## ==========JVM========

## rt.jar被什么类加载器加载，什么时间加载 √

Bootstrap 类加载器（Bootstrap是C++实现的并没有一个Class和它对应）

XX.java 文件 javac 编译成XX.class，ClassLoader将XX.class装载到内存。此时也会将用到的java的类库相关的类（String、Object等）也装载到内存（存在于rt.jar）。

## 自己写的类被什么加载，什么时间加载；√

被AppClassLoader加载

（自己类的类加载器：System.***out***.println(TestClassLoaderLevel.**class**.getClassLoader());

打印：sun.misc.Launcher$AppClassLoader@73d16e93）

当我们主动使用类的时候，类才会被加载

主动使用有一下6种情况：

1.创建类的实例。例如：new Class();

2.访问某个类或接口的静态变量，或者给静态变量赋值

3.调用类的静态方法

4.反射：Class.forName("java.lang.String");

5.初始化一个类的子类

6.Java虚拟机启动时被标明为启动类的类（包含Main方法）

## 类加载的过程||类是如何加载到JVM中的 √

一个class文件怎么从硬盘到内存，开始执行的



分三步：

1. Loading ：把一个class文件（一个个的二进制，一个个的字节）load到内存，
2. Linking
3. Verification ：校验装进来的class文件符不符合class文件的标准（cafebabe开头）
4. **Preparation：给class静态成员变量赋默认值（\*不是初始值，如int 赋为0，引用赋为null）**
5. Resolution（解析）：
   * 1. class文件里类、方法、属性等符号引用（字符串），解析为直接引用，直接指向内存
     2. 常量池里中的各种符号引用解析为指针、偏移量等内存地址的直接引用）（直接能够访问到的内容）
6. Initializing：
   * 1. 调用类初始化代码，给静态成员变量赋初始值，执行静态语句块

## 强引用、软引用、弱引用、虚引用

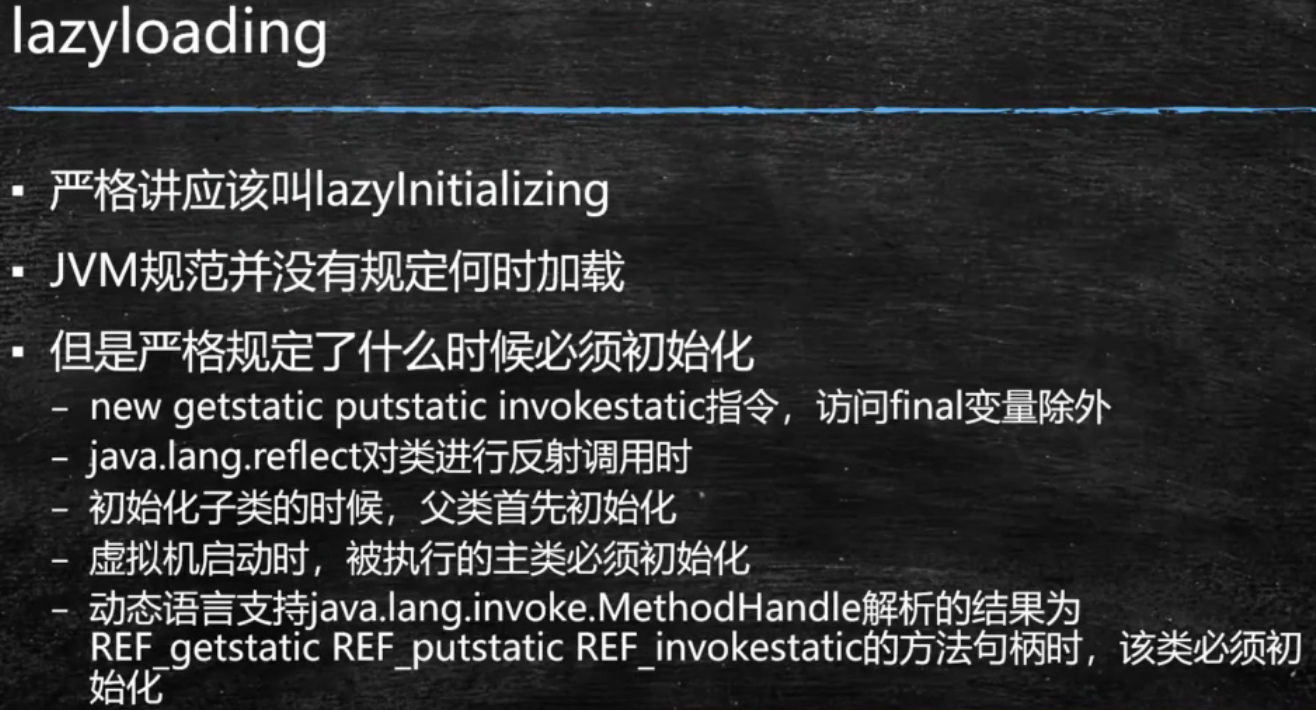
## 自己写的两个不同的类是被同一个类加载器加载的吗？为什么？ √

是，AppClassLoader应用类加载器,又称为系统类加载器,负责在JVM启动时,加载来自在命令java中的classpath或者java.class.path系统属性或者CLASSPATH操作系统属性所指定的JAR类包和类路径.

