0	0	
Σ Total: 5 entries			
> xml	87	2,576	
> w3c	48	1,536	
> sonatype	8	200	
> slf4j	5	96	
> omg	0	0	
> jdom	0	0	
> dom4j	0	0	
Σ Total: 11 entries			

按 superclass 对 histogram 进行分组提供了一种更简单的方式来做诸如找到 java.util.AbstractMap 所有子类这样的工作：

Superclass / Class	Objects	Shallow Heap	Retained Heap
char[]	184,380	18,864,024	18,864,024
> java.util.HashMap\$Node	218,565	7,098,448	
▼ java.util.AbstractMap	79,260	3,861,824	
> java.util.HashMap	77,525	3,764,792	
java.util.concurrent.ConcurrentHashMap	761	48,704	1,523,376
▼ java.util.WeakHashMap	595	28,608	
java.util.WeakHashMap	586	28,128	2,405,784
org.eclipse.emf.ecore.impl.ETypeParameterImpl	4	224	1,000
org.eclipse.emf.ecore.util.FeatureMapUtil\$Basic	2	112	1,720
com.sun.jmx.mbeanserver.MBeanIntrospector\$	1	48	1,104
com.sun.jmx.mbeanserver.MBeanIntrospector\$	1	48	3,520
com.sun.jmx.mbeanserver.DefaultMXBeanMap	1	48	4,224
java.lang.ClassValue\$ClassValueMap	0	0	16
<b>Σ Total: 7 entries</b>			
> java.util.TreeMap	235	11,280	
com.google.common.cache.LocalCache	32	4,096	220,096
java.util.IdentityHashMap	37	1,480	46,592
java.util.EnumMap	20	800	14,016
java.util.Collections\$SingletonMap	20	800	1,912
org.eclipse.sisu.wire.EntryMapAdapter	18	432	3,600
com.google.common.cache.LocalCache	2	256	731,472
org.eclipse.sisu.wire.MergedProperties	7	224	616
com.google.common.collect.HashBiMap	3	144	976
> sun.util.PreHashedMap	3	120	
> org.apache.commons.collections.map.AbstractHa	1	64	

## 第四步：Dominator Tree

Dominator Tree 展示了 HeapDump 中最大的几个对象。如果 dominator tree 中对象的父节点被移除的话那么，那么相应对象及其后代节点也面临被回收的状态。

如果想探究一个对象持有了哪些对象并使之处于 alive，Dominator Tree 会是个很有用的工具。此外还可以在 Dominator Tree 上按照 classloader 或 package 进行分组，从而简化分析的过程。

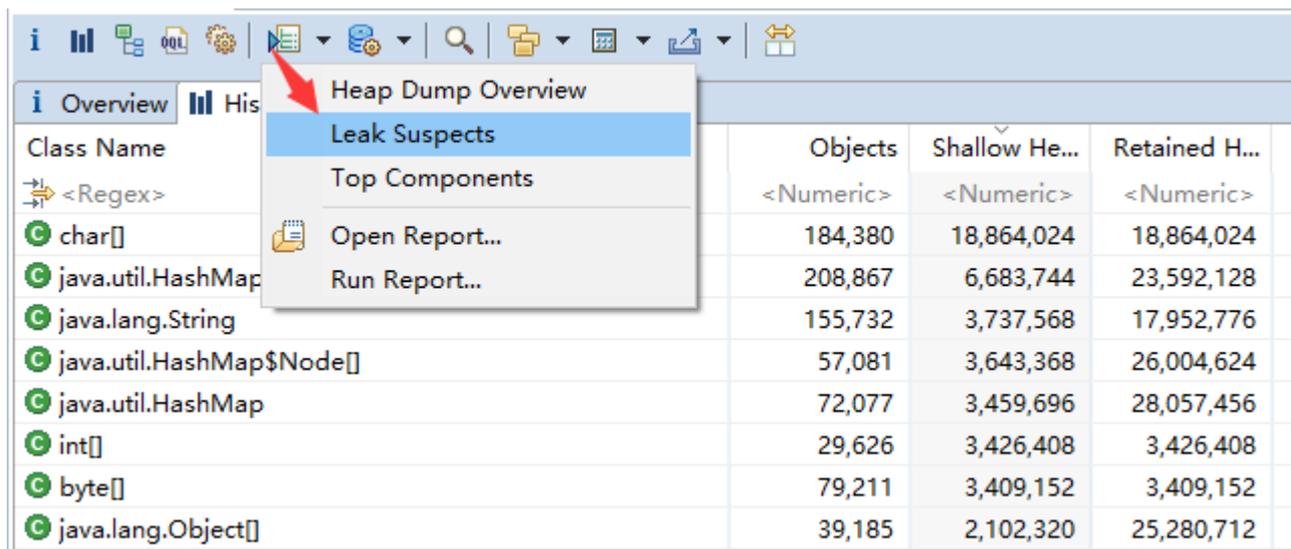
Class Name	Shallow Heap	Retained Heap	Percentage
> org.eclipse.osgi.internal.loader.EquinoxClassLoader @ 0xc1380d60 com.i	96	2,024,976	2.99%
> org.eclipse.equinox.launcher.Main\$StartupClassLoader @ 0xc0000000 Eq	88	1,236,376	1.83%
> org.eclipse.e4.ui.internal.workbench.E4XMIResource @ 0xc16cbe50	128	989,104	1.46%
> org.eclipse.e4.core.internal.contexts.EclipseContext @ 0xc211f9d0	56	837,264	1.24%
> org.eclipse.jdt.internal.core.JavaModelManager @ 0xc1c06fb8	200	818,784	1.21%
> org.eclipse.e4.core.internal.contexts.EclipseContext @ 0xc16b61e8	56	795,616	1.18%
> org.eclipse.e4.ui.css.swt.engine.CSSSWTEngineImpl @ 0xc216b1e8	80	779,160	1.15%
> org.eclipse.m2e.core.internal.project.registry.ProjectRegistryManager @ 0	48	759,840	1.12%
> org.eclipse.osgi.internal.loader.EquinoxClassLoader @ 0xc37b3a20 org.e	96	623,912	0.92%
> org.eclipse.core.internal.registry.RegistryObjectManager @ 0xc0d4e7e0	64	616,536	0.91%
> org.eclipse.osgi.internal.loader.EquinoxClassLoader @ 0xc1749c80 org.e	96	579,560	0.86%
> org.eclipse.sisu.inject.RankedSequence\$Content @ 0xc2dfe948	24	577,160	0.85%
> org.eclipse.e4.core.internal.di.InjectorImpl @ 0xc170c550	48	569,880	0.84%
> org.eclipse.sisu.inject.DefaultBeanLocator @ 0xc2c85ad8	32	496,952	0.73%
> sun.net.www.protocol.jar.URLJarFile @ 0xc086aa08	80	492,664	0.73%
> org.eclipse.e4.core.internal.contexts.osgi.EclipseContextOSGi @ 0xc170ad	64	483,056	0.71%
> org.eclipse.osgi.internal.loader.EquinoxClassLoader @ 0xc21b5550 org.e	96	469,448	0.69%
> com.google.inject.internal.InjectorImpl @ 0xc2a61ba0	56	465,208	0.69%
> com.google.inject.internal.InjectorImpl @ 0xc322c528	56	460,728	0.68%
> org.eclipse.osgi.internal.loader.EquinoxClassLoader @ 0xc28e3d50 org.a	96	425,624	0.63%
> org.eclipse.osgi.internal.loader.EquinoxClassLoader @ 0xc2300a70 org.e	96	424,800	0.63%
> com.ibm.icu.impl.Trie2_32 @ 0xc1fc7d88	64	375,360	0.55%
> org.eclipse.jst.javaee.ejb.internal.impl.EjbPackageImpl @ 0xc30db9b8	344	347,392	0.51%
> org.eclipse.osgi.internal.loader.EquinoxClassLoader @ 0xc2902d98 org.e	96	338,048	0.50%
Σ Total: 28 of 145,002 entries; 144,974 more			

## 第五步：到 GC Roots 的路径

Garbage Collections Roots (GC Roots) 是被虚拟机本身直接持有的对象。其中包括当前正在运行的线程对象、正在调用栈中的对象、以及被系统类加载器加载的类的信息。

从一个对象到一个 GCRoot 的 (倒序) 引用链——即到 GCRoot 的路径——说明了为什么对象不会被回收掉。通过引用路径可以帮助发现解决大部分 Java 中的内存泄露问题：内存泄漏发生的原因就是即使程序逻辑中某个对象不会再被已经访问了，但是该对象的引用仍然存在。





The screenshot shows the Memory Analyzer interface. A context menu is open over the 'Leak Suspects' column header, with a red arrow pointing to it. The menu options are: 'Heap Dump Overview', 'Leak Suspects' (highlighted), 'Top Components', 'Open Report...', and 'Run Report...'. The background table displays memory analysis data for various classes.

Class Name	Objects	Shallow He...	Retained H...
<Regex>	<Numeric>	<Numeric>	<Numeric>
char[]	184,380	18,864,024	18,864,024
java.util.HashMap	208,867	6,683,744	23,592,128
java.lang.String	155,732	3,737,568	17,952,776
java.util.HashMap\$Node[]	57,081	3,643,368	26,004,624
java.util.HashMap	72,077	3,459,696	28,057,456
int[]	29,626	3,426,408	3,426,408
byte[]	79,211	3,409,152	3,409,152
java.lang.Object[]	39,185	2,102,320	25,280,712

# 任务

## 1. 获取 Heap Dump 文件

### HPROF 二进制 Heap Dump 文件

获取 HPROF 文件有三种方式。

#### 1. 通过 `OutOfMemoryError` 获取 Heap Dump

通过设置如下的 JVM 参数，可以在发生 `OutOfMemoryError` 后获取到一份 HPROF 二进制 Heap Dump 文件：

**`-XX:+HeapDumpOnOutOfMemoryError`**

生成的文件会直接写入到工作目录。

这个方案适用于如下版本的虚拟机：Sun JVM (1.4.2\_12 or higher and 1.5.0\_07 or higher), HP-UX JVM (1.4.2\_11 or higher) and SAP JVM (since 1.5.0)。

#### 2. 主动触发一次 Heap Dump

也可以为虚拟机设置下面的参数，这样就可以在需要的时候按下 `CTRL+BREAK` 组合键随时获取一份 Heap Dump 文件。

**`-XX:+HeapDumpOnCtrlBreak`**

#### 3. 使用 HPROFagent

使用 `agent` 可以在程序执行结束时或者收到 `SIGQUIT` 信号时生成 `dump` 文件。使用 `agent` 也是要配置虚拟机参数的：

**`-agentlib:hprof=heap=dump,format=b`**

此外也可以选择下面的方法获取 `dump` 文件：

- 使用 `jmap`: `jmap -dump:format=b,file=<filename.hprof> <pid>`;
- 使用 `jconsole`，在上一节基础教程中有提过；
- 使用 `Memory Analyzer`，稍后会演示如何做。

## 从 IBM 虚拟机中获取 system dump 和 Heap Dump

Memory Analyzer 也可以从 IBM 的 system dumps 和 Portable Heap Dump (PHD)文件中读取内存相关的信息。这只需要在 Memory Analyzer 中安装对 IBM DTFJ 特性支持的插件即可，要求 Memory Analyzer 的版本最低为 0.8。如何安装 IBM DTFJ 特性支持请参考这篇文章：[《IBM DTFJ feature installation instructions》](#)。在成功安装后，再打开 File > Open Heap Dump 会增加如下选项：

- IBM DTFJ for 1.4.2 VMs
- IBM DTFJ for Portable Heap Dumps
- IBM SDK for Java (J9) Javacore
- IBM SDK for Java (J9) System dumps

要生成可供 Memory Analyzer 使用的 dump 文件，要求使用的 IBM 虚拟机最低版本是 IBM JDK 1.4.2 SR12, 5.0 SR8a and 6.0 SR2 t。尽管之前的版本也能生成 Memory Analyzer 可用的 dump 文件，但是 root 信息不够准确。

### 1. IBM Java5.0 和 IBM Java6 虚拟机的 dump 操作

对于 IBM Java5.0 和 IBM Java6 虚拟机来说，只需要添加如下的命令行参数即可：

-

```
Xdump:system+heap+java:events=systhrow+user,filter=java/lang/OutOfMemoryError,request=exclusive+prewalk+compact
```

命令参数前有一个“-”，请注意下。需要解释下命令行参数中几个主要信息：

#### dump 类型：

- **system**：一个系统进程 dump。将 system dump 文件加载进 Memory Analyzer 前需要先使用 jextract 处理一下。不要给处理后的文件添加一个.sdff 扩展名，因为这只用于 java 1.4.2 的 system dump 文件；
- **heap**：一个 Portable Heap Dump (PHD)文件，包含了所有类和对象的信息，但是没有线程的细节；
- **javacore**：一个可读文件，包含了类加载器信息，可以在用 Memory Analyzer 读取 PHD 文件时使用。

#### Events：

- **systhrow**：在系统发生异常时；
- **user**：用户使用 CTRL + BREAK 时。

#### Filter：

- `java/lang/OutOfMemoryError`: 指的是系统抛出的异常的类型。

**Request:**

- `exclusive`: 在生成 dump 文件时, 停止一切对堆内存的修改;
- `prewalk`: 保证生成 dump 文件时堆内存足够安全;
- `compact`: 压缩 dump 文件的大小。

**2. IBM Java 1.4.2 虚拟机的 dump 操作**

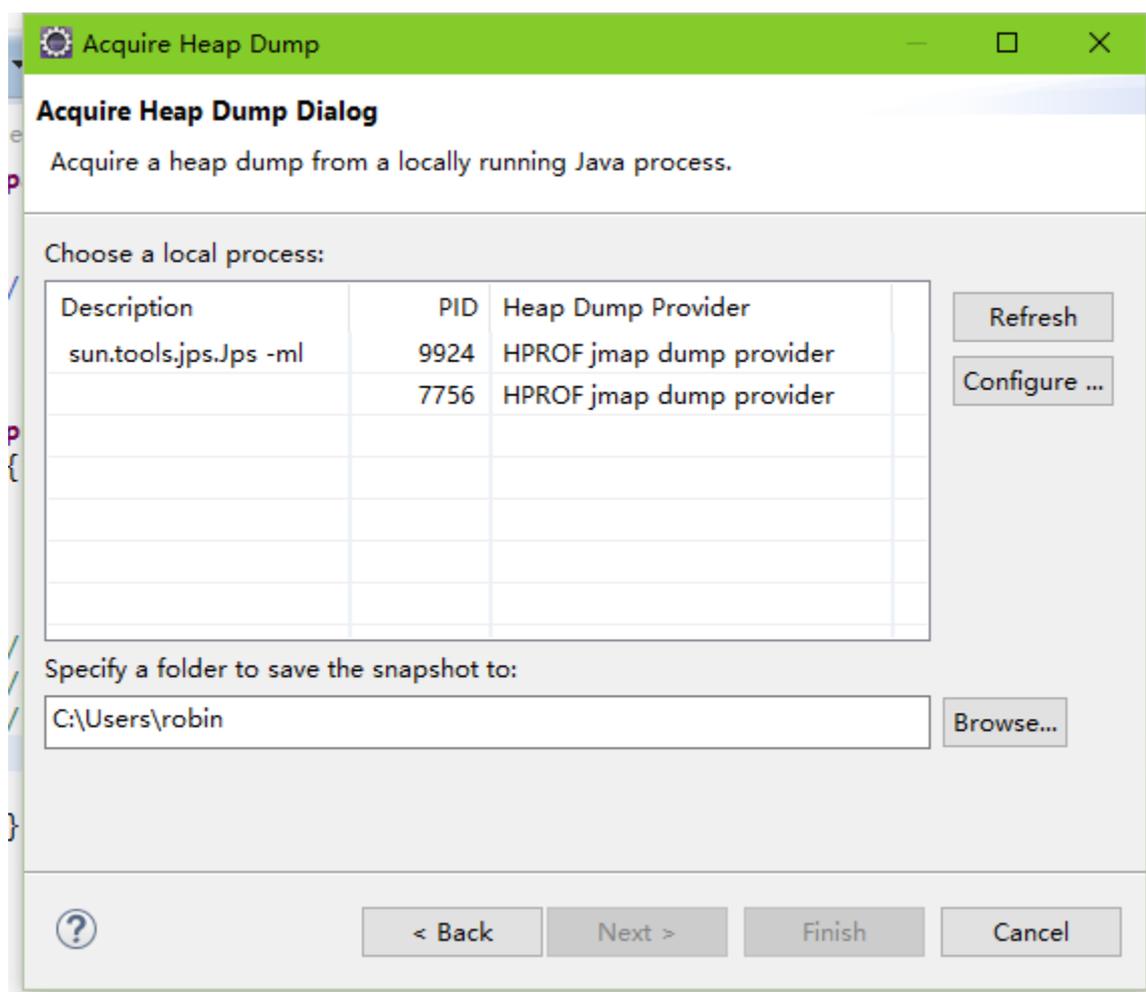
在非 z/OS 系统上, 需要先使用 JExtract 对系统 dump 文件进行处理, 最终生成一个扩展名为.sdff 的文件。在 z/OS 系统上, 需要将 dump 文件以 binary 模式拷贝至 Memory Analyzer 的系统上, 并修改扩展名为.dmp。

## 通过 Memory Analyze 获取 Heap Dump 文件

如果要获取 dump 文件的 Java 进程和 Memory Analyzer 在同一台机器上, 可以直接使用 Memory Analyzer 获取 dump 文件。采用这种方式获取 dump 文件后会直接将之解析并在 Memory Analyzer 中打开 (这个方式我太喜欢, 因为会在用户目录下生成很多零碎的文件)。

获取 Heap Dump 是受虚拟机支持的一种特定的功能。Memory Analyzer 提供了一些被称为 Heap Dump Provider 的概念: 比如支持 Sun 虚拟机 (需要用到 Sun JDK 的 jmap 功能) 的 Provider 或支持 IBM 虚拟机 (也需要一个 IBM JDK) 的 Provider。此外, Memory Analyzer 也提供了对用户自定义的 Heap Dump Provider 插件的扩展支持。

要使用 Memory Analyzer 获取 dump 文件需要先打开 Memory AnalysisPerspective, 而后点击 File -> Acquire Heap Dump...菜单项。之后会打开如下窗口:



在上图中，根据具体的运行环境，预装的 HeapDump Provider 按照默认设置展示了当前正在运行的 java 进程列表。一些 HeapDump Provider 也允许（或者说需要）设置一些额外的参数（比如 HeapDump 的类型），这个可以点击 Configure 按钮来设置。选择要 dump 的线程，点击 Next 按钮就可以生成 dump 文件了。

有的时候进程列表可能为空，这时有必要调整下 Heap Dump Provider 的设置了。点击 Configure 按钮，选择合适的 provider 并点击，然后找到要做调整的参数并做设置就好了。

## 一个 Heap Dump 文件中有多个内存快照

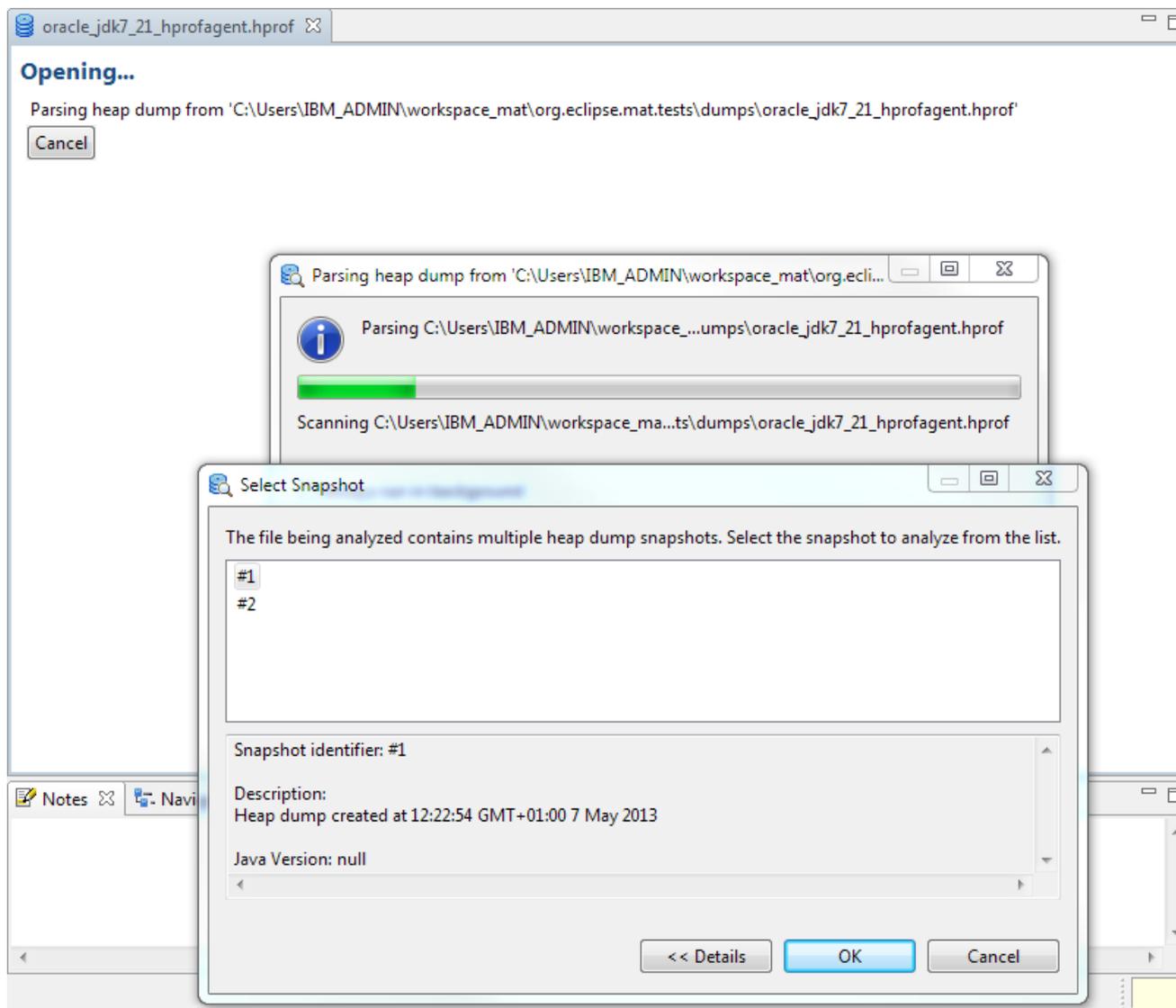
有的时候生成的一个 dump 文件中可能会有多重内存快照，比如，如果是使用如下参数生成的 dump 文件：

```
-agentlib:hprof=heap=dump,format=b
```

那么多次触发 Heap Dump 操作就有可能导致所有的 Heap Dump 信息都写到一个文件里。再比如，IBM 的 z/OS 上的系统 dump 有可能包含来自多个地址空间或者进程的数据，因此 dump 文件中也就有可能会有来自多个 java runtimes 的 Heap Dump 快照。

Memory Analyzer 1.2 及之前的版本会默认使用第一个 Heap Dump 快照——除非是在环境变量中或者 MAT DTFJ configuration 设置中选择了另一个。

Memory Analyzer 1.3 及之后的版本在检测到有多个 Heap Dump 快照后，就会弹出一个对话框，让用户自行选择需要的 Heap Dump 快照。



生成的索引文件的文件名中会包含一个快照的标识符，这样不同快照的索引文件就可以区别开来。这也意味着可以在 Memory Analyzer 中同时分析一个 Heap Dump 文件的多个内存快照。Memory Analyzer 会记住上次打开的快照的信息，如果要打开另一个快照，可以在打开之前曾打开的快照后再将之关闭，之后再次点击 File -> Open Heap Dump... 菜单项打开原来的 Heap Dump 文件，此时就又可以重新选择要解析的快照。之前解析过的快照也可以通过已生成的索引文件再次打开，这样子就可以同时分析多个快照的信息了。

## 总结

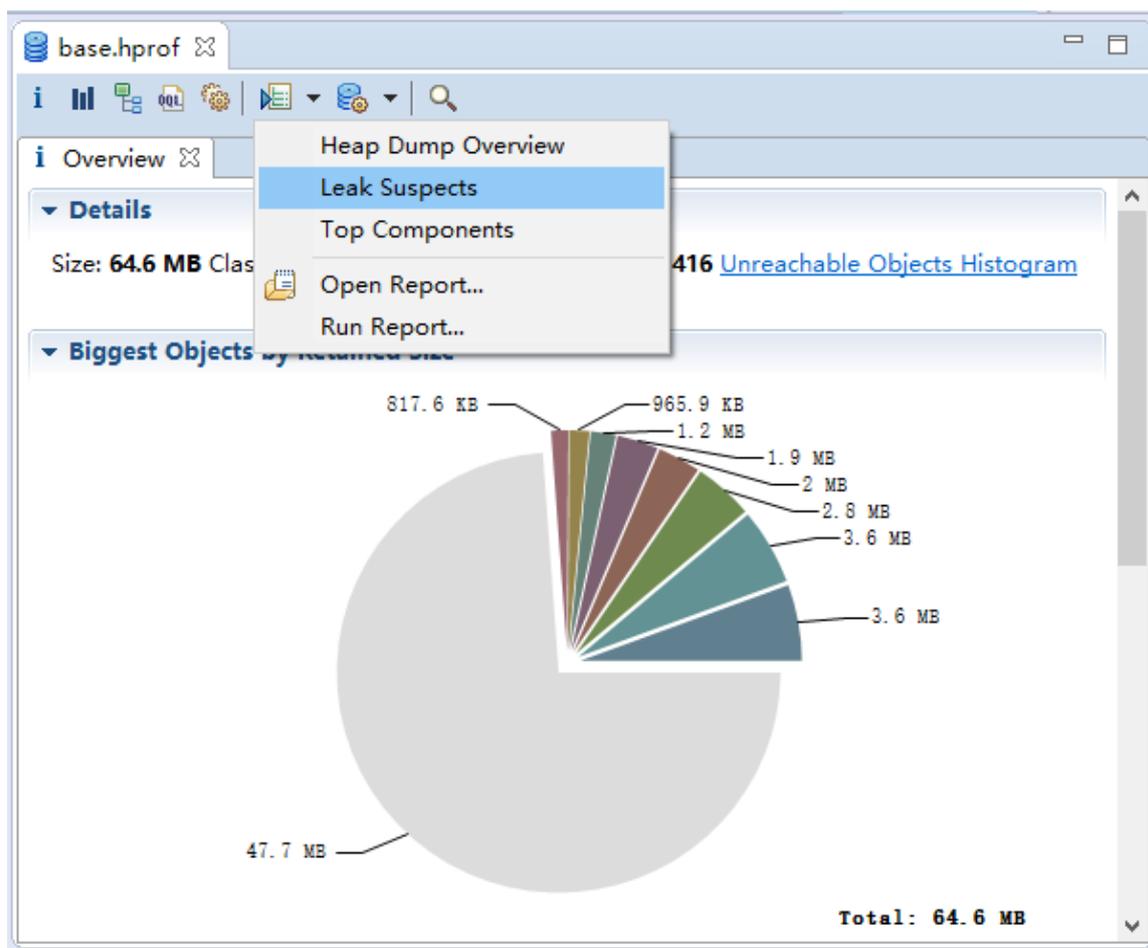
下面是一个表格，总结了在不同平台上使用虚拟机设置以及工具来获取 dump 的方式（为了适应 word 稍稍调了一下，凑合看吧）：

Vendor	Release	VM Parameter			Sun Tools		SAP Tool	MAT
		On out of memory	On Ctrl+Break	Agent	JMap	JConsole	JVMMon	acquire Heap Dump
Sun, HP	1.4.2_12	Yes	Yes	Yes				No
	1.5.0_07	Yes	Yes (Since 1.5.0_15)	Yes	Yes (Only Solaris and Linux)			Yes (Only Solaris and Linux)
	1.6.0_00	Yes	No	Yes	Yes	Yes		Yes
	1.7.0	Yes	No	Yes	Yes	Yes		Yes
	1.8.0	Yes	No	Yes	Yes	Yes		Yes
SAP	Any 1.5.0	Yes	Yes	Yes	Yes (Only Solaris and Linux)		Yes	
IBM	1.4.2 SR12	Yes	Yes	No	No	No	No	No
	1.5.0 SR8a	Yes	Yes	No	No	No	No	No
	1.6.0 SR2	Yes	Yes	No	No	No	No	No

Vendor	Release	VM Parameter			Sun Tools		SAP Tool	MAT
	1.6.0 SR6	Yes	Yes	No	No	No	No	Yes
	1.7.0	Yes	Yes	No	No	No	No	Yes
	1.8.0 Beta	Yes	Yes	No	No	No	No	Yes

## 2. 运行 Leak Suspect Report

在打开 dump 文件时 Memory Analyzer 就会询问是否打开 Leak Suspect Report 或其他 Report。在需要的时候可以点击 Memory Analyzer 工具栏上的 Run Expert System Test -> Leak Suspects 下拉菜单项。点击后就会打开一个 HTML 的报表，里面包含 HeapDump 的概览图以及疑似内存泄露信息。



这个 HTML 报表文件以及 Memory Analyzer 生成的其它文件将和 Heap Dump 文件保存在一起，并在再次打开 HeapDump 文件的时候省去解析的步骤。

Leak Suspect Report 的部分内容提供了指向独立查询内容的链接，这为后续进一步的分析提供了便利。

Overview default\_report org.eclipse.mat.api:suspects

#### Accumulated Objects by Class in Dominator Tree

Label	Number of Objects	Used Heap Size	Retained Heap Size
 <a href="#">java.util.LinkedHashMap\$Entry</a> First 10 of 564 objects	564	36,096	123,984
 <a href="#">java.lang.String[]</a> First 10 of 67 objects	67	3,752	22,776
 <a href="#">java.util.HashMap\$Entry[]</a> All 1 objects	1	8,216	8,216
 <a href="#">java.lang.String</a> First 10 of 33 objects	33	1,320	2,496
<b>Σ Total: 4 entries</b>	<b>665</b>	<b>49,384</b>	<b>157,472</b>

