**06 Java底层知识**

**开放的面试题——谈谈你对Java的理解？**

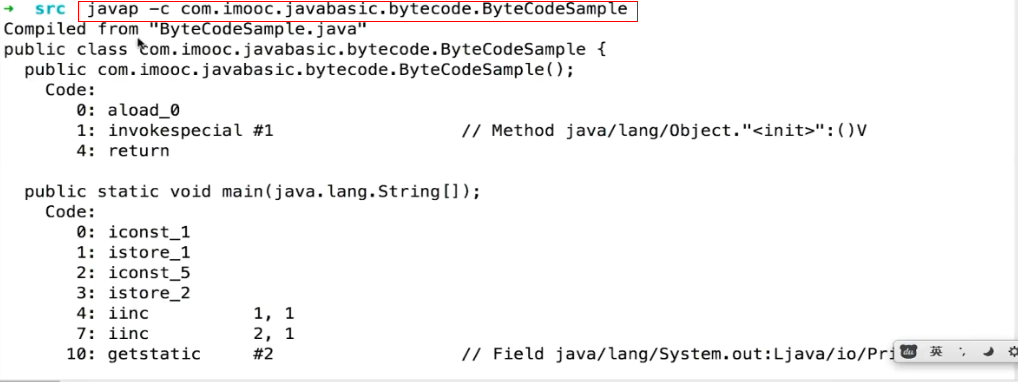
答： 考查是否真的掌握了Java，对基础知识的理解是否清楚，是否掌握了主要的模块和运行原理。需要列出以下点：

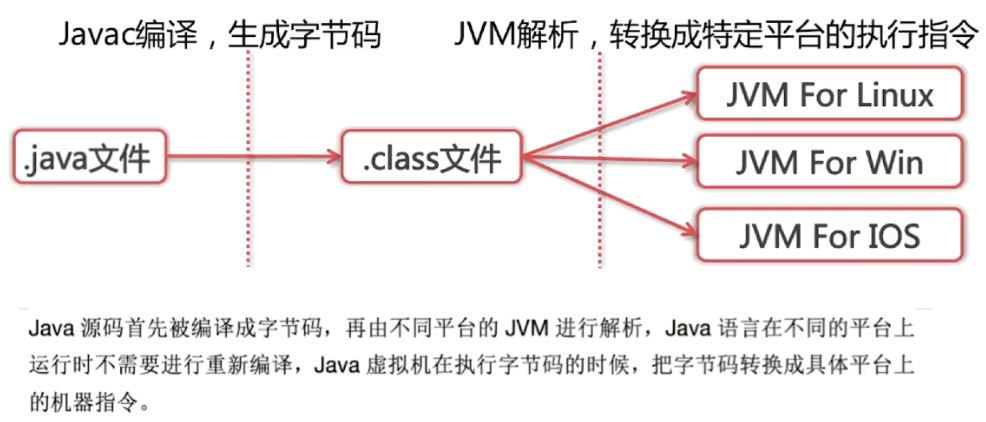
1. 平台无关性。一次编译，到处运行
2. GC。垃圾回收（garbage collection）机制，不用手动释放堆内存
3. 语言特性。泛型、反射、lambda表达式
4. 面向对象。封装、继承、多态
5. 类库。集合、并发库、网络库等
6. 异常处理。

**Compile Once,Run Anywhere（平台无关）如何实现？**

答：javac指令：将java源码.java文件编译成字节码.class文件。java指令：用JVM解析.class文件，将字节码加载进内存，最终转化成本操作系统可以识别的机器码去执行。javap -c指令：对代码进行反汇编，生成字节码内容

