JVM

目录

[JVM 1](#_Toc478819222)

[一、 JVM规范 3](#_Toc478819223)

[1.1 位运算 3](#_Toc478819224)

[1.1.1 整型Int 3](#_Toc478819225)

[1.1.2 单精度Float 3](#_Toc478819226)

[二、 JVM运行机制 4](#_Toc478819227)

[2.1 JVM启动流程 4](#_Toc478819229)

[2.2 JVA基本结构 5](#_Toc478819230)

[2.2.1 PC寄存器 5](#_Toc478819231)

[2.2.2 方法区 5](#_Toc478819232)

[2.2.3 Java堆 5](#_Toc478819233)

[2.2.4 Java栈 5](#_Toc478819234)

[三、 内存模型 7](#_Toc478819235)

[3.1 内存模型特性 8](#_Toc478819237)

[3.2 常用JVM参数配置 9](#_Toc478819238)

[Tract跟踪参数 9](#_Toc478819239)

[堆的分配参数 10](#_Toc478819240)

[栈的分配参数 10](#_Toc478819241)

[四、 GC的算法与种类 10](#_Toc478819242)

[4.1 GC算法 10](#_Toc478819244)

[4.2 可触及性 10](#_Toc478819245)

[4.3 Stop-The-World 11](#_Toc478819246)

[4.4 串行搜集器 11](#_Toc478819247)

[4.5 并行收集器 12](#_Toc478819248)

[4.5.1 ParNew 12](#_Toc478819249)

[4.5.2 Parallel收集器（可自定义的、灵活） 12](#_Toc478819250)

[4.5.3 CMS收集器 13](#_Toc478819251)

[4.6 GC参数整理 14](#_Toc478819252)

[五、 类装载器 15](#_Toc478819253)

[5.1 Class装载验证流程 15](#_Toc478819255)

[5.1.1 加载 15](#_Toc478819256)

[5.1.2 链接 16](#_Toc478819257)

[5.1.3 初始化 17](#_Toc478819258)

[六、 性能分析 17](#_Toc478819259)

[6.1 Java自带性能分析的工具 17](#_Toc478819261)

[6.1.1 Jps 17](#_Toc478819262)

[6.1.2 jinfo 17](#_Toc478819263)

[6.1.3 jmap 17](#_Toc478819264)

[6.1.4 jstack 18](#_Toc478819265)

[6.1.5 JConsole \* 18](#_Toc478819266)

[6.1.6 Visual VM 18](#_Toc478819267)

[6.1.7 实战过程 18](#_Toc478819268)

[6.2 Java堆分析 20](#_Toc478819269)

[6.2.1 内存溢出OOM原因 20](#_Toc478819270)

[七、 锁 22](#_Toc478819271)

[7.1 线程安全 22](#_Toc478819273)

[7.2 对象头Mark 23](#_Toc478819274)

[7.3 偏向锁 23](#_Toc478819275)

[7.4 轻量级锁 24](#_Toc478819276)

[7.5 自旋锁 24](#_Toc478819277)

[7.6 偏向锁vs轻量级锁vs自旋锁 24](#_Toc478819278)

[7.7 Java语言层面优化锁 25](#_Toc478819279)

[7.7.1 减少锁持有时间 25](#_Toc478819280)

[7.7.2 减小锁粒度 25](#_Toc478819281)

[7.7.3 锁分离 25](#_Toc478819282)

[7.7.4 锁粗化 25](#_Toc478819283)

[7.7.5 锁消除 26](#_Toc478819284)

[7.7.6 无锁 26](#_Toc478819285)

[八、 Class文件结构 27](#_Toc478819286)

[8.1 文件结构 27](#_Toc478819288)

[九、 JVM字节码执行 28](#_Toc478819289)

[9.1 javap 28](#_Toc478819291)

[9.2 JIT及其相关参数 28](#_Toc478819292)

# JVM规范

## 位运算

### 整型Int

* 原码：第一位符号位，0为正，1为负
* 反码：符号位不动，原码取反
* 补码
  + 正数补码：和源码相同
  + 负数补码：符号位不动，反码加1

-6

原码： 10000110

反码： 11111001

补码： 11111010

* 为何使用补码
  + 可以无歧义地表示0

不使用补码，将0看为

正数：0000 0000

负数：1000 0000

则不一致

使用补码：

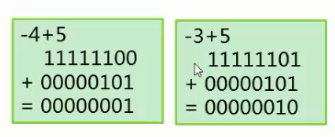
负数：1000 0000

反码：1111 1111

补码：0000 0000 = 正数

* + 正数和负数使用补码做运算相当于用加法做运算

计算时都是使用补码进行计算，



### 单精度Float

* 表示方式：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Flag1 | 指数位8  负责大数值 | | | | | | | 尾数位23  负责小数值 | | | | | | | | | | | | | | | | | | | | | | | |
| S | e | e | E | E | E | E | E | | E | M | M | M | M | M | M | M | M | M | M | M | M | M | M | M | M | M | M | M | M | M | M | | m |

当指数位：

全为0，尾数附加位为0

不全为0，则尾数附加位为1

如此，尾数位就凑足了24位

* 计算方式

S\*M\*2^(e-127)

Eg: -5的单精度表示

11000000101000000000000000000000

符号位 S：1，表示负数 -1

指数位E：10000001 ,e =129

尾数附加位：指数位不全为0，则为1

尾数M: 1+2^-2 ;(-2,尾数位由右往左数第二位)

结果：-1 \* ( 1+2^-2) \* 2^( 129 - 127) = -5

# JVM运行机制



## JVM启动流程

Java XXX

装载配置

根据配置寻找JVM.dll

根据当前路径和系统版本寻找jvm.cfg

初始化JVM

获得JNIEnv接口

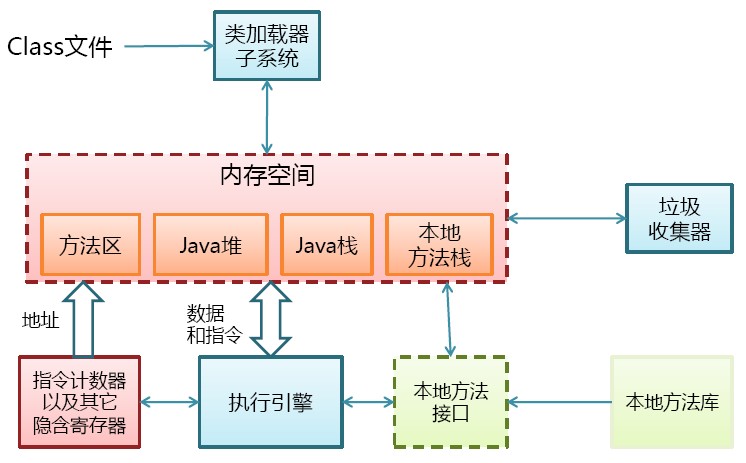
JVM.dll为JVM主要实现

JNIEnv为JVM接口，findClass等操作通过它实现

找到main方法

并运行

## JVA基本结构



方法区物理上存在于堆里，而且是在堆的持久代里面；但在逻辑上，方法区和堆是独立的

方法区（method area）只是JVM规范中定义的一个概念，用于存储类信息、常量池、静态变量、JIT编译后的代码等数据，具体放在哪里，不同的实现可以放在不同的地方。而永久代是Hotspot虚拟机特有的概念，是方法区的一种实现，别的JVM都没有这个东西

Java 6:方法区中包含的数据，除了JIT编译生成的代码存放在native memory的CodeCache区域，其他都存放在永久代；

Java 7: Symbol的存储从PermGen移动到了native memory，并且把静态变量从instanceKlass末尾（位于PermGen内）移动到了java.lang.Class对象的末尾（位于普通Java heap内）；

Java 8:永久代被彻底移除，取而代之的是另一块与堆不相连的本地内存——元空间（Metaspace）,‑XX:MaxPermSize 参数失去了意义，取而代之的是-XX:MaxMetaspaceSize

### PC寄存器

1. 每一个线程拥有一个PC寄存器
2. 在线程创建时创建
3. 指向下一条指令
4. 执行本地方法时，PC值为undefined ？

### 方法区

1. 保存装载的类信息：字段、方法信息、方法字节码
2. 通常和永久区(perm)关联在一起

### Java堆

1. 对象保存在堆中
2. 所有线程共享java堆
3. GC工作空间

eden

s0

s1

tenured

复制算法

### Java栈

1. 线程私有
2. 栈由一系列帧组成（故也叫帧栈）
3. 帧保存每个方法的局部变量表，操作数栈，常量池指针，程序计数器
4. 每一次方法调用创建一个帧，并压栈
5. 帧中有局部变量表

public **static** int runStatic

(int i,long l,float f,Object o ,byte b){

return 0;

}}

0 int int i

1 long long l

3 float float f

4 reference Object o

5 int byte b

Static

**0 reference this**

1 int char c

2 int short s

3 int boolean b

public int runInstance

(char c,short s,boolean b){

return 0;