#### All Accumulated Objects by Class

Class Name	Objects	Shallow Heap
 <a href="#">char[]</a> First 10 of 1,116 objects	1,116	59,672
 <a href="#">java.lang.String</a> First 10 of 1,116 objects	1,116	44,640
 <a href="#">java.util.LinkedHashMap\$Entry</a> First 10 of 564 objects	564	36,096
 <a href="#">java.lang.String[]</a> First 10 of 158 objects	158	8,848
 <a href="#">java.util.HashMap\$Entry[]</a> All 1 objects	1	8,216
 <a href="#">java.util.LinkedHashMap</a> All 1 objects	1	80
<b>Σ Total: 6 entries</b>	<b>2,956</b>	<b>157,552</b>

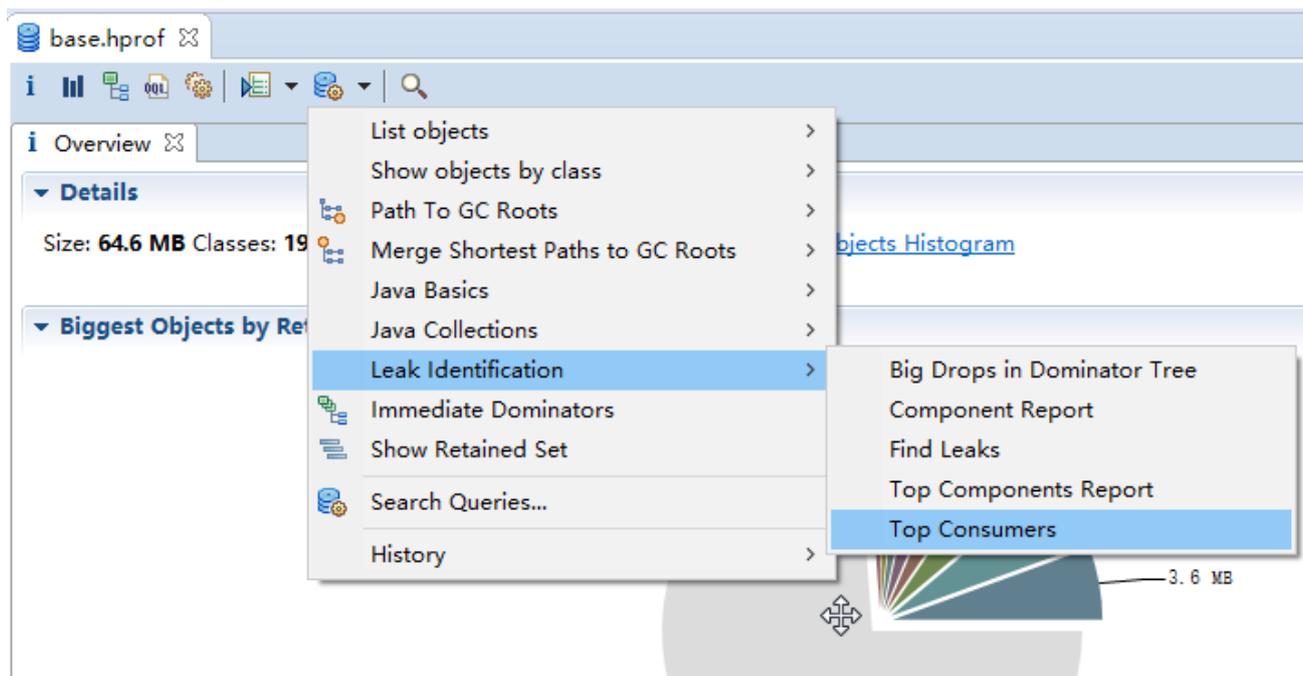
[Table Of Contents](#)

Created by [Eclipse Memory Analyzer](#)

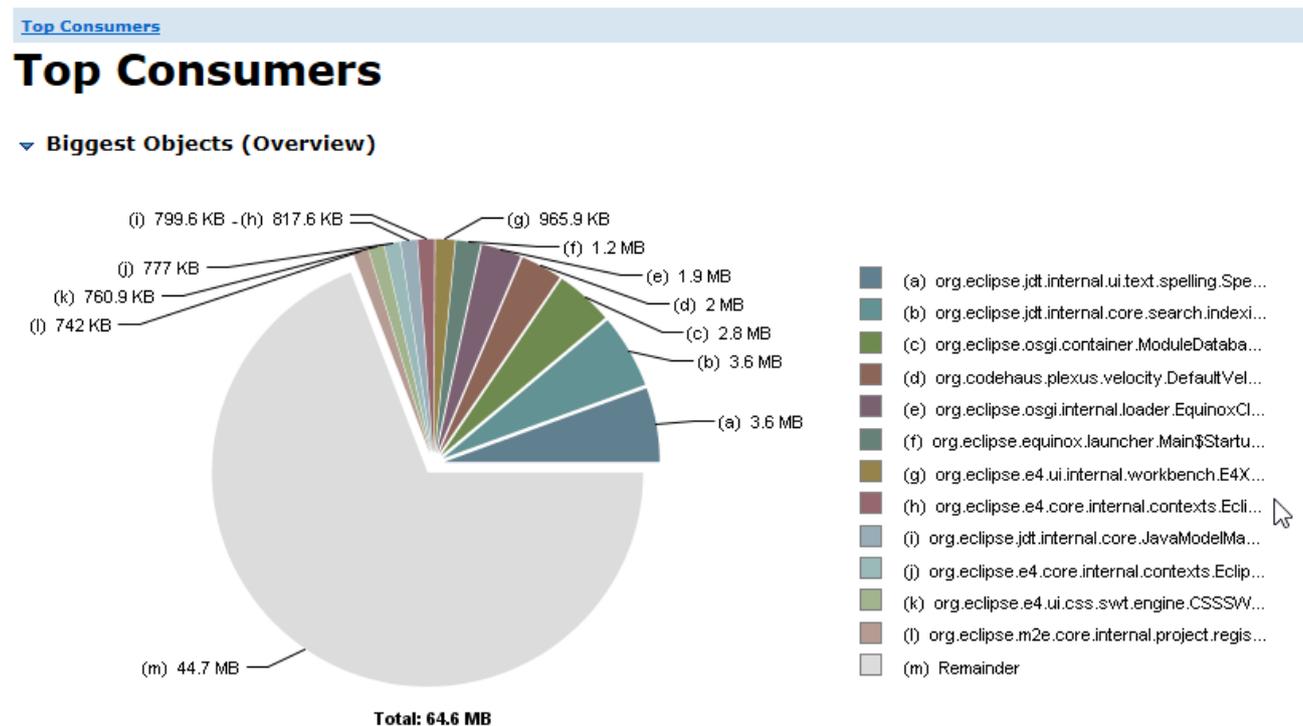
想要了解更多关于 Leak Suspect Report 的内容，可以看一下这篇文章：[Automated Heap Dump Analysis: Finding Memory Leaks with One Click.](#)

### 3. 列出最大的对象

Top Consumer 查询可以按类、类加载器和包进行分组并找出每组最大的对象。在 Memory Analyzer 工具栏中依次打开如下选项 Open Query Browser -> Leak Identification -> Top Consumer 即可执行 Top Consumer 查询：



查询结果是一个 HTML 页，其中的对象会以超链接的形式展示，点击超链接，可以通过左键菜单项做进一步的的分析。



对象列表:

### ▼ Biggest Objects

Class Name	Shallow Heap	Retained Heap
<a href="#">org.eclipse.jdt.internal.ui.text.spelling.SpellCheckEngine @ 0xc29b4b50 »</a>	32	3,789,640
<a href="#">org.eclipse.jdt.internal.core.search.indexing.IndexManager @ 0xc1c07688 Busy Monitor »</a>	80	3,772,696
<a href="#">org.eclipse.osgi.container.ModuleDatabase @ 0xc0786e90 »</a>	64	2,934,272
<a href="#">org.codehaus.plexus.velocity.DefaultVelocityComponent @ 0xc3351d50 »</a>	32	2,121,008
<a href="#">org.eclipse.osgi.internal.loader.EquinoxClassLoader @ 0xc1380d60 com.ibm.icu »</a>	96	2,024,976
<a href="#">org.eclipse.equinox.launcher.Main\$StartupClassLoader @ 0xc0000000 Equinox Startup Class Loader »</a>	88	1,236,376
<a href="#">org.eclipse.e4.ui.internal.workbench.E4XMIResource @ 0xc16cbe50 »</a>	128	989,104
<a href="#">org.eclipse.e4.core.internal.contexts.EclipseContext @ 0xc211f9d0 »</a>	56	837,264
<a href="#">org.eclipse.jdt.internal.core.JavaModelManager @ 0xc1c06fb8 »</a>	200	818,784
<a href="#">org.eclipse.e4.core.internal.contexts.EclipseContext @ 0xc16b61e8 »</a>	56	795,616
<a href="#">org.eclipse.e4.ui.css.swt.engine.CSSSWTEngineImpl @ 0xc216b1e8 »</a>	80	779,160
<a href="#">org.eclipse.m2e.core.internal.project.registry.ProjectRegistryManager @ 0xc2a61740 »</a>	48	759,840
<b>Σ Total: 12 entries</b>		

如前文所述，Top Consumer 页按照类、类加载器和包也分别提供了对应的类似图表信息。

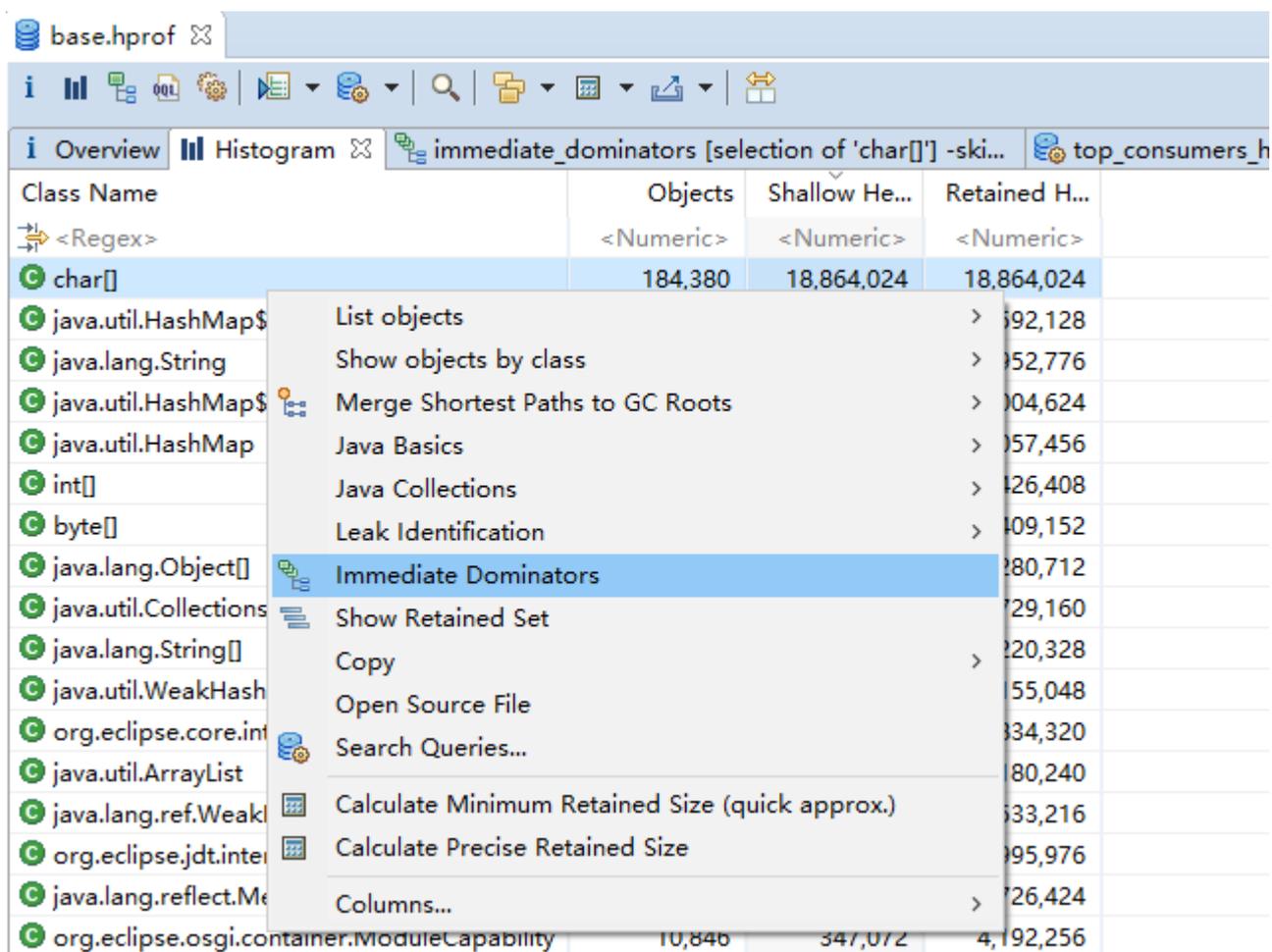
## 4. 找出 Responsible Objects

在前面（概念->Dominator Tree）曾提到直接支配者（immediate dominator）的概念，也就是在对象引用关系图中，某个对象的支配者是要引用该对象必须经过的点，而直接支配者则是所有支配者中离这个对象最近的一个。我们也可以称直接支配者为对应对象的 Responsible Object（责任对象），这里说的 responsibility 指的不只是提供了一个指向该对象的引用，而是保证这个对象的存活（alive）。

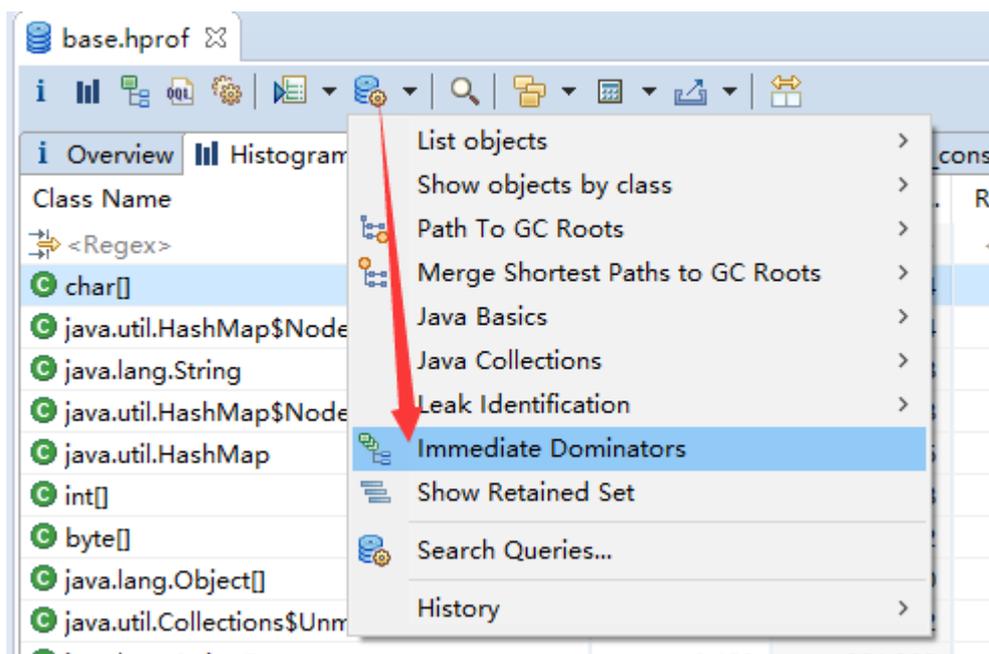
Memory Analyzer 提供了 immediate dominator 查询，使用这个查询可以将一组相同类的对象的所有直接支配者找出来，这些支配者就是这组对象的 Responsible Objects。因为每个对象只有一个直接支配者，这样就可以过滤掉大量的无聊的且让人头疼的底层引用（比如 java.\* 这样的类），从而直接找到该对象在应用代码中的调用关系。

执行 immediate dominator 查询的方式：

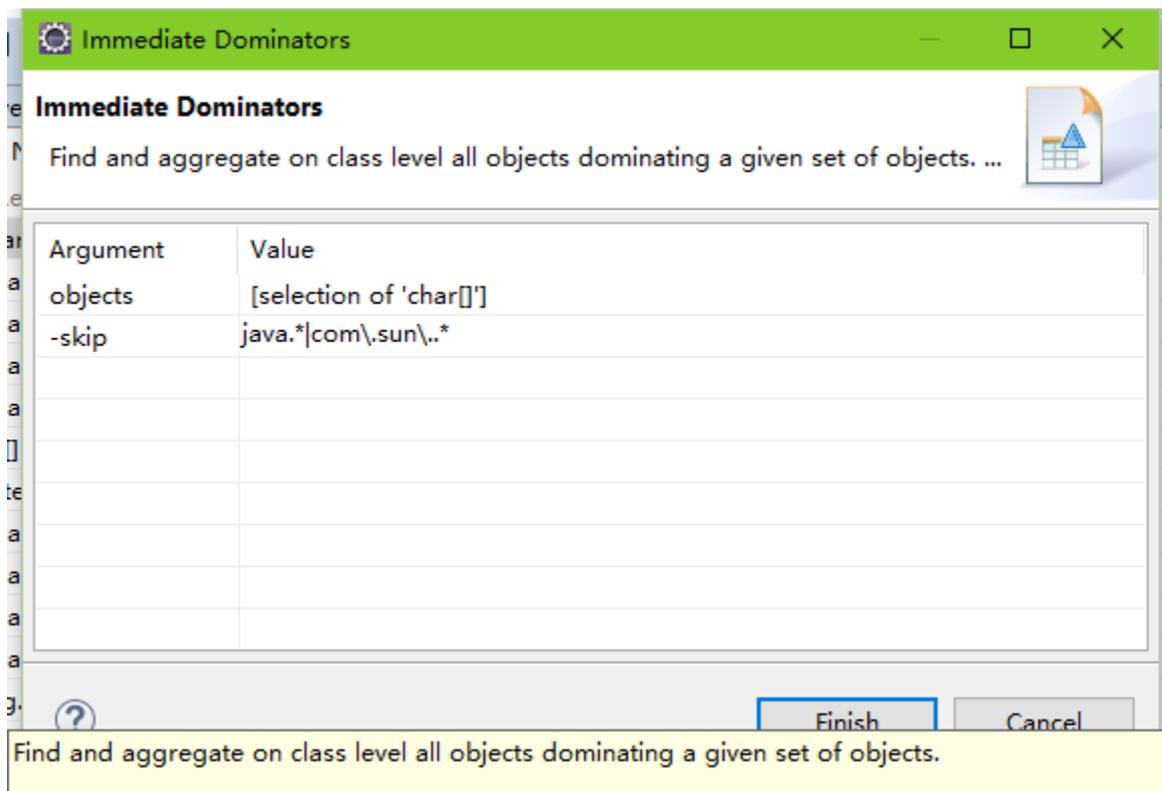
- 在 histogram 中选择相应的条目，在右键菜单中选择 Immediate Dominators:



- 在 Memory Analyzer 的工具栏中选择 Query Browser > Immediate Dominators，随后会打开一个引导窗口，在引导窗口中输入要查询的对象类即可：



引导窗口:



下图是查询结果:

Class Name	Objects	Dom. Objec...	Shallow He...	Dom. Shall...
<Regex>	<Numeric>	<Numeric>	<Numeric>	<Numeric>
<ROOT>	1	54,480	0	5,506,824
char[][]	313	26,014	188,088	2,873,104
org.eclipse.osgi.internal.loader.classpath.ClasspathEntry	70	16,568	2,240	1,765,984
org.eclipse.osgi.internal.loader.EquinoxClassLoader	230	15,473	22,080	1,664,080
org.eclipse.e4.core.internal.contexts.EclipseContext	17	2,600	952	387,128
org.apache.velocity.runtime.parser.VelocityCharStream	40	40	2,560	328,320
sun.net.www.protocol.jar.URLJarFile	3	2,155	240	269,040
org.eclipse.e4.ui.internal.workbench.E4XMIResource	2	3,831	256	245,112
org.eclipse.e4.core.internal.contexts.osgi.EclipseContextO...	1	1,571	64	223,136
org.eclipse.osgi.storage.ManifestLocalization\$Localizatio...	8	2,292	384	194,600
org.eclipse.core.commands.Command	1,078	1,936	86,240	174,752
org.eclipse.core.internal.preferences.InstancePreferences	948	1,937	68,256	173,520
org.eclipse.jface.resource.URLImageDescriptor	997	2,231	15,952	170,776
org.eclipse.osgi.storage.BundleInfo\$Generation	632	716	50,560	163,096
org.eclipse.emf.ecore.xml.impl.StringSegment\$Element	66	1,975	1,584	147,624
org.eclipse.rse.services.clients.messages.SystemMe...	2	2,104	48	144,512
org.apache.maven.plugin.descriptor.Parameter	402	1,121	22,512	140,584
org.eclipse.ui.internal.handlers.ActionCommandMapping...	1	664	16	133,800
org.eclipse.e4.ui.css.swt.dom.preference.EclipsePreferenc...	938	1,875	45,024	127,696
org.eclipse.core.runtime.Path	1,132	1,966	27,168	121,448

在查询结果图中列出了所有 char[] 数组对象的直接支配者对象。这些直接支配者保证了 char[] 对象的生存。因为在查询使用了 -skip 选项，略过了 java.lang.String 这样的 JDK 类的对象，所以我们在结果图中看到的大部分都是 eclipse 相关类（我们使用的 Heap Dump 就是 eclipse 的运行时 dump）。

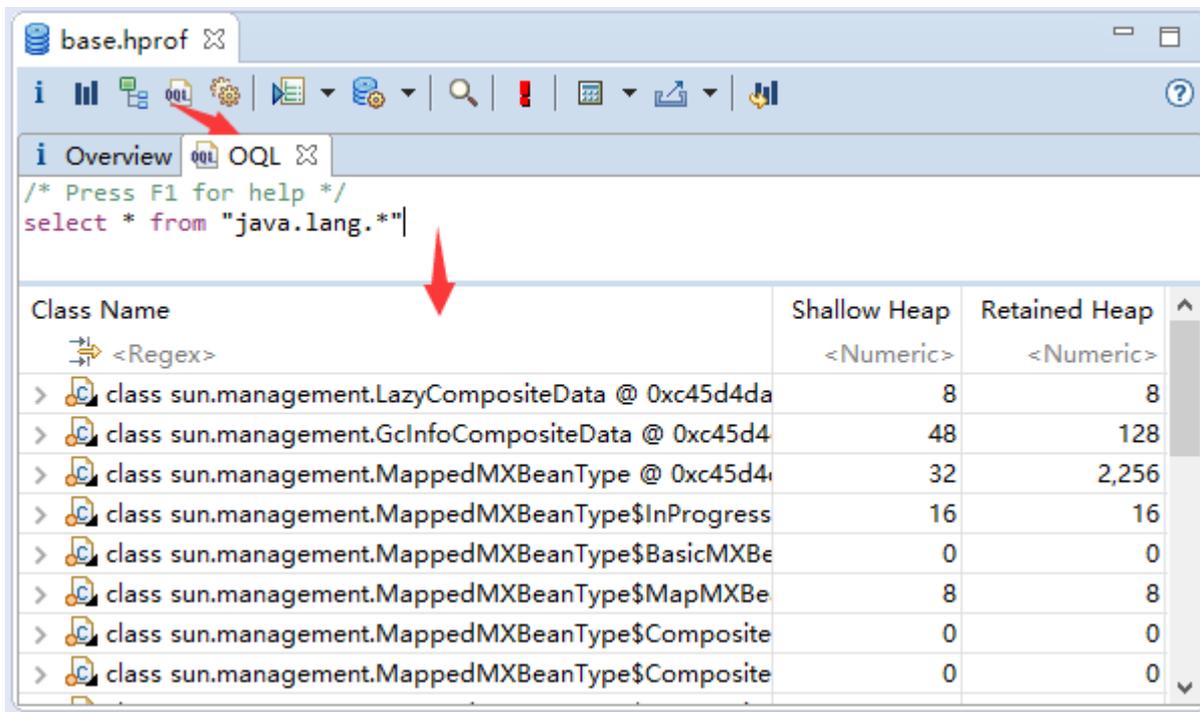
## 5. 使用 OQL 查询 Heap 对象

在 Memory Analyzer 中，用户可以使用类似 SQL 的查询语句查询在 Heap 中的对象。因为是查询对象的语句所以被称为 OQL（Objects Query Language）。与 SQL 对应时，OQL 中的类被视为表，类的实例被视为一行记录，类的属性被视为表中的字段。

```
SELECT *
FROM [ INSTANCEOF ] <class name="name">
[ WHERE <filter-expression> ]
</filter-expression></class>
```

打开 OQL 编辑器可以使用 Memory Analyzer 工具栏上的  图标。OQL 编辑器窗口被分成了两部分：

- 上方的文字输入区域；
- 下方的查询结果展示区域。



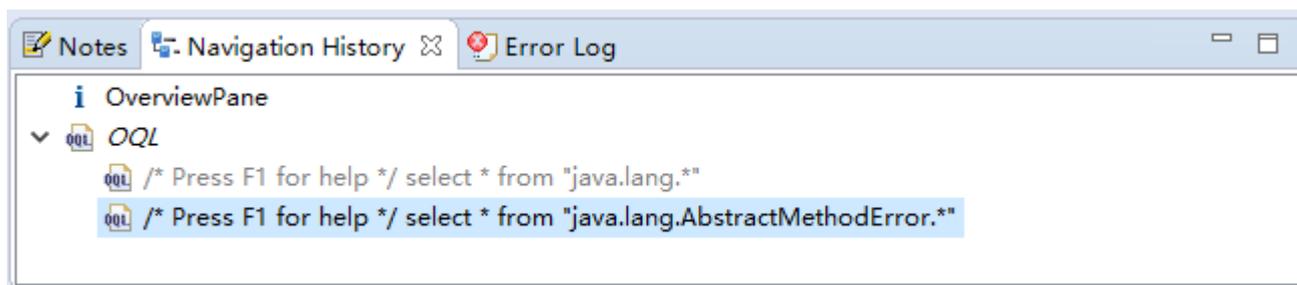
输入查询语句后，可以按 F5 或者 Ctrl+Enter 按钮或者点击工具栏中的红色  按钮来执行查询。

基础的 OQL 语法如下：

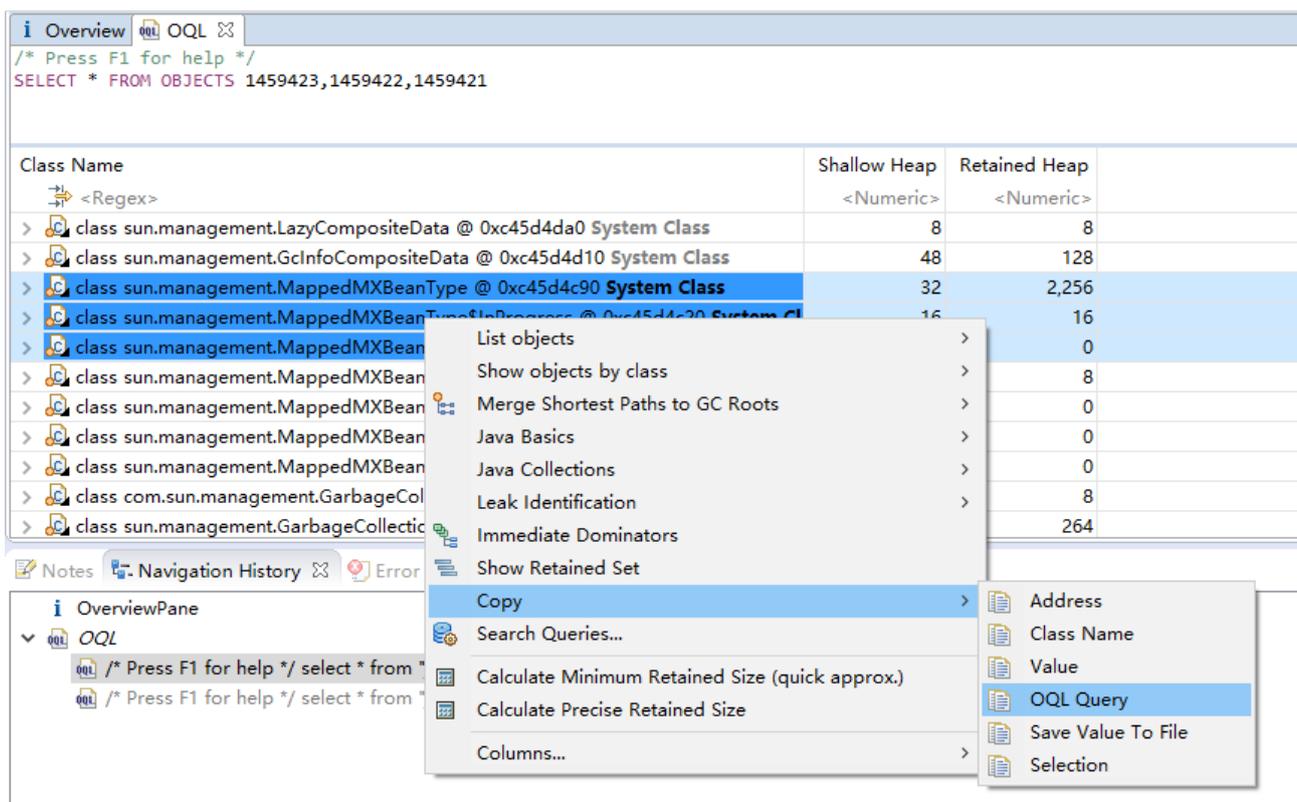
```
SELECT *
FROM [ INSTANCEOF ] <class name>
[ WHERE <filter-expression> ]
```

