笔记时间:

2016-8-24

参考书籍:《深入理解java虚拟机,JVM高级特性与最佳实践》

# JVM结构

1. java虚拟机在执行java 程序的过程中会把它所管理的内存划分为若干个不同的数据区域.这些区域都有各自的用途,以及创建和销毁的时间,有的区域随着虚拟机的进程的启动而存在,有些区域依赖用户线程的启动和结束而创建和销毁.
2. JVM数据区分为七个（前三个生命周期与线程一致）:

一)程序计数器:可以看做当前线程所执行的字节码的行号指示器.(控制执行顺序).每条线程都需要一个独自的程序计数器,各个线程之间的计数器互不影响独立存储,这种内存区域为”线程私有”的内存.

二)java虚拟机栈:java虚拟机栈也是线程私有的,.每个方法在执行的同时都会创建一个栈帧用于存储局部变量、操作数栈、动态链接、方法出口等信息。

三)本地方法栈:本地方法栈与java虚拟机栈发挥的作用是非常相似的，区别在于本地方法栈是为虚拟机中使用到的Native方法服务的，而java虚拟机栈是为执行java方法服务（即字节码）

四)java堆:java堆是java虚拟所管理的内存中最大的一块。它是被所有线程共享的一块内存区域，伴随虚拟机的启动而创建。此内存区域的唯一目的就是存放对象实例。Java堆也是垃圾回收器管理的主要区域，在这个区域垃圾回收期采用了分代收集算法，所以java堆中还可以细分为：新生代和老年代。

五)方法区:方法区与java堆是一样的，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据

六)运行时常量池:Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项就是常量池，用于存放编译期生成的各种自变量和符号引用，着部分内容将在类加载进入方法区的运行时常量池中存放。

七)直接内存:不是虚拟机运行时的数据区，也不是java虚拟机规范中定义的内存区域。但是这部分内存也被频繁使用，新加入的类，引入了一种基于通道与缓冲区的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在java堆中的DirectByteBuffer对象作为这块内存引用进行操作。

# 对象管理

1. 对象的创建：虚拟机遇到一条new指令的时候，首先会去常量池中检测是否能够定位到该类的符号引用，并且检测这个符号引用代表的类是否已经被加载、解析和初始化。对象所需要的内存大小在类加载完成后便可以完全确定，然后通过“指针碰撞”的方式在java堆中划分对应大小的内存区域。由于使用了指针碰撞的方式，所以虚拟机就必须维护一个“空闲列表”。在对象分配好内存空间后初始化都为零值（对象头除外），因为虚拟机要为分配好内存的实例进行必要的设置（如：实例的类，如何找到类的元数据信息，对象的哈希码，对象GC分代年龄等信息.这些都存在对象头中）执行new指令之后,执行init方法把对象按照程序员的意愿进行数据初始化.这样整个对象的创建就完成了。
2. 对象的内存布局：对象在内存中存储的布局可以分为三块区域：对象头、实例数据、对齐填充（没有特别含义，仅仅起着占位符作用）。
3. 对象的访问定位：java程序需要通过栈上的reference数据来操作堆上具体对象。由于reference类型在java虚拟机规范中只规定了一个指向对象的具体位置，所以对象访问方式也是取决于虚拟机实现而定的。目前主流的访问方式有使用句柄和直接指针两种。如果使用句柄，那么java堆中将会划分出一块内存出来作为句柄池。而就Sun HotSpot而言，它是使用直接指针得方式进行对象访问的。

# 内存管理

虽然java有GarbageCollection自动管理内存，但是不同的内存区域还是有可能会出现两种内存的异常情况：

1. 内存泄露：内存泄露是指分配出去的内存无法回收。
2. 内存溢出：内存溢出是指所需要分配的内存超过了系统所能分配的范围。

# 垃圾收集器与内存分配策略

1. 判断对象是否存活：在主流的商用程序语言中主流实现都是通过可达性分析（Reachability Analysis）来判定对象是否存活的。这个算法的基本思想就是通过一系列的成为“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之为因引用链（Reference Chain），当一个对象到GC Roots 没有任何引用链相连时(就是从GC Roots到这个对象是不可达的),则证明此对象是不可用的。
2. 在JAVA中，可以作为GC Roots的对象包括几种:
3. 虚拟机栈（栈帧中的本地变量表）中引用的对象。
4. 方法区中静态属性引用对象。
5. 方法区中常量引用的对象。
6. 本地方法栈中JNI（就是Native方法）引用的对象。
7. 何时杀死对象:经历两次可达性算法中不可达的对象会被置放到一个叫做“F-Queue”的队列之中。

