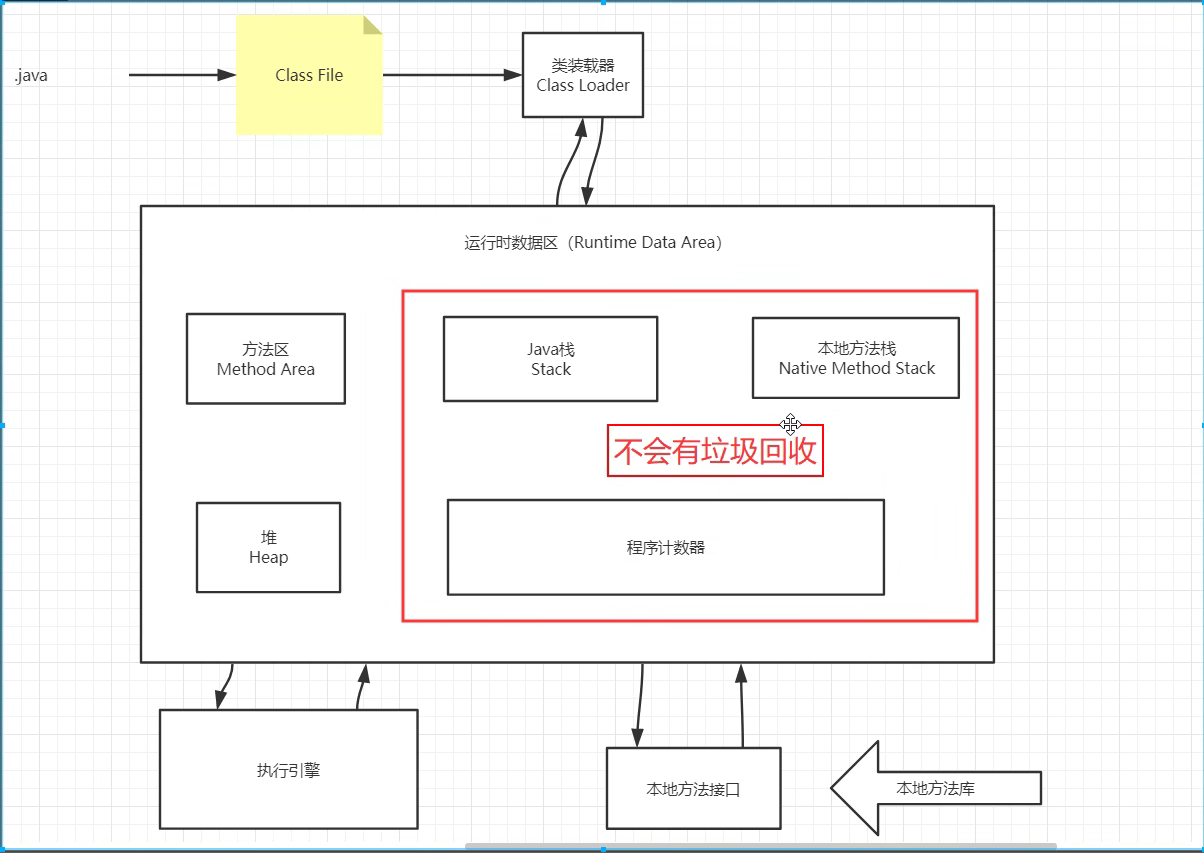
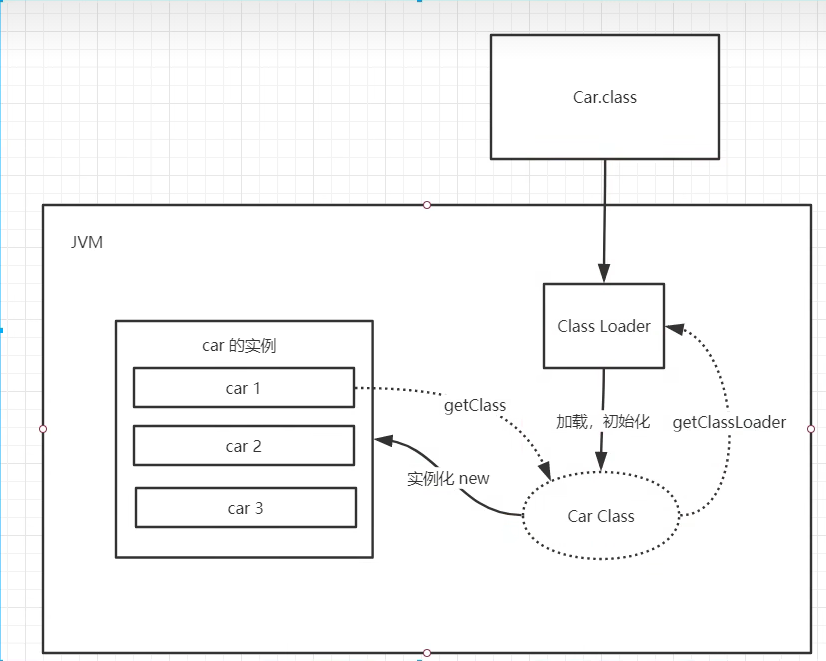
**JVM**

