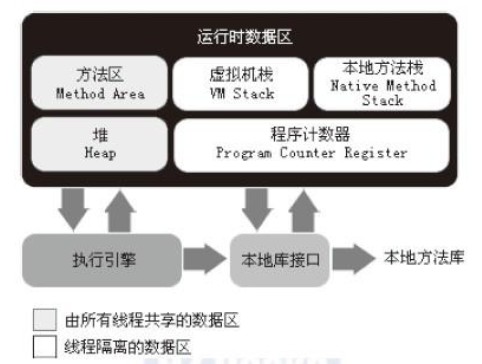
**1. 内存模型以及分区，需要详细到每个区放什么。**

答案：

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若干个不同的数据区域。这些区域都有各自的用途，以及创建和销毁的时间，有的区域随着虚拟机进程的启动而存在，有些区域则是依赖用户线程的启动和结束而建立和销毁。根据《Java虚拟机规范（第2版）》的规定，Java虚拟机所管理的内存将会包括以下几个运行时数据区域，如下图所示：



**1、程序计数器（线程私有）**

程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。 **由于Java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间的计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。** 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Natvie方法，这个计数器值则为空（Undefined）。**此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。**

**2、Java虚拟机栈（线程私有）**

与程序计数器一样，Java虚拟机栈（Java Virtual Machine Stacks）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：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）**用于存储局部变量表、操作栈、动态链接、方法出口等**信息。**每一个方法被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从入栈到出栈的过程。**

　　局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference类型），它不等同于对象本身，根据不同的虚拟机实现，它可能是一个指向对象起始地址的引用指针，也可能指向一个代表对象的句柄或者其他与此对象相关的位置）和returnAddress类型（指向了一条字节码指令的地址）。

3、**本地方法栈（线程私有）**

本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其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，而本地方法栈则是为虚拟机使用到的Native方法服务。虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机（譬如Sun HotSpot虚拟机）直接就把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出StackOverflowError和OutOfMemoryError异常。

4、**Java堆（共享）**

　　对于大多数应用来说，Java堆（Java Heap）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。这一点在Java虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配，但是随着JIT编译器的发展与逃逸分析技术的逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也渐渐变得不是那么“绝对”了。

　　Java堆是垃圾收集器管理的主要区域，因此很多时候也被称做“GC堆”（Garbage Collected Heap，幸好国内没翻译成“垃圾堆”）。如果从内存回收的角度看，由于现在收集器基本都是采用的**分代收集算法**，所以Java堆中还可以细分为：新生代和老年代；再细致一点的有Eden空间、From Survivor空间、To Survivor空间等。如果**从内存分配的角度看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区**（Thread Local Allocation Buffer，TLAB）。不过，无论如何划分，都与存放内容无关，无论哪个区域，存储的都仍然是对象实例，进一步划分的目的是为了更好地回收内存，或者更快地分配内存。在本章中，我们仅仅针对内存区

域的作用进行讨论，Java堆中的上述各个区域的分配和回收等细节将会是下一章的主题。

根据Java虚拟机规范的规定，Java堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘空间一样。在实现时，既可以实现成固定大小的，也可以是可扩展的，不过当前主流的虚拟机都是按照可扩展来实现的（通过-Xmx和-Xms控制）。如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常。

**5、方法区（共享）**

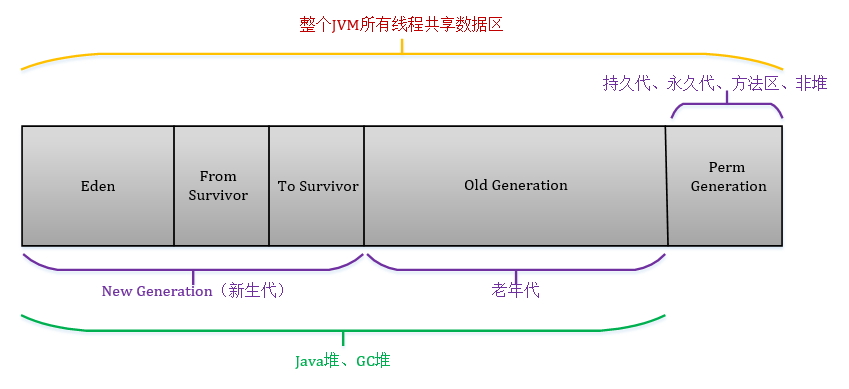
方法区（Method Area）与Java堆一样，是各个线程共享的内存区域，**它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据**。虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做Non-Heap（非堆），目的应该是与Java堆区分开来。

