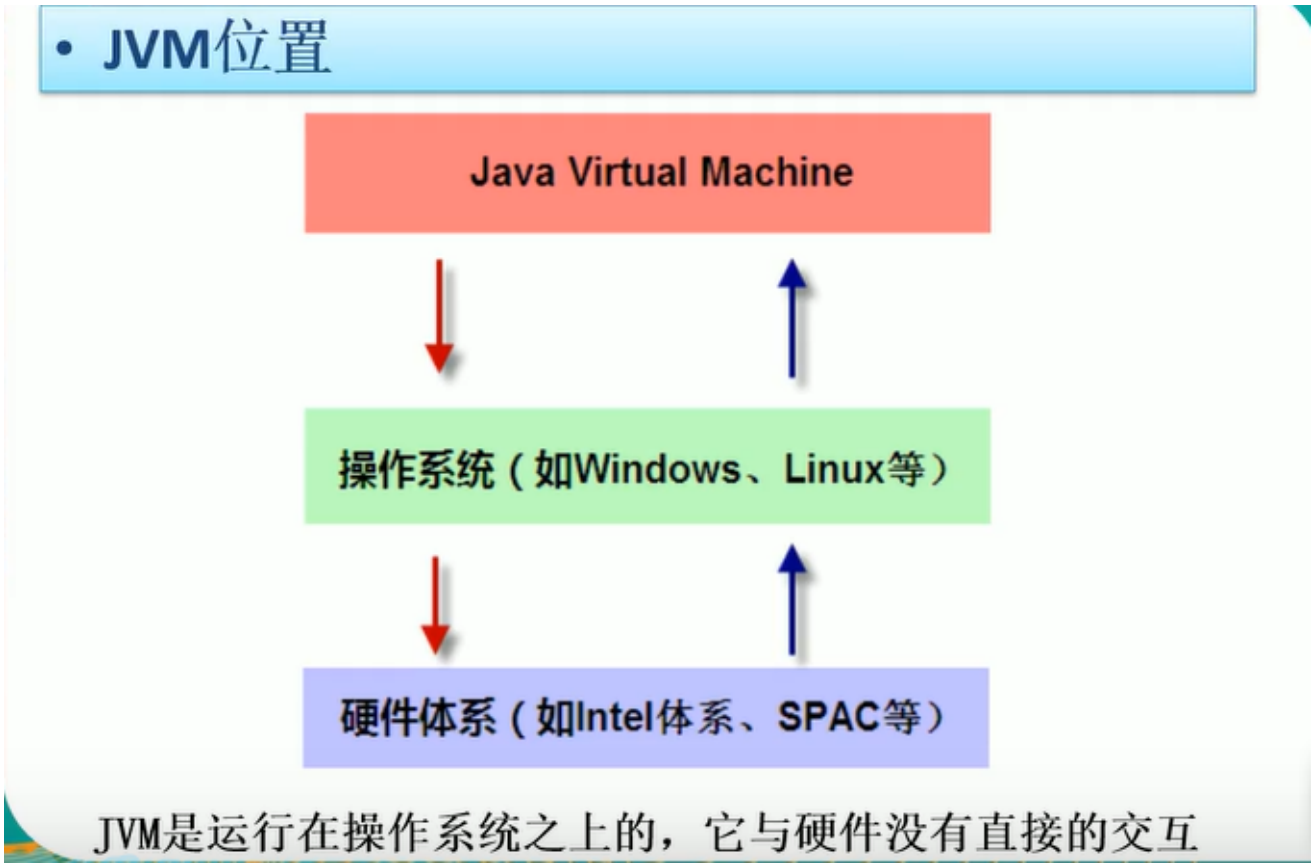
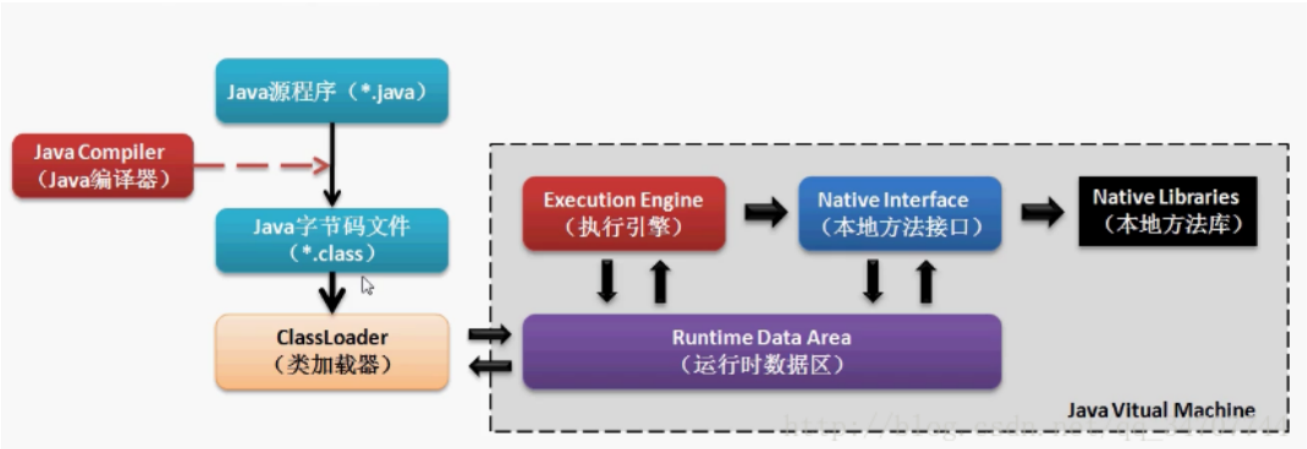
