## 001前言

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.性能调优案例实战

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

## 002教程内容

1.了解历史

2.内存结构

3.垃圾回收机制

5.性能调优案例4.性能监控工具

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

## 003环境搭建及jdk,jre,jvm的关系.txt

10.java线程高级

1.jdk包含jre包含jvm

2.jdk：java development kit

jre：java runtime environment

java virtual machine

3.RIA:(富英特网应用程序)富客户端应用程序

## 004内存溢出场景模拟.txt

1.public class Main {

public static void main(String[] args) {

List<Demo> demoList=new ArrayList<Demo>();

while(true){

demoList.add(new Demo());

}

}

}

2.-XX:+HeapDumpOnOutOfMemoryError -Xms20m -Xmx20m

3.eclipse memory analyer

<http://www.eclipse.org/mat/>

## 005jvm可视化监控工具.txt

1.jdk->bin->jconsole

2.堆内存

--老年代

--新生代

--Eden

--survivor

--

--

## 006杂谈

1.谈java

2.本套课程意义

3.适用人群

4.讲课风格

----------

java中间件:tomcat

hadoop:

lucene:工作流引擎

## 007java发展历史.txt

1.java之父:詹姆斯 高林斯

2.oak

## 008java的发展历史续.txt

## 009java技术体系.txt

1.java程序设计语言

2.各硬件平台上的虚拟机

3.class文件格式

4.java API

5.javaSE

javaME

javaEE:13种技术集合

## 010jdk1-8新特性.txt

## 011lanmbda表达式.txt

public class JMain extends JFrame{

private JButton jb;

public JMain(){

this.setBounds(200, 200, 400, 400);

this.setTitle("lanmbda测试");

jb=new JButton();

// jb.addActionListener(new ActionListener() {

//

// @Override

// public void actionPerformed(ActionEvent e) {

// System.out.println("1");

//

// }

// });

jb.addActionListener(event->System.out.println("1"));

this.add(jb);

this.setVisible(true);

this.setDefaultCloseOperation(EXIT\_ON\_CLOSE);

}

public static void main(String[] args) {

new JMain();

}

}

## 012java虚拟机-classc.txt

1.Sun HotSpot

Bea Jrockit

IBM J9

2.Sun Classic VM

--第一款商用java虚拟机，已淘汰)

--只能使用纯解释器的方式执行java代码

## 013java虚拟机-ExactVM.txt

Exact VM

--Exact Memory ManageMent准确式内存管理(可判断数据对象类型--方便回收)

--编译器和解释器混合工作以及两级即时编译器

--只在Solaris平台发布

--jdk1.2之前使用

## 014Java虚拟机-HotSpotVM.txt

1.HotSpot的历史

2.优势

--

## 015java虚拟机-kvm.txt

KVM

--Kilobyte 简单，轻量，高度可移植

--在手机平台运行

--

## 016java虚拟机-JRockit.txt

JRockit

1.BEA公司研发(08年被oracle收购)

3.专注于服务器端应用

4.优势

--垃圾收集器

--MissionControl服务套件

## 017java虚拟机-j9.txt

1.IBM

2.

## 018java虚拟机-dalvik.txt

1.不是一个java虚拟机，因为它没遵循java规范

2.与java有很大联系

3.执行Dex(dalvik Executalbe)文件，可从class文件转化而来

## 019java虚拟机-MicrosoftJVM.txt

## 020java虚拟机-高性能java虚拟机.txt

## 021java虚拟机-taobaoVM.txt

## 022java内存区域-简介.txt

线程共享区

--方法区

--java堆

线程独占区

--虚拟机栈

--本地方法栈

--程序计数器

## 023java内存区域-程序计数器.txt

程序计数器

1.程序计数器是一块较小的内存空间，它可以看做是当前线程所执行的

字节码的行号指示器

2.程序计数器处于线程独占区(每一个线程都有程序计数器)

3.如果线程执行的是java方法，这个计数器记录的是正在执行的虚拟机字节码

地址，如果执行的是native方法，这个计数器的值是undefined

4.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(开发者不需要操作程序计数器)

public class Hello{

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

int a=10;

int b=20;

syso(a+b)

}

}

goto:java 保留字(java未使用，以后可能会用)

## 024java内存区域-java虚拟机栈.txt

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

3.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

--（局部变量表的内存空间在编译期完成分配，当进入一个方法时，这个方法需要在贞中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

