05 大厂面试题：得心应手应对 OOM 的疑难杂症

在前面几个课时中，我们不止一次提到了堆（heap），堆是一个巨大的对象池。在这个对象池中管理着数量巨大的对象实例。

而池中对象的引用层次，有的是很深的。一个被频繁调用的接口，每秒生成对象的速度，也是非常可观的。对象之间的关系，形成了一张巨大的网。虽然 Java 一直在营造一种无限内存的氛围，但对象不能只增不减，所以需要垃圾回收。

那 JVM 是如何判断哪些对象应该被回收？哪些应该被保持呢？

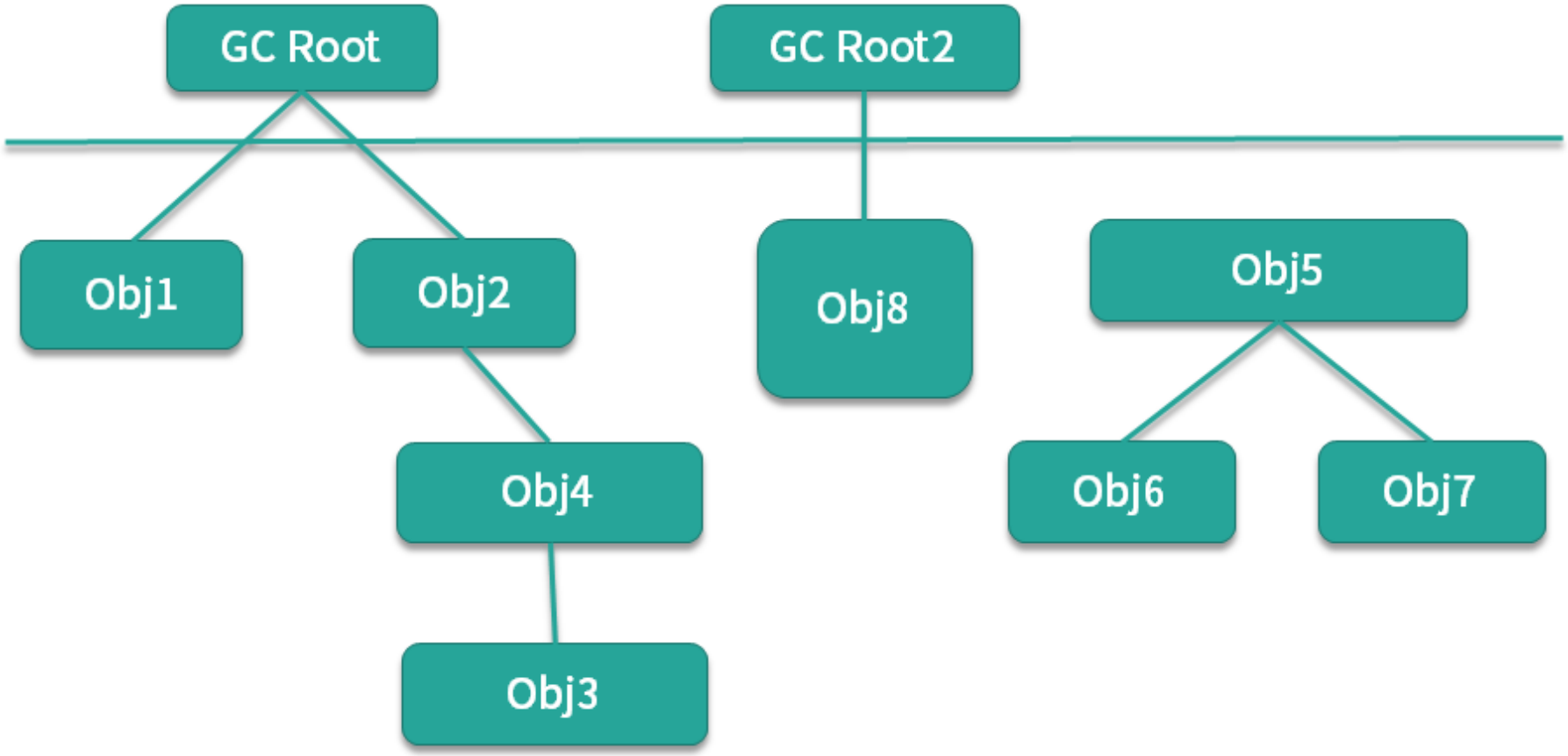
在古代，刑罚中有诛九族一说。指的是有些人犯大事时，皇上杀一人不足以平复内心的愤怒时，会对亲朋好友产生连带责任。诛九族时首先需要追溯到一个共同的祖先，再往下细数连坐。堆上的垃圾回收也有同样的思路。我们接下来就具体分析 JVM 中是如何进行垃圾回收的。