在文字输入区域数据查询语句时，可以注意到 Memory Analyzer 提供了自动补全功能——可以针对类名、类名表达式、属性名、域名称和方法名进行自动补全。后文会有专门介绍 OQL 自动补全的内容。

在 Memory Analyzer Perspective 中的 NavigationHistory 窗口下可以看到曾经执行过的 OQL 语句，双击其中的任意一条语句可以再次执行。



在查询结果窗口中，选择其中的一些对象，在右键菜单中执行 Copy > OQL Query 可以生成一条代表这些对象的 OQL 语句。生成的语句可以直接粘贴在 OQL 编辑器输入区域执行。



## 6. 类加载器分析

类加载器负责将类加载到 JVM 内存中。在分析堆内存时，重视对类加载器的分析出于两点考虑：

- 首先，应用通常使用独立的类加载器加载应用组件（Component，指的是同一个包下的所有类的集合或者是由同一个类加载器加载的全部类的集合）；
- 其次，被加载的类通常也是被保存在独立的空间中（比如 permSpace），这些类也是可以回收的。

要查看 HeapDump 中的类加载器信息，可以在 Memory Analyzer 的工具栏中依次打开如下选项：Query Browser > Java Basics > Class Loader Explorer。

Class Name	Defined Classes	No. of Instances
<Regex>	<Numeric>	<Numeric>
<system class loader>	2,919	1,149,711
org.eclipse.m2e.maven.runtime	1,945	58,096
org.eclipse.ui.workbench	1,253	13,516
org.eclipse.jdt.core	982	3,642
org.eclipse.jdt.ui	731	25,979
org.eclipse.jst.j2ee.core	489	361
org.eclipse.swt	420	4,942
Equinox Startup Class Loader	416	57,174
org.eclipse.jface	360	6,150
org.eclipse.emf.ecore	351	48,020
org.eclipse.jface.text	340	484
org.eclipse.m2e.maven.indexer	320	1,544
com.google.guava	276	163
org.eclipse.core.resources	275	912
org.apache.lucene.core	227	198
parent org.eclipse.osgi.internal.framework.EquinoxContainerAdaptor\$1 @ 0xc0f1	0	0
org.apache.lucene.store.MMapDirectory\$MMapIndexInput		14
org.apache.lucene.util.Version		13
org.apache.lucene.util.VirtualMethod		10
org.apache.lucene.index.FieldInfo		8
org.apache.lucene.index.IndexFileDeleter\$RefCount		8
org.apache.lucene.util.SimpleStringInterner\$Entry		8
org.apache.lucene.util.CloseableThreadLocal		7
org.apache.lucene.util.UnicodeUtil\$UTF8Result		7

简单说一下 Class Loader Explorer 中的表格：

Memory Analyzer 为每个类加载器绑定了一个有意义的名称——如果使用了 OSGi bundles，那这个名称就是 bundle id。留神可能会有名称重复的情况。

在表格中紧挨着类加载器名称（Class Name）的是定义的类（Defined Class）和活动的实例的数量（No. of Instances）。如果同一个组件被加载多次，那么活着的实例数量可以指示哪个类加载器更加有活力，以及哪个应该被垃圾回收掉。

## 7. 线程分析

Memory Analyzer 提供了几种针对获取 dump 时的线程的查询方式。

### 线程概览（Overview）

要查看 Heap Dump 中所有的线程可以使用工具栏中的“Thread Overview”按钮，结果如下图所示。此

外也可以执行 Query Browser > Java Basics > Thread Overview and Stacks 来获取线程概览图。

Object / Stack Frame	Name	Shallow Heap	Retained Heap	Context Class Loader	Is Daemon
<Regex>	<Regex>	<Numeric>	<Numeric>	<Regex>	<Regex>
org.eclipse.osgi.framework.eventmgr.EventManager	Start Level: Equinox Container: a0595f35-4a12-0...	136	39,320	org.eclipse.osgi.internal.framework.ContextFinder @ 0x...	true
java.lang.Thread @ 0xc1344f40	main	120	25,496	org.eclipse.osgi.internal.framework.ContextFinder @ 0x...	false
java.lang.Thread @ 0xc3cee58	RMI TCP Connection(2)-192.168.0.102	120	19,368	sun.misc.Launcher\$AppClassLoader @ 0xc0c62a48	true
java.lang.Thread @ 0xc3cee390	RMI TCP Connection(4)-192.168.0.102	120	18,560	sun.misc.Launcher\$AppClassLoader @ 0xc0c62a48	true
java.lang.Thread @ 0xc3cedf8	RMI TCP Connection(5)-192.168.0.102	120	18,312	sun.misc.Launcher\$AppClassLoader @ 0xc0c62a48	true
org.eclipse.core.internal.jobs.Worker @ 0xc2902ae1	Worker-3	128	3,688	org.eclipse.osgi.internal.framework.ContextFinder @ 0x...	false
org.eclipse.jface.text.reconciler.AbstractReconciler\$org.eclipse.jdt.internal.ui.text.JavaReconciler	org.eclipse.jdt.internal.ui.text.JavaReconciler	128	3,192	org.eclipse.osgi.internal.framework.ContextFinder @ 0x...	true
org.eclipse.osgi.framework.eventmgr.EventManager	Framework Event Dispatcher: Equinox Containe...	136	3,024	org.eclipse.osgi.internal.framework.ContextFinder @ 0x...	true
org.eclipse.core.internal.jobs.Worker @ 0xc2902701	Worker-4	128	1,352	org.eclipse.osgi.internal.framework.ContextFinder @ 0x...	false
org.eclipse.rse.services.clients.server.messages.System	Thread-10	136	1,096	org.eclipse.osgi.internal.framework.ContextFinder @ 0x...	false
org.eclipse.rse.services.clients.server.messages.System	Thread-11	136	1,096	org.eclipse.osgi.internal.framework.ContextFinder @ 0x...	false
org.eclipse.osgi.framework.eventmgr.EventManager	EventAdmin Async Event Dispatcher Thread	136	896	org.eclipse.osgi.internal.framework.ContextFinder @ 0x...	true
java.lang.Thread @ 0xc1343848	[Timer] - Main Queue Handler	120	672	org.eclipse.osgi.internal.framework.ContextFinder @ 0x...	true
at java.lang.Object.wait(JV (Native Method)					
at org.eclipse.equinox.internal.util.impl.tpt.timer.Timer					
<local> org.eclipse.equinox.internal.util.impl		32	208		
<local> org.eclipse.equinox.internal.util.impl		64	184		
<local> java.lang.Object @ 0xc1343ba8 Bus		16	16		
Total: 3 entries					
at java.lang.Thread.run(JV (Thread.java:745)					
Total: 3 entries					
org.eclipse.equinox.internal.p2.updatechecker.Update	Thread-9	160	592	org.eclipse.osgi.internal.framework.ContextFinder @ 0x...	false
org.eclipse.osgi.framework.eventmgr.EventManager	Provisioning Event Dispatcher	136	520	org.eclipse.osgi.internal.framework.ContextFinder @ 0x...	true

在查询结果中可以看到线程的名称、类加载器和相关的对象等信息。

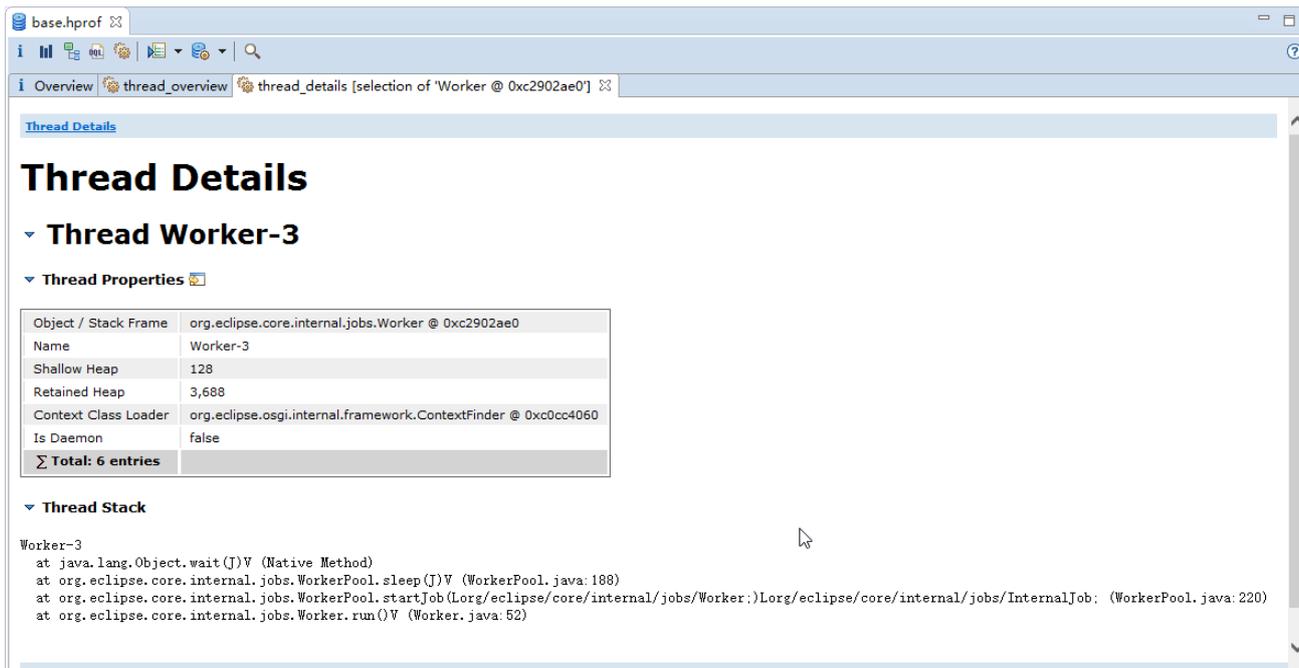
一些 Heap Dump 文件（比如 JVM6 及以后的 Heap Dump 文件以及 IBM 虚拟机的系统 dump 文件）中包含了线程调用栈的信息以及每个栈帧的局部 java 对象。

探索线程调用栈和局部 java 对象是一项非常强大的功能，提供了一种对 Heap Dump 这样的内存快照进行类似 debug 模式操作的功能。通过这个功能，我们可以深入分析一些内存密集型操作的细节。同时这也说明 Heap Dump 和 Memory Analyzer 不仅可以用于内存分析，也可以用来处理相当广泛地其他问题比如应用反应迟钝的问题。

## 线程细节（Thread Details）

可以使用右键菜单中的 Java Basics > Thread Details 来对单个线程进行分析。Memory Analyzer 提供了一个扩展点，这样的扩展可以提供有关线程活动的语义信息。Thread Detail 查询的结果中就包含了这些信息（如果可用的话）、一些概要信息以及线程堆栈跟踪信息。

对于基于 DTFJ 的 dump（IBM 系统 dump 以及 IBM PHD 文件相关的 java dump），在 Thread Details 查询中可以看到更多的信息，包括线程状态、线程优先级以及 native 栈跟踪。



## IBM 虚拟机的 dump 中的线程栈

DTFJ 解析器允许对线程栈的查看进行更多的控制。这个可以在 DTFJ Parser 的首选项 (preferences) 页面中进行设置。有如下选项：

### 1. Normal

只在线程栈窗口展示栈帧 (stack frame) 信息。

### 2. 只将栈帧视为一个伪对象

将栈帧视为一个伪对象展示在所有的视图中，如 Path to GC root、outbound references from threads (线程的对外引用)。栈帧中的局部变量被视为栈帧对象的对外引用。这有助于发现是哪些栈帧保证了对象的存活。不过这里栈帧对象的 size 指的是栈帧在虚拟机栈上占用的空间而非是在堆上占用的空间。

### 3. 将栈帧视为一个伪对象并将正在运行的方法视为一个伪类

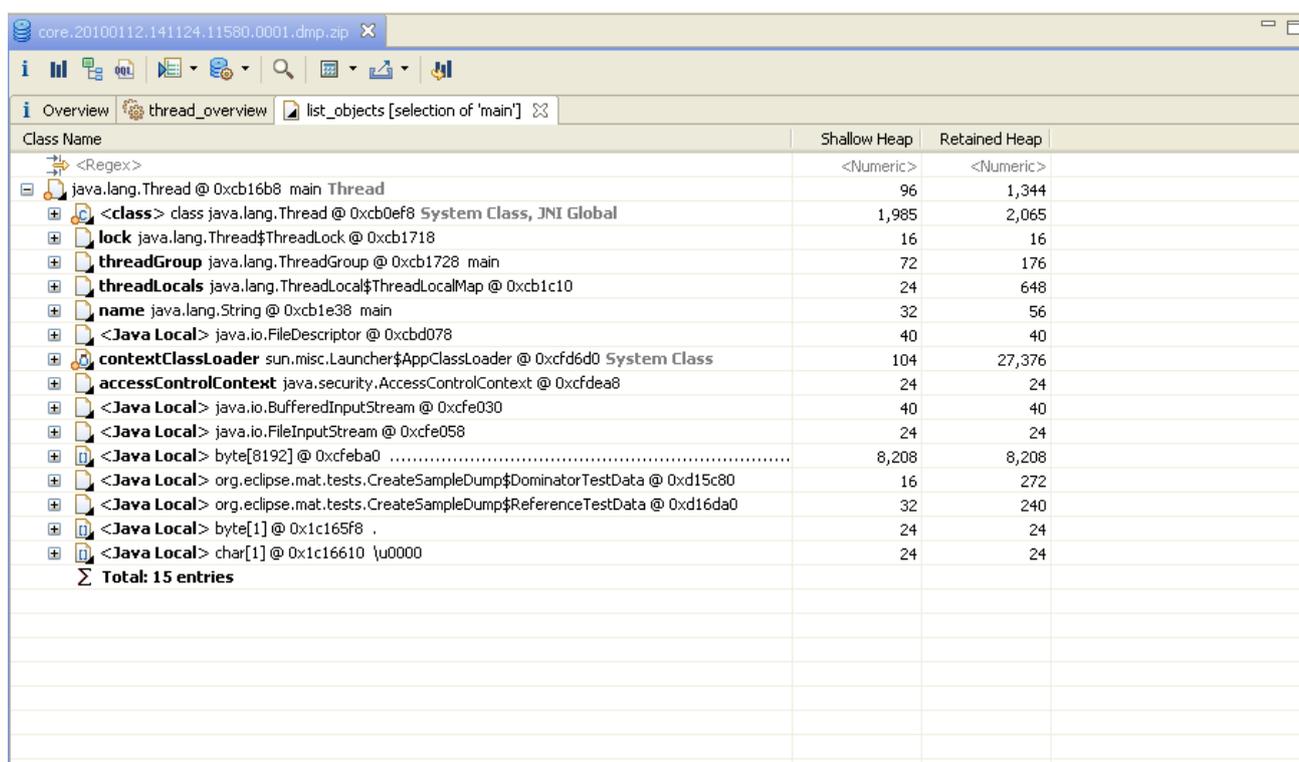
将栈帧视为一个伪对象展示在所有的视图中，如 Path to GC root、outbound references from threads (线程的对外引用)。栈帧中的局部变量被视为栈帧对象的对外引用。这有助于发现是哪些栈帧保证了对象的存活。在这里将栈帧中的方法视为一个伪类，并将之作为栈帧对象的类型。通过查找这种伪类的实例的数量，可以比较容易找出哪些方法在所有的线程中运行、哪些方法占用了大量的栈空间。这有助于解决 StackOverflowErrors 这样的问题。

### 4. 将栈帧视为一个伪对象并将所有的方法视为伪类

将栈帧视为一个伪对象展示在所有的视图中，如 Path to GC root、outbound references from threads（线程的对外引用）。栈帧中的局部变量被视为栈帧对象的对外引用。这有助于发现是哪些栈帧保证了对象的存活。在这里将栈帧中的方法视为一个伪类，并将之作为栈帧对象的类型。通过查找这种伪类的实例的数量，可以比较容易找出哪些方法在所有的线程中运行、哪些方法占用了大量的栈空间。这有助于解决 StackOverflowErrors 这样的问题。其他所有的方法也被创建为对应伪类的对象。方法伪类对象的 size 就是字节码以及 JIT 码的 size，在其他模式中这些会被累计入类定义占用空间的 size 中。这有助于找出哪些方法的字节码或机器码占用了大量的非堆内存。

可以分别看一下以上几种设置选项的视图：

### 1. Normal，栈帧不被视为是对象



Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.lang.Thread @ 0xcb16b8 main Thread	96	1,344
<class> class java.lang.Thread @ 0xcb0ef8 System Class, JNI Global	1,985	2,065
lock java.lang.Thread\$ThreadLock @ 0xcb1718	16	16
threadGroup java.lang.ThreadGroup @ 0xcb1728 main	72	176
threadLocals java.lang.ThreadLocal\$ThreadLocalMap @ 0xcb1c10	24	648
name java.lang.String @ 0xcb1e38 main	32	56
<Java Local> java.io.FileDescriptor @ 0xcbd078	40	40
contextClassLoader sun.misc.Launcher\$AppClassLoader @ 0xcfd6d0 System Class	104	27,376
accessControlContext java.security.AccessControlContext @ 0xcfd6e8	24	24
<Java Local> java.io.BufferedInputStream @ 0xcfe030	40	40
<Java Local> java.io.FileInputStream @ 0xcfe058	24	24
<Java Local> byte[8192] @ 0xcfeba0 .....	8,208	8,208
<Java Local> org.eclipse.mat.tests.CreateSampleDump\$DominatorTestData @ 0xd15c80	16	272
<Java Local> org.eclipse.mat.tests.CreateSampleDump\$ReferenceTestData @ 0xd16da0	32	240
<Java Local> byte[1] @ 0xd165f8 .	24	24
<Java Local> char[1] @ 0xd16610 \u0000	24	24
Σ Total: 15 entries		

### 2. 栈帧被视为伪类对象

注意，这里栈帧的类型是<stack frame>

Class Name	Shallow Heap
<Regex>	<Numeric>
java.lang.Thread @ 0xcb16b8 main Thread	96
<class> class java.lang.Thread @ 0xcb0ef8 System Class, JNI Global	1,985
<Java Stack Frame> <stack frame> @ 0x11b8d0 java.io.FileInputStream.readBytes([BII) (FileInputStream.java)	0
<class> class <stack frame> @ 0x1cb0018	0
<Java Local> java.io.FileDescriptor @ 0xcbd078	40
<Java Local> java.io.FileInputStream @ 0xcfe058	24
<Java Local> byte[8192] @ 0xcfeba0 .....	8,208
Σ Total: 4 entries	
<Java Stack Frame> <stack frame> @ 0x11b8ec java.io.BufferedInputStream.fill()V (BufferedInputStream.java:229)	0
<Java Stack Frame> <stack frame> @ 0x11b910 java.io.BufferedInputStream.read1([BII) (BufferedInputStream.java:269)	0
<Java Stack Frame> <stack frame> @ 0x11b93c java.io.BufferedInputStream.read([BII) (BufferedInputStream.java:328)	0
<Java Stack Frame> <stack frame> @ 0x11b964 com.ibm.jvm.io.ConsoleInputStream.read([BII) (ConsoleInputStream.java:177)	0
<Java Stack Frame> <stack frame> @ 0x11b978 com.ibm.jvm.io.ConsoleInputStream.read([B) (ConsoleInputStream.java:167)	0
<Java Stack Frame> <stack frame> @ 0x11b98c com.ibm.jvm.io.ConsoleInputStream.read()I (ConsoleInputStream.java:160)	0
<Java Stack Frame> <stack frame> @ 0x11b9a8 org.eclipse.mat.tests.CreateSampleDump.main([Ljava/lang/String;)V (CreateSampleDump.java:31)	0
lock java.lang.Thread\$ThreadLock @ 0xcb1718	16
threadGroup java.lang.ThreadGroup @ 0xcb1728 main	72
threadLocals java.lang.ThreadLocal\$ThreadLocalMap @ 0xcb1c10	24
name java.lang.String @ 0xcb1e38 main	32
contextClassLoader sun.misc.Launcher\$AppClassLoader @ 0xcfd6d0 System Class	104
accessControlContext java.security.AccessControlContext @ 0xcfd6a8	24
Σ Total: 15 entries	

### 3. 栈帧被视为伪对象，正在运行的方法被视为伪类

注意栈帧类型的不同，比如 `java.io.FileStream.getBytes([BIII);`

Class Name	Shallow Heap
<Regex>	<Numeric>
java.lang.Thread @ 0xcb16b8 main Thread	96
<class> class java.lang.Thread @ 0xcb0ef8 System Class, JNI Global	1,985
<Java Stack Frame> java.io.FileInputStream.readBytes([BII) @ 0x11b8d0 (FileInputStream.java)	256
<class> class java.io.FileInputStream.readBytes([BII) @ 0x1cb5654	0
<Java Local> java.io.FileDescriptor @ 0xcbd078	40
<Java Local> java.io.FileInputStream @ 0xcfe058	24
<Java Local> byte[8192] @ 0xcfeba0 .....	8,208
Σ Total: 4 entries	
<Java Stack Frame> java.io.BufferedInputStream.fill()V @ 0x11b8ec (BufferedInputStream.java:229)	28
<Java Stack Frame> java.io.BufferedInputStream.read1([BII) @ 0x11b910 (BufferedInputStream.java:269)	36
<Java Stack Frame> java.io.BufferedInputStream.read([BII) @ 0x11b93c (BufferedInputStream.java:328)	44
<Java Stack Frame> com.ibm.jvm.io.ConsoleInputStream.read([BII) @ 0x11b964 (ConsoleInputStream.java:177)	40
<Java Stack Frame> com.ibm.jvm.io.ConsoleInputStream.read([B) @ 0x11b978 (ConsoleInputStream.java:167)	20
<Java Stack Frame> com.ibm.jvm.io.ConsoleInputStream.read()I @ 0x11b98c (ConsoleInputStream.java:160)	20
<Java Stack Frame> org.eclipse.mat.tests.CreateSampleDump.main([Ljava/lang/String;)V @ 0x11b9a8 (CreateSampleDump.java:31)	28
lock java.lang.Thread\$ThreadLock @ 0xcb1718	16
threadGroup java.lang.ThreadGroup @ 0xcb1728 main	72
threadLocals java.lang.ThreadLocal\$ThreadLocalMap @ 0xcb1c10	24
name java.lang.String @ 0xcb1e38 main	32
contextClassLoader sun.misc.Launcher\$AppClassLoader @ 0xcfd6d0 System Class	104
accessControlContext java.security.AccessControlContext @ 0xcfd6a8	24
Σ Total: 15 entries	

下图的 Histogram 视图中只是展示了将运行的方法视为伪类的信息，而类对象的 size 是 0:

Class Name	Objects	Shallow Heap	Retained Heap
<Numeric>	<Numeric>	<Numeric>	<Numeric>
com.ibm.misc.SignalDispatcher.waitForSignal()I	1	256	
java.io.FileInputStream.readBytes([BII)I	1	256	
java.io.BufferedInputStream.read([BII)I	1	44	
com.ibm.jvm.io.ConsoleInputStream.read([BII)I	1	40	
java.io.BufferedInputStream.read1([BII)I	1	36	
com.ibm.misc.SignalDispatcher.run()V	1	32	
java.io.BufferedInputStream.fill()V	1	28	
org.eclipse.mat.tests.CreateSampleDump.main([Ljava/lang/String;)V	1	28	
com.ibm.jvm.io.ConsoleInputStream.read([B)I	1	20	
com.ibm.jvm.io.ConsoleInputStream.read()I	1	20	
<b>Total: 10 entries (415 filtered)</b>	<b>10</b>	<b>760</b>	

#### 4. 将栈帧视为伪对象，并将所有的方法视为伪类

对外引用树和上一种设置是一样的，但是在类直方图中多了许多实例为 0（即没有在运行）的伪类，同时伪类对象的 size 是一个非 0 的值。

Class Name	Objects	Shallow Heap	Retained Heap
<Numeric>	<Numeric>	<Numeric>	<Numeric>
com.ibm.misc.SignalDispatcher.waitForSignal()I	1	256	
java.io.FileInputStream.readBytes([BII)I	1	256	
java.io.BufferedInputStream.read([BII)I	1	44	
com.ibm.jvm.io.ConsoleInputStream.read([BII)I	1	40	
java.io.BufferedInputStream.read1([BII)I	1	36	
com.ibm.misc.SignalDispatcher.run()V	1	32	
java.io.BufferedInputStream.fill()V	1	28	
org.eclipse.mat.tests.CreateSampleDump.main([Ljava/lang/String;)V	1	28	
com.ibm.jvm.io.ConsoleInputStream.read()I	1	20	
com.ibm.jvm.io.ConsoleInputStream.read([B)I	1	20	
java.lang.Object.<init>()V	0	0	
java.lang.Object.equals(Ljava/lang/Object;)Z	0	0	
java.lang.Object.finalize()V	0	0	
<b>Total: 13 of 5,124 entries (415 filtered)</b>	<b>10</b>	<b>760</b>	

## 8. 分析 Java 集合的使用

java 中的集合使用存取并操作数据的对象。Memory Analyzer 提供了如下查询工具来分析 java 集合：

<b>Array Fill Ratio Query</b>	打印一个非直接类型数组的填充率频率分布。填充率指的是数组中非空元素的比例。因为直接类型数组不好判定是否为空，所以这个查询只对引用数组有效。
<b>Arrays Grouped by Size Query</b>	按 size 对数组进行分组。

<b>Collection Fill Ratio Query</b>	<p>打印指定集合的填充率频率分布。如下集合类型适用于该项查询：</p> <ul style="list-style-type: none"><li>• <code>java.util.ArrayList</code></li><li>• <code>java.util.HashMap</code></li><li>• <code>java.util.Hashtable</code></li><li>• <code>java.util.Properties</code></li><li>• <code>java.util.Vector</code></li><li>• <code>java.util.WeakHashMap</code></li><li>• <code>java.util.concurrent.ConcurrentHashMap\$Segment</code></li><li>• <code>java.beans.beancontext.BeanContextSupport</code></li><li>• <code>java.lang.ThreadLocal\$ThreadLocalMap</code></li><li>• <code>java.util.ArrayDeque</code></li><li>• <code>java.util.HashSet</code></li><li>• <code>java.util.IdentityHashMap</code></li><li>• <code>java.util.PriorityQueue</code></li><li>• <code>java.util.Vector</code></li><li>• <code>java.util.WeakHashMap</code></li><li>• <code>java.util.concurrent.ConcurrentHashMap</code></li><li>• <code>java.util.concurrent.CopyOnWriteArrayList</code></li><li>• <code>java.util.concurrent.CopyOnWriteArraySet</code></li><li>• <code>java.util.concurrent.DelayQueue</code></li><li>• <code>java.util.jar.Attributes</code></li><li>• <code>javax.script.SimpleBindings</code></li></ul> <p>另外，可以通过“<code>collection</code>”，“<code>size_attribute</code>”和“<code>array_attribute</code>”等参数来指定自定义集合类型（非 JDK 集合类型）。</p>
<b>Collections Grouped By Size Query</b>	<p>按 <code>size</code> 对指定集合进行分组，适用于如下集合类型：</p> <ul style="list-style-type: none"><li>• <code>java.util.ArrayList</code></li><li>• <code>java.util.TreeMap</code></li><li>• <code>java.util.HashMap</code></li><li>• <code>java.util.Hashtable</code></li><li>• <code>java.util.Properties</code></li><li>• <code>java.util.Vector</code></li><li>• <code>java.util.WeakHashMap</code></li><li>• <code>java.util.concurrent.ConcurrentHashMap</code></li><li>• <code>java.util.concurrent.ConcurrentHashMap\$Segment</code></li><li>• <code>java.util.ArrayDeque</code></li><li>• <code>java.util.HashSet</code></li><li>• <code>java.util.IdentityHashMap</code></li><li>• <code>java.util.LinkedList</code></li><li>• <code>java.util.PriorityQueue</code></li></ul>