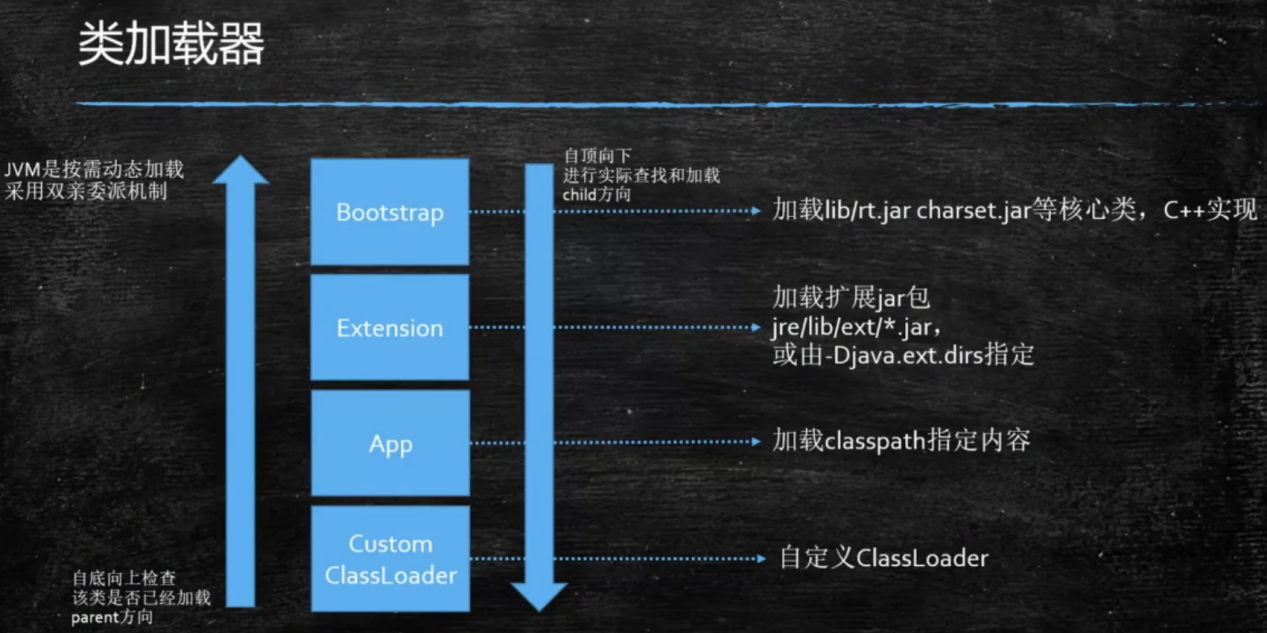
## 逃逸分析是什么，作用是什么，用途是什么;

## 软引用和弱引用的使用场景（软引用可以实现缓存，弱引用可以用来在回调函数中防止内存泄露）；

## 什么情况下会触发类加载； √

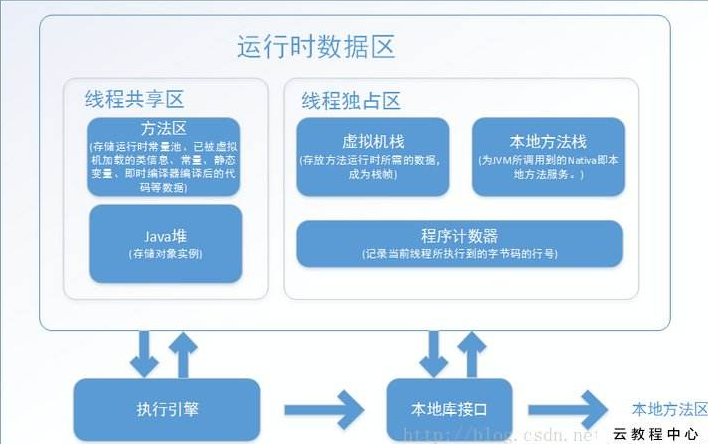


## JAVA 类加载器 √



## ==========JMM ==========

## JVM运行时内存区域划分 && Java内存模型JMM √



### PC（程序计数器）

记录当前线程所执行到的字节码的行号

### 虚拟机栈

存放方法运行时所需的数据，里面存的是栈帧（栈帧请看JVM学习笔记部分）

### 本地方法栈

JNI调用java虚拟机内部的C、C++的DLL

### 堆

存储对象实例

### 方法区

方法区是逻辑上的概念，装每个Class的结构，常量池

\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*面试题JDK1.7 & 1.8区别\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