**jvm体系图**



JVM体系图

**类加载器**



类加载过程图

**类加载器种类**

Java中提供如下四种类型的加载器，每一种加载器都有指定的加载对象，具体如下

Bootstrap ClassLoader（启动类加载器） ：主要负责加载Java核心类库，%JRE\_HOME%\lib下的rt.jar、resources.jar、charsets.jar和class等。

Extention ClassLoader（扩展类加载器）：主要负责加载目录%JRE\_HOME%\lib\ext目录下的jar包和class文件。

Application ClassLoader（应用程序类加载器） ：主要负责加载当前应用的classpath下的所有类

User ClassLoader（用户自定义类加载器） ： 用户自定义的类加载器,可加载指定路径的class文件

**双亲委派机制定义**

当一个[类加载](https://so.csdn.net/so/search?q=%E7%B1%BB%E5%8A%A0%E8%BD%BD&spm=1001.2101.3001.7020)器收到了类加载的请求的时候，他不会直接去加载指定的类，而是把这个请求委托给自己的父加载器去加载。只有父加载器无法加载这个类的时候，才会由当前这个加载器来负责类的加载。

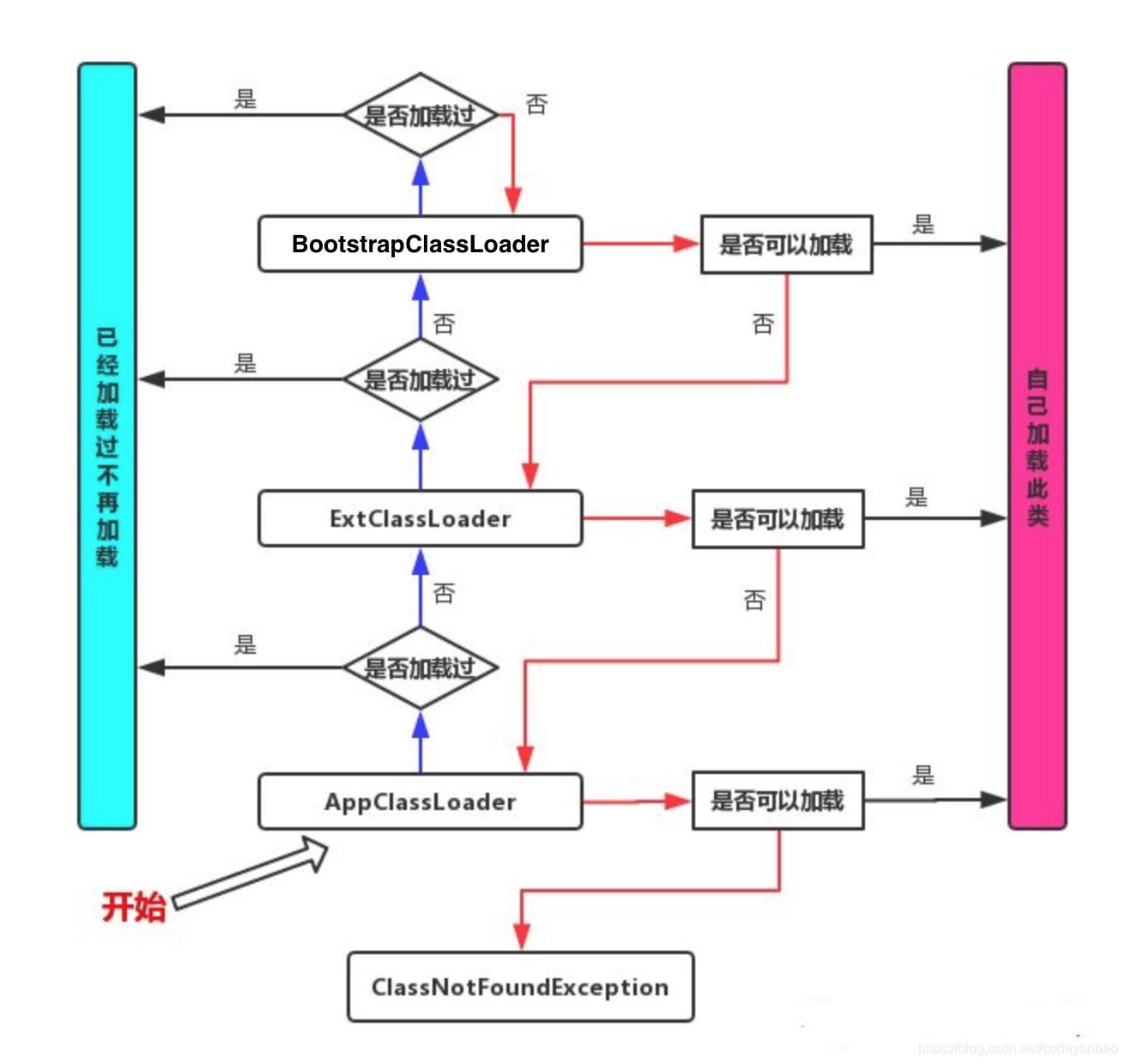
四种类加载器存在如下关系，当进行类加载的时候，虽然用户自定义类不会由bootstrap classloader或是extension classloader加载（由类加载器的加载范围决定），但是代码实现还是会一直委托到bootstrap classloader, 上层无法加载，再由下层是否可以加载，如果都无法加载，就会触发findclass,抛出classNotFoundException.