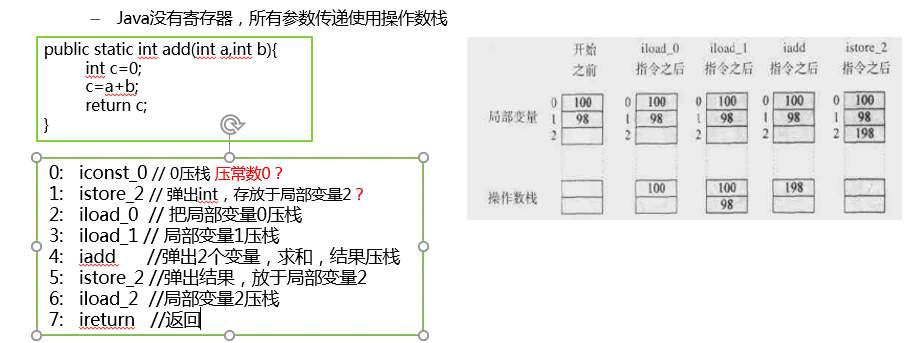
}

No Static

非static方法局部变量表的表头会保存当前引用

1. 操作数栈

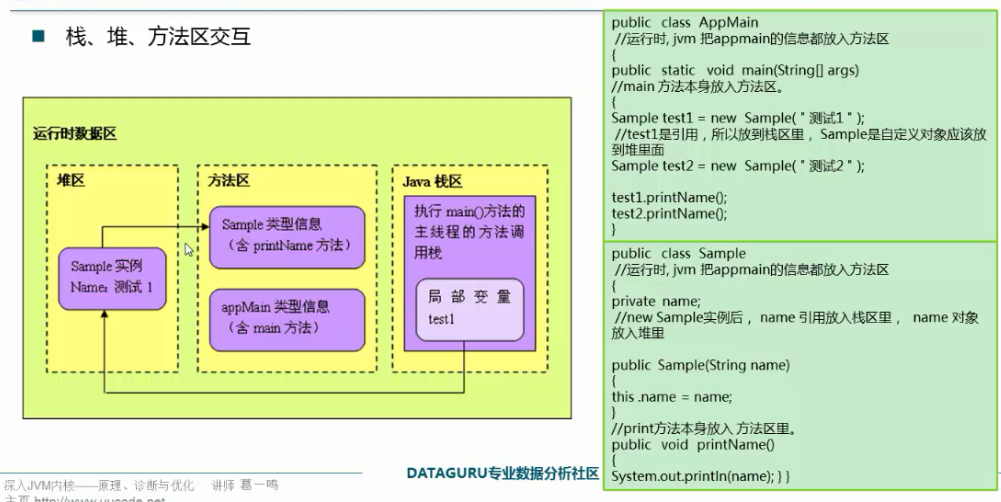
* Java没有寄存器，所有参数传递使用操作数栈



* 栈上分配空间

1. 小对象（几十bytes），在没有逃逸的情况下，可以直接分配在栈上
2. 直接分配在栈上，可以自动回收，减轻GC压力
3. 大对象或逃逸对象无法再栈上分配

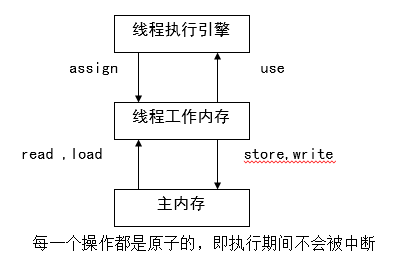
逃逸对象：栈内对象被外部对象引用，其作用范围脱离了当前方法栈

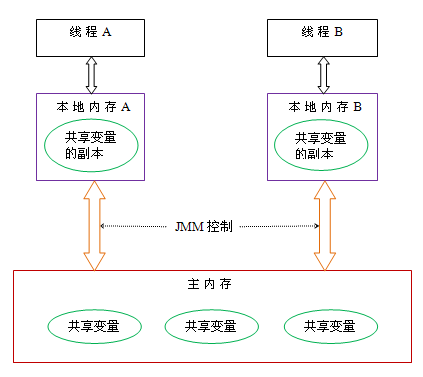


# 内存模型

每一个线程有一个工作内存和主存独立

工作内存存放主存中变量的值和拷贝





对于普通变量，一个线程中更新的值，不能马上反应在其他变量中

如果需要在其他线程中立即可见，需要使用 volatile 关键字



## 内存模型特性

* 可见性:一个线程修改了变量，其他线程可以立即知道
* 保证可见性的方法：

1. Volatile
2. Synchronized（unlock之前，写变量值回主存）
3. Final(一旦初始化完成，其他线程就可见)

* 有序性

在本线程内，操作是有序的

在线程外观察，操作时无序的（指令重排 或 主内存与线程内存同步延期）

* 指令重排

为了提高程序运行效率，调整指令执行次序

与写相邻的指令不可重排：读后写，写后读，写后写

编译器不考虑多线程间的语义

* 指令重排 – 破坏线程间的有序性

class OrderExample {

int a = 0;

boolean flag = false;

public void writer() {

a = 1;

flag = true;

}

public void reader() {

if (flag) {

int i = a +1;

……

}

}

}

线程A首先执行writer()方法

线程B线程接着执行reader()方法

线程B在int i=a+1 是**不一定**能看到a已经被赋值为1

线程A(重排)

flag=true

a=1

线程B

flag=true(此时a=0)

在writer中，两句话顺序可能打乱

* 指令重排 – 保证有序性的方法

对方法加上同步关键字synchronized

* 指令重排的基本原则
  + 程序顺序原则：一个线程内保证语义的串行性
  + volatile规则：volatile变量的写，先发生于读
  + 锁规则：解锁(unlock)必然发生在随后的加锁(lock)前
  + 传递性：A先于B，B先于C 那么A必然先于C
  + 线程的start方法先于它的每一个动作和/方法
  + 线程的所有操作先于线程的终结（Thread.join()）,最后才终结
  + 线程的中断（interrupt()）先于被中断线程的代码，中断立即停止
  + 对象的构造函数执行结束先于finalize()方法