javap -c ClassName 得到字节码内容：





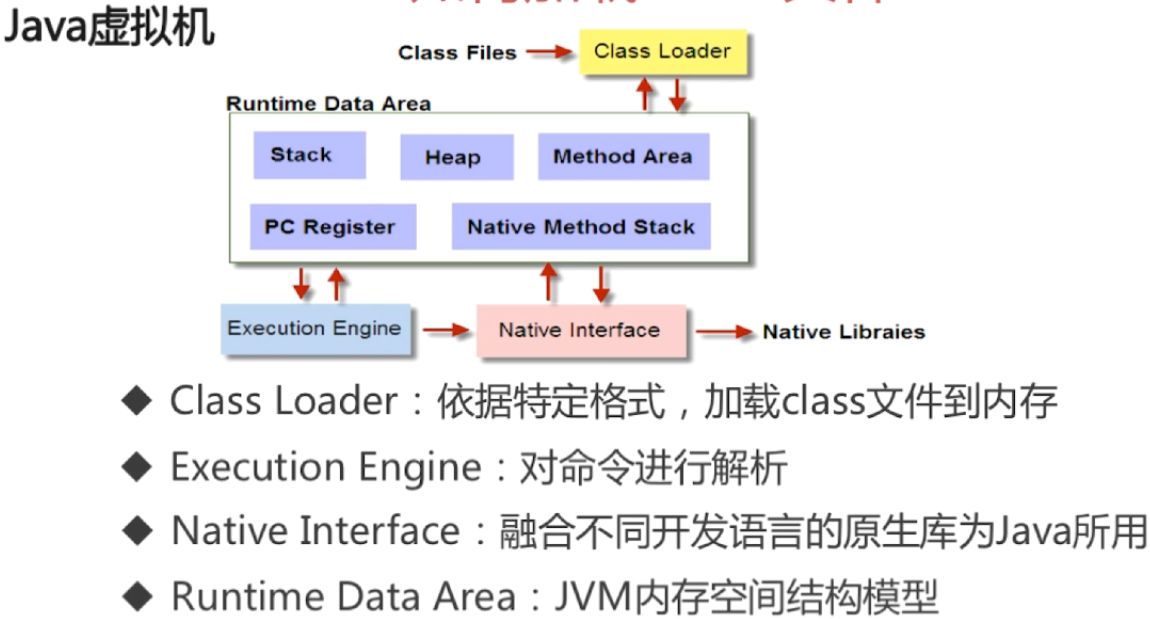
为什么JVM不直接将源码解析成机器码去执行？

答：

（1）性能：如果将源码直接解析成机器码执行，每次执行都需要进行各种准备工作，各种语法、句法、语义分析的检查，即重新编译，整体性能低。引入中间字节码能保证不需要多次校验。

（2）兼容性：可用别的语言（如：ruby）解析成字节码，被JVM调用，增加平台兼容扩展能力。

java虚拟机：虚拟机是一种抽象化的计算机，通过在真实计算机上仿真模拟各种计算机功能来实现。JVM有自己完善的硬件架构，如：处理器、堆栈、寄存器和相应的指令系统；屏蔽了与具体操作平台相关的信息，可以一次编译，到处运行。**JVM是一个内存中的虚拟机，它的存储就是内存，我们左右的所有类、常量、变量、方法都在内存中**。

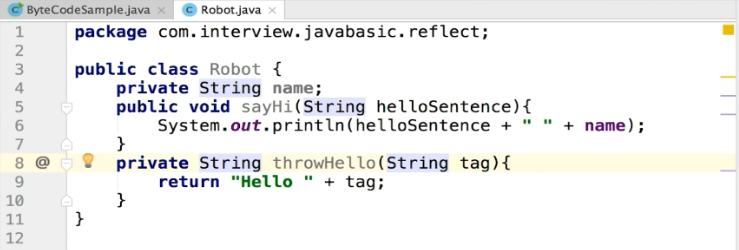


Java反射机制定义：Java反射机制是在运行状态中，对于任意一个类，都能够知道这个类的的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能成为java语言的反射机制。

