Java与c++的不同：内存动态分配和自动内存收集

垃圾收集（Gargage collection），即GC。

GC需要完成的3件事：

1. 哪些内存需要回收/释放
2. 什么时候回收/释放
3. 如何回收/释放

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，就需要对这些自动化的技术实施必要的监控和调节。

程序计数器、虚拟机栈和本地方法栈随线程而生，随线程而灭；栈帧随着方法的调用和返回而入栈和出栈，栈帧中的内存大小基本上也是确定的。这几个区域的内存分配和回收都具备确定性。方法结束或者线程结束时内存自然就跟着回收了。

而java堆和方法区则不同，这些区域只有在运行阶段才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，垃圾收集器所关注的是这部分的内存。

判断对象是否已不可能被任何途径使用的方式：

1. 引用计数法
2. 可达性分析法

引用计数法：给对象添加一个引用，每当有一个地方引用它时，计数器+1，当引用失效时计数器-1。当计数器为0时则是不可能被使用的。

实现简单判断效率高，但是至少主流的jvm没有选用引用计数算法来管理内存，其中最主要的原因是它很难解决对象之间的循环引用问题。

可达性分析：在主流的商用程序语言的主流实现中，都是通过可达性分析来判断对象是否存活的。该算法就是通过一系列的GC ROOTS为起点，向下搜索，所走过的路称为引用链，当在链上说明是可达的，否则就是不可达的，就是需要被回收的对象。

在java中，可作为GC ROOTS的对象包括：

1. 虚拟机栈（栈帧中本地变量表）中引用的对象
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（一般说法是Native方法）引用的对象

引用分为：

1. 强引用
   1. 类似Object obj=new Object();垃圾收集器永远不会回收掉被引用的对象
2. 软引用
   1. 用来描述一些还有用，但是并非必须的对象
   2. 在系统将要发生内存溢出异常之前，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则抛出内存溢出异常
3. 弱引用
   1. 也是用来描述非必须的对象的，但是它的强度比软引用更弱一些。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。当垃圾收集器开始工作时，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
4. 虚引用
   1. 也称为幽灵引用或者幻影引用，他是最弱的一种引用关系
   2. 为一个对象设置虚引用关联的唯一目的是能在这个对象被垃圾收集器回收时收到一个系统通知。

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，这时候他们暂时处于缓刑阶段，要真正宣告一个对象死亡，至少需要经历两次标记过程：

1. 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，那么他将会被第一次标记并且进行一次筛选
2. 第二次标记

建议尽量不要使用java中的finalize方法。

回收方法区（即hotspot虚拟机中的永久代）：

永久代的垃圾收集主要回收两部分内容：

1. 废弃常量：回收废弃常量与回收java堆中的对象非常相似。
2. 无用的类

判断一个类是否是无用的条件相对苛刻，需要同时满足以下条件：

1. 该类所有的实例都已经被回收，也就是java堆中不存在该类的任何实例
2. 加载该类的classloader已经被回收
3. 该类对应的java.lang.class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法

满足以上三个条件的被判定为无用类，可以被回收，但也仅仅是可以被回收，并不一定会必然被回收。是否对类进行回收，jvm提供了-Xnoclassgc参数进行控制，还可以使用-verbose:class以及-XX:+TraceClassLoading、-XX:+TraceClassUnLoading或查看类加载和卸载信息。

无用类在必要的时候需要被回收以保证永久代不会溢出。

垃圾 回收算法：

1. 标记-清除算法
2. 复制算法
3. 标记-整理算法
4. 分代收集算法

标记-清除算法：

1. 最基础的算法；算法分为标记和清除两个部分；

2. 首先标记出所有需要回收的对象，在标记完成之后统一回收所有被标记的对象

主要不足：

1. 效率低，标记和清除两个过程效率都不高
2. 空间问题，标记清除之后会产生大量的内存碎片

复制算法：

该算法将可用内存按容量分为大小相等的两块，每次只使用其中的一块。当一块的内存用完了，就将还存活的对象复制到另一块上面，然后把已经使用的内存空间一次清理掉。该方法使得只能利用内存的一半，代价高昂。

现在的商业虚拟机都采用这种算法来回收新生代。但内存划分比上面的复杂。

标记-整理算法：

标记过程与标记-清除算法的标记过程一样，但后续步骤不是直接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：

当前商业jvm的垃圾收集都采用分代收集算法。该算法根据对象的存活周期的不同将内存划分为几块，一般把java堆划分为新生代和老年代。这样就可以根据各个年代的特点采用最合适的收集算法。

在新生代中，每次垃圾收集都发现有大批对象死去，只有少量存活，那就使用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。

而老年代中因为对象存活率高没额外的空间对它进行分配担保，就必须使用标记-清理或者标记-整理算法进行回收。

垃圾收集器分类：

内存分配与回收策略：

Java体系中所提倡的自动内存管理最终可归结为两个问题：

1. 给对象分配内存
2. 回收分配给对象的内存

对象主要分配在新生代的Eden上。

对象优先在Eden分配：

当Eden区没有足够空间分配时，虚拟机将触发一次Minor GC。

大对象直接进入老年代：

所谓大对象是指，需要大量连续内存空间的java对象，最典型的大对象就是那种很长的字符串以及数组。

写程序时应该避免，经常出现大对象容易导致内存还有不少空间时就提前触发垃圾收集以获取足够的连续空间来安置它们。

虚拟机提供了-XX:PretenureSizeThreshold的参数，令大于这个设置值的对象直接在老年代分配。这样做的目的是避免在Eden区及两个Survivor区间之间发生大量的内存复制（新生代采用复制算法收集内存）。

长期存活的对象将进入老年代：

虚拟机采用了分代收集的思想来管理内存。那么内存回收时就必须能够识别哪些对象应该在新生代，哪些对象应该在老年代。

虚拟机为每个对象定义了一个对象年龄计数器：如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，将被移动到Survivor空间中，并且对象年龄设为1。对象在Survivor区中熬过一次Minor GC，年龄就增加1岁。当它的年龄增加到一定程度（默认15岁）就将被晋升到老年代中，对象晋升老年代的年龄阈值可以通过参数-XX:MaxTenuringThreshold设置。

动态对象年龄判定：

晋升老年代的年龄并不是必须的严格条件。如果在Survivor空间中相同年龄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就可以直接进入老年代。无需等到MaxTenuringThreshold要求的年龄。

空间分配担保：