1.介绍下Java内存区域（运行时数据区）。

Java虚拟机在执行Java程序的过程中会把它所管理的内存划分为以下6个运行时数据区域。

程序计数器（Program Counter Register）

一块较小的内存空间，可以看作当前线程所执行的字节码的行号指示器。如果线程正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Native方法，这个计数器值则为空。

Java虚拟机栈（Java Virtual Machine Stacks）

与程序计数器一样，Java虚拟机栈也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

本地方法栈（Native Method Stack）

本地方法栈与虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的Native方法服务。

Java堆（Java Heap）

对大多数应用来说，Java堆是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。

方法区（Method Area）

与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。方法区是JVM规范中定义的一个概念，具体放在哪里，不同的实现可以放在不同的地方。

运行时常量池（Runtime Constant Pool）

运行时常量池是方法区的一部分。Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

2.怎么判定对象已经“死去”?

常见的判定方法有两种：引用计数法和可达性分析算法，HotSpot中采用的是可达性分析算法。

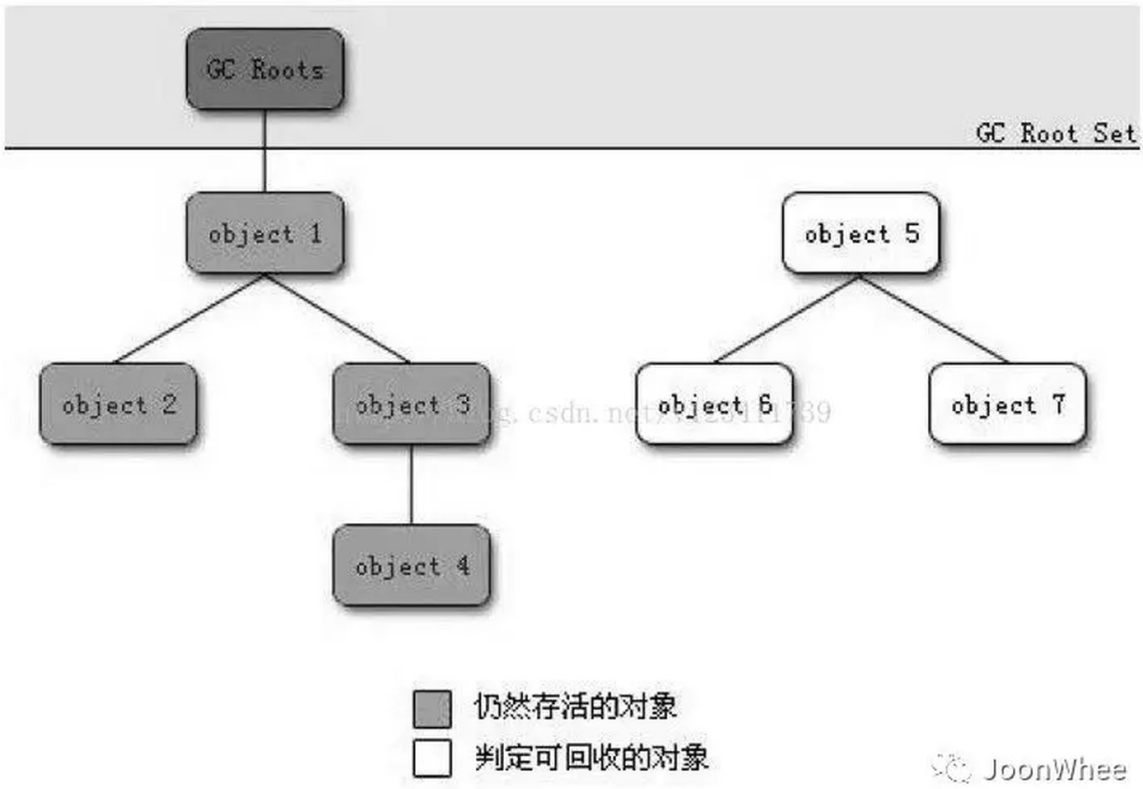
引用计数法

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，但是主流的Java虚拟机里面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相互循环引用的问题。

可达性分析算法

这个算法的基本思路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，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。如下图所示，对象object 5、object 6、object 7虽然互相有关联，但是它们到GC Roots是不可达的，所以它们将会被判定为是可回收的对象。



3.介绍下四种引用（强引用、软引用、弱引用、虚引用）？

强引用：在程序代码之中普遍存在的，类似“Object obj=new Object()”这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。

软引用：用来描述一些还有用但并非必需的对象，使用SoftReference类来实现软引用，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。

弱引用：用来描述非必需对象的，使用WeakReference类来实现弱引用，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。

虚引用：是最弱的一种引用关系，使用PhantomReference类来实现虚引用，一个对象是否有虚引用的存在，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。

4.垃圾收集有哪些算法，各自的特点?

