# 深入理解Java虚拟机

## 内容分类与环境搭建

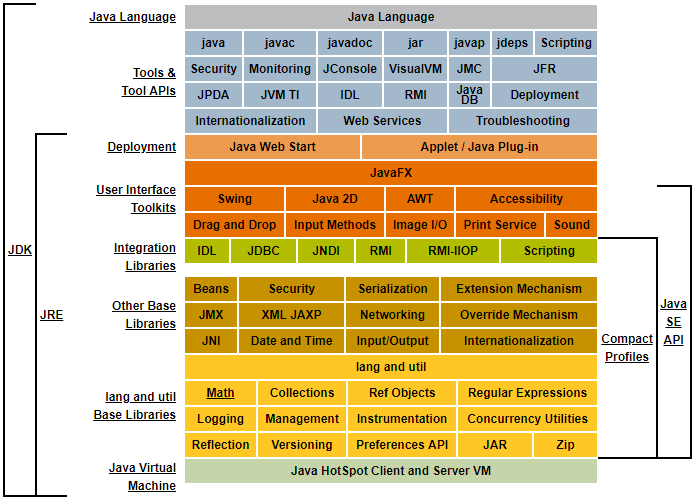
### 1.1概述

1. 了解历史
2. 内存结构
3. 垃圾回收机制
4. 性能监控工具
5. 性能调优方案实战
6. 认识类的文件结构
7. 类加载机制
8. 字节码执行引擎
9. 虚拟机编译及运行时优化
10. Java线程高级

### 1.2环境搭建以及jdk,jre,jvm的关系

1. 下载安装JDK
2. 环境变量配置
3. jdk,jre,jvm

<https://docs.oracle.com/javase/8/docs/>



### 1.3JVM初体验：内存溢出场景模拟

|  |
| --- |
| **public** **class** Demo {  } |

|  |
| --- |
| **public** **class** OutOfMemoryTest {    **public** **static** **void** main(String[] args) {  List<Demo> demoList = **new** ArrayList<>();  **while**(**true**) {  demoList.add(**new** Demo());  }  }  } |

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

at java.util.Arrays.copyOf(Unknown Source)

at java.util.Arrays.copyOf(Unknown Source)

at java.util.ArrayList.grow(Unknown Source)

at java.util.ArrayList.ensureExplicitCapacity(Unknown Source)

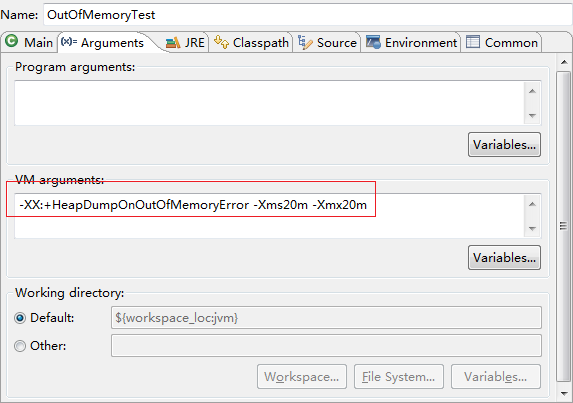
at java.util.ArrayList.ensureCapacityInternal(Unknown Source)

at java.util.ArrayList.add(Unknown Source)

at com.byf.jvm01OutOfMemory.OutOfMemoryTest.main(OutOfMemoryTest.java:11)

1）堆内存快照，JVM参数设置

-XX:+HeapDumpOnOutOfMemoryError -Xms20m -Xmx20m

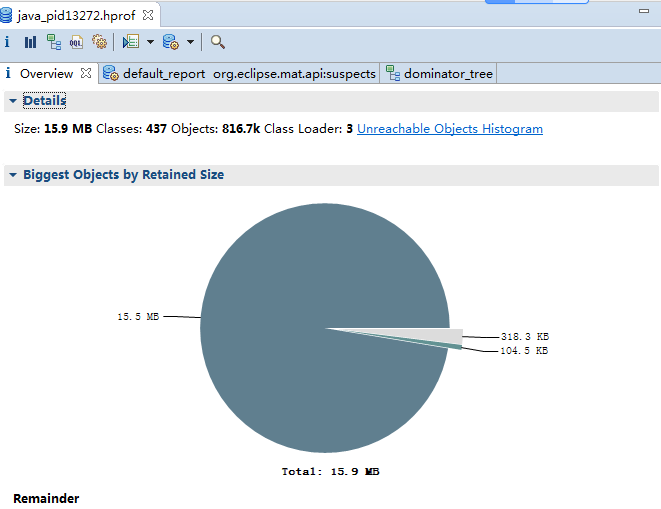


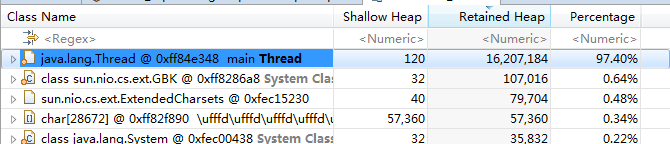
2）生成堆内存快照位置，默认在项目的根路径下：

C:\Users\BYF\workspace\jvm\java\_pid13272.hprof

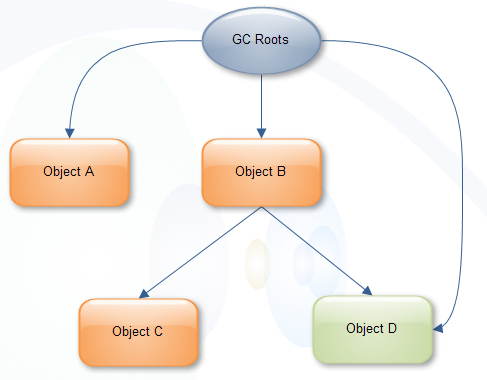
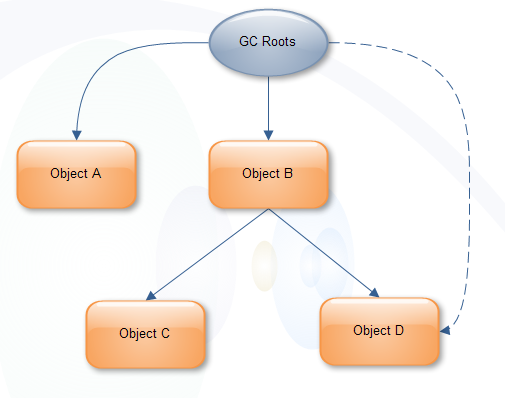
3）下载内存分析工具：

<http://www.eclipse.org/downloads/download.php?file=/mat/1.8.1/rcp/MemoryAnalyzer-1.8.1.20180910-win32.win32.x86_64.zip>

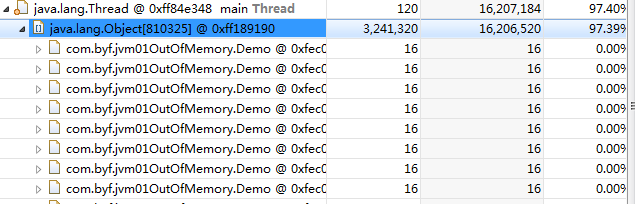




****Shallow Size****   
对象自身占用的内存大小，不包括它引用的对象。   
针对非数组类型的对象，它的大小就是对象与它所有的成员变量大小的总和。当然这里面还会包括一些java语言特性的数据存储单元。   
针对数组类型的对象，它的大小是数组元素对象的大小总和。   
  
****Retained Size****   
Retained Size=当前对象大小+当前对象可直接或间接引用到的对象的大小总和。(间接引用的含义：A->B->C, C就是间接引用)   
换句话说，Retained Size就是当前对象被GC后，从Heap上总共能释放掉的内存。   
不过，释放的时候还要排除被GC Roots直接或间接引用的对象。他们暂时不会被被当做Garbage。

****看图理解Retained Size****   
  
   
  
上图中，GC Roots直接引用了A和B两个对象。   
  
A对象的Retained Size=A对象的Shallow Size   
B对象的Retained Size=B对象的Shallow Size + C对象的Shallow Size   
  
****这里不包括D对象，因为D对象被GC Roots直接引用。****   
如果GC Roots不引用D对象呢？   
  
   
  
