1. 描述一下 JVM 加载 Class 文件的原理机制?

在面试 java 工程师的时候，这道题经常被问到，故需特别注意。

Java 中的所有类，都需要由类加载器装载到 JVM 中才能运行。类加载器本身也是一个类，而它的工作就是把 class 文件从硬盘读取到内存中。在写程序的时候，我们几乎不需要关心类的加载，因为这些都是隐式装载的，除非我们有特殊的用法，像是反射，就需要显式的加载所需要的类。

Java 类的加载是动态的，它并不会一次性将所有类全部加载后再运行，而是保证程序运行的基础类(像是基类)完全加载到 jvm 中，至于其他类，则在需要的时候才加载。这当然就是为了节省内存开销。

Java 的类加载器有三个，对应 Java 的三种类:

```
Bootstrap Loader // 负责加载系统类（指的是内置类，像是String，对应于C#中的System类和C/C++标准库中的类）
    |
  - - ExtClassLoader // 负责加载扩展类(就是继承类和实现类)
        |
      - - AppClassLoader // 负责加载应用类(程序员自定义的类)
```

三个加载器各自完成自己的工作，但它们是如何协调工作呢？哪一个类该由哪个类加载器完成呢？为了解决这个问题，Java 采用了委托模型机制。

委托模型机制的工作原理很简单：当类加载器需要加载类的时候，先请示其 Parent(即上一层加载器)在其搜索路径载入，如果找不到，才在自己的搜索路径搜索该类。这样的顺序其实就是加载器层次上自顶而下的搜索，因为加载器必须保证基础类的加载。之所以是这种机制，还有一个安全上的考虑：如果某人将一个恶意的基础类加载到 jvm，委托模型机制会搜索其父类加载器，显然是不可能找到的，自然就不会将该类加载进来。
我们可以通过这样的代码来获取类加载器:

```
ClassLoader loader = ClassName.class.getClassLoader();
ClassLoader ParentLoader = loader.getParent();
```

注意一个很重要的问题，就是 Java 在逻辑上并不存在 BootstrapKLoader 的实体！因为它是用 C++编写的，所以打印其内容将会得到 null。

前面是对类加载器的简单介绍，它的原理机制非常简单，就是下面几个步骤:

1.装载:查找和导入 class 文件;

2.连接:

```
(1)检查:检查载入的class文件数据的正确性;

(2)准备:为类的静态变量分配存储空间;

(3)解析:将符号引用转换成直接引用(这一步是可选的)
```

3. 初始化:初始化静态变量，静态代码块。

```
这样的过程在程序调用类的静态成员的时候开始执行，所以静态方法main()才会成为一般程序的入口方法。类的构造器也会引发该动作。
```

来源：https://www.cnblogs.com/wenjiang/archive/2013/04/26/3044132.html

2. 什么是类加载器？

类加载器是一个用来加载类文件的类。Java 源代码通过 javac 编译器编译成类文件。然后 JVM 来执行类文件中的字节码来执行程序。类加载器负责加载文件系统、网络或其他来源的类文件。

3. 类加载器有哪些？

有三种默认使用的类加载器：Bootstrap 类加载器、 Extension 类加载器和 Application 类加载器。每种类加载器都有设定好从哪里加载类。

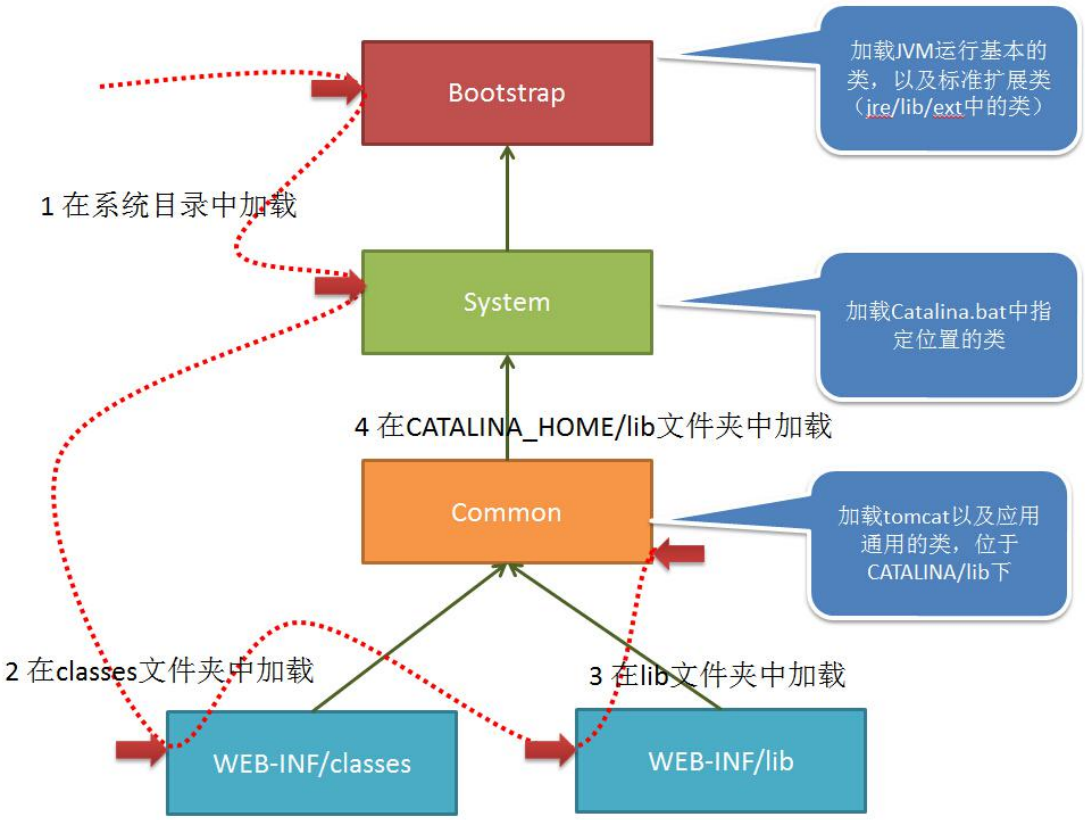
Bootstrap 类加载器负责加载 rt.jar 中的 JDK 类文件，它是所有类加载器的父加载器。Bootstrap 类加载器没有任何父类加载器，如果你调用 String.class.getClassLoader()，会返回 null，任何基于此的代码会抛出 NullPointerException 异常。Bootstrap 加载器被称为初始类加载器。