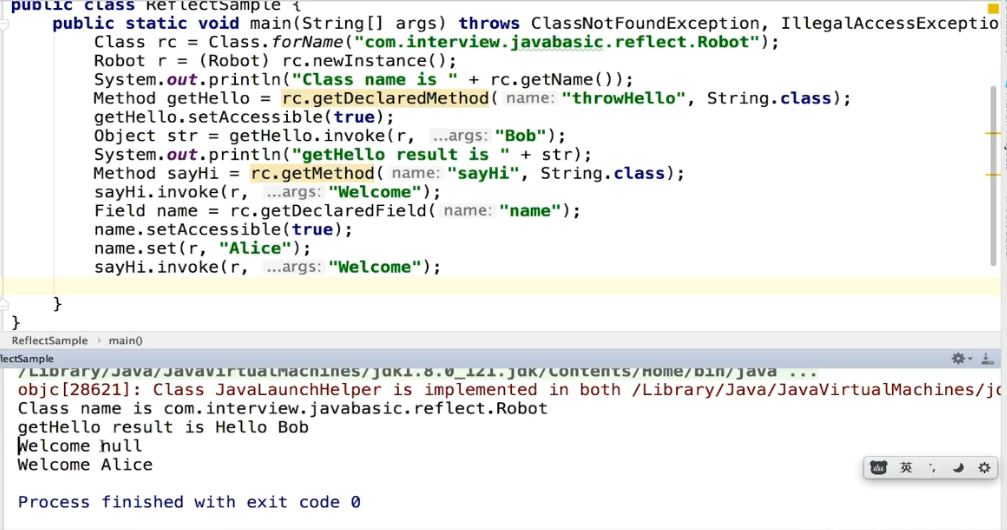
**反射**：把Java类中的各种成分，映射成一个个Java对象（Class、Method、Field等）