## 常用JVM参数配置

详情见ppt

### Tract跟踪参数

-XX:+TraceClassLoading：监控类的加载

-XX:+PrintClassHistogram ： 按下Ctrl+Break后，打印类的信息

### 堆的分配参数

XX:+HeapDumpOnOutOfMemoryError：OOM时导出堆到文件

-XX:OnOutOfMemoryError: 在OOM时，执行一个脚本

官方推荐新生代占堆的3/8

幸存代占新生代的1/10

### 栈的分配参数

Xss

* 通常只有几百K
* 决定了函数调用的深度
* 每个线程都有独立的栈空间
* 局部变量、参数 分配在栈上

# GC的算法与种类

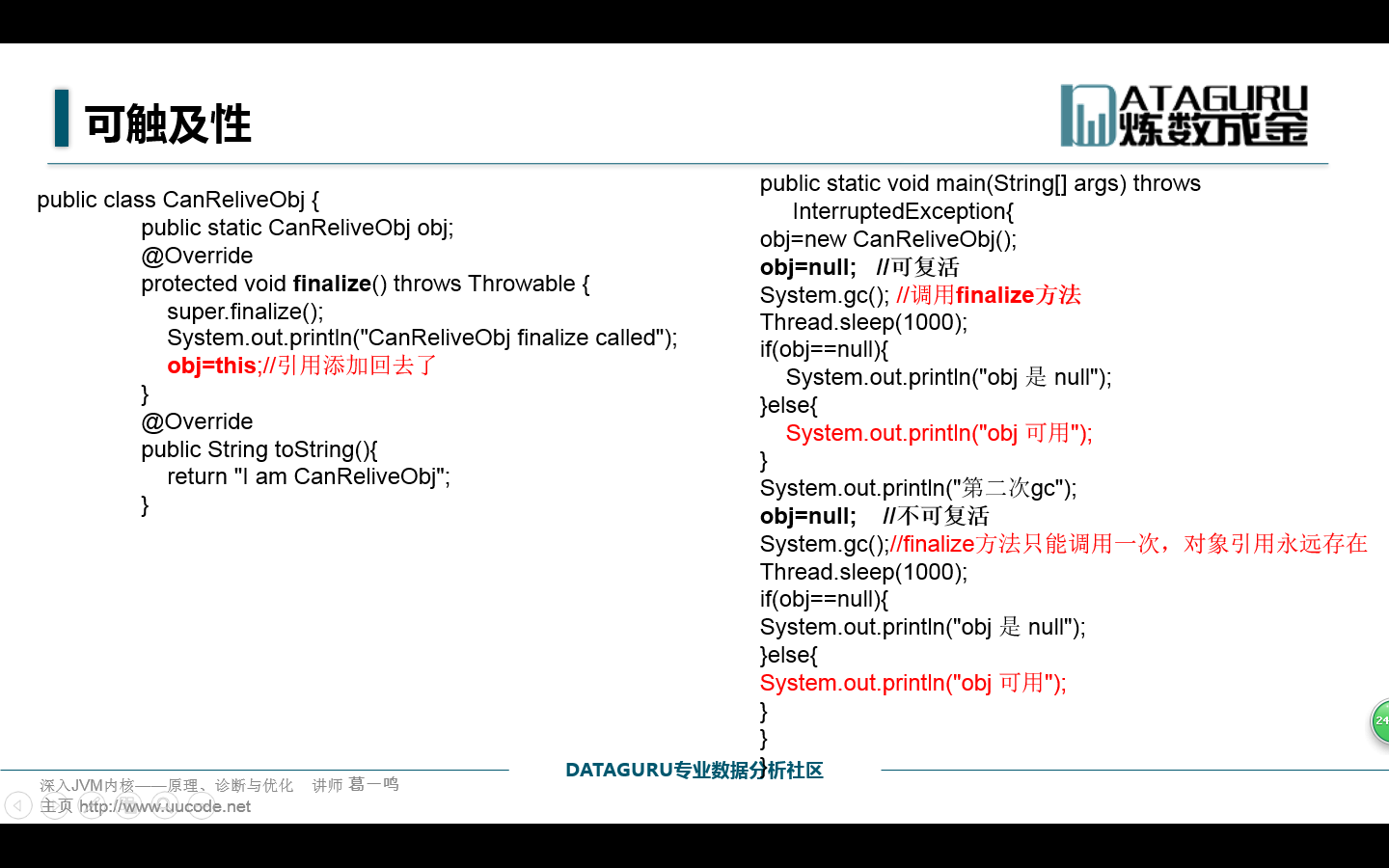


## GC算法

* 引用计数法：java中未使用
* 标记清除：老年代
* 标记压缩：老年代
* 复制算法：新生代
* 分代思想
* 依据对象的存活周期进行分类，短命对象归为新生代，长命对象归为老年代
* 根据不同代的特点，选取合适的收集算法
  + 少量对象存活，适合复制算法
  + 大量对象存活，适合标记清理或者标记压缩

## 可触及性

* 可触及的
* 从根节点可以触及到这个对象
* 根：（与方法区栈相关）
  + 栈中引用的对象
  + 方法区中静态成员或者常量引用的对象（全局对象）
  + JNI方法栈中引用对象
* 可复活的
* 一旦所有引用被释放，就是可复活状态，即不可达
* 但在finalize()中可能复活该对象
* 不可触及的
* 在finalize()后，可能会进入不可触及状态
* 不可触及的对象不可能复活
* 可以回收



避免使用finalize方法

对象中只能调用一次，操作不慎可能导致错误

优先级低，何时被调用， 不确定-何时发生GC不确定

可以使用try-catch-finally来替代

## Stop-The-World

Java中一种全局暂停的现象

全局停顿，所有Java代码停止，native代码可以执行，但不能和JVM交互

多半由于GC引起，也可以是Dump线程、死锁检查、堆Dump

## 串行搜集器

* 最古老，最稳定
* 效率高
* 可能会产生较长的停顿
* -XX:+UseSerialGC
* 新生代、老年代使用串行回收
* 新生代复制算法
* 老年代标记-压缩

应用程序线程

GC线程

应用程序暂停

应用程序线程

## 并行收集器

### ParNew

* -XX:+UseParNewGC
* 新生代并行
* 老年代串行
* Serial收集器新生代的并行版本
* 复制算法
* 多线程，需要多核支持
* -XX:ParallelGCThreads 限制线程数量

应用程序线程

GC线程

多线程并发

应用程序暂停

应用程序线程

0.834: [GC 0.834: [ParNew: 13184K->1600K(14784K), 0.0092203 secs] 13184K->1921K(63936K), 0.0093401 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

### Parallel收集器（可自定义的、灵活）

* 类似ParNew
* 新生代复制算法
* 老年代 标记-压缩
* 更加关注吞吐量
* -XX:+UseParallelGC
  + 新生代使用Parallel收集器+ 老年代串行
* -XX:+UseParallelOldGC
  + 新生代使用Parallel收集器+ 老年代并行

老年代不一样而已

1.500: [Full GC [PSYoungGen: 2682K->0K(19136K)] [ParOldGen: 28035K->30437K(43712K)] 30717K->30437K(62848K) [PSPermGen: 10943K->10928K(32768K)], 0.2902791 secs] [Times: user=1.44 sys=0.03, real=0.30 secs]

* 特殊参数
* -XX:MaxGCPauseMills
  + 最大停顿时间，单位毫秒
  + GC尽力保证回收时间不超过设定值
* -XX:GCTimeRatio
  + 0-100的取值范围
  + 垃圾收集时间占总时间的比
  + 默认99，即最大允许1%时间做GC
* 这两个参数是矛盾的。因为停顿时间和吞吐量不可能同时调优

### CMS收集器

* 特性

Concurrent Mark Sweep 并发标记清除（与用户线程一起执行 ）

标记-清除算法（不是标记压缩）

并发阶段会降低吞吐量（？）

只是针对老年代收集器（新生代使用ParNew）

-XX:+UseConcMarkSweepGC

* 运行过程
* 初始标记

根可以直接关联到的对象

速度快

独占CPU,全局停顿

* 并发标记（和用户线程一起）

标记的主要过程，标记全部对象

* 重新标记

重新修正标记

独占CPU,全局停顿

* 并发清除（和用户线程一起）

基于标记结果，直接清除对象

应用程序线程

初始标记

并发标记

重新标记

并发清理

并发重置

应用程序线程

CMS线程

* 优：

尽可能降低停顿，在并发标记过程中并不需要全局停顿

* 劣：

会影响系统整体吞吐量和性能

* + - 比如，在用户线程运行过程中，分一半CPU去做GC，系统性能在GC阶段，反应速度就下降一半

清理不彻底

* + - 在清理阶段，用户线程还在运行，会产生新的垃圾，无法清理

因为和用户线程一起运行，不能在空间快满时再清理

* + - -XX:CMSInitiatingOccupancyFraction设置触发GC的阈值
    - 如果不幸内存预留空间不够，就会引起concurrent mode failure,此时应该使用串行收集器作为后备，由于空间不足，此时一般停顿时间较长
* 碎片清理问题

CMS使用的是标记-清除算法，在清除后堆内存有效对象地址不连续，有内存碎片存在，故可设置内存压缩，整理内存碎片

即CMS为了性能考虑在老年代使用标记-清除算法，但仍可以设置使用标记-压缩算法

* -XX:+ UseCMSCompactAtFullCollection Full GC后，进行一次整理
  + 整理过程是独占的，会引起停顿时间变长
* -XX:+CMSFullGCsBeforeCompaction
  + 设置进行几次Full GC后，进行一次碎片整理
* -XX:ParallelCMSThreads
  + 设定CMS的线程数量，一般大约设成cpu核数

## GC参数整理

### 内存分配

|  |  |  |
| --- | --- | --- |
| 参数名称 | 含义 | 备注 |
| -Xms | 初始堆大小 | 默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制 |
| -Xmx | 最大堆大小 | 默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制 |
| -Xmn | 年轻代大小 | eden+ 2 survivor space  增大年轻代后,将会减小年老代大小,Sun官方推荐配置为整个堆的3/8 |
| -XX:PermSize | 设置持久代(perm gen)初始值 | 持久代是方法区的一种实现 |
| -XX:MaxPermSize | 设置持久代最大值 |  |
| -Xss | 每个线程的栈大小 | JDK5.0以后每个线程堆栈大小为1M  栈越大,线程越少,栈深度越深 |
| -XX:NewRatio | 年轻代(包括Eden和两个Survivor区)与年老代的比值(除去持久代) | -XX:NewRatio=4表示年轻代与年老代所占比值为1:4,年轻代占整个堆栈的1/5  Xms=Xmx并且设置了Xmn的情况下，该参数不需要进行设置。 |
| -XX:SurvivorRatio | Eden区与Survivor区的大小比值 | 设置为8,则两个Survivor区与一个Eden区的比值为2:8,一个Survivor区占整个年轻代的1/10 |
| -XX:MaxTenuringThreshold | 垃圾最大年龄 | 该参数只有在串行GC时才有效 |
| -XX:PretenureSizeThreshold | 对象超过多大是直接在旧生代分配 | 单位字节 新生代采用Parallel Scavenge GC时无效  另一种直接在旧生代分配的情况是大的数组对象,且数组中无外部引用对象. |

### 并行收集器相关参数

|  |  |  |
| --- | --- | --- |
|  |  |  |
| -XX:+UseParallelGC | 新生代使用Parallel收集器+ 老年代串行 |  |
| -XX:+UseParNewGC | 在新生代使用并行收集器 |  |
| -XX:ParallelGCThreads | 并行收集器的线程数 | 此值最好配置与处理器数目相等 也适用于CMS |
| -XX:+UseParallelOldGC | 新生代使用Parallel收集器+ 老年代并行 |  |
| -XX:MaxGCPauseMillis | 每次年轻代垃圾回收的最长时间(最大暂停时间) | 如果无法满足此时间,JVM会自动调整年轻代大小,以满足此值 |
| -XX:+UseAdaptiveSizePolicy | 自动选择年轻代区大小和相应的Survivor区比例 | 设置此选项后,并行收集器会自动选择年轻代区大小和相应的Survivor区比例,以达到目标系统规定的最低相应时间或者收集频率等,此值建议使用并行收集器时,一直打开. |