注意：这里存在的加载器之间的层级关系并不是以继承的方式存在的，而是以组合的方式处理的

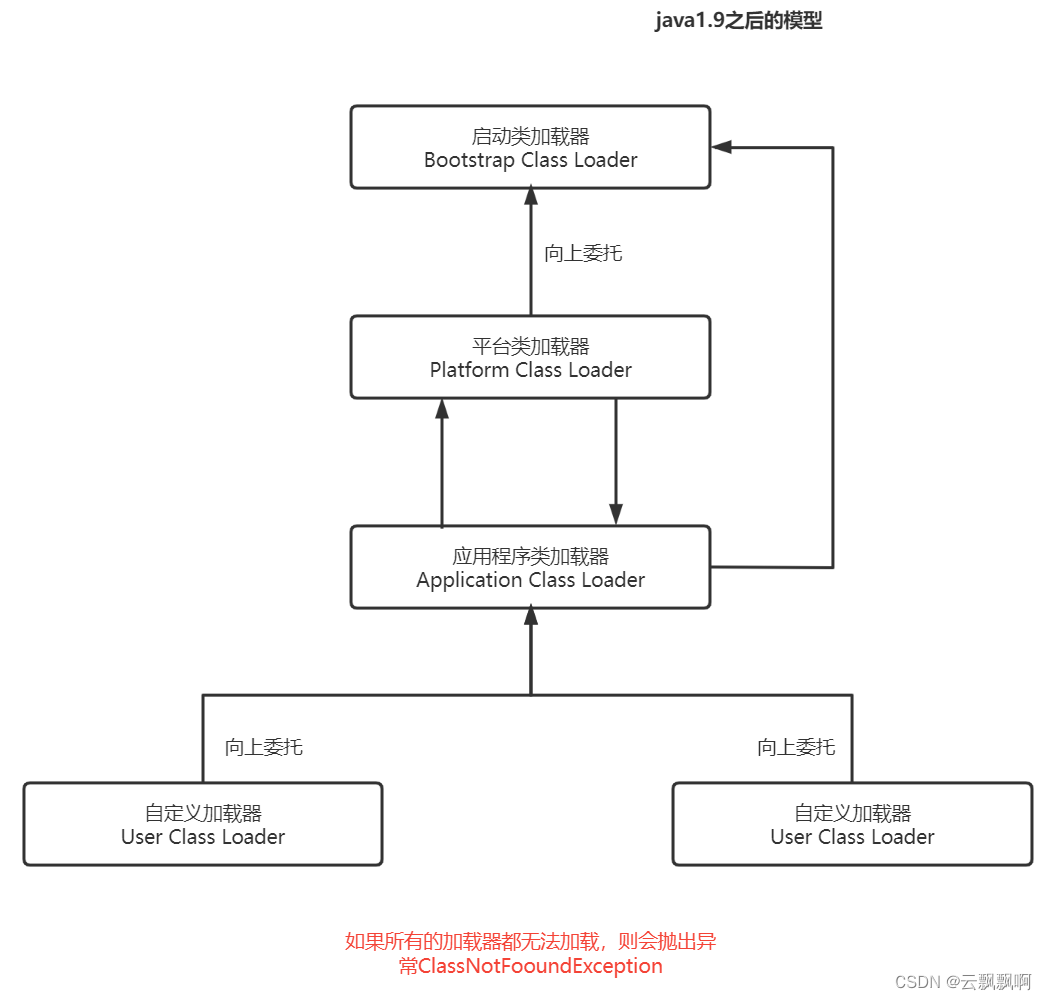
**双亲委派机制流程**

1. 首先会在AppClassLoader中检查是否加载过，如果有那就无需再加载了。
2. 如果没有，那么会拿到父加载器，然后调用父加载器的loadClass方法。父类中同理也会先检查自己是否已经加载过，如果没有再往上。注意这个类似递归的过程，直到到达Bootstrap classLoader之前，都是在检查是否加载过，并不会选择自己去加载。
3. 直到BootstrapClassLoader，已经没有父加载器了，这时候开始考虑自己是否能加载了，如果自己无法加载，会下沉到子加载器去加载，一直到最底层，如果没有任何加载器能加载，就会抛出ClassNotFoundException

