**JVM**

1. **如何确定垃圾**？

**引用计数法**

在 Java 中，引用和对象是有关联的。如果要操作对象则必须用引用进行。因此，很显然一个简单 的办法是通过引用计数来判断一个对象是否可以回收。简单说，即一个对象如果没有任何与之关 联的引用，即他们的引用计数都不为 0，则说明对象不太可能再被用到，那么这个对象就是可回收 对象。

**可达性分析**

为了解决引用计数法的循环引用问题，Java 使用了可达性分析的方法。通过一系列的“GC roots” 对象作为起点搜索。如果在“GC roots”和一个对象之间没有可达路径，则称该对象是不可达的。要注意的是，不可达对象不等价于可回收对象，不可达对象变为可回收对象至少要经过两次标记 过程。两次标记后仍然是可回收对象，则将面临回收。

**2.JAVA 四中引用类型**

**强引用**

在 Java 中最常见的就是强引用，把一个对象赋给一个引用变量，这个引用变量就是一个强引 用。当一个对象被强引用变量引用时，它处于可达状态，它是不可能被垃圾回收机制回收的，即 使该对象以后永远都不会被用到 JVM 也不会回收。因此强引用是造成 Java 内存泄漏的主要原因之 一。

**软引用**

软引用需要用 SoftReference 类来实现，对于只有软引用的对象来说，当系统内存足够时它 不会被回收，当系统内存空间不足时它会被回收。软引用通常用在对内存敏感的程序中。

**弱引用**

弱引用需要用 WeakReference 类来实现，它比软引用的生存期更短，对于只有弱引用的对象 来说，只要垃圾回收机制一运行，不管 JVM 的内存空间是否足够，总会回收该对象占用的内存。

**虚引用**

虚引用需要 PhantomReference 类来实现，它不能单独使用，必须和引用队列联合使用。虚 引用的主要作用是跟踪对象被垃圾回收的状态。

**3.GC 分代收集算法 VS 分区收集算法**

**分代收集算法**当前主流 VM 垃圾收集都采用”分代收集”(Generational Collection)算法, 这种算法会根据对象存活周期的不同将内存划分为几块, 如 JVM 中的 新生代、老年代、永久代，这样就可以根据各年代特点分别采用最适当的 GC 算法

**在新生代-复制算法**

每次垃圾收集都能发现大批对象已死, 只有少量存活. 因此选用复制算法, 只需要付出少量存活对象的复制成本就可以完成收集

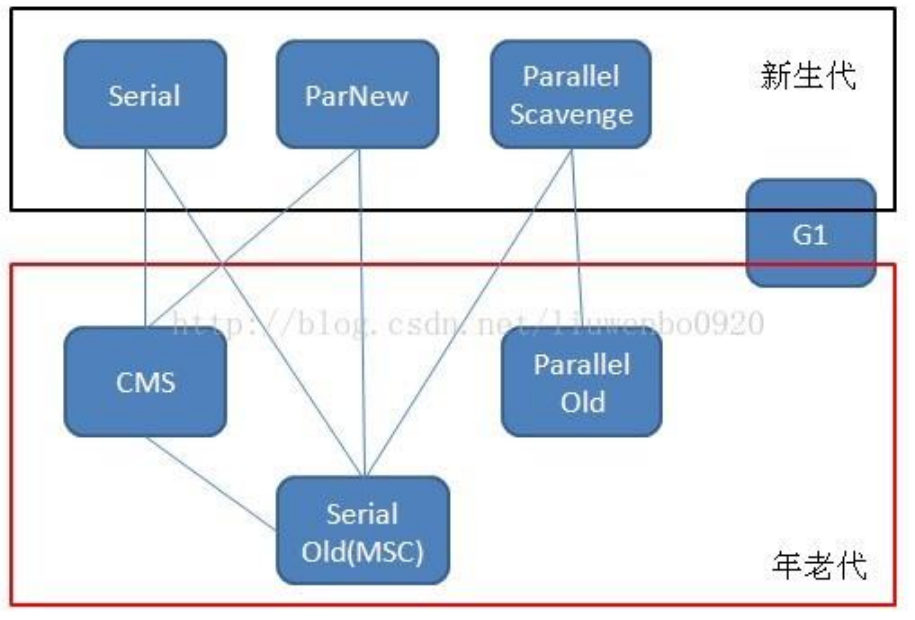
**在老年代-标记整理算法**

因为对象存活率高、没有额外空间对它进行分配担保, 就必须采用“标记—清理”或“标记—整理”算法来进行回收, 不必进行内存复制, 且直接腾出空闲内存.