而 Extension 将加载类的请求先委托给它的父加载器，也就是 Bootstrap，如果没有成功加载的话，再从 jre/lib/ext 目录下或者 java.ext.dirs 系统属性定义的目录下加载类。Extension 加载器由 sun.misc.Launcher$ExtClassLoader 实现。

第三种默认的加载器就是 Application 类加载器了。它负责从 classpath 环境变量中加载某些应用相关的类，classpath 环境变量通常由-classpath 或-cp 命令行选项来定义，或者是 JAR 中的 Manifest 的 classpath 属性。Application 类加载器是 Extension 类加载器的子加载器。通过 sun.misc.Launcher$AppClassLoader 实现。

4. 什么是 tomcat 类加载机制？

在 tomcat 中类的加载稍有不同，如下图：



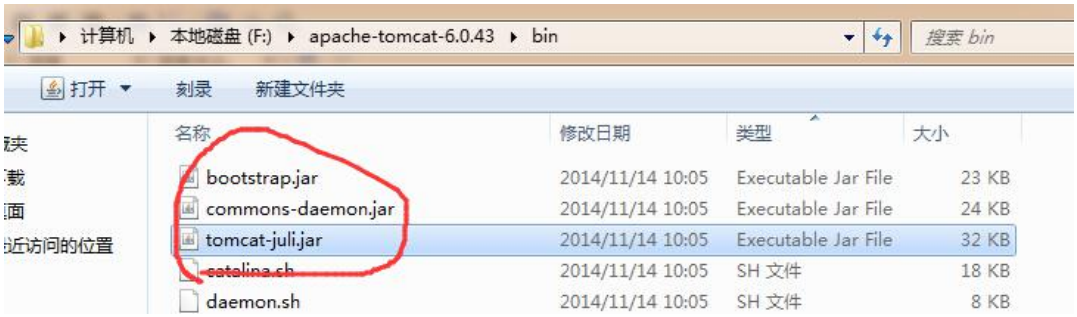
当 tomcat 启动时，会创建几种类加载器：

1 Bootstrap 引导类加载器

加载 JVM 启动所需的类，以及标准扩展类（位于 jre/lib/ext 下）

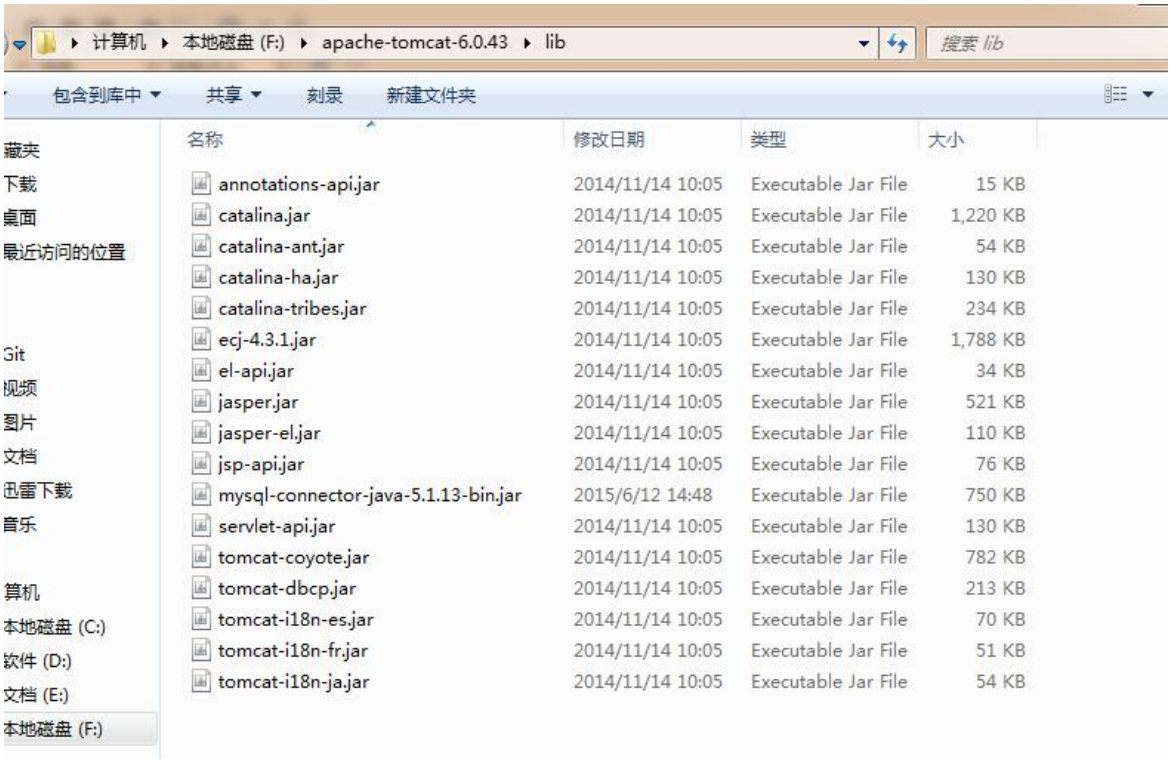
2 System 系统类加载器

加载 tomcat 启动的类，比如 bootstrap.jar，通常在 catalina.bat 或者 catalina.sh 中指定。位于 CATALINA_HOME/bin 下。



3 Common 通用类加载器

加载 tomcat 使用以及应用通用的一些类，位于 CATALINA_HOME/lib 下，比如 servlet-api.jar



4 webapp 应用类加载器

每个应用在部署后，都会创建一个唯一的类加载器。该类加载器会加载位于 WEB-INF/lib 下的 jar 文件中的 class 和 WEB-INF/classes 下的 class 文件。

当应用需要到某个类时，则会按照下面的顺序进行类加载：

1 使用 bootstrap 引导类加载器加载

2 使用 system 系统类加载器加载

3 使用应用类加载器在 WEB-INF/classes 中加载

4 使用应用类加载器在 WEB-INF/lib 中加载

5 使用 common 类加载器在 CATALINA_HOME/lib 中加载

参考：https://blog.csdn.net/dreamcatcher1314/article/details/78271251