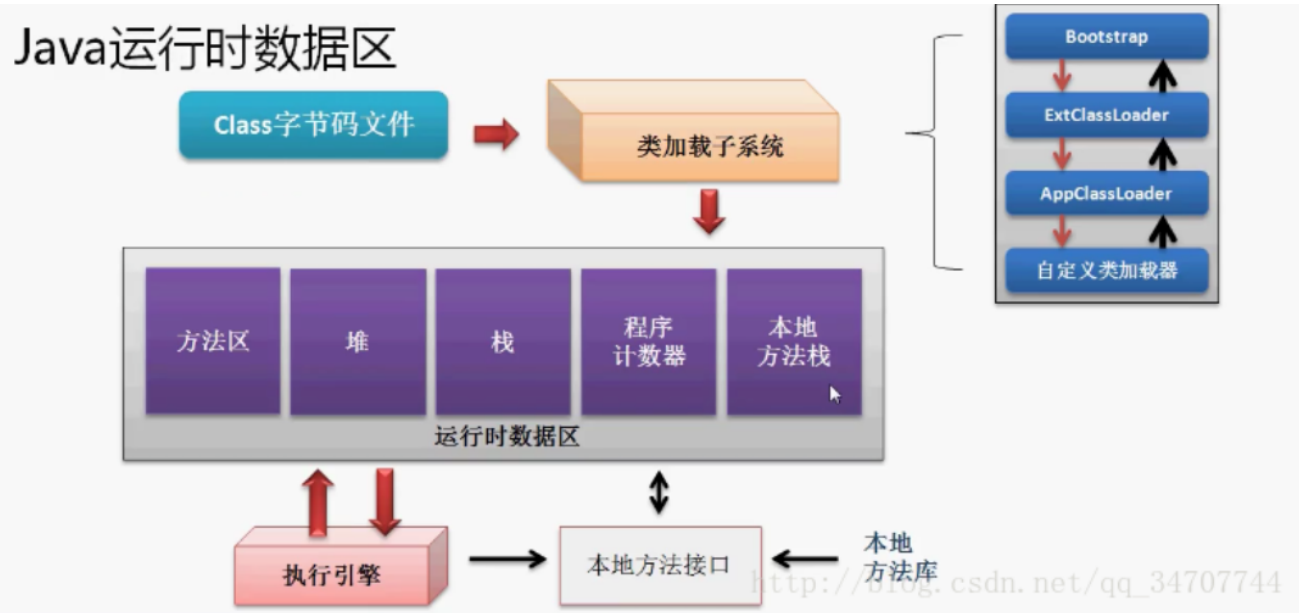
1、JVM是什么？Java运行原理？