　　Java虚拟机规范对这个区域的限制非常宽松，除了和Java堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入了方法区就如永久代的名字一样“永久”存在了。这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载，一般来说这个区域的回收“成绩”比较难以令人满意，尤其是类型的卸载，条件相当苛刻，但是这部分区域的回收确实是有必要的。在Sun公司的BUG列表中，曾出现过的若干个严重的BUG就是由于低版本的HotSpot虚拟机对此区域未完全回收而导致内存泄漏。 根据Java虚拟机规范的规定，当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。

**2. 堆里面的分区：Eden，survival from to，老年代，各自的特点。**

答案：

在JVM中共享数据空间划分如下图所示



上图中，刻画了Java程序运行时的堆空间,可以简述成如下4条

1.JVM中共享数据空间可以分成三个大区，新生代（Young Generation）、老年代（Old Generation）、永久代（Permanent Generation），其中JVM堆分为新生代和老年代

2.新生代可以划分为三个区，Eden区（存放新生对象），两个幸存区（From Survivor和To Survivor）（存放每次垃圾回收后存活的对象）

3.永久代管理class文件、静态对象、属性等（JVM uses a separate region of memory, called the Permanent Generation (or**PermGen** for short), to hold internal representations of java classes. PermGen is also used to store more information ）

4.JVM垃圾回收机制采用“分代收集”：新生代采用复制算法，老年代采用标记清理算法。

**3. 对象创建方法，对象的内存分配，对象的访问定位。**

（1）对象的创建方法：new的过程

**第一：进行类加载检查**。当遇到一个new指令，首先检查能否在方法区的常量池中能否定位到这个类的符号引用，并且检查类有没有进行加载、解析和初始化；

**第二：分配空间。**有两种常见的分配方式，一是指针碰撞，二是空闲列表，分别针对连续分配内存和不连续的，有空隙的，取决于虚拟机是否会压缩整理。内存分配的大小是在类加载完成之后就已经确定的，但是分配的时候修改指针的指向位置应该是线程安全的（栈上的Reference），第一种方式就保证原子性；第二种是给每个线程分配自己的一小块内存，成为本地线程分配缓冲(TLAB)，每个线程在自己的TLAB是哪个分配。

**第三：初始化。**将分配的内存初始化为0值。

**第四：基本设置。**进行基本的设置，确定这个对象是哪个类的实例，对象的HASH码，对象的年龄等等。

（2）对象的内存布局

对象在内存中的存储的布局可以分为3块区域：

第一：对象头（Header）

对象头包含两个部分的信息，第一部分是对象自身的运行时数据，如哈希码、GC分代年龄、持有的锁等等；第二部分是类型指针，指向它的类元数据的指针，通过这个虚拟机来确定这个对象是哪个类的实例。

第二：实例数据（Instance Data）

对象真正存储的数据，就是程序代码中定义的字段内容。

第三：对齐填充（Padding）

用于使对象的开头必须是8字节的整数倍，无特殊意义。

（3）对象的访问定位

对于这句代码：

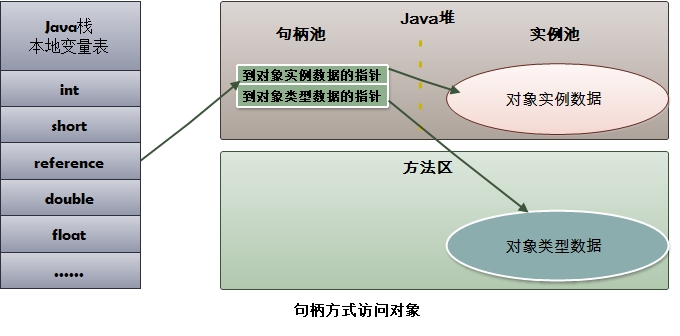
1. Object objectRef = **new** Object();

