Java封神之路(第二天Java内存管理)

原创

nerohouse 于 2018-05-05 15:44:01 发布 100 收藏

分类专栏： Java封神之路 文章标签： java jvm 内存管理 内存溢出 虚拟机

因在《深入理解Java虚拟机 JVM高级特性与最佳实践》中已对Java内存管理做了一些笔记，所以就直接使用笔记内容，兼顾复习。

1.1.2 Java内存管理

1.1.2.1 自动内存管理机制

1.Java内存区域与内存溢出异常

JVM运行时数据区

1)程序计数器
线程私有,记录下一次字节码解释器需要运行的位置，若方法为native则为空(undefined)，不会抛出OutOfMemoryError异常

2)虚拟机栈
方法执行时声称栈针，线程私有，存储变量表、操作栈、动态链接、方法出口
线程请求的栈深度大于JVM设置的最大栈深度则抛出StackOverFlow异常
线程若申请不到足够内存则抛出OutOfMemoryError异常

3)本地方法栈
native方法时使用，与虚拟机栈类似
4)Java堆
1.线程共享
2.存储对象的reference或句柄
3.虚拟机启动时创建
4.所有的对象和数组都要在堆中分配
5.物理上可以不连贯，逻辑上必须连贯
6.大小可以固定可以扩展，一般实现均为扩展-Xmx和-Xms控制
7.很多时候被称为GC堆
7.回收器基本使用分代算法
8.新生代、老年代、Eden空间、To Survivor空间、From Survivor空间
9.堆中没有内存完成实例分配且堆无法扩展时抛出OutOfMemoryError

5)方法区
线程共享
类信息、常量、静态变量、即时编译器编译后的代码

6)运行时常亮池
是方法区的一部分
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信息就是常量池

7)直接内存
不是Java虚拟机规范规定的内存区域
NIO类，引入基于通道Channel与缓冲区Buffer的I/O方式，可以使用Native函数库直接分配堆外内存

对象访问

Object obj = new Object();

Object obj将反应在Java栈的本地变量表中作为reference类型出现

Java堆中储存了Object类型所有的实例数据

堆中还必须存储可以找到此对象类型数据(对象类型、父类、实现的接口、方法等)的地址信息。此类信息存储在方法区中

由于reference只规定了是指向一个对象的引用，但是没有规定引用如何找到对象，所以找到对象有两种方式
1.句柄
堆中存储一个句柄池，分别存储指向对象的地址和指向类型数据的地址
2.指针
存储指向对象的地址，然后指向对象的内存中存储指向类型数据的地址

垃圾收集器与内存分配策略

引用计数算法(Reference Counting)

给每个对象生成一个计数器，每当有一个对当前对象的引用则计数器+1，每当有一个引用失效则-1，任何时候计数器为0的对象就是不可能再被引用的。
优点:效率高
缺点:无法解决相互引用的问题

根搜索算法(GC Roots Tracing)
通过一系列名为GC Roots对象为起点，开始向下搜索，搜索所走过的路径叫做引用链，当一个对象到GC Roots没有任何引用链相连时(用图论来说的话就是从这个对象到GC Roots不可达)，则证明该对象是不可用的

可被作为GC Roots的对象:
1.虚拟机栈(栈针中的本地变量表)中的引用对象
2.方法区中的静态属性引用的对象
3.方法区中的常量引用的对象
4.本地方法栈中JNI引用的对象

再谈引用

强引用(Strong Reference)
类似Object obj = new Object()这种引用，只要引用还在垃圾回收器永远都不会回收被引用的对象

软引用(Soft Reference)
还有用但并非必须的对象。在系统将要发生内存溢出异常之前，将会把这些对象列入回收范围之中并进行第二次回收

弱引用(Weak Reference)
比软引用更弱一点，能生存到下次垃圾回收发生之前

虚引用(Phantom Reference)
一个对象是否有虚引用完全不会对这个对象的生存时间构成影响。也无法通过虚引用获得一个对象实例，虚引用的目的在于在对象呗回收时收到一个系统通知