4.stackoverflow

## 025java内存区域-本地方法栈.txt

1.--虚拟机栈为虚拟机执行java方法服务

--本地方法栈为native方法服务

## 026java内存区域-堆内存.txt

--存放对象实例

--垃圾收集器管理的主要区域

--新生代，老年代，eden空间

--堆内存是线程共享的，但是会在堆内存中划分出许多线程私有的区域

--outofMemory

## 027java内存区域-方法区.txt

--和堆内存一样，属于线程共享区

--存储虚拟机加载的类信息(类的版本，字段，方法，接口)，常量，静态变量，即编译器编译后的代码等数据

--方法区和永久代(不等价)

--垃圾回收在方法区的行为

--异常的定义

## 028java内存区域-直接内存和运行时常量池.txt

--运行时常量池属于方法区的一块

--字符串的创建会放在运行时常量池

--new的字符串放到堆中

--intern()

--直接内存:堆外内存

## 029对象在内存中的布局-对象的创建.txt

1.对象的创建

--new 类名

--根据new的参数在常量池中定位一个类的符号引用

--如果没有找到这个符号引用，说明类还没有被加载，则进行类的加载，解析和初始化

--虚拟机为对象分配内存:指针碰撞

--将分配的内存初始化为零值

--调用对象的init方法

2.指针碰撞

## 030探究对象的结构.txt

1.对象的结构

--header(对象头)

--自身运行时数据(Mark Word)

-哈希码、GC分代年龄(分带收集算法)、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

-对象指向元数据的指针

--instanceData

--相同长度数据出现在一起

--父类定义的出现在子之前

--padding

--填充（对象的大小必须是8的镇数倍 ）

## 031深入理解对象的访问定位.txt

使用句柄

--句柄池:

直接指针

## 032垃圾回收-概述.txt

1.如何判定对象为垃圾对象

-引用计数法

-可达性分析

2.如何回收

-回收策略

-标记清除

-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

3.何时回收

## 033垃圾回收-判断对象是否存活算法-引用计数法.txt

--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，

当引用失效的时候，计数器的值-1(对象值置空)

-- -verbose:gc -XX:+PrintGCDetails

引用计数算法(Reference Counting)

很多教科书判断对象是否存活的算法是这样的：给对象中添加一个引用计数器，每当有一个地方引用它时，

计数器值加1；当引用失效时，计数器减1；任何时刻计数器都为0的对象就是不可能再被使用的。

引用计数算法的实现简单，判断效率也很高，在大部分情况下它都是一个不错的算法。

但是Java语言中没有选用引用计数算法来管理内存，其中最主要的一个原因是它很难解决对象之间相互循环引用的问题。

例如：

在testGC()方法中，对象objA和objB都有字段instance，赋值令objA.instance=objB及objB.instance=objA，

除此之外这两个对象再无任何引用，实际上这两个对象都已经不能再被访问，但是它们因为相互引用着对象方，

异常它们的引用计数都不为0，于是引用计数算法无法通知GC收集器回收它们。

/\*\*

\* 执行后，objA和objB会不会被GC呢？

\*/

public class ReferenceCountingGC {

public Object instance = null;

private static final int \_1MB = 1024 \* 1024;

/\*\*

\* 这个成员属性的唯一意义就是占点内存，以便能在GC日志中看清楚是否被回收过

\*/

private byte[] bigSize = new byte[2 \* \_1MB];

public static void main(String[] args) {

ReferenceCountingGC objA = new ReferenceCountingGC();

ReferenceCountingGC objB = new ReferenceCountingGC();

objA.instance = objB;

objB.instance = objA;

objA = null;

objB = null;

//假设在这行发生了GC，objA和ojbB是否被回收

System.gc();

}

}

https://www.cnblogs.com/suolu/p/6649417.html?utm\_source=itdadao&utm\_medium=referral