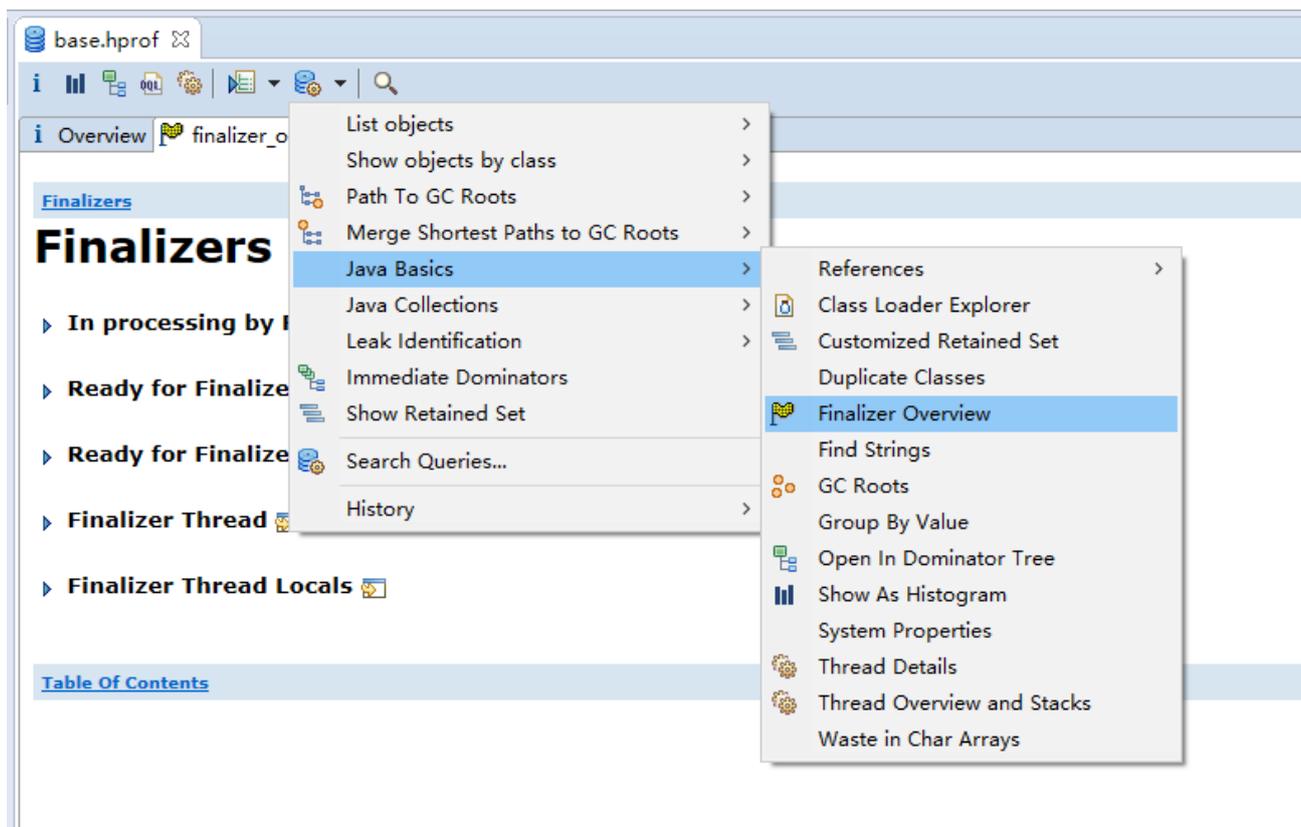
	<ul style="list-style-type: none"> <li>• java.util.TreeSet</li> <li>• java.util.concurrent.ConcurrentSkipListMap</li> <li>• java.util.concurrent.ConcurrentSkipListSet</li> <li>• java.util.concurrent.CopyOnWriteArrayList</li> <li>• java.util.concurrent.CopyOnWriteArraySet</li> <li>• java.util.concurrent.DelayQueue</li> <li>• java.util.concurrent.LinkedBlockingDeque</li> <li>• java.util.concurrent.LinkedBlockingQueue</li> <li>• java.util.concurrent.SynchronousQueue</li> <li>• java.util.jar.Attributes</li> <li>• java.beans.beancontext.BeanContextSupport</li> <li>• java.lang.ThreadLocal\$ThreadLocalMap</li> <li>• javax.script.SimpleBindings</li> <li>• javax.swing.UIDefaults</li> </ul> <p>可以通过“collection”，“size_attribute”和“array_attribute”等参数来指定自定义集合类型（非JDK集合类型）。</p>
<b>Extract List Values Query</b>	列出链表中的元素。适用于 LinkedList, ArrayList, Vector, CopyOnWriteArrayList, PriorityQueue, ArrayDeque。
<b>Hash Entries Query</b>	解析出 HashMap 和 hashtable 中的 key-value 对
<b>Extract Hash Set Value Query</b>	列出一个 HashSet 中的所有元素
<b>Map Collision Ratio Query</b>	<p>打印一个 Map 或类 Map 集合的碰撞率（这里说的是 hash 碰撞）的频率分布。主要适用于如下类型：</p> <ul style="list-style-type: none"> <li>• java.util.HashMap</li> <li>• java.util.Properties</li> <li>• java.util.Hashtable</li> <li>• java.util.WeakHashMap</li> <li>• java.util.concurrent.ConcurrentHashMap\$Segment</li> <li>• java.util.HashMap</li> <li>• java.util.HashSet</li> <li>• java.util.Hashtable</li> <li>• java.util.IdentityHashMap</li> <li>• java.util.concurrent.ConcurrentHashMap</li> <li>• java.util.concurrent.ConcurrentSkipListMap</li> <li>• java.util.concurrent.ConcurrentSkipListSet</li> </ul>

	<ul style="list-style-type: none"> <li>• java.util.jar.Attributes</li> <li>• java.beans.beancontext.BeanContextSupport</li> <li>• java.lang.ThreadLocal\$ThreadLocalMap</li> <li>• javax.script.SimpleBindings</li> <li>• javax.swing.UIDefaults</li> </ul> <p>可以通过“collection”，“size_attribute”和“array_attribute”等参数来指定自定义 Map 或类 Map 类型（非 JDK 集合类型）。</p>
<b>Primitive Arrays with a Constant Value</b>	<p>列出所有的直接类型数组（通过一个 Pattern 或 OQL）并为所有的数组元素提供一个相同的值。</p>

以上查询可以在 Memory Analyzer 工具栏中的下拉菜单 Open Query Browser > Java Collections 下找到，也可以在选定项的右键菜单-> Java Collections 中找到。

## 9. 分析 Finalizer

在 Java 的内部的垃圾回收机制清理对象时会执行 Finalizer。因为无法控制 finalizer 的执行，所以通常建议不要使用它。又因为只有在 finalize 方法执行完成后才能实现内存的释放，所以 finalizer 中执行时间太长的任务会阻塞垃圾回收的完成。可以从 query list 中选择 Finalizer Overview Query 来获取 finalizer overview:



Finalizer Query 中包含以下子查询：

<b>Finalizer in Processing</b>	<p>解压正在被 <b>finalizer</b> 线程处理的对象。如果存在 <b>finalizer</b> 线程的话，这个查询将会返回正在被 <b>finalizer</b> 线程处理的对象。对象能被返回可能出于以下原因：</p> <ul style="list-style-type: none"> <li>• 它（的 <b>finalize</b> 方法）被阻塞住了；</li> <li>• 它（的 <b>finalize</b> 方法）执行时间很长；</li> <li>• <b>finalizer</b> 队列满了。</li> </ul> <p>可以使用 <b>finalizer queue</b> 查询来检查 <b>queue</b>。</p>
<b>Ready for Finalizer Thread</b>	<p>这个查询展示了正在准备执行 <b>finalize</b> 方法的对象。亦即进入了 <b>finalizer queue</b> 的对象。如下原因可能导致 <b>finalizer queue</b> 填满：</p> <ul style="list-style-type: none"> <li>• 当前处理的对象被阻塞住了或执行时间过长（可以使用 <b>Finalizer in Processing</b> 来检查）；</li> <li>• 当前应用中有太多对象使用 <b>finalize()</b> 方法导致它们都在内存中排队。</li> </ul> <p>此外这里还会对对象提供一个类级别的汇总信息。</p>
<b>Finalizer Thread</b>	<p>这个查询会列出正在执行对象 <b>finalize</b> 方法的后台线程。</p>

<b>Finalizer Thread Locals</b>	<p>这个查询将会列出执行对象 <code>finalize()</code> 的后台线程的线程本地变量。如果这里有记录存在，那就暗示着可能出现了 <code>finalizer</code> 的错误（错误实现了 <code>finalize()</code> 方法）使用并有可能导致严重的问题（比如被 <code>finalizer</code> 线程长期持有的无用的内存或者 <code>finalizer</code> 处理的线程本地变量会影响程序的逻辑）。</p>
--	--

## 10. 比较对象

在深入了解 Memory Analyzer 所提供的比较功能前容我们先做些解释。Memory Analyzer 所提供 Heap Dump 上的对象 ID 只是对象被定位到的地址。因为在 GC 期间，对象一直在被移来移去，JVM 记录的地址也一直在变化。所以对象 ID 无法被用来比较对象。也就是说在比较两份不同的 Heap Dump 时，无法对具体的某个或某些对象进行比较，尽管这两份 Heap Dump 来自同一个进程。不过我们仍然可以对某一些聚集结果（比如说 class Histogram）进行比较，来分析对象总量以及所占用的内存的变化。

Memory Analyzer 所提供比较功能不只局限于两份 Heap Dump 文件的全局 histogram 的比较，它可以对任何数量的表格式的结果进行比较——比如说是三个不同对象的 retained set——而不用管这些用来比较的表格是否来自同一个或者不同的 Heap Dump。

这意味着我们可以做这样一些事：

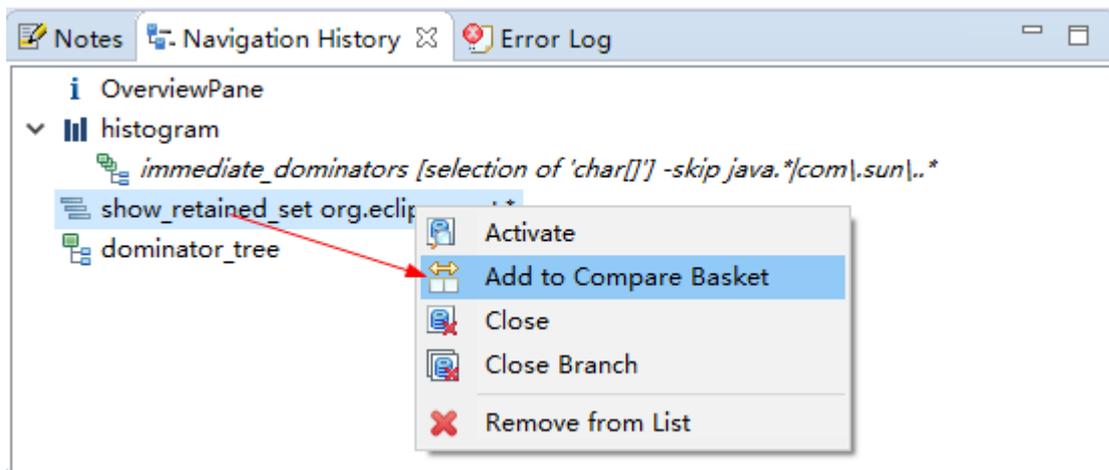
- 比较几个 Heap Dump 上的指定 package 的 retained set；
- 比较同一个应用上的不同对象 A1、A2、A3（同一个 Heap Dump）之间的不同之处。

下面的内容讲解了如何对不同的 table 进行比较。

### 1. 将所有要比较的 table 移到同一个 Compare Basket（都装进菜篮子...）

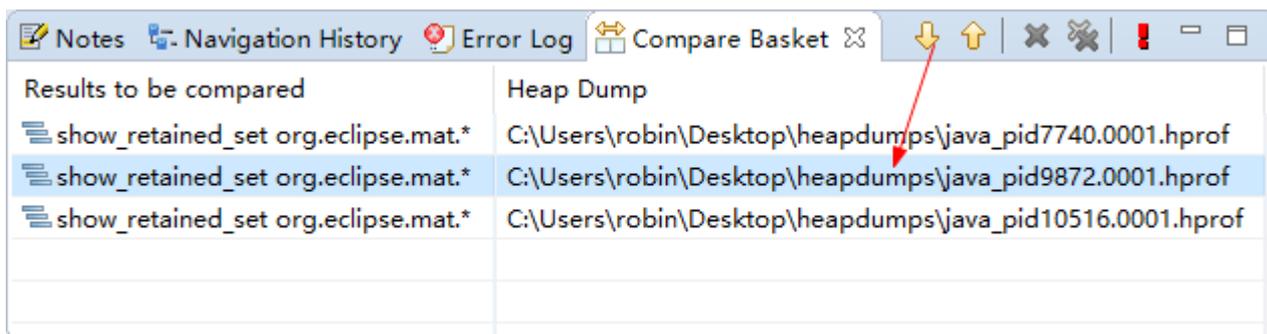
在 Memory Analyzer 上执行过的所有查询都可以在 Navigation History 窗口（需要 Memory Analysis Perspective 下查看）中找到。在这里可以将要比较的结果添加入 Compare Basket。不过 Navigation History 是对应每个 Heap Dump 的，要比较同一个 Heap Dump 上的查询结果可以将之一起添加到 Compare Basket 中，要比较不同 Heap Dump 上的则需要一个一个地添加。

Tree（如 dominator tree）也是可以进行比较的，不过在比较的时候为了方便会将之转为表格。此外 OQL 查询结果也是可以比较的，然而只有每个 OQL 编辑器的最后一次查询结果才可以添加到 Compare Basket 中。如果在同一个 OQL Editor 又执行了新的查询，那么之前的查询结果将会从 Compare Basket 中被抹掉。如果需要比较两个 OQL 的查询结果，那么需要同时打开两个 OQL Editor。



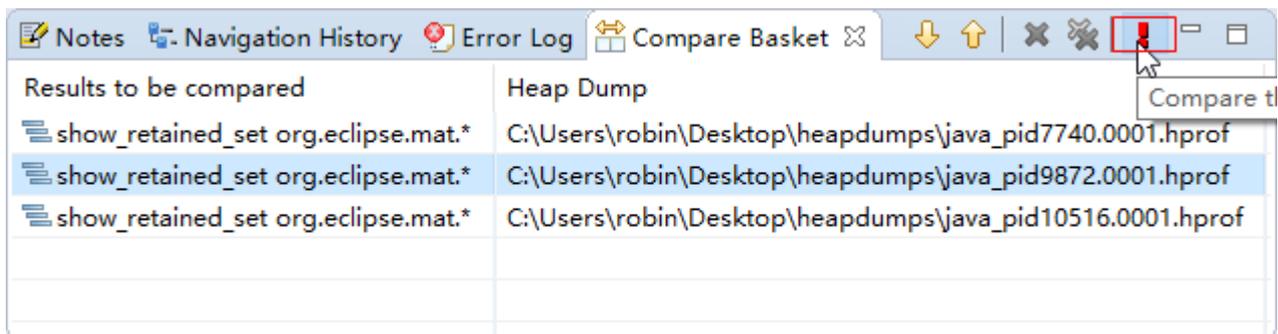
## 2. 调整要查询的表格的顺序

使用 Compare Basket 窗口中的工具条上的顺序调整工具可以调整要比较的表格的顺序，比如那个表格要放在最后面，哪个放在第一个等等：

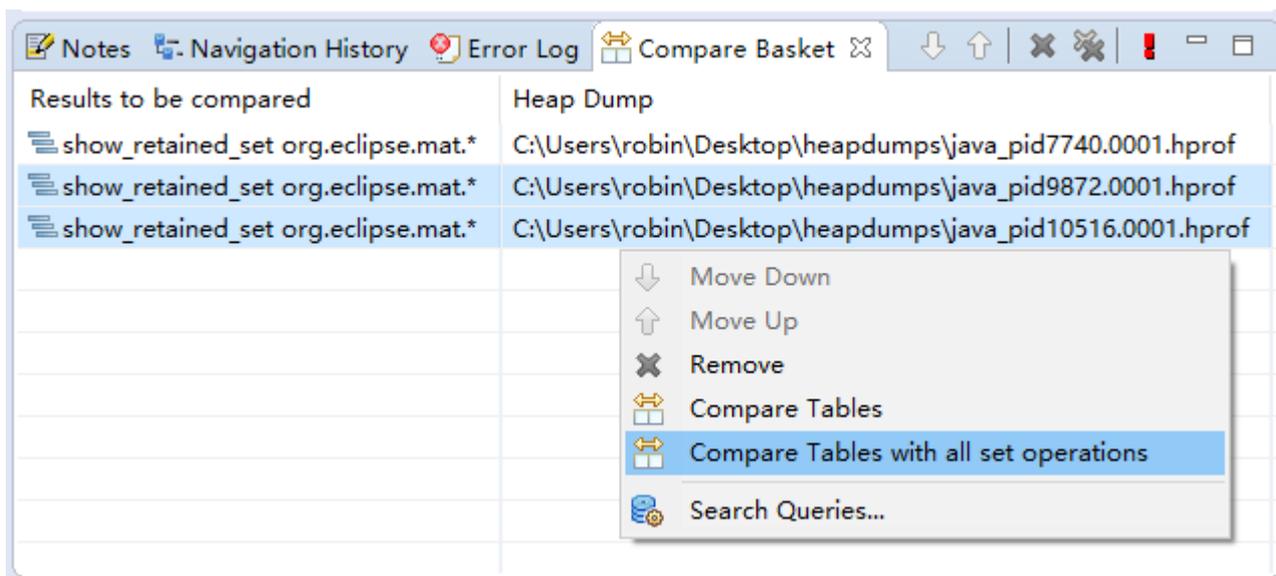


## 3. 执行比较

调整好了顺序后，点击执行按钮执行比较：



要比较待比较表格中的一部分或者一些子集，可以选择要比较的表格，然后在右键菜单中选择相应的选项。此时，如果要比较的表格来自同一个 Heap Dump，可以对比较结果执行一些不同的操作：

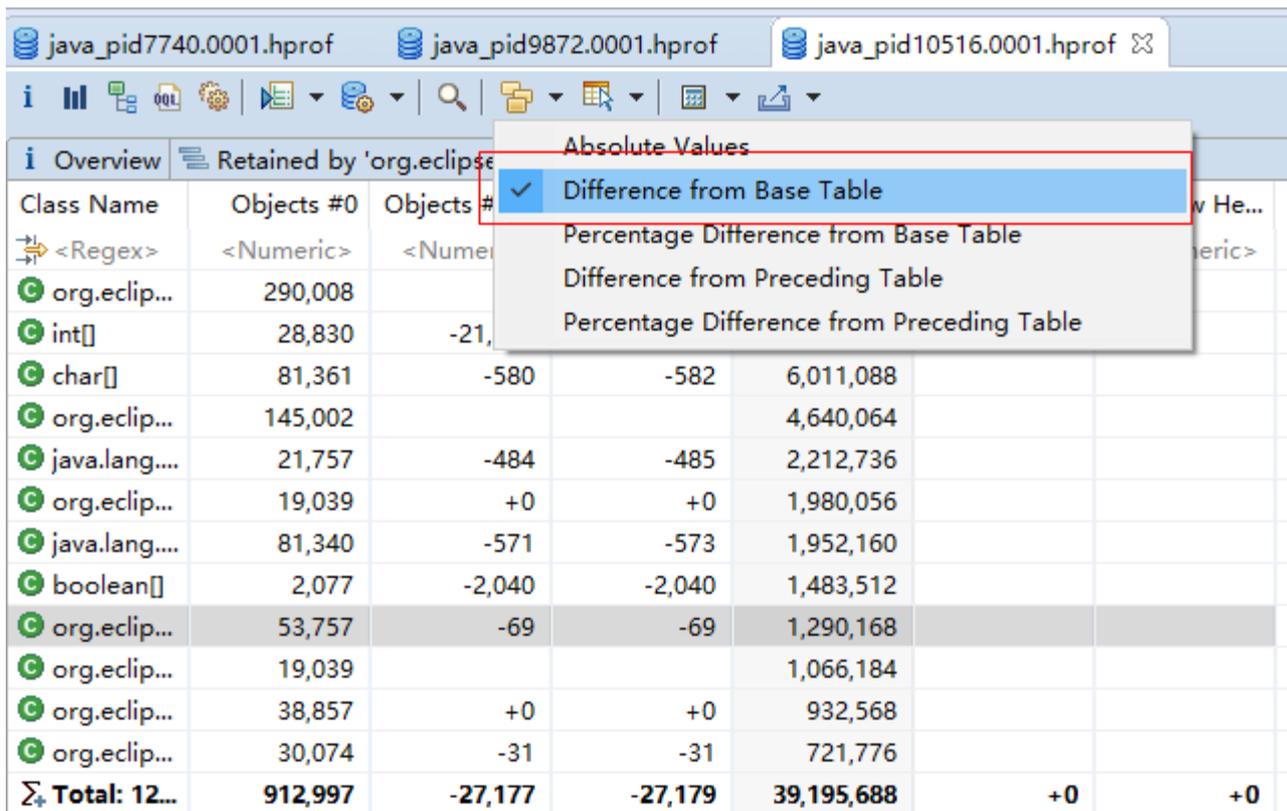


看一下比较结果:

Class Name	Objects #0	Objects #1	Objects #2	Shallow Heap #0	Shallow Heap #1	Shallow Heap #2
<Regex>	<Numeric>	<Numeric>	<Numeric>	<Numeric>	<Numeric>	<Numeric>
org.eclipse.mat.query.Bytes	290,008			6,960,192		
int[]	28,830	7,732	7,732	6,504,944	681,264	681,264
char[]	81,361	80,781	80,779	6,011,088	5,921,136	5,921,360
org.eclipse.mat.internal.snapshot.inspections.Do...	145,002			4,640,064		
java.lang.Object[]	21,757	21,273	21,272	2,212,736	1,359,104	1,359,048
org.eclipse.mat.parser.model.ClassImpl	19,039	19,039	19,039	1,980,056	1,980,056	1,980,056
java.lang.String	81,340	80,769	80,767	1,952,160	1,938,456	1,938,408
boolean[]	2,077	37	37	1,483,512	79,072	79,072
org.eclipse.mat.snapshot.model.Field	53,757	53,688	53,688	1,290,168	1,288,512	1,288,512
org.eclipse.mat.parser.model.XClassHistogramRe...	19,039			1,066,184		
org.eclipse.mat.snapshot.model.FieldDescriptor	38,857	38,857	38,857	932,568	932,568	932,568
org.eclipse.mat.snapshot.model.ObjectReference	30,074	30,043	30,043	721,776	721,032	721,032
<b>Total: 12 of 714 entries; 702 more</b>	<b>912,997</b>	<b>427,939</b>	<b>427,937</b>	<b>39,195,688</b>	<b>47,583,688</b>	<b>47,583,664</b>

#### 4. 自定义要展示的结果

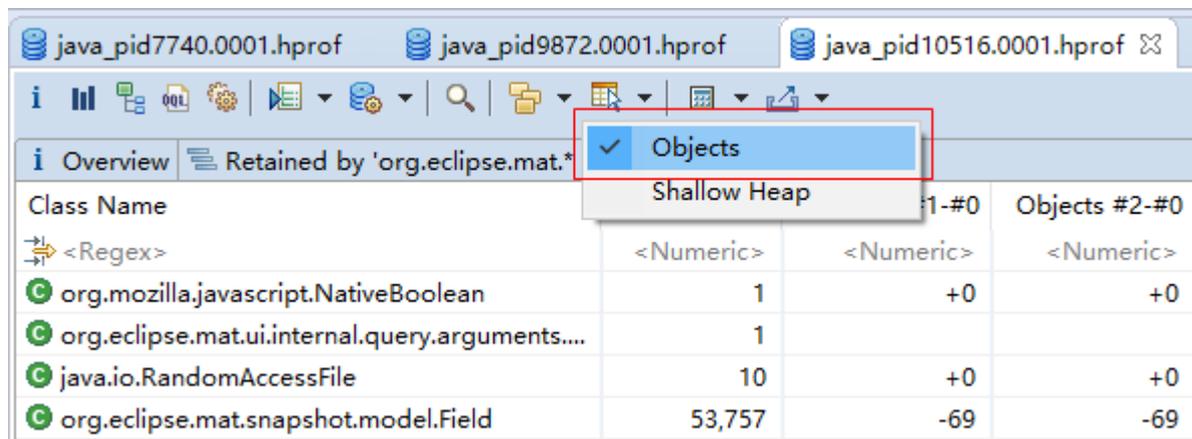
默认情况下在比较结果中展示的是所有比较项的绝对值，比如对象的数量、shallow size 等等。不过用户可以将比较项的值调整为基准差，也可以选择比较哪些列。



The screenshot shows the Memory Analyzer interface with three heap profiles: java\_pid7740.0001.hprof, java\_pid9872.0001.hprof, and java\_pid10516.0001.hprof. A table displays object counts for various classes. A dropdown menu is open over the table, showing options: Absolute Values, Difference from Base Table (selected), Percentage Difference from Base Table, Difference from Preceding Table, and Percentage Difference from Preceding Table.

Class Name	Objects #0	Objects #1	Objects #2	Objects #3	Objects #4	Objects #5	Objects #6
<Regex>	<Numeric>	<Numeric>					
org.eclip...	290,008						
int[]	28,830	-21,					
char[]	81,361	-580	-582	6,011,088			
org.eclip...	145,002			4,640,064			
java.lang...	21,757	-484	-485	2,212,736			
org.eclip...	19,039	+0	+0	1,980,056			
java.lang...	81,340	-571	-573	1,952,160			
boolean[]	2,077	-2,040	-2,040	1,483,512			
org.eclip...	53,757	-69	-69	1,290,168			
org.eclip...	19,039			1,066,184			
org.eclip...	38,857	+0	+0	932,568			
org.eclip...	30,074	-31	-31	721,776			
<b>Σ Total: 12...</b>	<b>912,997</b>	<b>-27,177</b>	<b>-27,179</b>	<b>39,195,688</b>	<b>+0</b>	<b>+0</b>	<b>+0</b>

上图默认以第 0 个要比较的表格为基准取的基准差。



The screenshot shows the Memory Analyzer interface with the same three heap profiles. The table displays object counts for various classes. A dropdown menu is open over the table, showing options: Objects (selected) and Shallow Heap.

Class Name	Objects #1-#0	Objects #2-#0
<Regex>	<Numeric>	<Numeric>
org.mozilla.javascript.NativeBoolean	1	+0
org.eclipse.mat.ui.internal.query.arguments....	1	
java.io.RandomAccessFile	10	+0
org.eclipse.mat.snapshot.model.Field	53,757	-69

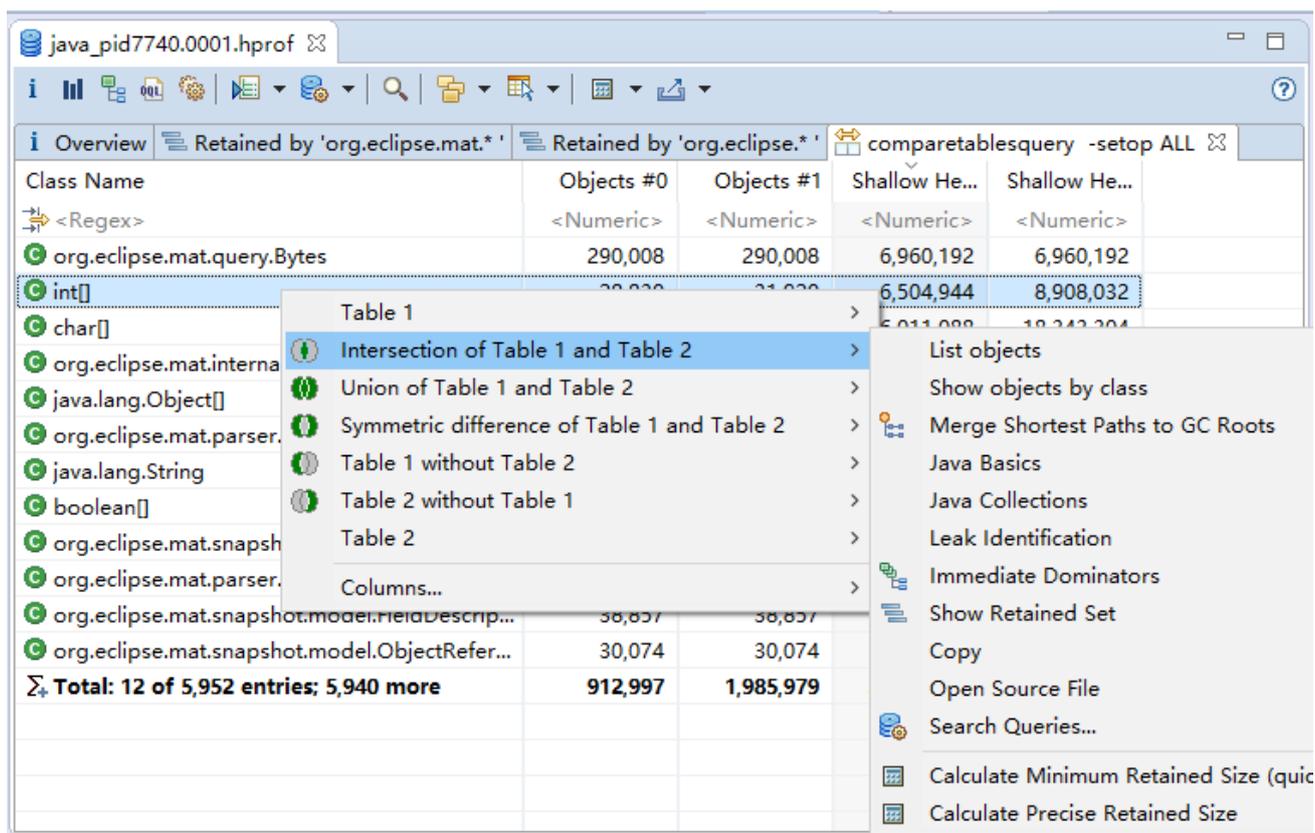
上图选择了只展示 Objects（对象数量）的比较信息。展示的数字仍然是取的基准差。下图是一张大图：

i Overview		Retained by 'org.eclipse.mat.*'		Compared Tables	
Class Name	Objects #0	Objects #1-#0	Objects #2-#0		
<Regex>	<Numeric>	<Numeric>	<Numeric>		
org.mozilla.javascript.NativeBoolean	1	+0	+0		
org.eclipse.mat.ui.internal.query.arguments....	1				
java.io.RandomAccessFile	10	+0	+0		
org.eclipse.mat.snapshot.model.Field	53,757	-69	-69		
org.eclipse.swt.custom.StyledText	2				
org.eclipse.mat.inspections.collections.Hash...	1	+0	+0		
org.eclipse.mat.ui.internal.views.NavigatorVi...	1				
java.lang.Integer	4,010	-10	-10		
java.lang.String[]	17	-8	-8		
org.eclipse.ui.part.PageBookView\$PageRec	2				
java.util.Vector	12				
org.eclipse.mat.dtfj.StackFrameResolver	1	+0	+0		
org.eclipse.mat.inspections.eclipse.EclipseN...	1	+0	+0		
org.eclipse.mat.ui.internal.views.NavigatorVi...	1				
org.eclipse.swt.widgets.Composite	4				
org.eclipse.mat.inspections.ImmediateDomi...	1	+0	+0		
java.util.HashMap\$EntrySet	4	+0	+0		
org.eclipse.mat.parser.index.IndexReader\$P...	3	+0	+0		
org.mozilla.javascript.NativeError	10	+0	+0		
org.eclipse.mat.internal.snapshot.inspection...	6	+0	+0		
org.eclipse.swt.graphics.Cursor	1	+0	+0		
org.eclipse.birt.chart.model.data.impl.TextD...	1	+0	+0		
org.eclipse.birt.chart.computation.DataPoint...	1	+0	+0		