反射的示例：

源类：

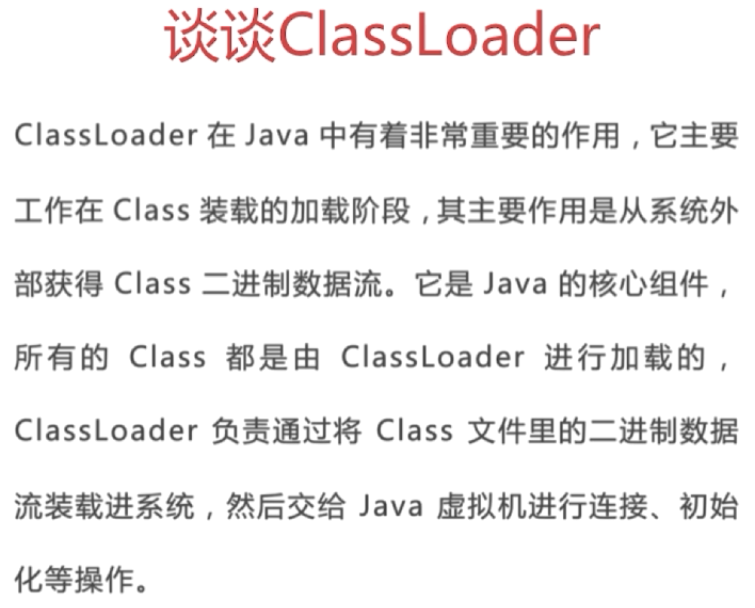


对源类进行的反射：



类从编译到执行的过程：

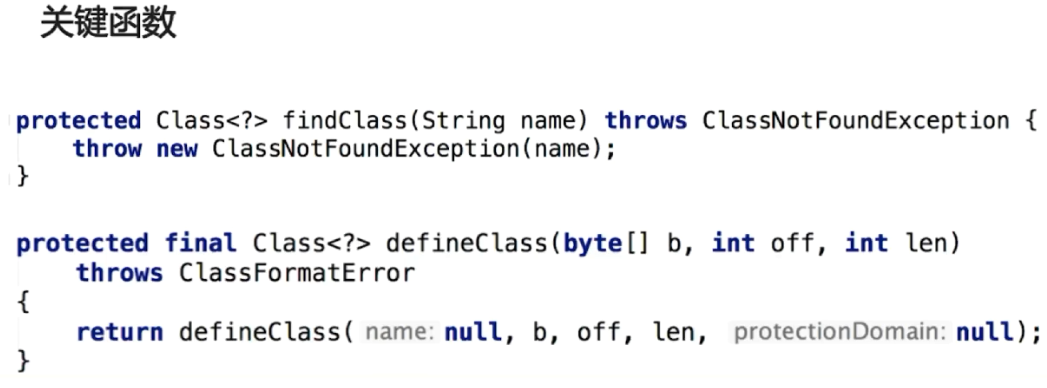
1. 编译器将Robot.java源文件编译为Robot.class字节码文件
2. ClassLoader将字节码转换为JVM中的Class<Robot>对象（以byte[]的形式传递给ClassLoader）
3. JVM利用Class<Robot>对象实例化为Robot对象



ClassLoader的种类：

1. BootStrapClassLoader：C++编写，加载核心库java.\*
2. ExtClassLoader:Java编写，加载扩展库javax.\*
3. AppClassLoader:Java编写，加载程序所在目录
4. 自定义ClassLoader：Java编写，定制化加载——可以不加载ClassPath中内容，甚至可以不是.class文件或者jar文件

自定义ClassLoader的实现：



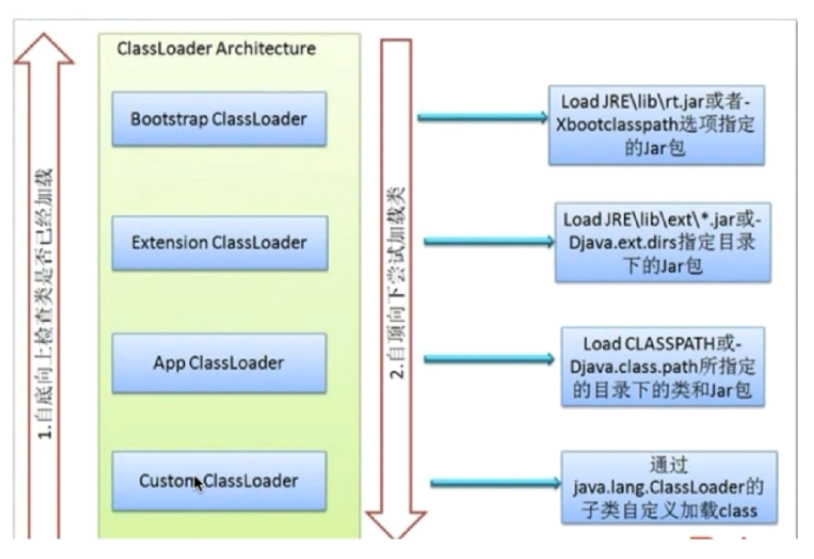
自定义的ClassLoader是一个继承了ClassLoader的类，通过override覆盖findClass和defineClass两个函数来实现自定义ClassLoader的行为。

函数 Class<?> findClass(String name)根据名称位置寻找.class文件，在自定义的目录下加载.class文件。转化成二进制流，调用defineClass处理并返回一个Class对象

