拉勾教育
—互联网人实战大学—

Java底层深入

垃圾回收

常见面试题

什么是垃圾回收

- 垃圾回收中如何判断对象已死
- 可作为GC Root的对象有哪些

谈谈对分代回收的理解

- GC的类型有哪些
- Minor GC 和 Full GC 触发条件

JVM 有哪些垃圾回收算法

- JVM 各种垃圾回收算法的优缺点

什么是垃圾回收

垃圾回收（Garbage Collection， 下文简称GC）也称垃圾收集，Java程序会不定时地被唤起检查是否有不再被使用的对象，并释放它们占用的内存空间。

垃圾回收需要完成的三件事情：

- 哪些内存需要回收
- 什么时候回收
- 如何回收

垃圾回收-判断对象是否已死

- 引用记数法

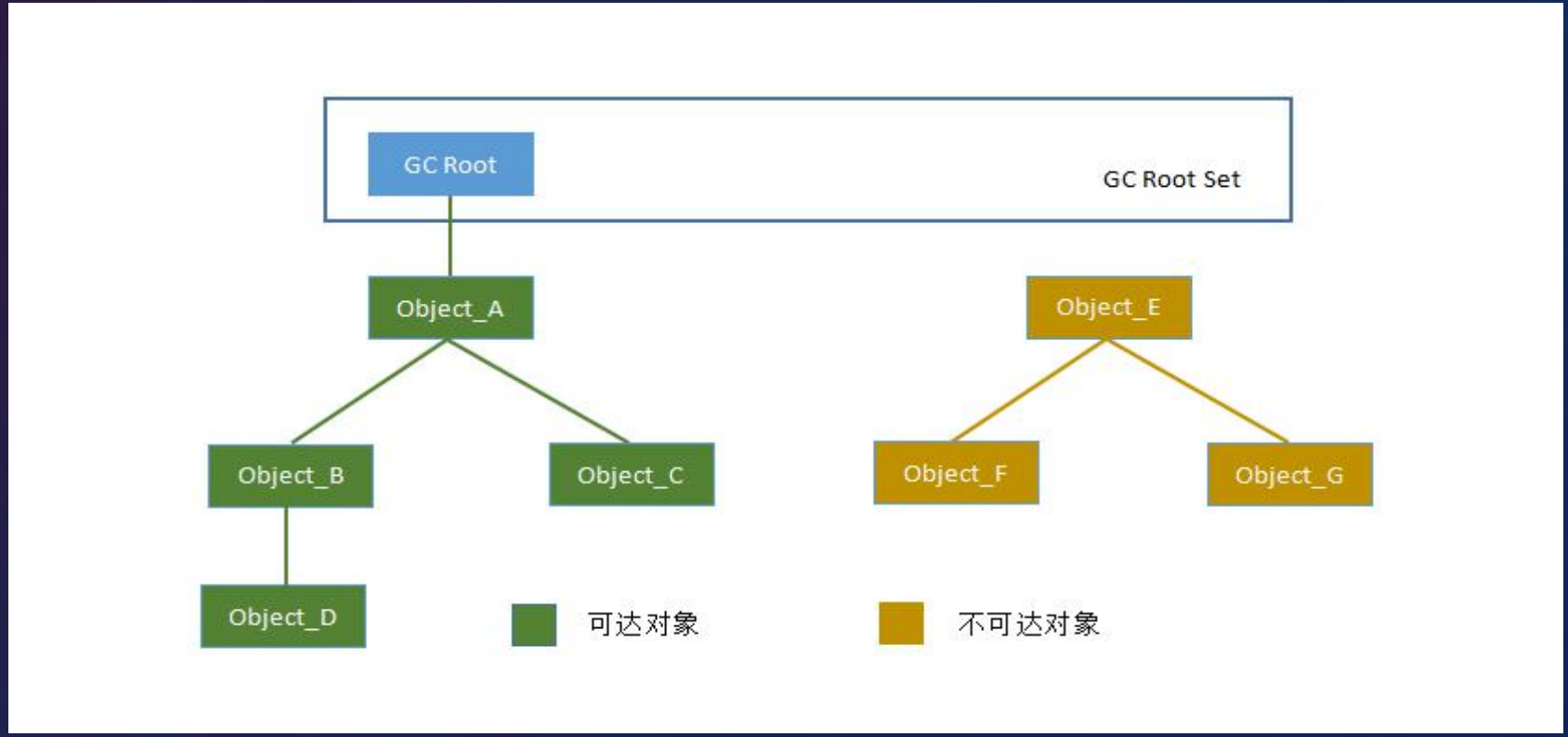
- 可达性分析算法

引用计数法

给每个创建的对象添加一个引用计数器

- 每当此对象被某个地方引用时，计数器+1
- 引用失效时，计数器-1
- 当计数值为0时表示对象已经不能被使用
- 引用计数算法大多数情况下是个比较不错的算法，简单直接，判定效率高
- 但很难解决对象直接相互循环引用的问题

