程序计数器私有?

程序计数器主要做两件事：
1.字节码解释器通过改变程序计数器来读取指令，实现代码的流程控制；
2.在多线程中，程序计数器主要记录当前线程的执行位置，保证线程被切换回来的时候能知道上次运行到什么位置了。
所以程序计数器私有化主要是为了保证线程切换后能恢复到正常的执行位置。

虚拟机栈和本地方法栈私有?

虚拟机栈：存放的是栈帧。每个java方法在执行的时候会创建一个栈帧放入虚拟机栈中。这个栈帧中保存了局部变量表、操作数栈、常量池引用等信息。(局部变量表中放方法中的参数和变量，还有对象的引用)
本地方法栈：和虚拟机栈作用相似。主要是存放native方法的数据。
私有化是为了保证线程中的局部变量不会被其他线程访问。

堆和方法区

堆和方法区是所有线程共享的资源，堆是进程中最大的一块内存，主要用于存放新创建的对象实例，几乎所有对象都在这里分配内存。方法区是存放已经加载了的类信息、常量、静态变量等数据。
对象实例数据：放在堆中，指对象中各个实例字段的数据；
对象类型数据：放在方法区，表示对象的类型、父类、实现的接口、方法等。

所有对象一定会在堆中创建吗？
不一定，通过逃逸分析，如果对象不发生逃逸，且在方法中创建对象，在方法中销毁，那么这个对象可以在栈上进行分配。这样做的好处是减少垃圾回收次数，提高性能。

为什么要将方法区变为元空间

jdk1.8之前：方法区
方法区还是放在堆里面的。方法区有JVM本身固定设置的大小上限，无法调整
jdk1.8之后：元空间
元空间放在本地内存中，拥有更大的内存空间，内存溢出概率减小，且能加载更多的类

对象的内存布局

对象头：
    记录对象在运行时需要的一些数据：
        哈希码、GC分代年龄、锁的状态标志、线程持有的锁、偏向线程ID、偏向时间戳
实例数据：
    成员变量的值，包括父类成员变量
对齐填充：
    确保对象的总长度为8字节的整数倍

对象创建过程

1.类加载检查
    检查new指令后面的类是否已经被加载、解析、初始化。如果没有，则先进行类加载
2.分配内存
    指针碰撞：适用于堆内存比较规整(采用标记-整理或者复制算法)
    空闲列表：适用于堆内存比较乱(采用标记-清除算法)

    分配内存时的并发安全机制：
        CAS+重试：保证原子性
        TLAB：为每一个线程预先在Eden区分配一块内存，JVM在给线程中对象分配内存时首先在TLAB中分配。当对象大于TLAB中的剩余内存或者内存已用尽再采用上面的方式进行分配。
3.初始化
    为对象中的成员变量赋初值，设置对象头的信息，调用构造方法初始化。

对象的访问方式

句柄访问方式：
    堆中额外建立一个句柄池，指向堆中的对象实例数据和方法区中的对象类型数据
直接指针访问方式：
    直接指向堆中的对象实例数据，但是实例数据中要多一个指向类型数据的指针

对象实例数据：放在堆中，指对象中各个实例字段的数据；
对象类型数据：放在方法区，表示对象的类型、父类、实现的接口、方法等。

垃圾回收策略

垃圾回收主要是回收堆和方法区中的数据。

判定对象是否存活：
    引用计数法：
        对象头维护一个计数器，被引用一次就+1，引用结束就-1；计数器为0则判断对象无效
        优点：简单，效率高
        缺点：一些对象会循环引用、交叉引用
    可达性分析法：
        和GC Roots相关联的对象就是有效对象，无关的则会被回收
    GC Roots：
        虚拟机栈和本地方法栈中引用的对象；
        方法区中常量引用的对象；
        方法区中静态属性引用的对象；
        被同步锁持有的对象；

引用的种类：

强引用：new一个对象这种。这种永远不会被回收。但是如果错误保持强引用，比如赋值给static变量，可能会造成内存泄漏。

软引用：在内存不足的时候会被回收

弱引用：无论内存是否充足，只要被扫描到都会被回收

虚引用：主要用来跟踪对象被垃圾回收的活动。

回收方法区中的垃圾：

废弃的常量：

常量池中的常量没有被任何变量或对象引用，就会被清除。

无用的类：

该类所有对象被清除；

该类的类加载器被回收；

没有通过反射访问该类的方法；

垃圾回收算法

标记-清除算法：

通过可达性分析法，将与GC Root相关的对象标记为存活对象。然后清除未被标记的对象以及这些标记，方便下次垃圾回收。

缺点：

效率问题：效率不高

空间问题：清除后会产生大量碎片，不利于后续内存分配

标记-整理算法(老年代)：

通过可达性分析法，将与GC Root相关的对象标记为存活对象。移动所有存活对象，放在一起，存活对象末端地址后面的对象全部回收。

复制算法(新生代)：

将内存分对半分，每次只用其中的一块，当需要进行垃圾回收的时候，将存活对象复制到另外一半，再回收这一半的内存。

优点：不会有碎片问题。

缺点：浪费空间。

改进：

将内存分为三部分：Eden、From Survivor、To Survivor，比例是8:1:1.这样只会浪费10%的内存空间，但是无法保证每次存活的对象都不到10%。当空间不够的时候，就需要老年代内存进行分配担保。