## 5. 右键菜单中的结果集操作

前文应该说过：如果要比较的结果表格来自同一个 Heap Dump 就可以对比较结果执行更丰富的操作。

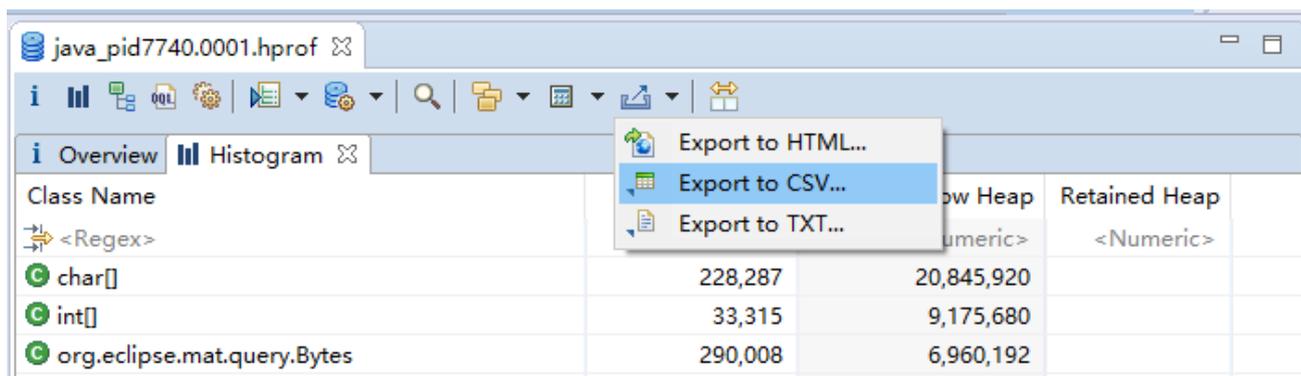
不过还得补充上一点：比较结果是通过右键菜单上的 **Compare Tables with all set operations** 生成的。满足了这两个条件后，在比较结果中点击右键就可以做更多的事了：



## 11. 导出数据

分析数据也还可以导出的：

1. 使用工具栏上的导出按钮，导出格式可以选择（txt、csv 和 HTML 三种格式）：



2. Memory Analyzer 中的视图都是 html 格式的，可以考虑直接复制粘贴，使用 Ctrl+C、Ctrl+V。
3. 也可以使用右键菜单中的 Copy 功能进行复制再粘贴到目标位置上，如果要复制的内容巨多可以使用 Copy -> Save Value to File 选项将内容复制到文件中。

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
org.eclipse.mat.query.Bytes	290,008	6,960,192	
char[]	228,287	20,845,920	
java.lang.String	225,933	5,422,392	
java.lang.Object	196,478	6,287,296	
org.eclipse.mat.query.Query	145,002	4,640,064	
java.lang.String	71,024	3,409,152	
java.lang.Object	60,770	1,944,640	
java.util.ArrayList	56,392	3,047,800	
org.eclipse.mat.query.Query	53,757	1,290,168	
java.lang.String	49,309	3,706,424	
java.lang.Object	42,350	1,016,400	
org.eclipse.mat.query.Query	19,039	1,066,184	
org.eclipse.mat.parser.model.ClassImpl	19,039	1,980,056	
java.lang.Class	17,321	197,752	
<b>Total: 20 of 17,312 entries; 17,292 more</b>	<b>2,085,019</b>	<b>94,352,328</b>	

## 12. Memory Analyzer 配置

### 内存相关项配置

使用 Memory Analyzer 分析较大的 Heap Dump 的时候需要给它多分配些内存（如是在 windows 上，可能需要 64 位的机器），可以在启动 Memory Analyzer 的时候添加些启动参数：

```
MemoryAnalyzer.exe -vmargs -Xmx4g
```

也可以编辑 MemoryAnalyzer.ini 文件，添加如下参数：

```
-vmargs  
-Xmx4g
```

如果是使用的是 eclipse 的 MAT 插件，就需要编辑 eclipse.ini 文件。

使用 Memory Analyzer 时最耗内存的是解析 dump 文件和构建 Dominator Tree。可以尝试在命令行中解析 Heap Dump 文件，也许需要在一个内存更大性能更好的机器上进行。在解析完成后可以将解析生成的 index 文件和 dump 文件再拷贝到更方便的机器上。完成了解析后，在 GUI 中打开就不会再占用太多内存了。我们可以用一个公式进行粗略的估算，假设对象的数量是 N、类的数量是 C，那么在解析 Heap Dump 文件和构建 Dominator Tree 时所占用的内存为 T byte:

$$T \approx N * 28.25 + C * 1000 + P$$

P 是被 DTFJ 和 HPROF 所占用的空间，对于一个 PHD 文件来说，这个空间可能是：

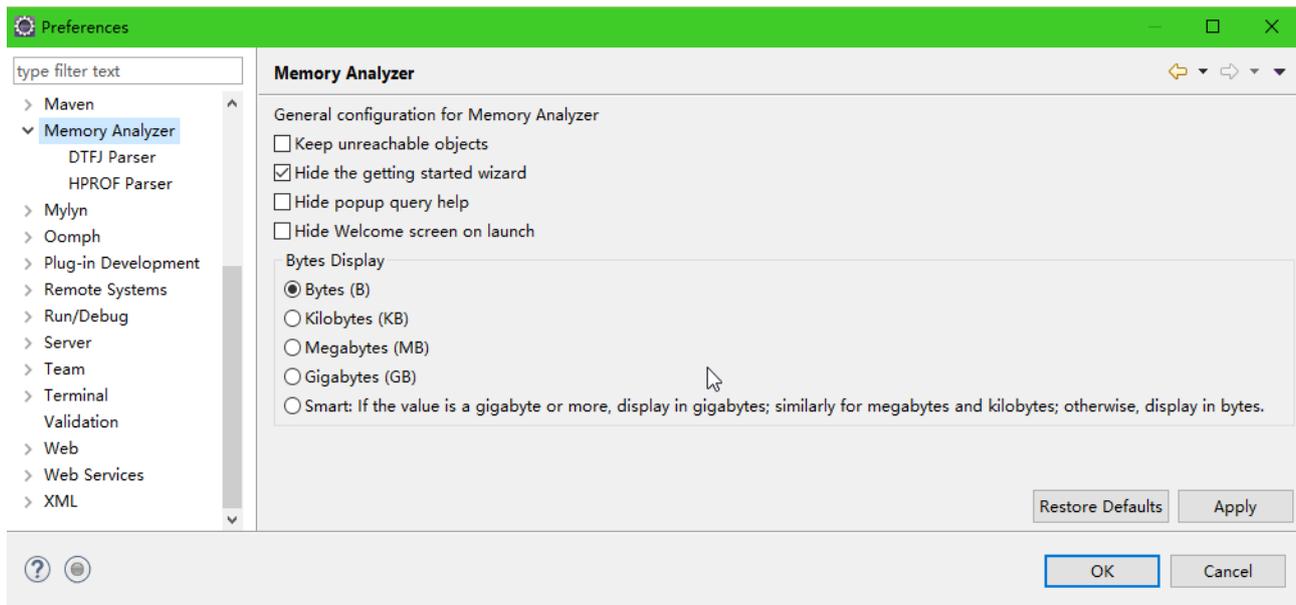
$$P \approx C * 1000$$

此外 Memory Analyzer 还需要一定的空间来缓存解析生成的 index 文件，所以可用的内存空间比解析 Heap Dump 所需的最小空间稍大一些会更好。

Memory Analyzer 支持的对象数量有一个结构上的上限  $2^{31} - 3$ ，目前支持的对象数量上限是  $2^{31} - 8 = 2,147,483,640$ 。不过还没有使用那么多的对象测试过。目前测试过的最大的是一个 48GB 的 dump 文件，其中包含 948,000,000 个对象，使用 MemoryAnalyzer 打开它占用了 58GB 的堆内存。

## 偏好配置

可以在菜单栏选择 Window -> Preference 按钮打开偏好配置窗口，找到 Memory Analyzer 进行配置：



解释下其中的配置项。

**Keep unreachable objects:** 那些看起来对 GC Root 不可达的对象将不会在 Memory Analyzer 早期的处理阶段被回收掉，而是会被保存起来以作进一步的分析。

**Hide the getting started wizard:** 控制是否显示启动时的弹出窗口。

**Hide popup query help:** 不展示查询窗口下面的帮助面板，除非按了 F1 按钮或者帮助按钮。

## Memory Analyzer > DTFJ Parser > Stack Frames 的配置

一般情况下 StackFrame 只在线程视图和栈查询中展示。通过这里的设置可以 stackframe 视为对象，这样就可以在 Memory Analyzer 的其它视图中看到 stackframe。

**Normal:** 只在线程视图和栈查询视图中显示 Stack Frame;

**Only stack frames as pseudo-objects:** Stack Frame 的类型是<stack frame>，size 是 0，同时也会展示方法名、源文件、访问字段所在行号以及一个名称解析器。在 Path to GC Roots 查询中可以找到 Stack Frames 时这个设置非常有用，通过局部变量我们可以找到涉及到进出栈的对象。

**Stack frames as pseudo-objects and running methods as pseudo-classes:** Stack Frame 的类型被视为 packageName.className.methodName(Signature)ReturnType 这样的组合，这个类继承了<Method>，表示正在执行的方法。size 就是 StackFrame 的 size，同时也会展示源文件、访问字段行号以及一个名称解析器。这些正在运行的方法被视为继承了<method type>的伪类，其 size 为 0。这项设置有助于找出正在运行的方法以及它们占用了多少栈空间。设置后，在 Histogram 视图中，使用 “\” 进行过滤，然后按 size 进行排序就可

以找出正在运行的方法。

Stack frames as pseudo-objects and all methods as pseudo-classes: Stack Frame 的类型被视为 `packageName.className.methodName(Signature)ReturnType` 这样的组合, 这个类继承了 `<method>codeph>`, 表示正在执行的方法。size 就是 StackFrame 的 size, 同时也会展示源文件、访问字段行号以及一个名称解析器。所有的方法都被视为继承了 `<method type>` 的伪类, 其 size 基于 JIT 和字节码进行评估。方法的 size 并不是类的 size 的一部分。这个选项有助于找出 JIT 和字节码较大的方法。可以在 histogram 视图里选择 `<method type>` 并列所有对象。

## MemoryAnalyzer > HPROF Parser > Parser Strictness

默认情况下, HPROF 解析器按 strict 模式运行。这意味着任何偏离 [HPROF 标准](#) 的内容都会被视为严重错误并因此中止对 dump 的解析。在这问题上曾经报过一次异常 ([bug #404679](#)), 而 MAT 开发组也为此做了很多工作。

较之以前的版本, 现在 MAT 检测到了这种问题时会在错误日志上记录一个警告并继续处理。这样做只是为了提醒用户 dump 文件或者 JVM 或者 MAT 本身存在一个问题。当然用户也可以选择按着 strict 模式运行 HPROF 解析器。

**strict:** 默认项。任何偏离 HPROF 标准导致的异常都会被抛出并中止 dump 的解析。在命令行中可以使用 `-DhprofStrictnessStop=true` 来指定使用这种模式。

**Warning:** 当发生了问题后不再会中止处理, 只会在错误日志上输出警告信息。不建议使用这个选项, 输出的警告意味着可能存在问题。建议在发生了问题时上报 **BUG**。可以在命令行设置 `-DhprofStrictnessWarning=true` 来使用这个选项。

## General 选项

General > Appearance > Colors and Fonts > Memory Analyzer > OQL comment color: 调整 Object Query Language (OQL) 注释的颜色。

General > Appearance > Colors and Fonts > Memory Analyzer > OQL keyword color: 调整 Object Query Language (OQL) 关键字的颜色。

# 参考

## 1. Inspections

### Component Report

这个报表用来分析组件信息，以发现潜在的导致内存浪费和其他影响程序运行效率的问题。

#### 简介

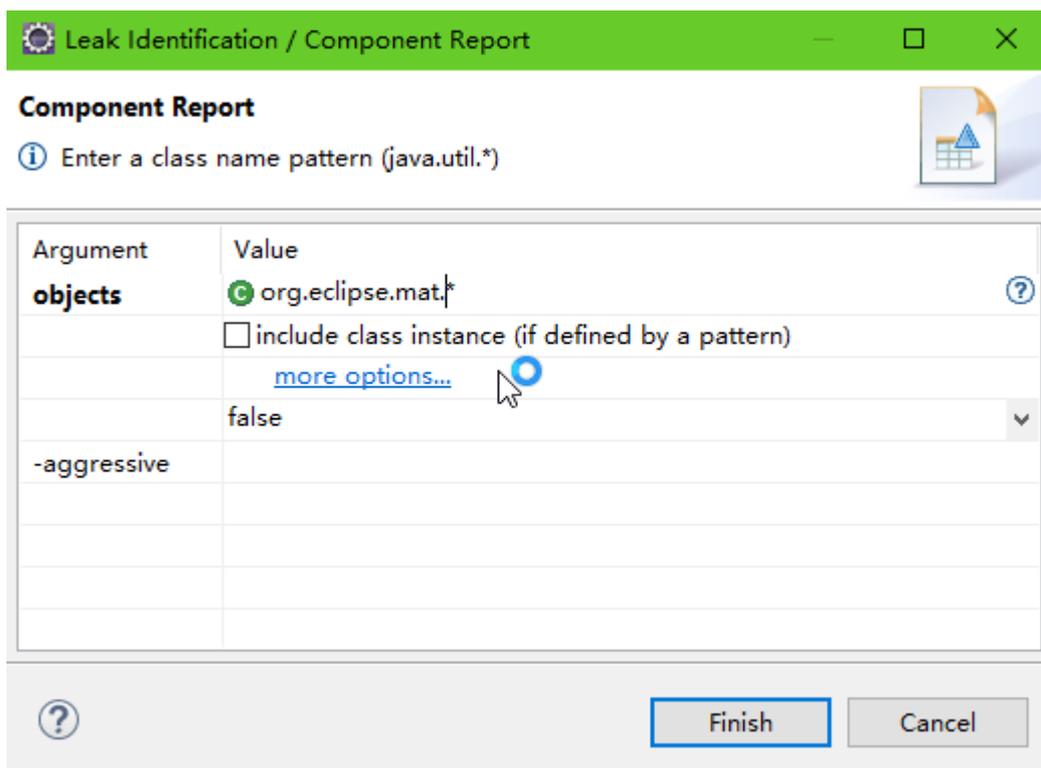
一个 HeapDump 文件中保存了百十万个对象，但是其中哪些对象是属于我们的 Component 的呢？而我们又可以从这些对象中得到什么样的信息？使用 Component Report 可以帮我们解答这些问题。

在开始之前需要先弄明白一个问题：是什么构成了一个 Component？不知道大家是如何理解的。通常一个 Component 指的是同一个包下的所有类的集合或者是由同一个类加载器加载的全部类的集合。

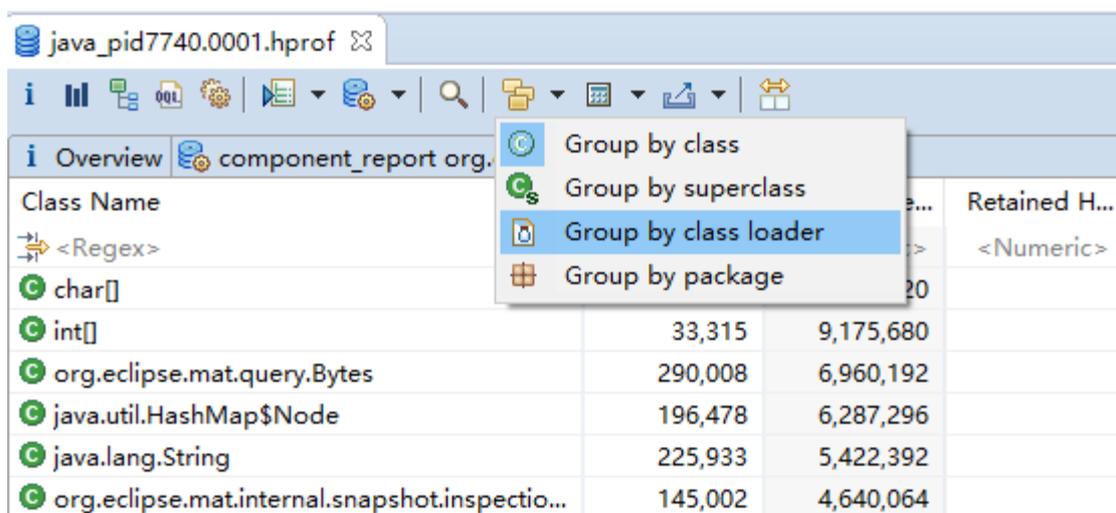
在这里，我们将 Component 类的对象集视为 root set。Component Report 可以计算出 root set 持有的对象集 (retained set)，其中包括全部因为 root set 而保持 alive 的对象。此外，在这个报表中假设所有可回收的对象已经被回收了，这样可以排除那些软引用对象。

#### 运行 Component Report

在工具栏中选择 Query Browser -> Leak Identification-> Component Report 可打开 Component Report，点击菜单项后，在弹出窗口中输入一个类表达式来获取相关的包：



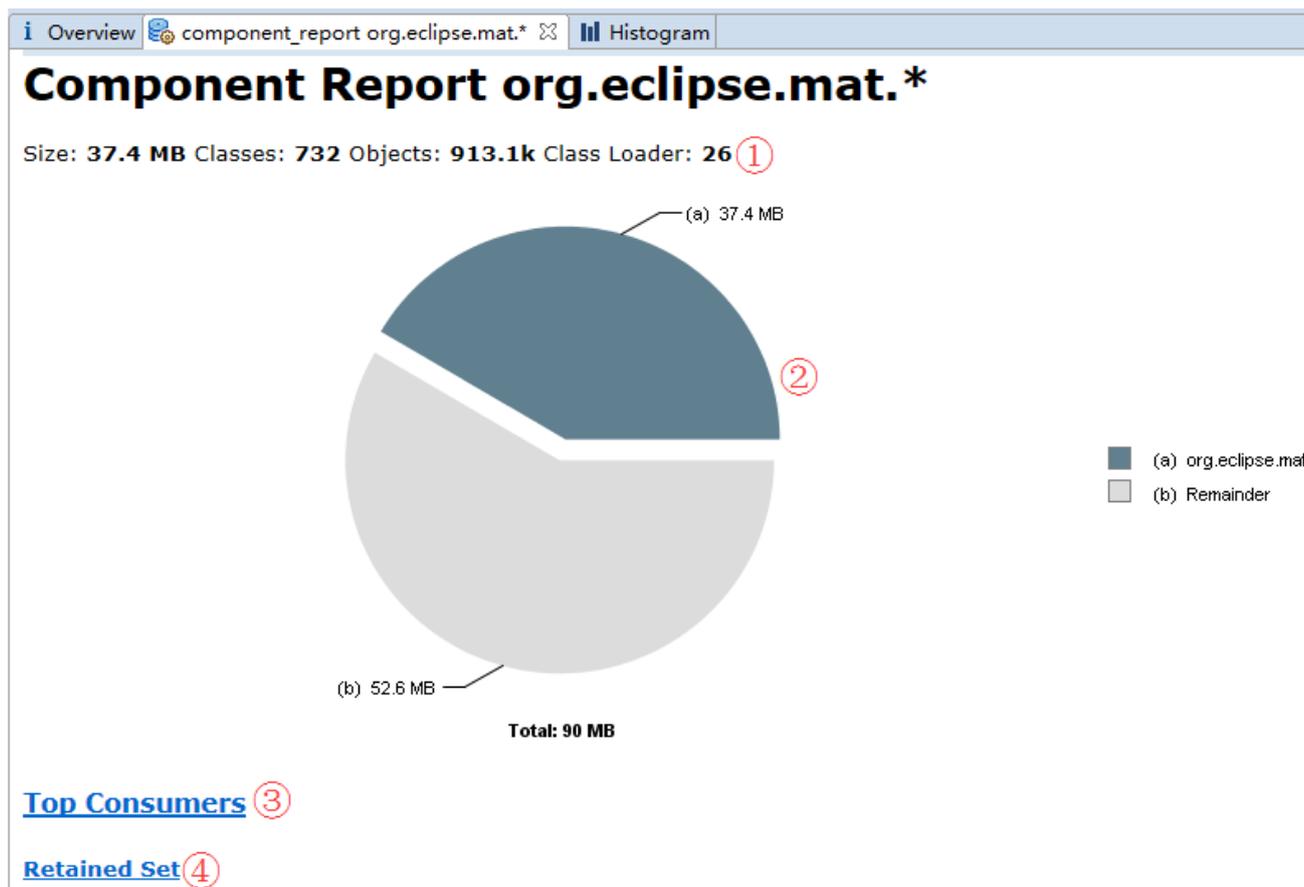
此外也可以从 histogram 视图中进入 Component Report: 在 histogram 视图中选择 Group by class loader, 完成对类加载器分组后选择合适的类加载器, 在右键菜单中选择 Leak Identification -> Component Report 直接进入 Component Report:



## 概览图

Component Report 是被解析为 HTML 格式的, 但是却以.zip 形式和 dump 文件保存在同一个目录下。

注意下下图中用红色数字标识的位置:



- ①：Component 的详细信息，包括 component 的 size，类的数量、对象的数量以及不同类加载器的数量；
- ②：一个饼图，显示了 component 的 size 在整个 heap 中占的比例；
- ③：在 Top Consumer 中会显示 component 持有的最大的对象、类、类加载器和包。在这里可以很好的看出到底是哪些类或对象因为 component 而存活；
- ④：显示 component 持有的对象，按类进行分组。

## Duplicate Strings - 重复的字符串

重复字符串是内存浪费的一个常见案例：多个 char 数组持有完全相同的内容。为了找到重复的内容，component report 按值对相关 char 数组进行分组，并列出了重复总数大于等于 10 的 char 数组。

通过查看 char 数组重复的内容可以得到一些减少重复率的方案：

- 有时候重复的字符串会被用来作为 HashMap 中的 key 或者 value。MAT 在读取 Heap Dump 文件时就常常会使用字符常量代表某一种属性类型，比如 MAT 过去常常使用“L”代表引用类型，“B”代表 byte 类型，“Z”代表 boolean 类型等等。如果使用一个 int 来替换 char，MAT 就可以节省一些

宝贵的内存。此外，也可以使用枚举类来达到同样的效果。

- 在读取 XML 文件时，标签名称、标签内容还有一些片段信息比如“UTF-8”都会保存在内存中。同样的，可以考虑使用枚举类来替换重复的内容。
- 另外一个选项是使用字符串的 `intern()` 方法，这会把字符串送到一个字符串池中。这个字符串池是由 `String` 类维护的。对于值相同的字符串，在池中只会保留一个实例。在向字符串池中 `intern` 字符串时需要注意：`String` 类需要花费代价维护一个很大的字符串池；不要期望 Java 的 GC 机制来回收字符串池中的字符串。

## Empty Collections - 空集合

即使集合是空的，它们内置的对象数组通常也会消费一部分内存。可以想象这么一个树结构：它的每个子节点都迫不及待地创建了数组来存放可能产生的后代节点，但实际上只有极少数节点会有后代节点。

一个补救方案是延迟初始化集合：只在需要的时候才创建集合对象。可以通过 `Dominator Tree` 找出是谁创建了空集合。

## Collection Fill Ratios - 集合填充率

和空集合类似的情况是，一个只有几个元素的集合却占用了大量的内存空间。导致这个问题的元凶也是内置数组。建议调整的思路是可以参考生产环境的 `Heap Dump` 获取的集合填充率，并据此设置集合的初始化空间。

## Soft Reference Statistics – 软引用数据

在需要更多内存的时候，虚拟机会清理软引用。通常，软引用主要是用来实现一种缓存机制：内存充裕的时候就保留软引用对象，内存紧张的时候就将之清理掉。

考虑到重新建对象的开销，一般对象都会被缓存。在整个应用中，不同软引用造成的内存负担也是有着很大区别的。但是，即使虚拟机使用最新的算法也无法对之进行甄别和清理。即使从外部来说，这也是很难预测或调和的。

此外，在垃圾回收阶段软引用还可能会导致“世界停止”。简单地说，Java 的 GC 机制在停止虚拟机后会对软引用进行标记。

## Finalizer Statistics – Finalizer 数据

实现了 `finalize` 方法（的类的）的对象也会在 **Component Report** 中出现，因为这些对象对虚拟机的内存影响很大。

- 每创建一个实现了 `finalize` 方法的对象，也会相应的创建一个 `java.lang.Finalizer` 对象。如果一个对象只对它的 `Finalizer` 可达的话，这个对象就会被放入 `finalizer` 队列并进行处理。只有这样，下次垃圾回收才会真的释放掉相关的内存。因此，要清理一个实现了 `finalize` 方法的对象实际上要用掉两次垃圾回收的机会。
- 在使用 Sun 的虚拟机实现时，`Finalizer` 线程是一个独立的按序处理 `Finalizer` 对象的线程。因此一个 `Finalizer` 阻塞队列经常会占用大量的内存（存放等待被 `finalize` 的对象）。
- 取决于具体的算法，在垃圾回收阶段 `Finalizer` 可能会需要一次“世界停止”。这当然会严重影响整个应用的性能表现。
- 最后但并非最不重要的，`finalizer` 的执行时间是由虚拟机决定的，因此也是不可预测的。

## Map Collision Ratios – Map Hash 冲突率

这一节主要分析下 `HashMap` 的 hash 冲突率。`Map` 根据 `key` 的 `hashCode` 将 `value` 放入不同的 `Bucket` 中。如果两个或多个 `key` 的 `hash code` 指向了同一个 `bucket`，那么就会需要进行一次线性比较。

较高的 `Hash` 冲突率意味着对 `hash code` 的分布进行一次局部优化。这并非是一个内存问题（较好的 `hash code` 分布并不能节省内存），而是性能问题：在比较极端的情况下，`HashMap` 的使用会更像一个链表。

## Immediate Dominators

`Immediate Dominators` 主要用来找出是谁使大量的对象处于存活状态。

## 说明

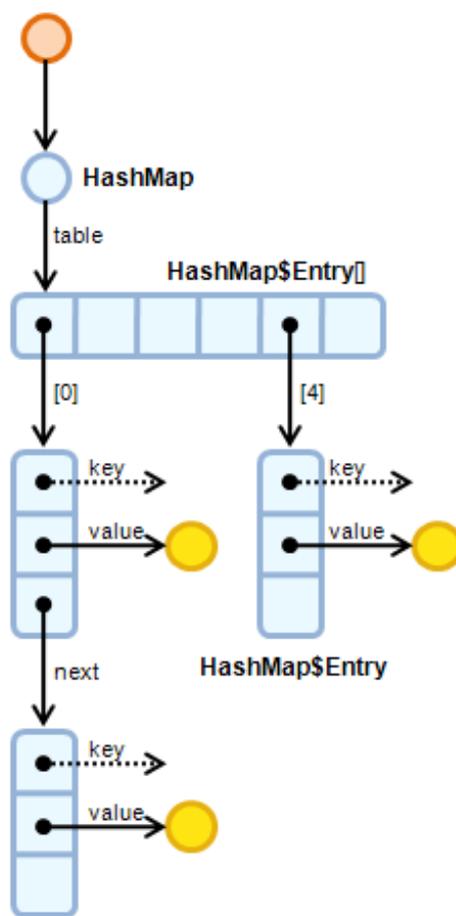
想要找出 heap 中的某个对象存活的原因还是比较简单的，直接使用 Path to GCRoots 找到到 GCRoot 的最短路径就可以了。但是如果几千个对象呢，挨个去找就太费时间了。此时可以使用 Immediate Dominator 来高效地找出是谁保证了对象的存活。

看一下左侧的对象关系图：蓝色标记的对象  表示了一个 java.util.HashMap，其中包括了：对象自身、内部数组、数组中的 bucket、最后是 HashMap 中的 Entry 以及其指向 key 和 value 的引用。黄色标记的对象  指代的是存储在 HashMap 中的 value，比如说是 String。红色标记的对象  持有了对图中 HashMap 对象的引用，也就是它阻止了这个 map 对象被回收。

在这个例子中，Dominator Tree 的结构和右侧对象关系图的结构是一致的。但是需要记住 Dominator Tree 和对象关系图的结构常常是完全不一样的。