可达性分析算法



在主流的商用程序语言如Java、C#等，都是通过可达性分析(Reachability Analysis)来判断对象是否存活的。

此算法的基本思路就是通过一系列的 "GC Roots" 的对象作为起始点，从起始点开始向下搜索到对象的路径。搜索所经过的路径称为引用链(Reference Chain)，当一个对象到任何GC Roots都没有引用链时，则表明对象不可达，即该对象是不可用的。

可以作为GC Root的对象

1、Java虚拟机栈(栈帧中的局部变量表)中的引用所引用的对象

2、方法区中类静态属性引用的对象

3、方法区中常量引用的对象

4、本地方法栈中JNI(Native方法)引用的对象

5、Java虚拟机内部的引用， 如基本数据类型对应的Class对象， 一些常驻的异常对象（比如NullPointExcepiton、 OutOfMemoryError）

6、所有被同步锁（synchronized关键字） 持有的对象

7、反映Java虚拟机内部情况的JMXBean、 JVMTI(JVM Tool Interface) 中注册的回调、 本地代码缓存等

分代回收

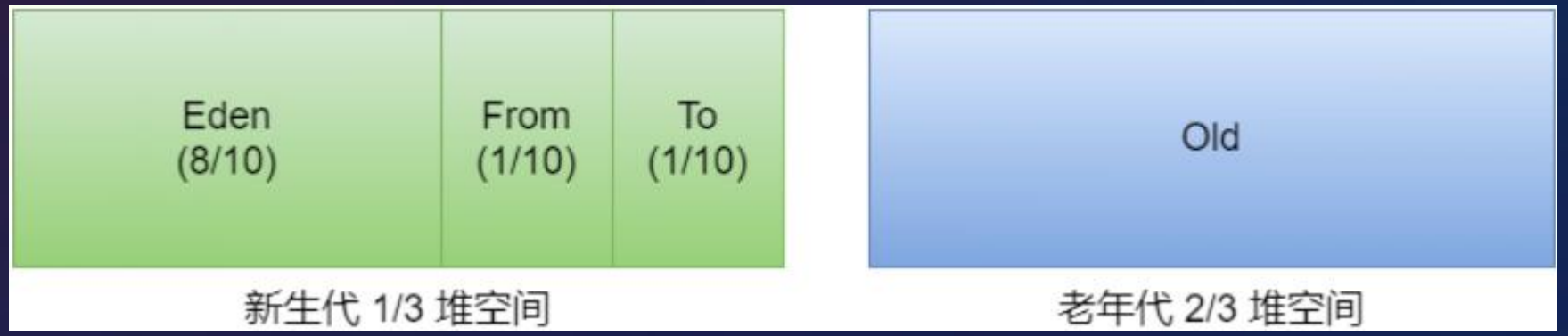
根据对象的生命周期将内存划分，然后进行分区管理。 当前商业虚拟机的垃圾收集器， 大多数都遵循了“分代收集”（Generational Collection）的理论进行设计。 分代收集名为理论， 实质是一套符合大多数程序运行实际情况的经验法则， 它建立在两个分代假说之上：

- 弱分代假说（Weak Generational Hypothesis）。绝大多数对象都是朝生夕灭的
- 强分代假说（Strong Generational Hypothesis）。熬过越多次垃圾收集过程的对象就越难以消亡

新生代 & 老年代

存储在 JVM 中的Java对象可以被划分为两类：

- 一类是生命周期较短的瞬时对象，这类对象的创建和消亡都非常迅速
- 另外一类对象的生命周期非常长，在某些极端的情况下与JVM的生命周期保持一致



- 几乎所有的Java对象都是在Eden区被new出来的
  - 有些大对象在Eden区无法存储时候，将直接进入老年代
- 绝大部分的Java对象的销毁都在新生代进行
  - 有研究表明，新生代中80%的对象都是 "朝生夕死" 的

