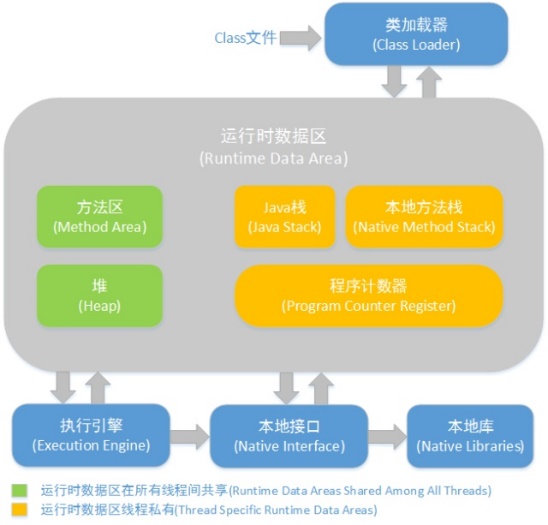
1. 什么是Java虚拟机？为什么Java被称作是“平台无关的编程语言”？

能够让java字节码文件运行的虚拟机。因为只要是符合jvm规范的字节码文件都能够再上面运行，所以不仅是java能够在jvm上运行，Kotlin等也能在jvm上运行。

1. java内存结构？



程序计数器（线程私有）： 比较小的内存空间，控制着线程的执行顺序，如循环，跳转，异常处理等，还有线程的中止，恢复等。

Java栈/虚拟 机栈（线程私有）：java栈对应着方法执行的内存模型，每个方法在执行的时候就会创建一个java栈，他存储着局部变量表，方法出口，操作栈等信息。

本地方法栈（线程私有）：他和java栈作用是相似的，不过java对应的是java的方法，而本地方法栈是对应的本地方法。

Java堆（线程共享）：堆里面存放了java中几乎所有的对象。他也是虚拟机管理的内存中最大的一块。

方法区：存储已经被虚拟机加载的信息，常量，静态变量等数据。

1. 对象的创建过程
2. 当虚拟机收到New的字节码指令，先会去方法区中寻找，并且检查是否已经被加载过，如果没有被加载，首先执行类的加载过程。
3. 为新生对象分配内存
4. 对对象进行设置，包括对象属于哪个类的实例，GC的年龄信息，元数据信息等。此时所有的数据都是零值
5. 初始化类，执行构造函数，真正的对象被构造出来了。

memory =allocate();    //1：分配对象的内存空间

ctorInstance(memory);  //2：初始化对象

instance =memory;     //3：设置instance指向刚分配的内存地址

1. OOM

OOM是所需要的内存超过申请的内存或者物理内存，也叫做内存溢出。在jvm中，以下几种情况会导致OOM：

1. **Java堆溢出**：当不断地创建对象，并且让GC root和对象之间有可达路径避免被垃圾回收机制清理，那么随着对象地增加，使存储对象的容量超过堆的总容量，则会报错。**OOM:Java head space.**
2. **虚拟机栈和本地方法栈溢出：**1.当栈深度超过虚拟机允许的最大深度，会抛出StackOverFlowError。2.当虚拟机栈内存允许动态扩展，但申请不到足够内存的时候，报OOM错误。（递归或者创建线程）
3. **方法区和运行时常量池（属于方法区）溢出：**方法区中存放着类的类型相关信息，如果加载很多的类，会将类信息放如方法区中，就可能会发生OOM。PermGen space

总结：内存模型中的5大区域，无论哪一个存放的数据所需要的内存超过了限制，都会发生OOM。

1. 如何判断哪些对象可以被回收

两种方式：

1. 引用计数法：在对象中添加一个引用计数器，每当有一个地方引用他，计数器加一，引用失效计数器减一，当计数器为0的时候，就表明不会被使用，会被回收了。但是由于存在循环引用的问题，现在主流的虚拟机都没有采用这种方法，而是采用第二种，可达性分析算法。
2. 可达性分析算法：如果一个对象从GC root，沿着引用链可以被搜索到，那么就是有用的，否则就是不能被使用，可能会被回收：GC root对象：1.方法中的局部变量，临时变量，参数等 2.引用类型静态变量，3.常量引用的对象，如字符串。4.本地方法引用的变量 5.虚拟机内部的引用，如常驻的异常对象（OOMError,NullPointError），基本类型对应的类对象。
3. 引用的类别