方法区是逻辑上的概念，具体的实现：

Perm Space (<1.8指的就是永久区）：字符串常量位于PermSpace FGC不会被清理

Meta Space (>1.8 元数据区) ：字符串常量位于堆，会触发FGC

### 直接内存

JVM可以直接访问的内核空间的内存（OS管理的内存），用于NIO，提高效率，实现零拷贝

（在JDK1.8中，永久代被移除，取而代之的是元空间概念，也就是使用本地内存）

## 直接内存如何管理的 √

JDK1.4中加入新的NIO（New Input/OutPut）类，引入了一种基于通道（Channel）与缓冲区（Buffer）的I/O方式，可以通过Native函数库直接分配堆外内存，然后通过Java堆中的DirectByteBuffer对象来对这块内存的引用进行操作，避免数据在Java堆与Native堆中数据的来回复制。

本机的直接内存是有限制的，因此肯定也会由于内存不足造成OutOfMemoryError异常。

在JDK规范中存在的是方法区，但是HotSpot是通过PermGen Space（永久代）实现的。永久代仍存在于JDK1.7中，并没完全移除，譬如符号引用(Symbols)转移到了native heap；字面量(interned strings)转移到了java heap，String.intern()方法的实现也有变化；类的静态变量(class statics)转移到了java heap；在JDK1.8中，永久代被移除，取而代之的是元空间概念，也就是使用本地内存。

不断的使用String.intern()方法，在JDK1.6中会产生java.lang.OutOfMemoryError: PermGen space这个错误，在JDK1.7中就会报java.lang.OutOfMemoryError: Java Heap space，而在JDK1.8 中，也是java.lang.OutOfMemoryError: Java Heap space，但是还会多出warning：对参数PermSize以及MaxPermSize的设置已经从1.8中移除。

具体从永久代向元空间的转换原因如下：

1、字符串存在永久代中，容易出现性能问题和内存溢出。

2、类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。

3、永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。

## 讲下JVM的大页模式

## 对象内存布局 √

### 普通对象



|  |  |  |
| --- | --- | --- |
| 名称 | 解释 | 内存大小 |
| MarkWord  标记字段 | 哈希码、分代年龄、锁标志位、偏向线程ID、偏向时间戳等信息 | 8个字节 |
| ClassPointer类型指针 | 即指向当前对象的类的元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例 | -XX:+UseCompressedClassPointers 为4字节 不开启为8字节 |
| 实例数据 | 成员变量，int m=2 ，int n=1；String就引用指向别处 | 引用类型：-XX:+UseCompressedOops 为4字节 不开启为8字节 Oops Ordinary Object Pointers |
| Padding对齐  8的倍数 | 64位的机器读是按块来读的，并不是按字节，来提高效率。比如15个字节，就填充成16字节 | 8的倍数 |

### 数组对象

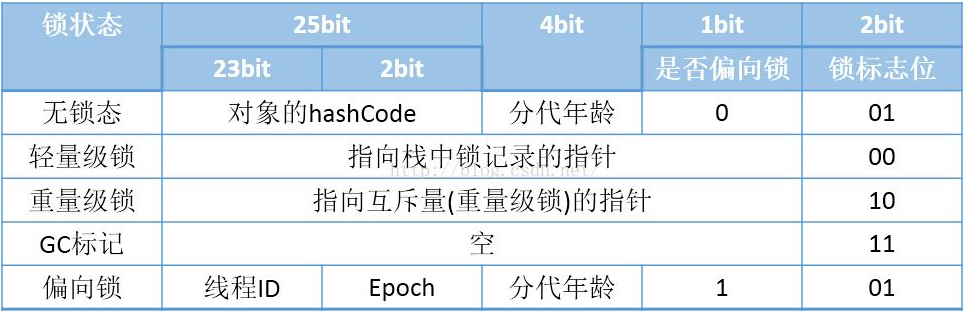
1. .对象头：markword 8
2. ClassPointer指针同上，数组装的class类型
3. 数组长度：4字节
4. 数组数据
5. 对齐 8的倍数