**分区收集算法**

分区算法则将整个堆空间划分为连续的不同小区间, 每个小区间独立使用, 独立回收. 这样做的好处是可以控制一次回收多少个小区间 , 根据目标停顿时间, 每次合理地回收若干个小区间(而不是整个堆), 从而减少一次 GC 所产生的停顿。

1. **GC 垃圾收集器**



**Serial 垃圾收集器（单线程、复制算法）**

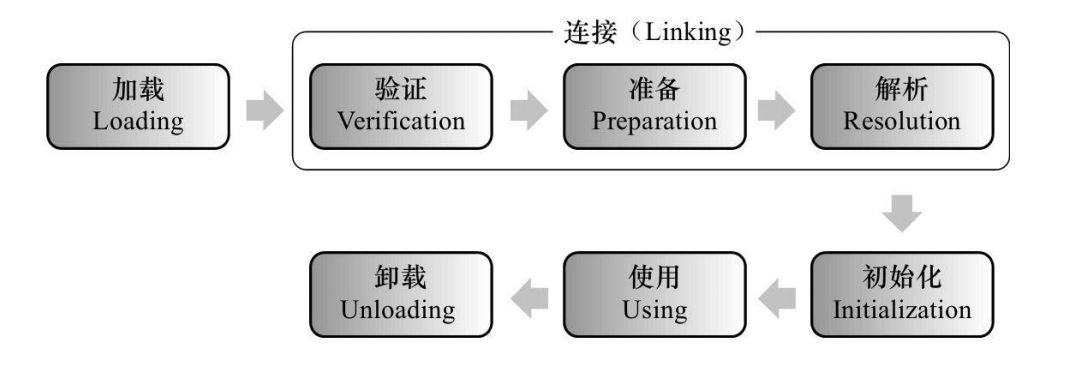
Serial（英文连续）是最基本垃圾收集器，使用复制算法，曾经是JDK1.3.1 之前新生代唯一的垃圾 收集器。Serial 是一个单线程的收集器，它不但只会使用一个 CPU 或一条线程去完成垃圾收集工 作，并且在进行垃圾收集的同时，必须暂停其他所有的工作线程，直到垃圾收集结束。 Serial 垃圾收集器虽然在收集垃圾过程中需要暂停所有其他的工作线程，但是它简单高效，对于限 定单个 CPU 环境来说，没有线程交互的开销，可以获得最高的单线程垃圾收集效率，因此 Serial 垃圾收集器依然是 java 虚拟机运行在 Client 模式下默认的新生代垃圾收集器。

**ParNew 垃圾收集器（Serial+多线程）**

ParNew 垃圾收集器其实是 Serial 收集器的多线程版本，也使用复制算法，除了使用多线程进行垃 圾收集之外，其余的行为和 Serial 收集器完全一样，ParNew 垃圾收集器在垃圾收集过程中同样也 要暂停所有其他的工作线程。 13/04/2018 Page 32 of 283 ParNew 收集器默认开启和 CPU 数目相同的线程数，可以通过-XX:ParallelGCThreads 参数来限 制垃圾收集器的线程数。【Parallel：平行的】 ParNew虽然是除了多线程外和Serial 收集器几乎完全一样，但是ParNew垃圾收集器是很多 java 虚拟机运行在 Server 模式下新生代的默认垃圾收集器。