JVM 的 GC 动作，是不受程序控制的，它会在满足条件的时候，自动触发。

在发生 GC 的时候，一个对象，JVM 总能够找到引用它的祖先。找到最后，如果发现这个祖先已经名存实亡了，它们都会被清理掉。而能够躲过垃圾回收的那些祖先，比较特殊，它们的名字就叫作 GC Roots。

从 GC Roots 向下追溯、搜索，会产生一个叫作 Reference Chain 的链条。当一个对象不能和任何一个 GC Root 产生关系时，就会被无情的诛杀掉。

如图所示，Obj5、Obj6、Obj7，由于不能和 GC Root 产生关联，发生 GC 时，就会被摧毁。



垃圾回收就是围绕着 GC Roots 去做的。同时，它也是很多内存泄露的根源，因为其他引用根本没有这样的权利。

那么，什么样的对象，才会是 GC Root 呢？这不在于它是什么样的对象，而在于它所处的位置。

GC Roots 有哪些

GC Roots 是一组必须活跃的引用。用通俗的话来说，就是程序接下来通过直接引用或者间接引用，能够访问到的潜在被使用的对象。

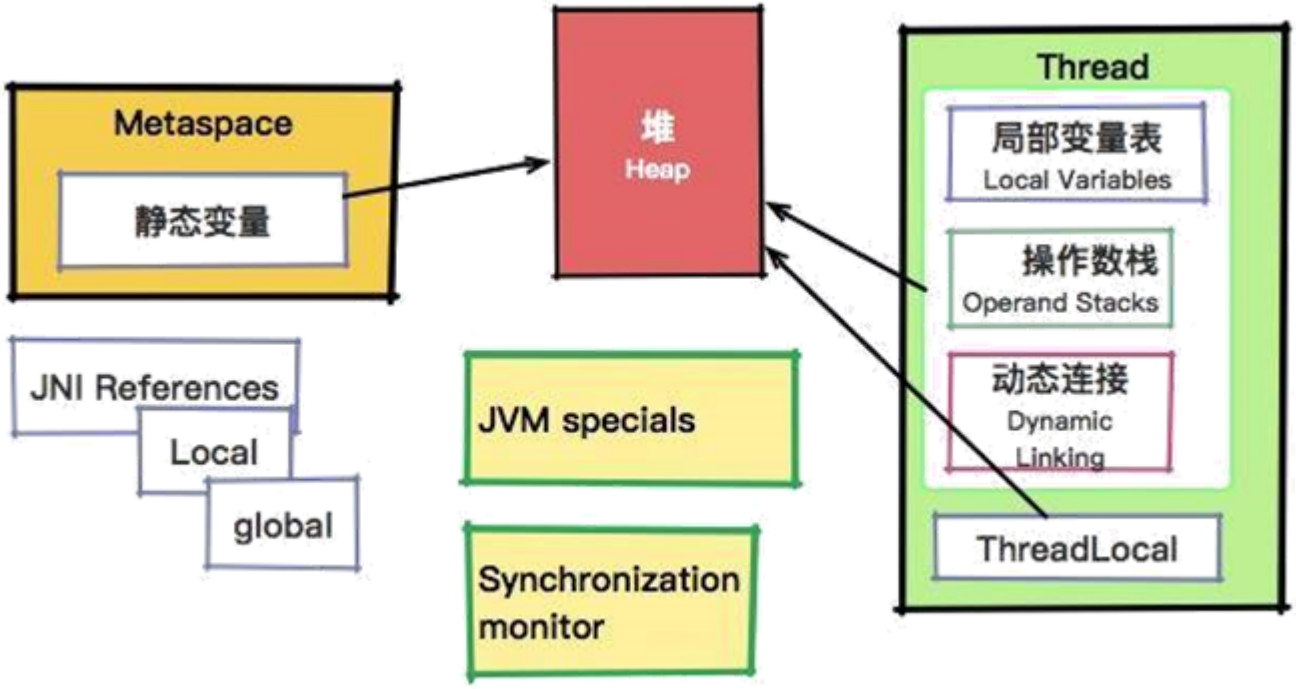
GC Roots 包括：