## 034垃圾回收-判断对象是否存活算法-可达性分析算法.txt

https://blog.csdn.net/ochangwen/article/details/51406779

可达性分析算法(GC Roots Analysis)：主流用这个判断

在主流的商用程序语言中(Java和C#)，都是使用可达性分析算法判断对象是否存活的。

这个算法的基本思路就是通过一系列名为"GC Roots"的对象作为起始点，从这些节点开始向下搜索，

搜索所走过的路径称为引用链(Reference Chain)，当一个对象到GC Roots没有任何引用链相连时，

则证明此对象是不可用的，下图对象object5, object6, object7虽然有互相判断，但它们到GC Roots是不可达的，

所以它们将会判定为是可回收对象。

在Java语言里，可作为GC Roots对象的包括如下几种：

a.虚拟机栈(栈桢中的本地变量表)中的引用的对象

b.方法区中的类静态属性引用的对象

c.方法区中的常量引用的对象

d.本地方法栈中JNI的引用的对象

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

## 035垃圾回收算法-标记清除算法.txt

https://blog.csdn.net/linsongbin1/article/details/51577310

## 036垃圾回收算法-复制算法.txt

https://blog.csdn.net/sinat\_36246371/article/details/53002209

-堆

--新生代

---Eden(新创建的对象)80%

---Survivor1,survivor 存活区10%

---Tenured Gen 10%

--老年代

---

内存担保

## 037垃圾回收算法-标记整理算法和分代收集算法.txt

针对老年代对象

## 038垃圾收集器-serial收集器详解.txt

最基本,发展最悠久

单线程垃圾收集器

## 039垃圾收集器-parnew收集器详解.txt

多线程

## 040垃圾收集器-parallel收集器详解

多线程

多线程收集器

吞吐量

## 041垃圾收集器-cms收集器详解



1. cms:concurrent Mark Sweep
2. 基于标记清除算法
3. 并发与并行的区别

你吃饭吃到一半，电话来了，你一直到吃完了以后才去接，这就说明你不支持并发也不支持并行。

你吃饭吃到一半，电话来了，你停了下来接了电话，接完后继续吃饭，这说明你支持并发。

你吃饭吃到一半，电话来了，你一边打电话一边吃饭，这说明你支持并行

1. 初始标记：引用计数法、可达性分析法
2. 并发标记
3. 重新标记
4. 并发清理

初始标记、从新标记这两个步骤仍然需要“stop the world”，初始标记仅仅只是标记一下GC Roots能直接关联到的对象，熟读很快，并发标记阶段就是进行GC Roots Tracing，而重新标记阶段则是为了修正并发标记期间因用户程序继续运作而导致标记产生表动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长点，但远比并发标记的时间短。

  CMS是一款优秀的收集器，主要优点：并发收集、低停顿。

缺点：

1）CMS收集器对CPU资源非常敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分线程而导致应用程序变慢，总吞吐量会降低。

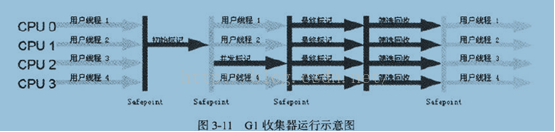
2）CMS收集器无法处理浮动垃圾，可能会出现“Concurrent Mode Failure（并发模式故障）”失败而导致Full GC产生。

浮动垃圾：由于CMS并发清理阶段用户线程还在运行着，伴随着程序运行自然就会有新的垃圾不断产生，这部分垃圾出现的标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC中再清理。这些垃圾就是“浮动垃圾”。

3）CMS是一款“标记--清除”算法实现的收集器，容易出现大量空间碎片。当空间碎片过多，将会给大对象分配带来很大的麻烦，往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次Full GC。

042垃圾收集器-g1垃圾收集器详解

## 042垃圾收集器-G1收集器



G1是一款面向服务端应用的垃圾收集器。G1具备如下特点：

1、并行于并发：G1能充分利用CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短stop-The-World停顿时间。部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让java程序继续执行。

2、分代收集：虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但是还是保留了分代的概念。它能够采用不同的方式去处理新创建的对象和已经存活了一段时间，熬过多次GC的旧对象以获取更好的收集效果。

3、空间整合：与CMS的“标记--清理”算法不同，G1从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的。

4、可预测的停顿：这是G1相对于CMS的另一个大优势，降低停顿时间是G1和ＣＭＳ共同的关注点，但Ｇ１除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内，

5、G1运作步骤：

1、初始标记；2、并发标记；3、最终标记；4、筛选回收

上面几个步骤的运作过程和CMS有很多相似之处。初始标记阶段仅仅只是标记一下GC Roots能直接关联到的对象，并且修改TAMS的值，让下一个阶段用户程序并发运行时，能在正确可用的Region中创建新对象，这一阶段需要停顿线程，但是耗时很短，并发标记阶段是从GC Root开始对堆中对象进行可达性分析，找出存活的对象，这阶段时耗时较长，但可与用户程序并发执行。而最终标记阶段则是为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程Remenbered Set Logs里面，最终标记阶段需要把Remembered Set Logs的数据合并到Remembered Set Logs里面，最终标记阶段需要把Remembered Set Logs的数据合并到Remembered Set中，这一阶段需要停顿线程，但是可并行执行。最后在筛选回收阶段首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划。

