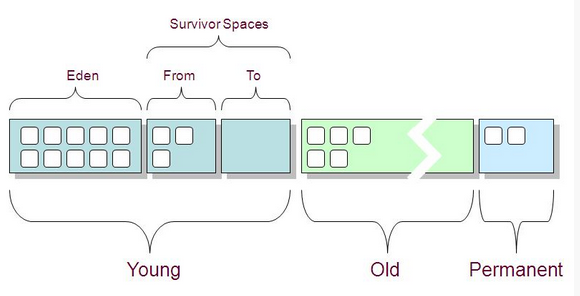
## **第四章 虚拟机性能监控与故障处理工具（待完善...）**

## **Java堆内存模型**



一个JVM实例只存在一个堆内存，它被所有线程共享，堆内存的大小是可以调节的，对于大多数应用，Java堆是JVM管理内存中最大的一块。堆在JVM启动时创建，当类加载器读取了类文件后，需要把类、方法、常量、变量放到堆内存中，以方便执行器执行。几乎所以的对象实例都在这里分配内存。Java堆是垃圾收集器管理的主要区域，因此很多时候也被称作“GC堆”。

Java堆可以细分为：新生代和老年代以及永久代；新生代在细致一点分为：Eden空间、From Survivor空间和To Survivor空间等。如果堆中没有更多的内存完成实例分配，并且堆也无法扩展时，将会抛出OutOfMemoryError异常。

由于Java堆的新生代中的对象98%是“朝生夕死”的，所以并不需要按照1:1的比例划分内存空间，而是将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor。当回收时，将Eden和Survivor中还存活着的对象一次性地复制到另一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8:1，也就是新生代中可用的内存空间为整个新生代容量的90%，只有10%的内存会被“浪费”。

Young：主要是用来存放新生的对象。

Old：主要存放应用程序中生命周期长的内存对象。

Permanent：是指内存的永久保存区域，主要存放Class和Meta的信息,Class在被 Load的时候被放入PermGen space区域. 它和存放Instance的Heap区域不同,GC(Garbage Collection)不会在主程序运行期对PermGen space进行清理，所以如果你的APP会LOAD很多CLASS的话,就很可能出现PermGen space错误。

新生代GC（Minor GC）：指发生在新生代的垃圾收集动作，因为Java对象大多都具备朝生夕灭的特性，所以Minor GC非常频繁，回收速度也比较快。

老年代GC（Full GC/Major GC）：指发生在老年代的GC，出现了Full GC，经常会伴随至少一次的Minor GC（但也不绝对）。Full GC的速度都比较慢。

另外，Minor GC会把Eden中的所有活的对象都移到Survivor区域中，如果Survivor区中放不下，那么剩下的活的对象就被移到Old generation 中。

## Jps

Jps（JavaVirtual Machine Process Status Tool）是JDK1.5提供的一个显示当前所有java进程pid的命令 。显示当前系统的java进程情况及进程id。我们可以通过它来查看我们到底启动了几个java进程（因为每一个java程序都会独占一个java虚拟机实例），并可通过opt来查看这些进程的详细启动参数。

命令格式：jps [options ] [ hostid ]

[ options ] 选项 ：  
 -q：仅输出VM标识符，不包括类名、jar包名和方法入参  
 -m：输出main method的参数   
 -l：输出完全的包名，应用主类名，jar的完全路径名   
 -v：输出jvm参数   
 -V：输出通过flag文件传递到JVM中的参数(.hotspotrc文件或-XX:Flags=所指定的文件 

[ hostid ] 选项：

命令对应的服务器ip，默认不加参数，代码查看本机

### **示例1：jps -q**

**-q 只显示pid，不显示class名称,jar文件名和传递给main方法的参数**

D:\IdeaProjects\SpringMVC\_Source\_3.2.9\_Fourm>jps -q

7860

1928

5884

### **示例2：jps -m**

**-m 输出传递给main方法的参数，在嵌入式jvm上可能是null**

D:\IdeaProjects\SpringMVC\_Source\_3.2.9\_Fourm>jps -m

1928 RemoteMavenServer

5884

7580 Jps -m

### **示例3：jps -l**

**-l 输出应用程序main class的完整package名或者应用程序的jar文件完整路径名**

D:\IdeaProjects\SpringMVC\_Source\_3.2.9\_Fourm>jps -l

5684 sun.tools.jps.Jps

1928 org.jetbrains.idea.maven.server.RemoteMavenServer

5884

### 示例4：jps -v

**-v 输出传递给JVM的参数**

D:\IdeaProjects\SpringMVC\_Source\_3.2.9\_Fourm>jps -v

4884 Jps -Denv.class.path=.C:\Program Files\Java\jdk1.8.0\_65/lib/dt.jar;C:\Program Files\Java\jdk1.8.0\_65/lib/tool.jar;