1.1、JVM是Java虚拟机的缩写，是指负责将字节码解释成为特定的机器码进行运行，是运行在操作系统之上的，与硬件没有直接交互。



1.2、Java程序执行流程





堆：保存所有引用数据的地址信息

栈：基本类型、运算、指向堆空间指针

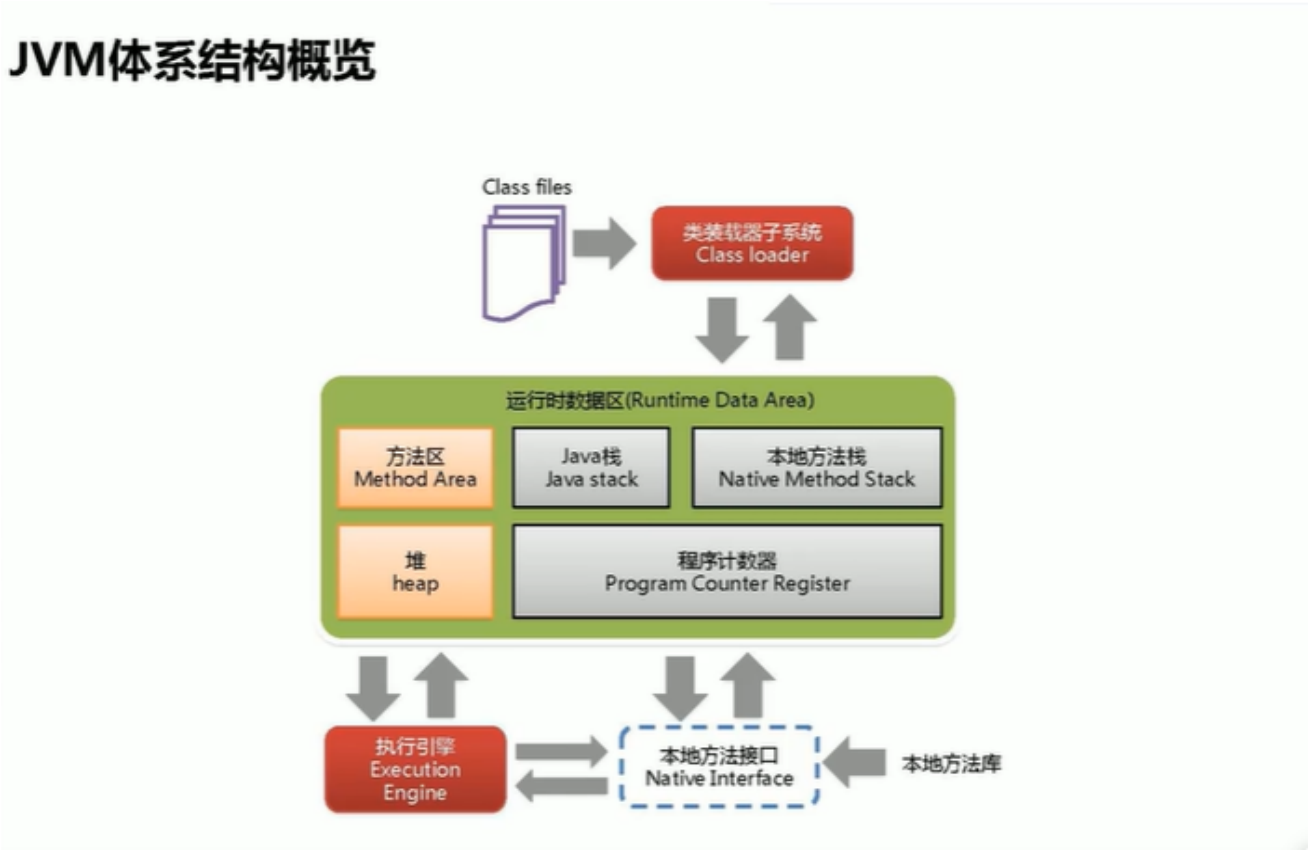