**5.JVM 类加载机制**

JVM 类加载机制分为五个部分：加载，验证，准备，解析，初始化，下面我们就分别来看一下这五个过程。



加载：是类加载过程中的一个阶段，这个阶段会在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法区这个类的各种数据的入口。

**验证**： 这一阶段的主要目的是为了确保 Class 文件的字节流中包含的信息是否符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

**准备**：准备阶段是正式为类变量分配内存并设置类变量的初始值阶段，即在方法区中分配这些变量所使 用的内存空间。注意这里所说的初始值概念，比如一个类变量定义为：

public static int v = 8080;

实际上变量 v 在准备阶段过后的初始值为 0 而不是 8080，将 v 赋值为 8080 的 put static 指令是 程序被编译后，存放于类构造器方法之中。

但是注意如果声明为：

public static final int v = 8080;

在编译阶段会为 v 生成 ConstantValue 属性，在准备阶段虚拟机会根据 ConstantValue 属性将 v 赋值为 8080。

**解析** 解析阶段是指虚拟机将常量池中的符号引用替换为直接引用的过程。符号引用就是 class 文件中 的：

1. CONSTANT\_Class\_info

2. CONSTANT\_Field\_info

3. CONSTANT\_Method\_info 等类型的常量。

**初始化**

初始化阶段是类加载最后一个阶段，前面的类加载阶段之后，除了在加载阶段可以自定义类加载 器以外，其它操作都由 JVM 主导。到了初始阶段，才开始真正执行类中定义的 Java 程序代码。

**类构造器**

初始化阶段是执行类构造器方法的过程。方法是由编译器自动收集类中的类变 量的赋值操作和静态语句块中的语句合并而成的。虚拟机会保证子方法执行之前，父类 的方法已经执行完毕，如果一个类中没有对静态变量赋值也没有静态语句块，那么编译 器可以不为这个类生成()方法。

注意以下几种情况不会执行类初始化：

1. 通过子类引用父类的静态字段，只会触发父类的初始化，而不会触发子类的初始化。
2. 定义对象数组，不会触发该类的初始化。

3. 常量在编译期间会存入调用类的常量池中，本质上并没有直接引用定义常量的类，不会触 发定义常量所在的类。

4. 通过类名获取 Class 对象，不会触发类的初始化。

5. 通过 Class.forName 加载指定类时，如果指定参数 initialize 为 false 时，也不会触发类初 始化，其实这个参数是告诉虚拟机，是否要对类进行初始化。

6. 通过 ClassLoader 默认的 loadClass 方法，也不会触发初始化动作。

**类加载器**

虚拟机设计团队把加载动作放到 JVM 外部实现，以便让应用程序决定如何获取所需的类，JVM 提 供了 3 种类加载器：

**启动类加载器(Bootstrap ClassLoader**) 1. 负责加载 JAVA\_HOME\lib 目录中的，或通过-Xbootclasspath 参数指定路径中的，且被 虚拟机认可（按文件名识别，如 rt.jar）的类。

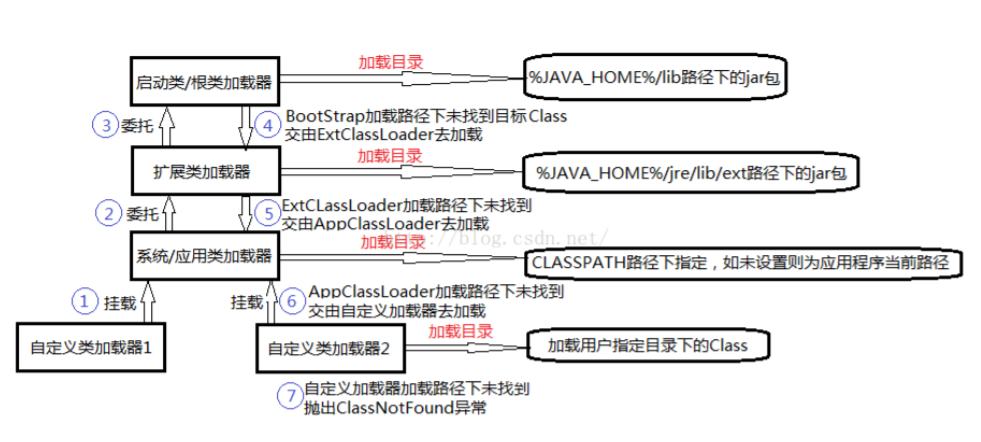
**扩展类加载器(Extension ClassLoader)** 2. 负责加载 JAVA\_HOME\lib\ext 目录中的，或通过 java.ext.dirs 系统变量指定路径中的类 库。