-Dapplication.home=C:\Program Files\Java\jdk1.8.0\_65 -Xms8m

1928 RemoteMavenServer -Djava.awt.headless=true -Didea.version==2017.1.3 -Xmx768m -Didea.maven.embedder.version=3.5.0 -Dfile.encoding=GBK

5884 -Xms128m -Xmx750m -XX:ReservedCodeCacheSize=240m -XX:+UseConcMarkSweepGC

-XX:SoftRefLRUPolicyMSPerMB=50 -ea -Dsun.io.useCanonCaches=false -Djava.net.preferIPv4Stack=true -XX:+HeapDumpOnOutOfMemoryError

-XX:-OmitStackTraceInFastThrow -Djb.vmOptionsFile=D:\Program Files (x86)\IntelliJ IDEA 2017.1.3\bin\idea64.exe.vmoptions

-Xbootclasspath/a:D:\Program Files (x86)\IntelliJ IDEA 2017.1.3\lib\boot.jar -Didea.jre.check=true -Didea.paths.selector=IntelliJIdea2017.1

-XX:ErrorFile=C:\Users\wb-whz291815\java\_error\_in\_idea\_%p.log -XX:HeapDumpPath=C:\Users\wb-whz291815\java\_error\_in\_idea.hprof

如下是使用jps-v命令，查Tomcat启动的一些参数情况：

6188 Bootstrap

-Djava.util.logging.config.file=C:\Users\wb-whz291815\.IntelliJIdea2017.1\system\tomcat\Unnamed\_(1)\_SpringMVC\_Source\_3\_2\_9\_Fourm\conf\logging.properties

-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager

-agentlib:jdwp=transport=dt\_socket,address=127.0.0.1:57342,suspend=y,server=n -javaagent:C:\Users\wb-whz291815\.IntelliJIdea2017.1\system\groovyHotSwap\gragent.jar

-Dcom.sun.management.jmxremote=

-Dcom.sun.management.jmxremote.port=1099

-Dcom.sun.management.jmxremote.ssl=false

-Dcom.sun.management.jmxremote.authenticate=false

-Djava.rmi.server.hostname=127.0.0.1

-Djdk.tls.ephemeralDHKeySize=2048

-Djava.endorsed.dirs=D:\Program Files (x86)\apache-tomcat-7.0.78\endorsed

-Dcatalina.base=C:\Users\wb-whz291815\.IntelliJIdea2017.1\system\tomcat\Unnamed\_(1)\_SpringMVC\_Source\_3\_2\_9\_Fourm

-Dcatalina.home=D:\Program Files (x86)\apache-tomcat-7.0.78

-Djava.io.tmpdir=D:\Program Files (x86)\apache-tomcat-7.0.78\temp

## Jstat

一个极强的监视VM内存工具。可以用来监视VM内存内的各种堆和非堆的大小及其内存使用量。jstat工具特别强大，有众多的可选项，详细查看堆内各个部分的使用量，以及加载类的数量。使用时，需加上查看进程的进程id，和所选参数。

以下详细介绍各个参数的意义

jstat -class pid:显示加载class的数量，及所占空间等信息。   
    jstat -compiler pid:显示VM实时编译的数量等信息。   
    jstat -gc pid:可以显示gc的信息，查看gc的次数，及时间。其中最后五项，分别是young gc的次数，young gc的时间，full gc的次数，full gc的时间，gc的总时间。   
    jstat -gccapacity:可以显示，VM内存中三代（young,old,perm）对象的使用和占用大小，如：PGCMN显示的是最小perm的内存使用量，PGCMX显示的是perm的内存最大使用量，PGC是当前新生成的perm内存占用量，PC是但前perm内存占用量。其他的可以根据这个类推， OC是old内纯的占用量。   
    jstat -gcnew pid:new对象的信息。   
    jstat -gcnewcapacity pid:new对象的信息及其占用量。   
    jstat -gcold pid:old对象的信息。   
    jstat -gcoldcapacity pid:old对象的信息及其占用量。   
    jstat -gcpermcapacity pid: perm对象的信息及其占用量。   
    jstat -util pid:统计gc信息统计。   
    jstat -printcompilation pid:当前VM执行的信息。  
    除了以上一个参数外，还可以同时加上两个数字，如：jstat -printcompilation 3024 250 6是每250毫秒打印一次，一共打印6次，还可以加上-h3每三行显示一下标题。

### 示例1：jstat -class pid

**查看Tomcat进程加载的Class数量和内存占用情况：**

D:\IdeaProjects\SpringMVC\_Source\_3.2.9\_Fourm>jps -l

1928 org.jetbrains.idea.maven.server.RemoteMavenServer

5884

6188 org.apache.catalina.startup.Bootstrap

6476 org.jetbrains.jps.cmdline.Launcher

7324 sun.tools.jps.Jps

D:\IdeaProjects\SpringMVC\_Source\_3.2.9\_Fourm>jstat -class 6188

Loaded Bytes Unloaded Bytes Time

9384 18962.8 0 0.0 8.53

## Jmap

打印出某个java进程内存里的所有‘对象’的情况（如：产生那些对象，及其数量）。jmap 是一个可以输出所有内存中对象的工具，甚至可以将VM 中的heap，以二进制输出成文本。