、



jdk1.8即之前的



**双亲委派机制存在的意义：**

1.通过委派的方式，可以避免类的重复加载，当父加载器已经加载过某一个类时，子加载器就不会再重新加载这个类。

2.通过双亲委派的方式，还保证了安全性。因为Bootstrap ClassLoader在加载的时候，只会加载JAVA\_HOME中的jar包里面的类，如java.lang.Integer，那么这个类是不会被随意替换的，除非有人跑到你的机器上， 破坏你的JDK。那么，就可以避免有人自定义一个有破坏功能的java.lang.Integer被加载。这样可以有效的防止核心Java API被篡改。

双亲委派机制是在classLoader里的loadclass方法里实现的，

**Native**

1. native :凡是带了native 关键字的，说明java的作用范围达不到了，回去调用底层c语言的库!
2. 会进入本地方法栈
3. 调用本地方法本地接口JNI
4. JNI作用:扩展ava的使用，融合不同的编程语言为Java所用!最初: C、C++。
5. Java诞生的时候C、C++横行，想要立足，必须要有调用c、C++的程序~
6. 它在内存区域中专门开辟了一块标记区域:Native Method Stack，登记native方法/在最终执行的时候，加载本地方法库中的方法通过JNI

**PC寄存器**

程序计数器:Program Counter Register

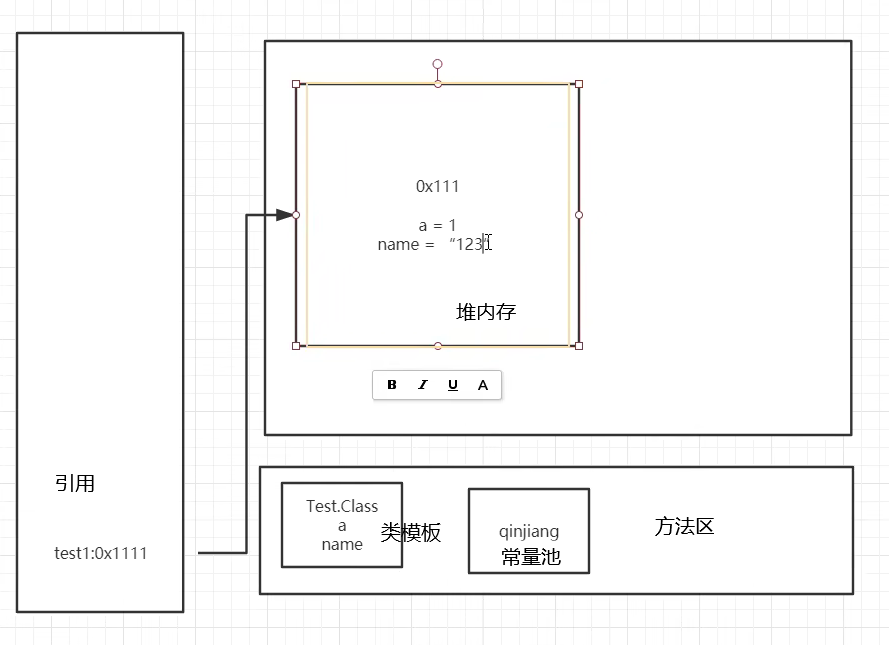
每个线程都有一个程序计数器，是线程私有的，就是一个指针，指向方法区中的方法字节码(用来存储指向像一条指令的地址，也即将要执行的指令代码)，在执行引擎读取下一条指令，是一个非常小的内存空间，几乎可以忽略不计