  
此时, B对象的Retained Size=B对象的Shallow Size + C对象的Shallow Size + D对象的Shallow Size

5）堆内存溢出的原因，有太多的Demo对象创建。



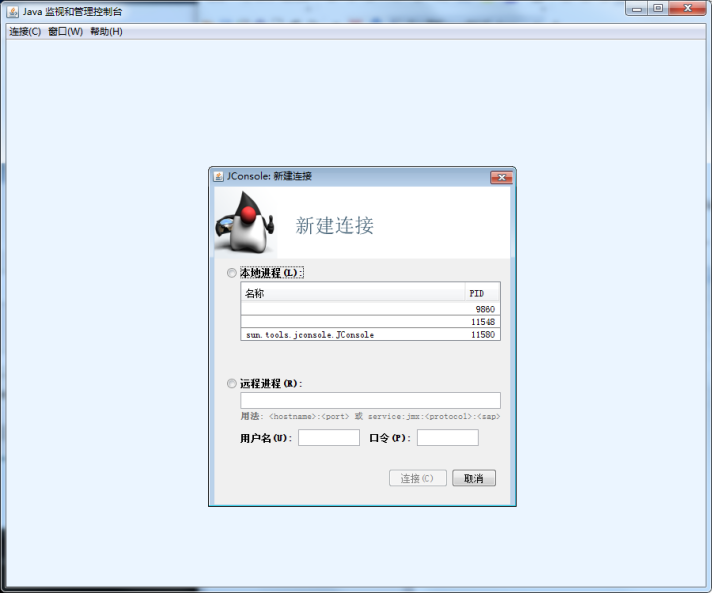
### 1.4JVM再体验：JVM监控工具

性能监控

故障监控

图形化

1. Jconsole



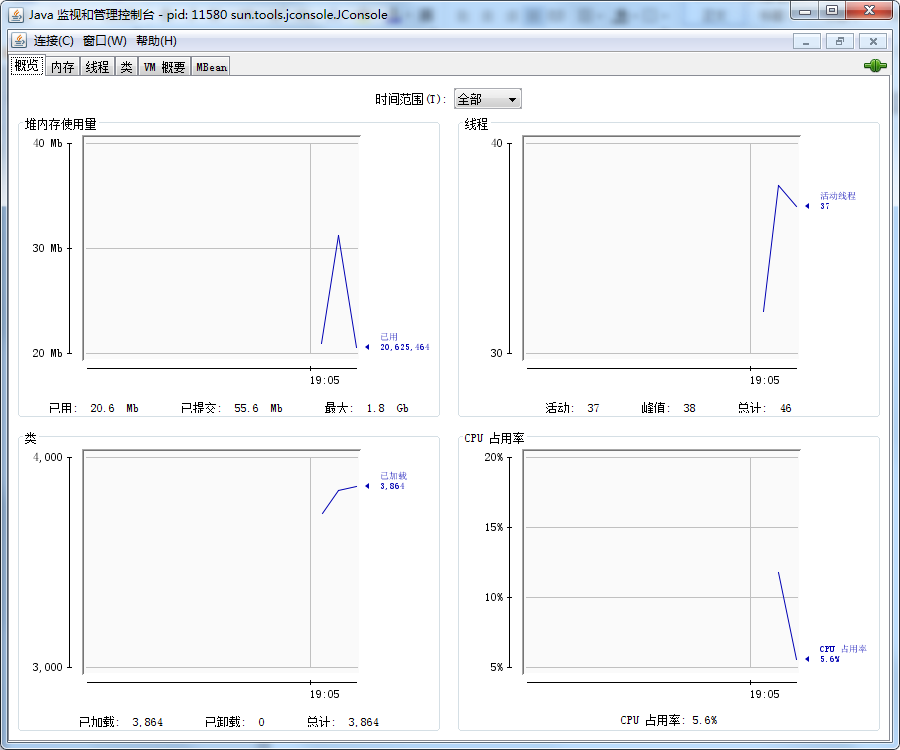
C:\Users\BYF>jps

11428 Jps

9860

11548

11580 JConsole



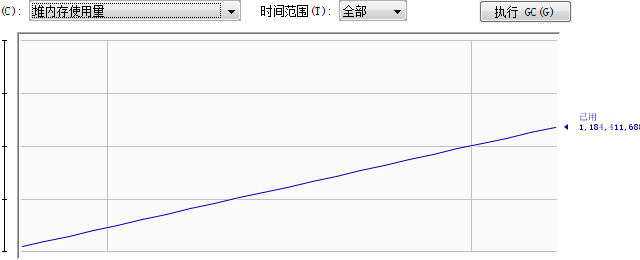
2）测试代码

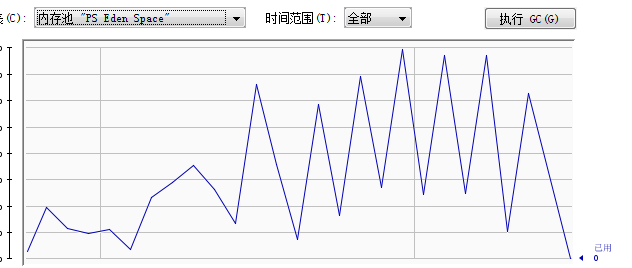
|  |
| --- |
| **public** **class** JConsoleTest {  // public byte[] b1 = new byte[128\*10224];    **public** JConsoleTest(){  **byte**[] b1 = **new** **byte**[128\*10224];  }    **public** **static** **void** main(String[] args) {  **try** {  Thread.*sleep*(5000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  *fill*(1000);  }  **private** **static** **void** fill(**int** i) {  List<JConsoleTest> jList = **new** ArrayList<>();  **for** (**int** j = 0; j < i; j++) {  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  jList.add(**new** JConsoleTest());  }  }  } |

区别：

// public byte[] b1 = new byte[128\*10224];

// 放在外边，没有被回收



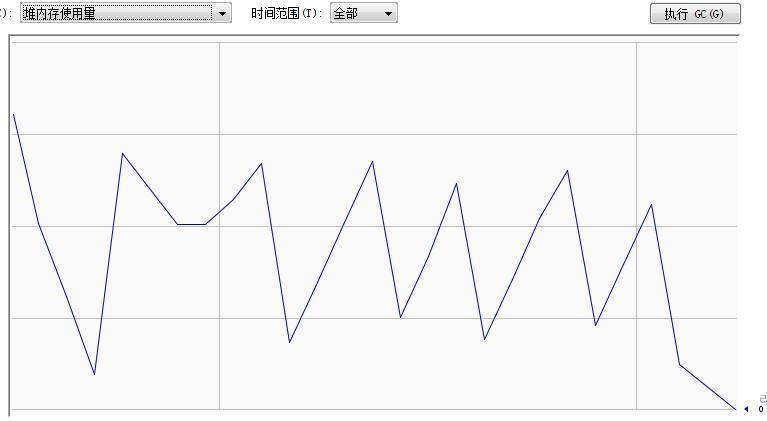


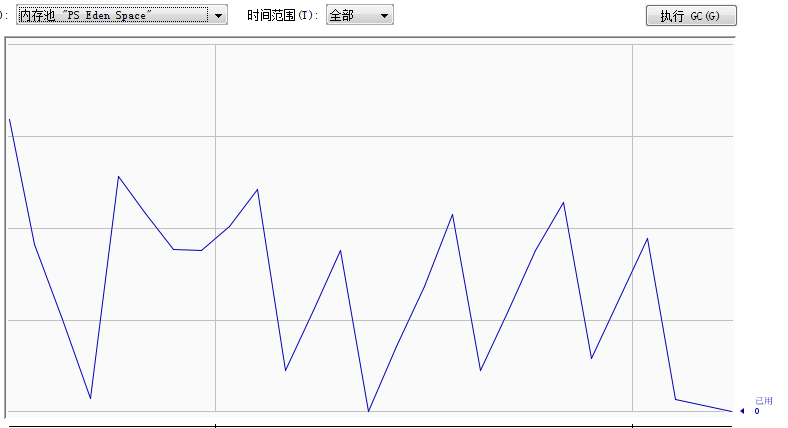
**public** JConsoleTest(){

// 局部变量，没人使用，过段时间就会被JVM回收，放在堆内存，被回收

**byte**[] b1 = **new** **byte**[128\*10224];

}





局部变量被回收，堆内存和Eden变化差不多近似

### 1.5杂谈

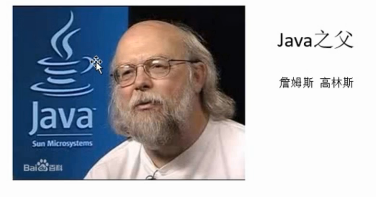
谈Java

java中间件：tomcat/jetty

Hadoop

Lucene搜索

### 1.6Java的发展史



Oak：嵌入式编程语言

Applet

1995.05：Java1.0 Write once run anywhere

1996.01：jvm sun Classic VM

1996.9首届JavaOne大会

1997.02 jdk1.1 内部类，反射，jar文件格式，jdbc，javabeans，rmi

1998 jdk1.2 Java EE Java ME Java SE swing，jit（Just in time），Hotspot VM

2000.5 jdk1.3 Timer，java2d

2002.02 jdk1.4 Structs、Hibernate，Spring1.X，正则表达式，Nio，日志，Xml解析器

2004 jdk1.5 tiger 自动装箱拆箱，泛型，注解，枚举，变长参数，增强for循环，Spring2.X，JUC

2006 jdk1.6 j2ee JDK6 提供脚本语言支持（动态语言），提供了编译api以及http服务器api，开源

2009 jdk1.7 Lambda Jigsaw OSGI

2011.07 jdk1.7

2014.3 jdk1.8

### 1.7Java技术体系

1. Java程序设计语言
2. 各硬件平台上的Java虚拟机
3. Class文件格式
4. Java API
5. 第三方java类库

Java SE Standard edition（桌面）

Java ME Mobile Edition（移动）

Java EE 13种技术综合

### 1.8jdk8的新特性

接口的默认方法和静态方法

Java8使用两个新概念扩展了接口的含义，默认方法和静态方法。

默认方法

可以在不破坏二进制兼容性的前提下，往现存接口中添加新的方法，即不在强调哪些实现类该接口的类也同时实现这个新方法。

Java8引入Lambda的利弊

1. 不用再写大量的匿名内部类，可读性提高
2. 集合操作改善，引入StreamAPI 把Map，Reduce，filter这样的基本函数式编程的概念与Java集合紧密集合，可以告别for、while、if。
3. 学习成本

Data API

重复注解

更好的类型推断：Java编译器的优化

Nashorm JavaScript引擎

使用Mataspace代替持久代PermCen space。在JVM参数方面，使用-XX:MetaSpaceSize和-XX:MaxMetaspaceSize代替原来的-XX:PermSize和-XX:MaxPermSize

### 1.9lanmbda表达式简介

New JButton.addActionListener(event -> System.out.println(“hello”));

## 第2节Java虚拟机

### 2.1classicvm

1. 世界上第一款商用的java虚拟机；
2. 只能纯解释器性，来执行java代码；
3. 基于Handler的对象查找方式；

### 2.2ExactVM

1. Exact Memory Management准确式内存管理；查找某个内存位置对象的类型；可以判断对象是否需要及时回收；
2. 编译器和解释器混合工作及两级及时编译器；具备高性能虚拟机的特性；
3. 只在Solaris平台发布；
4. 英雄气短；

### 2.3HotSpotVM

1. HotSpot VM的历史：
2. 优势：热点代码探测技术，程序计数器探测程序中经常被访问的方法；
3. 称霸武林；

### 2.4kvm

1. Kilobyte，简单，轻量，高度可一直；
2. 在手机平台上运行；

### 2.5JRockit

1. BEA
2. 世界上最快的Java虚拟机
3. 专注服务器端应用；
4. 优势：垃圾收集器；MissionControl服务套件

BEA JRockit Mission Control（JRMC）用来诊断漏洞并指出根本原因，该工具开销非常小，因此可以使用它来寻找生产环境中的系统内泄露；

JRMC于2005年12月面试，并从JRockitR26.0.0版本开始捆绑了整个工具套件，它是一组极地的开销来监控、管理和分析生产环境中的应用程序的工具。它包含三个独立的应用程序；内存泄露检测器（Memory Leak Detector）、JVM运行时分析器（Runtime Analyzer）和管理控制台（Management Console）

### 2.6j9

Azul VM ：高性能的Java虚拟机

Liquid VM：不需要操作系统

### 2.7dalvik

1. Android平台，没有遵循Java虚拟机规范；
2. Dex Dalvik Executale

### 2.8MicrosoftJVM

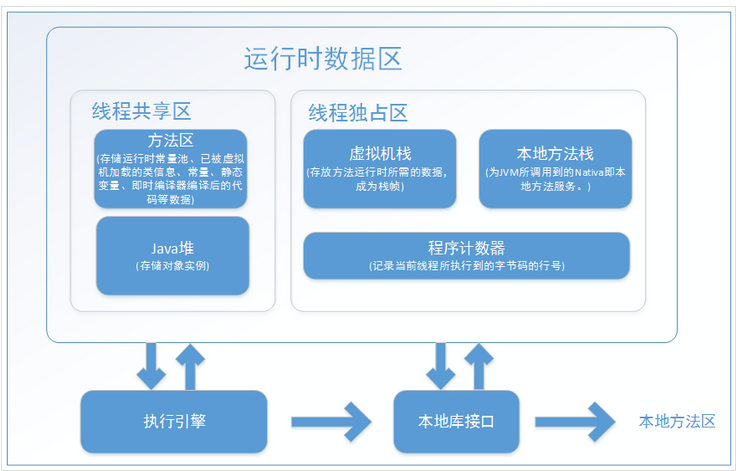
被Sun公司起诉，以微软败诉终止

### 2.9TaobaoVM

1）淘宝定制，依赖IntelCPU，垃圾收集，java本地方法调用降低开销

## 第3节Java内存区域

### 3.1简介



### 3.2程序计数器

1. 一块较小的内存空间，它可以看作是当前线程的所执行的字节码行号指示器；
2. 处于线程独占区；
3. Java方法<--记录正在执行的虚拟机字节码指令的地址；