使用方法 jmap -histo pid。如果连用 SHELL jmap -histo pid>a.log可以将其保存到文本中去（windows下也可以使用），在一段时间后，使用文本对比工具，可以对比出GC回收了哪些对象。

jmap -dump:format=b,file=f1 3024可以将3024进程的内存heap输出出来到f1文件里。 

### 示例1：jmap -heap pid

**查看Tomcat进程这个java进程的JVM内存状态，每个java进程都有一个JVM实例，通过jps命令我们可以知道，这个Tomcat进程的pid为6188**

D:\IdeaProjects\SpringMVC\_Source\_3.2.9\_Fourm>jps -l

1928 org.jetbrains.idea.maven.server.RemoteMavenServer

5884

6188 org.apache.catalina.startup.Bootstrap

6476 org.jetbrains.jps.cmdline.Launcher

7324 sun.tools.jps.Jps

D:\IdeaProjects\SpringMVC\_Source\_3.2.9\_Fourm>jmap -heap 6188

Attaching to process ID 6188, please wait...

Debugger attached successfully.

Server compiler detected.

JVM version is 25.65-b01

using thread-local object allocation.

Parallel GC with 4 thread(s)

Heap Configuration:

MinHeapFreeRatio = 0

MaxHeapFreeRatio = 100

MaxHeapSize = 2136997888 (2038.0MB)

NewSize = 44564480 (42.5MB)

MaxNewSize = 711983104 (679.0MB)

OldSize = 89653248 (85.5MB)

NewRatio = 2

SurvivorRatio = 8

MetaspaceSize = 21807104 (20.796875MB)

CompressedClassSpaceSize = 1073741824 (1024.0MB)

MaxMetaspaceSize = 17592186044415 MB

G1HeapRegionSize = 0 (0.0MB)

Heap Usage:

PS Young Generation

Eden Space:

capacity = 535298048 (510.5MB)

used = 199002456 (189.78353118896484MB)

free = 336295592 (320.71646881103516MB)

37.176010027221324% used

From Space:

capacity = 28835840 (27.5MB)

used = 0 (0.0MB)

free = 28835840 (27.5MB)

0.0% used

To Space:

capacity = 48758784 (46.5MB)

used = 0 (0.0MB)

free = 48758784 (46.5MB)

0.0% used

PS Old Generation

capacity = 131072000 (125.0MB)

used = 59090848 (56.353424072265625MB)

free = 71981152 (68.64657592773438MB)

45.0827392578125% used

25574 interned Strings occupying 2897072 bytes.

### 示例2：jmap -histo pid

**查看JVM堆中对象详细占用情况，如下是部分输出信息：**

D:\IdeaProjects\SpringMVC\_Source\_3.2.9\_Fourm>jps -l

1928 org.jetbrains.idea.maven.server.RemoteMavenServer

5884

6188 org.apache.catalina.startup.Bootstrap

6476 org.jetbrains.jps.cmdline.Launcher

7324 sun.tools.jps.Jps

D:\IdeaProjects\SpringMVC\_Source\_3.2.9\_Fourm>jmap -histo 6188

...

4071: 1 16 org.springframework.http.MediaType$2

4072: 1 16 org.springframework.http.converter.xml.Jaxb2RootElementHttpMessageConverter$1

4073: 1 16 org.springframework.http.converter.xml.SourceHttpMessageConverter$1

4074: 1 16 org.springframework.http.converter.xml.SourceHttpMessageConverter$2

4075: 1 16 org.springframework.jdbc.datasource.SimpleConnectionHandle

4076: 1 16 org.springframework.orm.hibernate3.SpringSessionContext

4077: 1 16 org.springframework.orm.hibernate3.SpringTransactionFactory

4078: 1 16 org.springframework.oxm.support.AbstractMarshaller$1

4079: 1 16 org.springframework.transaction.interceptor.NameMatchTransactionAttributeSource

4080: 1 16 org.springframework.ui.context.support.DelegatingThemeSource

4081: 1 16 org.springframework.ui.velocity.CommonsLoggingLogSystem

4082: 1 16 org.springframework.util.ReflectionUtils$4

4083: 1 16 org.springframework.util.ReflectionUtils$5

4084: 1 16 org.springframework.util.ReflectionUtils$6

4085: 1 16 org.springframework.validation.beanvalidation.LocaleContextMessageInterpolator

4086: 1 16 org.springframework.validation.beanvalidation.SpringConstraintValidatorFactory

...