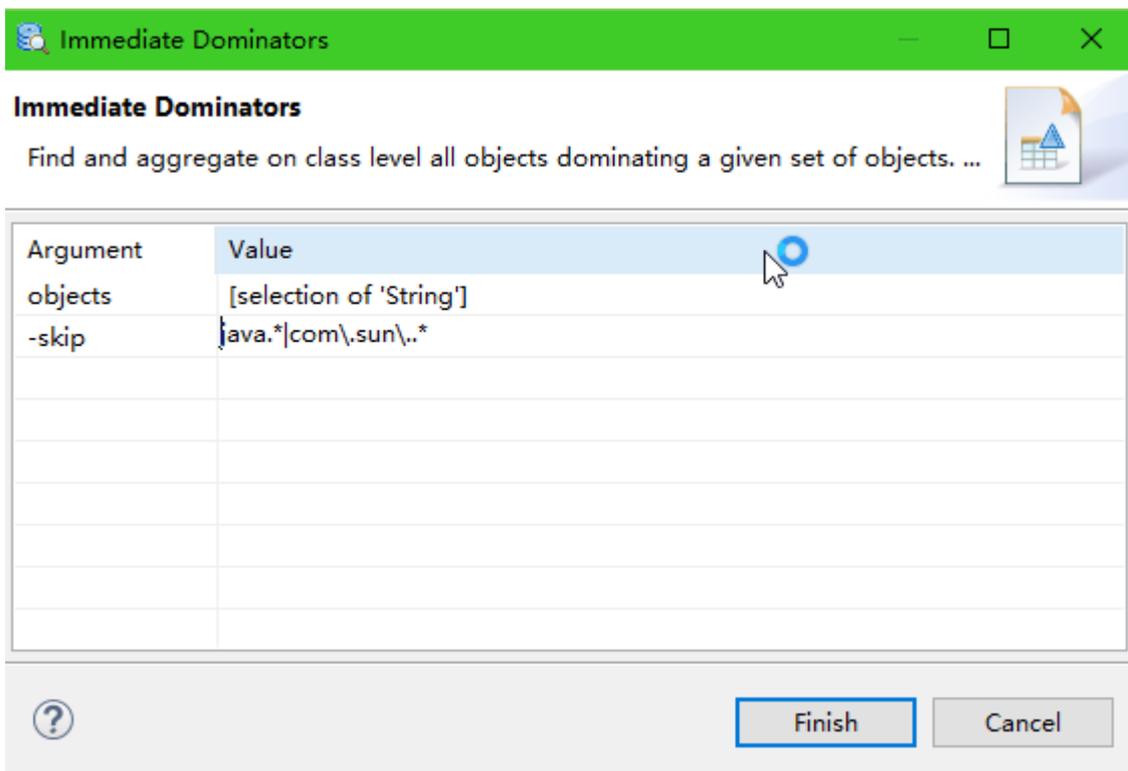
黄色标记对象的 Immediate Dominator 是 HashMap 的 Entry 对象，如果 Entry 的引用都没有了，那么黄色标记的对象也面临着被回收。

在 histogram 中点击对象，右键菜单选择 Immediate Dominator，弹出窗口参数中的 skip pattern 告知 MAT 在查询时跳过匹配 pattern 的 Immediate Dominator。在查看前面的示例图时，如果使用的是默认 pattern 值 “java.\*|com\.sun\.\*”，那么就会跳过所有的蓝色 HashMap 相关对象，直抵红色目标对象。在结果表中就可以看到：是这个红色对象保证了 3 个黄色对象的存活。



## 参数

查看 Immediate Dominator 时会弹出一个窗口要求配置参数：



下表解释了参数:

参数	描述
objects	任意一组要分析的对象
-skip	一个表达式，指出了从要分析的对象上溯 Dominator Tree 时要跳过哪些 dominator。如果一个 immediate dominator 匹配这个表达式，就会取这个 dominator 的 dominator，依此向上推，直到找到一个不匹配这个表达式的 dominator。 如果对象不被任何其它对象支配，那么它就会被归类为 ROOT。

## 结果

下图展示了某个 HeapDump 中所有字符串的 Immedia Dominator:

Class Name	Objects	Dom. Objec...	Shallow He...	Dom. Shall...
<Regex>	<Numeric>	<Numeric>	<Numeric>	<Numeric>
<ROOT>	1	76,225	0	1,829,400
org.eclipse.mat.snapshot.model.Field	36,603	36,603	878,472	878,472
org.eclipse.mat.snapshot.model.FieldDescriptor	14,516	14,516	348,384	348,384
org.eclipse.osgi.internal.loader.classpath.ClasspathEntry	69	13,993	2,208	335,832
org.eclipse.osgi.internal.loader.EquinoxClassLoader	229	13,359	21,984	320,616
org.eclipse.e4.ui.internal.workbench.E4XMIResource	2	3,806	256	91,344
org.eclipse.core.internal.registry.ConfigurationElement	287	2,507	13,776	60,168
sun.net.www.protocol.jar.URLJarFile	3	2,155	240	51,720

看一下在结果表中被选中的那一行，做些说明：共有 36,603 个 Field 对象，保证了 36,603 个 String 对象的存活，每个 Field 对应一个 String。Field 对象占用的空间是 878,472，而其所 dominate 的对象占用的内存也是 878,472。

通常我们可以从每组对象的数量和 shallow size 来判断接下来的分析方向。不过出于性能考虑，在 immedia Dominator 结果表中并没有列出每组对象的 retained size。可以使用右键菜单中 Show RetainedSet 计算每组对象的数量和 retained size。

结果表中的 ROOT 指代的是那些不被其他任何对象 dominate 的对象。这些对象都是常见的实例。这些对象可以比较容易保持存活，因为在引用链上有多条路径经过这些对象最终指向不同的 GCRoot。在 Dominator Tree 上这些 ROOT 对象也经常是根节点。

Class Name	Objects	Dom. Objec...	Shallow He...	Dom. Shall...
<Regex>	<Numeric>	<Numeric>	<Numeric>	<Numeric>
<ROOT>	1	76,225	0	1,829,400
org.eclipse.mat.snapshot.model.Field	36,603	36,603	878,472	878,472
org.eclipse.mat.snapshot.model.FieldDescriptor			348,384	348,384
org.eclipse.osgi.internal.loader.classpath.ClasspathEntry			335,832	335,832
org.eclipse.osgi.internal.loader.EquinoxClassLoader			320,616	320,616
org.eclipse.e4.ui.internal.workbench.E4XMIResource			91,344	91,344
org.eclipse.core.internal.registry.ConfigurationElement			60,168	60,168
sun.net.www.protocol.jar.URLJarFile			51,720	51,720
org.eclipse.rse.services.clientsserver.messages.SystemMessage			50,448	50,448
org.eclipse.emf.ecore.xmi.impl.StringSegment\$Element			47,976	47,976
org.eclipse.core.commands.Command			46,728	46,728
org.eclipse.core.internal.preferences.InstancePreference			46,656	46,656
org.eclipse.core.runtime.spi.RegistryContributor			45,480	45,480
org.eclipse.e4.ui.css.swt.dom.preference.EclipsePreference			45,000	45,000
org.eclipse.core.runtime.Path			41,064	41,064

如上图所示，可以通过右键菜单访问 dominator 的对象集合信息，也可以查看被 dominate 的对象集合

信息。

## Group by Value

按 `value` 对对象集进行分组。

### 用途

这个功能是按对象的值对同一个类的所有对象进行分组，有相同可打印值（不拘泥于对象 `toString()` 的值，也可以是对应的成员变量的值）的对象会被分到同一组中。这有助于对类实例的使用进行分析。

### 参数

参数	描述
<code>objects</code>	任意一组要按值进行分组的对象
<code>-field</code>	一个可选项。可以使用对象的一个成员变量来作为执行对象分组的依据。比如可以使用 <code>HashMap</code> 的 <code>modCount</code> 属性来对 <code>HashMap</code> 对象集进行分组。

## Path to GC Roots

用来找出是谁使一个对象集保持 `alive`。

### 用途

发现 `Heap Dump` 中有消耗内存特别多的对象，想要找出是谁使这个对象保持 `alive` 的就可以使用 `Path to GC Roots` 功能了。

### 参数

参数	描述
<code>object</code>	目标对象，就是为了找它到 <code>GCRoots</code> 的路径

-excludes	特定类的成员变量，在查询的时候可以忽略掉。举例说，一些软引用和弱引用就可以忽略掉
-numberofpaths	要展示的 Path to GC Roots 数目

## Retained Set

所有的因为所选对象而保持 **alive** 的对象的集合。

### 参数

参数	描述
objects	任意一组要进行分析的对象

## Top Consumers

找出是哪些类、类加载器以及包占用了最多的内存。

### 用途

用来找出占用内存最多的几个重要 component

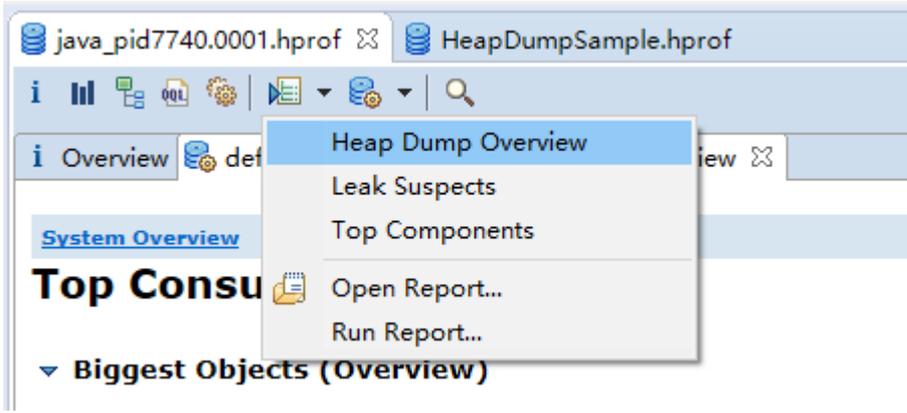
### 参数

参数	描述
objects	要进行分析的对象集
t	一个阈值，占用 heap 的百分比，告诉 MAT 超过这个阈值的都要被纳入分析

## 2. Query Matrix

### overview

内存分析并没有特定的算法。通过 MAT 提供的工具我们可以从不同角度分析 dump 文件，从而了解内存。下表中按用途对 MAT 现有的 HeapDump 查询做了分类：

Histogram	<p>histogram 中列出了 Heap Dump 中所有的对象。在 histogram 中可以看到每个类的对象的数量、retained size 和 shallow size。可以点击表头调整排序顺序。</p> <p>histogram 为进一步的分析算是开了一个好头。可以点击工具栏上的  图标打开 histogram，也可以在 overview 中点击 histogram 的链接打开。</p>
Top Consumers	<p>Top Consumer 查询以 HTML 页的形式返回了整个 Heap Dump 中最大的对象的信息。并以类、类加载器和包进行分组。</p>
Heap Dump Overview (System Overview)	<p>在打开 HeapDump 文件后，看到的 overview 编辑器就展示了对 HeapDump 的初步分析（这里指的是 MAT 的 Overview）：</p> <ul style="list-style-type: none"> <li>➤ 计算了 HeapDump 的大小，对象、类和类加载器的数量；</li> <li>➤ 可以直观的看到最大的对象。</li> </ul> <p>可以在工具栏中打开 HeapDump Overview 报表：</p>  <p>这个 overview 报表中还包含了一个 histogram 以及一个 Top Consumer 报表。</p>

## Finding Memory Leak - 找出内存泄露

下面的四步已经被证明是查找内存问题最高效的方案：

1. 查看 HeapDump Overview（参考上一节）；

2. 找出占用内存最多的一块（单个对象或者对象 group）；
3. 检查这块内存中的内容；
4. 如果这块内存中的内容大得异常的话，找出是谁保持这块内存中的对象 alive 的。

Memory Analyzer 中的 Leak Suspect Report 所做的事情正是如上这几步。

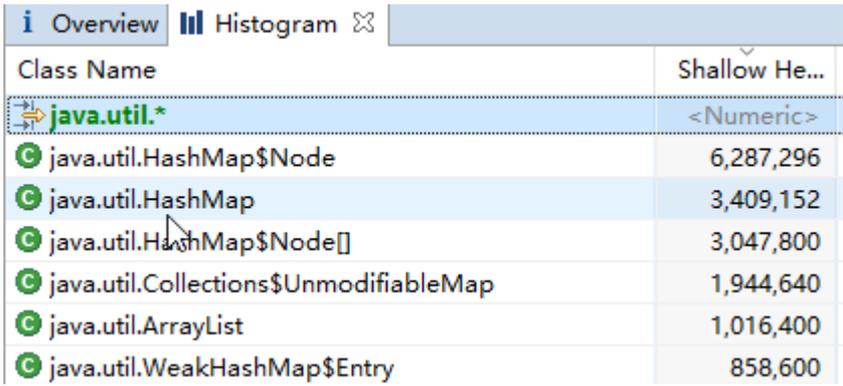
下表中列出了在分析内存泄漏时最重要的几个查询：

Dominator Tree	在 Dominator Tree 中，每个节点都负责保证子节点的存活。因为 Dominator Tree 默认是按 retained size 进行排序的，所以比较容易找到占用内存最多的那个对象。如果无法定位是哪个对象消费内存最多（或者说是消费内存较多的是多个对象），则可以尝试按类或类加载器进行分组。
Top Consumers	Top Consumers 是按类、类加载器和包分别进行查询并返回各自占用内存最多的对象。
Paths to GC Roots	Paths to GC Roots 有助于找出在 heap 中是谁保证了某个对象的存活。通常在发现了可疑对象（内存堆积点）以后再执行这个功能。可以使用 Big Drops in Dominator Tree Query（下面会提到）来找出可疑对象。
Duplicate Classes	列出被加载多次的类。结果按类加载器进行分组，目标是加载同一个类多次的类加载器。问题原因可能是：部署应用时使用了同一个库的多个版本。
Big Drops in Dominator Tree Query	这个功能可以在 Query->Leak Identification-> Big Drops in Dominator Tree Query 中找到。使用这个功能可以找到 Dominator Tree 中的内存堆积点。查询结果中展示的是 retained size 与父节点和子节点存在很大差异的对象。以及在内存累积时第一个值得注意的对象。这些通常都是支配了许多小对象的关键节点。
Leak Suspects Report	Leak Suspects Report 查询分析了 Heap Dump 并尝试找出内存泄漏，最后在生成的报表中对检测到可疑点做了详细说明。

## Analyzing Memory Consumption - 分析内存消费

如果开始着手进行内存优化了，那么 Component Report 是一个不错的起点。此外，如下的查询对进行内存分析也是很有用的。

Class Histogram	通过使用 Class Histogram 中的过滤功能，开发者可以专注于代码中的特定部分
-----------------	--

	<p>进行分析：</p>  <table border="1" data-bbox="478 268 1321 654"> <thead> <tr> <th>Class Name</th> <th>Shallow He...</th> </tr> </thead> <tbody> <tr> <td>java.util.*</td> <td>&lt;Numeric&gt;</td> </tr> <tr> <td>java.util.HashMap\$Node</td> <td>6,287,296</td> </tr> <tr> <td>java.util.HashMap</td> <td>3,409,152</td> </tr> <tr> <td>java.util.HashMap\$Node[]</td> <td>3,047,800</td> </tr> <tr> <td>java.util.Collections\$UnmodifiableMap</td> <td>1,944,640</td> </tr> <tr> <td>java.util.ArrayList</td> <td>1,016,400</td> </tr> <tr> <td>java.util.WeakHashMap\$Entry</td> <td>858,600</td> </tr> </tbody> </table>	Class Name	Shallow He...	java.util.*	<Numeric>	java.util.HashMap\$Node	6,287,296	java.util.HashMap	3,409,152	java.util.HashMap\$Node[]	3,047,800	java.util.Collections\$UnmodifiableMap	1,944,640	java.util.ArrayList	1,016,400	java.util.WeakHashMap\$Entry	858,600
Class Name	Shallow He...																
java.util.*	<Numeric>																
java.util.HashMap\$Node	6,287,296																
java.util.HashMap	3,409,152																
java.util.HashMap\$Node[]	3,047,800																
java.util.Collections\$UnmodifiableMap	1,944,640																
java.util.ArrayList	1,016,400																
java.util.WeakHashMap\$Entry	858,600																
Retained Set	查看对象持有的子对象集合（子对象所属的类以及占用的内存）也许能找到一些优化的方向。																
Collections query group	MAT 提供的几个集合查询功能使我们可以从不同角度对集合进行分析，比如查看集合的填充率、查看集合的 size、查看 HashMap 的 hash 冲突概率等等。可以参考前面的《分析 Java 集合的使用》																
Group by Value	Group By Value 可以按指定字段的值对对象进行分组。这在查找冗余数据的时候还是有用的																
Immediate Dominators	<p>在发现消耗内存很多的可疑对象后，可以使用 Immediate Dominator 找出是谁保持这些可疑对象存活。使用这个查询时还可以跳过一些通用的包比如 java.* 这些包：</p> <pre data-bbox="478 1310 837 1355">-skip java.* com\.sun\ .*</pre>																
OQL	<p>最常见的两种浪费内存的方式：</p> <ul style="list-style-type: none"> <li>➤ 低效率的使用集合和 Map 等数据结构，在内存中保留了数万个十几个空的 list 或 HashMap。使用 OQL 可以很轻松地找出从没有使用过的空集合对象： <pre data-bbox="534 1646 1324 1680">SELECT * FROM java.util.ArrayList WHERE size=0 AND modCount=0</pre> </li> <li>➤ 大量的冗余数据，比如字符串或 char[]，如下的两个例子演示了如何使用 OQL 对字符串进行操作： <pre data-bbox="534 1825 1197 1859">SELECT * FROM java.lang.String s WHERE s.count &gt;= 100</pre> <pre data-bbox="534 1892 1300 1926">SELECT * FROM java.lang.String s WHERE toString(s) LIKE ".*day"</pre> </li> </ul>																

## 3. OQL 语法

Memory Analyzer 有一个内置的对象查询语言（OQL），是我们可以使用自定义的类 SQL 语言查询 Heap Dump。只需要将类视为表、对象视为记录行、成员变量视为表中的字段。基础的查询语法如下：

```
SELECT * FROM [ INSTANCEOF ] <class_name> [ WHERE <filter-expression>]
```

### SELECT 子句

select 子句说明了要从 Heap Dump 中查询哪些信息。要展示对象以及浏览 outgoing reference 可以使用“\*”符号：

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

### select 特定的字段

当然，也可以选择展示类中的一些成员变量：

```
SELECT toString(s), s.count, s.value FROM java.lang.Strings
```

有些潜在的信息没有在结果表中展示出来，可以使用右键菜单作进一步的分析。

要查看一些 Java 属性或对象的方法可以使用“@”符号：

```
SELECT toString(s), s.@usedHeapSize, s.@retainedHeapSize FROM java.lang.String s
```

此外还有大量的内置函数可以使用。

Property Accessors - 属性访问这一节说明了一些常用的属性的详细信息。

### 重命名查询结果列名

可以使用“AS”来使用字段的别名：

```
SELECT toString(s) AS Value,  
       s.@usedHeapSize AS "Shallow Size",  
       s.@retainedHeapSize AS "Retained Size"  
FROM java.lang.String s
```

使用 AS RETAINED SET 关键字来获取选择的对象的 retained set：

```
SELECT AS RETAINED SET * FROM java.lang.String
```

## 将查询项加入对象列表

使用 OBJECTS 关键字将 select 子句中的查询项诠释为对象：

```
SELECT OBJECTS dominators(s) FROM java.lang.String s
```

dominators()函数的返回值是一个对象数组。因此这条查询的结果就是一个对象数组的 list。通过使用 OBJECTS 关键字，使 OQL 将查询结果转译为一个对象列表。

有点说不明白，看个实例好了。不使用 OBJECTS 关键字：

The screenshot shows the Memory Analyzer interface with the OQL query: `SELECT dominators(s) FROM java.util.HashMap s`. The results table has a header `dominators(s)` and a column `<Regex>`. The results are a list of memory addresses: `[I@226e07e7`, `[I@7c31e410`, `[I@9880a15`, `[I@6febec41`, and `[I@3de88f64`.

dominators(s)
<Regex>
[I@226e07e7
[I@7c31e410
[I@9880a15
[I@6febec41
[I@3de88f64

使用 OBJECTS 关键字：

The screenshot shows the Memory Analyzer interface with the OQL query: `SELECT OBJECTS dominators(s) FROM java.util.HashMap s`. The results table has columns: `Class Name`, `Shallow Heap`, and `Retained Heap`. The results are a list of objects with their class names, memory addresses, and heap sizes.

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
> java.util.HashMap\$Node[16] @ 0xeac16938	80	192
> java.util.HashMap\$Node[16] @ 0xeac16878	80	168
> java.util.HashMap\$Node[16] @ 0xeac16730	80	168
> java.util.HashMap\$Node[16] @ 0xeabc87b0	80	632
> java.util.HashMap\$Node[2] @ 0xc5d66840	24	208
> java.util.HashMap\$Node[2] @ 0xc5d66670	24	208
> java.util.HashMap\$Node[2] @ 0xc5d66470	24	208

一图胜千言。

## 使用 DISTINCT 关键字

DISTINCT 关键字用于返回唯一不同的值，或者说是过滤重复值：

```
SELECT DISTINCT * FROM OBJECTS 0,1,1,2,1336
```

“0,1,1,2,2,1336”这几个数字指代的是 MAT 解析后的对象（MAT 为解析出来的每个对象都编了号），OBJECTS 关键字则将这个数字集合转为对象集合，DISTINCT 关键字完成了去重工作。所以查询结果中会有四个对象。

在 select 字句中组合使用 DISTINCT OBJECTS 关键字可以过滤掉很多重复值：

```
SELECT DISTINCT OBJECTS classof(s) FROM java.lang.String s
```

函数 classof()返回的是一个 Class 对象，表示目标对象的类。所有的字符串都有相同的类。OBJECTS 关键字将 classof()的返回结果转换为具体得对象，而 DISTINCT 关键字完成了去重工作，因此这条查询只会返回一行记录。

## 表达式（仍在试验阶段，MAT1.4）

可以在 select 子句中使用表达式，包括字符串的拼接：

```
SELECT s.@objectId, (s.@objectId * 2), ("The object ID is " + @objectId) FROM OBJECTS 0,1,1,2 s
```

在 MAT1.4 中已经支持表达式和子查询了。较复杂的表达式可能会需要使用括号。

## FROM 字句

### 指定类

from 字句定义了要在哪些类上执行查询操作。可以使用以下任意一种方式指定要操作的类。

1. 通过类名：

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

2. 通过一个匹配类名的常规表达式：

```
SELECT * FROM "java\.lang\.*"
```

3. 通过类的一个对象在内存中的地址：

```
SELECT * FROM 0x2b7468c8
```

4. 也可以通过多个类的对象的地址:

```
SELECT * FROM 0x2b7468c8,0x2b74aee0
```

5. 可以通过类的一个对象的 ID:

```
SELECT * FROM 20815
```

6. 也可以通过多个类的对象的 ID:

```
SELECT * FROM 20815,20975
```

7. 通过子查询:

```
SELECT * FROM ( SELECT *  
  
FROM java.lang.Class c  
  
WHERE c implements org.eclipse.mat.snapshot.model.IClass )
```

这条查询语句返回了 heap 中的全部对象。在 where 子句中的判断也是必要的, 因为 Heap Dump 中可能会存在因为反射而生成的 java.lang.Class 实例或者一些代表直接类型的类比如 int.class 或者 Integer.TYPE。

下面的查询语句可以实现同样的效果, 这条语句直接调用了 ISnapshot 对象上的一个方法:

```
SELECT * FROM ${snapshot}.getClasses()
```

## 包含 subClass

使用 INSTANCEOF 关键字可以将子类的对象也纳入查询结果:

```
SELECT * FROM INSTANCEOF java.lang.ref.Reference
```

这条语句的查询结果包含了 WeakReference 和 SoftReference 的对象以及其他对象。因为这两个类都继承自 java.lang.ref.Reference。使用下面的语句可以达到同样的效果:

```
SELECT * FROM ${snapshot}.getClassesByName("java.lang.ref.Reference", true)
```

## 避免将一些术语的解释当做类

如果不想将一些术语当做类处理, 可以使用 OBJECTS 关键字。可以用如下方式指定对象:

1. 通过类名:

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

查询结果是一个 Class 对象, java.lang.String 的 Class 对象。

2. 通过某个对象在内存中的地址:

```
SELECT * FROM OBJECTS 0x2b7468c8
```

3. 通过多个对象的地址:

```
SELECT * FROM OBJECTS 0x2b7468c8,0x2b746868
```

4. 通过某个对象的 ID:

```
SELECT * FROM OBJECTS 20815
```

5. 通过多个对象的 ID:

```
SELECT * FROM OBJECTS 20815,20814
```

6. 通过二级表达式:

```
SELECT * FROM OBJECTS (1 + ${snapshot}.GCRoots.length)
```

## 自动补完

OQL 编辑器现在支持对类名、类名表达式、字段名、属性和方法的自动补全。

## WHERE 字句

### >=, <=, >, <, [ NOT ] LIKE, [ NOT ] IN, IMPLEMENTS (关系型操作符)

where 子句用于指定查询条件,从而将不需要的数据从查询结果中移除。如下的操作符按优先级排序。

操作符的优先级就是如下指定的顺序:

```
SELECT * FROM java.lang.String s WHERE s.count >= 100
```

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

```
SELECT * FROM java.lang.String s WHERE s.value NOT IN dominators(s)
```

```
SELECT * FROM java.lang.Class c WHERE c IMPLEMENTS org.eclipse.mat.snapshot.model.IClass
```

=, !=

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

## AND

```
SELECT * FROM java.lang.String s WHERE s.count > 100 AND s.@retainedHeapSize > s.@usedHeapSize
```

## OR

```
SELECT * FROM java.lang.String s WHERE s.count > 1000 OR s.value.@length > 1000
```

运算符可以用于表达式、常量值和子查询。下一节会说明关于表达式的使用。

## 字面值表达式

下例演示了 Boolean, String, Integer, Long, Character and null 值表达式的使用:

```
SELECT * FROM java.lang.String s
    WHERE ( s.count > 1000 ) = true
    WHERE toString(s) = "monday"
    WHERE dominators(s).size() = 0
    WHERE s.@retainedHeapSize > 1024L
    WHERE s.value != null AND s.value.@valueArray.@length >= 1 AND s.value.@valueArray.get(0) = 'j'
    WHERE s.@GCRootInfo != null
```

## Property Accessors - 属性访问

### 访问对象的 field

要访问对象的 field 只需要使用一个 “.” 符号:

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

alias, 别名, 可以在 from 字句中定义, 以表示当前对象。不使用别名的话, OQL 也会默认选择的 field 为当前对象的 fields 中的一个。fields 就是 Heap Dump 中 java 对象的属性。可以使用 OQL 的自动补完或者 Object Inspector 来找出对象的所有 fields。

## 访问 Java Bean 属性

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

Memory Analyzer 中有一些特殊的对象: 它们由 MemoryAnalyzer 生成, 用来表示 Heap Dump 中的对象, 可以称之为**底层对象**。要使用 OQL 访问底层对象的属性可以使用“@”符号。底层对象的属性可以使用 Bean Inspection 来解析。可以使用 OQL 的自动补全功能来查看常见的 bean 名称。下表中也列出了一些常见的属性:

堆内存中的任何对象	IObject	objectId	对象 ID
		objectAddress	对象在内存中的地址
		class	对象的类
		clazz	对象的 IClass, 也可以参考 classof(object)
		usedHeapSize	shallow heap size
		retainedHeapSize	retained heap size
		displayName	display name
Class object	IClass	classLoaderId	类加载器 ID
任何数组	IArray	length	数组长度
直接类型数组	IPrimitiveArray	valueArray	数组的值
引用类型数组	IObjectArray	referenceArray	数组中的对象(如果值太长的话, 就显示对象的地址)。访问一个特定的元素可以使用 get()。要将之转换为对象可以使用 OBJECTS 关键字。

## 调用 Java 方法

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

这里的方法调用指的是对底层对象的方法的调用。使用圆括号“()”强制 OQL 将<method>认作是一个 java 方法的调用。这种调用是通过反射来进行的。下表中列出了一些常见的底层对象的方法:

<code>#{snapshot}</code>	<code>ISnapshot</code>	<code>getClasses()</code>	一个所有类的集合
		<code>getClassesByName(String name, boolean includeSubClasses)</code>	类的集合
Class object	<code>IClass</code>	<code>hasSuperClass()</code>	如果类有超类的话就返回 true
		<code>isArrayType()</code>	如果类是数组类型的话就返回 true
堆中的任何对象	<code>IObject</code>	<code>getObjectAddress()</code>	一个长整型的值, 表示对象的地址
直接类型的数组	<code>IPrimitiveArray</code>	<code>getValueAt(int index)</code>	从数组中取值
数组或者列表(List, 通过反射获得)	<code>[] or List</code>	<code>get(int index)</code>	从数组或 list 中取值

## 访问数组

在 Memory Analyzer 1.3 及以后的版本中, 可以使用索引直接访问 snapshot 中的数组以及反射方法返回的 Java 数组或 Java List。索引以 0 为基数。如果数组为空, 或索引值超出范围, 返回值也为空。

在 Memory Analyzer 1.4 中允许对数组进行切片式的访问。也就是可以这样使用 `[index1, index2]`, 其中的 `index1` 和 `index2` 都在访问区间内。如果索引值为负数, 就表示从数组尾部开始, 比如 `-1` 就表示要访问数组中的最后一个元素。

## 从直接类型数组中取值

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

这种方式适用于 MAT 1.3 及以后的版本。

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

这种方式适用于所有的版本。这条语句读取了所有长度至少为 3 的 `int[]` 数组中索引为 2 的元素。

## 从引用类型数组中取值

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

这种方式适用于 MAT1.3 及以后的版本。s[2]是一个 IObject 对象，所以也可以访问它的 field 和 Java Bean 属性。

```
SELECT OBJECTS s[2] FROM java.lang.Object[] s
```

这种方式适用于 MAT1.3 及以后的版本。这条语句中使用了 Objects 关键字，将表格式的结果转换成了一个可以通过 Tree 视图访问的对象。这条语句不需要再使用 where 字句过滤可能因为长度不够而返回 null 的数组，因为 OBJECTS 关键字在转换过程中自动跳过了返回为 null 的结果。

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

这种方式适用于 MAT1.1 及之后的版本。这条语句读取了所有长度至少为 3 的 Object[]数组的索引为 2 的元素。代表每个元素的是一个长整型的地址。随后使用 OBJECTS 关键字将之转换为对象。

```
SELECT OBJECTS s.getReferenceArray(2,1) FROM java.lang.Object[] s WHERE (s.@length > 2)
```