- Java 线程中，当前所有正在被调用的方法的引用类型参数、局部变量、临时值等。也就是与我们栈帧相关的各种引用。

- 所有当前被加载的 Java 类。
- Java 类的引用类型静态变量。
- 运行时常量池里的引用类型常量（String 或 Class 类型）。
- JVM 内部数据结构的一些引用，比如 sun.jvm.hotspot.memory.Universe 类。
- 用于同步的监控对象，比如调用了对象的 wait() 方法。
- JNI handles，包括 global handles 和 local handles。

这些 GC Roots 大体可以分为三大类，下面这种说法更加好记一些：

- 活动线程相关的各种引用。
- 类的静态变量的引用。
- JNI 引用。



有两个注意点：

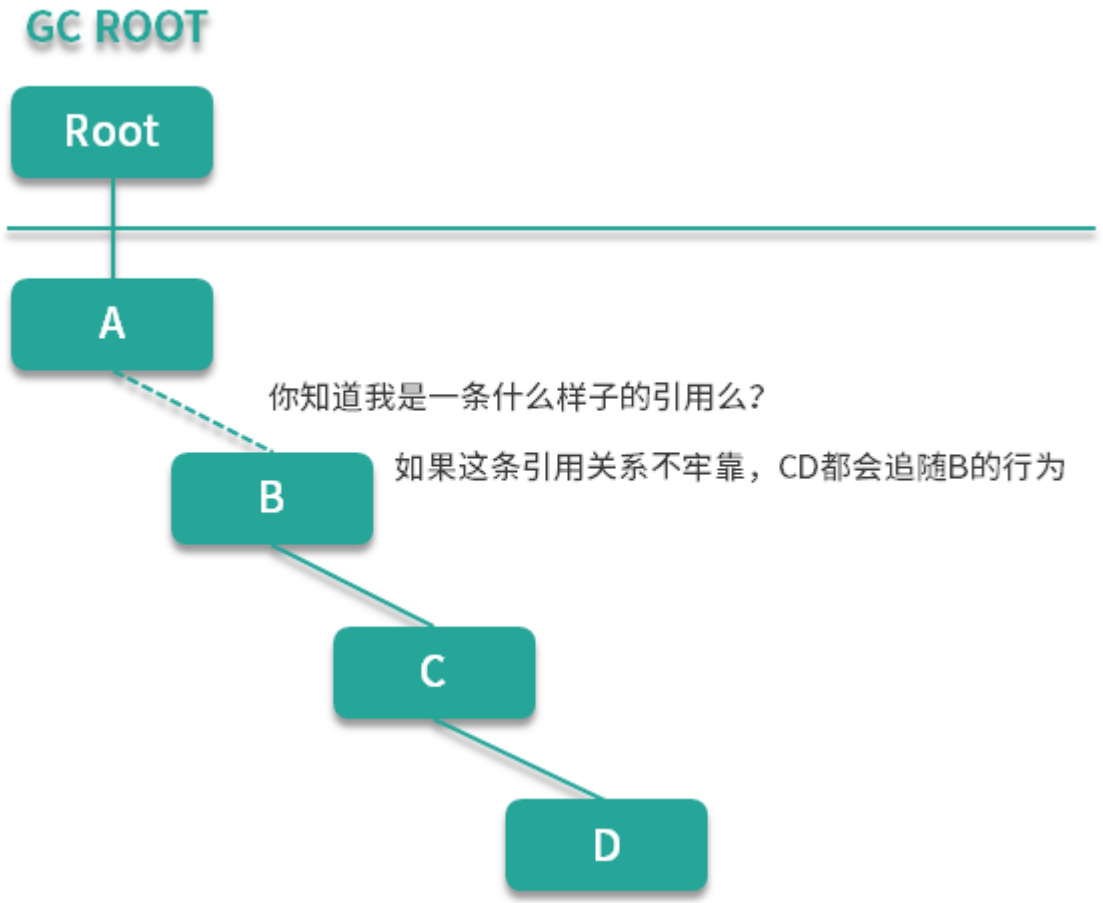
- 我们这里说的是活跃的引用，而不是对象，对象是不能作为 GC Roots 的。
- GC 过程是找出所有活对象，并把其余空间认定为“无用”；而不是找出所有死掉的对象，并回收它们占用的空间。所以，哪怕 JVM 的堆非常的大，基于 tracing 的 GC 方式，回收速度也会非常快。