### **示例3：jmap -dump:format=b,file=文件名 [pid]**

**导出整个JVM 中内存信息到文件中**

# JDK的可视化工具

JDK中处理提供大量的命令行工具外，还有两个功能强大的可视化工具：JConsole和VisualVm，这两个工具的JDK的正式成员，没有被贴上“unsupported and experimental(不支持和实验)”的标签。

其中JConsole是在JDK1.5时期就已经提供的虚拟机监控工具，而VisualVM在JDK 1.6 Update7中才首次发布，现在已经成为Sun（Oracle）主力推动的多合一故障处理工具，并且已经从JDK中分离出来成为可以独立发展的开源项目。

## jconsole

一个java GUI监视工具，可以以图表化的形式显示各种数据。并可通过远程连接监视远程的服务器VM。

JConsole（Java Monitoring and Management Console）是一款基于JMX的可视化监视和管理的工具。它管理部分的功能是针对JMX MBean进行管理，MBean可以使用代码、中间件服务器的管理控制台或者所有符合JMX规范的软件进行访问。

### **JMX简介**

JMX是一种JAVA的正式规范，它主要目的是让程序有被管理的功能，那么怎么理解所谓的“被管理”呢？试想你开发了一个软件（如ＷＥＢ网站），它是在２４小时不间断运行的，那么你可能会想要“监控”这个软件的运行情况，比如收到了多少数据，有多少人登录等等。或者你又想“配置”这个软件，比如现在访问人数比较多，你想把数据连接池设置得大一些。

　　当然，你也许会专门为这些管理来开发软件，但如果你借助JMX，则会发现创建这样的管理程序是如此简单。因为你无需为管理程序来开发界面，已经有通用的JMX管理软件，如MC4J，或者是用一般都附带提供的HTML网页来管理，你要做的仅仅是将自己要被管理和监控类的按照JMX规范修改一下即可。

　　中间件软件WebLogic的管理页面就是基于JMX开发的，而JBoss则整个系统都基于JMX构架。

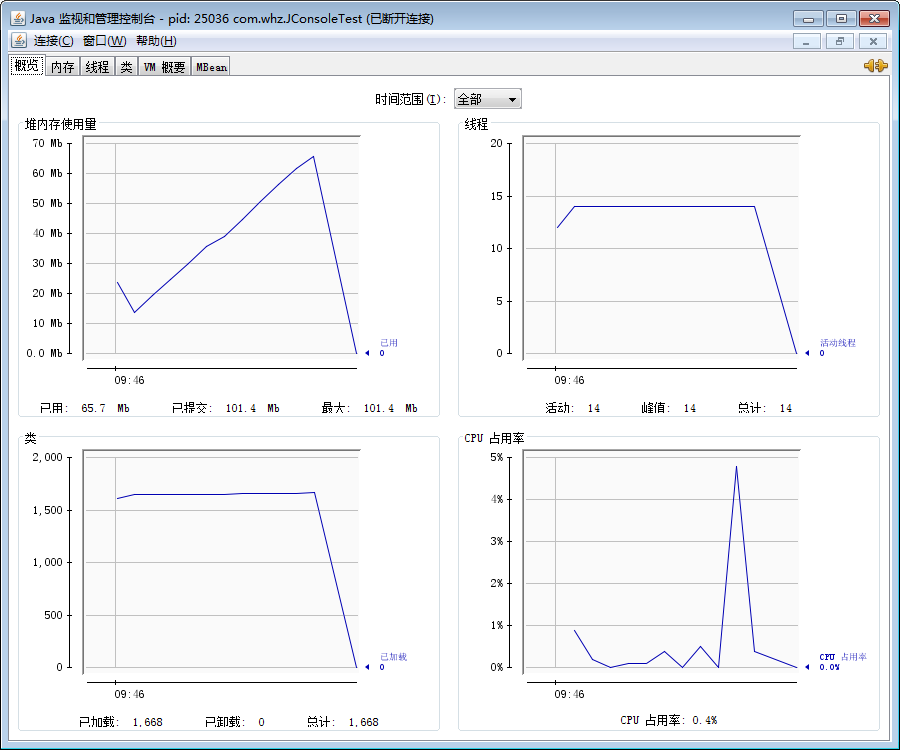
启用JConsole

通过JDK/bin目录下的“jconsole.exe”启动JConsole后，将自动搜索出本地运行的所有虚拟机进程，不需要用户自己在使用jps来查询了，如下图，双击选择其中一个进程即可开始监控，也可以使用下面的“远程进程”功能来连接远程服务器，对远程虚拟机进行监控。



我们看到，例如arg.apache.catalina.startup.Bootstrp 是一个启动Tomcat后的java进程；另外一个JConsoleTest进程是本人测试的一个虚拟机进程，等下我们会进行介绍。我们除了可以使用这种方式进入监控的可视化界面外，我们还可以通过命令进入，例如JConsoleTest的pid为20496，那我们可以使用jConsole 20496来进入监控界面。