Object objectRef  这部分将会反映到Java栈的本地变量中，作为一个reference类型数据出现。而“new Object()”这部分将会反映到Java堆中，形成一块存储Object类型所有实例数据值的结构化内存，根据具体类型以及虚拟机实现的对象内存布局的不同，这块内存的长度是不固定。另外，在java堆中还必须包括能查找到此对象类型数据（如对象类型、父类、实现的接口、方法等）的地址信息，这些数据类型存储在方法区中。

有两种基本的定位方式：

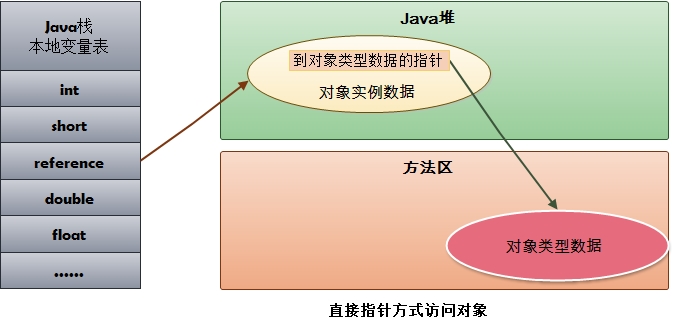
**第一：句柄访问（间接）**

在Java堆中划分一块内存作为句柄池（即一个句柄列表），reference中存储的就是对象在句柄池中的地址，得到了句柄池的地址就可以知道对象的实例数据和类型数据的位置。



**第二：直接指针访问（直接）**

reference中存储的直接就是对象的实例数据的地址，而实例数据中自己有一个指针存储对象类型数据的地址（方法区中），不需要reference来存储。



**4. GC的两种判定方法：引用计数与引用链。**

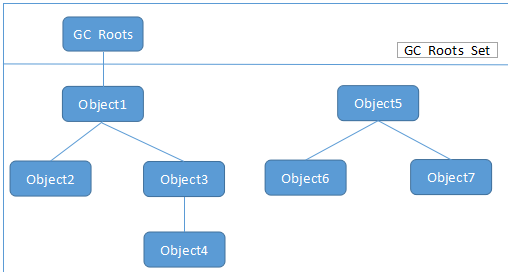
答案：

（1）引用计数

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，计数器值加1；当引用失效时，计数器减1；任何时刻计数器都为0的对象就是不可能再被使用的。

引用计数算法的实现简单，判断效率也很高，在大部分情况下它都是一个不错的算法。但是Java语言中没有选用引用计数算法来管理内存，其中最主要的一个原因是它很难解决对象之间相互循环引用的问题。

（2）可达性分析（引用链）

在主流的商用程序语言中(Java和C#)，都是使用可达性分析算法判断对象是否存活的。这个算法的基本思路就是通过一系列名为"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时，则证明此对象是不可用的，下图对象object5, object6, object7虽然有互相判断，但它们到GC Roots是不可达的，所以它们将会判定为是可回收对象。  
  
  在Java语言里，可作为GC Roots对象的包括如下几种：  
    a.虚拟机栈(栈桢中的本地变量表)中的引用的对象  
    b.方法区中的类静态属性引用的对象  
    c.方法区中的常量引用的对象  
    d.本地方法栈中JNI的引用的对象

**5. GC的三种收集方法：标记清除、标记整理、复制算法的原理与特点，分别用在什么地方，如果让你优化收集方法，有什么思路？**

（1）**标记清除算法**

分为标记和清除两个阶段，首先标记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。它主要有两个问题：一个是效率问题，标记清除的效率较低；第二个是碎片问题，清除后会产生大量不连续的内存碎片。

（2）**复制算法**—回收新生代，回收比例较大

原理就是把内存划分为大小相等的两块，每次只使用其中的一块。当这一块上的内存用完了，就把还存活的对象（可达性分析）复制到另一块上，然后把已使用过的内存空间一次性清理掉，这样每次都不用考虑内存碎片的情况，并且实现的更加简单。如果存活的比例很低，那么复制的操作就很小，效率会比较高。只是如果一次只使用其中一半，那么代价太大了。