4种，分别是强引用，软引用，弱引用和虚引用

**强引用**：普遍存在的引用赋值，如Object a = new Object();永远不会被回收

**软引用**：还有用，但是非必须的对象，在即将OOM之前会被回收。

SoftReference<Object> a = new SoftReference<Object>(); 可以用来实现高速缓存。

**弱引用**：也是还有用，但非必须，比软引用要更弱一点。不会被主动回收，但是发生垃圾回收时就会被回收，无论空间是否够。可以解决HashMap中OOM的问题，将弱引用对象作为键。WeakReference<Object> a = new WeakReference<Object>();

**虚引用**：对对象的生存时间没影响。作用：在被回收的时候收到通知

1. 什么时候进行垃圾回收（对象分配规则）
2. 当对象实例化的时候查看eden区是否有足够的内存，如果有，直接放在eden区
3. 如果不足，就进行一个minor gc,回收eden区和survior区的对象，然后再判断eden区是否有足够的空间，有则直接放在eden区
4. 如果不足，就把部分存活对象放在survior区，再尝试放在eden区
5. 如果还是不足，向老年代发出请求，如果够，将部分存活对象放入老年区，将新对象放入eden区。
6. 如果还是不足，进行full gc，然后再进行判断
7. 如果还是不足，抛出OOM。
8. 什么时候会被回收

需要被两次标记才会真正被回收。第一次：当被**判定不可达**的时候，会被第一次标记。并加入回收队列中。第二次：当回收时,如果对象没有覆盖**finalize方法**或者已经被调用过，则被视为没有必要执行(覆盖的finalize方法，可以进行逃脱)。

1. 回收方法区

主要回收废弃的常量和不再使用的类型。

废弃的常量：如字符串常量“java”，没有任何字符串值指向该常量，即没有人使用，当垃圾回收时，可能会被回收。

类被回收：1.该类以及子类所有实例被回收。2.加载该类的类加载器被回收,此条件一般很难达成，因为一般是通过基类加载器加载的，只有是替换后的类加载器可能被回收 3.Class对象没有被引用，没有无法通过反射访问该类。满足后，允许被回收，对象是没有引用了肯定会被回收，类需要通过参数进行控制，设置类需要被回收才行。

1. 垃圾收集算法
2. 标记-清除算法

标记（通过GC root无法找到）所有需要回收的对象，统一回收所有被标记的对象.

优点：简单。

缺点：1.执行效率不稳定，当大部分需要被回收时，需要进行大量的标记。2.导致空间碎片太多，从而有大对象无法放入时，提前触发垃圾回收。

1. 标记-复制算法(新生代常用)

内存分为大小相等的两块，每次只使用一半，当回收时，将存活的对象连续复制到另一半。如果空间不足，需要老年代进行分配担保。

不过当时IBM公司曾经研究过，有98%的对象挨不过第一轮回收，因此并不需要1：1的空间，实际中为8:1:1,一个eden，两个survivor，新生代都在eden区，survivor为上一次存活的对象，当进行垃圾回收的时候，将存活的对象放到另外一个空的survivor区中，然后对eden和survivor进行回收，这样只有10%被浪费。不过由于10%，当存活对象超过10%的时候，就需要其他内存区域进行分配担保，通常是老年代，使存活对象直接进入老年代。