进入JConsole主界面后，可以看到主界面共包括“概述”、“内存”、“线程”、“类”、“VM摘要”和“MBean”六个页签，如图：



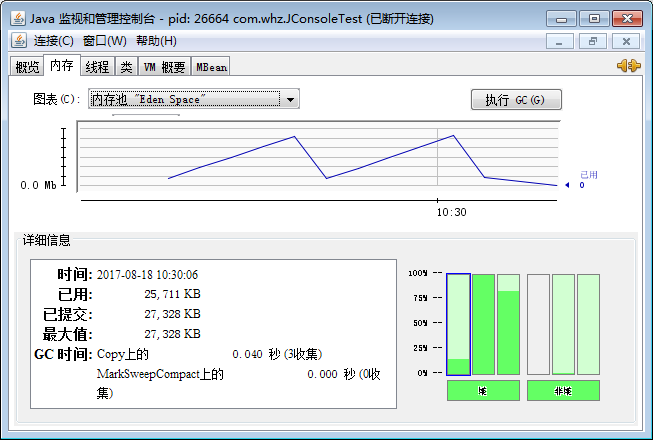
“概述”页签显示的是整个虚拟机主要运行数据的概览，其中包括“堆内存使用情况”、“线程”、“类”和“CPU使用情况”四项信息的曲线图，这些曲线图是后面“内存”、“线程”、“类”页签的信息汇总。

### **内存监控**

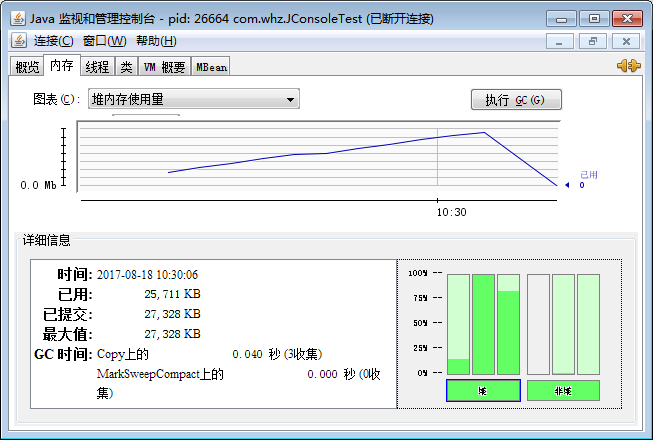
“内存”页签相当于可视化的jstat命令，用于监视收集器管理的虚拟机内存（Java堆和永久代）的变化趋势。我们通过以下JConsoleTest程序来体验一下它的监视功能。运行时设置虚拟机的参数为：-Xms100m -Xmx100m -XX:+UseSerialGC，这段代码的作用是以64KB/50毫秒的速度忘Java堆中填充数据，一共填充1000次，使用JConsole的“内存”页签进行监视，观察曲线和柱状指示图的变化。

*// 内存占位符对象，一个OOMObject大约占64K***public class** JConsoleTest {  
  
 **static class** OOMObject {  
 **public byte**[] **placeholder** = **new byte**[64 \* 1024];  
 }  
  
 **public static void** fillHeap(**int** num) **throws** InterruptedException {  
 List<OOMObject> list = **new** ArrayList<OOMObject>();  
 **for**(**int** i = 0;i < num; i++) {  
 *// 稍作延时，令监视曲线的变化更加明显* Thread.*sleep*(50);  
 list.add(**new** OOMObject());  
 }  
 System.*gc*();  
 }  
  
 *//-Xms100m -Xmx100m -XX:+UseSerialGC* **public static void** main(String[] args) **throws** Exception {  
 *fillHeap*(1000);  
*// System.gc();  
// System.out.println("执行GC...");  
// Thread.sleep(3000);* }  
  