### CMS相关参数

|  |  |  |
| --- | --- | --- |
|  |  |  |
| -XX:+UseConcMarkSweepGC | 使用CMS内存收集 | 新生代使用并行收集器ParNew，老年代使用CMS+串行收集器 |
| -XX:CMSFullGCsBeforeCompaction | 多少次后进行内存压缩 | 由于并发收集器不对内存空间进行压缩,整理,所以运行一段时间以后会产生"碎片",使得运行效率降低.此值设置运行多少次GC以后对内存空间进行压缩,整理 |
| -XX+UseCMSCompactAtFullCollection | 在FULL GC的时候， 对年老代的压缩 | CMS是不会移动内存的， 因此， 这个非常容易产生碎片， 导致内存不够用， 因此， 内存的压缩这个时候就会被启用。 增加这个参数是个好习惯。  可能会影响性能,但是可以消除碎片 |
| -XX:CMSInitiatingPermOccupancyFraction |  | 当永久区占用率达到这一百分比时，启动CMS回收 |

### 辅助信息

|  |  |
| --- | --- |
|  |  |
| -XX:+PrintGC |  |
| -XX:+PrintGCDetails |  |
| -XX:+PrintGCTimeStamps |  |
| -XX:+PrintGCApplicationStoppedTime | 打印垃圾回收期间程序暂停的时间.可与上面混合使用 |
| -XX:+PrintHeapAtGC | 打印GC前后的详细堆栈信息 |
| -Xloggc:filename | 把相关日志信息记录到文件以便分析 |
| -XX:+HeapDumpOnOutOfMemoryError |  |
| -XX:HeapDumpPath |  |
| -XX:+PrintCommandLineFlags | 打印出已经被设置过的详细的 XX 参数的名称和值 |

# 类装载器



## Class装载验证流程

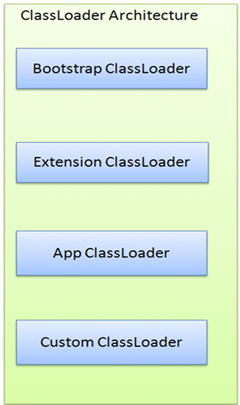
### 加载

转为方法区数据结构

在Java堆中生成对应的java.lang.Class对象

* 类装载器ClassLoader
* ClassLoader是一个抽象类
* ClassLoader的实例将读入Java字节码将类装载到JVM中
* ClassLoader可以定制，满足不同的字节码流获取方式（比如网络）

Jdk默认类加载过程(tomcat和OSGi有做更改)



rt.jar /-Xbootclasspath

%JAVA\_HOME%/lib/ext/\*.jar

Classpath下

完全自定义路径

Example：类从上往下加载

在工程目录中添加A.java,自动编译生成A.class

又指定根加载目录path，-Xbootclasspath/a:path，重新放一个同名A.class

此时会加载指定根加载目录下的class文件

注意：以上是jdk默认的类加载模式，但tomcat和OSGi有自己的加载方式

Tomcat：Tomcat的WebappClassLoader 就会先加载自己的Class，找不到再委托parent

OSGi的ClassLoader形成网状结构，根据需要自由加载Class

### 链接

* 验证

目的：保证Class流的格式是正确的

* + 文件格式的验证
    - 是否以0xCAFEBABE开头
    - 版本号是否合理：class文件由什么版本jdk编译生成，与执行class的jdk是否兼容
  + 元数据验证（基本信息验证）
    - 是否有父类:class中指定了父类，检查父类是否存在
    - 继承了final类？
    - 非抽象类实现了所有的抽象方法
  + 字节码验证 (复杂)
    - 运行检查
    - 栈数据类型和操作码数据参数吻合
    - 跳转指令指定到合理的位置
  + 符号引用验证
    - 常量池中描述类是否存在:引用的类必须存在
    - 访问的方法或字段是否存在且有足够的权限：private…
* 准备
  + 分配内存，并为类设置初始值 （方法区中）
    - public static int v=1;
    - 在准备阶段中，v会被设置为0
    - 在初始化的<clinit>中才会被设置为1
    - 对于static final类型，在准备阶段就会被赋上正确的值—在初始化之前就赋值
    - public static final int v=1;
* 解析

符号引用替换为直接引用：即类名应用，直接替换为内存地址指针

### 初始化

* 执行类构造器<clinit>
  + static变量 赋值语句 ： 注意，static final 在准备阶段已经赋值了
  + static{}语句
* 子类的<clinit>调用前保证父类的<clinit>被调用
* <clinit>是线程安全的，即单线程执行

# 性能分析



## Java自带性能分析的工具

直接在控制台输入命令，参数具体使用可使用-help 命令

### Jps

一般是第一步，方便后续其他命令调用

列出java进程，类似于ps命令

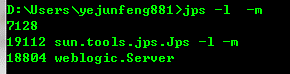
参数-q可以指定jps只输出进程ID ，不输出类的短名称

参数-m可以用于输出传递给Java进程（主函数）的参数

参数-l可以用于输出主函数的完整路径

参数-v可以显示传递给JVM的参数

Example



### Jinfo

查看进程参数

可以用来查看正在运行的Java应用程序的扩展参数，甚至支持在运行时，修改部分参数

-flag <name> 进程ID：打印指定JVM的参数值

-flag [+|-]<name> 进程ID：设置指定JVM参数的布尔值

-flag <name>=<value> 进程ID：设置指定JVM参数的值



### jmap

生成Java应用程序的堆快照和对象的统计信息



num #instances #bytes class name

----------------------------------------------

1: 370469 32727816 [C

2: 223476 26486384 <constMethodKlass>

3: 260199 20815920 java.lang.reflect.Method

…..