**应用程序类加载器(Application ClassLoader)**： 3. 负责加载用户路径（classpath）上的类库。 JVM 通过双亲委派模型进行类的加载，当然我们也可以通过继承 java.lang.ClassLoader 实现自定义的类加载器。

**双亲委派**

当一个类收到了类加载请求，他首先不会尝试自己去加载这个类，而是把这个请求委派给父 类去完成，每一个层次类加载器都是如此，因此所有的加载请求都应该传送到启动类加载其中， 只有当父类加载器反馈自己无法完成这个请求的时候（在它的加载路径下没有找到所需加载的 Class），子类加载器才会尝试自己去加载。

采用双亲委派的一个好处是比如加载位于 rt.jar 包中的类 java.lang.Object，不管是哪个加载 器加载这个类，最终都是委托给顶层的启动类加载器进行加载，这样就保证了使用不同的类加载 器最终得到的都是同样一个 Object 对象。

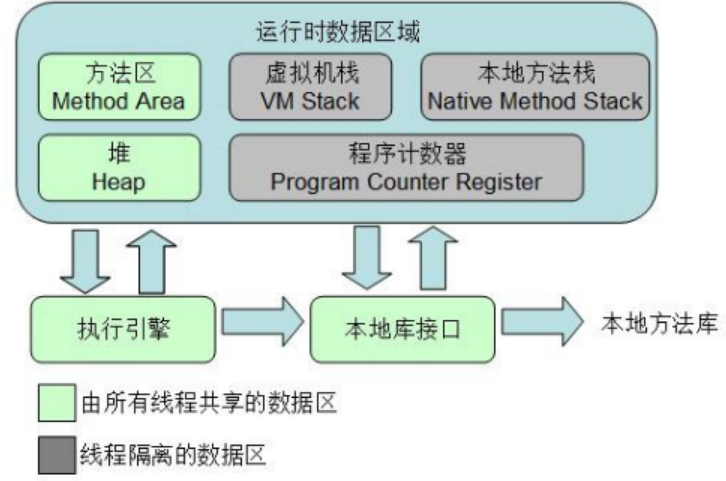


**例题：**

### **1. 说一下Java的垃圾回收机制。**

    它使得 Java 程序员在编写程序的时候不再需要考虑内存管理。垃圾回收器 通常是作为一个单独的低级别的线程运行,不可预知的情况下对内存堆中已经 死亡的或者长时间没有使用的对象进行清除和回收,程序员不能实时的调用垃 圾回收器对某个对象或所有对象进行垃圾回收。程序员可以手动执行 System.gc(),通知 GC 运行,但是 Java 语言规范并不保证 GC 一定会执行。   
     垃圾回收机制可以用3个词来概括:where,when和how?   
     Where:运行时的内存分布情况。见下一题。  
     When:对象何时需要被回收的?也就是何时回收无效对象,已死对象的?   
     这里涉及到两种做法:引用计数法和可达性分析算法。这里还涉及到java 中4种引用方式:强引用,软引用,弱引用和虚引用,其引用强度越来越来低,意味着引用越弱的对象越容易被垃圾回收的。  
     how:对象如何被回收的?4 种垃圾回收算法。