}

程序运行后，在“内存”页签中可以看到内存池Eden区的运行趋势呈现折线状，如图：



而监视范围扩大值整个堆后，会发现曲线是一条向上增长的平滑曲线，如图：

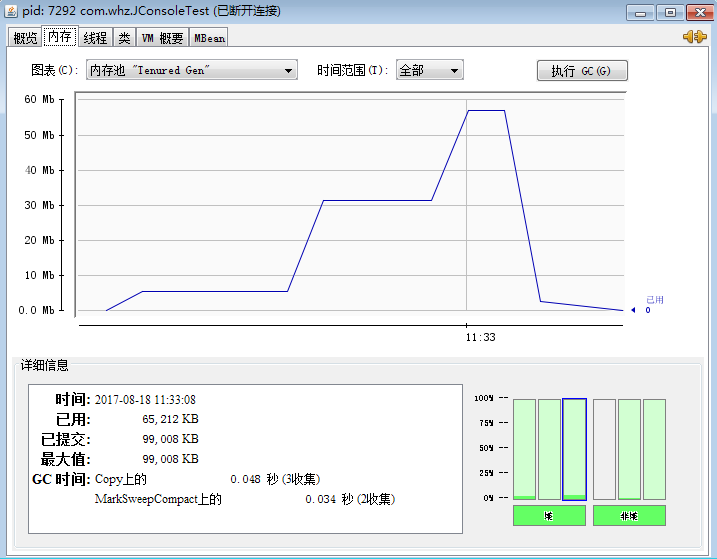


**这里我们分析一个问题：**在1000次循环执行结束，运行了System.gc()后，虽然整个新生代Eden和Survivor区都基本被清空了，但是代表老年代的柱状图和曲线图仍然保持峰值状态，说明被填充进堆中的数据在System.gc()方法执行之后仍然存活着。那么为何执行了System.gc()之后，图中代表老年代的柱状图仍然显示为峰值状态，代码需要如何调整才能System.gc()回收掉填充到堆中的对象呢？

**分析：**其实在System.gc()之后，空间未能回收是因为List<OOMObject> list对象仍然存活着，fillHeap()方法仍然没有退出，因此list对象在执行System.gc()时仍然处于作用域之内。如果把System.gc()移到fillHeap()方法，调用就可以回收掉全部内存。修改的代码如下：

**public static void** main(String[] args) **throws** Exception {  
 *fillHeap*(1000);  
 System.*gc*();  
 System.***out***.println(**"执行GC..."**);  
 Thread.*sleep*(3000);  
}

**效果图如下：**



### **线程监控**

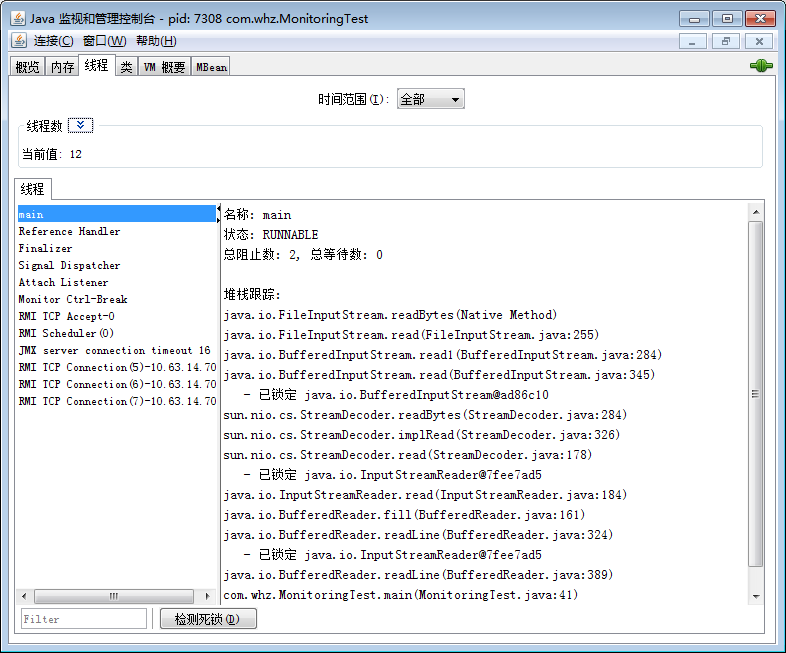
如果上面的“内存”页签相当于可视化的jstat命令的话，“线程”页签的功能则相当于可视化的jstack命令，遇到线程停顿的时候可以使用这个页签进行监控分析。前面讲解jstack命令的时候提到过线程长时间停顿的主要原因有：等待外部资源（如：数据库连接、网络资源、设备资源等）、死循环、锁等待（活锁和死锁）。

**示例1**

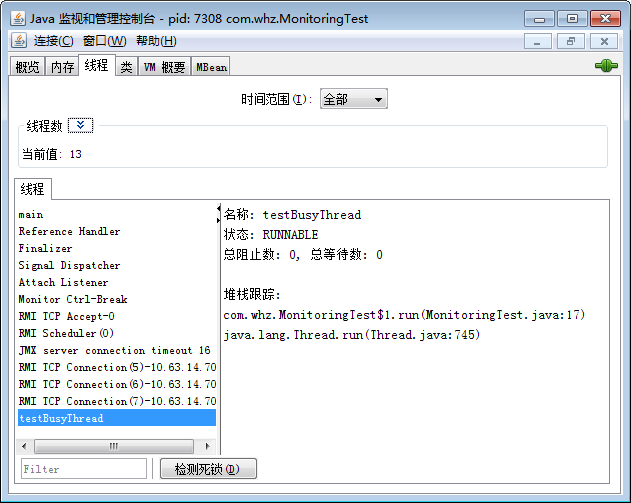
我们通过如下代码来演示上述几种情况：

**public class** MonitoringTest {  
  
 *// 线程死循环演示* **public static void** createBusyThread() {  
 Thread thread = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **while**(**true**) {}  
 }  
 },**"testBusyThread"**);  
 thread.start();  
 }  
  