优点：无空间碎片

缺点：1.当大量对象存活的时候，需要进行大量内存空间复制的开销。2.浪费了一半的空间。

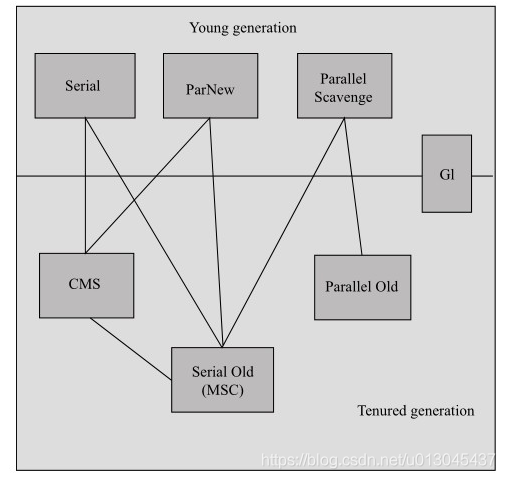
1. 标记-整理算法（老年代常用）

前面和标记-清除算法一样，不过为了解决空间碎片的问题，在清除之后会对对象进行整理，将对象向内存空间的一端进行移动。

优点：无空间碎片

缺点：移动对象并更新引用极为负担，并需要stop the world。

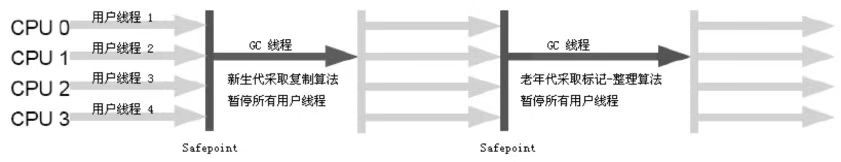
1. 垃圾收集器



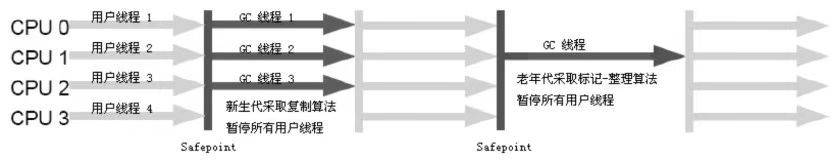
上图中7种作用于不同分代的收集器，如果两个之间存在连线，就说明他们可以搭配使用。收集器所处于的区域，则代表它是属于新生代收集器还是老年代收集器。

**新生代垃圾收集器：**

**Serial：**基于标记复制算法单线程收集，只会有一个CPU或者一个线程进行垃圾回收，当进行垃圾收集的时候，必须暂停其他所有线程（Stop the world）,直到收集结束。优点：额外内存消耗最小，简单而高效。缺点：产生停顿，用户体验不好。



**ParNew:**Serial的多线程版本，能够多线程进行新生代的垃圾收集，其他完全一致。



**Parallel Scavenge：**新生代收集器，也是多线程的，不过追求的是吞吐量，在短时间内完成垃圾回收的工作

**老年代垃圾收集器**

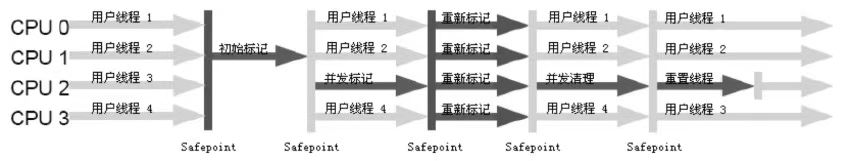
**Serial Old**:Serial 的老年代版本，同样是单线程收集器，会stop the world

**Paralled Old:** Parallel Scavenge的老年代版本，支持多线程收集，基于标记整理算法。

**CMS**：以最短回收停顿时间为目标，基于标记清除算法。

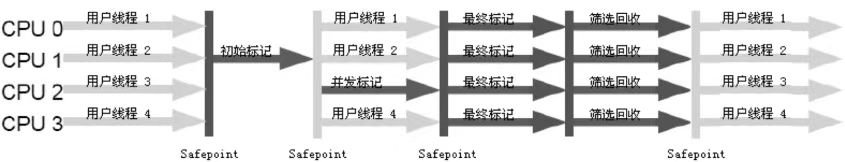
* 初始标记：Stop The World，仅标记和GC root直接关联的对象。
* 并发标记：多线程，遍历整个对象图，找出废弃对象，很耗时，但是不需要停顿。
* 重新标记：Stop The World，修正并发标记期间产生变动的对象。
* 并发清除：只使用一条 GC 线程，与用户线程并发执行，清除刚才标记的对象。这个过程非常耗时，但也不需要停顿

优点：并发收集，停顿低。缺点：标记清除算法，产生空间碎片，导致频繁full GC,吞吐量低。



**G1**:没有新生代和老年代之分，将堆划分为一块块的Region,每次垃圾回收的时候，首先估算每个Region区域的垃圾数量，每次从价值最大的开始回收：

* 初始标记：Stop The World，仅使用一条初始标记线程对所有与 GC Roots 直接关联的对象进行标记。
* 并发标记：使用**一条**标记线程与用户线程并发执行。此过程进行可达性分析，速度很慢。
* 最终标记：Stop The World，使用多条标记线程并发执行。
* 筛选回收：回收废弃对象，此时也要 Stop The World，并使用多条筛选回收线程并发执行。