5、类加载器双亲委派模型机制？

什么是双亲委派模型(Parent-Delegation Model)？为什么使用双亲委派模型？

JVM 中加载类机制采用的是双亲委派模型，顾名思义，在该模型中，子类加载器收到的加载请求，不会先去处理，而是先把请求委派给父类加载器处理，当父类加载器处理不了时再返回给子类加载器加载；

为什么使用双亲委派模型？

因为安全。使用双亲委派模型来组织类加载器间的关系，能够使类的加载也具有层次关系，这样能够保证核心基础的 Java 类会被根加载器加载，而不会去加载用户自定义的和基础类库相同名字的类，从而保证系统的有序、安全。

6. Java 内存分配？

一、 基本概念
每运行一个 java 程序会产生一个 java 进程，每个 java 进程可能包含一个或者多个线程，每一个 Java 进程对应唯一一个 JVM 实例，每一个 JVM 实例唯一对应一个堆，每一个线程有一个自己私有的栈。进程所创建的所有类的实例（也就是对象）或数组（指的是数组的本身，不是引用）都放在堆中,并由该进程所有的线程共享。Java 中分配堆内存是自动初始化的，即为一个对象分配内存的时候，会初始化这个对象中变量。虽然 Java 中所有对象的存储空间都是在堆中分配的，但是这个对象的引用却是在栈中分配,也就是说在建立一个对象时在堆和栈中都分配内存，在堆中分配的内存实际存放这个被创建的对象的本身，而在栈中分配的内存只是存放指向这个堆对象的引用而已。局部变量 new 出来时，在栈空间和堆空间中分配空间，当局部变量生命周期结束后，栈空间立刻被回收，堆空间区域等待 GC 回收。

具体的概念：JVM 的内存可分为 3 个区：堆(heap)、栈(stack)和方法区(method，也叫静态区)：

堆区：

存储的全部是对象，每个对象都包含一个与之对应的 class 的信息(class 的目的是得到操作指令）；
jvm 只有一个堆区(heap)，且被所有线程共享，堆中不存放基本类型和对象引用，

只存放对象本身和数组本身；
栈区：

每个线程包含一个栈区，栈中只保存基础数据类型本身和自定义对象的引用；
每个栈中的数据(原始类型和对象引用)都是私有的，其他栈不能访问；
栈分为 3 个部分：基本类型变量区、执行环境上下文、操作指令区(存放操作指令)；
方法区（静态区）：

被所有的线程共享，方法区包含所有的 class（class 是指类的原始代码，要创建一个类的对象，首先要把该类的代码加载到方法区中，并且初始化）和 static 变量。
方法区中包含的都是在整个程序中永远唯一的元素，如 class，static 变量。