引用级别

接下来的一道面试题就有意思多了：能够找到 Reference Chain 的对象，就一定会存活么？

我在面试的时候，经常会问这些问题，比如“弱引用有什么用处”？令我感到奇怪的是，即使是一些工作多年的 Java 工程师，对待这个问题也是一知半解，错失了很多机会。

对象对于另外一个对象的引用，要看关系牢靠不牢靠，可能在链条的其中一环，就断掉了。



根据发生 GC 时，这条链条的表现，可以对这个引用关系进行更加细致的划分。

它们的关系，可以分为强引用、软引用、弱引用、虚引用等。

强引用 Strong references

当内存空间不足，系统撑不住了，JVM 就会抛出 OutOfMemoryError 错误。即使程序会异常终止，这种对象也不会被回收。这种引用属于最普通最强硬的一种存在，只有在和 GC Roots 断绝关系时，才会被消灭掉。

这种引用，你每天的编码都在用。例如：new 一个普通的对象。

```
Object obj = new Object()
```

这种方式可能是有问题的。假如你的系统被大量用户（User）访问，你需要记录这个 User 访问的时间。可惜的是，User 对象里并没有这个字段，所以我们决定将这些信息额外开辟一个空间进行存放。

```
static Map<User,Long> userVisitMap = new HashMap<>();

...

userVisitMap.put(user, time);
```

当你用完了 User 对象，其实你是期望它被回收掉的。但是，由于它被 userVisitMap 引用，我们没有其他手段 remove 掉它。这个时候，就发生了内存泄漏（memory leak）。

这种情况还通常发生在一个没有设定上限的 Cache 系统，由于设置了不正确的引用方式，加上不正确的容量，很容易造成 OOM。

软引用 Soft references

软引用用于维护一些可有可无的对象。在内存足够的时候，软引用对象不会被回收，只有在内存不足时，系统则会回收软引用对象，如果回收了软引用对象之后仍然没有足够的内存，才会抛出内存溢出异常。

可以看到，这种特性非常适合用在缓存技术上。比如网页缓存、图片缓存等。

Guava 的 CacheBuilder，就提供了软引用和弱引用的设置方式。在这种场景中，软引用比强引用安全的多。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，Java 虚拟机就会把这个软引用加入到与之关联的引用队列中。

我们可以看一下它的代码。软引用需要显式的声明，使用泛型来实现。

```
// 伪代码

Object object = new Object();

SoftReference<Object> softRef = new SoftReference(object);
```

这里有一个相关的 JVM 参数。它的意思是：每 MB 堆空闲空间中 SoftReference 的存活时间。这个值的默认时间是1秒（1000）。

```
-XX:SoftRefLRUPolicyMSPerMB=<N>
```

这里要特别说明的是，网络上一些流传的优化方法，即把这个值设置成 0，其实是错误的，这样容易引发故障，感兴趣的话你可以自行搜索一下。

这种比较偏门的优化手段，除非在你对其原理相当了解的情况下，才能设置一些比较特殊的值。比如 0 值，无限大等，这种值在 JVM 的设置中，最好不要发生。

弱引用 Weak references

弱引用对象相比较软引用，要更加无用一些，它拥有更短的生命周期。

当 JVM 进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。弱引用拥有更短的生命周期，在 Java 中，用 java.lang.ref.WeakReference 类来表示。

它的应用场景和软引用类似，可以在一些对内存更加敏感的系统里采用。它的使用方式类似于这段的代码：

```
// 伪代码

Object object = new Object();
```

```
WeakReference<Object> softRef = new WeakReference(object);
```

虚引用 Phantom References

这是一种形同虚设的引用，在现实场景中用的不是很多。虚引用必须和引用队列（ReferenceQueue）联合使用。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