### **2. JVM的内存布局/内存模型。(需要详细到每个区放什么)**



### **3. JVM的4种引用和使用场景?**

     这4种级别由高到低依次为:强引用、软引用、弱引用和虚引用。   
     (1)强引用(StrongReference)   
     强引用是使用最普遍的引用。如果一个对象具有强引用,那垃圾回收器绝 不会回收它。当内存空间不足, Java 虚拟机宁愿抛出 OutOfMemoryError 错误,使程序异常终止,也不会靠随意回收具有强引用的对象来解决内存不足的问 题。 ps:强引用其实也就是我们平时 A a = new A()这个意思。   
     (2)软引用(SoftReference)   
     如果一个对象只具有软引用, 则内存空间足够, 垃圾回收器就不会回收它; 如果内存空间不足了,就会回收这些对象的内存。只要垃圾回收器没有回收它, 该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存(下文给出示例)。   
     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器回收,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。   
     示例:实现学生信息查询操作时有两套数据操作的方案。   
     一、将得到的信息存放在内存中, 后续查询则直接读取内存信息(优点: 读取速度快; 缺点: 内存空间一直被占, 若资源访问量不高,则浪费内存空间)。   
     二、每次查询均从数据库读取, 然后填充到 TO 返回。(优点: 内存空间 将被 GC 回收, 不会一直被占用;缺点: 在 GC 发生之前已有的 TO 依然存在, 但还是执行了一次数据库查询,浪费 IO)。可以通过软引用来解决。   
     (3)弱引用(WeakReference)   
     弱引用与软引用的区别在于: 只具有弱引用的对象拥有更短暂的生命周期。 在垃圾回收器线程扫描它所管辖的内存区域的过程中, 一旦发现了只具有弱引 用的对象, 不管当前内存空间足够与否, 都会回收它的内存。不过,由于垃圾 回收器是一个优先级很低的线程, 因此不一定会很快发现那些只具有弱引用的对象。   
     弱引用可以和一个引用队列(ReferenceQueue)联合使用,如果弱引用所 引用的对象被垃圾回收,Java 虚拟机就会把这个弱引用加入到与之关联的引用 队列中。   
     (4)虚引用(PhantomReference)   
     “虚引用”顾名思义,就是形同虚设,与其他几种引用都不同,虚引用并不会 决定对象的生命周期。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器回收。   
     虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱 引用的一个区别在于:虚引用必须和引用队列 (ReferenceQueue)联合使用。 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 的内存之前,把这个虚引用加入到与之关联的引用队列中。   
ReferenceQueue queue = new ReferenceQueue ();   
PhantomReference pr = new PhantomReference (object, queue);   
     比较容易理解的是 Java 垃圾回收器会优先清理可达强度低的对象。   
     那现在问题来了,若一个对象的引用类型有多个,那到底如何判断它的可 达性呢?其实规则如下:(“单弱多强”)

1. 单条引用链的可达性以最弱的一个引用类型来决定;
2. 多条引用链的可达性以最强的一个引用类型来决定;

### **4. 说一下引用计数法与可达性分析算法**

引用计数法

在 Java 中，引用和对象是有关联的。如果要操作对象则必须用引用进行。因此，很显然一个简单 的办法是通过引用计数来判断一个对象是否可以回收。简单说，即一个对象如果没有任何与之关 联的引用，即他们的引用计数都不为 0，则说明对象不太可能再被用到，那么这个对象就是可回收 对象。

可达性分析

为了解决引用计数法的循环引用问题，Java 使用了可达性分析的方法。通过一系列的“GC roots” 对象作为起点搜索。如果在“GC roots”和一个对象之间没有可达路径，则称该对象是不可达的。要注意的是，不可达对象不等价于可回收对象，不可达对象变为可回收对象至少要经过两次标记 过程。两次标记后仍然是可回收对象，则将面临回收。

### **5.Minor GC与Full GC分别在什么时候发生?**

     如果 Eden 空间占满了, 会触发 minor GC。 Minor GC 后仍然存活的对 象会被复制到 S0 中去。这样 Eden 就被清空可以分配给新的对象。 又触发了一次 Minor GC , S0 和 Eden 中存活的对象被复制到 S1 中, 并且 S0 和 Eden 被清空。 在同一时刻, 只有 Eden 和一个 Survivor Space 同时被操作。 当每次对象从 Eden 复制到 Survivor Space 或者从 Survivor Space 中的一个复制 到另外一个,有一个计数器会自动增加值。 默认情况下如果复制发生超过 16次, JVM 会停止复制并把他们移到老年代中去。 如果一个对象不能在 Eden 中被创建,它会直接被创建在老年代中。 如果 老年代的空间被占满会触发老年代的 GC,也被称为 Full GC。Full GC 是一个 压缩处理过程,所以它比 Minor GC 要慢很多。

