Java 内存区域

写在前面（常见面试题）

下面是面试官可能在"Java内存区域"知识点问你的问题，快拿出小本本记下来！

基本问题：

- **介绍下Java内存区域（运行时数据区）。**
- **Java对象的创建过程（五步，建议能默写出来并且要知道每一步虚拟机做了什么）**
- **对象的访问定位的两种方式（句柄和直接指针两种方式）**

拓展问题：

- **String类和常量池**
- **8种基本类型的包装类和常量池**

1 概述（为什么要学习 Java 内存区域）

对于 Java 程序员来说，在虚拟机自动内存管理机制下，不再需要像 C/C++程序开发程序员这样为内一个 new 操作去写对应的 delete/free 操作，不容易出现内存泄漏和内存溢出问题。正是因为 Java 程序员把内存控制权利交给 Java 虚拟机，一旦出现内存泄漏和溢出方面的问题，如果不了解虚拟机是怎样使用内存的，那么排查错误将会是一个非常艰巨的任务。

2 运行时数据区域

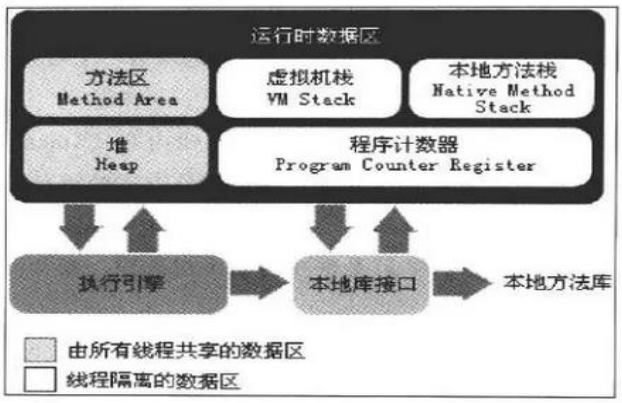

图 2-1 Java 虚拟机运行时数据区

线程私有的：程序计数器、虚拟机栈、本地方法栈

线程共享的：堆、方法区、直接内存

2.1 程序计数器

程序计数器是一块较小的内存空间，可以看作是当前线程所执行的字节码的行号指示器。**字节码解释器工作时通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等功能都需要依赖这个计数器来完。**

另外，**为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。**

2.2 Java 虚拟机栈

与程序计数器一样，Java 虚拟机栈也是线程私有的，它的生命周期和线程相同，描述的是 Java 方法执行的内存模型。

Java 内存可以粗糙的区分为堆内存（Heap）和栈内存(Stack),其中栈就是现在说的虚拟机栈，或者说是虚拟机栈中局部变量表部分。

局部变量表主要存放了编译器可知的各种数据类型（boolean、byte、char、short、int、float、long、double）、**对象引用**（reference 类型，它不同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置）。

2.3 本地方法栈

和虚拟机栈所发挥的作用非常相似，区别是： **虚拟机栈为虚拟机执行 Java 方法 （也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。**在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。

2.4 堆

Java 虚拟机所管理的内存中最大的一块，Java 堆是所有线程共享的一块内存区域，在虚拟机启动时创建。**此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。**

Java 堆是垃圾收集器管理的主要区域，因此也被称作 **GC 堆（Garbage Collected Heap）**.从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和老年代：在细致一点有：Eden 空间、From Survivor、To Survivor 空间等。**进一步划分的目的是更好地回收内存，或者更快地分配内存。**

2.5 方法区

方法区与 Java 堆一样，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。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 Non-Heap（非堆），目的应该是与 Java 堆区分开来。

HotSpot 虚拟机中方法区也常被称为 **“永久代”**，本质上两者并不等价。仅仅是因为 HotSpot 虚拟机设计团队用永久代来实现方法区而已，这样

HotSpot 虚拟机的垃圾收集器就可以像管理 Java 堆一样管理这部分内存了。但是这并不是一个好主意，因为这样更容易遇到内存溢出问题。

相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入方法区后就“永久存在”了。

2.6 运行时常量池

运行时常量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，还有常量池信息（用于存放编译期生成的各种字面量和符号引用）

既然运行时常量池时方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常。

JDK1.7 及之后版本的 JVM 已经将运行时常量池从方法区中移了出来，在 Java 堆（Heap）中开辟了一块区域存放运行时常量池。同时在 jdk 1.8 中移除整个永久代，取而代之的是一个叫元空间（Metaspace）的区域