8067: 1 8 sun.reflect.GeneratedMethodAccessor35

Total 4431459 255496024

### jstack

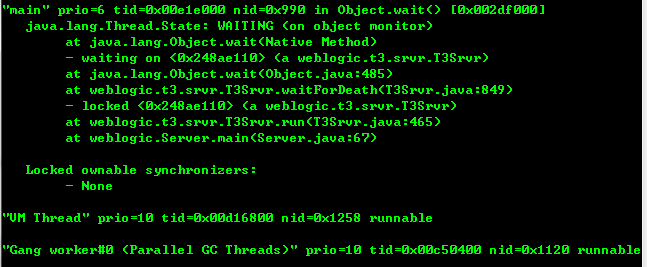
打印线程dump

-l 打印锁信息

-m 打印java和native的帧信息

-F 强制dump，当jstack没有响应时使用

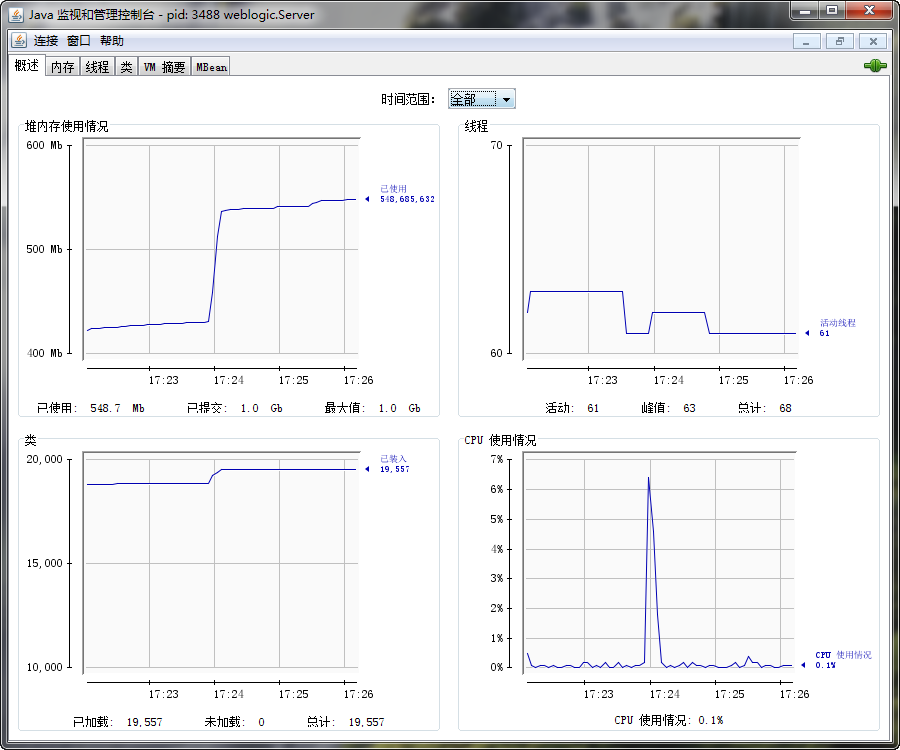
Jdk1.6版本只有 –l选项



### JConsole \*

图形化监控工具

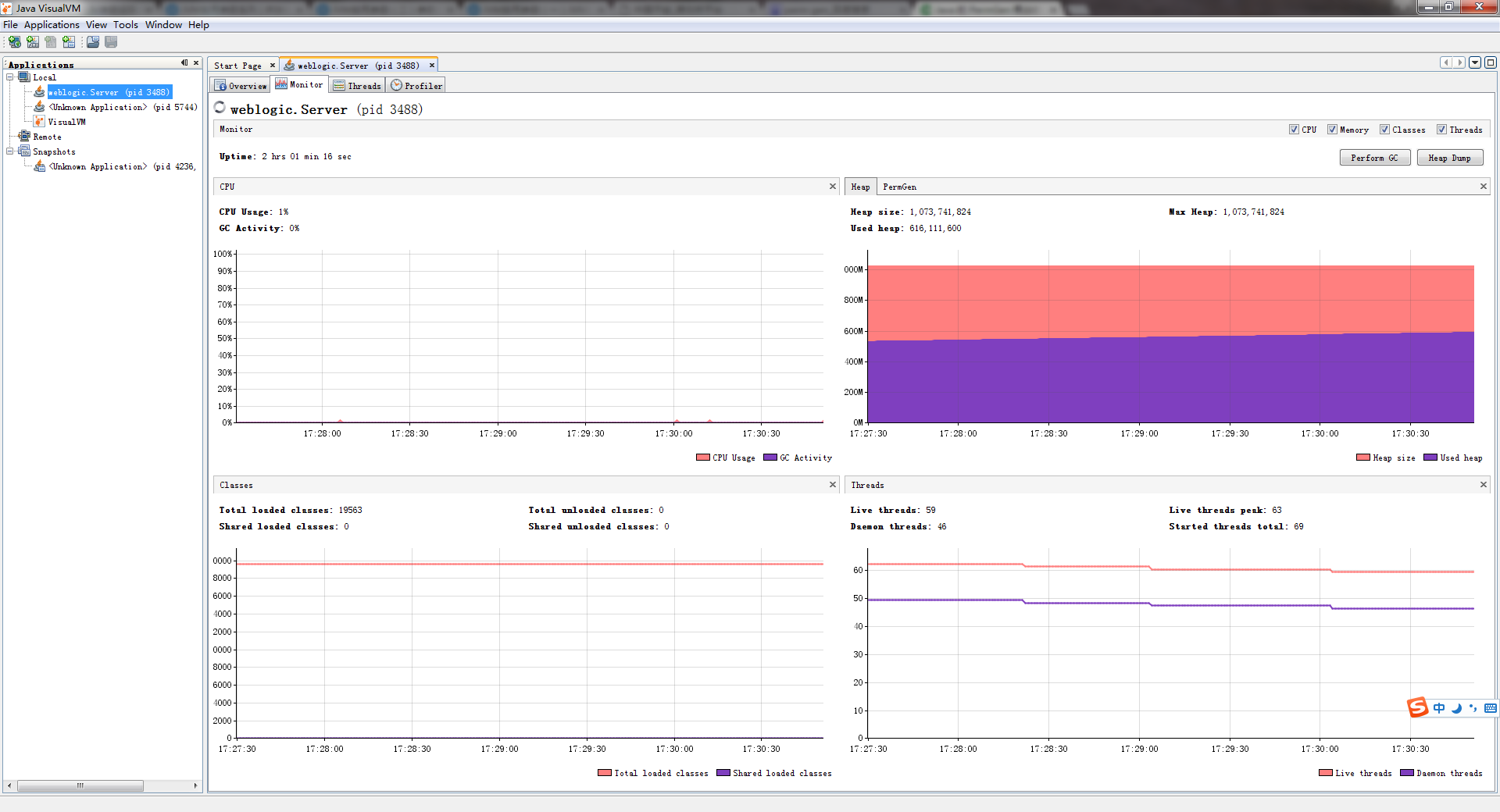
可以查看Java应用程序的运行概况，监控堆信息、永久区使用情况、类加载情况等



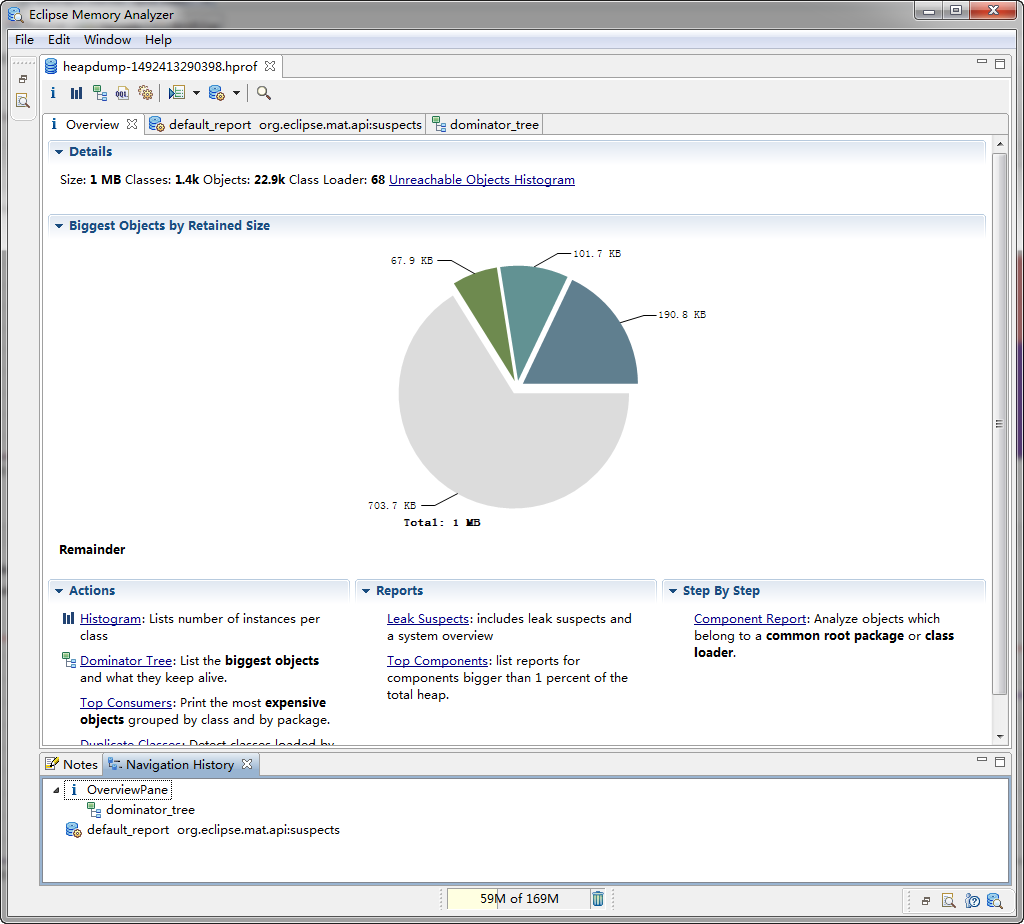
### Visual VM

Visual VM是一个功能强大的多合一故障诊断和性能监控的可视化工具





### MAT



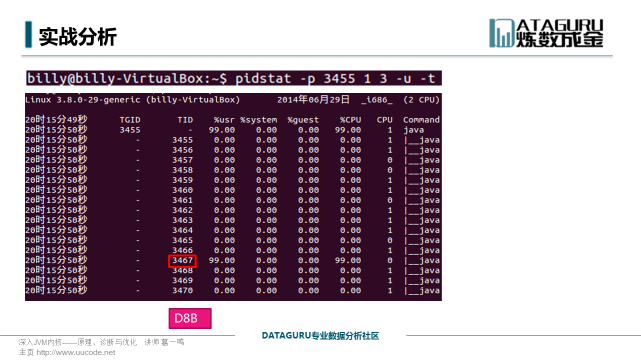
### 实战过程

Jps找到运行进程号

根据进程号查看cpu运行信息-任务管理器

根据进程号查看线程运行情况- jstack



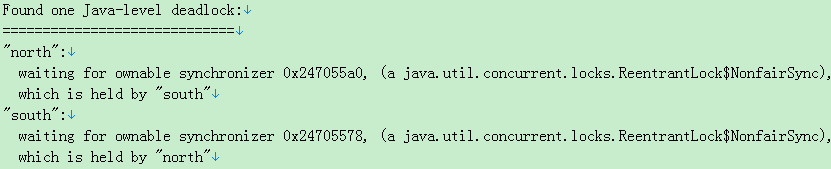
 

注意将线程号转为16进制



可以用jstack查找死锁





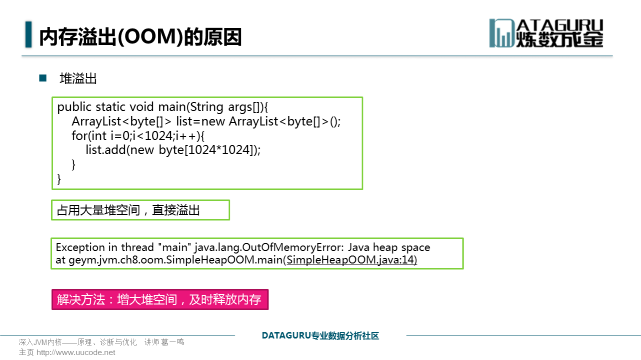
## Java堆分析

### 内存溢出OOM原因

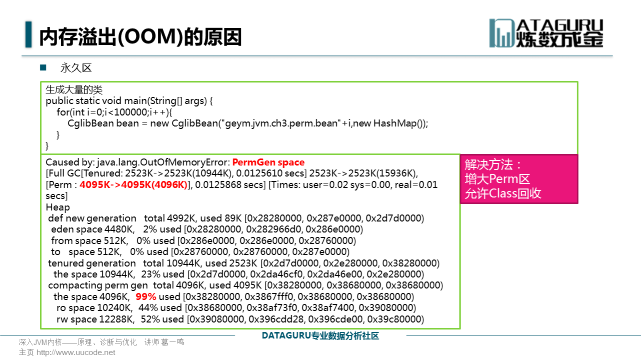
Jvm内存区间：堆、永久区、线程栈、直接内存

堆+线程栈 +直接内存<= 操作系统可分配空间

* 堆溢出



* 永久区溢出

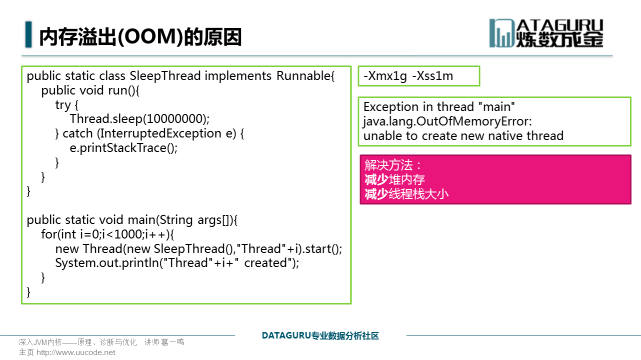


* Java栈溢出

这里的栈溢出指，在创建线程的时候，需要为线程分配栈空间，这个栈空间是向操作系统请求的，如果操作系统无法给出足够的空间，就会抛出OOM

Eg:对空间1G，每个线程栈空间1m

-Xmx1g –Xss1m



注意：堆+线程栈+直接内存 <= 操作系统可分配空间

* 直接内存溢出

ByteBuffer.allocateDirect()：申请堆外的直接内存

直接内存也可以被GC回收



更多实战请看ppt

# 锁



## 线程安全

* 实例

public static List<Integer> numberList =new ArrayList<Integer>();

public static class AddToList implements Runnable{

int startnum=0;

public AddToList(int startnumber){

startnum=startnumber;

}

@Override

public void run() {

int count=0;

while(count<1000000){

numberList.add(startnum);

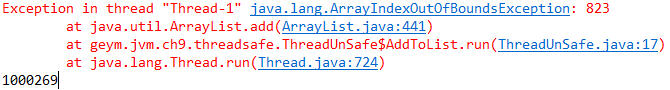
startnum+=2;

count++;

}

}

}



public static void main(String[] args) throws InterruptedException {

Thread t1=new Thread(new AddToList(0));

Thread t2=new Thread(new AddToList(1));

t1.start();

t2.start();