生存还是死亡

根搜索算法中无法到达GC Roots的对象也不是非死不可的，会有两次标记。
第一次标记，若对象发现没有到达GC Roots的连接链，则对其进行标记，并进行筛选，筛选的条件是是否有finalize方法，finalize方法是否有必要执行。若执行过一次则不再需要执行。
若需要执行则进入F-Queue队列，并在一个虚拟机自建的，低优先级的Finalizer中执行finalize方法且不保证该方法的执行效果，因为会有无限循环的方法存在导致内存崩溃，如果在执行finalize方法过程中对象重新与引用链上任何一个对象建立关系即可，但是只有在第一次有效，因为finalize方法只执行一次。

回收方法区

废弃常量

没有任何引用指向此常量

无用的类
Java堆中不存在该类的实例
该类的ClassLoader被回收
该类对应的java.lang.class对象没有在任何地方使用，无法在任何地方通过反射获访问该类的方法

垃圾收集算法
标记-清除算法(Mark-Sweep)
分为标记、清除两个阶段
标记所有需要清除的对象
统一回收标记的对象

缺点:标记和清除的效率不高
清除完的内存不连续

复制算法(copying)
将可使用的内存分为两块，当一块使用完之后，将所有存活的对象复制到另一块内存中去，然后统一将这一块内存清除
优点:实现简单，运行高效
缺点:内存大小为原来的一半
所有的商业虚拟机都使用该算法回收新生代
因为新生代的对象都是朝生夕死的，所以不必1:1进行分配，分配一块Eden区与两块较小的Survivor区域，当回收时将Eden区域与一块Survivor区域存活的对象全部拷贝到另一块Survivor区域，然后清空Eden区域与该Survivor区域
默认Eden和Survivor区域的比为8:1，所以每次浪费10%的内存

标记-整理算法(Mark-Compact)
根据老年代的特点，使用标记-整理算法
标记过程与标记-清除算法一样，但是后续步骤不是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以外的内存

分代收集算法(Generational Collection)
根据对象存活的周期将内存分为几块，通常分为新生代和老年代
新生代使用复制算法
老年代使用 标记-清除算法或标记整理算法

垃圾收集器
是垃圾收集算法的具体实现

Serial收集器
最基本最老的收集器
单线程收集器
垃圾收集时需要用户线程暂停等待
client模式下默认收集器

ParNew收集器
Serial的多线程版
可以与CMS收集器共用

Parallel Scavenge收集器
新生代收集器，复制算法
目标是达到可控制的吞吐量(吞吐量 CPU运行用户代码时间与消耗CPU时间的比值 即吞吐量= CPU执行用户代码时间 / (CPU执行用户代码时间 + 垃圾收集时间))
停顿时间短适合与用户交互，高吞吐量适合后台运算

Serial Old收集器
Serial收集器的老年代版本
使用标记-整理算法
单线程

Parallel Old收集器
Parallel Scavenge收集器的老年代版本
使用标记-整理算法
多线程

CMS收集器
最短收集停顿时间为目标
重视服务的响应速度
标记-清除算法
初始标记
并发标记
重新标记
并发清除
优点:并发收集、低停顿
缺点:对CPU资源敏感
无法处理浮动垃圾(由于清除过程中用户程序还在运行，无法处理新产生的垃圾，这部分垃圾叫做浮动垃圾)
收集时产生大量内存碎片

G1收集器
标记-整理算法
精确控制停顿，指定一个在M毫秒的时间片段内，消耗在垃圾回收上的时间不得超过N秒
收集器参数

内存分配与回收策略

对象优先在Eden分配

大多数情况下对象优先在Eden上分配，如果Eden上没有足够的空间那么就会触发一次Minor GC

大对象直接进入老年代

避免Eden区与Survivor区发生大量内存拷贝

长期存活的对象进入老年代

年龄(Age)计数器，经过一次Minor GC且可以被Survivor空间容纳，年龄计数器+1，增加至一个阀值(默认15)就会进入老年代

动态对象年龄判定

如果在Survivor空间中相同年龄所有对象的大小总和大于Survivor的一半则大于等于该年龄的对象进入老年代

空间分配担保

每次进入老年代时，会检测老年代进入的平均值与当前老年代剩余值进行比较，若大于则进行一次Full GC，如果小于则查看HandlePromotionFailure设置是否允许担保失败，若允许则只进行Minor GC，否则也进行Full GC，如果担保失败仍旧进行Full GC