3 Java 中垃圾回收器相比较于之前的语言优势是什么？

如果面试官让举例说明，在现网中怎样断定是内存泄漏

C++中要求程序员显示地分配内存、释放内存，如果因为某种原因使内存始终没有得到释放，如果该任务不断的重复，程序最终会耗尽内存并异常终止，无法继续运行。Java 在创建对象时会自动分配内存，当该对象的引用不存在时释放这块内存。

垃圾回收器的作用：回收释放不再使用的内存并且保证内存不被错误的回收

4 Java 垃圾回收算法?

4.1 引用计数法

给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加 1；当引用失效，计数器就减 1；任何时候计数器为 0 的对象就是不可能再被使用的。

这个方法实现简单，效率高，但是目前主流的虚拟机中并没有选择这个算法来管理内存，其最主要的原因是它很难解决对象之间相互循环引用的问题。

4.2 可达性分析算法

这个算法的基本思想就是通过一系列的称为 **“GC Roots”** 的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连的话，则证明此对象是不可用的。

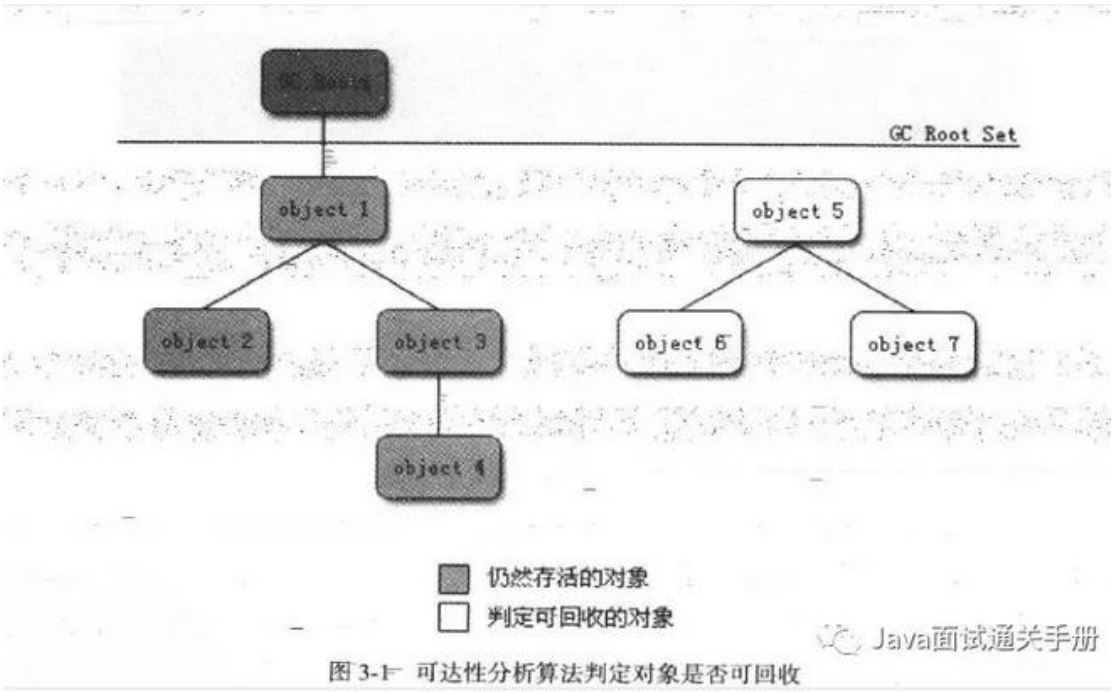

图 3-1 可达性分析算法判定对象是否可回收

4.3 标记-清除算法

算法分为“标记”和“清除”阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。它是最基础的收集算法，会带来两个明显的问题；1：效率问题和 2：空间问题（标记清除后会产生大量不连续的碎片）