**方法区**

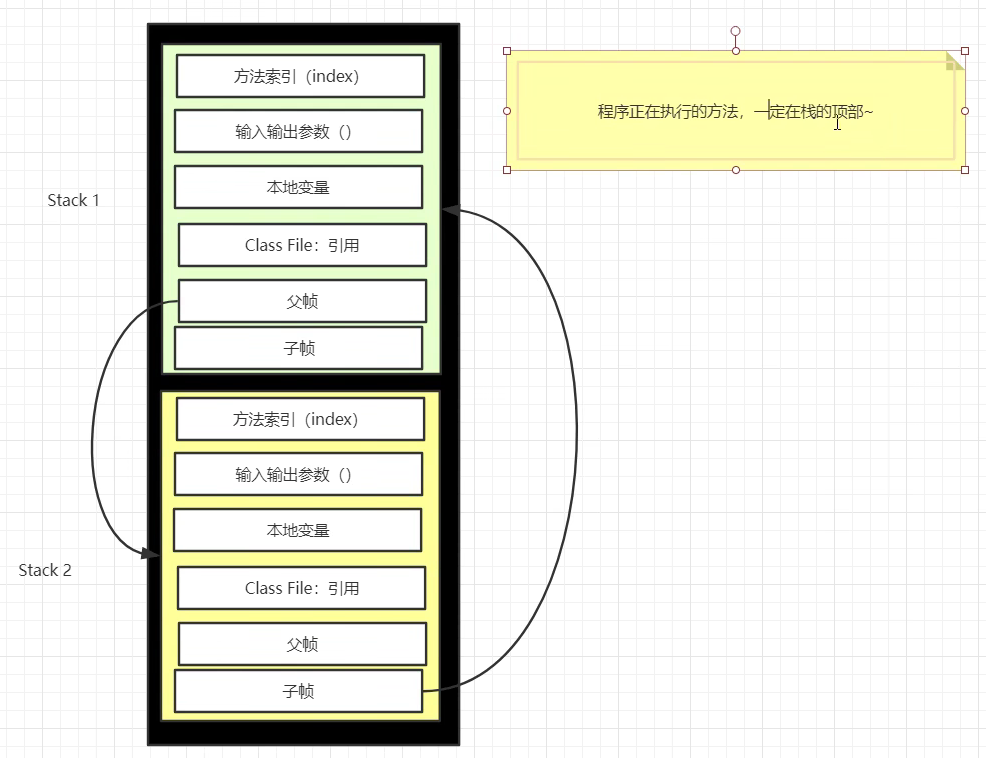
Method Area方法区

方法区是被所有线程共享，所有字段和方法字节码，以及一些特殊方法，如构造函数，接口代码也在此定义，简单说，所有定义的方法的信息都保存在该区域，此区域属于共享区间;

|  |
| --- |
| 静态变量、常量、类信息(构造方法、接口定义)、运行时的常量池存在方法区中，但是实例变量存在堆内存中，和方法区无关 |



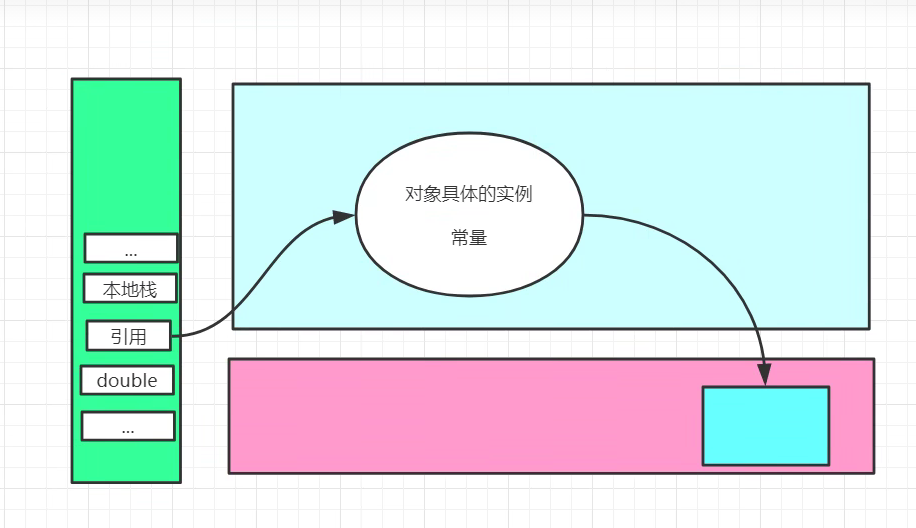
**栈**



栈结构

1. 栈:栈内存，主管程序的运行，生命周期和线程同步;
2. 线程结束，栈内存也就是释放，对于栈来说，不存在垃圾回收问题一旦线程结束，栈就Over!
3. 栈:8大基本类型＋对象引用＋实例的方法
4. 栈运行原理:栈帧
5. 栈满了: StackOverflowError
6. 栈有一个很重要的特殊性，就是存在栈中的数据可以共享

**栈+堆+方法区交互关系**



**堆**

Heap，一个JVM只有一个堆内存，堆内存的大小是可以调节的。

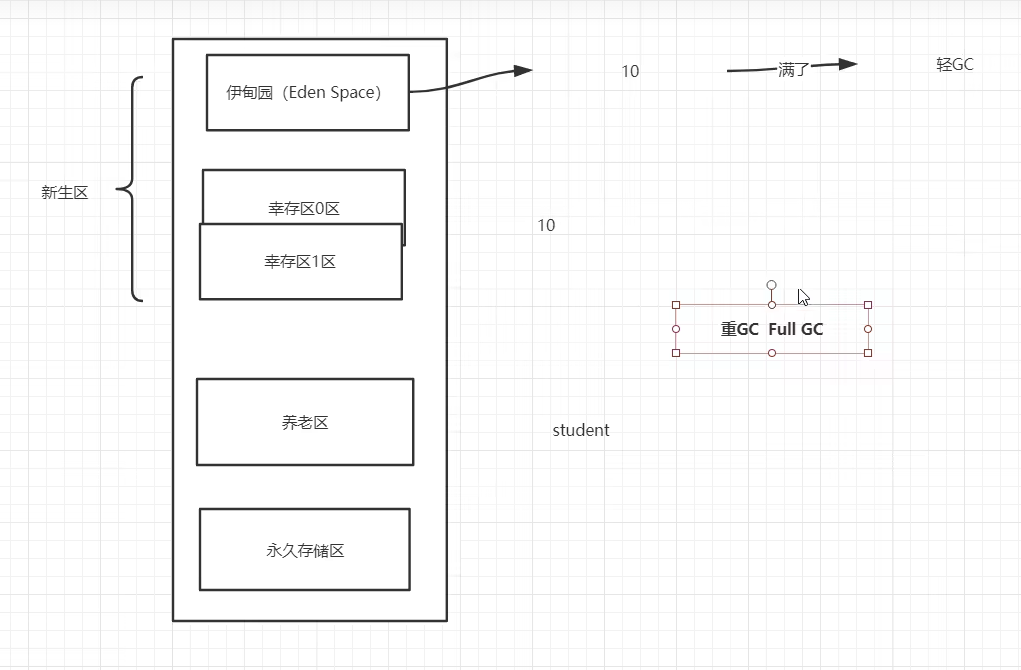
类加载器读取了类文件后，一般会把什么东西放到堆中?类，方法，常量，变量~，保存我们所有引用类型的真实对象;