实际上，虚引用的 get，总是返回 null。

```
Object  object = new Object();

ReferenceQueue queue = new ReferenceQueue();

// 虚引用，必须与一个引用队列关联

PhantomReference pr = new PhantomReference(object, queue);
```

虚引用主要用来跟踪对象被垃圾回收的活动。

当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象之前，把这个虚引用加入到与之关联的引用队列中。

程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

下面的方法，就是一个用于监控 GC 发生的例子。

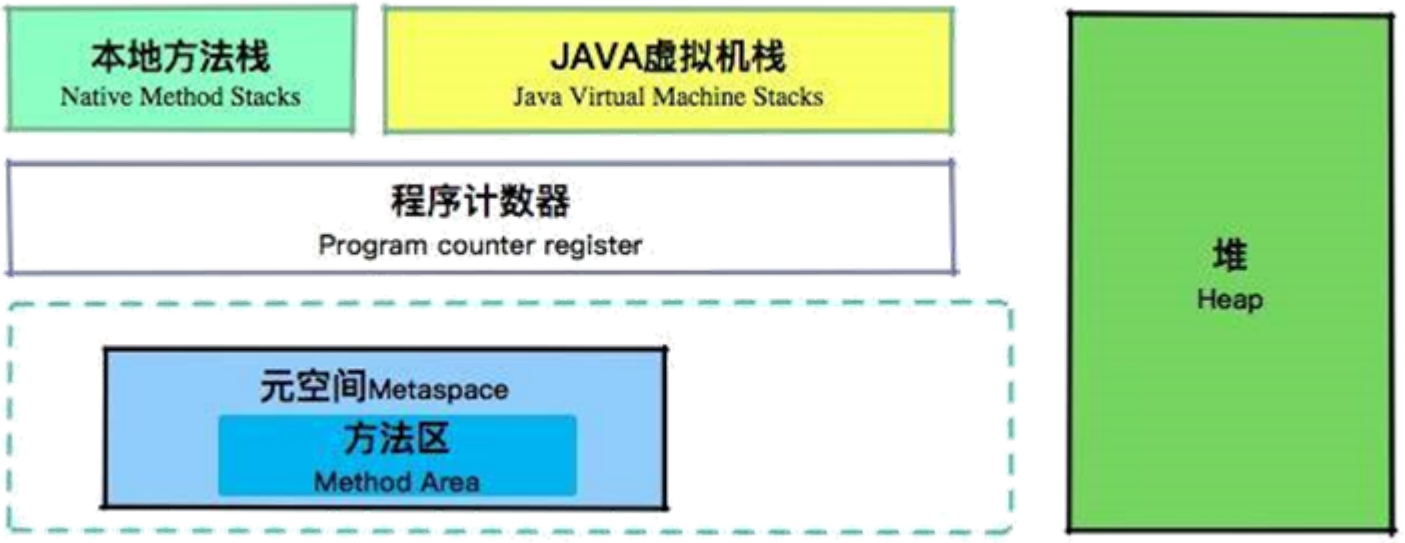
```
private static void startMonitoring(ReferenceQueue<MyObject> referenceQueue, Reference<MyObject> ref) {

     ExecutorService ex = Executors.newSingleThreadExecutor();
```

```
     ex.execute(() -> {
         while (referenceQueue.poll()!=ref) {
             //don't hang forever
             if(finishFlag){
                 break;
             }
         }
         System.out.println("-- ref gc'ed --");
     });
     ex.shutdown();
}
```