现在主要都是用于新生代的回收，98%的新生代对象都是很快就不用的，所以只需要把内存分为一块较大的Eden空间和两块Survivor空间，每次只使用Eden和一块Survivor空间，当回收时把两个区域存活的对象复制到那一块剩余的Survivor上。一般比例为8:1，这样浪费的空间只有10%。如果Survivor内存不够，就是用老年代的内存进行担保。

（3）**标记整理算法**—回收老年代，回收比例较小

对于老年代，存活的比例一般会比较高，如果使用复制算法那么复制操作的效率就会比较低。根据老年代的特点，提出了标记整理算法，先标记需要回收的对象；第二部不是进行回收，而是将存活对象都移动向一端，然后直接清除掉边界外的内存。

（4）**分代收集算法**—常见虚拟机的方式

分代收集算法就是综合2、3，对于堆中的不同区域使用不同的收集算法。

**6. GC收集器有哪些？CMS收集器与G1收集器的特点。**

**（1）Serial收集器**

**Serial收集器是最基本的收集器，这是一个单线程收集器，它“单线程”的意义不仅仅是说明它只用一个线程去完成垃圾收集工作，更重要的是在它进行垃圾收集工作时，必须暂停其他工作线程，直到它收集完成**。Sun将这件事称之为”Stop the world“。

没有一个收集器能完全不停顿，只是停顿的时间长短。

虽然Serial收集器的缺点很明显，但是它仍然是JVM在Client模式下的默认新生代收集器。它有着优于其他收集器的地方：简单而高效（与其他收集器的单线程比较），Serial收集器由于没有线程交互的开销，专心只做垃圾收集自然也获得最高的效率。在用户桌面场景下，分配给JVM的内存不会太多，停顿时间完全可以在几十到一百多毫秒之间，只要收集不频繁，这是完全可以接受的。

**（2）ParNew收集器**

ParNew是Serial的多线程版本，在回收算法、对象分配原则上都是一致的。ParNew收集器是许多运行在Server模式下的默认新生代垃圾收集器，其主要在于除了Serial收集器，目前只有ParNew收集器能够与CMS收集器配合工作。

**（3）Parallel Scavenge收集器**

Parallel Scavenge收集器是一个新生代垃圾收集器，其使用的算法是**复制算法**，也是并行的多线程收集器。

Parallel Scavenge 收集器更关注可控制的吞吐量，吞吐量等于运行用户代码的时间/(运行用户代码的时间+垃圾收集时间)。直观上，只要最大的垃圾收集停顿时间越小，吞吐量是越高的，但是GC停顿时间的缩短是以牺牲吞吐量和新生代空间作为代价的。比如原来10秒收集一次，每次停顿100毫秒，现在变成5秒收集一次，每次停顿70毫秒。停顿时间下降的同时，吞吐量也下降了。

停顿时间越短就越适合需要与用户交互的程序；而高吞吐量则可以最高效的利用CPU的时间，尽快的完成计算任务，主要适用于后台运算。

**（4）Serial Old收集器**

Serial Old收集器是Serial收集器的老年代版本，也是一个单线程收集器，采用“标记-整理算法”进行回收。其运行过程与Serial收集器一样。

**（5）Parallel Old收集器**

Parallel Old收集器是Parallel Scavenge收集器的老年代版本，使用多线程和标记-整理算法进行垃圾回收。其通常与Parallel Scavenge收集器配合使用，“吞吐量优先”收集器是这个组合的特点，在注重吞吐量和CPU资源敏感的场合，都可以使用这个组合。

**（6）CMS 收集器**

CMS（Concurrent Mark Sweep）收集器是一种以获取最短停顿时间为目标的收集器，CMS收集器采用标记--清除算法，运行在老年代。主要包含以下几个步骤：

* 初始标记
* 并发标记
* 重新标记
* 并发清除

其中初始标记和重新标记仍然需要“Stop the world”。初始标记仅仅标记GC Root能直接关联的对象，并发标记就是进行GC Root Tracing过程，而重新标记则是为了修正并发标记期间，因用户程序继续运行而导致标记变动的那部分对象的标记记录。

由于整个过程中最耗时的并发标记和并发清除，收集线程和用户线程一起工作，所以总体上来说，CMS收集器回收过程是与用户线程并发执行的。虽然CMS优点是并发收集、低停顿，很大程度上已经是一个不错的垃圾收集器，但是还是有三个显著的缺点：