Native方法<--计数器的值为undefined

1. 此区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。
2. goto java保留关键字，不让开发者用，防止后续成为jdk的关键字使用，重构代码；

### 3.3Java虚拟机栈

1. 虚拟机栈描述的是Java方法执行的动态内存模型；
2. 栈帧：每个方法执行，都会创建一个栈帧，伴随着方法从创建到执行完成。用于存储局部变量表，操作数栈，动态链接，方法出口等。

|  |
| --- |
| **public** **class** StackOverflowTest {  **public** **void** test() {  System.***out***.println("方法调用...");  test();  }    **public** **static** **void** main(String[] args) {  **new** StackOverflowTest().test();  }  } |

执行结果：

StackOverflow

栈足够多，则会出现OutOfMemory

1. 局部变量表

存放编译器可知的各种基本数据类型，引用类型，returnAddress类型。

内存空间在编译器完成分配，当进入一个方法时，这个方法需要在帧分配多少内存是固定的，在方法运行期间是不会改变局部变量表的大小

### 3.4本地方法栈

虚拟机栈：虚拟机执行Java方法服务；

本地方法栈：虚拟机执行native方法服务；

### 3.5堆内存

1. 存放对象实例；
2. 垃圾收集器管理的区域；
3. 新生代、老年代、Eden空间；
4. OutOfMemory

-Xmx -Xms

### 3.6方法区

1）存储运行时常量池，已被虚拟机加载的类信息，常量，静态变量，即时编译后的代码信息。

类信息：

类的版本

字段

方法

接口

2）方法区和永久代；在Hotspot中不区分，JVM规范中并不等价

3）垃圾回收在方法区的行为；

4）异常的定义：OutOfMemoryError

### 3.7直接内存和运行时常量池

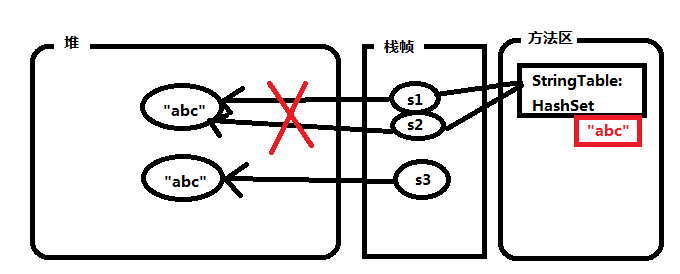
1）运行时常量池

|  |
| --- |
| **public** **static** **void** main(String[] args) {  // 字节码常量  String s1 = "abc";  String s2 = "abc";    // 运行时常量  String s3 = **new** String("abc");    System.***out***.println(s1 == s2);  System.***out***.println(s1 == s3);    // 将堆内存中的s3的值，放入运行时常量池，intern是一个navive方法  System.***out***.println(s1 == s3.intern());  } |

执行结果：

true

false

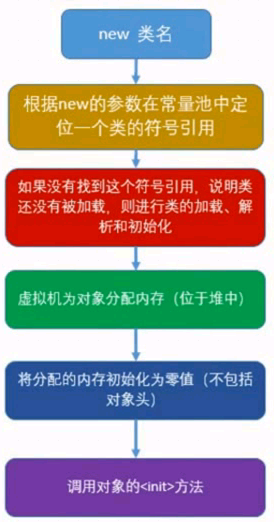
true

2）直接内存

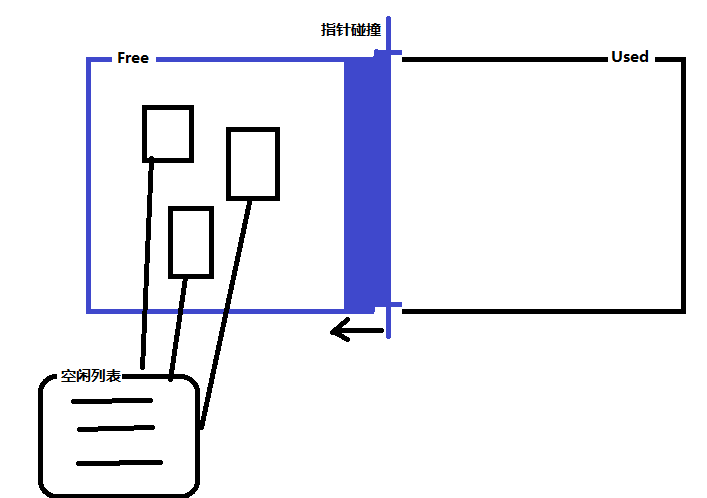
Jdk1.4弥补io的Nio

Nio：引入了一种基于通道的缓冲区的IO方式，它可以使用native函数库分配堆外内存，然后通过一个java堆中的buffer对象，作为这块内存的引用进行操作，分配堆外内存，不受java虚拟机的制约，会受到操作系统物理内存的制约。

### 3.8对象在内存中的布局-对象的创建



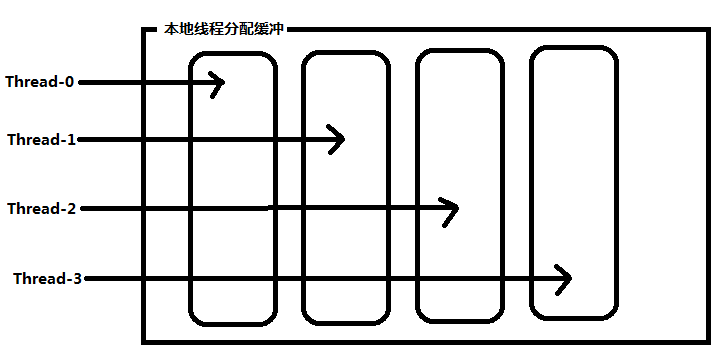
1. 给对象分配内存



1. 线程安全性问题

创建对象的线程，在堆内存分配时的线程安全问题，可以加锁，但效率太低；

本地线程分配缓冲



1. 初始化对象
2. 执行构造方法

|  |
| --- |
| **public** **class** User {  **private** Integer age;  **private** String name;    **public** User() {  **super**(); // 执行父类Object对象的构造方法  System.***out***.println("User has bean created~!");  }  **public** Integer getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  @Override  **public** String toString() {  **return** "User [age=" + age + ", name=" + name + "]";  }    } |
| **public** **class** CreateObjectTest {  **public** **static** **void** main(String[] args) {  User user = **new** User();  System.***out***.println(user);  }  } |

执行结果：

User has bean created~!

User [age=null, name=null]

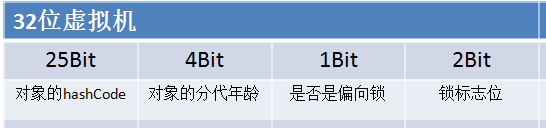
### 3.9探究对象的结构

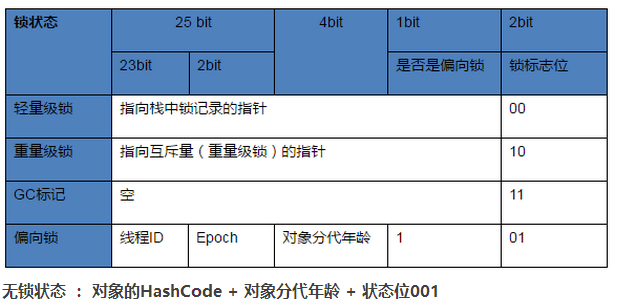
1. 对象的结构

①Header（对象头）32位/64位

自身运行时的数据（Mark Word）

哈希值，GC分代年龄（分代收集算法），锁状态标志，线程持有的锁，偏向线程ID，偏向时间戳





类型指针：对象元数据指向他的指针，（数组会有一个计算数组长度的length）

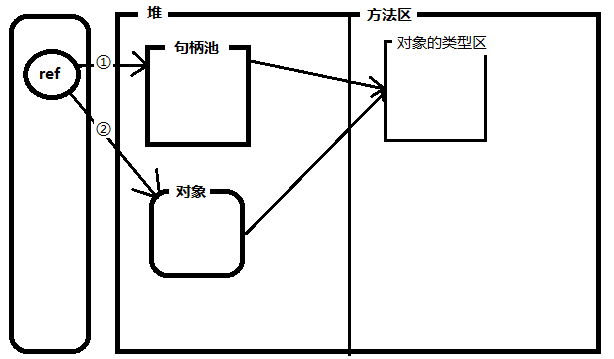
②InstanceData

Longs Doubles short/chars

③Padding

### 3.10深入理解对象的访问定位

1. 使用句柄：对象地址改变，对象的引用不需要改变，只需要将句柄进行改变即可
2. 直接指针：对象的实例指针；对象类型的指针；



## 第4节垃圾回收

### 4.1概述

1. 如何判断对象为垃圾对象？

* 引用计数器；
* 可达性分析法；

1. 如何回收？

回收策略

标记--清除算法

复制算法

标记--整理算法

分代收集算法

垃圾回收器

Serial

Parnew

Cms

G1

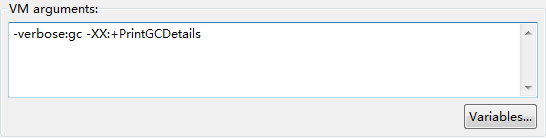
1. 何时回收？

### 4.2判断对象是否存活算法-引用计数法详解

1. 引用计数算法

在对象中添加一个引用计数器，当有地方引用这个对象的时候，引用计数器的值就+1，当引用失效，计数器-1；

-verbose:gc -XX:+PrintGCDetails



|  |
| --- |
| **public** **class** RefCounterGCTest {    **private** Object instance;    **public** RefCounterGCTest() {  **byte**[] m = **new** **byte**[20\*1024\*1024];  }  **public** **static** **void** main(String[] args) {  RefCounterGCTest m1 = **new** RefCounterGCTest();  RefCounterGCTest m2 = **new** RefCounterGCTest();    m1.instance = m2;  m2.instance = m1;    m1 = **null**;  m2 = **null**;    System.*gc*();  }    } |

[GC (System.gc()) [PSYoungGen: 22354K->720K(36352K)] 42834K->21208K(119808K), 0.0197557 secs] [Times: user=0.00 sys=0.00, real=0.02 secs]

[Full GC (System.gc()) [PSYoungGen: 720K->0K(36352K)] [ParOldGen: 20488K->532K(83456K)] 21208K->532K(119808K), [Metaspace: 2691K->2691K(1056768K)], 0.0053040 secs] [Times: user=0.06 sys=0.00, real=0.01 secs]

Heap

PSYoungGen total 36352K, used 312K [0x00000000d7b00000, 0x00000000da380000, 0x0000000100000000)

eden space 31232K, 1% used [0x00000000d7b00000,0x00000000d7b4e2b8,0x00000000d9980000)

from space 5120K, 0% used [0x00000000d9980000,0x00000000d9980000,0x00000000d9e80000)

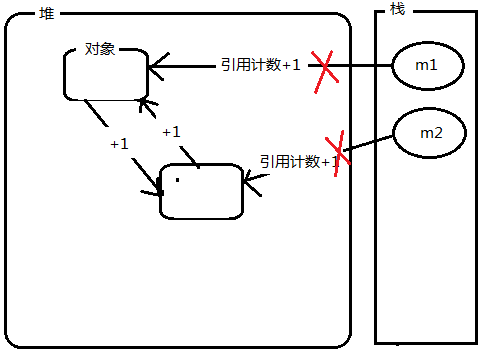
to space 5120K, 0% used [0x00000000d9e80000,0x00000000d9e80000,0x00000000da380000)