堆内存中还要细分为三个区域:

**·新生区(伊甸园区)Young/New**

类:诞生和成长的地方，甚至死亡;

* 伊甸园，所有的对象都是在伊甸园区new出来的!
* 幸存者区(o,1)



伊甸园满了后触发轻GC，活下来的到幸存区，幸存区满了，触发重GC，活下来的到养老区，养老区也满了就OOM

**·养老区 old**

**·永久区 Perm**

真理:经过研究，99%的对象都是临时对象!

这个区域常驻内存的。用来存放IDK自身携带的Class对象。Interface元数据，存储的是Java运行时的一些环境或类信息~，这个区域不存在垃圾回收!关闭VM虚拟就会释放这个区域的内存~

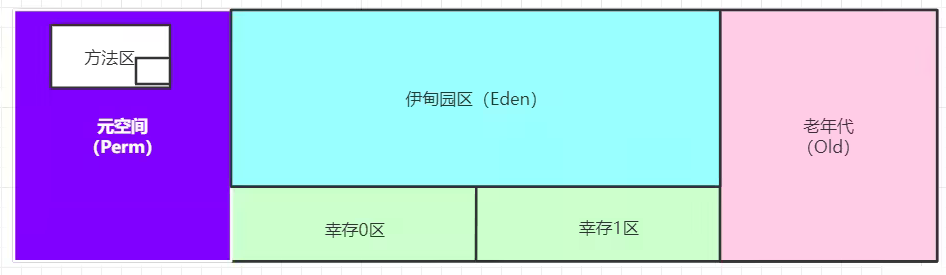
一个启动类，加载了大量的第三方jar包。Tomcat部署了太多的应用，大量动态生成的反射类。不断的被加载。直到内存满，就会出现OOM;

* jdk1.6之前:永久代，常量池是在方法区;
* jdk1.7:永久代，但是慢慢的退化了，去永久代，常量池在堆中.
* jdk1.8之后∶无永久代，常量池在元空间

GC垃圾回收，主要是在伊甸园区和养老区~

假设内存满了，OOM，堆内存不够! java.lang.OutOfMemoryError: Java heap space

在JDK8以后，永久存储区改了个名字(元空间);



堆空间构造

元空间：逻辑上存在，物理上不存在

默认情况下:分配的总内存是电脑内存的1/4，而初始化的内存:1/64

OOM:

1. 尝试扩大堆内存看结果
2. 分析内存，看一下那个地方出现了问题（专业工具)

在一个项目中，突然出现了OOM故障，那么该如何排除～研究为什么出错~

能够看到代码第几行出错:内存快照分析工具，MAT，Jprofiler

.Dubug，一行行分析代码!

MAT，Jprofiler 作用：

* ·分析Dump内存文件，快速定位内存泄露;
* ·获得堆中的数据
* ·获得大的对象~

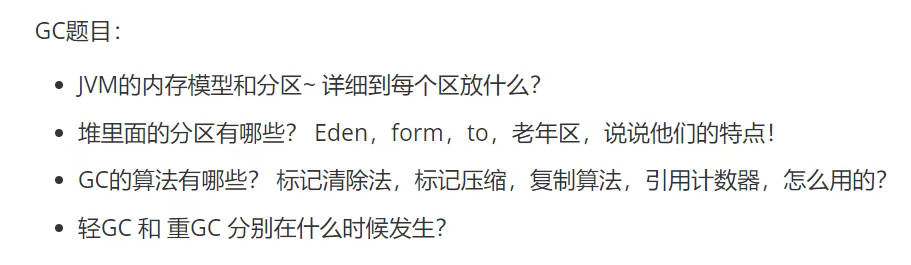
|  |
| --- |
| * -xms设置初始化内存分配大小/164-Xmx设置最大分配内存，默认1/4 * -XX : +PrintGCDetails 1/打日IGC垃圾回收信息l * -XX : +HeapDumponoutOfMemoryErrorl loom DUMP * -Xms1m -Xmx8m -XX :+HeapDumponoutOfMemoryError |