* **CMS收集器对CPU资源很敏感**。在并发阶段，虽然它不会导致用户线程停顿，但是会因为占用一部分线程（CPU资源）而导致应用程序变慢。
* **CMS收集器不能处理浮动垃圾**。所谓的“浮动垃圾”，就是在并发标记阶段，由于用户程序在运行，那么自然就会有新的垃圾产生，这部分垃圾被标记过后，CMS无法在当次集中处理它们，只好在下一次GC的时候处理，这部分未处理的垃圾就称为“浮动垃圾”。也是由于在垃圾收集阶段程序还需要运行，即还需要预留足够的内存空间供用户使用，因此CMS收集器不能像其他收集器那样等到老年代几乎填满才进行收集，需要预留一部分空间提供并发收集时程序运作使用。要是CMS预留的内存空间不能满足程序的要求，这是JVM就会启动预备方案：临时启动Serial Old收集器来收集老年代，这样停顿的时间就会很长。
* **由于CMS使用标记--清除算法，所以在收集之后会产生大量内存碎片**。当内存碎片过多时，将会给分配大对象带来困难，这是就会进行Full GC。

**（6）G1收集器**

G1收集器与CMS相比有很大的改进：

* G1收集器采用标记--整理算法实现。
* 可以非常精确地控制停顿。

**G1收集器可以实现在基本不牺牲吞吐量的情况下完成低停顿的内存回收，这是由于它极力的避免全区域的回收，G1收集器将Java堆（包括新生代和老年代）划分为多个区域（Region），并在后台维护一个优先列表，每次根据允许的时间，优先回收垃圾最多的区域** 。

**7. Minor GC与Full GC分别在什么时候发生？**

由于对象进行了分代处理，因此垃圾回收区域、时间也不一样。GC有两种类型：Scavenge GC和Full GC。对于一个拥有终结方法的对象，在垃圾收集器释放对象前必须执行终结方法。但是当垃圾收集器第二次收集这个对象时便不会再次调用终结方法。

**（1）Minor GC**

一般情况下，当新对象生成，并且在Eden申请空间失败时，就会触发Minor  GC，对Eden区域进行GC，清除非存活对象，并且把尚且存活的对象移动到Survivor区，然后整理Survivor的两个区。这种方式的GC是对新生代的Eden区进行，不会影响到老年代。因为大部分对象都是从Eden区开始的，同时Eden区不会分配的很大，所以Eden区的GC会频繁进行。

**（2）Full GC**

对整个堆进行整理，包括Young、Tenured和Perm。Full GC因为需要对整个对进行回收，所以比Minor GC要慢，因此应该尽可能减少Full GC的次数。在对JVM调优的过程中，很大一部分工作就是对于FullGC的调节。有如下原因可能导致Full GC：

* 老年代（Tenured）被写满
* 持久代（Perm）被写满
* System.gc()被显示调用

**8. Java的四种引用，强弱软虚，用到的场景。**

**（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。

如下：

|  |  |
| --- | --- |
| 1 | Object o=new Object();   //  强引用 |

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。如果不使用时，要通过如下方式来弱化引用，如下：

|  |  |
| --- | --- |
| 1 | o=null;     // 帮助垃圾收集器回收此对象 |

  显式地设置o为null，或超出对象的生命周期范围，则gc认为该对象不存在引用，这时就可以回收这个对象。具体什么时候收集这要取决于gc的算法。

举例：

|  |  |
| --- | --- |
| 1  2  3  4 | public void test(){      Object o=new Object();      // 省略其他操作  } |

在一个方法的内部有一个强引用，这个引用保存在栈中，而真正的引用内容（Object）保存在堆中。当这个方法运行完成后就会退出方法栈，则引用内容的引用不存在，这个Object会被回收。

但是如果这个o是全局的变量时，就需要在不用这个对象时赋值为null，因为强引用不会被垃圾回收。

强引用在实际中有非常重要的用处，举个ArrayList的实现源代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | private transient Object[] elementData;  public void clear() {          modCount++;          // Let gc do its work          for (int i = 0; i < size; i++)              elementData[i] = null;          size = 0;  } |