ParOldGen total 83456K, used 532K [0x0000000087000000, 0x000000008c180000, 0x00000000d7b00000)

object space 83456K, 0% used [0x0000000087000000,0x0000000087085010,0x000000008c180000)

Metaspace used 2698K, capacity 4486K, committed 4864K, reserved 1056768K

class space used 297K, capacity 386K, committed 512K, reserved 1048576K



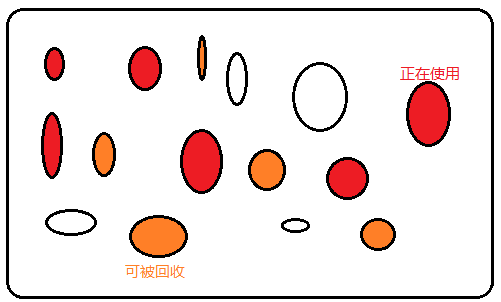
当引用被赋值为null，引用计数虽然-1，但仍有对象的引用不为0，若按照引用计数回收，那么堆中相互引用的对象无法回收。

### 4.3判断对象是否存活算法-可达性分析法详解

1. 作为GCRoots的对象（引用链/路径不可达）

* 虚拟机栈（栈帧中局部变量表）
* 方法区的类属性引用的对象
* 方法区中常量引用的对象
* 本地方法栈中引用的对象

### 4.4算法-标记清除算法



效率问题；

空间问题；

### 4.5算法-复制算法

堆

新生代

Eden 伊甸园

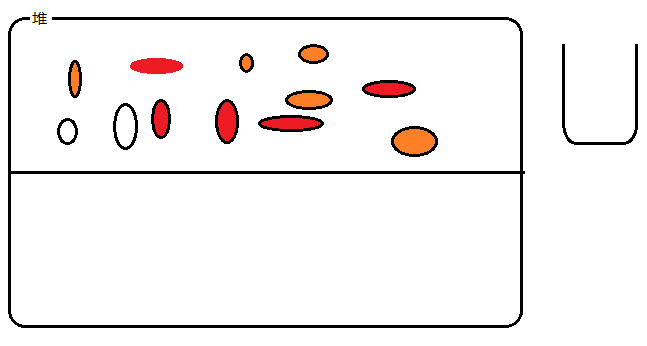
Survivor 存活区

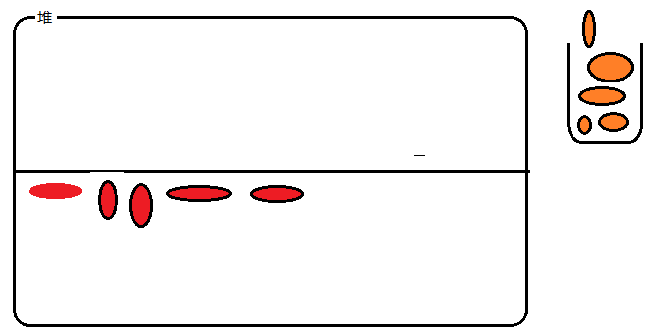
Tenured Gen

老年代

方法区

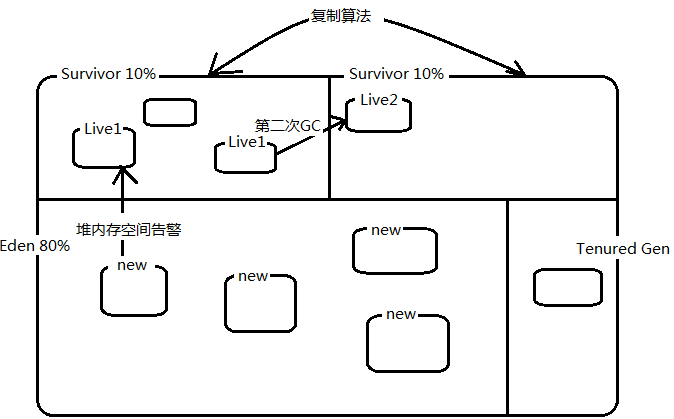
栈 本地方法栈 程序计数器





内存同时用的只有一半，造成了极大的浪费

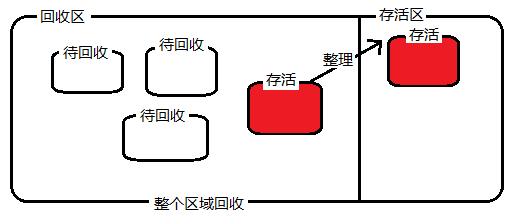
堆分区：



Suvivor1 和Suvivor2就相当于复制算法中的两块内存空间，这样即达到了内存复制的效果，同时，只占用内存的20%。

### 4.6算法-标记整理算法和分代收集算法

1）标记--整理--清除算法



2）分代收集算法（标记整理算法+复制算法）

新生代内存回收率较高的区域选择复制算法；

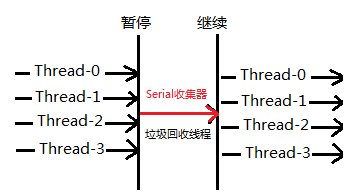
老年代内存回收率较低的区域选择标记整理算法

### 4.7收集器-serial收集器详解

虚拟机规范中对这块并没有要求，不同的公司可以自己实现。

1）JDK1.3之前，回收新生代唯一的垃圾收集器

2）单线程垃圾收集器



3）桌面应用

### 4.8收集器-parnew收集器详解

1. （新生代收集器）从单线程回收，变为多线程回收，但是在桌面应用，多线程回收可能效率不如单线程回收。



2）关注点：缩短垃圾回收的时间，应用中断的时间

### 4.9收集器-parallel收集器详解

1. 复制算法（新生代收集器，和parnew一样是多线程）
2. 多线程收集器
3. 关注点：达到可控制的吞吐量，CPU用于运行用户代码的时间与CPU消耗的总时间的比值；

吞吐量 = （执行用户代码的时间）/（执行用户代码的时间 + 垃圾回收的时间）

-XX:MaxGCPauseMillis 垃圾收集器最大停顿时间

值设置过小，垃圾收集频率增高

-XX:GCTimeRatio 吞吐量大小

（0,100）

99：垃圾回收占1%

### 4.10收集器-cms收集器详解（Concurrent Mark Sweep 并发标记--清除）

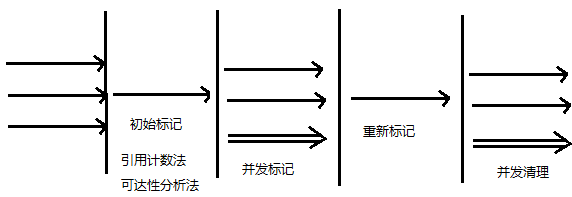
1. Jdk1.5出现CMS，并发收集器，边扔垃圾，边打扫，回收老年代内存；
2. 工作过程：

初始标记

并发标记

重新标记

并发清理



1. 优点

并发收集

低停顿

1. 缺点
   1. 占用大量的CPU资源
   2. 无法处理浮动垃圾
   3. 出现Current Mode Failure
   4. 空间碎片

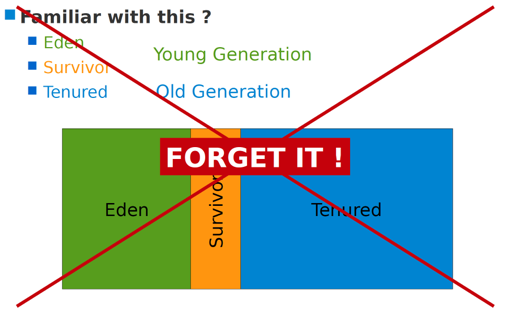
### 4.11最牛的垃圾收集器-g1收集器详解

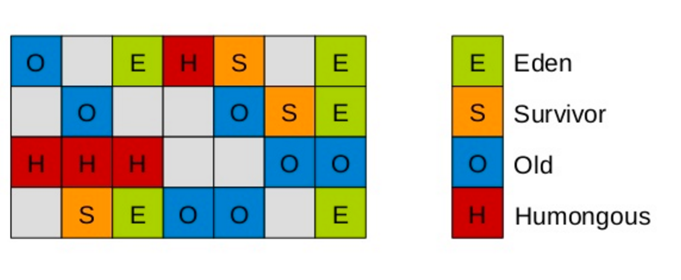
G：垃圾

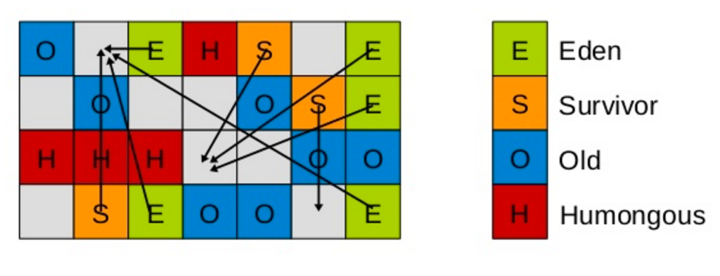
1. 历史：jdk9替代cms垃圾收集器
2. 优势
   * 1. 并行和并发
     2. 分代收集

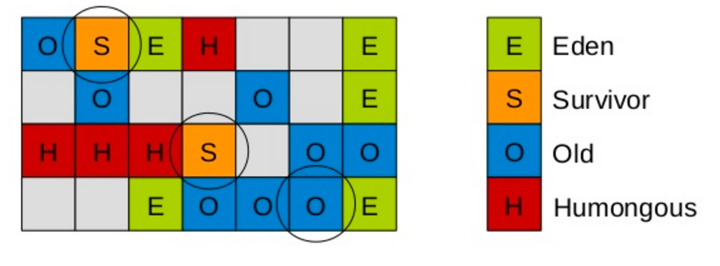
<https://www.cnblogs.com/ASPNET2008/p/6496481.html>

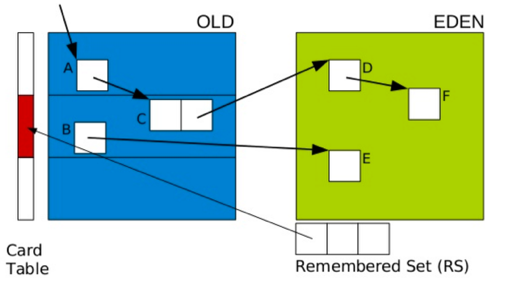
<https://www.cnblogs.com/lirenzuo/p/7576622.html?utm_source=debugrun&utm_medium=referral>











1. 步骤

标记整理

1. 与CMS比较

与CMS类似，G1更关注的是延迟；

其实通过上面好多项都可以看见把CMS参数去掉很多，之后一些相关也废弃了，就是想让G1替代CMS，由于CMS采用的是标记清除，而G1使用的是标记整理所以G1在若干次GC后，不需要类似CMS那样必须进行一次~~碎片整理~~。