基于虚引用，有一个更加优雅的实现方式，那就是 Java 9 以后新加入的 Cleaner，用来替代 Object 类的 finalizer 方法。

典型 OOM 场景

OOM 的全称是 Out Of Memory，那我们的内存区域有哪些会发生 OOM 呢？我们可以从内存区域划分图上，看一下彩色部分。



可以看到除了程序计数器，其他区域都有OOM溢出的可能。但是最常见的还是发生在堆上。

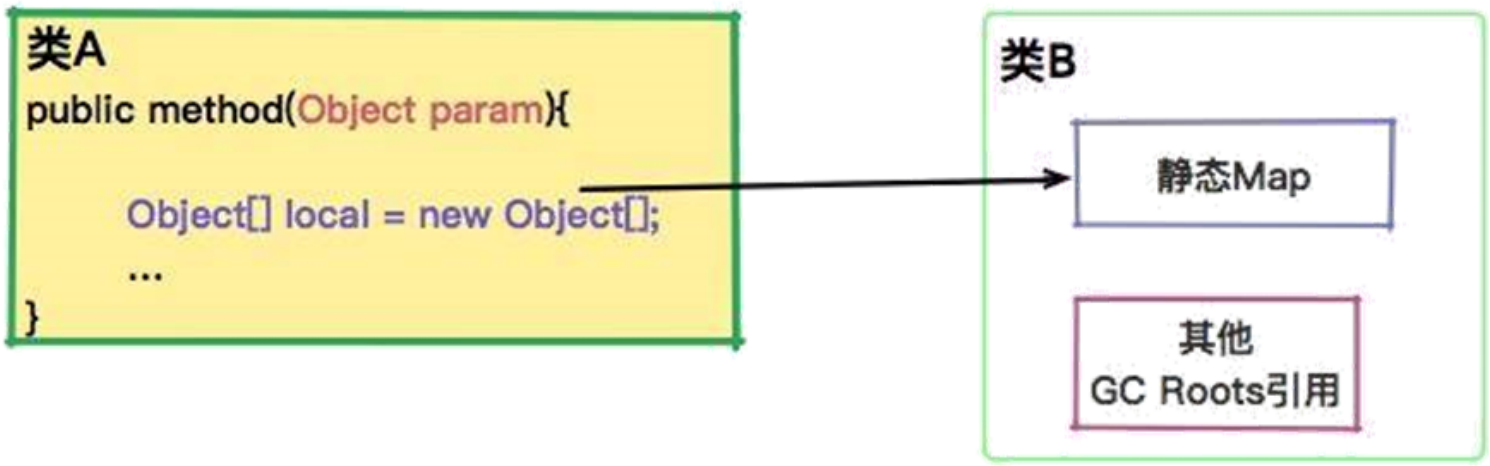
区域	是否线程私有	是否会发生OOM
程序计数器	是	否
虚拟机栈	是	是
本地方法栈	是	是
方法区	否	是
直接内存	否	是
堆	否	是

所以 OOM 到底是什么引起的呢？有几个原因：

- 内存的容量太小了，需要扩容，或者需要调整堆的空间。

- 错误的引用方式，发生了内存泄漏。没有及时的切断与 GC Roots 的关系。比如线程池里的线程，在复用的情况下忘记清理 ThreadLocal 的内容。
- 接口没有进行范围校验，外部传参超出范围。比如数据库查询时的每页条数等。
- 对堆外内存无限制的使用。这种情况一旦发生更加严重，会造成操作系统内存耗尽。

典型的内存泄漏场景，原因在于对象没有及时的释放自己的引用。比如一个局部变量，被外部的静态集合引用。



你在平常写代码时，一定要注意这种情况，千万不要为了方便把对象到处引用。即使引用了，也要在合适时机进行手动清理。关于这部分的问题根源排查，我们将在实践课程中详细介绍。

小结

你可以注意到 GC Roots 的专业叫法，就是可达性分析法。另外，还有一种叫作引用计数法的方式，在判断对象的存活问题上，经常被提及。

因为有循环依赖的硬伤，现在主流的 JVM，没有一个是采用引用计数法来实现 GC 的，所以我们大体了解一下就可以。引用计数法是在对象头里维护一个 counter 计数器，被引用一次数量 +1，引用失效记数 -1。计数器为 0 时，就被认为无效。你现在可以忘掉引用计数的方式了。

本课时，我们详细介绍了 GC Roots 都包含哪些内容。HostSpot 采用 tracing 的方式进行 GC，内存回收的速度与处于 living 状态的对象数量有关。

这部分涉及的内容较多，如果面试被问到，你可以采用白话版的方式进行介绍，然后举例深入。

接下来，我们了解到四种不同强度的引用类型，尤其是软引用和虚引用，在平常工作中使用还是比较多的。这里面最不常用的就是虚引用，但是它引申出来的 Cleaner 类，是用来替代 finalizer 方法的，这是一个比较重要的知识点。

本课时最后讨论了几种典型的 OOM 场景，你可能现在对其概念比较模糊。接下来的课时，我们将详细介绍几个常见的垃圾回收算法，然后对这些 OOM 的场景逐个击破。

上一页 下一页