在ArrayList类中定义了一个私有的变量elementData数组，在调用方法清空数组时可以看到为每个数组内容赋值为null。不同于elementData=null，强引用仍然存在，避免在后续调用 add()等方法添加元素时进行重新的内存分配。**使用如clear()方法中释放内存的方法对数组中存放的引用类型特别适用，这样就可以及时释放内存。**

**（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。

|  |  |
| --- | --- |
| 1  2 | String str=new String("abc");                                     // 强引用   SoftReference<String> softRef=new SoftReference<String>(str);     // 软引用 |

  当内存不足时，等价于：

|  |  |
| --- | --- |
| 1  2  3  4 | If(JVM.内存不足()) {     str = null;  // 转换为软引用     System.gc(); // 垃圾回收器进行回收  } |

软引用在实际中有重要的应用，例如浏览器的后退按钮。按后退时，这个后退时显示的网页内容是重新进行请求还是从缓存中取出呢？这就要看具体的实现策略了。

（1）如果一个网页在浏览结束时就进行内容的回收，则按后退查看前面浏览过的页面时，需要重新构建

（2）如果将浏览过的网页存储到内存中会造成内存的大量浪费，甚至会造成内存溢出

这时候就可以使用软引用

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Browser prev = new Browser();               // 获取页面进行浏览  SoftReference sr = new SoftReference(prev); // 浏览完毕后置为软引用  if(sr.get()!=null){      rev = (Browser) sr.get();           // 还没有被回收器回收，直接获取  }else{      prev = new Browser();               // 由于内存吃紧，所以对软引用的对象回收了      sr = new SoftReference(prev);       // 重新构建  } |

这样就很好的解决了实际的问题。

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。

**3、弱引用**

如果一个对象只具有弱引用，那就类似于可有可无的生活用品。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。**当你想引用一个对象，但是这个对象有自己的生命周期，你不想介入这个对象的生命周期，这时候你就是用弱引用。这个引用不会在对象的垃圾回收判断中产生任何附加的影响。比如说Thread中保存的ThreadLocal的全局映射，因为我们的Thread不想在ThreadLocal生命周期结束后还对其造成影响，所以应该使用弱引用，这个和缓存没有关系，只是为了防止内存泄漏所做的特殊操作。**

**4、幽灵引用(虚引用)**

**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后，把这个虚引用加入到与之关联的引用队列中。**程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否被垃圾回收。如果程序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回收后采取必要的行动。**由于Object.finalize()方法的不安全性、低效性，常常**使用虚引用完成对象回收后的资源释放工作**。当你创建一个虚引用时要传入一个引用队列，如果引用队列中出现了你的虚引用，说明它已经被回收，那么你可以在其中做一些相关操作，主要是实现细粒度的内存控制。比如**监视缓存，当缓存被回收后才申请新的缓存区。**

**9. 类加载的五个过程：加载、验证、准备、解析、初始化。**

Java类加载器是Java运行时环境（Java Runtime Environment）的一部分，负责动态加载Java类到Java虚拟机的内存空间中。**类通常是按需加载，即第一次使用该类时才加载。** 由于有了类加载器，Java运行时系统不需要知道文件与文件系统。每个Java类必须由某个类加载器装入到内存。

类装载器子系统涉及Java虚拟机的其他几个组成部分，以及几个来自java.lang库的类。比如，用户自定义的类装载器只是普通的Java对象，它的类必须派生自java.lang.ClassLoader。ClassLoader中定义的方法为程序提供了访问类装载器机制的接口。此外，对于每个被装载的类型，Java虚拟机都会为他创建一个java.lang.Class类的实例来代表该类型。和所有其他对象一样，用户自定义的类装载器以及Class类的实例都放在内存中的堆区，而装载的类型信息都位于方法区。

类装载器子系统除了要定位和导入二进制class文件外，还必须负责验证被导入类的正确性，为变量分配初始化内存，以及帮助解析符号引用。这些动作必须严格按以下顺序完成：