## 对象头 || Markword里面装的是什么？

至少说出来2点

1. **锁定信息：**有两位（严格说3位，还有偏向锁）代表对象有没有被锁定，锁定就是synchronized ( this 、o、p、...)这个对象，当所谓的synchronized这个对象的时候，实际上是对象脑袋上的两位，来代表锁的标志位。
2. **GC的标志位：**我这个对象被回收了多少次了，分代的年龄。

（这两项说出来，面试就八九不离十了）



## JMM里边的原子性、可见性、有序性是如何体现出来的，JMM中内存屏障是什么意思

### 概念：

CPU为了提高指令 执行效率，会在一条指令执行过程中（比如去内存读数据（慢100倍）），去同时执行另一条指令，前提是两条指令没有依赖关系。【volatile防止指令重拍序，就是防止这里的指令重排】

### 举例：乱序读

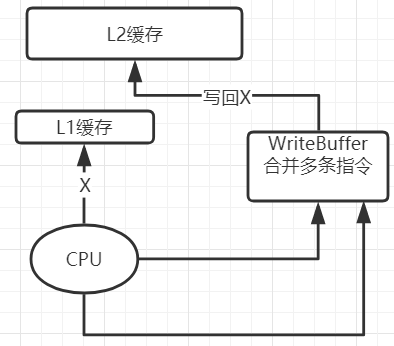
CPU从内存一次性读取5条指令，指令1需要从内存读取一个数，如果在执行第一条指令的时候，需要等着这个数读回来，才能执行下一条指令，需要等待100倍的时间。CPU在等待读数据的期间，CPU会分析下面的指令如果下面的指令跟上边的指令没有直接的依赖关系，下面的指令就先运行。

### 合并写

写操作也可以合并

1. CPU将某个值X写回L1缓存，发现L1缓存没有X，
2. CPU将X写回L2缓存（此处速度有点慢），接下来别的指令也修改了X的值，而此时写回L2还没执行完，

CPU会把这些指令扔到一个合并缓存里，输出一个最终的计算结果写回L2里



**内存屏障：保证指令不能重排序**

特定的位置插进去内存屏障，将指令分开。指令不能重排

## ==========GC ==========

## JVM如何设置参数

## JVM性能调优

## 常见的GC回收算法及其含义

## 内存溢出OOM和堆栈溢出SOE的示例及原因、如何排查与解决

## 如何判断对象是否可以回收或存活

## 常见的JVM性能监控和故障处理工具类：jps、jstat、jmap、jinfo、jconsole等

## Minor GC与Full GC分别在什么时候发生？什么时候触发Full GC;

## GC收集器有哪些？CMS收集器与G1收集器的特点。

## Java在什么时候会出现内存泄漏；

## 为什么新生代内存需要有两个Survivor区？

## 几种常用的内存调试工具：jmap、jstack、jconsole；

## G1停顿吗，CMS回收步骤，CMS为什么会停顿，停顿时间；

## 堆分为哪几块，比如说新生代老生代，那么新生代又分为什么？

## 为什么JVM调优经常会将-Xms和-Xmx参数设置成一样；

## CMS垃圾回收过程；

## 然后讲下对象的死亡过程？

## Full GC次数太多了，如何优化；

## 如果你的项目出现了内存泄露，怎么监控这个问题呢；

## 标记清除和标记整理的区别和优缺点，为何标记整理会发生stop the world；

## 开闭原则，解析工厂方法模式，建造者模式，区别。手撸出来。

## 写个后缀表达式，为什么要设计后缀表达式，有什么好处？然后写下中缀。

## 我看你做过性能优化，比如你怎么分析项目里面的OOM的，内存泄露呢？详细说思路;

## 涉及OOM、JVM优化、源码问题、数据库优化、多线程等问题;

## CPU高？什么情况CPU高？解决什么问题？

## 引用计数法与GC Root可达性分析法区别；

## 开闭原则说一下

## Java内存抖动严重，优化的思路；

## Java 内存结构（注：不是 Java 内存模型，别搞混）

## 怎么判断对象是否可 GC？Java 对象有哪些引用类型？有什么区别？

## OOM 出现的有哪些场景？为什么会发生？

## Minor GC 和 Full GC 有什么区别？分析过 GC 日志吗？

## 说下你知道的垃圾回收算法

## CMS 和 G1 的区别知道吗？使用场景分别是？你项目中用的是哪个？

## 你还知道哪些 JVM 调优参数？

## 假如线上服务发生 OOM，有哪些措施可以找到问题？

## 假如线上服务 CPU 很高该怎么做？有哪些措施可以找到问题？

## 假如线上应用频繁发生 Full GC，有哪些措施可以找到问题？

## 一般线上环境遇到 JVM 问题，你会使用哪些工具来分析？找到问题后又该如何去解决呢？