标记 - 清除算法

首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。它的主要不足有两个：一个是效率问题，标记和清除两个过程的效率都不高；另一个是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

复制算法

为了解决效率问题，一种称为“复制”（Copying）的收集算法出现了，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为了原来的一半，未免太高了一点。

标记 - 整理算法

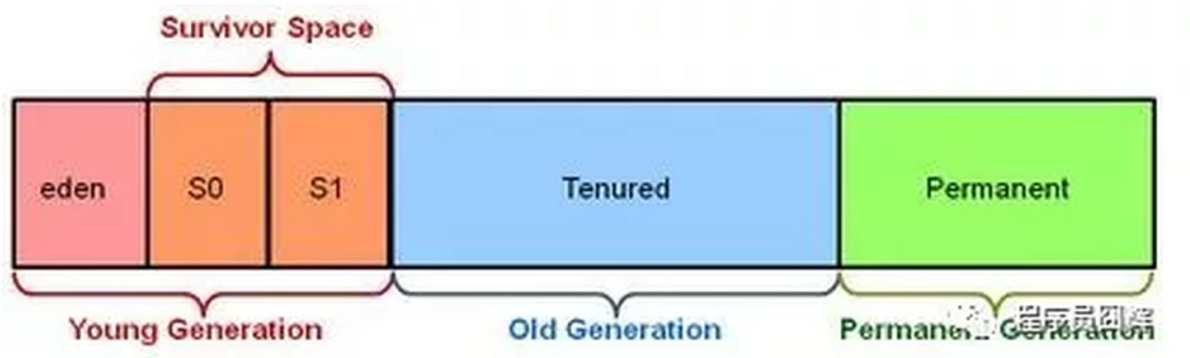
复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）算法，标记过程仍然与“标记-清除”算法一样，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。

5.HotSpot为什么要分为新生代和老年代?

HotSpot根据对象存活周期的不同将内存划分为几块，一般是把Java堆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适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记—清理”或者“标记—整理”算法来进行回收。

其中新生代又分为1个Eden区和2个Survivor区，通常称为From Survivor和To Survivor区。



6.新生代中Eden区和Survivor区的默认比例?

在HotSpot虚拟机中，Eden区和Survivor区的默认比例为8:1:1，即-XX:SurvivorRatio=8，其中Survivor分为From Survivor和ToSurvivor，因此Eden此时占新生代空间的80%。

7.HotSpot GC的分类?

针对HotSpot VM的实现，它里面的GC其实准确分类只有两大种：

1. **Partial GC：**并不收集整个GC堆的模式，具体如下：
    1. Young GC/Minor GC：只收集新生代的GC。
    2. Old GC：只收集老年代的GC。只有CMS的concurrent collection是这个模式。
    3. Mixed GC：收集整个新生代以及部分老年代的GC，只有G1有这个模式。

2. **Full GC/Major GC**：收集整个GC堆的模式，包括新生代、老年代、永久代（如果存在的话）等所有部分的模式。

8.HotSpot GC的触发条件?

这里只说常见的Young GC和Full GC。

Young GC：当新生代中的Eden区没有足够空间进行分配时会触发Young GC。

Full GC：

1. 当准备要触发一次Young GC时，如果发现统计数据说之前Young GC的平均晋升大小比目前老年代剩余的空间大，则不会触发Young GC而是转为触发Full GC。（通常情况）
2. 如果有永久代的话，在永久代需要分配空间但已经没有足够空间时，也要触发一次Full GC。
3. System.gc()默认也是触发Full GC。
4. heap dump带GC默认也是触发Full GC。
5. CMS GC时出现Concurrent Mode Failure会导致一次Full GC的产生。

9.Full GC后老年代的空间反而变小?