## 第5节内存分配

### 5.1概述

1. 内存分配策略
   1. 优先分配到eden；
   2. 大对象直接分配到老年代；
   3. 长期存活的对象分配到老年代；
   4. 空间分配担保；
   5. 动态对象年龄判断。

### 5.2Eden区域

-verbose:gc -XX:+PrintGCDetails -XX:+UseSerialGC

|  |
| --- |
| **public** **class** RefCounterGCTest {  **public** **static** **void** main(String[] args) {  **byte**[] b1 = **new** **byte**[4 \* 1024 \* 1024];  }  } |

[GC (Allocation Failure) [DefNew: 22476K->532K(37440K), 0.0016562 secs] 22476K->532K(120768K), 0.0019324 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[Full GC (System.gc()) [Tenured: 0K->530K(83328K), 0.0023321 secs] 21658K->530K(120768K), [Metaspace: 2692K->2692K(1056768K)], 0.0023844 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

Heap

def new generation total 37568K, used 668K [0x0000000087000000, 0x00000000898c0000, 0x00000000af550000)

eden space 33408K, 2% used [0x0000000087000000, 0x00000000870a7218, 0x00000000890a0000)

from space 4160K, 0% used [0x00000000890a0000, 0x00000000890a0000, 0x00000000894b0000)

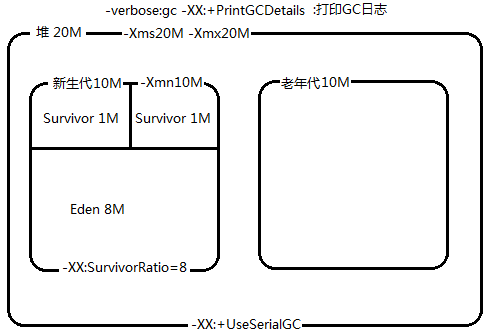
to space 4160K, 0% used [0x00000000894b0000, 0x00000000894b0000, 0x00000000898c0000)

tenured generation total 83328K, used 530K [0x00000000af550000, 0x00000000b46b0000, 0x0000000100000000)

the space 83328K, 0% used [0x00000000af550000, 0x00000000af5d4900, 0x00000000af5d4a00, 0x00000000b46b0000)

Metaspace used 2698K, capacity 4486K, committed 4864K, reserved 1056768K

class space used 297K, capacity 386K, committed 512K, reserved 1048576K



-verbose:gc -XX:+PrintGCDetails -XX:+UseSerialGC -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8

|  |
| --- |
| **public** **class** EdenAreaTest {  **public** **static** **void** main(String[] args) {  **byte**[] b1 = **new** **byte**[2 \* 1024 \* 1024];  **byte**[] b2 = **new** **byte**[2 \* 1024 \* 1024];  **byte**[] b3 = **new** **byte**[2 \* 1024 \* 1024];  **byte**[] b4 = **new** **byte**[4 \* 1024 \* 1024];    System.*gc*();  }  } |

[GC (Allocation Failure) [DefNew: 7131K->532K(9216K), 0.0067887 secs] 7131K->6676K(19456K), 0.0068376 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]

[Full GC (System.gc()) [Tenured: 6144K->6144K(10240K), 0.0025242 secs] 10932K->10770K(19456K), [Metaspace: 2692K->2692K(1056768K)], 0.0025838 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

Heap

def new generation total 9216K, used 4790K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)

eden space 8192K, 58% used [0x00000000fec00000, 0x00000000ff0ad9e0, 0x00000000ff400000)

from space 1024K, 0% used [0x00000000ff500000, 0x00000000ff500000, 0x00000000ff600000)

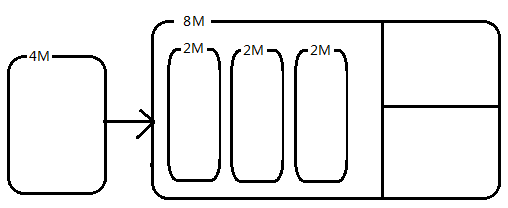
to space 1024K, 0% used [0x00000000ff400000, 0x00000000ff400000, 0x00000000ff500000)

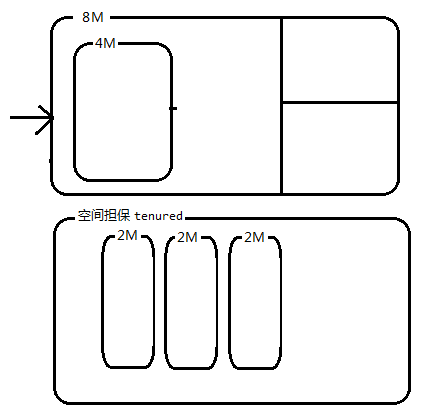
tenured generation total 10240K, used 6144K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)

the space 10240K, 60% used [0x00000000ff600000, 0x00000000ffc00030, 0x00000000ffc00200, 0x0000000100000000)

Metaspace used 2698K, capacity 4486K, committed 4864K, reserved 1056768K

class space used 297K, capacity 386K, committed 512K, reserved 1048576K





### 5.3大对象直接进老年代

-verbose:gc -XX:+PrintGCDetails -XX:+UseSerialGC -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8 -XX:PretenureSizeThreshold=6M

-XX:PretenureSizeThreshold=6M 指定大对象的阈值

|  |
| --- |
| **public** **class** EdenAreaTest {  **public** **static** **void** main(String[] args) {  // byte[] b1 = new byte[2 \* 1024 \* 1024];  // byte[] b2 = new byte[2 \* 1024 \* 1024];  // byte[] b3 = new byte[2 \* 1024 \* 1024];  // byte[] b4 = new byte[4 \* 1024 \* 1024];  **byte** [] b5 = **new** **byte**[7 \* 1024 \* 1024];    System.*gc*();  }  } |

[Full GC (System.gc()) [Tenured: 7168K->7699K(10240K), 0.0029489 secs] 8155K->7699K(19456K), [Metaspace: 2691K->2691K(1056768K)], 0.0030139 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]

Heap

def new generation total 9216K, used 82K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)

eden space 8192K, 1% used [0x00000000fec00000, 0x00000000fec14920, 0x00000000ff400000)

from space 1024K, 0% used [0x00000000ff400000, 0x00000000ff400000, 0x00000000ff500000)

to space 1024K, 0% used [0x00000000ff500000, 0x00000000ff500000, 0x00000000ff600000)

tenured generation total 10240K, used 7699K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)

the space 10240K, 75% used [0x00000000ff600000, 0x00000000ffd84fe0, 0x00000000ffd85000, 0x0000000100000000)

Metaspace used 2698K, capacity 4486K, committed 4864K, reserved 1056768K

class space used 297K, capacity 386K, committed 512K, reserved 1048576K

### 5.4长期存活的对象进入老年代

-XX:MaxTenuringThreshold 15

Age 1+1+1

定义一个小于Survivor区域的对象，多次GC后，如果达到存活次数超过15，对象进入老年代，Survivor区域为0；

### 5.5空间分配担保

-XX:+HandlePromotionFailure

+开启，-禁用

### 5.6逃逸分析与栈上分配

逃逸分析：分析对象的作用域。

当一个对象定义在方法体内部有效，则没有逃逸

|  |
| --- |
| **public** **class** StackAllocationTest {  **public** StackAllocationTest obj;  **public** **static** **void** main(String[] args) {    }  /\*\*  \* 方法返回StackAllocationTest对象，发生逃逸  \* **@return** StackAllocationTest  \*/  **public** StackAllocationTest getInstance() {  **return** obj == **null** ? **new** StackAllocationTest(): obj;  }    /\*\*  \* 为类的成员属性赋值，发生逃逸  \*/  **public** **void** setObj(){  **this**.obj = **new** StackAllocationTest();  }    /\*\*  \* 没有发生逃逸  \*/  **public** **void** useStackAllocationTest() {  StackAllocationTest s = **new** StackAllocationTest();  }    /\*\*  \* 调用成员方法，发生逃逸  \*/  **public** **void** useStackAllocationTest2() {  StackAllocationTest s = getInstance();  }  } |

## 第6节虚拟机工具

### 6.1介绍

如果经常创建朝生夕死的大对象，对虚拟机的影响会很大。

Jps

Jstat

Jinfo

Jmap

Jhat

Jstack

jconsole

### 6.2jps详解

M 运行时传入主类的参数；

V 虚拟机参数；

L 运行时主类全名或者 jar包的名称

JVM Process Status：任务管理器的详细信息

C:\Users\BYF>jps

7608

10748 Jps

本地虚拟机唯一id lvmid local virtual machine id

C:\Users\BYF>jps -l 查看包路径及类

11152 sun.tools.jps.Jps

12032 com.byf.jvm09JpsTest.JpsTest

11496 com.byf.jvm09JpsTest.JpsTest

7608

C:\Users\BYF>jps -m 查看运行时入参

12032 JpsTest

9008 Jps -m

11496 JpsTest

7608

C:\Users\BYF>jps -v 查看应用运行时的参数

12032 JpsTest -Dfile.encoding=GBK

11496 JpsTest -Dfile.encoding=GBK

7608 -Dosgi.requiredJavaVersion=1.8 -XX:+UseG1GC -XX:+UseStringDeduplication -Dosgi.requiredJavaVersion=1.8 -Xms256m -Xmx1024m

10572 Jps -Denv.class.path=.;F:\Program Files\Java\jdk1.8.0\_172\lib\dt.jar;F:\Program Files\Java\jdk1.8.0\_172\lib\tools.jar;E:\R

tmq-all-4.2.0-bin-release\bin; -Dapplication.home=F:\Program Files\Java\jdk1.8.0\_172 -Xms8m

### 6.3jstat详解

类装载，内存，垃圾收集，jit编译的信息

#### jstat命令使用

jstat命令可以查看堆内存各部分的使用量，以及加载类的数量。命令的格式如下：

jstat [-命令选项] [vmid] [间隔时间/毫秒] [查询次数]

注意：使用的jdk版本是jdk8.