* 内存分配规则:   
  1.对象优先分配在 Eden 区,如果 Eden 区没有足够的空间时,虚拟机执行一次 Minor GC。  
  2.大对象直接进入老年代(大对象是指需要大量连续内存空间的对象)。这样做的目的是 避免在 Eden 区和两个 Survivor 区之间发生大量的内存拷贝(新生代采用复制算法收集内存)。  
  3.长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器,如果对象经过 了 1 次 Minor GC 那么对象会进入 Survivor 区,之后每经过一次 Minor GC 那么对象 的年龄加 1,直到达到阀值,对象进入老年区。   
  4.动态判断对象的年龄。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。   
  5.空间分配担保。每次进行 Minor GC 时,JVM 会计算 Survivor 区移至老年区的对象的 平均大小,如果这个值大于老年区的剩余值大小则进行一次 Full GC,如果小于检查 HandlePromotionFailure 设置,如果 true 则只进行 Monitor GC,如果 false 则进 行 Full GC。

### **6. 对象创建方法,对象的内存布局,对象的访问定位。**

四种不同的方法创建 java 对象。  
1.用 new 语句创建对象,这是最常用的创建对象的方式。  
2.调用对象的 clone()方法。  
MyObject anotherObject = new MyObject();  
MyObject object = anotherObject.clone();  
使用 clone()方法克隆一个对象的步骤:  
1.被克隆的类要实现 Cloneable 接口。  
2.被克隆的类要重写 clone()方法。

* 扩展:   
       原型模式主要用于对象的复制,实现一个接口(实现 Cloneable 接口), 重写一个方法(重写 Object 类中的 clone 方法),即完成了原型模式。   
  原型模式中的拷贝分为"浅拷贝"和"深拷贝":   
  浅拷贝: 对值类型的成员变量进行值的复制,对引用类型的成员变量只复制 引用,不复制引用的对象.   
  深拷贝: 对值类型的成员变量进行值的复制,对引用类型的成员变量也进行 引用对象的复制.   
  (Object 类的 clone 方法只会拷贝对象中的基本数据类型的值,对于数组、 容器对象、引用对象等都不会拷贝,这就是浅拷贝。如果要实现深拷贝,必须 将原型模式中的数组、容器对象、引用对象等另行拷贝。)   
  原型模式的优点。   
  1.如果创建新的对象比较复杂时,可以利用原型模式简化对象的创建过程。   
  2.使用原型模式创建对象比直接 new 一个对象在性能上要好的多,因为 Object 类的 clone 方法是一个本地方法,它直接操作内存中的二进制流,特别 是复制大对象时,性能的差别非常明显。   
  原型模式的使用场景。   
  因为以上优点,所以在需要重复地创建相似对象时可以考虑使用原型模式。 比如需要在一个循环体内创建对象,假如对象创建过程比较复杂或者循环次数 很多的话,使用原型模式不但可以简化创建过程,而且可以使系统的整体性能 提高很多。   
  3.运用反射手段, 使用 Class.forName()   
  MyObject object =(MyObject)Class.forName("subin.rnd.MyObject ").newInstance();   
  其他部分见“反射的原理”部分。   
  4.运用反序列化手段,调用 java.io.ObjectInputStream 对象的 readObject()方法。

### **9.说一下几种垃圾收集算法的原理和特点,应用的场景。怎么优化复制算法?**

优化复制算法:由于每次执行复制算法的时候,所有存活的对象都要被复 制,这样效率很低。由于程序中创建的大部分对象的生命周期都很短,只有一 部分对象有较长的生命周期,因此可以针对这个特点对复制算法进行优化,采 用分代垃圾回收算法。