 *// 线程锁等待演示* **public static void** createLockThread(**final** Object lock) {  
 Thread thread = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **synchronized** (lock) {  
 **try**{  
 lock.wait();  
 } **catch** (InterruptedException e) {e.printStackTrace();}  
 }  
 }  
 },**"testLockThread"**);  
 thread.start();  
 }  
  
 **public static void** main(String[] args) **throws** IOException {  
 BufferedReader br = **new** BufferedReader(**new** InputStreamReader(System.***in***));  
 *// 主线程暂停，直到用户输入，才开启 testBusyThread 线程* br.readLine();  
 *createBusyThread*();  
 *// 主线程暂停，直到用户输入，才开启 testLockThread 线程* br.readLine();  
 Object obj = **new** Object();  
 *createLockThread*(obj);  
 *// 主线程终止* }  
}

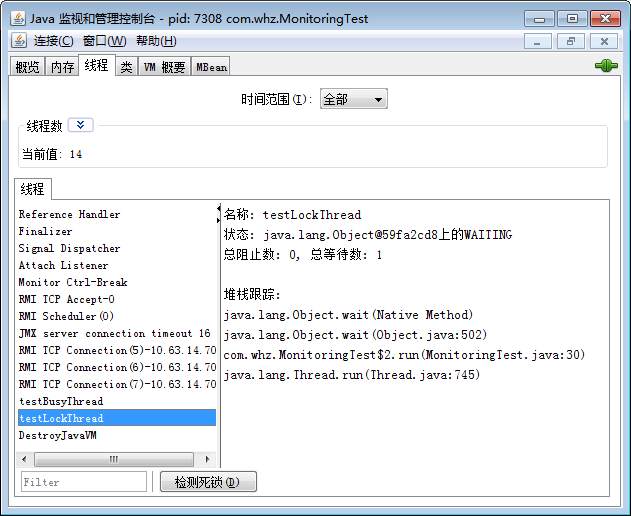
程序开始后，我们看到主线程阻塞在第一个br.readLine();等用户输入，堆栈追踪显示BufferedReader在readBytes方法中等待System.in的键盘输入，这时候线程为Runnable状态，Runnable状态的进行会被分配运行时间，但是readBytes方法检查到流没有更新时，就会归还执行令牌（这是程序会阻塞），这种等待只消耗很小的CPU资源。如图：



用户输入信息后，我们看到主线程继续执行，创建了testBusyThread线程，然后主线程又再一次阻塞在了第二个br.readLine() ，而testBusyThread线程一直在执行循环，从堆栈追踪可以看到testBusyThread一直停留在com.whz.MonitoringTest$1.run(MonitoringTest.java:17)，这时候线程为Runnable状态，而且没有归还线程执行令牌的动作，而在空循环上用尽全部执行时间知道线程切换，这种等待会消耗较多的CPU资源。



继续输入信息回车，我们看到主线程继续执行，创建了testLockThread线程，然后主线结束生命周期。而testLockThread线程在等待着lock对象的notify或notifyAll方法的出现，线程这时候处于WAITING状态，在被唤醒前不会被分配执行时间。testLockThread线程这时候处于正常的活锁等待状态，只要lock对象的notify()或notifyAll()方法被调用，这个线程便能激活以继续执行。