二、实例演示
AppMain.java

```
public class AppMain          //运行时，jvm 把appmain的代码全部都放入方法区
{
  public static void main(String[] args) //main 方法本身放入方法区。
  {
    Sample test1 = new Sample( " 测试1 " );  //test1是引用，所以放到栈区里， Sample是自定义对象应该放到堆里面
    Sample test2 = new Sample( " 测试2 " );

    test1.printName();
    test2.printName();
  }
}

public class Sample     //运行时，jvm 把appmain的信息都放入方法区
{
  /** 范例名称 */
  private String name;    //new Sample实例后， name 引用放入栈区里，name 对应的 String 对象放入堆里

  /** 构造方法 */
  public Sample(String name)
  {
    this .name = name;
    }

     /** 输出 */
  public void printName()  //在没有对象的时候，print方法跟随sample类被放入方法区里。
  {
      System.out.println(name);
  }
}
```

运行该程序时，首先启动一个 Java 虚拟机进程，这个进程首先从 classpath 中找到 AppMain.class 文件，读取这个文件中的二进制数据，然后把 Appmain 类的类信息存放到运行时数据区的方法区中，这就是 AppMain 类的加载过程。

接着，Java 虚拟机定位到方法区中 AppMain 类的 Main()方法的字节码，开始执行它的指令。这个 main()方法的第一条语句就是：

```
Sample test1 = new Sample("测试1");
```

该语句的执行过程：

1、Java 虚拟机到方法区找到 Sample 类的类型信息，没有找到，因为 Sample 类还没有加载到方法区（这里可以看出，java 中的内部类是单独存在的，而且刚开始的时候不会跟随包含类一起被加载，等到要用的时候才被加载）。Java 虚拟机立马加载 Sample 类，把 Sample 类的类型信息存放在方法区里。

2、Java 虚拟机首先在堆区中为一个新的 Sample 实例分配内存，并在 Sample 实例的内存中存放一个方法区中存放 Sample 类的类型信息的内存地址。

3、JVM 的进程中，每个线程都会拥有一个方法调用栈，用来跟踪线程运行中一系列的方法调用过程，栈中的每一个元素就被称为栈帧，每当线程调用一个方法的时候就会向方法栈压入一个新帧。这里的帧用来存储方法的参数、局部变量和运算过程中的临时数据。

4、位于“=”前的 Test1 是一个在 main()方法中定义的一个变量（一个 Sample 对象的引用），因此，它被会添加到了执行 main()方法的主线程的 JAVA 方法调用栈中。而“=”将把这个 test1 变量指向堆区中的 Sample 实例。

5、JVM 在堆区里继续创建另一个 Sample 实例，并在 main 方法的方法调用栈中添加一个 Test2 变量，该变量指向堆区中刚才创建的 Sample 新实例。

6、JVM 依次执行它们的 printName()方法。当 JAVA 虚拟机执行 test1.printName()方法时，JAVA 虚拟机根据局部变量 test1 持有的引用，定位到堆区中的 Sample 实例，再根据 Sample 实例持有的引用，定位到方法去中 Sample 类的类型信息，从而获得 printName()方法的字节码，接着执行 printName()方法包含的指令，开始执行。

三、辨析

在 Java 语言里堆(heap)和栈(stack)里的区别 ：

栈(stack)与堆(heap)都是 Java 用来在 Ram 中存放数据的地方。与 C++不同，Java 自动管理栈和堆，程序员不能直接地设置栈或堆。
栈的优势是，存取速度比堆要快，仅次于直接位于 CPU 中的寄存器。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。另外，栈数据可以共享（详见下面的介绍）。
堆的优势是可以动态地分配内存大小，生存期也不必事先告诉编译器，Java 的垃圾收集器会自动收走这些不再使用的数据。但缺点是，由于要在运行时动态分配内存，存取速度较慢。

Java 中的 2 种数据类型：

一种是基本类型(primitive types)，共有 8 类，即 int, short, long, byte, float, double, boolean, char(注意，并没有 string 的基本类型)。这种类型的定义是通过诸如 int a = 3; long b = 255L; 的形式来定义的，称为自动变量。自动变量存的是字面值，不是类的实例，即不是类的引用，这里并没有类的存在。如 int a = 3; 这里的 a 是一个指向 int 类型的引用，指向 3 这个字面值。这些字面值的数据，由于大小可知，生存期可知(这些字面值固定定义在某个程序块里面，

程序块退出后，字段值就消失了)，出于追求速度的原因，就存在于栈中。

栈有一个很重要的特性:存在栈中的数据可以共享。假设我们同时定义: int a = 3;　　int b = 3;编译器先处理 int a = 3；首先它会在栈中创建一个变量为 a 的引用，然后查找有没有字面值为 3 的地址，如果没找到，就开辟一个存放 3 这个字面值的地址，然后将 a 指向 3 的地址。接着处理 int b = 3；在创建完 b 的引用变量后，由于在栈中已经有 3 这个字面值，便将 b 直接指向 3 的地址。这样，就出现了 a 与 b 同时均指向 3 的情况。