方法区：共享区、所有定义方法的信息均在这

程序计数器：是一块非常小的内存空间，用来保证程序依次执行

本地方法栈：每一次执行递归方法的时候，都会将上一个方法入栈

2、JVM模型

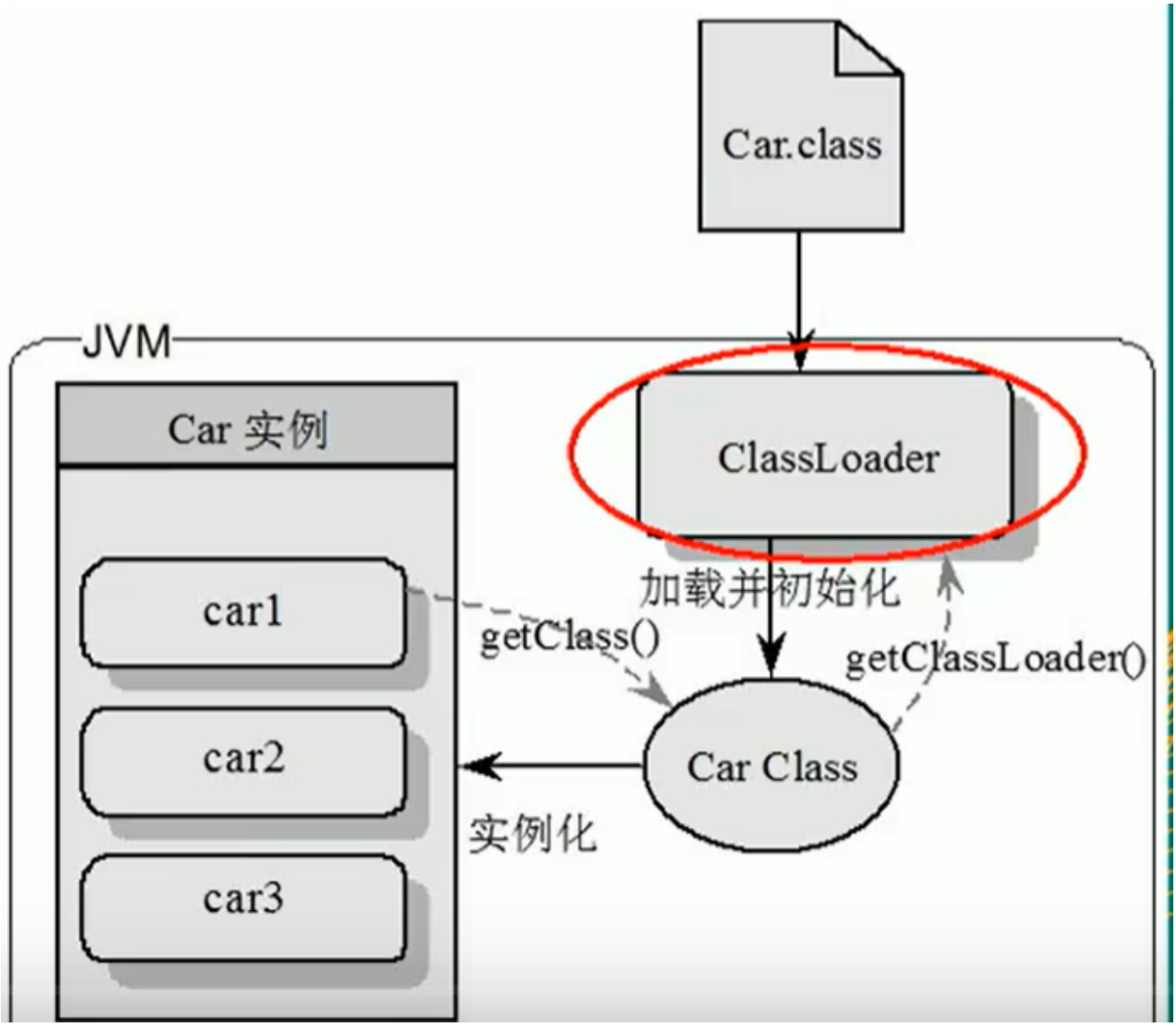
2.0、JVM体系概览



2.1、类加载器