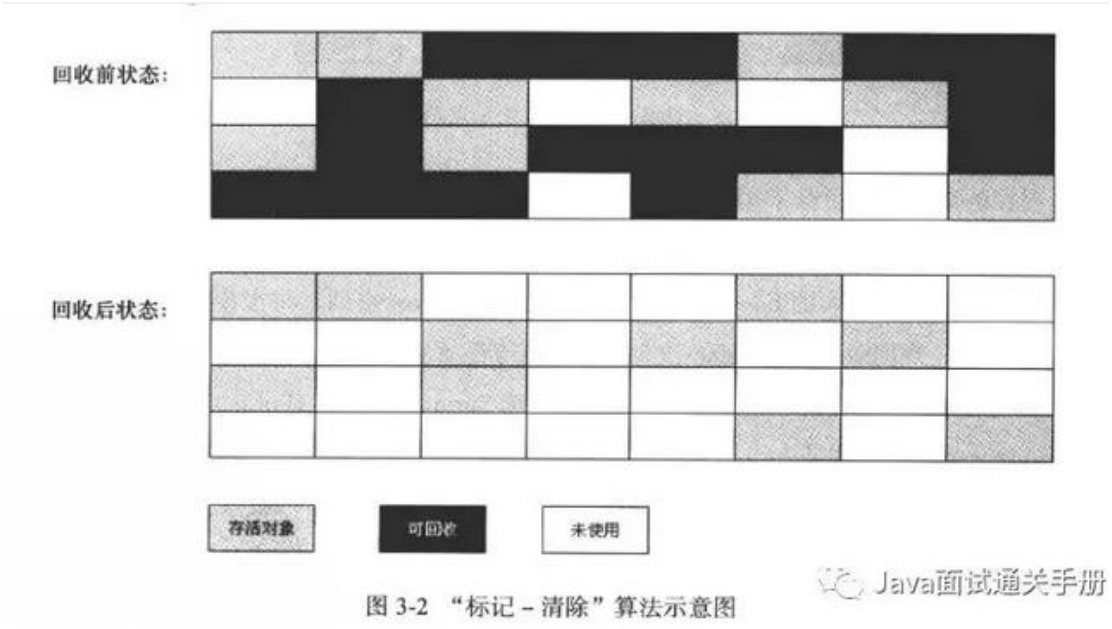

图 3-2 “标记－清除”算法示意图

4.4 复制算法

为了解决效率问题，“复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。

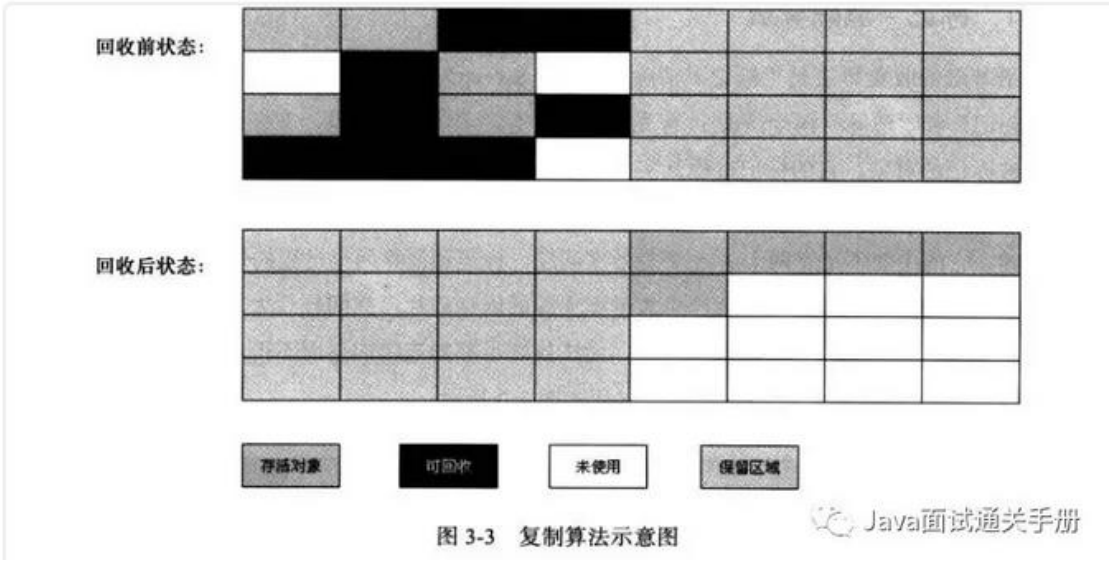

图 3-3　复制算法示意图

4.5 标记-整理算法

根据老年代的特点特出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一段移动，然后直接清理掉端边界以外的内存。

