垃圾回收机制

一、概述

说起垃圾收集(Gargbage Collection,GC),大部分人都把这项技术当做 Java 语言的伴生产物,事实上,GC 的历史比 Java 久远,1960 年诞生于 MIT 的 Lisp 是第一门真正使用内存动态分配和垃圾收集技术的语言.当 Lisp 还在胚胎的时期时,人们就在思考 GC 需要完成的 3 件事情:

1 哪些内存需要回收?

2 什么时候回收?

3 如何回收?

经过半个多世纪的发展,目前内存的动态分配与内存回收技术已经相当成熟,一切看起来都进入了"自动化"时代,那为什么我们还要去了解 GC 和内存分配呢?答案很简单:当需要排查各种内存溢出,内存泄漏问题 时,当垃圾收集成为系统达到更高并发量的瓶颈时,我们就需要对这些"自动化"的技术实施必要的监控和调节.

二、对象回收机制算法

1 引用计数算法

很多教科书判断对象是否存活的算法是这样的:给对象中添加一个引用计数器,每当有一个地方引用它时,计数器的值就加 1; 当引用失效时,计数器值就减 1; 任何时刻计数器为 0 的对象就是不可能再被使用的. 客观地说,引用计数算法(Reference Counting)的实现简单,判定效率也很高,在大部分情况下它都是一个不错的算法,也有一些比较著名的应用案例,例如微软公司的 COM(Component Object Model)技术,使用 ActionScript3 的 FlashPlayer,Python 语言和在游戏脚本领域被广泛应用的 Squirrel 中都使用了引用计数算法进行内存管理.但是,至少主流的 Java 虚拟机里没有选用引用计算算法来管理内存,其中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, 虽然两个对象内部都使用引用指向对方对象, 但 A,B 对象已经没有了外部引用指向它们, 显然这两个对象应当被回收,但是如果使用计数算法,显然是无能无力的.

2 可达性分析算法

在主流的商用程序语言(Java, C#,甚至包括前面提到的古老的 Lisp)的主流实现中,都是通过可达性分析(Reachablity 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称为"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为引用链(Reference Chain),当一个对象到 GC Roots 没有任何引用链相连(用图论的话来说,就是从 GC Roots 到这个对象不可达)时,则证明此对象是不可用的. 这样, 虽然有些对象内部仍然有引用在互相引用,但是如果通过 GC Roots 不能到达它们, 它们就仍然会被判定为是垃圾对象.

三、再谈引用:

无论是通过引用计数算法判断对象的引用数量,还是通过可达性分析算法判断对象的引用链是否可达,判定对象是否存活都与"引用"有关.在 JDK1.2 以前,Java 中的引用的定义很传统:如果 reference 类型的数据中存储的数值代表的是另外一块内存的起始地址,就称这块内存代表着一个引用。这种定义很纯粹,但是太过狭隘,一个对象在这种定义下只有被引用或者没有被引用两种状态,对于如何描述一些"食之无味,弃之可惜"的对象就显得无能为力.我们希望能描述这样一类对象:当内存空间还足够时,则能保留在内存之中;如果内存空间在进行垃圾收集后还是非常紧张,则可以抛弃这些对象.很多系统的缓存功能都符合这样的应用场景.

在 JDK1.2 之后,Java 对引用的概念进行了扩充,将引用分为强引用(Strong Reference),软引用(Soft Reference),弱引用(Weak Reference),虚引用(Phantom Reference)4 种,这 4 种引用强度依次逐渐减弱.

强引用 就是在指在程序代码之中普遍存在的,类似"Object obj = new Object()" 这类的引用,只要强引用还在,垃圾收集器永远不会加收掉被引用对象.

软引用 就是用来描述一还有用但并非必需的对象. 对于软引用关联着的对象,在系统将要发生内存溢出之前,将会把这些对象列进回收范围之中进行第二次回收.如果这次回收还没有足够的内存,才会抛出内存溢出异常.在 JDK1.2 之后,提供了 SoftReference 类来实现软引用.

弱引用 也是用来描述非必需对象的,但是它的强度比软引用更弱一,被弱引用关联的对象只能生存到下一次垃圾收集发生之前.当垃圾收集器工作时,无论当前内存是否足够,都会回收掉只被弱引用关联的对象,在 JDK1.2 之后,提供了 WeakReference 类来实现弱引用.