**实现自定义ClassLoader示例见“剑指Java 6-5 谈谈ClassLoader”**

**类加载器的双亲委派机制**

JVM中各种ClassLoader的分工合作——先自底向上检查是否已加载，再自顶向下尝试加载类



**双亲委派机制示例见“剑指Java 6-6 ClassLoader的双亲委派机制”**

为什么要使用双亲委派机制？

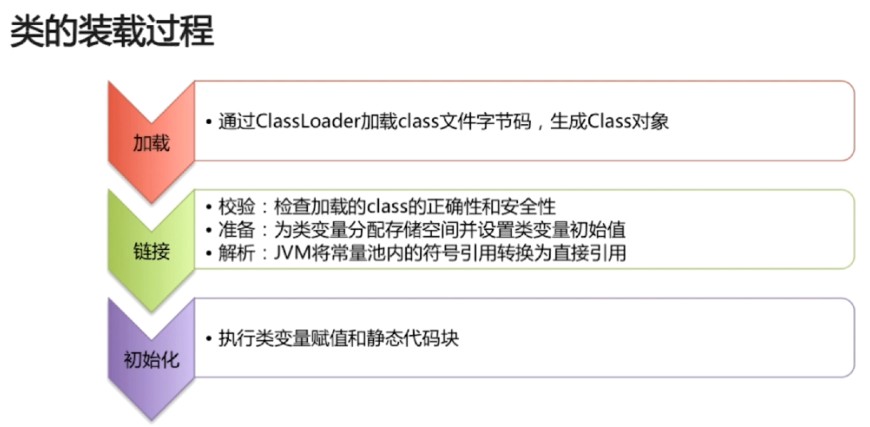
答：避免多份同样字节码的加载。Class类对象不像普通类对象，不需要创建多份，只需要创建一份以节省内存

类加载的方式有哪些？

1. 隐式加载：**new**——关键字new隐式调用类加载器加载对应的类到内存中，可自动获取对象实例，支持调用带参数的构造方法
2. 显示加载：**loadClass**，**forName**等——需要调用Class.newInstance方法生成对象的实例，newInstance不支持传参

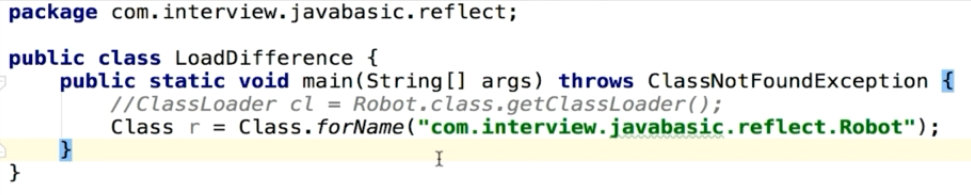
loadClass方法和forName方法的区别？

答：



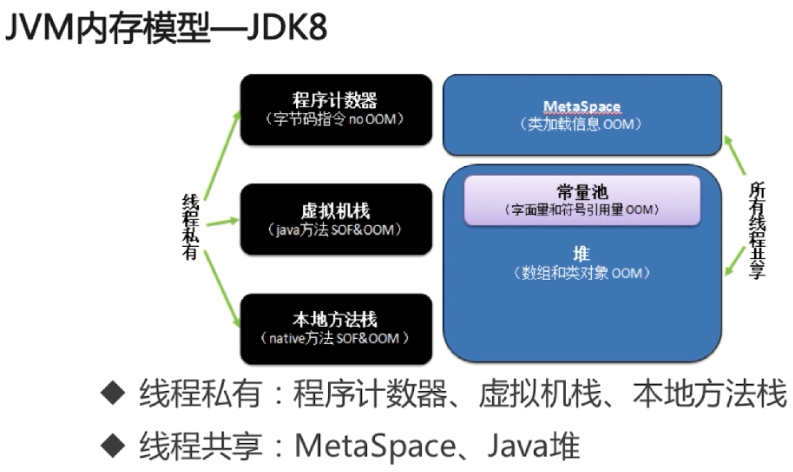
Class.forName得到的class是已经完成初始化的，而Classloader.loadClass得到的class是还没有连接的

代码示例（**具体示例见“剑指Java 6-7 谈谈loadClass和forName的区别”**）：



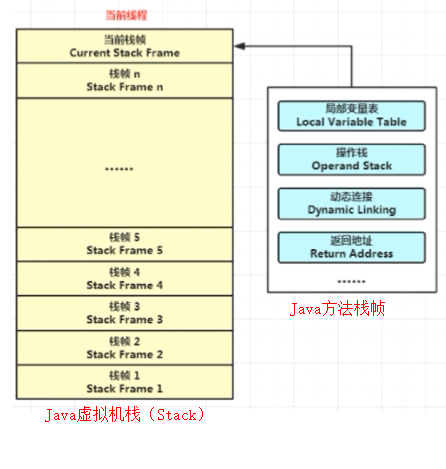
这种区别的作用：forName可以直接完成装载的所有工作，loadClass可以实现延迟加载（lazy loading）加快初始化速度