while(t1.isAlive() || t2.isAlive()){

Thread.sleep(1);

}

System.out.println(numberList.size());

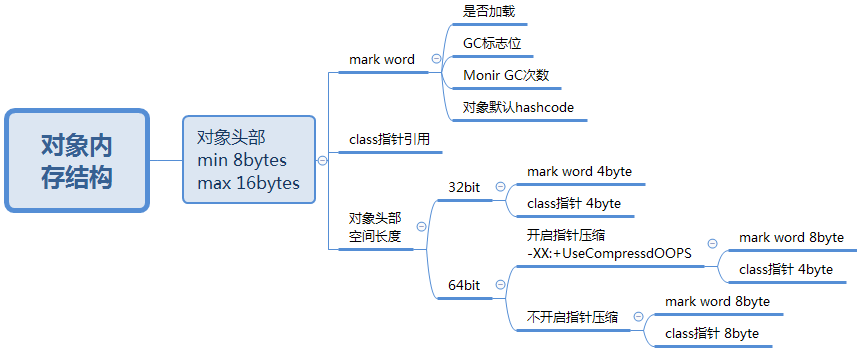
}

* 分析

ArrayList 不是线程安全的集合对象，在两个线程添加元素的过程中，当数组填满，正在自动扩展时，另一个线程却还是在添加元素，在ArrayList底层就是不可变长的数组，则抛出下表越界异常

## 对象头Mark

HotSpot虚拟机中，对象在内存中存储的布局可以分为三块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。



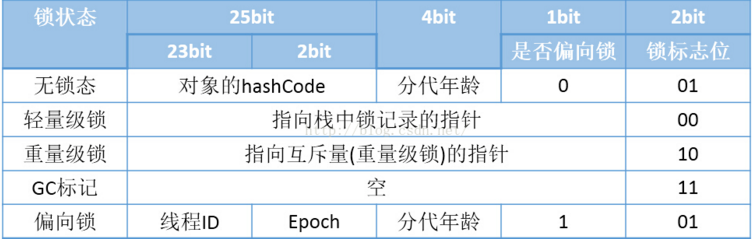


Figure 1 Mark Word 32bit

随着锁的竞争，锁可以从偏向锁升级到轻量级锁，再升级的重量级锁（但是锁的升级是单向的，也就是说只能从低到高升级，不会出现锁的降级）

## 偏向锁

大多数情况下锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低而引入了偏向锁。偏向锁只能在单线程下起作用。

偏向锁，就是在锁对象的对象头中有个ThreaddId字段，这个字段如果是空的，第一次获取锁的时候，就将自身的ThreadId写入到锁的ThreadId字段内，将锁头内的是否偏向锁的状态位置1.,这样下次获取锁的时候，直接检查ThreadId是否和自身线程Id一致，如果一致，则认为当前线程已经获取了锁，因此不需再次获取锁，略过了轻量级锁和重量级锁的加锁阶段。提高了效率。

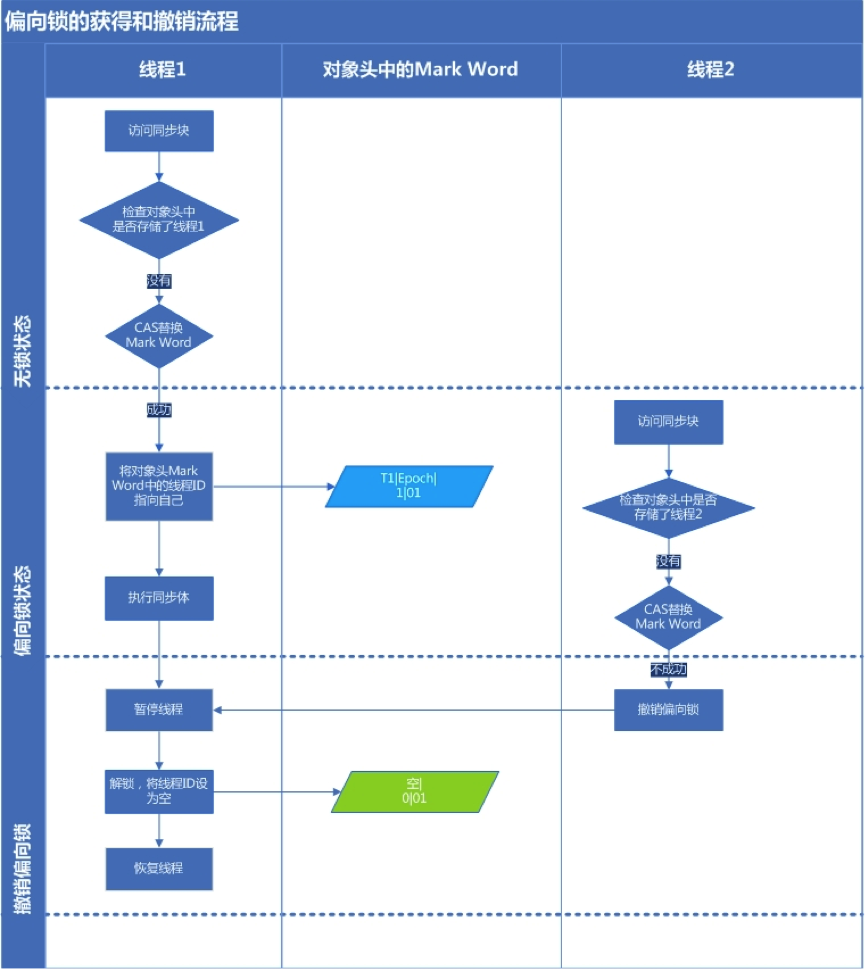
* 大部分情况是没有竞争的，所以可以通过偏向来提高性能
* 所谓的偏向，就是偏心，即锁会偏向于当前已经占有锁的线程
* 将对象头Mark的标记设置为偏向，并将线程ID写入对象头Mark
* 只要没有竞争，获得偏向锁的线程，在将来进入同步块，不需要做同步
* 当其他线程请求相同的锁时，偏向模式结束,在全局安全点（在这个时间点上没有字节码正在执行）撤销偏向锁,采用其他锁
* -XX:+UseBiasedLocking
  + 默认启用

在竞争激烈的场合，偏向锁会增加系统负担

开启偏向锁

-XX:+UseBiasedLocking -XX:BiasedLockingStartupDelay=0

系统启动后，并不会立即开启偏向锁，而是会延迟，可以设置延迟时间为0



## 轻量级锁

普通的锁处理性能不够理想，轻量级锁是一种快速的锁定方法

轻量级锁是为了在线程交替执行同步块时提高性能

* 如果对象没有被锁定

将对象头的Mark指针保存到锁对象中

将对象头设置为指向锁的指针（在线程栈空间中）

即对象和锁都互相保存引用

**轻量级锁加锁**：线程在执行同步块之前，JVM会先在当前线程的栈桢中创建用于存储锁记录的空间，并将对象头中的Mark Word复制到锁记录中，官方称为Displaced Mark Word。

然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。

**轻量级锁解锁：**轻量级解锁时，会使用原子的CAS操作来将Displaced Mark Word替换回到对象头，如果成功，则表示没有竞争发生。

如果失败，表示当前锁存在竞争，锁就会膨胀成重量级锁

* lock位于线程栈中

由上可知，判断一个线程是否持有轻量级锁，只要判断对象头的指针，是否在线程的栈空间范围内

* 特性
  + 如果轻量级锁失败，表示存在竞争，升级为重量级锁（常规锁，操作系统，进程级）
  + 在没有锁竞争的前提下，减少传统锁使用OS互斥量产生的性能损耗
  + 在竞争激烈时，轻量级锁会多做很多额外操作，导致性能下降