#### **6.3.1类加载统计：**

C:\Users\Administrator>jstat -class 2060

Loaded Bytes Unloaded Bytes Time

15756 17355.6 0 0.0 11.29

* Loaded:加载class的数量
* Bytes：所占用空间大小
* Unloaded：未加载数量
* Bytes:未加载占用空间
* Time：时间

#### **6.3.2编译统计**

C:\Users\Administrator>jstat -compiler 2060Compiled Failed Invalid Time FailedType FailedMethod

9142 1 0 5.01 1 org/apache/felix/resolver/ResolverImpl mergeCandidatePackages

* Compiled：编译数量。
* Failed：失败数量
* Invalid：不可用数量
* Time：时间
* FailedType：失败类型
* FailedMethod：失败的方法

#### **6.3.3垃圾回收统计**

C:\Users\Administrator>jstat -gc 2060

S0C S1C S0U S1U EC EU OC OU MC MU CCSC CCSU YGC YGCT FGC FGCT GCT

20480.0 20480.0 0.0 13115.3 163840.0 113334.2 614400.0 436045.7 63872.0 61266.5 0.0 0.0 149 3.440 8 0.295 3.735

* S0C：第一个幸存区的大小
* S1C：第二个幸存区的大小
* S0U：第一个幸存区的使用大小
* S1U：第二个幸存区的使用大小
* EC：伊甸园区的大小
* EU：伊甸园区的使用大小
* OC：老年代大小
* OU：老年代使用大小
* MC：方法区大小
* MU：方法区使用大小
* CCSC:压缩类空间大小
* CCSU:压缩类空间使用大小
* YGC：年轻代垃圾回收次数
* YGCT：年轻代垃圾回收消耗时间
* FGC：老年代垃圾回收次数
* FGCT：老年代垃圾回收消耗时间
* GCT：垃圾回收消耗总时间

#### **6.3.4堆内存统计**

C:\Users\Administrator>jstat -gccapacity 2060

NGCMN NGCMX NGC S0C S1C EC OGCMN OGCMX OGC OC MCMN MCMX MC CCSMN CCSMX CCSC YGC FGC

204800.0 204800.0 204800.0 20480.0 20480.0 163840.0 614400.0 614400.0 614400.0 614400.0 0.0 63872.0 63872.0 0.0 0.0 0.0 149 8

* NGCMN：新生代最小容量
* NGCMX：新生代最大容量
* NGC：当前新生代容量
* S0C：第一个幸存区大小
* S1C：第二个幸存区的大小
* EC：伊甸园区的大小
* OGCMN：老年代最小容量
* OGCMX：老年代最大容量
* OGC：当前老年代大小
* OC:当前老年代大小
* MCMN:最小元数据容量
* MCMX：最大元数据容量
* MC：当前元数据空间大小
* CCSMN：最小压缩类空间大小
* CCSMX：最大压缩类空间大小
* CCSC：当前压缩类空间大小
* YGC：年轻代gc次数
* FGC：老年代GC次数

#### **6.3.5新生代垃圾回收统计**

C:\Users\Administrator>jstat -gcnew 7172

S0C S1C S0U S1U TT MTT DSS EC EU YGC YGCT

40960.0 40960.0 25443.1 0.0 15 15 20480.0 327680.0 222697.8 12 0.736

* S0C：第一个幸存区大小
* S1C：第二个幸存区的大小
* S0U：第一个幸存区的使用大小
* S1U：第二个幸存区的使用大小
* TT:对象在新生代存活的次数
* MTT:对象在新生代存活的最大次数
* DSS:期望的幸存区大小
* EC：伊甸园区的大小
* EU：伊甸园区的使用大小
* YGC：年轻代垃圾回收次数
* YGCT：年轻代垃圾回收消耗时间

#### **6.3.6新生代内存统计**

C:\Users\Administrator>jstat -gcnewcapacity 7172

NGCMN NGCMX NGC S0CMX S0C S1CMX S1C ECMX EC YGC FGC

409600.0 409600.0 409600.0 40960.0 40960.0 40960.0 40960.0 327680.0 327680.0 12 0

* NGCMN：新生代最小容量
* NGCMX：新生代最大容量
* NGC：当前新生代容量
* S0CMX：最大幸存1区大小
* S0C：当前幸存1区大小
* S1CMX：最大幸存2区大小
* S1C：当前幸存2区大小
* ECMX：最大伊甸园区大小
* EC：当前伊甸园区大小
* YGC：年轻代垃圾回收次数
* FGC：老年代回收次数

#### **6.3.7老年代垃圾回收统计**

C:\Users\Administrator>jstat -gcold 7172

MC MU CCSC CCSU OC OU YGC FGC FGCT GCT

33152.0 31720.8 0.0 0.0 638976.0 184173.0 12 0 0.000 0.736

* MC：方法区大小
* MU：方法区使用大小
* CCSC:压缩类空间大小
* CCSU:压缩类空间使用大小
* OC：老年代大小
* OU：老年代使用大小
* YGC：年轻代垃圾回收次数
* FGC：老年代垃圾回收次数
* FGCT：老年代垃圾回收消耗时间
* GCT：垃圾回收消耗总时间

#### **6.3.8老年代内存统计**

C:\Users\Administrator>jstat -gcoldcapacity 7172

OGCMN OGCMX OGC OC YGC FGC FGCT GCT

638976.0 638976.0 638976.0 638976.0 12 0 0.000 0.736

* OGCMN：老年代最小容量
* OGCMX：老年代最大容量
* OGC：当前老年代大小
* OC：老年代大小
* YGC：年轻代垃圾回收次数
* FGC：老年代垃圾回收次数
* FGCT：老年代垃圾回收消耗时间
* GCT：垃圾回收消耗总时间

#### **6.3.9元数据空间统计**

C:\Users\Administrator>jstat -gcmetacapacity 7172

MCMN MCMX MC CCSMN CCSMX CCSC YGC FGC FGCT GCT

0.0 33152.0 33152.0 0.0 0.0 0.0 12 0 0.000 0.736

* MCMN:最小元数据容量
* MCMX：最大元数据容量
* MC：当前元数据空间大小
* CCSMN：最小压缩类空间大小
* CCSMX：最大压缩类空间大小
* CCSC：当前压缩类空间大小
* YGC：年轻代垃圾回收次数
* FGC：老年代垃圾回收次数
* FGCT：老年代垃圾回收消耗时间
* GCT：垃圾回收消耗总时间

#### **6.3.10总结垃圾回收统计**

C:\Users\Administrator>jstat -gcutil 7172

S0 S1 E O M CCS YGC YGCT FGC FGCT GCT

62.12 0.00 81.36 28.82 95.68 - 12 0.736 0 0.000 0.736

* S0：幸存1区当前使用比例
* S1：幸存2区当前使用比例
* E：伊甸园区使用比例
* O：老年代使用比例
* M：元数据区使用比例
* CCS：压缩使用比例
* YGC：年轻代垃圾回收次数
* FGC：老年代垃圾回收次数
* FGCT：老年代垃圾回收消耗时间
* GCT：垃圾回收消耗总时间

#### **6.3.11JVM编译方法统计**

C:\Users\Administrator>jstat -printcompilation 7172

Compiled Size Type Method

4608 16 1 org/eclipse/emf/common/util/SegmentSequence$SegmentSequencePool$SegmentsAccessUnit reset

* Compiled：最近编译方法的数量
* Size：最近编译方法的字节码数量
* Type：最近编译方法的编译类型。
* Method：方法名标识。

### 6.4jinfo详解

C:\Users\BYF>jinfo

Usage:

jinfo [option] <pid>

(to connect to running process)

jinfo [option] <executable <core>

(to connect to a core file)

jinfo [option] [server\_id@]<remote server IP or hostname>

(to connect to remote debug server)

where <option> is one of:

-flag <name> to print the value of the named VM flag

-flag [+|-]<name> to enable or disable the named VM flag

-flag <name>=<value> to set the named VM flag to the given value

-flags to print VM flags

-sysprops to print Java system properties

<no option> to print both of the above

-h | -help to print this help message

实时查看和调整虚拟机的各项参数

C:\Users\BYF>jinfo -flag UseSerialGC 12032

-XX:-UseSerialGC

C:\Users\BYF>jinfo -flag UseG1GC 7608

-XX:+UseG1GC

（Eclipse使用G1垃圾收集器）

### 6.5jmap详解

-XX:HeapDumpOnOutOfMemoryError

C:\Users\BYF>jmap -dump:format=b,file=d:\a.bin 12032

Dumping heap to D:\a.bin ...

Heap dump file created

### 6.6jhat详解

JVM Heap Analysis Status

C:\Users\BYF>jhat d:\a.bin

Reading from d:\a.bin...

Dump file created Sun Nov 04 19:41:12 CST 2018

Snapshot read, resolving...

Resolving 17256 objects...

Chasing references, expect 3 dots...

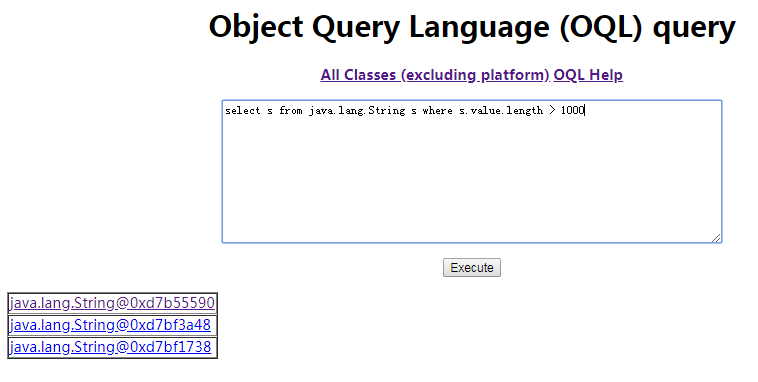
Eliminating duplicate references...

Snapshot resolved.

Started HTTP server on port 7000

Server is ready.





### 6.7jstack详解

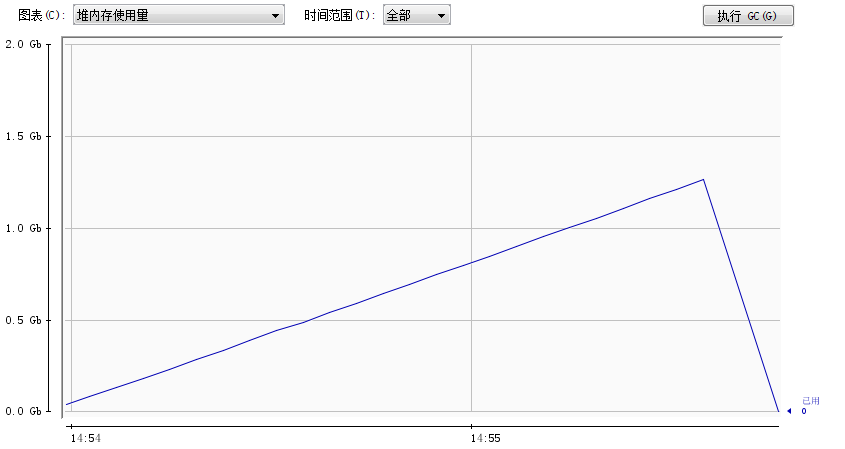
Thread.getAllStackTraces();

### 6.8可视化工具-Jconsole内存监控

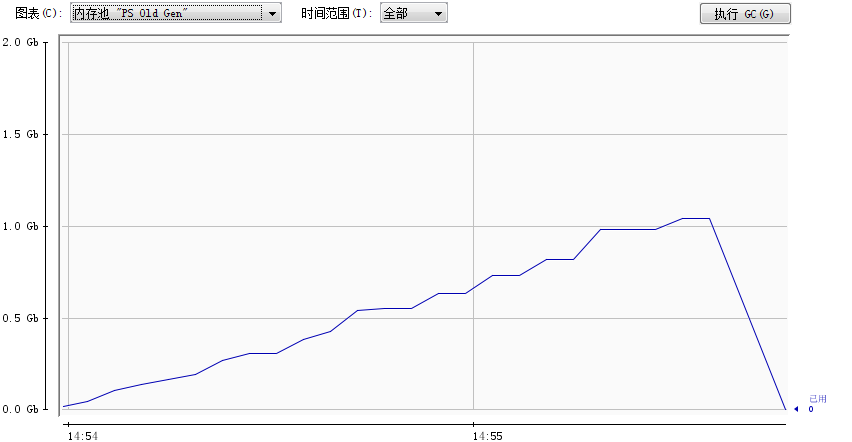
1. JConsole内存监控

|  |
| --- |
| **public** **class** JConsoleTest {  // 防止垃圾回收，声明public，防止在类的局部位置  **public** **byte**[] b1 = **new** **byte**[128\*10224];    /\*public JConsoleTest(){  byte[] b1 = new byte[128\*10224];  }\*/    **public** **static** **void** main(String[] args) {  **try** {  Thread.*sleep*(5000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  *fill*(1000);  }  **private** **static** **void** fill(**int** i) {  List<JConsoleTest> jList = **new** ArrayList<>();  **for** (**int** j = 0; j < i; j++) {  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  jList.add(**new** JConsoleTest());  }  }  } |