这种字面值的引用与类对象的引用不同。假定两个类对象的引用同时指向一个对象，如果一个对象引用变量修改了这个对象的内部状态，那么另一个对象引用变量也即刻反映出这个变化。相反，通过字面值的引用来修改其值，不会导致另一个指向此字面值的引用的值也跟着改变的情况。如上例，我们定义完 a 与 b 的值后，再令 a=4；那么，b 不会等于 4，还是等于 3。在编译器内部，遇到 a=4；时，它就会重新搜索栈中是否有 4 的字面值，如果没有，重新开辟地址存放 4 的值；如果已经有了，则直接将 a 指向这个地址。因此 a 值的改变不会影响到 b 的值。

另一种是包装类数据，如 Integer, String, Double 等将相应的基本数据类型包装起来的类。这些类数据全部存在于堆中，Java 用 new()语句来显示地告诉编译器，在运行时才根据需要动态创建，因此比较灵活，但缺点是要占用更多的时间。

7. Java 堆的结构是什么样子的？

JVM 的堆是运行时数据区，所有类的实例和数组都是在堆上分配内存。它在 JVM 启动的时候被创建。对象所占的堆内存是由自动内存管理系统也就是垃圾收集器回收。

堆内存是由存活和死亡的对象组成的。存活的对象是应用可以访问的，不会被垃圾回收。死亡的对象是应用不可访问尚且还没有被垃圾收集器回收掉的对象。一直到垃圾收集器把这些对象回收掉之前，他们会一直占据堆内存空间。

永久代是用于存放静态文件，如 Java 类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些 class，例如 Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类，永久代中一般包含：
类的方法(字节码…)
类名(Sring 对象)
.class 文件读到的常量信息
class 对象相关的对象列表和类型列表 (e.g., 方法对象的 array).
JVM 创建的内部对象
JIT 编译器优化用的信息

虚拟机中的共划分为三个代：
年轻代（Young Generation）、年老代（Old Generation）和持久代（Permanent Generation）。其中持久代主要存放的是 Java 类的类信息，与垃圾收集要收集的 Java 对象关系
不大。年轻代和年老代的划分是对垃 圾收集影响比较大的。

年轻代:
所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。年轻代分三个区。一个 Eden 区，两个 Survivor 区(一般而言)。大部分对象在 Eden 区中生成。当 Eden 区满时，还存活的对象将被复制到 Survivor 区（两个中的一个），当这个 Survivor 区满时，此区的存活对象将被复制到另外一个 Survivor 区，当这个 Survivor 去也满了的时候，从第一个 Survivor 区复制过来的并且此时还存活的对象，将被复制“年老区(Tenured)”。需要注意，Survivor 的两个区是对称的，没先后关系，所以同一个区中可能同时存在从 Eden 复制过来对象，和从前一个 Survivor 复制过来的对象，而复制到年老区的只有从第一个 Survivor 去过来的对象。而且，Survivor 区总有一个是空的。同时，根据程序需要，Survivor 区是可以配置为多个的（多于两个），这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。

年老代:
在年轻代中经历了 N 次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

持久代:
用于存放静态文件，如今 Java 类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些 class，例如 Hibernate 等，在这种时候需要设置一个比较大的持
久代空间来存放这些运行过程中新增的类。持久代大小通过-XX:MaxPermSize=进行设置。

注意：
JDK1.8 中，永久代已经从 java 堆中移除，String 直接存放在堆中，类的元数据存储在 meta space 中，meta space 占用外部内存，不占用堆内存。
可以说，在 java8 的新版本中，持久代已经更名为了元空间（meta space）。

8. 简述各个版本内存区域的变化？

参考：https://blog.csdn.net/rainnnbow/article/details/50541079

9. 说说各个区域的作用？

1、运行时数据区域
运行时数据区域包括方法区、虚拟机栈、本地方法栈、堆、程序计数器。其中方法区和堆是所有线程共享的数据区，其他的是线程隔离的数据区。

1.1、程序计数器
程序计数器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，确定下一条需要执行的字节码指令。java 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，在任何确定的一个时刻，一个处理器只会执行一条线程中的指令。为了线程切换之后能恢复到正确的执行位 置，每个线程都需要有一个独立的程序计数器，各个线程之间的计数器互不影响。如果线程正在执行的是一个 java 方法，则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，则计数器值为空。

1.2、java 虚拟机栈

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：每个方法被执行的时候都会创建一个栈帧用 于存在局部变量表、操作栈、动态链接、方法出口等信息。通过所说的栈是局部变量表，即与对象内存分配关系最密切的内存区域。局部变量表的内存空间在编译期 间完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是确定的，在运行期不会改变。

java 虚拟机栈有两种异常：如果线程请求的栈深度大于虚拟机所允许的深度，则抛弃 StackOverflowError 异常；如果虚拟机栈可以动态扩展的，当扩展时无法申请到足够的内存时会抛出 OutOfMemoryError 异常。

1.3、本地方法栈

本地方法栈与虚拟机栈所发挥的作用是相似的，区别在于虚拟机栈为虚拟机执行 java 方法的服务，本地方法栈则是为虚拟机使用到 native 方法服务。

1.4、java 堆

java 堆是虚拟机所管理的内存中最大的一块，是虚拟机启动是创建的能被所有线程共享的一块内存区域。java 堆的唯一目的就是存放对象实例，几乎 所有的对象实例和数组都在这里分配内存（随着 JTI 编译器的发展，在栈上也有可能分配）。java 堆是垃圾收集器管理的主要区域，在物理上可以使不连续的 内存空间，但在逻辑上是联系的。