对象回收与自救测试代码：

package study.jdk;  
*/\*\*  
 \* Created by lizhupeng on 2016/8/26.  
 \*/*public class FinalizeEscapeGC {  
 public static FinalizeEscapeGC *SAVE\_HOOK* = null;  
  
 public void isAlive(){  
 System.*out*.println("I am still alive :");  
 }  
 //在内存回收系统中。对象在被真正回收前都会执行对象的finalize方法(仅第一次)  
 //所以在finalize方法中使得对象从新与引用链建立关联可以完成对象的自救  
 protected void finalize() throws Throwable {  
 super.finalize();  
 System.*out*.println("Finalize method executed !");  
 //从新与引用链建立关联,那么再第二次标记时它  
 FinalizeEscapeGC.*SAVE\_HOOK* = this;  
 }  
 public static void main(String[] args) throws InterruptedException {  
 *SAVE\_HOOK*= new FinalizeEscapeGC();  
 //对象实现第一次自我拯救  
 *SAVE\_HOOK* = null;  
 System.*gc*();  
 Thread.*sleep*(500);  
 if(*SAVE\_HOOK* != null){  
 *SAVE\_HOOK*.isAlive();  
 }  
 else{  
 System.*out*.println("I am dead :");  
 }  
 //从结果看来对象的第二次自救失败  
 //是因为任何一个对象的finalize()方法只会被系统自动调用一次.  
 //所以当对象第二次面临回收的时候,他的finalize()方法不会再次执行  
 *SAVE\_HOOK* = null;  
 System.*gc*();  
 Thread.*sleep*(500);  
 if(*SAVE\_HOOK* != null){  
 *SAVE\_HOOK*.isAlive();  
 }  
 else{  
 System.*out*.println("I am dead :");  
 }  
 }  
}

测试结果:

Finalize method executed !

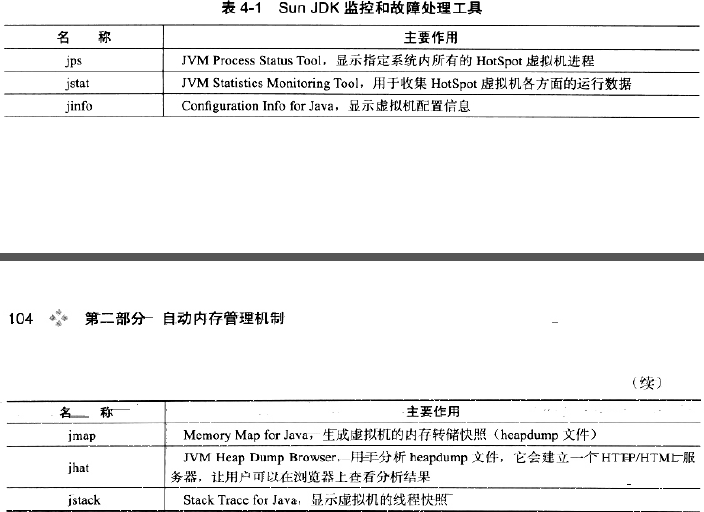
I am still alive :

I am dead :

1. 垃圾收集算法:
2. 标记-清除算法:分为两个阶段,第一个阶段对需要回收的内存进行标记,然后第二阶段对被标记的内存进行回收
3. 复制算法:将可用内存按容量划分为大小想的呢的两块,每次只使用其中的一块,当这一块内存使用完了,就将还存活着的对象复制到另外一块上面,然后再把已使用过的内存一次性清理掉.
4. 标记-整理算法:与标记清除算法一样第一步骤还是急需要进行标记,然后第二步骤不是直接对可回收对象进行清理,而是让所有存活的对象都向一端移动,然后直接清理掉端边界意外的内存.
5. 分代收集算法:根据对象存活周期的不同将内存划分为几块.