HotSpot的Full GC实现中，默认新生代里所有活的对象都要晋升到老年代，实在晋升不了才会留在新生代。假如做Full GC的时候，老年代里的对象几乎没有死掉的，而新生代又要晋升活对象上来，那么Full GC结束后老年代的使用量自然就上升了。

10.什么情况下新生代对象会晋升到老年代?

1. 如果新生代的垃圾收集器为Serial和ParNew，并且设置了-XX:PretenureSizeThreshold参数，当对象大于这个参数值时，会被认为是大对象，直接进入老年代。
2. Young GC后，如果对象太大无法进入Survivor区，则会通过分配担保机制进入老年代。
3. 对象每在Survivor区中“熬过”一次Young GC，年龄就增加1岁，当它的年龄增加到一定程度（默认为15岁，可以通过-XX:MaxTenuringThreshold设置），就将会被晋升到老年代中。
4. 如果在Survivor区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，无须等到MaxTenuringThreshold中要求的年龄。

11.介绍下垃圾收集机制（在什么时候，对什么，做了什么）？

在什么时候?

在触发GC的时候，具体如下，这里只说常见的Young GC和Full GC。

触发Young GC：当新生代中的Eden区没有足够空间进行分配时会触发Young GC。

触发Full GC：

1. 当准备要触发一次Young GC时，如果发现统计数据说之前Young GC的平均晋升大小比目前老年代剩余的空间大，则不会触发Young GC而是转为触发Full GC。（通常情况）
2. 如果有永久代的话，在永久代需要分配空间但已经没有足够空间时，也要触发一次Full GC。
3. System.gc()默认也是触发Full GC。
4. heap dump带GC默认也是触发Full GC。
5. CMS GC时出现Concurrent Mode Failure会导致一次Full GC的产生。

对什么?

对那些JVM认为已经“死掉”的对象。即从GC Root开始搜索，搜索不到的，并且经过一次筛选标记没有复活的对象。

做了什么?

对这些JVM认为已经“死掉”的对象进行垃圾收集，新生代使用复制算法，老年代使用标记-清除和标记-整理算法。

12.GC Root有哪些?

在Java语言中，可作为GC Roots的对象包括下面几种：

- 虚拟机栈（栈帧中的本地变量表）中引用的对象。
- 方法区中类静态属性引用的对象。
- 方法区中常量引用的对象。
- 本地方法栈中JNI（即一般说的Native方法）引用的对象。