分配担保：如果垃圾回收后有超过10%的对象存活，则存活对象进入老年代。

垃圾收集器

新生代：

Serial垃圾收集器(单线程)：

只有一个线程在回收垃圾，同时垃圾回收过程中用户线程停止。使用复制算法

ParNew垃圾收集器(多线程)：

多个线程并行回收垃圾，同时垃圾回收过程中用户线程停止。使用复制算法

Parallel Scavenge垃圾收集器(多线程)：使用复制算法

注重回收效率

老年代：

Serial Old垃圾收集器(单线程)：

只有一个线程在回收垃圾，同时垃圾回收过程中用户线程停止。使用标记-整理算法

Parallel old垃圾收集器(多线程)：

Parallel Scavenge的老年代版本，注重回收效率。使用标记-整理算法

CMS垃圾收集器(并发标记清除)：

注重于最短的停顿时间；

在垃圾收集的时候用户线程可以和垃圾回收线程并发执行。

初始标记：对所有与GC Roots关联的对象进行标记。暂停用户线程

并发标记：与用户线程并发执行，进行可达性分析，但是这样无法保证可达性分析法的实时性。
重新标记：整合前两次标记。暂停用户线程
并发清除：与用户线程并发执行，清理未被标记的对象。

优点：
并发收集、低停顿
缺点：
对Cpu资源敏感；
无法处理浮动垃圾；
使用的是标记-清除算法，有大量碎片

G1通用垃圾收集器：
没有新生代和老年代的概念。将堆划分为一块块的Region。进行垃圾回收的时候，估计每块Region中的垃圾数量，从回收价值大的开始回收。
每个Region都有新生代Eden+Survivor、老年代Old、未使用区域和连续的大象区(Humongous)
整体上采用标记-整理算法，局部采用复制算法。
执行步骤：
初始标记；
并发标记；
最终标记；
筛选回收；
Rset(Remembered Set):可能这块Region区的对象会引用其他Region区的对象，因此通过Rset记录可以避免扫描整个堆空间。
垃圾回收模式：
young GC：对新生代执行复制算法
mixed GC：老年代占比过多，触发mixed GC，类似于CMS
full GC：mixed GC回收失败，使用单线程的Serial Old开始Full GC

内存分配与回收

新生对象一般分配在eden区，如果对象太大，则分配到老年区。
minor GC：新生代的GC，执行复制算法；执行时机：对象无法放在eden区
Major GC：老年代的GC，执行标记-整理算法；执行时机：老年代也放不下

对于发生一次minor GC后没有被回收的对象，在其对象头中的分代年龄会+1，超过默认的15后转到老年代中。

Class文件结构

魔数：class文件身份识别
版本信息：分为次版本号(minor version)、主版本号(major version)，高版本可以向下兼容
常量池：
字面量(字符串、final常量等)
符号引用：
类和接口的全限定名
字段的名称和描述符
方法的名称和描述符
访问标志：用于识别一些类或者接口层次的访问信息。比如这个Class是类还是接口；是否是public类型等等。
类索引、父类索引、接口索引的集合：由这三个来确定这个类的继承关系
类索引：确定这个类的全限定名
父类索引：确定这个类的父类的全限定名
接口索引集合：确定这个类实现哪些接口
字段表集合：描述接口或者类中声明的变量(不包括在方法中声明的局部变量)
作用域(public、private、protected)
实例变量还是类变量(static)
可变性(final)

并发可见性(volatile)
可被序列化(transient)
数据类型(基本数据类型、对象、数组)
字段名称
方法表集合：
访问标志
名称索引
描述符索引
属性表集合
方法中的代码经过编译器编译成了字节码指令，存放在方法属性表集合中的一个名为Code的属性里面
属性表集合：

类的生命周期

加载、验证、准备、解析、初始化、使用、卸载

验证、准备、解析：统称为连接

类加载过程中“初始化"开始的时机

遇到new、和静态字段相关的指令时，如果类还没有初始化，则需要初始化；
对类进行反射调用的时候；
初始化一个类的时候，如果父类还没有初始化，则需要先触发父类的初始化；
main()方法的主类；

类加载的过程

加载：
通过类的全限定名获取该类的二进制字节流；
将二进制字节流所代表的静态结构转化为方法区的运行时数据结果；
在内存中创建一个代表该类的java.lang.Class对象，作为方法区这个类的各种数据访问的入口。
验证：
文件格式验证；
元数据验证；
字节码验证；
符号引用验证；
准备：
为类变量分配内存并设置初始值的阶段
解析：
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：
类加载的最后一步
卸载：
即该类的Class对象被GC
需满足三个要求：
该类的所有实例对象都被GC；
该类没有在其他任何地方被引用(反射)；
类加载器已经被GC；

类加载器种类

启动类加载器：加载jre/lib下的jar包
扩展类加载器：加载jre/lib/ext的jar包
应用程序类加载器：面向用户的加载器，加载当前应用classpath下所有jar包和类

双亲委派模型

在类加载的时候，系统会首先判断当前类是否被加载过。已经被加载的类会直接返回，否则才会尝试加载。加载的时候，首先会把该请求委派该父类加载器的 loadClass() 处理，因此所有的请求最终都应该传送到顶层的启动类加载器 BootstrapClassLoader 中。当父类加载器无法处理时，才由自己来处理。当父类加载器为null时，会使用启动类加载器 BootstrapClassLoader 作为父类加载器。

好处：保证java程序的稳定运行，避免重复加载。自定义java.lang.Object类会出现不同的Object类。