**从线程角度看Java内存模型：**



**程序计数器**（Program Counter Register）：线程私有；当前线程所执行的字节码行号指示器；是逻辑计数器；为了线程切换后能回复正确的执行位置，每个线程有独立的PCR；只为java方法计数，native方法PCR值为Undefined；不会发生内存泄漏

**Java虚拟机栈**（Stack）：线程私有；Java方法执行时的内存模型；每个方法被执行时创建一个栈帧，方法结束时销毁（栈的内存不需要GC回收，会自动释放）；



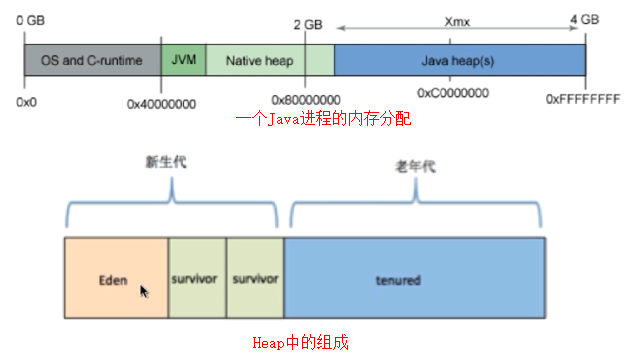
1. 局部变量表：包含方法执行过程中的所有变量。This引用、方法参数、boolean、byte、char、long、short、int、float、double等
2. 操作数栈：执行字节码指令时被用到，类似于原生CPU的寄存器

递归为什么会引发java.lang.StackOverflowError异常？

答：线程每执行一个方法时创建一个栈帧压入Java虚拟机栈的顶部，直到方法执行完毕弹出栈并销毁栈帧。每递归一层便创建一个栈帧压入，**递归过深，栈帧数超出栈深度，**就会报StackOverflowError。解决思路：限制递归次数，或使用迭代替换递归

**本地方法栈**（Native Method Stack）：线程私有；与Java虚拟机栈相似，主要作用与标注了native的方法

**Java堆**（Heap）：线程共享；JVM内存模型中最大的一块；存放类对象实例和数组；GC管理的主要区域；



**从存储角度看Java内存模型（常考问题）：**

1. **JVM三大性能调优参数-Xms、-Xmx、-Xss的含义？**

示例：java -Xms128m -Xmx128m -Xss256k -jar xxx.jar

答：

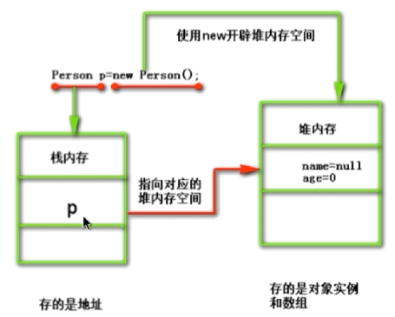
-Xss:规定每个线程虚拟机栈（Stack）的大小，影响并发线程数大小

-Xms:堆的初始值大小

-Xmx：堆可以扩展到的最大值（一般和-Xms大小相同，否则会引发内存抖动）

1. **堆和栈的联系？**

答：引用对象、数组时，栈里定义的变量保存堆中目标的首地址



1. **堆和栈的区别？**

答：

1. 管理方式：栈自动释放，堆需要GC
2. 空间大小：栈比堆小很多
3. 碎片相关：栈的碎片远小于堆
4. 效率：栈的操作简单，灵活度低，效率较高；堆操作复杂，灵活度高，效率低。