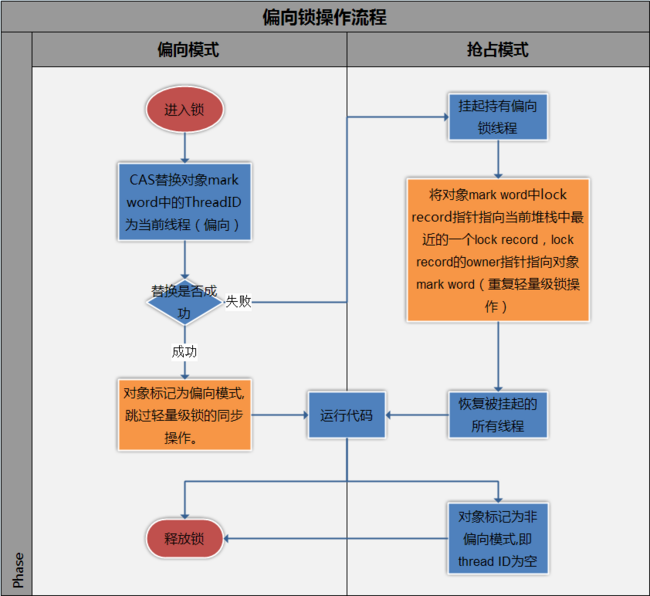
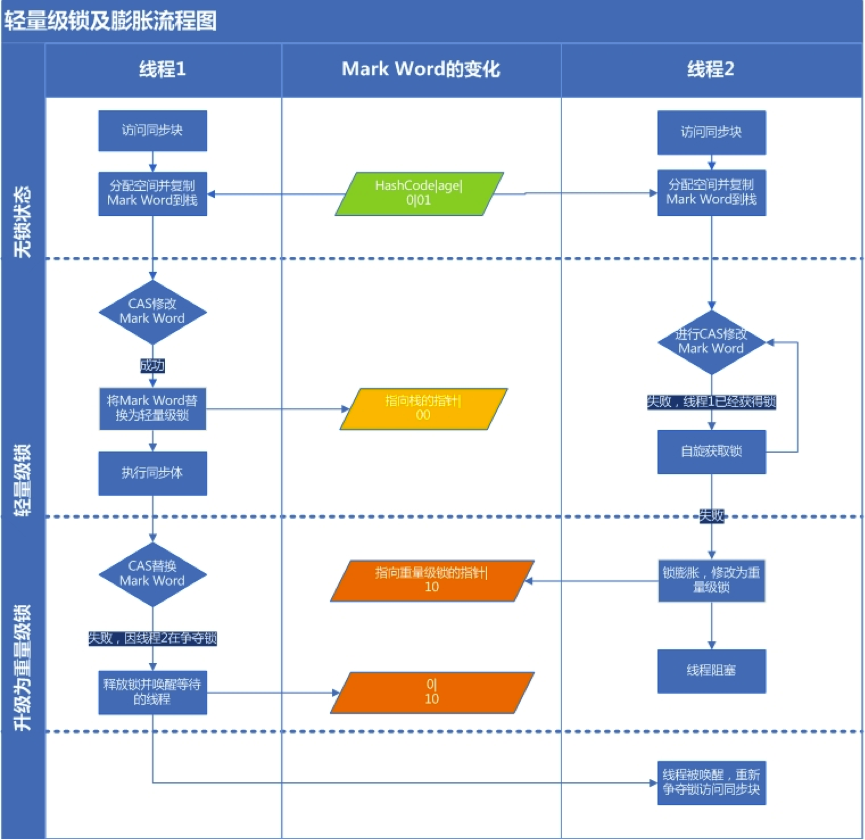


Figure 偏向锁升级进入轻量级锁

mark word中的lock record指向堆栈最近的一个线程的lock record，其实就是按照先来后到模式进行了轻量级的加锁



## 自旋锁 spin lock

* 尽量减少系统级别的线程挂起
* 当竞争存在时，如果线程可以很快获得锁，那么可以不在OS层挂起线程，让线程做几个空操作（自旋）等待获得锁
* JDK1.6中-XX:+UseSpinning开启
* JDK1.7中，去掉此参数，改为内置实现
* 如果同步块很长，自旋失败，会降低系统性能—空占线程操作，最后还是要在OS层挂起，自旋锁空耗资源
* 如果同步块很短，自旋成功，节省线程挂起切换时间，提升系统性能

当发生争用时，若Owner线程能在很短的时间内释放锁，则那些正在争用线程(未阻塞)可以稍微等一等（自旋），在Owner线程释放锁后，争用线程可能会立即得到锁，从而避免线程阻塞。

## 偏向锁vs轻量级锁vs自旋锁

* 不是Java语言层面的锁优化方法
* 内置于JVM中的获取锁的优化方法和获取锁的步骤
  + 偏向锁可用会先尝试偏向锁
  + 轻量级锁可用会先尝试轻量级锁
  + 以上都失败，尝试自旋锁
  + 再失败，尝试普通锁(重量级锁)，使用OS互斥量在操作系统层挂起

|  |  |  |  |
| --- | --- | --- | --- |
| 锁 | 优点 | 缺点 | 适用场景 |
| 偏向锁 | 加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距。 | 如果线程间存在锁竞争，会带来额外的锁撤销的消耗。 | 适用于只有一个线程访问同步块场景。 |
| 轻量级锁 | 竞争的线程不会阻塞，提高了程序的响应速度。 | 如果始终得不到锁竞争的线程使用自旋会消耗CPU。有竞争时会比重量级锁更慢 | 追求响应时间。  同步块执行速度非常快。 |
| 重量级锁 | 线程竞争不使用自旋，不会消耗CPU。 | 线程阻塞，响应时间缓慢。 | 追求吞吐量。  同步块执行速度较长。 |

偏向锁与轻量级锁理念上的区别：

轻量级锁：在无竞争的情况下使用CAS操作去消除同步使用的互斥量

偏向锁：在无竞争的情况下把整个同步都消除掉，连CAS操作都不做了

## Java语言层面优化锁

### 减少锁持有时间

同步范围减少

### 减小锁粒度

将大对象拆成小对象，增加并行度，降低锁竞争

偏向锁和轻量级锁成功率提高——粒度大，竞争激烈，偏向锁，轻量级锁失败概率就高

* + ConcurrentHashMap

若干个Segment ：Segment<K,V>[] segments

Segment中维护HashEntry<K,V>

put操作时

先定位到Segment，锁定一个Segment，执行put

在减小锁粒度后， ConcurrentHashMap允许若干个线程同时进入

### 锁分离

* 读写锁ReadWriteLock

|  |  |  |
| --- | --- | --- |
|  | 读锁 | 写锁 |
| 读锁 | 可访问 | 不可访问 |
| 写锁 | 不可访问 | 不可访问 |

* LinkedBlockingQueue

只要操作互不影响，锁就可以分离

take

A

B

C

D

put

### 锁粗化

如果对同一个锁不停的进行请求、同步和释放，其本身也会消耗系统宝贵的资源，反而不利于性能的优化

* Example1:

public void demoMethod(){

synchronized(lock){

//do sth.

}

//做其他不需要的同步的工作，但能很快执行完毕

synchronized(lock){

//do sth.

}

}

直接扩大范围

public void demoMethod(){

//整合成一次锁请求

synchronized(lock){

//do sth.