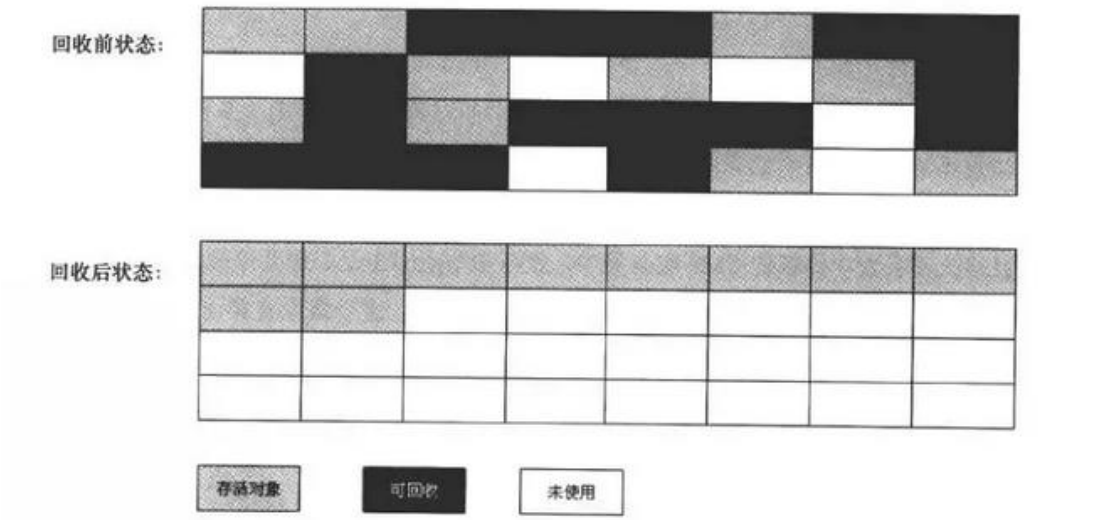

图 3-4 “标记－整理”算法示意图

3.4 分代收集算法

当前虚拟机的垃圾手机都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。一般将 java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

比如在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的所以我们可以选择“标记-清理”或“标记-整理”算法进行垃圾收集。

延伸面试问题： HotSpot 为什么要分为新生代和老年代?

根据上面的对分代收集算法的介绍回答。

5 Java 是如何进行内存管理的?

Java 的内存管理就是对象的分配和释放问题。程序员通过关键字 new 为每个对象申请内存空间（基本类型除外），所有的对象都会在堆中分配空间。对象的释放是通过过 GC 决定和执行的。这种收支两条线简化了程序员的工作，但同时加重了 JVM 的工作，这也时 Java 程序运行速度较慢的原因之一。 因为 GC 为了能够正确的释放对象，必须监控每一个对象的运行状态，包括对象的申请、引用、被引用、赋值等，GC 都需要监控

6 Java 中是否存在内存泄漏？典型的造成内存泄漏的原因?

内存泄漏是不在使用的对象或者变量仍占用着内存，由于 Java 引用垃圾回收机制，可以通过定时的垃圾回收释放内存，判一个内存空间是否符合垃圾回收的标准有两个（1）给对象赋予空值 null，之后没有再使用过（2）给对象赋予了新值，重新分配了内存空间。

内存泄漏主要有两种情况：（1）堆中申请的内存没有释放（2）对象已经不再被使用，但仍然在内存中保留着。垃圾回收器可以解决（1），但是无法解决（2）

典型的造成内存泄漏的原因：

（1） 静态集合类，例如 HashMap 和 Vector，如果这些容器为静态的，由于对象的生命周期和程序的生命周期是一样的，因此容器中的对象在程序调用结束之前不能被释放

（2） 各种练级，数据库连接等

（3） 监听器

（4） 变量的不合理作用域

（5） 单例模式可能造成内存泄漏

如何确定内存泄漏的位置：

当应用程序出现 OutOfMemoryError 错误时，（1）利用 jstat -gcutil 快速定位 GC 问题，要点：老生代内存使用率一直接近 100%，FCG 列一直增长（2）增加 Xmx 参数，观察 O 列是否还是满负荷状态，如果是满负荷，则基本断定是内存泄漏（3）使用 jmap -histo 和 jmap -histo：live 对比找出前几列就是内存泄漏的应用