这种方式适用于 MAT1.1 及之后的版本。这条语句对所有长度至少为 3 的 Object[]数组进行了切片，切片从索引 2 开始，截取了长度为 1 的部分，返回的是一个每个元素都是长度为 1 的 long[]数组的数组。最后使用 OBJECTS 关键字将之转换为对象。

## 读反射方法返回的 java 数组

```
SELECT s.@GCRoots[2] FROM OBJECTS ${snapshot} s
```

这种方式适用于 MAT1.3 及之后的版本。

```
SELECT s.get(2) FROM OBJECTS ${snapshot} s WHERE s.@GCRoots.@length > 2
```

这种方式适用于 MAT 所有版本。

## 读反射方法返回的 Java List

```
SELECT s.@GCRoots.subList(1,3)[1] FROM OBJECTS ${snapshot} s
```

这种方式适用于 MAT1.3 及以后的版本。

```
SELECT s.@GCRoots.subList(1,3).get(1) FROM OBJECTS ${snapshot} s
```

这种方式适用于 MAT 的所有版本。

## OQL 内置函数

<function>( <parameter>)

内置函数表:

toHex( number )	按 16 进制打印数字
toString( object )	返回一个对象的值，比如一个字符串
dominators( object )	当前对象直接支配的对象
outbounds( object )	目标对象引用的所有其他对象
inbounds( object )	引用目标对象的所有对象
classof( object )	当前对象的类
dominatorof( object )	当前对象的直接支配配置，如果没有的话返回-1
eval( expression )	(在 MAT1.4 中实现) (Experimental in Memory Analyzer 1.4) 对参数表达式进行运算并返回结果。对子查询或表达式的结果进行访问时也许会遇到。

## BNF for the Object Query Language - OQL BNF 范式

SelectStatement	::=	"SELECT" SelectList FromClause ( WhereClause )? ( UnionClause )?
SelectList	::=	(( "DISTINCT"   "AS RETAINED SET" )? ( "*"   "OBJECTS" SelectItem   SelectItem ( "," SelectItem ) * ))
SelectItem	::=	( PathExpression   EnvVarPathExpression ) ( "AS" ( <STRING_LITERAL>   <IDENTIFIER> ) )?
PathExpression	::=	( ObjectFacet   BuiltInFunction ) ( "." ObjectFacet   "[" SimpleExpression ( ":" SimpleExpression )? "]" ) *
EnvVarPathExpression	::=	( "\$" "{" <IDENTIFIER> "}" ) ( "." ObjectFacet   "[" SimpleExpression ( ":" SimpleExpression )? " ] " ) *
ObjectFacet	::=	( ( "@" )? <IDENTIFIER> ( ParameterList )? )
ParameterList	::=	"(" ( ( SimpleExpression ( "," SimpleExpression ) * ) )? ")"
FromClause	::=	

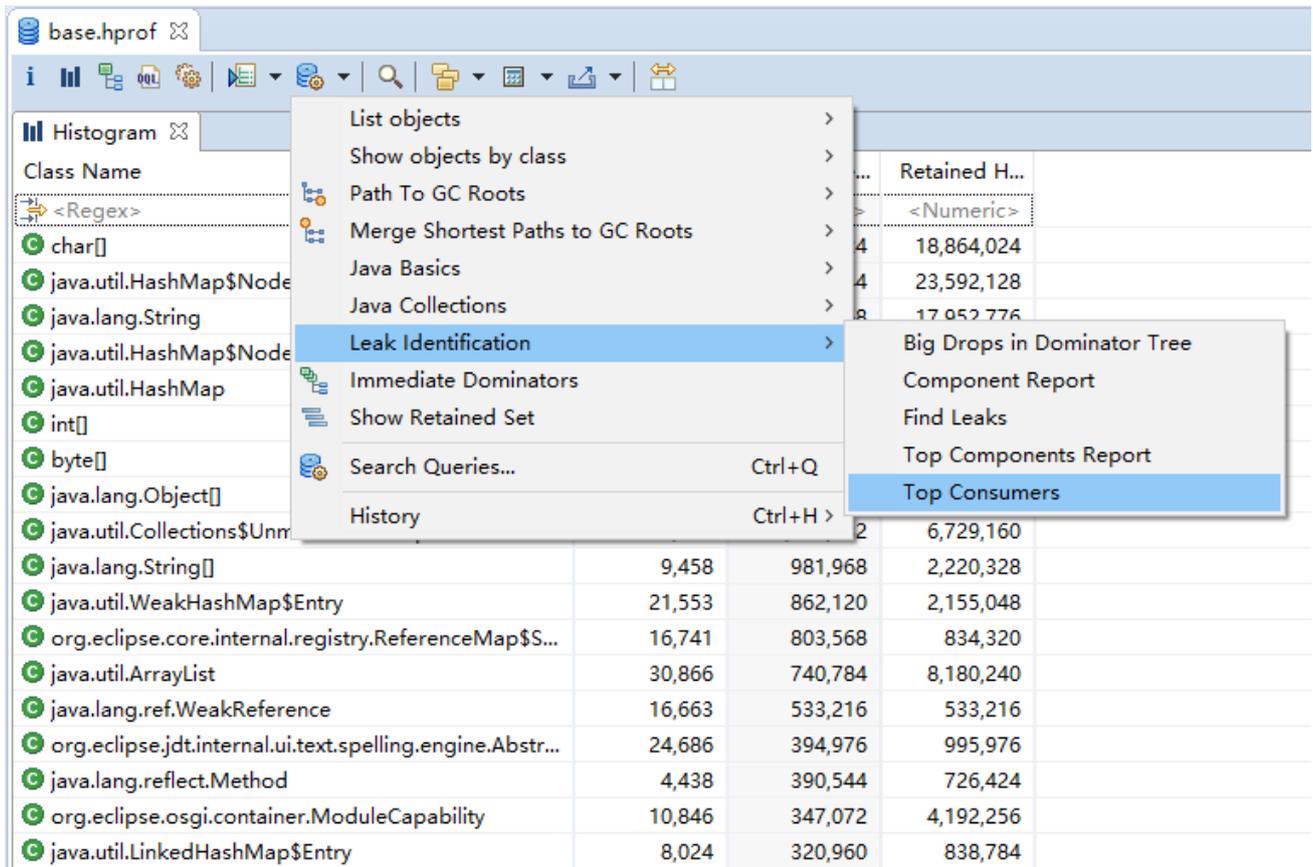
		"FROM" ( "OBJECTS" )? ( "INSTANCEOF" )? ( FromItem   "(" SelectStatement ")" ) ( <IDENTIFIER> )?
FromItem	::=	( ClassName   <STRING_LITERAL>   ObjectAddress ( "," ObjectAddress )*   ObjectId ( "," ObjectId )*   EnvVarPathExpression )
ClassName	::=	( <IDENTIFIER> ( "." <IDENTIFIER> )* ( "[" ]" )* )
ObjectAddress	::=	<HEX_LITERAL>
ObjectId	::=	<INTEGER_LITERAL>
WhereClause	::=	"WHERE" ConditionalOrExpression
ConditionalOrExpression	::=	ConditionalAndExpression ( "or" ConditionalAndExpression )*
ConditionalAndExpression	::=	EqualityExpression ( "and" EqualityExpression )*
EqualityExpression	::=	RelationalExpression ( ( "=" RelationalExpression   "!=" RelationalExpression ) )*
RelationalExpression	::=	( SimpleExpression ( ( "<" SimpleExpression   >" SimpleExpression   "<=" SimpleExpression   >=" SimpleExpression   ( LikeClause   InClause )   "implements" ClassName ) )? )
LikeClause	::=	( "NOT" )? "LIKE" <STRING_LITERAL>
InClause	::=	( "NOT" )? "IN" SimpleExpression
SimpleExpression	::=	MultiplicativeExpression ( "+" MultiplicativeExpression   "-" MultiplicativeExpression )*
MultiplicativeExpression	::=	PrimaryExpression ( "*" PrimaryExpression   "/" PrimaryExpression )*
PrimaryExpression	::=	Literal
		"(" ( ConditionalOrExpression   SubQuery ) ")
		PathExpression
		EnvVarPathExpression

SubQuery	::=	SelectStatement
Function	::=	( ( "toHex"   "toString"   "dominators"   "outbounds"   "inbounds"   "classof"   "dominatorof" ) "(" ConditionalOrExpression ")" )
Literal	::=	( <INTEGER_LITERAL>   <LONG_LITERAL>   <FLOATING_POINT_LITERAL>   <CHARACTER_LITERAL>   <STRING_LITERAL>   BooleanLiteral   NullLiteral )
BooleanLiteral	::=	"true"
		"false"
NullLiteral	::=	<NULL>
UnionClause	::=	( "UNION" "(" SelectStatement ")" )+

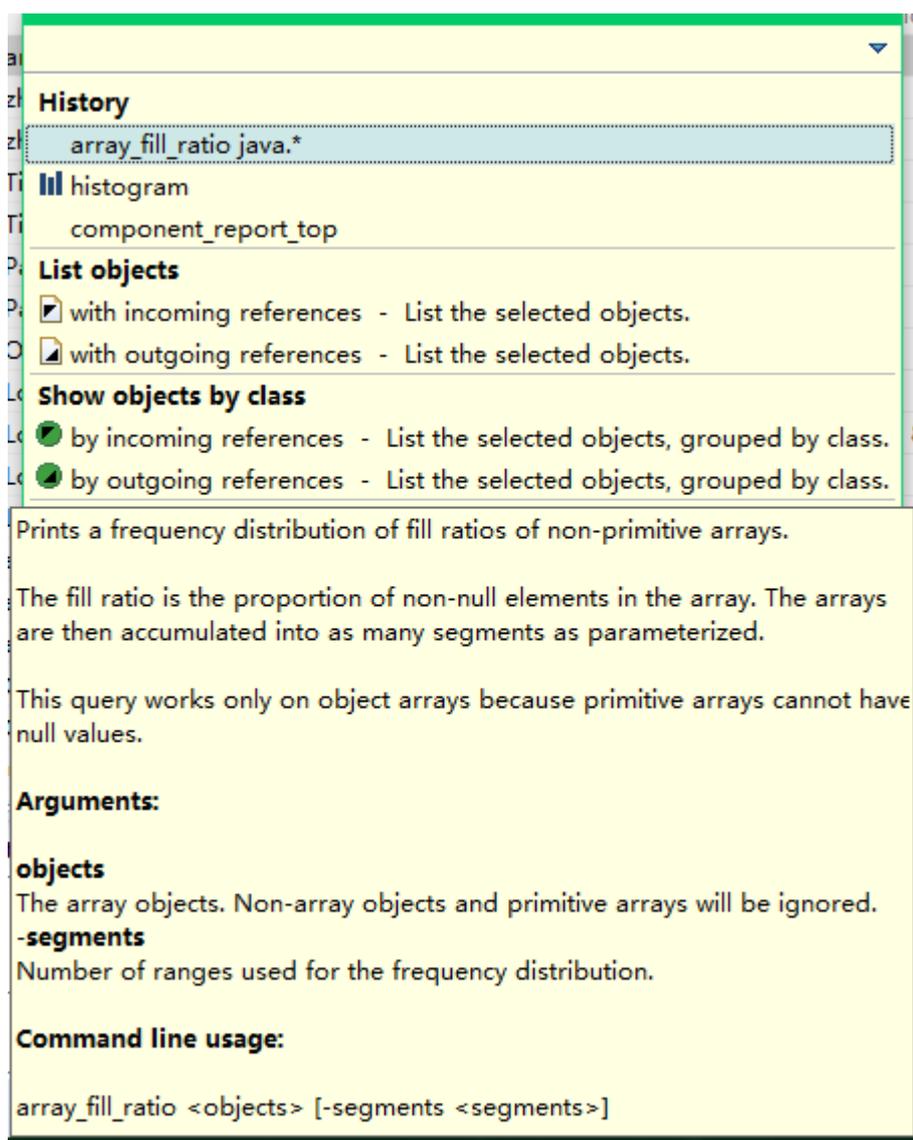
## 4. 选择查询工具

memory Analyzer 提供了一系列的查询工具来分析 HeapDump。这些查询工具被分成了几大类。

可以在工具栏的下拉菜单选择一个查询工具执行查询：



在下拉菜单中还可以查看历史查询（History）项。也可以使用“Ctrl+Q”组合键快速打开查询选择器（也可以使用下拉菜单中的“Search Queries...”项）：



查询选择器中还有一个输入框，在输入框下方可以看到历史执行过的查询。在历史查询下方就是所有可用查询，并做了分类。当选定了一个查询后，就可以在下方看到关于这个查询的详细描述。在描述里也包含了查询的参数信息。

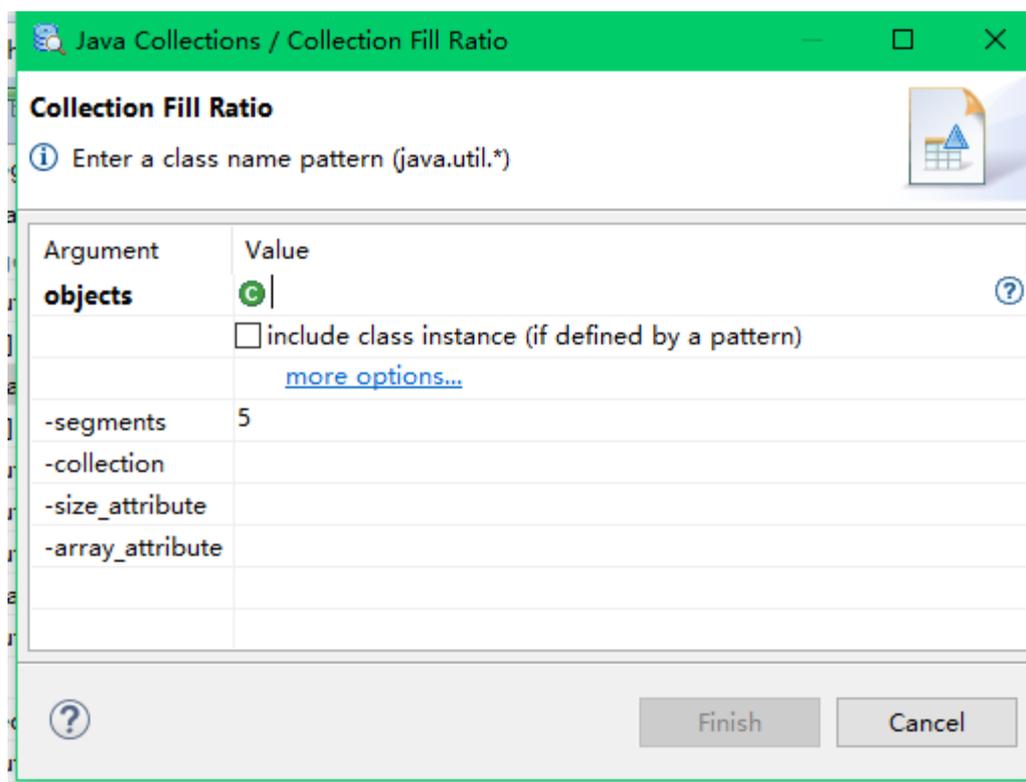
也可以在右键菜单中快速打开“Search Querys...”。这样只可以看到当前选择项可用的查询，此外还可以看到每个查询项的帮助信息。

Class Name	Objects	Shallow He...	Retained H...
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.util.HashMap\$Node	208,867	6,683,744	23,592,128
char[]	184,380	18,864,024	18,864,024
java.lang.String	155,722	2,727,568	17,952,776
byte[]		2	3,409,152
java.util.HashMap		5	28,057,456
java.util.Collections		2	6,729,160
java.util.HashMap		3	26,004,624
java.lang.Object[]		0	25,280,712
java.util.ArrayList		4	8,180,240
int[]		3	3,426,408
org.eclipse.jdt.int		5	995,976
java.util.WeakHas		0	2,155,048
java.lang.Class		4	9,283,592
org.eclipse.core.i		3	834,320
java.lang.ref.Wea		5	533,216
java.util.HashSet		5	3,273,144
org.eclipse.osgi.c		2	4,192,256
java.util.jar.Attributes\$Name	9,661	231,864	234,648

Class Name	Objects	Shallow He...	Retained H...
<Regex>			
java.util.HashMap\$Node			
char[]			
java.lang.String			
byte[]			
java.util.HashMap			
java.util.Collections			
java.util.HashMap\$Node			
java.lang.Object[]			
java.util.ArrayList			
int[]			

如果一个查询有参数的话，那么在选择了这个查询以后会弹出一个窗口。在窗口中会以列表的形式列出所有的参数。其中必填的参数会用粗体显示。在窗口上方会展示必填的参数信息，在下方会有一个类似查询

选择器的描述框:



Prints a frequency distribution of fill ratios of given collections.

The below mentioned collections are known to the query. One additional custom collection (e.g. non-JDK) collection can be specified by the "collection", "size\_attribute" and "array\_attribute" argument.

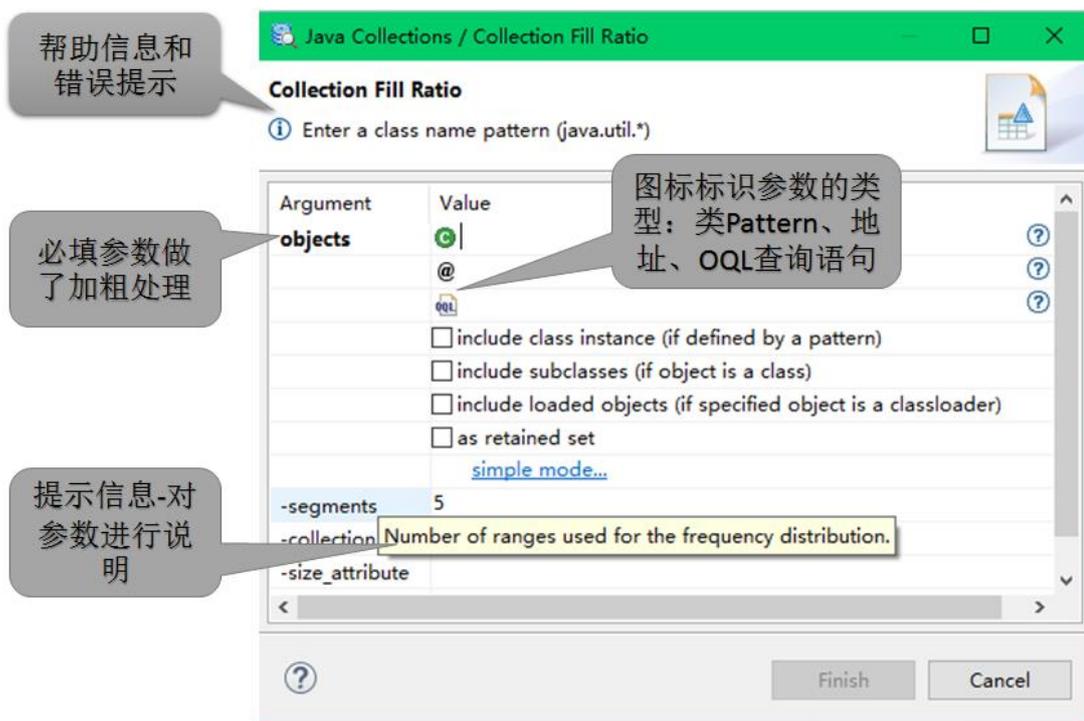
Known collections:

```
java.util.ArrayList
java.util.HashMap
java.util.Hashtable
java.util.Properties
java.util.Vector
java.util.WeakHashMap
java.util.concurrent.ConcurrentHashMap$Segment
```

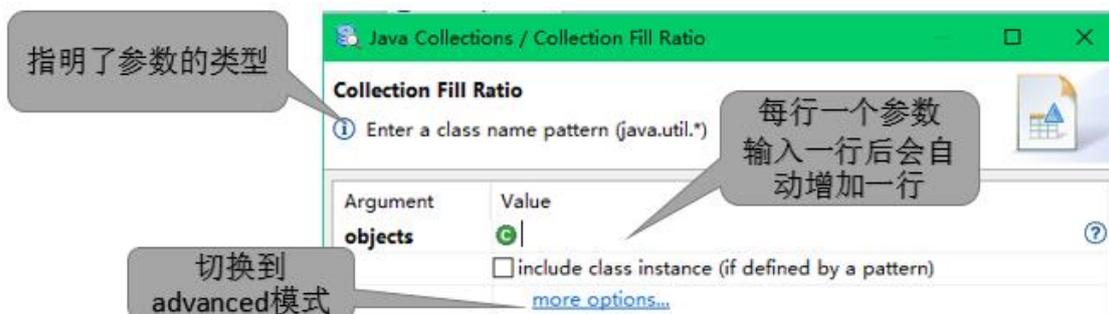
**Arguments:**

## 查询参数

如前文所说，可以在参数窗口中输入参数信息：



必填参数做了加粗处理。帮助和提示信息提供了对参数的描述。对话框的消息区域描述了参数的信息，在输入不合理的值后还会给出错误提示。有时候还会需要传递一组对象参数，此时可以使用参数对话框中的 simple mode 和 advanced mode。下图是 simple mode，以 Pattern 的形式指明了对象组。



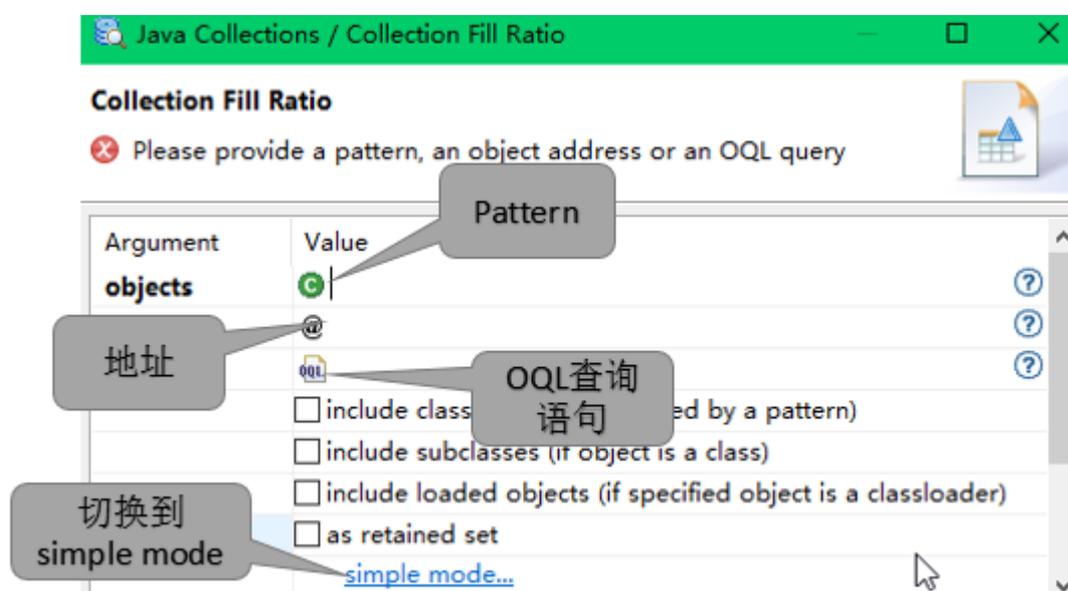
## Pattern 语法

Pattern 支持的通配符包括: (, [, {, \, ^, -, \$, |, }, ), ?, \* and +。

Pattern 语法	详情
[abc]	a, b, 或者 c (标识类的简单方式)
[^abc]	除了 abc 以外的任意字符 (排除方式)
outbounds( object )	引用的外部对象
[a-zA-Z]	a-z 或者 A-Z, 即所有的字母
[a-d[m-p]]	字母集部分组, 或者也可以这样表示 [a-dm-p]

[a-z&&[def]]	d、e 或者 f (取交集)
[a-z&&[^bc]]	a-z 间的字母——除了 b 和 c: [ad-z] (排除法)
[a-z&&[^m-p]]	a-z 间的字母, 不包括 m-p: [a-lq-z] (排除法)
\d	数字: [0-9]
\D	非数字: [^0-9]
\s	空白字符: [\t\n\x0B\f\r]
\S	非空白字符: [^\s]
\w	包括下划线的任意单词字符: [a-zA-Z_0-9]
\W	任何非单词字符: [^\w]

在 advanced 模式下, 可以使用 Pattern、对象地址和 OQL 语句定义参数对象。对话框中的图标和帮助信息提供了关于参数类型的提示:



每行输入一个参数, 参数可以是 Pattern、对象地址或者 OQL 语句。输入完成后, 对应的类型会再增加一行可输入参数, 这样可以输入多个参数。

## 对象地址语法

对象的地址是一个十六进制形式的数字。地址需要以 0x 开头, 比如: 0x36d99c88。

## 5. 图标说明

在 MAT 中使用快捷键 ALT+I 可以得到一个图标说明。下表中也简单介绍了下一些常见的图标:

图标	说明
	数组对象

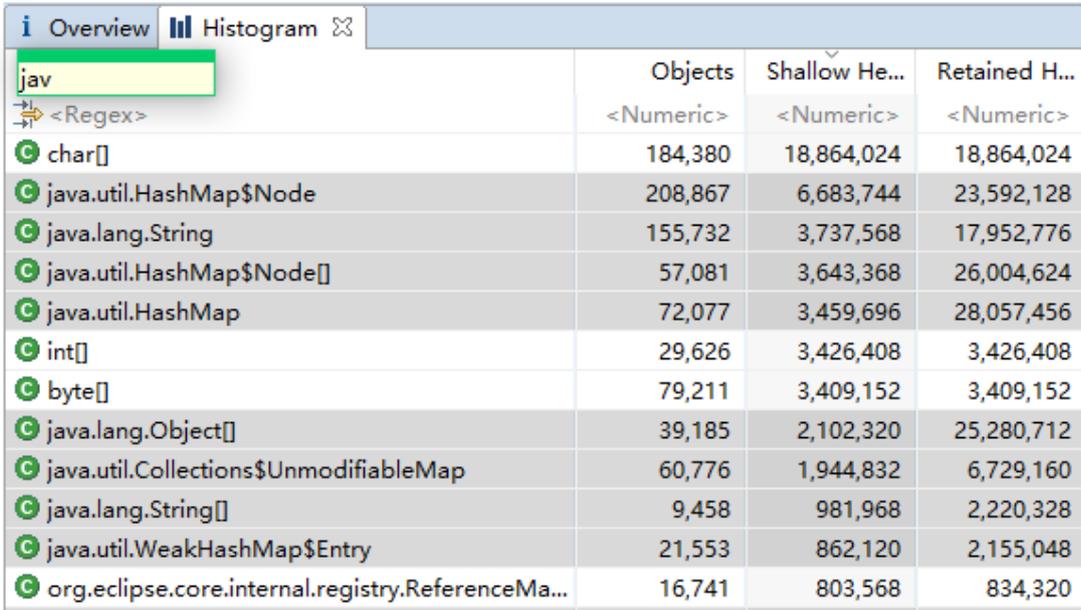
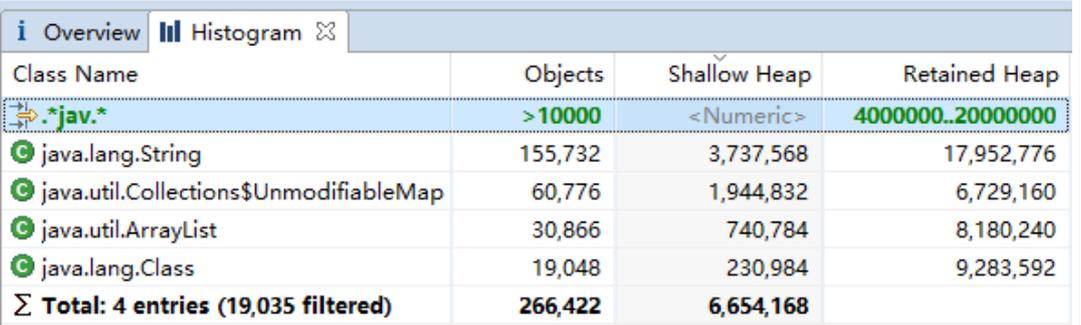
	按类进行分组的实例
	按类进行分组的实例，部分已经展示过了（这个我没见过，只能音译）
	按类进行分组的实例，全部都已经展示过了（这个我也没见过）
	包
	超类
	类对象
	类加载器对象
	Heap Dump
	对象地址
	其他对象，非类对象、类加载器对象或者数组对象
	堆对象的其他附加指标
	向上引用的对象
	向下引用的对象
	对象是一个 GCRoot
	工具条和右键菜单图标
	dominator tree
	query browser
	Retained set 或者 Customized retained set
	运行报表
	比较，和其他 HeapDump 进行比较
	以直方图的形式展示
	计算 retained size
	immediate dominators（直接支配者）
	Heap Dump 直方图
	线程相关的查询
	Paths to GC Root
	导出下拉菜单
	按内存地址搜索对象