如果再堆中没有内存完成实例的分配，并且堆也无法在扩展的时候，将会抛出 OutOfMemoryError 异常。

1.5、方法区

方法区也是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。这块区域很少进行垃圾回收，甚 至可以不实现垃圾收集，主要是针对常量池的回收和对类型的卸载。当方法区无法分配内存的时候，将抛出 OutOfMemoryError 异常。

Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池，用于存放编译期的各种字面量和符号引用。并非预置入 Class 文件中的常量才能进入常量池，运行期间也可能将新的常量放入池中，开发中用的比较多的是 String 类的 intern()方法。

2、例子解析

```
Object obj = new Object();
```

假设这句代码出现在方法体中，那么 Object obj 将会反映到 java 栈的局部变量表中，作为一个 reference 类型数据出现，new Object()将会反映到 java 堆中，形成一块存储了 Object 类型的实例数据的结构化内存，此对象类型数据，如对象类型、父类、实现的接口、方法 等信息存储在方法区。

10. Java 中会存在内存泄漏吗，简述一下？

理论上 Java 因为有垃圾回收机制（GC）不会存在内存泄露问题（这也是 Java 被广泛使用于服务器端编程的一个重要原因）；然而在实际开发中，可能会存在无用但可达的对象，这些对象不能被 GC 回收，因此也会导致内存泄露的发生。例如 Hibernate 的 Session（一级缓存）中的对象属于持久态，垃圾回收器是不会回收这些对象的，然而这些对象中可能存在无用的垃圾对象，如果不及时关闭（close）或清空（flush）一级缓存就可能导致内存泄露。

11. Java 类加载过程？

在 Java 中，类装载器把一个类装入 Java 虚拟机中，要经过三个步骤来完成：装载、链接和初始化，其中链接又可以分成校验、准备、解析
装载：查找和导入类或接口的二进制数据；
链接：执行下面的校验、准备和解析步骤，其中解析步骤是可以选择的；
校验：检查导入类或接口的二进制数据的正确性；
准备：给类的静态变量分配并初始化存储空间；
解析：将符号引用转成直接引用；
初始化：激活类的静态变量,初始化 Java 代码和静态 Java 代码块

12. 什么是 GC? 为什么要有 GC？

GC 是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，

忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。Java 程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾收集，可以调用下面的方法之一：System.gc()或 Runtime.getRuntime().gc()，但 JVM 可以屏蔽掉显示的垃圾回收调用。

垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低优先级的线程运行，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。

在 Java 诞生初期，垃圾回收是 Java 最大的亮点之一，因为服务器端的编程需要有效的防止

内存泄露问题，然而时过境迁，如今 Java 的垃圾回收机制已经成为被诟病的东西。移动智能终端用户通常觉得 iOS 的系统比 Android 系统有更好的用户体验，其中一个深层次的原因就在于 Android 系统中垃圾回收的不可预知性。

采用“分代式垃圾收集”。这种方法会跟 Java 对象的生命周期将堆内存划分为不同的区域，在垃圾收集过程中，可能会将对象移动到不同区域：

伊甸园（Eden）：这是对象最初诞生的区域，并且对大多数对象来说，这里是它们唯一存在过的区域。
幸存者乐园（Survivor）：从伊甸园幸存下来的对象会被挪到这里。
终身颐养园（Tenured）：这是足够老的幸存对象的归宿。年轻代收集（Minor-GC）过程是不会触及这个地方的。当年轻代收集不能把对象放进终身颐养园时，就会触发一次完全收集（Major-GC），这里可能还会牵扯到压缩，以便为大对象腾出足够的空间。
与垃圾回收相关的 JVM 参数：

· -Xms / -Xmx —堆的初始大小 / 堆的最大大小

· -Xmn — 堆中年轻代的大小

补充：

Java 是由 C++发展来的。

它摈弃了 C++中一些繁琐容易出错的东西。其中有一条就是这个 GC。

写 C/C++程序，程序员定义了一个变量，就是在内存中开辟了一段相应的空间来存值。内存再大也是有限的，所以当程序不再需要使用某个变量的时候，就需要释放这个内存空间资源，好让别的变量来用它。在 C/C++中，释放无用变量内存空间的事情要由程序员自己来解决。就是说当程序员认为变量没用了，就应当写一条代码，释放它占用的内存。这样才能最大程度地避免内存泄露和资源浪费。

但是这样显然是非常繁琐的。程序比较大，变量多的时候往往程序员就忘记释放内存或者在不该释放的时候释放内存了。而且释放内存这种事情，从开发角度说，不应当是程序员所应当关注的。程序员所要做的应该是实现所需要的程序功能，而不是耗费大量精力在内存的分配释放上。

Java 有了 GC，就不需要程序员去人工释放内存空间。当 Java 虚拟机发觉内存资源紧张的时候，就会自动地去清理无用变量所占用的内存空间。当然，如果需要，程序员可以在 Java 程序中显式地使用 System.gc()来强制进行一次立即的内存清理。