**示例2**

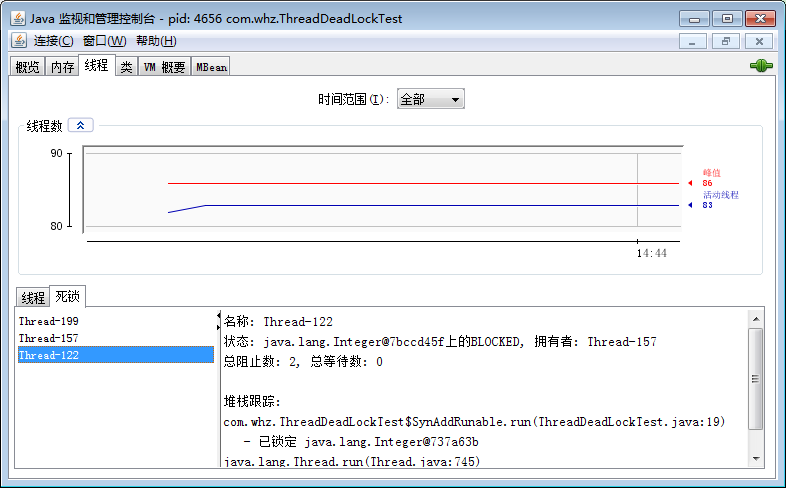
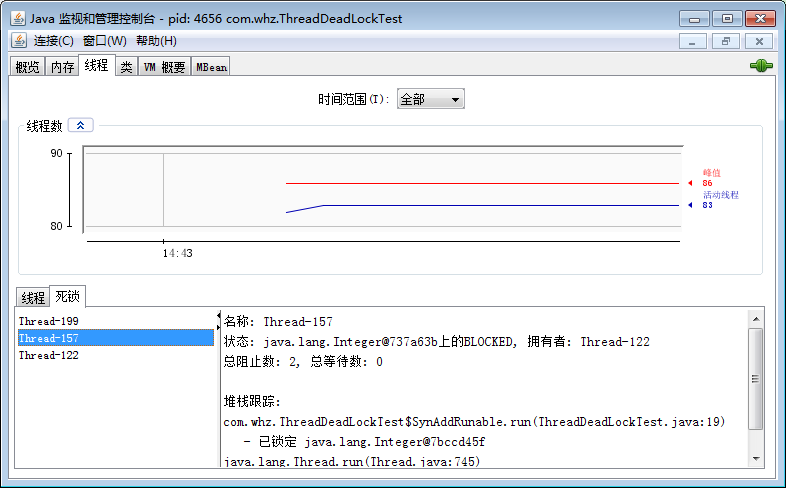
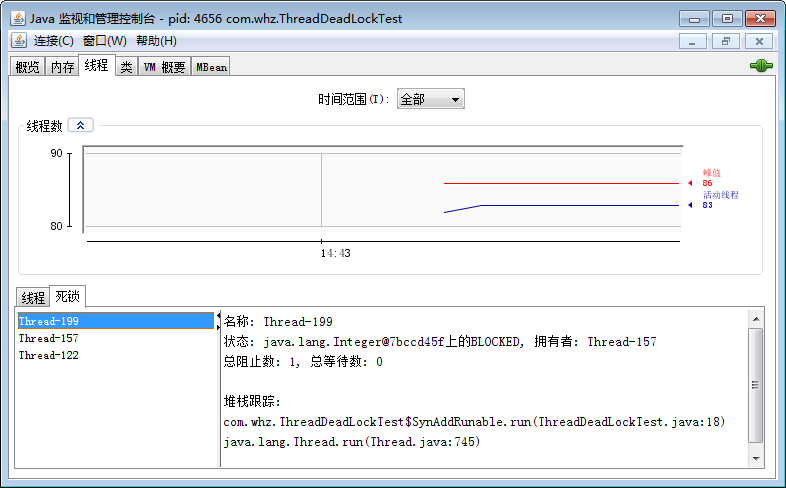
我们使用如下代码来测试监控到的死锁：

**public class** ThreadDeadLockTest {  
  
 **static class** SynAddRunable **implements** Runnable {  
 **int a**,**b**;  
 **public** SynAddRunable(**int** a,**int** b) {  
 **this**.**a** = a;  
 **this**.**b** = b;  
 }  
 @Override  
 **public void** run() {  
 **synchronized** (Integer.*valueOf*(**a**)) {  
 **synchronized** (Integer.*valueOf*(**b**)) {  
 System.***out***.println(**a**+**b**);  
 }  
 }  
 }  
 }  
  
 **public static void** main(String[] args) {  
 **for**(**int** i=0;i<100;i++) {  
 **new** Thread(**new** SynAddRunable(1,2)).start();  
 **new** Thread(**new** SynAddRunable(2,1)).start();  
 }  
 }  
}

**分析：**这段代码开了200个线程分别取计算1+2和2+1的值，其实for循环是可以省略的，两个线程也可能会导致死锁，不过那样概率太小，需要尝试运行很多次才能看到效果。而这段代码也出现死锁的概率会比较大，出现死锁后程序无法结束。

造成死锁的原因是Integer.valueOf()方法基于减少对象创建次数和节省内存的考虑，[-128,127]之间的数字会被缓存，当valueOf()方法在这个范围之内出入参数，直接返回缓存中的对象。也就是说代码调用了200次Integer.valueOf()方法一就只返回两个不同的对象。假如在某个线程的两个synchronized块之间发生了一次线程切换，就会出现线程A等着被线程B持有的Integer.valueOf(1)，线程B又等着被线程A持有的Integer.valueOf(2),结果大家都跑步下去的情景。

出现线程死锁之后，点击JConsole线程面板的“检测到死锁”的按钮，将出现一个新的“死锁”页签，如图：



图中显示线程Thread-199在等待一个被线程Thread-157持有的Integer对象，线程Thread-157等待被线程Thread-122持有的Integer对象，而线程Thread-122同时又在等线程Thread-199释放Integer对象，这样就操作了死锁。