//做其他不需要的同步的工作，但能很快执行完毕

}

}

* Example2

for(int i=0;i<CIRCLE;i++){

synchronized(lock){

}

}

synchronized(lock){

for(int i=0;i<CIRCLE;i++){

}

}

### 锁消除

在即时编译器时，如果发现不可能被共享的对象，则可以消除这些对象的锁操作

锁不是由程序员引入的，JDK自带的一些库，可能内置锁

栈上对象，不会被全局访问的，没有必要加锁

* + Example

public static void main(String args[]) throws InterruptedException {

long start = System.currentTimeMillis();

for (int i = 0; i < CIRCLE; i++) {

craeteStringBuffer("JVM", "Diagnosis");

}

long bufferCost = System.currentTimeMillis() - start;

System.out.println("craeteStringBuffer: " + bufferCost + " ms");

}

public static String craeteStringBuffer(String s1, String s2) {

//StringBuffer线程安全对象，内置锁

StringBuffer sb = new StringBuffer();

sb.append(s1);

sb.append(s2);

return sb.toString();

}

-server -XX:+DoEscapeAnalysis -XX:+EliminateLocks

* + 栈上对象（方法局部变量），不会被全局访问的，没有必要加锁

### 无锁

无锁的一种实现方式

CAS(Compare And Swap)

非阻塞的同步

CAS(V,E,N)：if V==E then V=N

CAS算法的过程： CAS(V,E,N)。V表示要更新的变量，E表示预期值，N表示新值。

仅当V值等于E值时，才会将V的值设为N，如果V值和E值不同，则说明已经有其他线程做了更新，则当前线程什么都不做。

最后，CAS返回当前V的真实值。

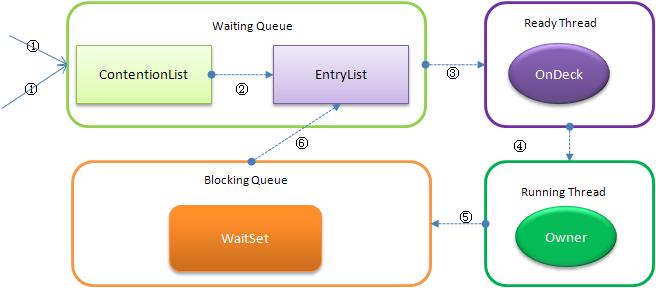
CAS操作是抱着乐观的态度进行的，它总是认为自己可以成功完成操作。当多个线程同时使用CAS操作一个变量时，只有一个会胜出，并成功更新，其余均会失败。失败的线程不会被挂起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作。基于这样的原理，CAS操作即时没有锁，也可以发现其他线程对当前线程的干扰，并进行恰当的处理。

java.util.concurrent.atomic包使用无锁实现，性能高于一般的有锁操作

## 线程状态及装换

当多个线程同时请求某个对象监视器时，对象监视器会设置几种状态用来区分请求的线程：

* Contention List：所有请求锁的线程将被首先放置到该竞争队列
* Entry List：Contention List中那些有资格成为候选人的线程被移到Entry List
* Wait Set：那些调用wait方法被阻塞的线程被放置到Wait Set
* OnDeck：任何时刻最多只能有一个线程正在竞争锁，该线程称为OnDeck
* Owner：获得锁的线程称为Owner
* !Owner：释放锁的线程



那些处于ContetionList、EntryList、WaitSet中的线程均处于阻塞状态，阻塞操作由操作系统完成（在Linxu下通过pthread\_mutex\_lock函数）。

线程被阻塞后便进入内核（Linux）调度状态，这个会导致系统在用户态与内核态之间来回切换，严重影响锁的性能

* Synchronized加锁

每一个线程在准备获取共享资源时：

1. 检查MarkWord里面是不是放的自己的ThreadId ,如果是，表示当前线程是处于 “偏向锁”
2. 如果MarkWord不是自己的ThreadId,锁升级，这时候，用CAS来执行切换，新的线程根据MarkWord里面现有的ThreadId，通知之前线程暂停，之前线程将Markword的内容置为空。
3. 两个线程都把对象的HashCode复制到自己新建的用于存储锁的记录空间，接着开始通过CAS操作，把共享对象的MarKword的内容修改为自己新建的记录空间的地址的方式竞争MarkWord,
4. 第三步中成功执行CAS的获得资源，失败的则进入自旋
5. 自旋的线程在自旋过程中，成功获得资源(即之前获的资源的线程执行完成并释放了共享资源)，则整个状态依然处于 轻量级锁的状态，如果自旋失败
6. 进入重量级锁的状态，这个时候，自旋的线程进行阻塞，等待之前线程执行完成并唤醒自己

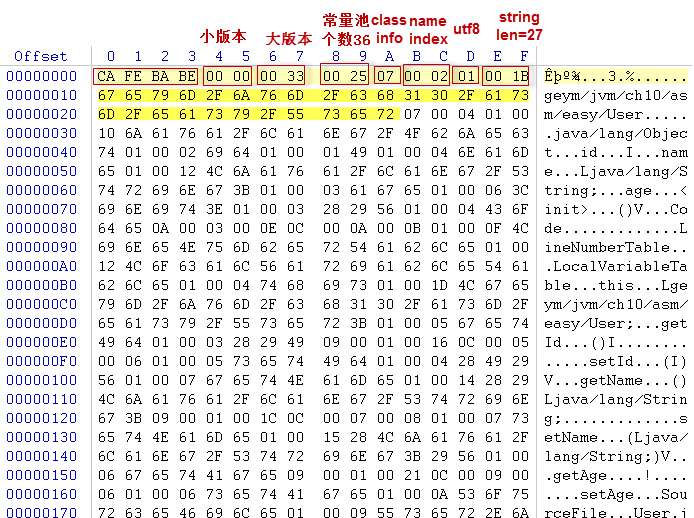
# Class文件结构



## 文件结构

U4:无符号整型，4个字节

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **名称** | **数量** | **备注** |
| u4 | magic | 1 | 0xCAFEBABE:表示java class文件类型 |
| u2 | minor\_version | 1 | Jdk编译版本 |
| u2 | major\_version | 1 | Jdk编译版本 |
| u2 | constant\_pool\_count | 1 |  |
| cp\_info | constant\_pool | constant\_pool\_count - 1 | 链式引用基本类型-被各处引用-要减1 |
| u2 | access\_flags | 1 | 访问修饰符&class type |
| u2 | this\_class | 1 | 指向常量池的class |
| u2 | super\_class | 1 | 指向常量池的class |
| u2 | interfaces\_count | 1 |  |
| u2 | interfaces | interfaces\_count | 每个接口指向常量池CONSTANT\_Class索引 |
| u2 | fields\_count | 1 |  |
| field\_info | fields | fields\_count | access\_flags,name\_index ,descriptor\_index ,attributes\_count,attribute\_info attributes[attributes\_count] |
| u2 | methods\_count | 1 |  |
| method\_info | methods | methods\_count |  |
| u2 | attribute\_count | 1 |  |
| attribute\_info | attributes | attributes\_count |  |



# JVM字节码执行



## javap

* 线程帧栈中的数据：
  + 程序计数器：每个线程都有一个，用于指向当前线程执行的指令地址
  + 局部变量表
  + 操作数栈

## JIT及其相关参数

* JIT Just-In-Time

字节码执行性能较差，所以可以对于热点代码（Hot Spot Code）编译成机器码再执行，在运行时的编译

当虚拟机发现某个方法或代码块运行特别频繁时，就会把这些代码认定为“Hot Spot Code”（热点代码），为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码）

* 辨别热点代码

方法调用计数器：方法调用次数

回边计数器：方法内循环次数，可以在栈上直接替换为机器码

* 编译设置

-XX:CompileThreshold=1000 ：执行超过一千次即为热点代码

-XX:+PrintCompilation ：打印编译为机器码的代码

-Xint：解释执行

-Xcomp：全部编译执行

-Xmixed：默认，混合

# 实践

## 修改参数

|  |
| --- |
| {Heap before GC invocations=43 (full 0):  par new generation total 471872K, used 471872K [0x04720000, 0x24720000, 0x24720000)  eden space 419456K, 100% used [0x04720000, 0x1e0c0000, 0x1e0c0000)  from space 52416K, 100% used [0x213f0000, 0x24720000, 0x24720000)  to space 52416K, 0% used [0x1e0c0000, 0x1e0c0000, 0x213f0000)  tenured generation total 524288K, used 161079K [0x24720000, 0x44720000, 0x44720000)  the space 524288K, 30% used [0x24720000, 0x2e46de38, 0x2e46e000, 0x44720000)  compacting perm gen total 262144K, used 106236K [0x44720000, 0x54720000, 0x54720000)  the space 262144K, 40% used [0x44720000, 0x4aedf190, 0x4aedf200, 0x54720000)  No shared spaces configured.  80.646: [GC 80.646: [ParNew: 471872K->52416K(471872K), 0.0847621 secs] 632951K->234152K(996160K), 0.0848136 secs] [Times: user=0.17 sys=0.03, real=0.09 secs]  Heap after GC invocations=44 (full 0):  par new generation total 471872K, used 52416K [0x04720000, 0x24720000, 0x24720000)  eden space 419456K, 0% used [0x04720000, 0x04720000, 0x1e0c0000)  from space 52416K, 100% used [0x1e0c0000, 0x213f0000, 0x213f0000)  to space 52416K, 0% used [0x213f0000, 0x213f0000, 0x24720000)  tenured generation total 524288K, used 181736K [0x24720000, 0x44720000, 0x44720000)  the space 524288K, 34% used [0x24720000, 0x2f89a0e8, 0x2f89a200, 0x44720000)  compacting perm gen total 262144K, used 106236K [0x44720000, 0x54720000, 0x54720000)  the space 262144K, 40% used [0x44720000, 0x4aedf190, 0x4aedf200, 0x54720000)  No shared spaces configured.  } |

把par new generation新生代空间增大试试看

启动时间65s

## 本机信息

java version "1.6.0\_14"

Java(TM) SE Runtime Environment (build 1.6.0\_14-b08)

Java HotSpot(TM) Client VM (build 14.0-b16, mixed mode)

JAVA Memory arguments: -Xms256m -Xmx512m -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=D:/dump\_info2 -XX:+PrintCommandLineFlags -XX:CompileThreshold=8000 -XX:PermSize=256m -XX:MaxPermSize=256m

启动时间:97s