因为显式声明是做堆内存全扫描，也就是 Full GC，是需要停止所有的活动的（Stop The World Collection），你的应用能承受这个吗？而其显示调用 System.gc()只是给虚拟机一个建议，不一定会执行，因为 System.gc()在一个优先级很低的线程中执行。

13. 简述一下 Java 垃圾回收机制？

什么是垃圾回收机制：在系统运行过程中，会产生一些无用的对象，这些对象占据着一定的内存，如果不对这些对象清理回收无用对象的内存，可能会导致内存的耗尽，所以垃圾回收机制回收的是内存。同时 GC 回收的是堆区和方法区的内存。

JVM 回收特点：(stop-the-world)当要进行垃圾回收时候，不管何种 GC 算法，除了垃圾回收的线程之外其他任何线程都将停止运行。被中断的任务将会在垃圾回收完成后恢复进行。GC 不同算法或是 GC 调优就是减少 stop-the-world 的时间。à(为何非要 stop-the-world)，就像是一个同学的聚会，地上有很多垃圾，你去打扫，边打扫边丢垃圾怎么都不可能打扫干净的哈。当在垃圾回收时候不暂停所有的程序，在垃圾回收时候有 new 一个新的对象 B，此时对象 A 是可达 B 的，但是没有来及标记就把 B 当成无用的对象给清理掉了，这就会导致程序的运行会出现错误。

如何判断哪些对象需要回收呢：

引用计数算法（java 中不是使用此方法）：每个对象中添加一个引用计数器，当有别人引用它的时候，计数器就会加 1，当别人不引用它的时候，计数器就会减 1，当计数器为 0 的时候对象就可以当成垃圾。算法简单，但是最大问题就是在循环引用的时候不能够正确把对象当成垃圾。

根搜索方法（这是后面垃圾搜集算法的基础）：这是 JVM 一般使用的算法，设立若干了根对象，当上述若干个跟对象对某一个对象都不可达的时候，这个对象就是无用的对象。对象所占的内存可以回收。

根搜索算法的基础上，现代虚拟机的实现当中，垃圾搜集的算法主要有三种，分别是标记-清除算法、复制算法、标记-整理算法。

标记-消除算法：当堆中的有效内存被耗尽的时候，就会停止整个系统，就会调用标记-消除算法，主要做两件事，1 就是标记，2 就是清除。然后让程序恢复。

标记：遍历所有 GCroots 把可达的对象标记为存活的对象。

清除：把未标记为存活的对象清楚掉。

缺点：

就是效率相对比较低。会导致 stop-the-world 时间过长。

因为无用的对象内存不是连续的因此清理后的内存也不是连续的，(会产生内存碎片)因此 JVM 还要维持一个空闲列表，增加一笔开销，同时在以后内存使用时候，去查找可用的内存这个效率也是很低的。

复制算法：(这个算法一般适合在新生代 GC)，将原有的内存分为两块，每次只适用其中的一

块，在垃圾回收的时候，将一块正在使用的内存中存活(上述根搜索的算法)的对象复制到另一块没有使用的内存中，原来的那一块全部清除。与上述的标记-清除算法相比效率更高，但是不太适合使用在对象存活较多的情况下(如老年代)。

缺点：每次对整个半区内存回收，因此效率比上面的要高点，同时在分配内存的时候不需要考虑内存的碎片。按照顺序分配内存。简单高效。

但是最大的问题在于此算法在对象存活率非常低的时候使用，将可用内存分为两份，每次只使用一份这样极大浪费了内存。

注意（重要）：现在的虚拟机使用复制算法来进行新生代的内存回收。因为在新生代中绝大多数的对象都是“朝生夕亡”，所以不需要将整个内存分为两个部分，而是分为三个部分，一块为 Eden 和两块较小的 Survivor 空间(比例->8:1:1)。每次使用 Eden 和其中的一块 Survivor，垃圾回收时候将上述两块中存活的对象复制到另外一块 Survivor 上，同时清理上述 Eden 和 Survivor。所以每次新生代就可以使用 90%的内存。只有 10%的内存是浪费的。(不能保证每次新生代都少于 10%的对象存活，当在垃圾回收复制时候如果一块 Survivor 不够时候，需要老年代来分担，大对象直接进入老年代)

标记-整理算法：(老年代 GC)在存活率较高的情况下，复制的算法效率相对比较低，同时还要考虑存活率可能为 100%的极端情况，因此又不能把内存分为两部分的复制算法。

在上面标记-复制算法的基础之上，演变出了一个新的算法就是标记-整理算法。首先从 GCroots 开始标记所有可达的对象，标记为存活的对象。然后将存活的对象压缩到内存一端按照内存地址的次序依次排列，然后末端内存地址之后的所有内存都清除。