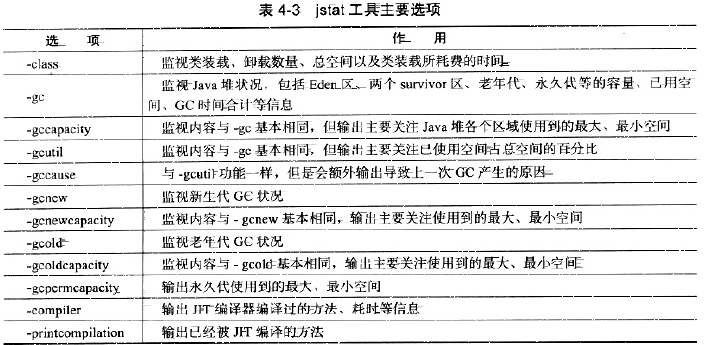
# 虚拟机性能监控与故障处理命令行工具

JDK安装后的bin目录下有几个由JDK开发团队采用Java代码来实现的监控工具,下面将一一介绍几个可以用于监控和故障处理的工具:



1. jps-虚拟机进程状况工具(JVM Process Status Tool):可以列出正在运行的虚拟机进程,并显示虚拟机执行主类
2. jstat-虚拟机统计信息监视工具:用于监视虚拟机各种运行状态信息的命令行工具.它可以显示本地或者远程虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据

命令格式： jstat –option [vmid] [sims] [count] 例子:jstat –gc 5166 250 20



显示情况解读:执行:jstat –gcutil 1400后结果如下:



结果表明:新生代区(E,表示Eden)使用了6.2%的空间,两个Survivor去(S0、S1，表示两个survivor区)，老年代(O,表示Old)和永久代(P,表示Permanent)分别使用了99.97%和14.70%的空间.程序运行以来共发生Minor GC(YGC,表示Young GC)16次,共消耗15.135秒.发生Full GC(FGC,表示Full GC)9次,共消耗39.536秒,Young GC和Full GC 两中类型的GC一共消耗54.671秒

1. jinfo-Java配置信息工具:实时地查看和调整虚拟机各项参数.使用jps命令的-V参数可以查看虚拟机启动是显示指定的参数列表.

例:jinfo 6568

1. jmap-java内存映像工具:用于生成堆转储快照(一般为heapdump或dump文件)



1. jhat-虚拟机堆转储快照分析工具
2. jstack-java堆栈跟踪工具:命令用于生成虚拟机当前时刻的线程快照。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现场时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等都是导致线程长时间停顿的常见原因。
3. HSDIS-JIT生成代码反汇编：

# JDK的可视化工具

1. JConsole-Java监视与管理控制口：可以监控虚拟机的运行数据的概览，包括“堆内存使用情况”、“线程”、“类”、“CPU使用情况”四种信息的曲线图和使用情况。还有内存监控，相当于可视化的jsata命令，用于监视受收集器管理的虚拟机内存的变化趋势。
2. VisualVM-多合一故障处理工具：VisualVm是基于NetBeans平台开发的一个插件
3. 其他第三方工具:spotlight

# JVM调优

1. 堆和栈内存调整适中
2. 尽量少的调用外部命令，例如通过java的Runtime.getRuntime().exec()方法来调用shell脚本,并且每次执行shell脚本的时候都会创建一个进程,即使外部脚本能够快速执行完毕,但是频繁调用时,创建进程的开销也非常大。
3. 控制外部Web服务的调用，并且定时跟踪外部服务的性能

# ClassLoader

1.Classloader是一个抽象类,使用实例对象来装载类,它负责将java字节码装载到jvm中(动态加载)。Java默认提供三个ClassLoader:

1)Bootstrap ClassLoader:负责加载jdk中的核心类库(由C++编写)

2)Extension ClassLoader:负责加载扩展类库

3)App ClassLoader 负责加载应用程序classpath目录的jar和class

2.ClassLoader的装载过程:

3.