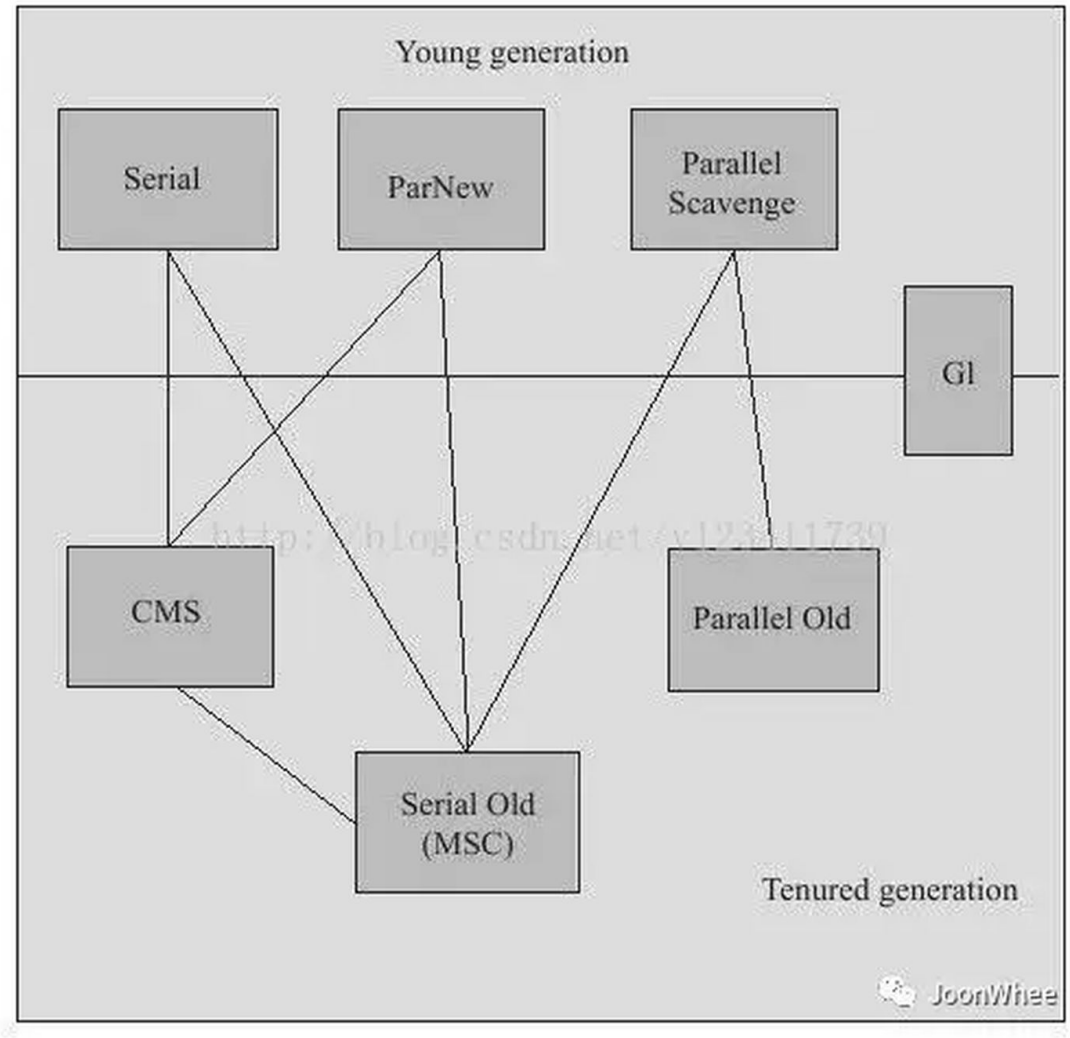
13.发生Young GC的时候需要扫描老年代的对象吗?

在分代收集中，新生代的规模一般都比老年代要小许多，新生代的收集也比老年代要频繁许多，如果回收新生代时也不得不同时扫描老年代的话，那么Young GC的效率可能下降不少。显然是不可能区扫描老年代的，那么是通过什么办法来解决这个问题了?

在大多垃圾收集器中（G1有不同的地方），通过CardTable来维护老年代对年轻代的引用，CardTable可以说是Remembered Set（RS）的一种特殊实现，是Card的集合。Card是一块2的幂字节大小的内存区域，例如HotSpot用512字节，里面可能包含多个对象。CardTable要记录的是从它覆盖的范围出发指向别的范围的指针。以分代式GC的CardTable为例，要记录老年代指向年轻代的跨代指针，被标记的Card是老年代范围内的。当进行年轻代的垃圾收集时，只需要扫描年轻代和老年代的CardTable即可保证不对全堆扫描也不会有遗漏。CardTable通常为字节数组，由Card的索引（即数组下标）来标识每个分区的空间地址。

14.垃圾收集器有哪些?

目前HotSpot中有7种作用于不同分代的收集器，如下图所示，如果两个收集器之间存在连线，就说明它们可以搭配使用。



15.介绍CMS垃圾收集器的特点?

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS收集器就非常符合这类应用的需求。

从名字(包含“Mark Sweep”)上就可以看出，CMS收集器是基于“标记—清除”算法实现的，它的运作过程可以分为6个步骤，包括：初始标记、并发标记、预处理、重新标记、并发清除、重置。

CMS是一款优秀的收集器，它的主要优点在名字上已经体现出来了：并发收集、低停顿，但是CMS还远达不到完美的程度，它有以下3个明显的缺点：

1. CMS收集器对CPU资源非常敏感。
2. CMS收集器无法处理浮动垃圾(Floating Garbage)，可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。
3. CMS是一款基于“标记—清除”算法实现的收集器，这意味着收集结束时会有大量空间碎片产生。

了解CMS更多内容，查看我的另一篇文章：

JoonWhee：Java虚拟机：垃圾收集器2 赞同 · 0 评论文章

16.介绍下G1垃圾收集器的特点？（较复杂，可以考虑跳过）

G1(Garbage-First)收集器是当今收集器技术发展的最前沿成果之一。G1是一款面向服务端应用的垃圾收集器。与其他GC收集器相比，G1具备如下特点：并行与并发、分代收集、空间整合、可预测的停顿。