总结：将标记存活的对象按照内存地址顺序排列到内存另一端，末端内存地址之后的内存都会被清除。

比较：相比较于标记-清楚算法 (传统的)，该算法可以解决内存碎片问题同时还可以解决复制算法部分内存不能利用的问题。但是标记-整理算法的效率也不是很高。

->上述算法都是根据根节点搜索算法来判断一个对象是不是需要回收，而支撑根节点搜索算法能够正常工作理论依据就是语法中变量作用域的相关内容。

三种算法比较：

效率：复制算法>标记-整理算法>标记-清除算法；

内存整齐度：复制算法=标记-整理算法>标记-清除算法

内存利用率：标记-整理算法=标记-清除算法>复制算法

分代收集算法：

现在使用的 Java 虚拟机并不是只是使用一种内存回收机制，而是分代收集的算法。就是将内存根据对象存活的周期划分为几块。一般是把堆分为新生代、和老年代。短命对象存放在新生代中，长命对象放在老年代中。

对于不同的代，采用不同的收集算法：

新生代：由于存活的对象相对比较少，因此可以采用复制算法该算法效率比较快。

老年代：由于存活的对象比较多哈，可以采用标记-清除算法或是标记-整理算法

（注意）新生态由于根据统计可能有 98%对象存活时间很短因此将内存分为一块比较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中一块 Survivor。当回收时，将 Eden 和 Survivor 中还存活着的对象一次性地复制到另外一块 Survivor 空间上，最后清理掉 Eden 和刚才用过的 Survivor 空间。

上述是垃圾回收机制的算法，但是垃圾回收器才是垃圾回收的具体实现：

常见有五个垃圾回收器：

一：串行收集器：（Serial 收集器）

该收集器最古老、稳定简单是一个单线程的收集器，（stop-the-world）可能会产生长时间的停顿. serial 收集器一定不能用于服务器端。这个收集器类型仅应用于单核 CPU 桌面电脑。

新生代和老年代都会使用 serial 收集器。新生代使用复制算法（内存分三块的那个复制算法）。老年代使用标记-整理算法。

二：并行收集器：（Parallel 收集器）

parallel 收集器使用多线程并行处理 GC，因此更快。当有足够大的内存和大量芯数时，parallel 收集器是有用的。它也被称为“吞吐量优先垃圾收集器。”

三：并行收集器：（Parallel Old 垃圾收集器）

相比于 parallel 收集器，他们的唯一区别就是在老年代所执行的 GC 算法的不同。它执行三个步骤：标记-汇总-压缩（mark – summary – compaction）。汇总步骤与清理的不同之处在于，其将依然幸存的对象分发到 GC 预先处理好的不同区域，算法相对清理来说略微复杂一点。

四：并行收集器：（CMS 收集器）

（ConcurrentMark Sweep：并发标记清除）是一种以获取最短回收停顿时间为目标的收集器。适合应用在互联网站或者 B/S 系统的服务器上，这类应用尤其重视服务器的响应速度，希望系统停顿时间最短。

五：G1 收集器

这个类型的垃圾收集算法是为了替代 CMS 收集器而被创建的，因为 CMS 收集器在长时间持续运行时会产生很多问题。

14. 如何判断一个对象是否存活？

在堆里面存放着 Java 世界中几乎所有的对象实例，垃圾收集器对堆内存进行回收前，都会先判断这些

对象之中哪些还“存活”着，哪些已经“死去”(即不可能在被任何途径使用的对象)。一共有两种算法：

引用计数算法
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。

JVM 里面并没有选用引用计数算法来管理内存，主要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
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时，则证明此对象是不可用的。

15. 垃圾回收的优点和原理，并考虑 2 种回收机制？基本原理是什么？

Java 语言中一个显著的特点就是引入了垃圾回收机制，使 c++程序员最头疼的内存管理 的问题迎刃而解，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。由于 有个垃圾回收机制，Java 中的对象不再有"作用域"的概念，只有对象的引用才有"作用 域"。垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。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楚和回收，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。回收机制有分代复制垃圾回收和标记垃圾回收，增量 垃圾回收。

16. 深拷贝和浅拷贝？

浅拷贝被复制对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用仍然指向原来的对象。即对象的浅拷贝会对“主”对象进行拷贝，但不会复制主对象里面的对象。”里面的对象“会在原来的对象和它的副本之间共享。
简而言之，浅拷贝仅仅复制所考虑的对象，而不复制它所引用的对象。

深拷贝深拷贝是一个整个独立的对象拷贝，深拷贝会拷贝所有的属性,并拷贝属性指向的动态分配的内存。当对象和它所引用的对象一起拷贝时即发生深拷贝。深拷贝相比于浅拷贝速度较慢并且花销较大。
简而言之，深拷贝把要复制的对象所引用的对象都复制了一遍。

17. 什么是分布式垃圾回收（DGC）？它是如何工作的？

RMI 子系统实现基于引用计数的“分布式垃圾回收”(DGC)，以便为远程服务器对象提供自动内存管理设施。
当客户机创建（序列化）远程引用时，会在服务器端 DGC 上调用 dirty()。当客户机完成远程引用后，它会调用对应的 clean() 方法。
针对远程对象的引用由持有该引用的客户机租用一段时间。租期从收到 dirty() 调用开始。在此类租约到期之前，客户机必须通过对远程引用额外调用 dirty() 来更新租约。如果客户机不在租约到期前进行续签，那么分布式垃圾收集器会假设客户机不再引用远程对象。