**垃圾回收GC**

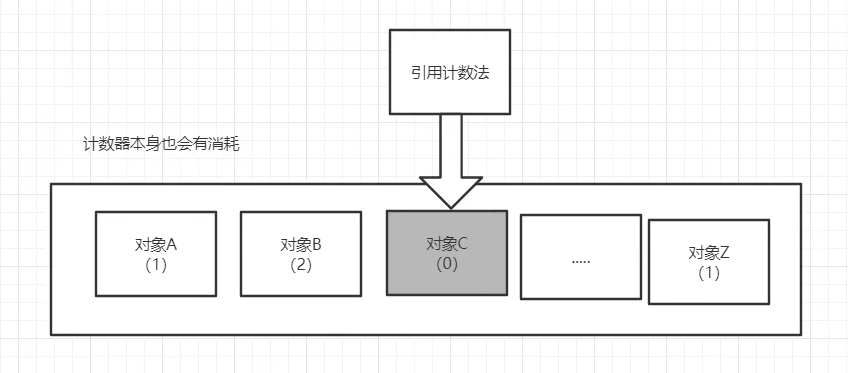
JVM在进行GC时，并不是对这三个区域统一回收。大部分时候，回收都是新生代~

* 新生代
* 幸存区(form , to)
* 老年区

GC两种类:轻GC(普通的GC)，重GC(全局GC)