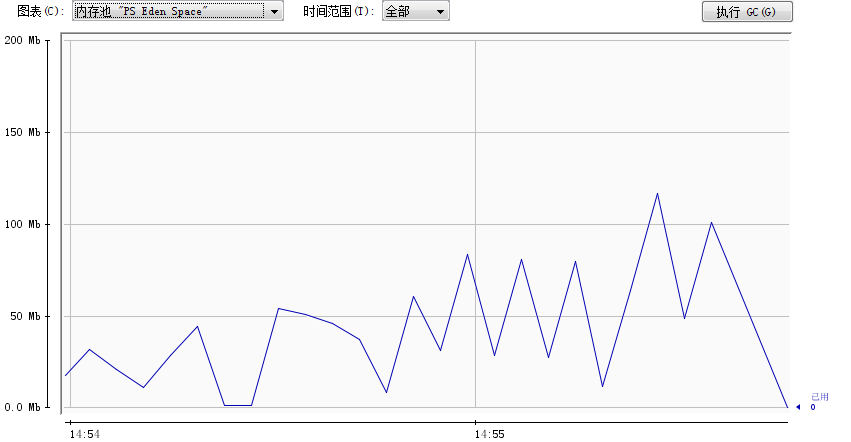
堆内存使用量监控：



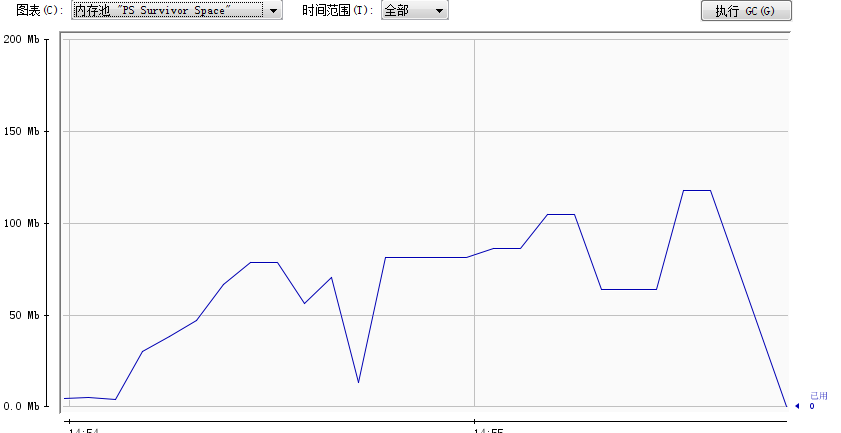
老年代不断上升：



新生代交替上涨：



Survivor区交替上涨

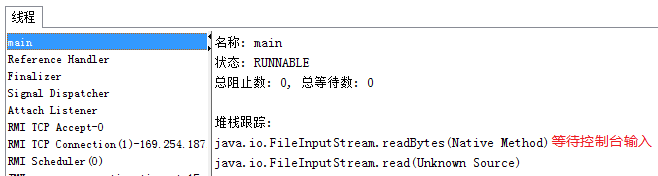


### 6.9可视化工具-Jconsole线程监控

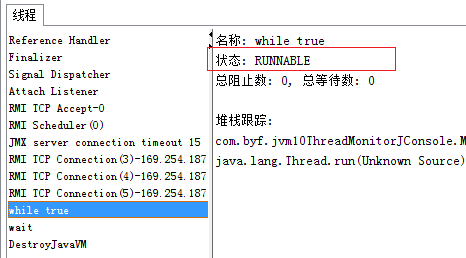
JConsole线程监控

|  |
| --- |
| **public** **class** MonitorThreadTest {    **public** **static** **void** main(String[] args) {  Scanner sc = **new** Scanner(System.***in***);  sc.nextInt();    **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  **while**(**true**){    }  }  },"while true").start();    *testWait*(**new** Object());  }  **private** **static** **void** testWait(Object obj) {  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  **synchronized**(obj){  **try** {  obj.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  },"wait").start();  }    } |

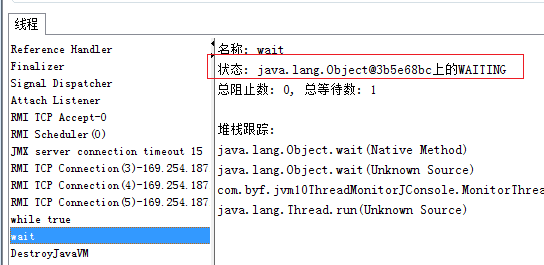
执行后，打开JConsole的线程监控界面，可以看到main线程是Runnable状态



输入一个整数，while true线程执行，同样是Runnbale状态



可以看到wait线程以及出于等待状态，等待其他线程唤醒



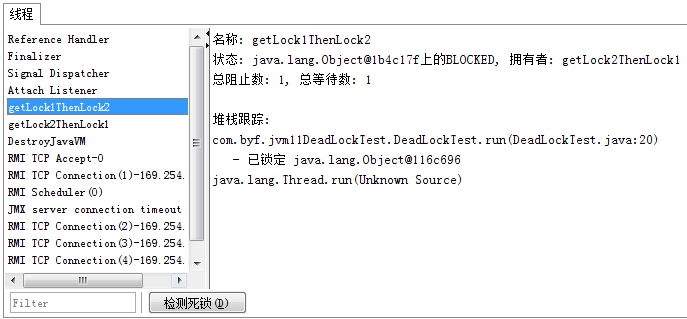
### 6.10死锁原理以及可视化虚拟机工具-Jconsole线程死锁监控

JConsole死锁监控

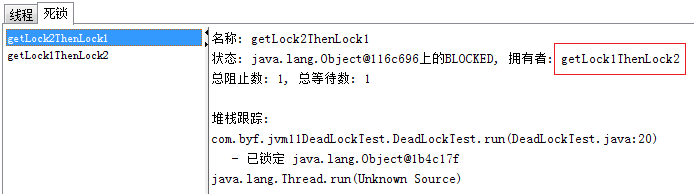
|  |
| --- |
| **public** **class** DeadLockTest **implements** Runnable{    **private** Object obj1;  **private** Object obj2;  **public** DeadLockTest(Object obj1,Object obj2){  **this**.obj1 = obj1;  **this**.obj2 = obj2;  }  @Override  **public** **void** run() {  **synchronized**(obj1){  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **synchronized**(obj2){  System.***out***.println("Hello");  }  }  }    } |

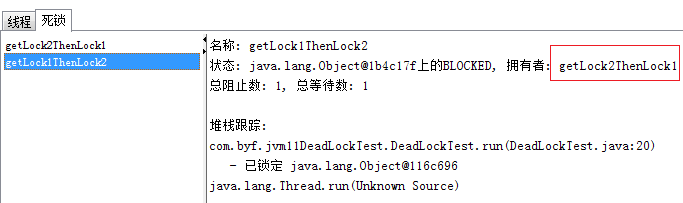
|  |
| --- |
| **public** **class** MainTest {  **public** **static** **void** main(String[] args) {  Object obj1 = **new** Object();  Object obj2 = **new** Object();    **new** Thread(**new** DeadLockTest(obj1,obj2), "getLock1ThenLock2").start();  **new** Thread(**new** DeadLockTest(obj2,obj1), "getLock2ThenLock1").start();    }    } |

打开JConsole线程监控工具，查看两个线程，出于锁定状态



点击检测死锁，可以查看当前存在的死锁



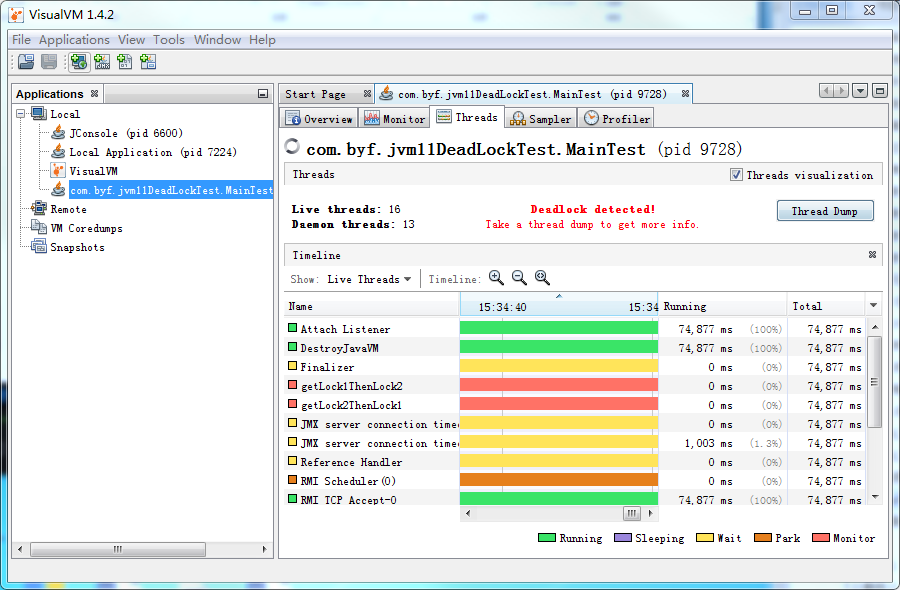


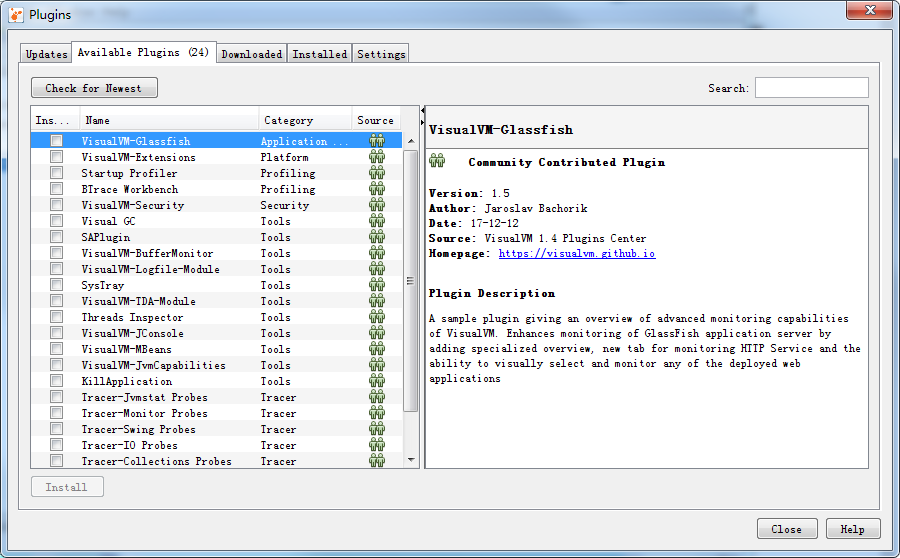
### 6.11VisualVM使用详解

功能更强的可视化内存监控工具

<https://visualvm.github.io/download.html>

VisualVM/bin/visualvm.exe





## 第7节性能调优

知识：理论基础

工具：监控工具

数据：程序代码

经验：经验分析

### 7.1概述

钟表：

调钟摆的螺丝，往下是时间变慢，往上是变快

### 7.2案例1

场景：

绩效考核系统，会针对每个考核员工，生成一个各考核点的考核结果，形成Excel文档，供用户下载。文档中包含用户提交的考核点信息以及分析信息。Excel文档由用户请求的时候生成，下载，并保存到内存服务器一份。