1. **加载--查找并装载类型的二进制数据。**
2. **链接--执行验证、准备以及解析(可选)**
   * **验证 确保被导入类型的正确性**
   * **准备 为类变量（static修饰的变量，分配在方法区中）分配内存，并将其初始化为默认值，不是你设置的值，是0值。**
   * **解析 把类型中的符号引用转换为内存直接引用。符号引用指的是一组无歧义的可以定位到目标的字面量，而内存直接引用则是直接指向目标的指针、句柄或者偏移量，是一个直接在内存中的位置。**
3. **初始化--把类变量初始化为正确的初始值，这是真正开始执行用户定义的程序代码的地方。是执行类构造器<clinit>的地方，对类变量赋值和static语句块进行合并，静态语句块之后的变量静态语句块内只能赋值不能访问。**

**10. 双亲委派模型：Bootstrap ClassLoader、Extension ClassLoader、ApplicationClassLoader。**

（1）类和类加载器

类加载器拥有自己独立的命名空间，两个类是否相等（instanceof），只有在这两个类是同一个类加载器加载的才有意义。

（2）双亲委派模型

从Java虚拟机的角度来说，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（HotSpot虚拟机中）**，是虚拟机自身的一部分；另一种就是所有其他的类加载器，**这些类加载器都有Java语言实现**，独立于虚拟机外部，**并且全部继承自抽象类java.lang.ClassLoader。**

从开发者的角度，类加载器可以细分为：

* 启动（Bootstrap）类加载器：负责将 **Java\_Home/lib**下面的类库加载到内存中（比如rt.jar）。由于引导类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作。
* 标准扩展（Extension）类加载器：是由 Sun 的 ExtClassLoader（sun.misc.Launcher$ExtClassLoader）实现的。它负责将**Java\_Home /lib/ext**或者由系统变量 **java.ext.dir指定位置中的类库**加载到内存中。开发者可以直接使用标准扩展类加载器。
* 应用程序（Application）类加载器：是由 Sun 的 AppClassLoader（sun.misc.Launcher$AppClassLoader）实现的。它负责将系统类路径（CLASSPATH）中指定的类库加载到内存中。开发者可以直接使用系统类加载器。由于这个类加载器是ClassLoader中的getSystemClassLoader()方法的返回值，因此一般称为**系统（System）加载器**。

除此之外，还有自定义的类加载器，它们之间的层次关系被称为类加载器的**双亲委派模型**。该模型要求除了顶层的启动类加载器外，其余的类加载器都应该有自己的父类加载器，而这种父子关系一般通过**组合（Composition）关系**来实现，而不是通过继承（Inheritance）。

**双亲委派模型：**

某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。**这一点是从安全角度考虑的，试想如果有人编写了一个恶意的基础类（如java.lang.String）并装载到JVM中将会引起多么可怕的后果。但是由于有了“全盘负责委托机制”，java.lang.String永远是由根装载器来装载的，这样就避免了上述事件的发生。**

**双亲委派模型的实现：**

很简单，全在loadClass中实现，在java.lang.ClassLoader的loadClass()方法中，先检查是否已经被加载过，若没有加载则调用父类加载器的loadClass()方法，若父加载器为空则默认使用启动类加载器作为父加载器。如果父加载失败，则抛出ClassNotFoundException异常后，再调用**自己的findClass()**方法进行加载。

protected synchronized Class<?> loadClass(String name,boolean resolve)throws ClassNotFoundException{

//check the class has been loaded or not

Class c = findLoadedClass(name);

if(c == null){

try{

if(parent != null){

c = parent.loadClass(name,false);

}else{

c = findBootstrapClassOrNull(name);

}

}catch(ClassNotFoundException e){

//if throws the exception ,the father can not complete the load

}

if(c == null){

c = findClass(name);

}

}

if(resolve){

resolveClass(c);

}

return c;

}

**双亲委派模型打破：**

**重写loadClass方法。**

**11. 分派：静态分派与动态分派。**

答案：

（1）在Java中，符合“编译时可知，运行时不可变”这个要求的方法主要是静态方法和私有方法。这两种方法都不能通过继承或别的方法重写，因此它们适合在类加载时进行解析。

Java虚拟机中有四种方法调用指令：

* invokestatic：调用静态方法。
* invokespecial：调用实例构造器方法，私有方法和super。
* invokeinterface：调用接口方法。
* invokevirtual：调用以上指令不能调用的方法（虚方法）。