在G1之前的其他收集器进行收集的范围都是整个新生代或者老年代，而G1不再是这样。使用G1收集器时，Java堆的内存布局就与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域，虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分Region（不需要连续）的集合。

G1收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划地避免在整个Java堆中进行全区域的垃圾收集。G1跟踪各个Region里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region（这也就是Garbage-First名称的来由）。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了G1收集器在有限的时间内可以获取尽可能高的收集效率。

Mixed GC是G1垃圾收集器特有的收集方式，Mixed GC大致可划分为**全局并发标记（global concurrent marking）**和**拷贝存活对象（evacuation）**两个大部分：

global concurrent marking是基于SATB形式的并发标记，包括以下4个阶段：初始标记（Initial Marking）、并发标记（Concurrent Marking）、最终标记（Final Marking）、清理（Clean Up）。Evacuation阶段是全暂停的。它负责把一部分region里的活对象拷贝到空region里去，然后回收原本的region的空间。

了解G1更多内容，查看我的另一篇文章：

JoonWhee：Java虚拟机：垃圾收集器2 赞同 · 0 评论文章

17.类加载的过程。

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载7个阶段。其中验证、准备、解析3个部分统称为连接。

加载：

“类加载”过程的一个阶段，在加载阶段，虚拟机需要完成以下3件事情：

1. 通过一个类的全限定名来获取定义此类的二进制字节流。
2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
3. 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访问入口。

验证：

连接阶段的第一步，这一阶段的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。从整体上看，验证阶段大致上会完成下面4个阶段的检验动作：文件格式验证、元数据验证、字节码验证、符号引用验证。

准备：

该阶段是正式为类变量（static修饰的变量）分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配。这里所说的初始值“通常情况”下是数据类型的零值，下表列出了Java中所有基本数据类型的零值。

数据类型	零值	数据类型	零值
int	0	boolean	false
long	0L	float	0.0f
short	(short)0	double	0.0d
char	'\u0000'	reference	null
byte	(byte)0		程序员囧辉

解析：

该阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

初始化：

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所有类变量（static修饰的变量）的赋值动作和静态语句块（static{}块）中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的。如果该类存在父类，则虚拟机会保证在执行子类的()方法前，父类的()方法已经执行完毕。因此在虚拟机中第一个被执行()方法的类肯定是java.lang.Object。

18.Java虚拟机中有哪些类加载器?

从Java虚拟机的角度来讲，只存在两种不同的类加载器：一种是启动类加载器（Bootstrap ClassLoader），这个类加载器使用C++语言实现，是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类加载器都由Java语言实现，独立于虚拟机外部，并且全都继承自抽象类java.lang.ClassLoader。

从Java开发人员的角度来看，绝大部分Java程序都会使用到以下3种系统提供的类加载器。

启动类加载器（Bootstrap ClassLoader）：

这个类加载器负责将存放在<JAVA_HOME>\lib目录中的，或者被-Xbootclasspath参数所指定的路径中的，并且是虚拟机识别的（仅按照文件名识别，如rt.jar，名字不符合的类库即使放在lib目录中也不会被加载）类库加载到虚拟机内存中。

扩展类加载器（Extension ClassLoader）：

这个加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载<JAVA_HOME>\lib\ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。

应用程序类加载器（Application ClassLoader）：

这个类加载器由sun.misc.Launcher$AppClassLoader实现。由于这个类加载器是ClassLoader中的getSystemClassLoader()方法的返回值，所以一般也称它为系统类加载器。它负责加载用户类路径（ClassPath）上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

我们的应用程序都是由这3种类加载器互相配合进行加载的，如果有必要，还可以加入自己定义的类加载器。这些类加载器之间的关系一般如图所示。

19.什么是双亲委派模型?

如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

20.使用双亲委派模型的好处?

使用双亲委派模型来组织类加载器之间的关系，有一个显而易见的好处就是Java类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类java.lang.Object，它存放在rt.jar之中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己编写了一个称为java.lang.Object的类，并放在程序的ClassPath中，那系统中将会出现多个不同的Object类，Java 类型体系中最基础的行为也就无法保证，应用程序也将会变得一片混乱。