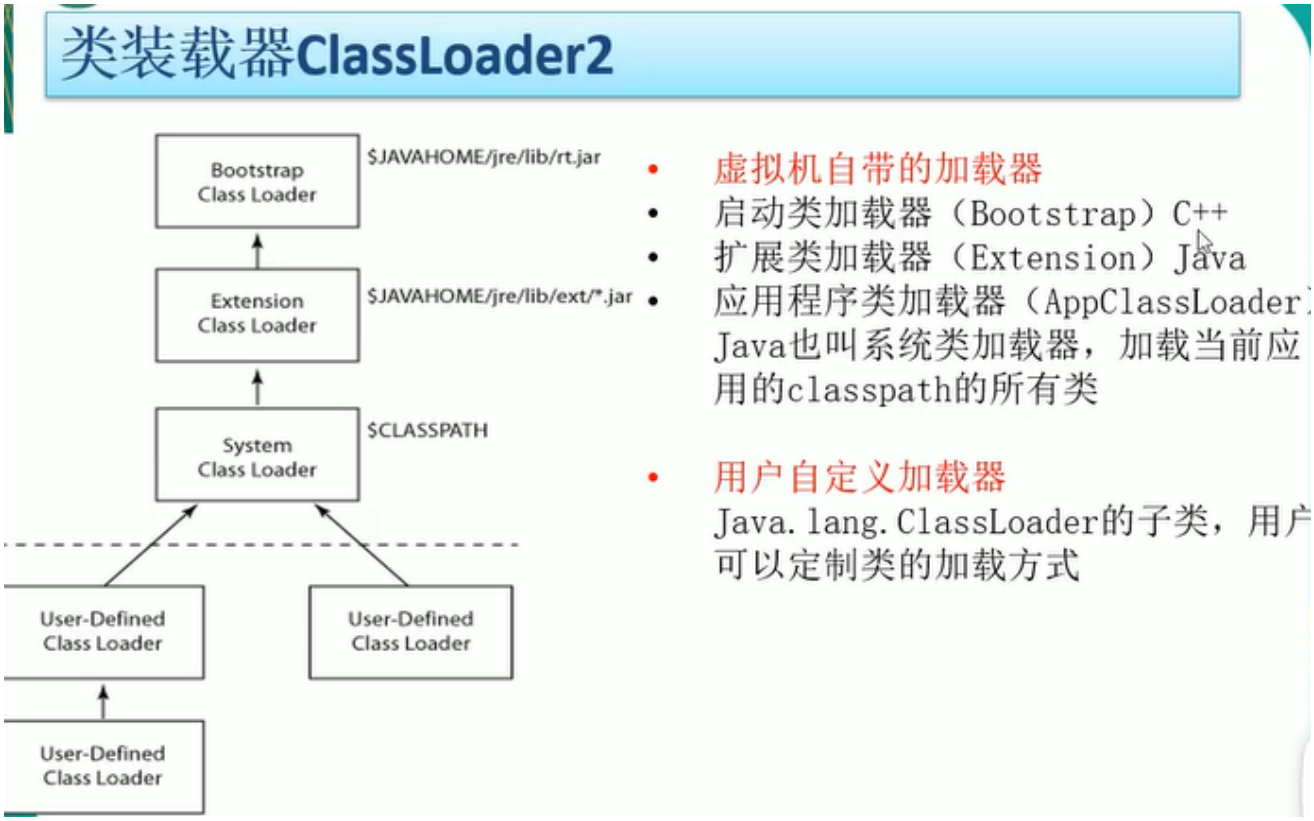
2.1.1、类加载器定义

类加载器，负责加载class文件，class文件在文件开头有特定的文件标识，将class字节码内容加载到内存中并将这些内容转换成方法区中的运行时数据结构，并且classLoader只负责class文件的加载，至于他是否运行，则由Execution Engine决定。