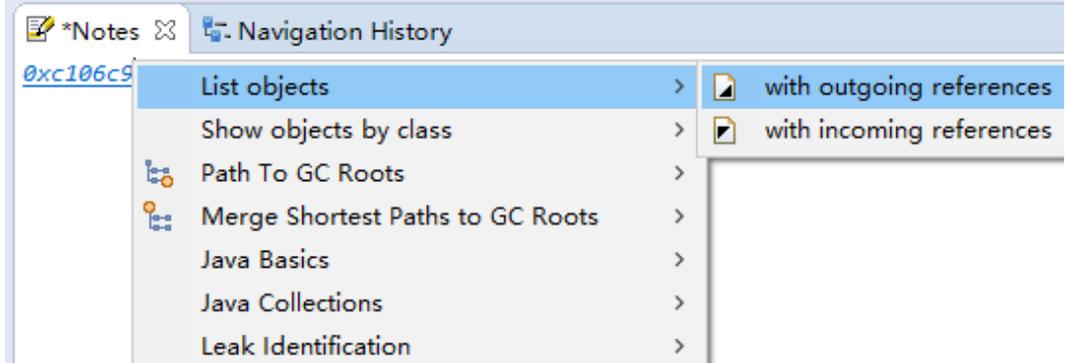
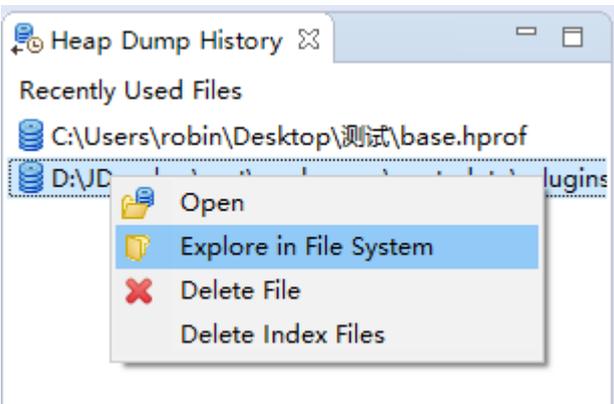
	Filter, 过滤器
	OQL 查询编辑器
	shallow size 或是 heap size
	分组功能下拉按钮
	固定当前 Inspector 视图
	内存溢出跟踪 (并非一直可用)
	所有行的统计。用“+”修饰说明并非所有的子项都已经列了出来, 列表还可以扩展
	打开专家系统报表
	打开 Overview 面板
	打开 Heap Dump
	从一个文件打开 Heap Dump
	Finalizer 概览
	导出为.csv 文件
	导出为.html 文件
	视图
	笔记视图。在这个视图里可以编辑笔记, 所有的笔记都会和 Heap Dump 保存在一起
	历史视图。最近曾经打开过的 Heap Dump 文件都可以在这里看到
	Heap Dump 详情
	对象查看视图。选中的对象的详细信息
	错误日志。将错误上报到 MAT 的 Bugzilla

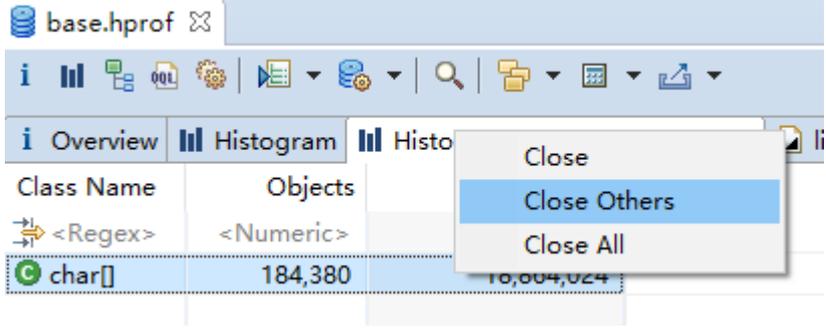
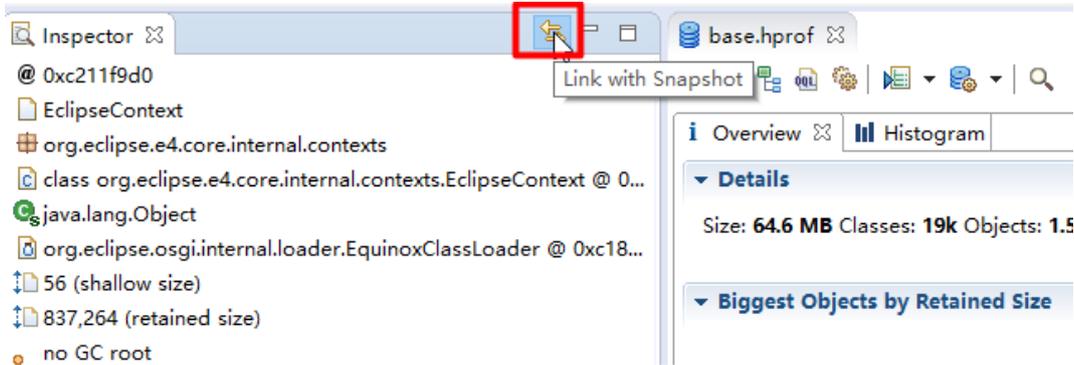
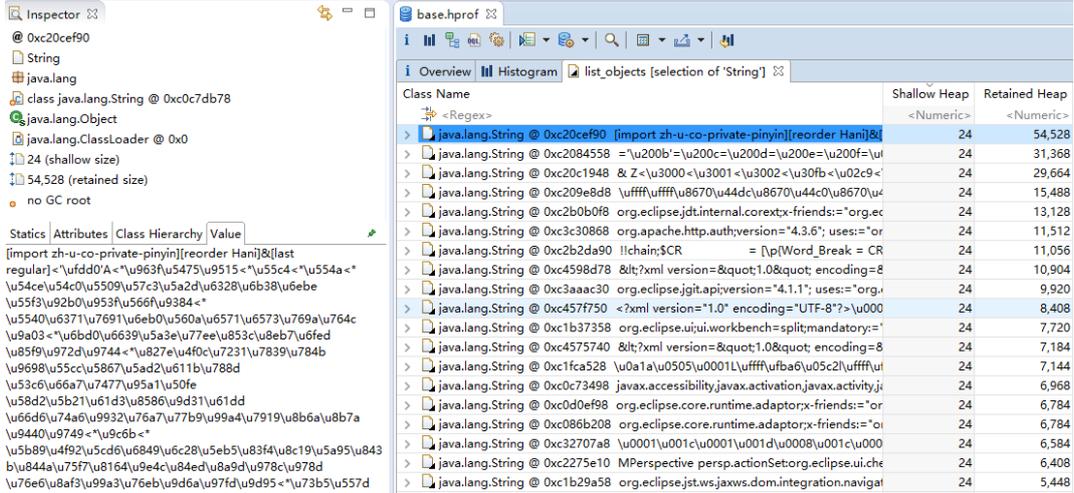
## 6. 技巧和窍门

如下是一些使用 Memory Analyzer 的小技巧:

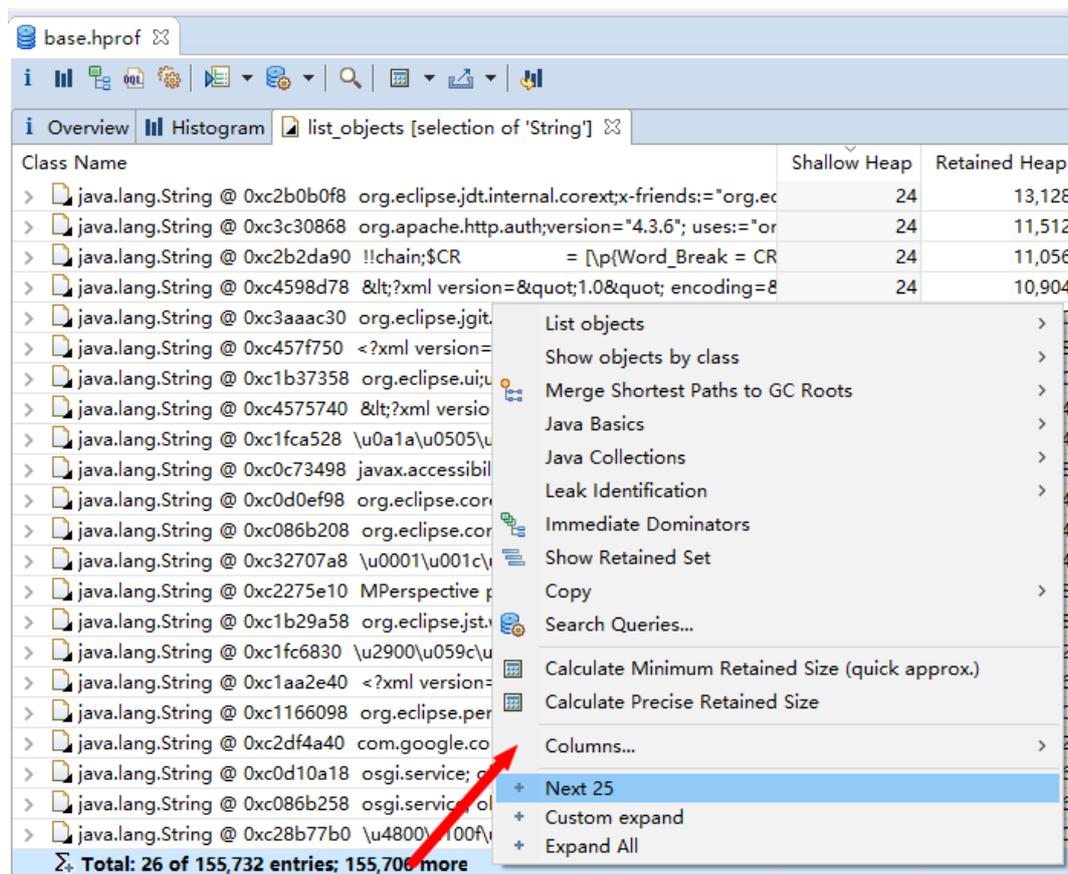
技巧	详解
快捷键	<p>Ctrl+Q: 打开查询浏览器</p> <p>Ctrl+H: 打开查询记录</p> <p>Alt+I: 打开图标说明</p>
快速搜索	在直方图或查询结果中直接输入字符即可进行快速查找。要求输入字符不能小于三个,

	<p>匹配项会直接高亮显示：</p>  <table border="1"> <thead> <tr> <th>Class Name</th> <th>Objects</th> <th>Shallow He...</th> <th>Retained H...</th> </tr> </thead> <tbody> <tr> <td>jav</td> <td>&lt;Numeric&gt;</td> <td>&lt;Numeric&gt;</td> <td>&lt;Numeric&gt;</td> </tr> <tr> <td>char[]</td> <td>184,380</td> <td>18,864,024</td> <td>18,864,024</td> </tr> <tr> <td>java.util.HashMap\$Node</td> <td>208,867</td> <td>6,683,744</td> <td>23,592,128</td> </tr> <tr> <td>java.lang.String</td> <td>155,732</td> <td>3,737,568</td> <td>17,952,776</td> </tr> <tr> <td>java.util.HashMap\$Node[]</td> <td>57,081</td> <td>3,643,368</td> <td>26,004,624</td> </tr> <tr> <td>java.util.HashMap</td> <td>72,077</td> <td>3,459,696</td> <td>28,057,456</td> </tr> <tr> <td>int[]</td> <td>29,626</td> <td>3,426,408</td> <td>3,426,408</td> </tr> <tr> <td>byte[]</td> <td>79,211</td> <td>3,409,152</td> <td>3,409,152</td> </tr> <tr> <td>java.lang.Object[]</td> <td>39,185</td> <td>2,102,320</td> <td>25,280,712</td> </tr> <tr> <td>java.util.Collections\$UnmodifiableMap</td> <td>60,776</td> <td>1,944,832</td> <td>6,729,160</td> </tr> <tr> <td>java.lang.String[]</td> <td>9,458</td> <td>981,968</td> <td>2,220,328</td> </tr> <tr> <td>java.util.WeakHashMap\$Entry</td> <td>21,553</td> <td>862,120</td> <td>2,155,048</td> </tr> <tr> <td>org.eclipse.core.internal.registry.ReferenceMa...</td> <td>16,741</td> <td>803,568</td> <td>834,320</td> </tr> </tbody> </table>	Class Name	Objects	Shallow He...	Retained H...	jav	<Numeric>	<Numeric>	<Numeric>	char[]	184,380	18,864,024	18,864,024	java.util.HashMap\$Node	208,867	6,683,744	23,592,128	java.lang.String	155,732	3,737,568	17,952,776	java.util.HashMap\$Node[]	57,081	3,643,368	26,004,624	java.util.HashMap	72,077	3,459,696	28,057,456	int[]	29,626	3,426,408	3,426,408	byte[]	79,211	3,409,152	3,409,152	java.lang.Object[]	39,185	2,102,320	25,280,712	java.util.Collections\$UnmodifiableMap	60,776	1,944,832	6,729,160	java.lang.String[]	9,458	981,968	2,220,328	java.util.WeakHashMap\$Entry	21,553	862,120	2,155,048	org.eclipse.core.internal.registry.ReferenceMa...	16,741	803,568	834,320
Class Name	Objects	Shallow He...	Retained H...																																																						
jav	<Numeric>	<Numeric>	<Numeric>																																																						
char[]	184,380	18,864,024	18,864,024																																																						
java.util.HashMap\$Node	208,867	6,683,744	23,592,128																																																						
java.lang.String	155,732	3,737,568	17,952,776																																																						
java.util.HashMap\$Node[]	57,081	3,643,368	26,004,624																																																						
java.util.HashMap	72,077	3,459,696	28,057,456																																																						
int[]	29,626	3,426,408	3,426,408																																																						
byte[]	79,211	3,409,152	3,409,152																																																						
java.lang.Object[]	39,185	2,102,320	25,280,712																																																						
java.util.Collections\$UnmodifiableMap	60,776	1,944,832	6,729,160																																																						
java.lang.String[]	9,458	981,968	2,220,328																																																						
java.util.WeakHashMap\$Entry	21,553	862,120	2,155,048																																																						
org.eclipse.core.internal.registry.ReferenceMa...	16,741	803,568	834,320																																																						
过滤	<p>只显示表格或 tree 中符合过滤规则的行。点击结果中的首行，或者选择了第一行后按 Enter 键即可输入过滤规则。可以使用多重过滤规则。在下面的例子中最后一行的“(filtered)”意味着有行被过滤器过滤掉了。</p> <p>对于文字列，一个常规的表达式可以作为过滤规则。</p> <p>对于数字列，可以使用算数比较作为过滤规则：</p> <ul style="list-style-type: none"> <li>➤ 间隔：100...10000 或者 10%...100%；</li> <li>➤ 大于比较：&gt;1000 或者 &gt;=5%；</li> <li>➤ 小于比较：&lt;=10000 或者 &lt;90%。</li> </ul>  <table border="1"> <thead> <tr> <th>Class Name</th> <th>Objects</th> <th>Shallow Heap</th> <th>Retained Heap</th> </tr> </thead> <tbody> <tr> <td>*jav.*</td> <td>&gt;10000</td> <td>&lt;Numeric&gt;</td> <td>4000000..20000000</td> </tr> <tr> <td>java.lang.String</td> <td>155,732</td> <td>3,737,568</td> <td>17,952,776</td> </tr> <tr> <td>java.util.Collections\$UnmodifiableMap</td> <td>60,776</td> <td>1,944,832</td> <td>6,729,160</td> </tr> <tr> <td>java.util.ArrayList</td> <td>30,866</td> <td>740,784</td> <td>8,180,240</td> </tr> <tr> <td>java.lang.Class</td> <td>19,048</td> <td>230,984</td> <td>9,283,592</td> </tr> <tr> <td><b>Σ Total: 4 entries (19,035 filtered)</b></td> <td><b>266,422</b></td> <td><b>6,654,168</b></td> <td></td> </tr> </tbody> </table> <p>(不知为什么我在 Shallow Heap 中尝试任何过滤规则都会把全部内容给过滤掉)</p>	Class Name	Objects	Shallow Heap	Retained Heap	*jav.*	>10000	<Numeric>	4000000..20000000	java.lang.String	155,732	3,737,568	17,952,776	java.util.Collections\$UnmodifiableMap	60,776	1,944,832	6,729,160	java.util.ArrayList	30,866	740,784	8,180,240	java.lang.Class	19,048	230,984	9,283,592	<b>Σ Total: 4 entries (19,035 filtered)</b>	<b>266,422</b>	<b>6,654,168</b>																													
Class Name	Objects	Shallow Heap	Retained Heap																																																						
*jav.*	>10000	<Numeric>	4000000..20000000																																																						
java.lang.String	155,732	3,737,568	17,952,776																																																						
java.util.Collections\$UnmodifiableMap	60,776	1,944,832	6,729,160																																																						
java.util.ArrayList	30,866	740,784	8,180,240																																																						
java.lang.Class	19,048	230,984	9,283,592																																																						
<b>Σ Total: 4 entries (19,035 filtered)</b>	<b>266,422</b>	<b>6,654,168</b>																																																							
Note 视图中的超链接	<p>Notes 视图会自动识别对象地址并添加超链接。按住 Ctrl 键，并左键点击超链接会打开一个菜单：</p>																																																								

	
<p>在 Note 视图和 OQL 面板中 Undo/redo</p>	<p>在 Notes 视图和 OQL 面板中可以使用 Ctrl+Z 或者 Ctrl+Y 实现撤销/重做。</p>
<p>复制到剪贴板</p>	<p>使用 Ctrl+C 可以将内容复制到剪贴板中，复制的内容仍将保持原来的结构：</p> <pre data-bbox="375 828 1292 963"> Class Name            Objects   Shallow Heap   Retained Heap ----- java.util.HashMap\$Entry[]        86      139.200   java.lang.String[]          42      132.928   ----- </pre>
<p>管理曾经打开过的 Heap Dump</p>	<p>在 Heap Dump History 视图中能看到曾经打开过的 heapdump 。可以使用右键菜单来管理这些 dump:</p>  <p>可以选择从文件系统或者只从 History 视图中删除。通过使用 Explore in File System 可以找到 HeapDump 文件在文件系统中的位置</p>
<p>整理 editor 选项卡</p>	<p>右键点击已打开页的页签可以使用右键菜单来管理这些页签。</p>

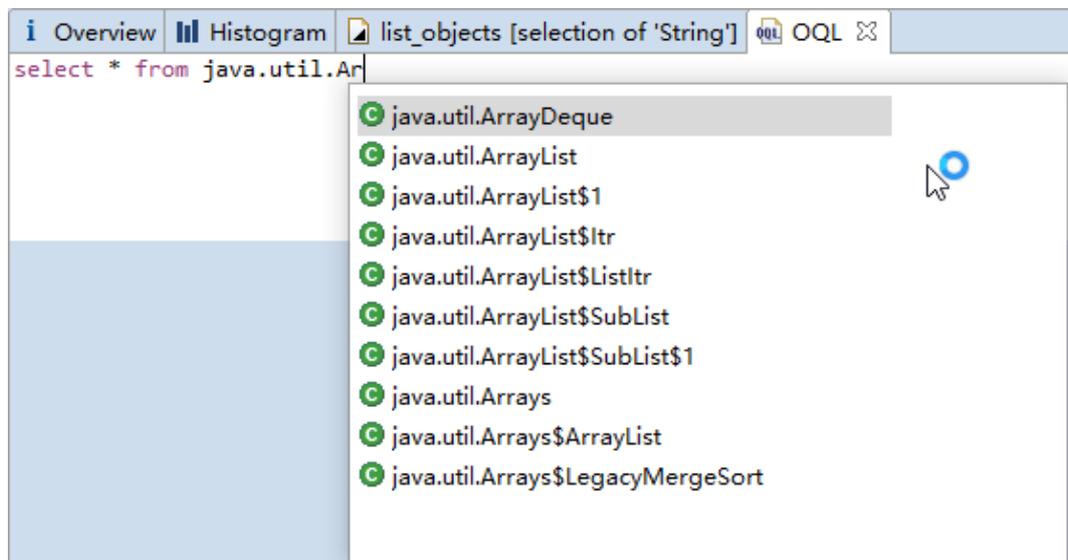
	 <p>在打开的页签很多的情况下这个非常有用。</p>
<p>固定 Inspector 视图</p>	<p>Object Inspector 视图展示了每个对象的细节。但是默认情况下它会随着鼠标光标的经过的对象而不断的变化。可以点击 Inspector 视图右上方的 Link with Snapshot 按钮将之固定下来。选择 Window-&gt;Inspector 可以再打开一个新的 Inspector。此时可以对两个 Inspector 视图中的对象进行比较。</p> 
<p>Inspector 视图中的 value 选项卡</p>	<p>Inspector 视图中展示了对象的详细信息。Inspector 视图中的 value 选项卡则列出了 MAT 名称解析器扩展提供的对象的值的详情：</p> 
<p>Total 行的右键菜单</p>	<p>树或者表的最后一行通常是 Total 行。Total 行代表的是树或者表的其他未展示项。如果选中 Total 行并点击右键菜单则右键菜单的操作对象是树或者表的其他未展示项。如果</p>

选择了全部的行，比如使用了 Ctrl+A，也就是同时选中已展示行和 Total 行，再进行右键菜单的操作，那么操作的对象就是表或者树中的全部对象。

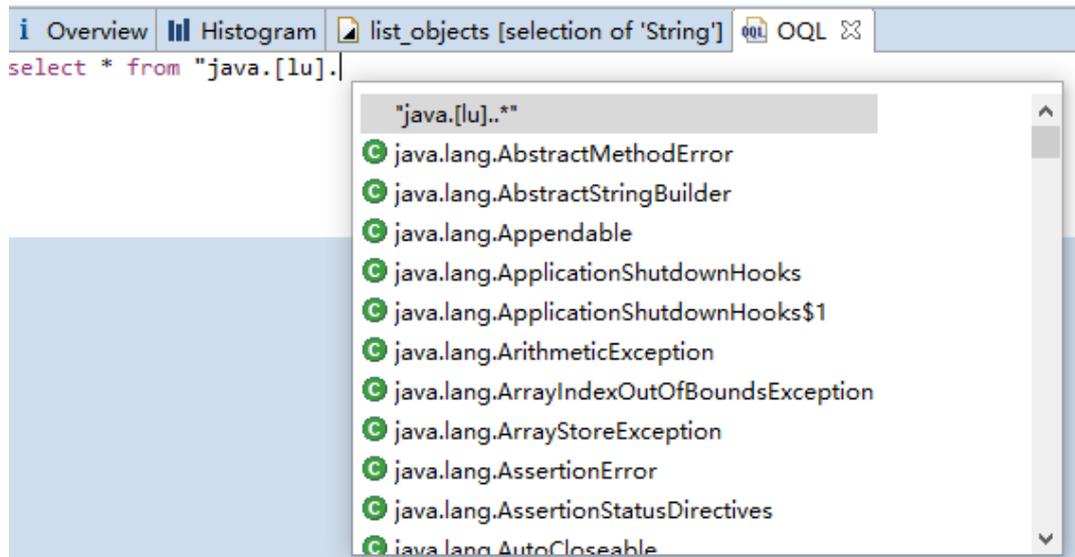


OOQL 自动补完

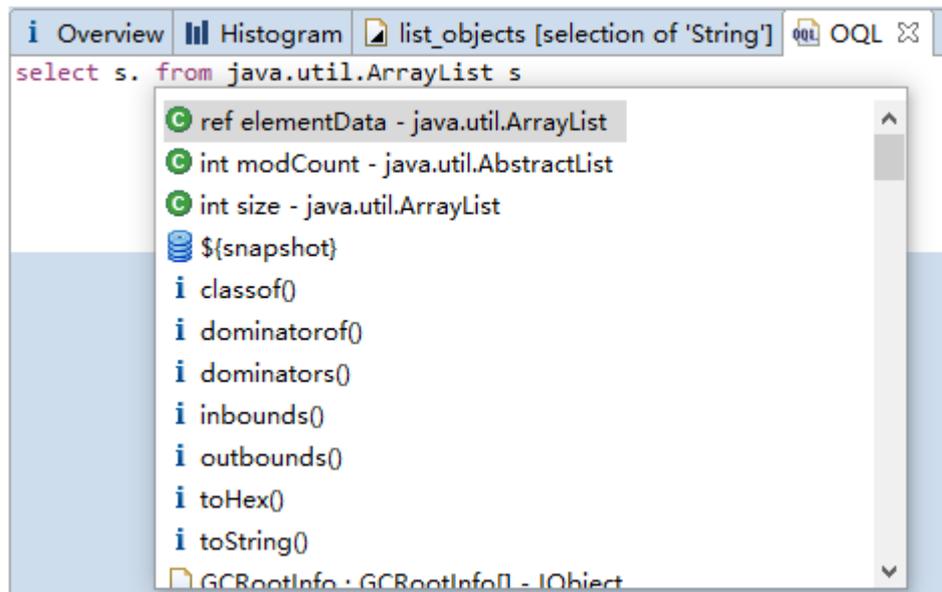
OOQL 面板提供了 OOQL 关键词高亮的功能。也提供 SELECT FROM 字句输入提示的功能，提示的对象主要是简单的类名和一些常规表达式。触发提示的是“.”符号。或者直接 Ctrl+Space 键也可以：



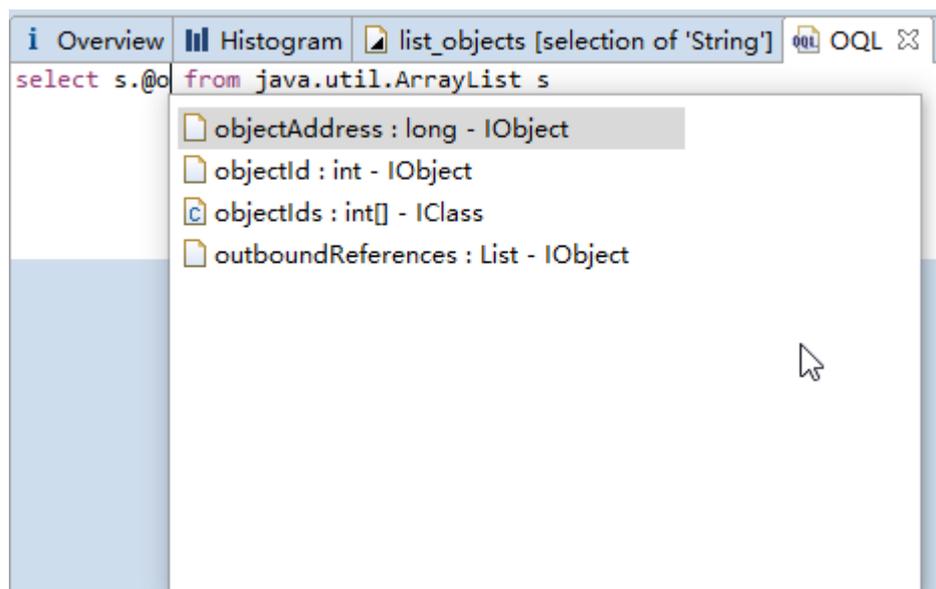
输入双引号（半角字符）可以弹出常规表达式的提示：



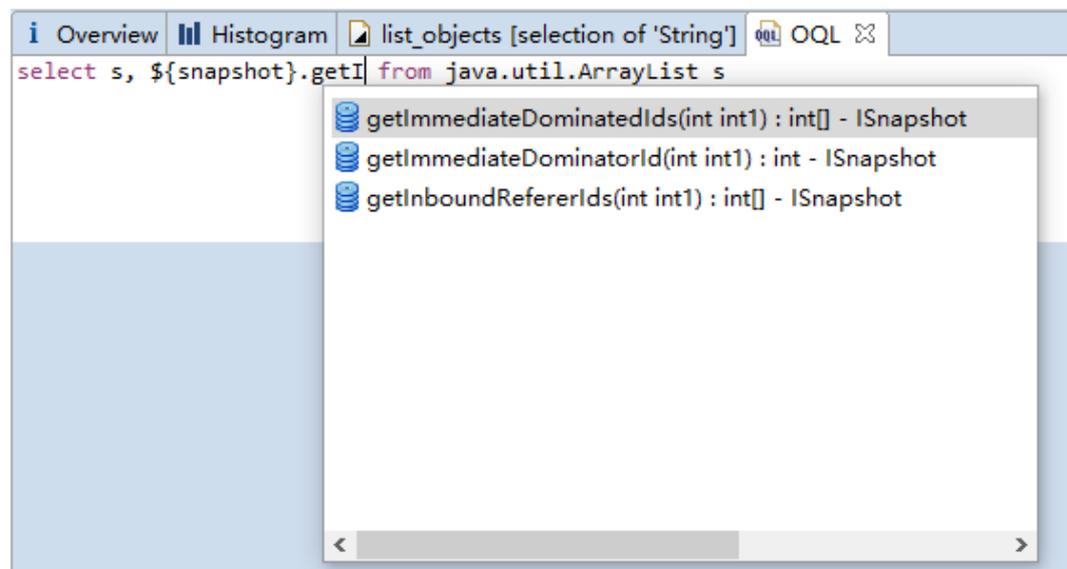
WHERE 和 SELECT 子句中的自动补完功能对类的 Field 也适用。触发动态提示的是“.”符号或“@”符号或者 Ctrl+Space。



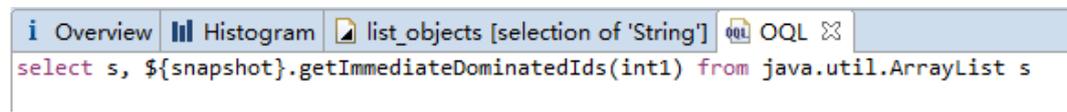
以@符号开头的属性的动态提示：



MAT 也提供了对方法的动态提示:



在动态提示中选中目标项，按 Enter 即可



## 参考文档

- 官方文档：  
<http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.mat.ui.help%2Fwelcome.html&cp=46>
- Eclipse MAT: Understand Incoming and Outgoing References:  
<http://xmlandmore.blogspot.hk/2014/01/eclipse-mat-understand-incoming-and.html>
- MAT 分析工具中 with incoming references 的意思：  
<http://blog.csdn.net/zgf1991/article/details/48353477>
- 深入探讨 Java 类加载器：<http://www.ibm.com/developerworks/cn/java/j-lo-classloader/>
- 使用 Java Native Interface 的最佳实践：<http://www.ibm.com/developerworks/cn/java/j-jni/>
- Java 中 native 关键字：<http://blog.csdn.net/funneies/article/details/8949660>
- Call stack 与 Stack frame 的概念：<http://tobylyx.iteye.com/blog/1728439>

最后，如果您愿意的话，可以通过微信向翻译提供小小打赏：