虚引用 也称为幽灵引用或者幻影引用,它是最弱的一种引用关系.一个对象是否有虚引用存在,完全不会对其生存时间构成影响,也无法通过虚引用来取得一个对象实例.为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知.在 JDK1.2 之后,提供了 PhantomReference 类实现虚引用

四、生存还是死亡:

即使在可达性分析算法中不可达的对象,也并非是"非死不可"的,这时候它们暂时处于"缓刑"阶段,要真正宣告一个对象死亡,至少要经历两次标记过程:如果对象在进行可达性分析后发现没有与 GC Roots 相连接的引用链,那它将会被第一次标记并且进行一次筛选,筛选的条件是此对象是否有必要执行 finalize()方法.当对象没有覆盖 finalize()方法,或者 finalize()方法已经被虚拟机调用过,虚拟机将这两种情况都视为"没有必要执行".

如果这个对象被判定为有必要执行 finalize()方法,那么这个对象将会放置在一个叫做 F-Queue 的队列之中,并在稍后由一个由虚拟机自动建立的,低优先的 Finalizer 线程去执行它. 这里所谓的"执行" 是指虚拟机会触发这个方法,但并不承诺会等待它运行结束, 这样做的原因是,如果一个对象在 finalze()方法中执行缓慢,或者发生了死循环, 将很可能会导致 F-Queue 队列中的其他对象永久处于等待,甚至导致整个内存回收系统崩溃.finalize()方法是对象逃脱死亡命运的最后一次机会, 稍后 GC 将对 F-Queue 中的对象进行第二次小规模的标记,如果对象要在 finalze()中成功拯救自己 -- 只要重新与引用链上的任何一个对象建立关联即可,譬如把 this 赋值给某个类变量或者对象的成员变量,那在第二次标记时它将被移

除出"即将回收"的集合;如果对象这时候还没有逃脱,那基本上它就真的被回收了.

下面的代码演示一个对象的自我拯救

```
public class FinalizeEscapeGC {
	public static FinalizeEscapeGC saveHook = null;

	public void isAlive() {
		System.out.println("yes, I am still alive");
	}

	@Override
	protected void finalize() throws Throwable {
		super.finalize();
		System.out.println("finalize method executed!");
		saveHook = this;
	}

	public static void main(String[] args) {
		saveHook = new FinalizeEscapeGC();

		// 对象第一次拯救自己 (成功)
		saveHook = null;
		System.gc();
		// 因为 finalize 方法优先级很低,所以暂停 0.5 秒等待它
		Thread.sleep(500);
		if (saveHook != null) {
			saveHook.isAlive();
		} else {
			System.out.println("no, I am dead");
		}

		// 对象第二次拯救自己 (失败)
		saveHook = null;
		System.gc();
		// 因为 finalize 方法优先级很低,所以暂停 0.5 秒等待它
		Thread.sleep(500);
		if (saveHook != null) {
			saveHook.isAlive();
		} else {
			System.out.println("no, I am dead");
		}
	}
}
```

以上程序执行结果:
finalize method executed!
yes, I am still alive
no, I am dead

对象的 finalize()方法确实 GC 收集器触发过,并且在被收集前成功逃脱了. 另外一个值得注意的地方是,代码中有两段完全一样的代码片断,执行结果却是一次逃脱成功,一次失败,这是因为任何一个对象的 finalze()方法都只会被系统自动调用一次,如果对象面临下一次回收,它的 finalize()方法不会被再次执行,因此第二段代码的自救行动失败了.

上面的代码虽然演示了一个对象自救过程,并且能够成功, 但是建议尽量避免使用它,因为它不是 C/C++中的析构函数,而是 Java 刚诞生时为了使 C/C++程序员更容易接受它所做出的一个妥协.它的运行代价高昂, 不确定性大,无法保证各个对象的调用顺序.有些人描述它适合做"关闭外部资源"之类的工作,这完全是对这个方法用途的自我安慰,finalize()能做的工作, 使用 try-finally 或者其他方式都可以做的更好,更及时. 所以尽量避免依赖 finalize()方法.