## 044内存分配-概述

<https://www.cnblogs.com/xiaoxi/p/6557473.html>

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

## 044内存分配-Eden区域

1. Eden区与Survivor区的大小比值–XX:SurvivorRatio=8
2. GC、Full GC

## 045内存分配-长期存活的对象进入老年代

1. -xx:maxTenuringThreshold

## 047内存分配-空间分配担保

1.-xx:+HandlePromotionFailure

048内存分析-逃逸分析与栈上分配

https://blog.csdn.net/blueheart20/article/details/52050545

1. 什么是栈上分配？

栈上分配主要是指在Java程序的执行过程中，在方法体中声明的变量以及创建的对象，将直接从该线程所使用的栈中分配空间。 一般而言，创建对象都是从堆中来分配的，这里是指在栈上来分配空间给新创建的对象。

2. 什么是逃逸？

逃逸是指在某个方法之内创建的对象，除了在方法体之内被引用之外，还在方法体之外被其它变量引用到；这样带来的后果是在该方法执行完毕之后，该方法中创建的对象将无法被GC回收，由于其被其它变量引用。正常的方法调用中，方法体中创建的对象将在执行完毕之后，将回收其中创建的对象；故由于无法回收，即成为逃逸。

逃逸的几种情况：

1. static V global\_v;
2. public void a\_method(){
3. V v=b\_method();
4. c\_method();
5. }
6. public V b\_method(){
7. V v=new V();
8. return v;
9. }
10. public void c\_method(){
11. global\_v=new V();
12. }

其中b\_method方法内部生成的V对象的引用被返回给a\_method方法内的变量v，c\_method方法内生成的V对象被赋给了全局变量global\_v。这两种场景都发生了（引用）逃逸。

3.  逃逸分析

   在JDK 6之后支持对象的栈上分析和逃逸分析，在JDK 7中完全支持栈上分配对象。 其是否打开逃逸分析依赖于以下JVM的设置：

**[html]** [view plain](https://blog.csdn.net/blueheart20/article/details/52050545) [copy](https://blog.csdn.net/blueheart20/article/details/52050545)

1. -XX:+DoEscapeAnalysis

4. 栈上分配与逃逸分析的关系

  进行逃逸分析之后，产生的后果是所有的对象都将由栈上分配，而非从JVM内存模型中的堆来分配。

5. 逃逸分析／栈上分配的优劣分析

     优势表现在以下两个方面：

* 消除同步。线程同步的代价是相当高的，同步的后果是降低并发性和性能。逃逸分析可以判断出某个对象是否始终只被一个线程访问，如果只被一个线程访问，那么对该对象的同步操作就可以转化成没有同步保护的操作，这样就能大大提高并发程度和性能。
* 矢量替代。逃逸分析方法如果发现对象的内存存储结构不需要连续进行的话，就可以将对象的部分甚至全部都保存在CPU寄存器内，这样能大大提高访问速度。

      劣势：  栈上分配受限于栈的空间大小，一般自我迭代类的需求以及大的对象空间需求操作，将导致栈的内存溢出；故只适用于一定范围之内的内存范围请求。

6.  代码样例分析

   源代码如下：

1. package org.eds.homework.jvm;
3. public class StackOnTest {
4. public static void alloc() {
5. byte[] b = new byte[2];
6. b[0] = 1;
7. }
9. public static void main(String[] args) {
10. long b = System.currentTimeMillis();
11. for (int i = 0; i < 100000000; i++) {
12. alloc();
13. }
14. long e = System.currentTimeMillis();
15. System.out.println(e - b);
16. }
18. }

   设置JVM的参数：

1. -server -Xmx10m -Xms10m -XX:+DoEscapeAnalysis -XX:+PrintGC

未进行逃逸分析的设置：

1. -server -Xmx10m -Xms10m -XX:-DoEscapeAnalysis -XX:+PrintGC

序的结果分析：

        在进行逃逸分析的运行结果中，只执行了9次就退出程序了。而未进行逃逸分析的结果是1360次，就是说未进行逃逸分析的代码可以执行更多的调用次数。换句话来讲，就是未进行逃逸分析的堆空间远大于进行逃逸分析后使用的栈空间，堆的空间大于栈，这就是根本原因。

7.  总结

    栈上分配可以提升代码性能，降低在多线程情况下的锁使用，但是会受限于其空间的大小。

8. 参考资料

<http://blog.csdn.net/ykdsg/article/details/6255618>

<http://www.cnblogs.com/gaiwen/archive/2013/09/24/3337835.html>

049虚拟机工具介绍

1.bin->oracle公司提供的虚拟机

050虚拟机工具-jps详解

1. Java Virtual Machine Process Status Tool
2. –l 程序全名
3. –m接收参数
4. Jps –v 虚拟机参数