### **什么是内存泄露和内存溢出。**

内存泄漏的典型例子是一个没有重写 hashCode 和 equals 方法的 Key 类在 HashMap 中保存的情况,最后会生成很多重复的对象。所有的内存泄露最后都 会抛出OutOfMemoryError异 常 ( Exceptionjava.lang.OutOfMemoryError: Java heap space)。

内存泄露的解决方案(重要!!):   
1、避免在循环中创建对象。   
2、尽早释放无用对象的引用。(最基本的建议)   
3、尽量少用静态变量,因为静态变量存放在永久代(方法区),永久代基本不参与垃圾回收。   
4、使用字符串处理,避免使用 String,应大量使用 StringBuffer,每一个 String对象都得独立占用内存一块区域。

在实际场景中,你怎么查找内存泄露?  
可以使用 Jconsole。

内存溢出:指程序运行过程中无法申请到足够的内存而导致的一种错误。

导致内存溢出的原因:   
1.内存中加载的数据量过于庞大,如一次从数据库取出过多数据;   
2.集合类中有对对象的引用,使用完后未清空,使得 JVM 不能回收;   
3.代码中存在死循环或循环产生过多重复的对象实体;   
4.启动参数内存值设定的过小。

内存溢出的解决方法:   
第一步,修改 JVM 启动参数,直接增加内存。(-Xms,-Xmx 参数一定不要忘记加。   
一般要将-Xms 和-Xmx 选项设置为相同,以避免在每次 GC 后调整堆的大小;建议堆的最大值设置为可用内存的最大值的 80%)。 第二步,检查错误日志,查看“OutOfMemory”错误前是否有其它异常或错误。   
第三步,对代码进行走查和分析,找出可能发生内存溢出的位置。   
第四步,使用内存查看工具动态查看内存使用情况(Jconsole)。

### **11. 如何减少GC出现的次数(Java内存管理)。(重点!!!!)**

(1)对象不用时最好显式置为 Null

(2)尽量少用 System.gc()

(3)尽量少用静态变量   静态变量属于全局变量,不会被 GC 回收,它们会一直占用内存。

(4)尽量使用 StringBuffer,而不用 String 来累加字符串。

### **12.JVM 常见的启动参数。**

     -Xms: 设置堆的最小值。  
     -Xmx: 设置堆的最大值。  
     -Xmn: 设置新生代的大小。  
     -Xss: 设置每个线程的栈大小。  
     -XX:NewSize: 设置新生代的初始值。  
     -XX:MaxNewSize: 设置新生代的最大值。  
     -XX:PermSize: 设置永久代的初始值。  
     -XX:MaxPermSize: 设置永久代的最大值。  
     -XX:SurvivorRatio: 年轻代中 Eden 区与 Survivor 区的大小比值。  
     -XX:PretenureSizeThreshold: 令大于这个设置值的对象直接在老年代分配。

### **13.说下几种常用的内存调试工具: jps、 jmap、 jhat、 jstack、 jconsole,jstat。**

Java 内存泄露的问题调查定位: jmap,jstack 的使用等等。  
     jps: 查看虚拟机进程的状况,如进程 ID。  
     jmap: 用于生成堆转储快照文件(某一时刻的)。  
     jhat: 对生成的堆转储快照文件进行分析。  
     jstack: 用来生成线程快照(某一时刻的)。生成线程快照的主要目的是定位线程长时停顿的原因(如死锁,死循环,等待 I/O 等), 通过查看各个线程的调用堆栈,就可以知道没有响应的线程在后台做了什么或者等待什么资源。   
     jstat: 虚拟机统计信息监视工具。如显示垃圾收集的情况,内存使用的情况。  
     Jconsole: 主要是内存监控和线程监控。内存监控:可以显示内存的使用情况。线程监控:遇到线程停顿时,可以使用这个功能。

### **14.new 的对象如何不分配在堆而分配在栈上?**

1.逃逸分析；2在方法内部new对象