对象分配过程

- new的对象先放伊甸园区
- 当伊甸园空间填满，又需要为新的对象分配空间时，JVM垃圾回收器将对伊甸园区进行垃圾回收（Minor GC），将伊甸园区中的不再被其他对象所引用的对象进行销毁。再加载新的对象放到伊甸园区
- 然后将伊甸园中的剩余对象移动到幸存者0区
- 如果再次触发垃圾回收，此时上次幸存下来的放到幸存者0区的，如果没有回收，就会放到幸存者1区
- 如果再次经历垃圾回收，此时会重新放回幸存者0区，接着再去幸存者1区
- 啥时候能去养老区呢？默认是15次GC，仍然存活的对象
- 当养老区内存不足时，再次触发GC：Major GC，进行老年代的内存清理
- 老年代执行了Major GC之后，发现依然无法进行对象的保存，就会产生OOM异常

对象分配

- 优先分配到Eden
- 大对象直接分配到老年代
- 尽量避免程序中出现过多的大对象
- 长期存活的对象分配到老年代
- 动态对象年龄判断【特殊情况】
  - 如果 Survivor 区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代，无须等到 MaxTenuringThreshold 中要求的年龄

JVM 垃圾回收算法分类

- 标记-清除算法

标记无用对象，然后进行清除回收

- 标记-复制算法

按照容量划分二个大小相等的内存区域，当一块用完的时候将活着的对象复制到另一块上，然后再把已使用的内存空间一次清理掉。

- 标记-整理算法/标记清除整理算法

标记无用对象，让所有存活的对象都向一端移动，然后直接清除掉端边界以外的内存。

标记-清除

标记-清除是基础的垃圾回收算法，其他垃圾收集算法都是在此算法的基础上进行改进

标记无用对象，然后进行清除回收。整个过程为两个阶段：

- 标记阶段：标记出可以回收的对象
- 清除阶段：回收被标记的对象所占用的空间

优点：实现简单，不需要对象进行移动。

缺点：标记、清除过程效率低，产生大量不连续的内存碎片，提高了垃圾回收的频率

标记-复制算法（Mark-Copy）

为了解决标记-清除算法的效率不高的问题，产生了复制(Copying)算法：

- 将内存空间划为两个相等的区域，每次只使用其中一个区域
- 垃圾收集时，遍历当前使用的区域，把存活对象复制到另外一个区域中
- 最后将当前使用的区域的可回收的对象进行回收

优点：按顺序分配内存即可，实现简单、运行高效，不用考虑内存碎片

缺点：可用的内存大小缩小为原来的一半，对象存活率高时会频繁进行复制

标记-整理算法

标记-整理算法（Mark-Compact）算法，与标记-清除算法不同的是，在标记可回收的对象后将所有存活的对象压缩到内存的一端，使它们紧凑的排列在一起，然后对端边界以外的内存进行回收。回收后，已用和未用的内存都各自一边。

优点：解决了标记-清理算法存在的内存碎片问题

缺点：需要进行局部对象移动，一定程度上降低了效率

垃圾回收算法总结

- 分代收集是目前大部分JVM的垃圾收集器采用的思想。它的核心思想是根据对象存活的生命周期将内存划分为若干个不同的区域。一般情况下将堆区划分为老年代（Tenured Generation）和新生代（Young Generation）
- 目前大部分垃圾收集器对新生代都采取复制算法。因为新生代中每次垃圾回收都要回收大部分对象，也就是说需要复制对象少
- 老年代的对象存活率会较高，这样会有较多的复制操作，导致效率变低。标记-清除算法可以应用在老年代中，但是它效率不高，在内存回收后容易产生大量内存碎片。由于老年代的特点是每次回收都只回收少量对象，一般使用的是标记-整理算法（压缩法）

GC的类型

在Java堆划分出不同的区域之后， 垃圾收集器才可以每次只回收其中某一个或者某些部分的区域 ，因次就有了Minor GC、Major GC、Full GC这样的回收类型的划分；

堆划分了区域能够针对不同的区域安排与里面存储对象存亡特征相匹配的垃圾收集算法；

为避免产生混淆，针对不同分代的类似名词在这里统一定义 ：

- 部分收集（Partial GC）：指目标不是完整收集整个Java堆的垃圾收集， 其中又分为：
  - 新生代收集（Minor GC/Young GC）：指目标只是新生代的垃圾收集
  - 老年代收集（Major GC/Old GC）：指目标只是老年代的垃圾收集，目前只有CMS收集器会有单 独收集老年代的行为
  - 混合收集（Mixed GC）：指目标是收集整个新生代以及部分老年代的垃圾收集。 目前只有G1收集器会有这种行为
- 整堆收集（Full GC）：收集整个Java堆和方法区的垃圾收集

Minor GC 和 Full GC 触发条件

Minor GC触发条件：

当Eden区满时 触发Minor GC，Minor GC会引发STW(stop the world)。

Full GC触发条件：

（1）调用System.gc时，系统建议执行Full GC，但是不必然执行

（2）老年代空间不足

（3）方法区空间不足

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存