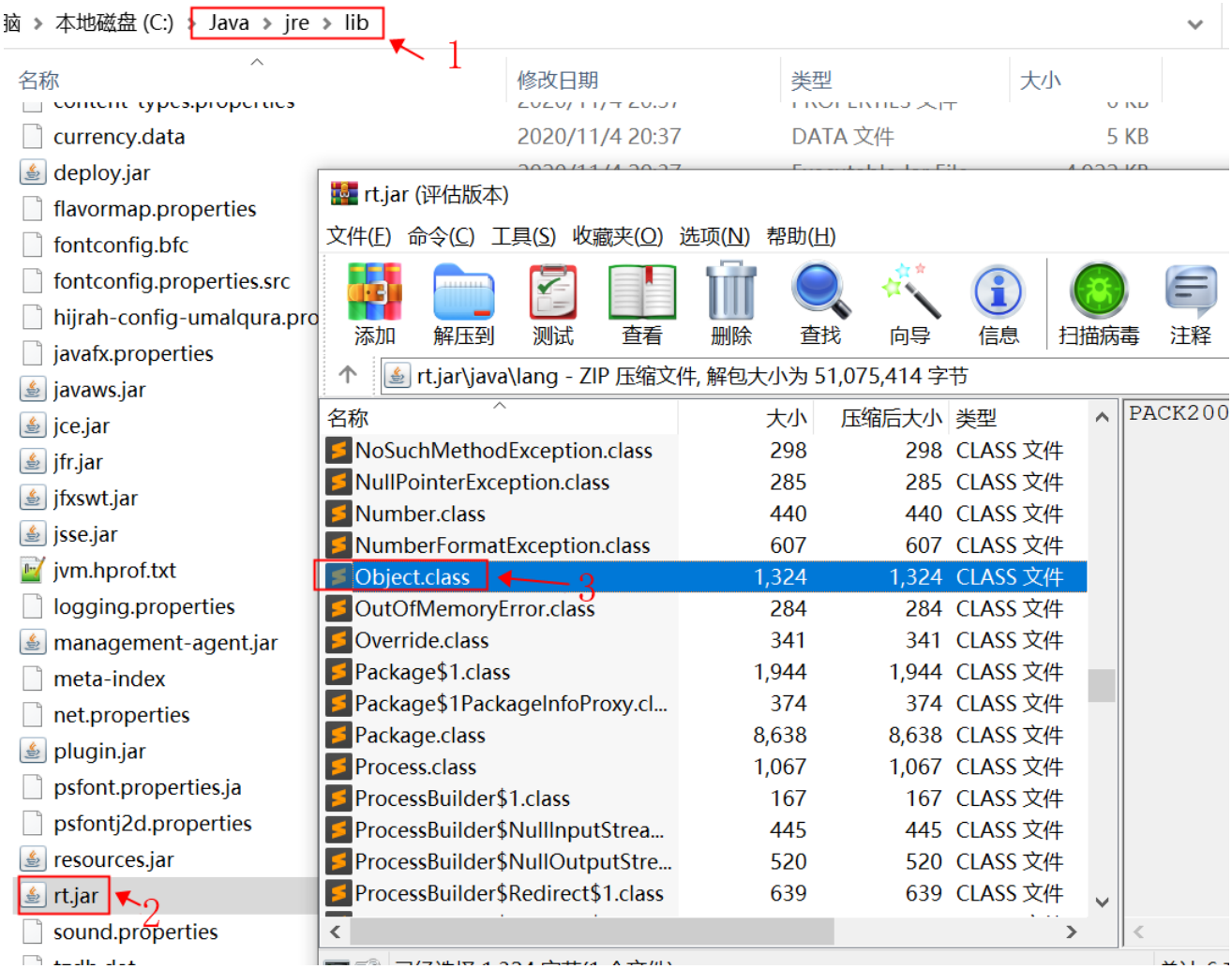
2.1.2、类加载器

2.1.2.1、类加载器概述


类装载器ClassLoader2
Bootstrap Class Loader
$JAVAHOME/jre/lib/rt.jar
Extension Class Loader
$JAVAHOME/jre/lib/ext/*.jar
System Class Loader
$CLASSPATH
User-Defined Class Loader
User-Defined Class Loader
User-Defined Class Loader
虚拟机自带的加载器
启动类加载器（Bootstrap）C++
扩展类加载器（Extension）Java
应用程序类加载器（AppClassLoader）
Java也叫系统类加载器，加载当前应用的classpath的所有类
用户自定义加载器
Java.lang.ClassLoader的子类，用户可以定制类的加载方式

2.1.2.2、类加载器作用

为什么Object、ArrayList及String等类可以直接使用？都是通过BootStrap（根类加载器）直接加载进来，因此可以直接使用。




```
Object object = new Object();
//直接获取为null，由于bootstrap 后台直接获取为null
System.out.println(object.getClass().getClassLoader());
//自定义类 MyJVM
MyJVM myJVM = new MyJVM() ;
//打印根类加载器 bootstrap ---> null
System.out.println(myJVM.getClass().getClassLoader().getParent().getParent());
//打印扩展类加载器 ExtClassLoader
System.out.println(myJVM.getClass().getClassLoader().getParent());
//打印程序类加载器 AppClassLoader
System.out.println(myJVM.getClass().getClassLoader());
```

2.1.2.3、类加载器执行机制

双亲委派机制：不停向上找（）

当一个类收到了类加载请求，他首先不会尝试自己去加载这个类，而是把这个请求委派给父类去完成，每一个层次的类加载器都是如此，因此所有的加载请求都应该传送到启动类加载器中，只有当父类加载器反馈自己无法完成这个请求时（即在他的加载路径下没有找到所需加载的class），子类加载器才会去尝试自己去加载。