只要能被invokestatic和invokespecial指令调用的方法，都可以在解析阶段确定唯一的调用版本，符合这个条件的有：静态方法、私有方法、实例构造器、父类方法，他们在类加载的时候就会把符号引用解析为改方法的直接引用。这些方法被称为非虚方法，反之其他方法称为**虚方法**（final方法除外）。

（2）静态和动态分派

首先明确一个概念：对于以下代码

Human man = new Man();

在这里对于创建的对象，Human是对象的静态类型，Man是对象的实际类型。静态类型是编译期间虚拟机就清楚的，而实际类型在编译期间虚拟机并不能知晓，只有在运行时才能确定。

**（1）静态分派——针对重载（Overload）**

对于重载的方法，编译器会根据参数的静态类型，在编译时就确定要调用哪一个具体的方法，也就是说和参数的实际类型没有关系（实际类型编译时也不知道），因此得名静态分派。

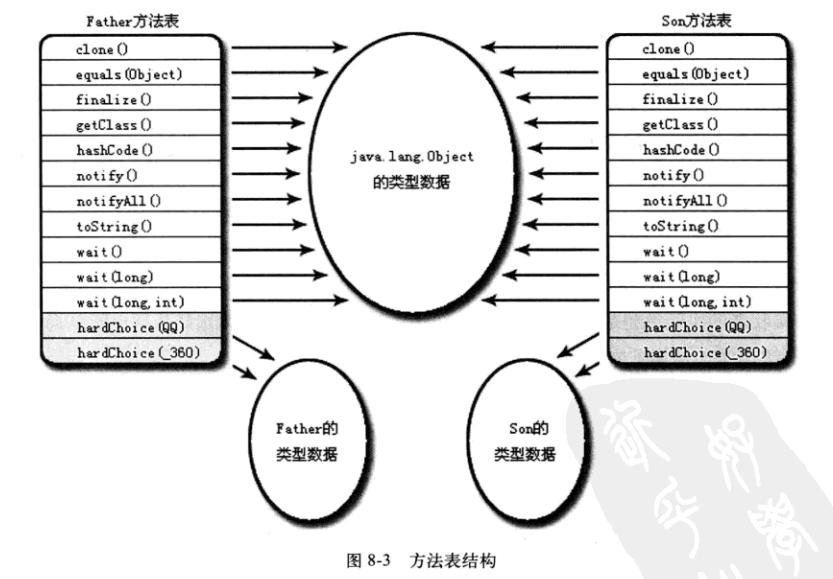
**（2）动态分派——针对重写（Override）**

动态分派与多态性有着密切的关联，可以说决定了多态的实现。对于重写的方法的编译，编译器在编译成字节码时选择的方法与调用对象的静态类型有关，比如对于Human man = new Man();，如果man.say();进行编译，编译的函数的类型是Human的，**使用invokevirtual指令调用。**只是到了执行的时候，会找到此操作数（man）的实际类型，即Man，如果在Human中找到和此方法描述符、名称都相同的方法，则会进行调用；如果没找到则从继承关系中自下往上进行查找、调用。如果最终都没找到，就是AbstractMethodError异常。这也就是我们说的**invokevirtual查找过程：**

* 找到操作数栈顶的第一个元素所指向的对象的实际类型，记为C。
* 如果在类型C中找到与常量中的描述符合简单名称都相符的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找过程结束；如果权限校验不通过，返回java.lang.IllegalAccessError异常。
* 否则，按照继承关系从下往上一次对C的各个父类进行第2步的搜索和验证过程。
* 如果始终没有找到合适的方法，则抛出 java.lang.AbstractMethodError异常。

（3）虚拟机动态分派实现方式

由于动态分派是非常繁琐的动作，而且动态分派的方法版本选择需要考虑运行时在类的方法元数据中搜索合适的目标方法，**因此在虚拟机的实现中基于性能的考虑，在方法区中建立一个虚方法表**（invokeinterface有接口方法表），来提高性能。

[](https://github.com/FrankLee96/interview/blob/master/java/dispatcher.bmp)

虚方法表中存放各个方法的实际入口地址。如果某个方法在子类没有重写，那么子类的虚方法表里的入口和父类入口一致，如果子类重写了这个方法，那么子类方法表中的地址会被替换为子类实现版本的入口地址。