1. 什么是类的加载机制

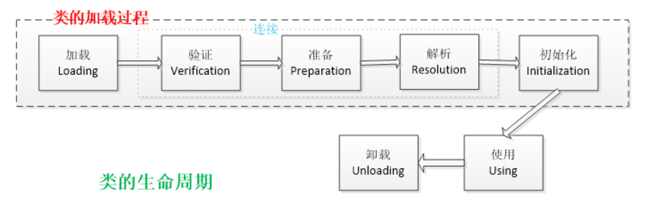
类加载机制就是把数据从class文件，类文件加载到内存，并堆数据进行校验，转换解析以及初始化，最终形成可以被虚拟机直接使用的java类型。

1. 类的生命周期

一个类从加载到内存，至被垃圾回收，会经历，加载，验证，准备，解析，初始化，使用，卸载7个阶段。其中验证，准备，解析统称为链接。

什么时候进行加载没有规定，但是什么时候进行初始化进行了规定：

* 1. 使用New实例化对象（new 数组不会被初始化）
  2. 读取或设置一个类型的静态字段(final修饰的除外，因为在编译期已经放入常量池)
  3. 调用静态方法
  4. 反射调用，但没有进行过初始化时
  5. 主类，有main方法的类
  6. 如果实现类的接口中有默认方法，那么该接口要先与实现类初始化。



**加载**：通过全限定类名获取此类的二进制字节流并将其转化为一个class文件。

**验证**：确保class文件的信息符合java虚拟机的全部约束要求

1. 文件格式验证：输入的字节流能正确的解析并存储在方法区。是否有不符合UTF-8的数据，是否有不符合的常量类型。
2. 元数据的验证：多态的验证，是否有父类，继承是否对（继承了final修饰的类），是否实现了抽象父类所有方法，重载是否对
3. 字节码验证：数据流，控制流，语义是否合法
4. 符号引用验证（在解析的时候发生）：该类是否缺少或者被禁止访问它依赖的某些外部类，方法，字段等资源。如：访问了不存在的类，类的私有方法。

**准备**：为类种的静态变量分配内存并且设置该类变量的初始值（不包括final，因为已经在编译阶段分配了）。类变量会在方法区中（JDK8之前，JDK8及以后都在堆中(物理内存上共享)，方法区只是一种逻辑表述），而实例变量会随着类在堆中。

**解析：**将常量池内的符号引用替换为直接引用的过程。符号引用：任何能够定位到目标的字面常量。直接引用：指向目标的指针，偏移量或者句柄，即和内存布局直接相关的。

1. 类和接口的解析
2. 字段解析
3. 方法解析

**初始化：**类的初始化就是执行类构造器的<init>方法，具体就是类**静态变量**的赋值还有**静态代码块(静态代码块只执行一次，普通代码块和构造器在每次new时候都会执行,** **因为静态的资源属于类所有，在类加载时就要分配内存空间。而非静态资源属于对象所有，只有在实例化时才会分配内存空间)**（静态代码块只能访问在其之前的变量）（结构不能使用静态代码块，但是有变量的初始化赋值操作，也会生成init方法，其并不需要先初始化父类的init方法，只有父类被使用的时候才会初始化）

1. 类加载器

类加载器的作用是把类加载进内存，两个对象必须是同一个类加载器加载的才相等（即使是同一个Class文件，同一个虚拟机加载）。

1. 类加载原理机制

java虚拟机通过双亲委派模型加载类。从虚拟机的角度，有两种类加载器

1. 启动类加载器（Bootstrap ClassLoader）:属于虚拟机的一部分，由c++实现。

负责加载<java\_home>\lib目录的类，处于安全考虑，只加载java,javax,sun开头的类，如Object类。

1. 其他类加载器，独立于虚拟机之外，都继承自抽象类,java.lang.ClassLoader。具体还可以分为两类。
   1. 扩展类加载器(ExtClassLoader,JDK 9以上是PlatformClassLoader),父类是启动类加载器。

负责加载<JAVA\_HOME >\lib\ext的目录中的类

* 1. 应用程序类（系统类）加载器

负责加载用户类路径(classpath)上所有的类库，我们自己写的类都是由这个类来加载

双亲委派模式：某个类加载器在收到类加载请求的时候，首先把加载任务委托给父类加载，如果父类不能加载，再由子类加载。越是基础的类由越是上层的加载器加载。**从而保证类在各种类加载器的环境中都是同一个类。**

**好处：保证java程序的稳定运作。**避免重复加载 + 避免核心类篡改。当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次

1. java对象结构
2. Java对象的定位方式

1. GC日志分析
2. 调优工具
3. 调优命令

1. JVM性能调优
2. 内存分配和回收策略