环境：

2个IntelXeon E5 4620 V4 CPU

64内存

Centos

Tomcat7.0

JDK7

问题：

经常有用户反映长时间出现卡顿的现象。

处理思路：

~~优化sql~~

~~监控CPU~~

监控内存：FullGC 20-30秒

大对象，长期存活对象直接进入老年代

Excel的workbook，以及图标对象占用内存较大

堆内存设置的非常大，那么垃圾回收的实际就比较长，所以卡顿

解决方案：

部署多个web容器，每个web容器的堆内存指定为4g，前面增加Nginx，使用IPHash负载均衡

总结经验：

堆内存设置的比较大，垃圾回收时间就比较长，那么单节点就不如多节点分担内存的使用；如果是没有大对象的情况也许单节点要由于多节点。

### 7.3案例2

场景：

简单数据抓取系统，抓取网站上的一些数据，分发到其他应用。

环境：

Windows Server 2003

JDK5

2G内存

Intel Core i3

问题：

不定期堆内存溢出，把堆内存加大，也无济于事。导出堆转储快照信息，没有任何信息。内存监控，正常。

处理思路：

1. 将应用挪到8G的机器上运行；
2. Bytebuffer，NIO会申请对外内存，这里操作系统的内存也不会触发垃圾回收。

解决方案：

DirectMemory改大一些，这个问题就没有了；

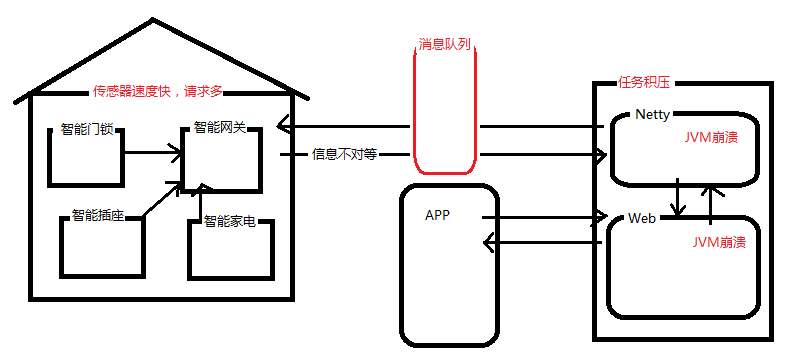
总结经验：

如果应用层面的问题解决起来找不到任何证据，可以向操作系统的层面思考。

### 7.4案例3

场景：

物联网架构



处理思路：

消息队列

总结经验：

请求与响应消息不对等，采用消息队列存储

## 第8节前半部分内容整体回顾

原理 + 工具 + 案例

原理：

1. Java虚拟机的运行时区域

线程独占区：虚拟机栈（栈帧，局部变量表），程序计数器（唯一一个没有OutOfMemory的区域），本地方法栈（本地方法的栈的在结构）

线程共享区：堆内存，方法区

1. 对象的创建和回收

垃圾对象的标记方法：

引用计数法；

可达性分析法；

垃圾收集算法：

标记--清除算法；（较慢/一般不采用）

复制算法；（适用新生代）

标记--整理算法；（适用老年代）

分代收集算法；（根据年龄划分）

垃圾收集器：

Serial（单线程/结构简单/效率高）、

Par new（可以和CMS结合/多线程）、

Parallel（服务端默认/与Par new区别针对吞吐量）、

CMS（并行垃圾收集器）、

G1（最高端的垃圾收集器）

1. 对象内存分配原则

对象首先在Eden分配；

大对象直接进入老年代；

长期存活的对象进入老年代；

空间分配担保；

逃逸分析以及栈上分配；

工具：

命令行工具：jps、jstat、jinfo、jmap、jhat、jstat

图形化工具：JConsole、Visual VM

案例：

## 第9节Class文件

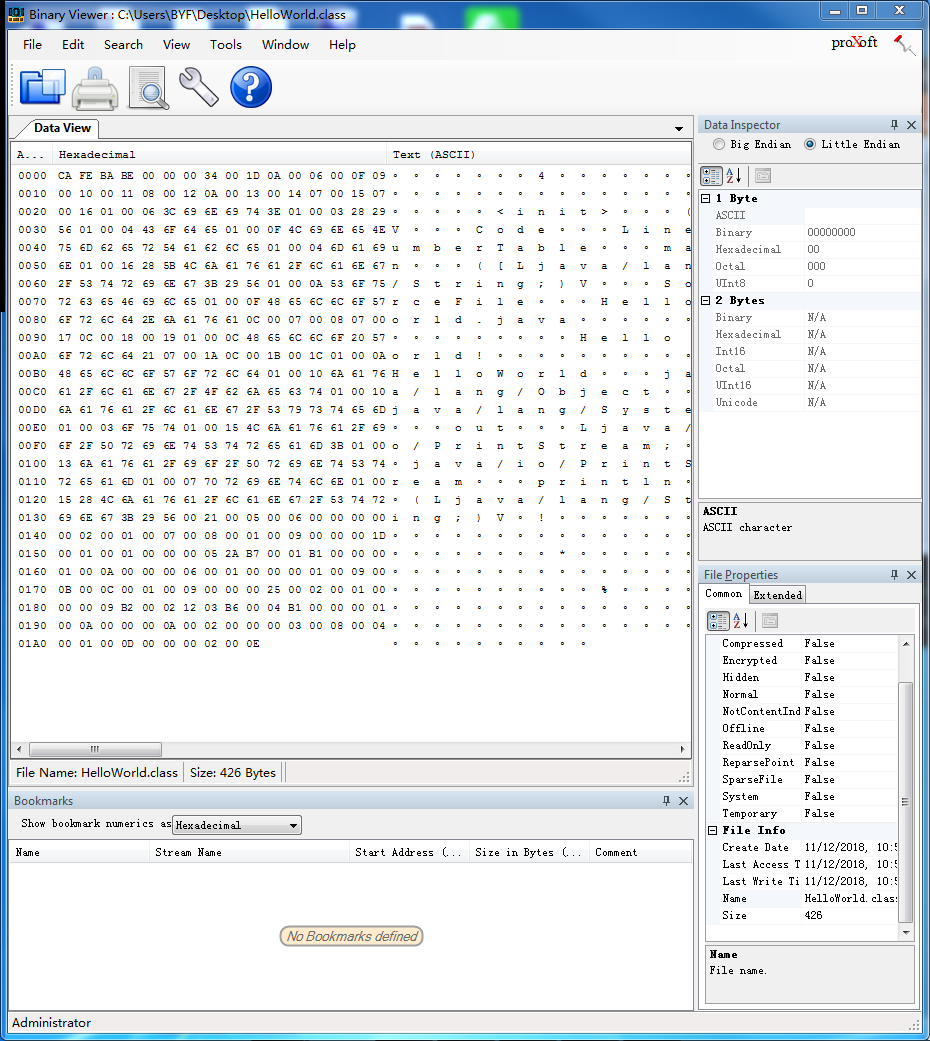
### 9.1简介和发展历史

新建HelloWorld.java文件

|  |
| --- |
| public class HelloWorld{  public static void main(String[] args){  System.out.println("Hello World!");  }  } |

打开cmd，执行javac HelloWorld.java，生成HelloWorld.class文件。

打开Binary View查看HelloWorld.class文件。



Eclipse编译Java代码，是使用javac吗？

Jdk1-9变化很大，用版本标志，但class文件的变化是非常小的，访问标志、属性表，例如枚举原来有1,2，现在多了3,4

### 9.2结构概述

Xml、JSON描述文件的格式

XML：<xml>

<xml> </xml>

</xml>

占位符，而class文件是二进制流，放弃了可读性，增加了性能。

Class的文件结构：

Class文件是一组以8位字节为基础单位的二进制流，各个数据项目严格按照顺序紧凑的排列在Class文件中，中间没有添加任何分隔符，整个Class文件中存储的内存几乎全部是程序运行的必要数据，没有空隙存在。

当遇到8位字节以上的空间数据项时，则会按照高位在前的方式分隔成若干个8位字节进行存储。

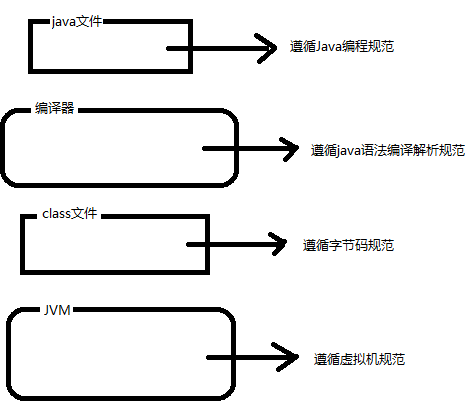
Class文件中有两种数据类型，分别是无符号数和表。



### 9.3设计理念以及意义

面向接口编程

运行在jvm之上的语言：Clojure，groovy，Jrthon，Scala



## 第10节文件结构

### 10.1魔数

### 10.2常量池

### 10.3访问标志

### 10.4类索引

### 10.5字段表集合

### 10.6方法表集合

### 10.7属性表集合

## 第11节字节码指令

### 11.1简介

### 11.2字节码与数据类型

### 11.3加载指令

### 11.4运算指令

### 11.5类型转换指令

### 11.6对象创建与访问指令

### 11.7操作树栈指令

### 11.8控制转移指令

### 11.9方法调用和返回指令

### 11.10异常处理指令

### 11.11同步指令

## 第12节类加载机制

### 12.1概述

### 12.2类加载时机

### 12.3类加载过程

### 12.4类加载过程-验证

### 12.5类加载的过程-准备

### 12.6类加载过程-解析

### 12.7类加载过程-初始化

### 12.8类加载器

### 12.9双亲委派模型

## 第13节虚拟机字节码执行引擎

### 13.1运行时栈帧结构

### 13.2局部变量表

### 13.3操作数栈

### 13.4动态连接

### 13.5方法返回地址和附加信息

### 13.6方法调用-解析调用

### 13.7方法调用-静态分派调用

### 13.8方法调用-动态分派调用

### 13.9动态类型语言支持

### 13.10字节码执行引擎小结

## 第14节总结与回顾

### 14.1happens-before简单概述

### 14.2重排序问题

### 14.3锁的内存语义

### 14.4volatile的内存语义

### 14.5final域内存语义