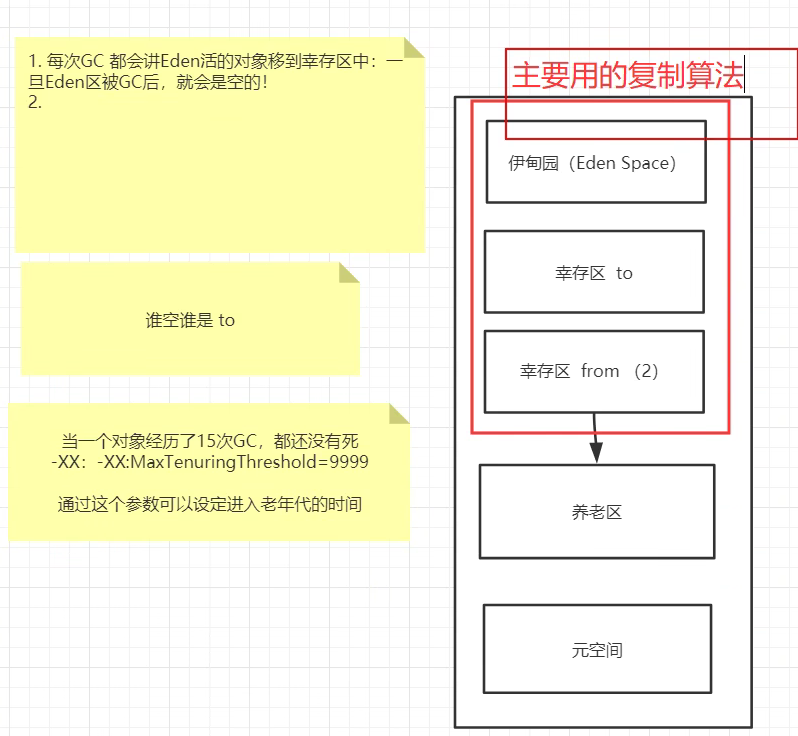


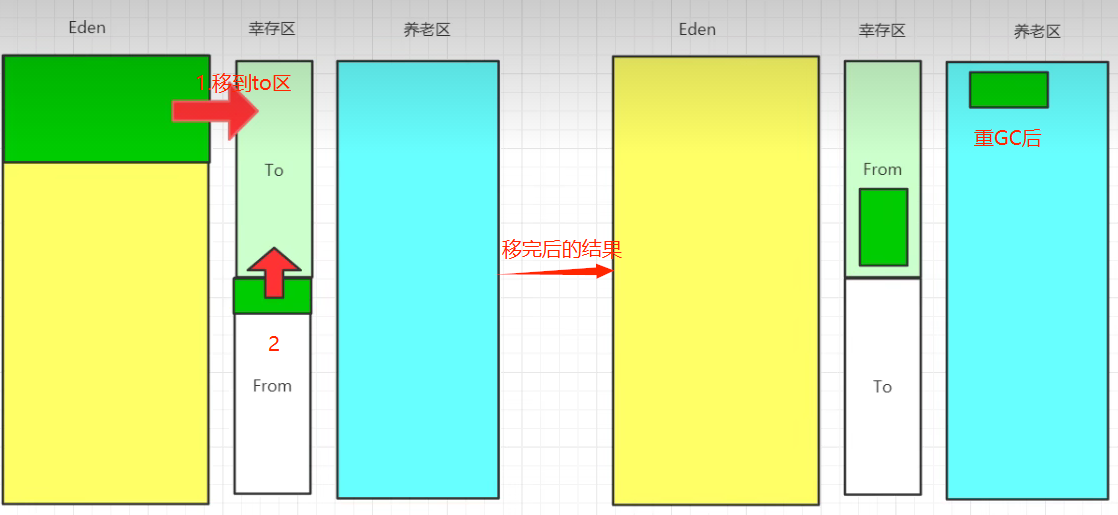
**引用计数法**



引用计数法

**复制算法**



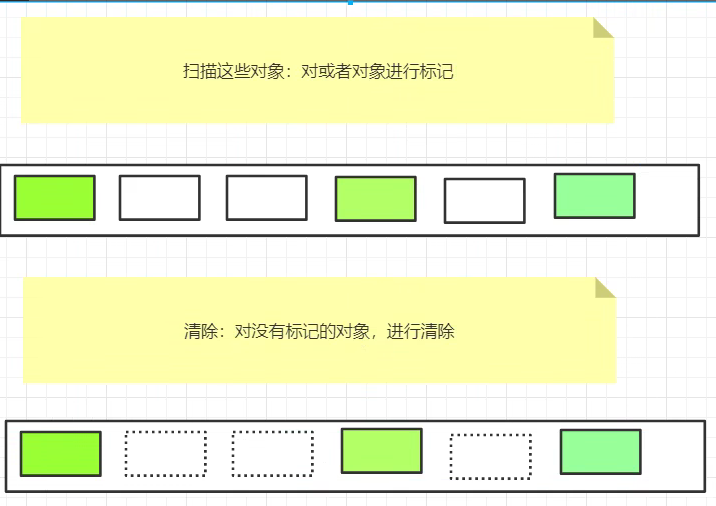


GC过程描述

* 好处:没有内存的碎片~
* 坏处:浪费了内存空间~:多了一半空间永远是空to。假设对象100%存活〈极端情况)

|  |
| --- |
| 复制算法最佳使用场景:对象存活度较低的时候;新生区~ |

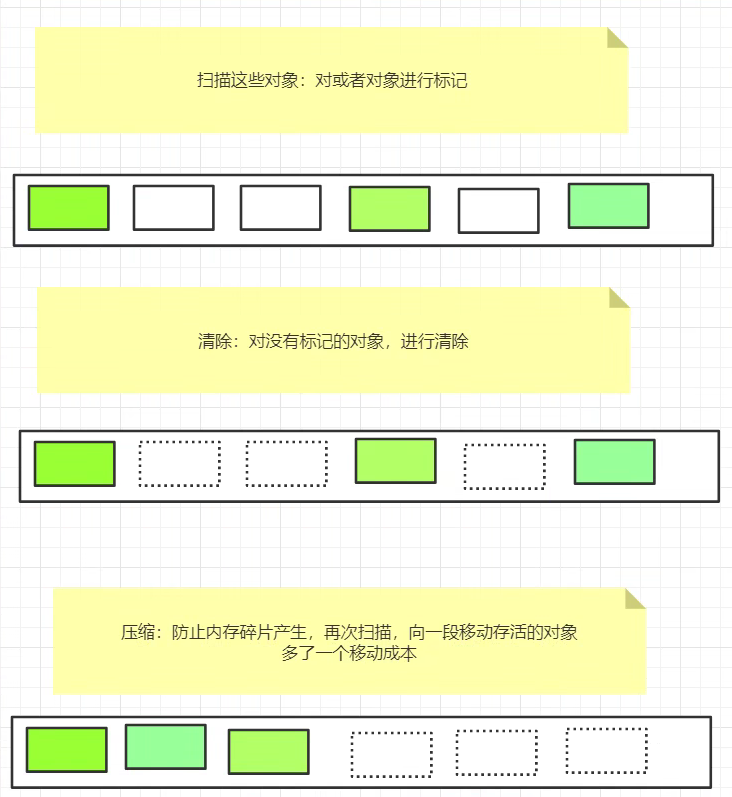
**标记清除算法**



* 优点:不需要额外的空间!
* 缺点:两次扫描，严重浪费时间，会产生内存碎片|。

**标记压缩算法**

对上面的再优化



**总结**

* 内存效率:复制算法>标记清除算法>标记压缩算法(时间复杂度)
* 内存整齐度:复制算法=标记压缩算法>标记清除算法
* 内存利用率:标记压缩算法=标记清除算法>复制算法

思考一个问题:难道没有最优算法吗?

答案:没有，没有最好的算法，只有最合适的算法----->GC:分代收集算法

|  |
| --- |
| 年轻代:  存活率低  复制算法! |

|  |
| --- |
| 老年代:  区域大:存活率  标记清除(内存碎片不是太多)+标记压缩混合实现 |

**如何学习**