采用双亲委派机制的一个好处是，比如加载位于rt.jar包中的类Java.lang.String。不管哪个类加载器加载这个类，最终都是委托给顶层的启动类加载器进行加载，这样就保证了使用不同的类加载器最终得到的都是同一个String对象。

沙箱安全机制：建议参考博客（!--https://www.cnblogs.com/yanl55555/p/12625332.html--）

2.2、运行时数据区

2.2.1、栈--运行 （线程私有）

程序 = 算法 + 数据结构

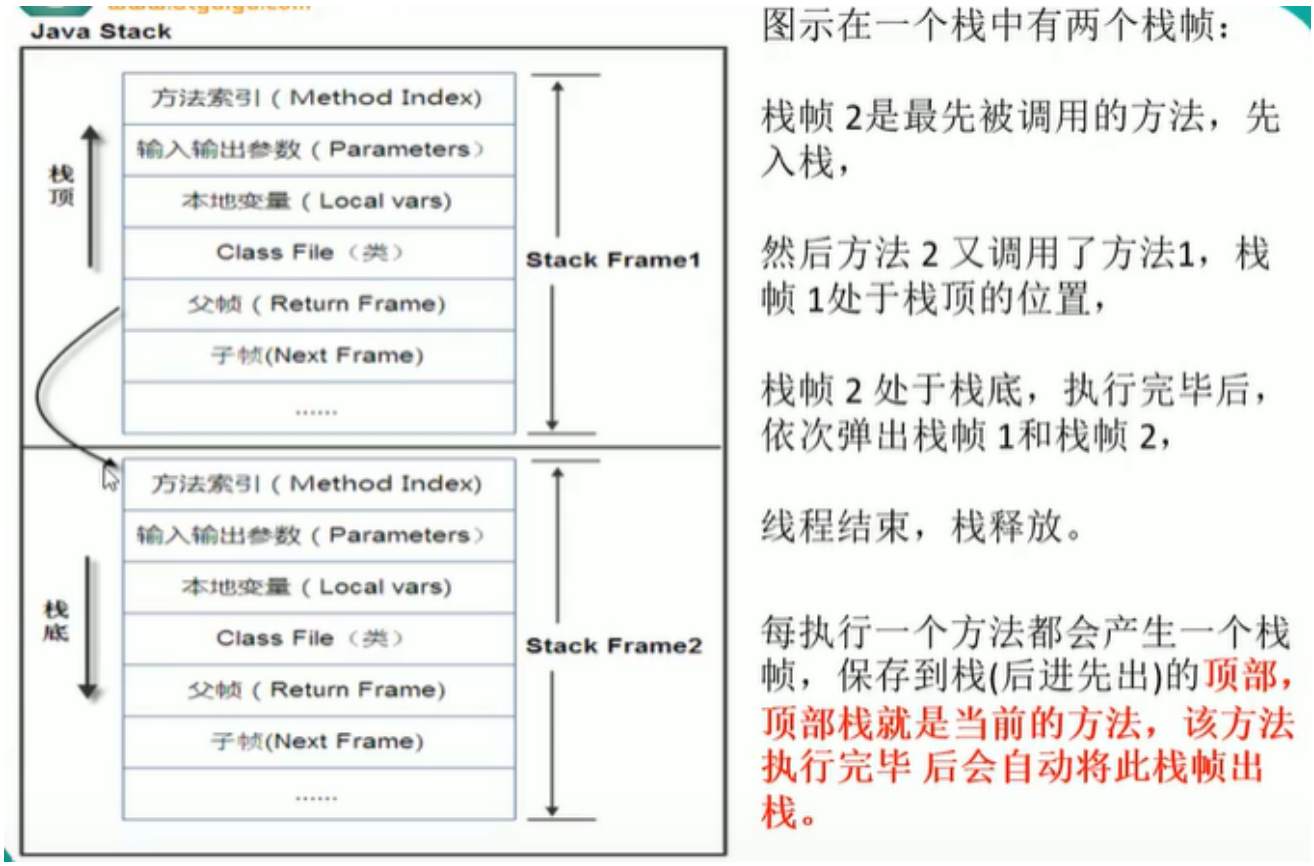
程序 = 框架 + 业务逻辑

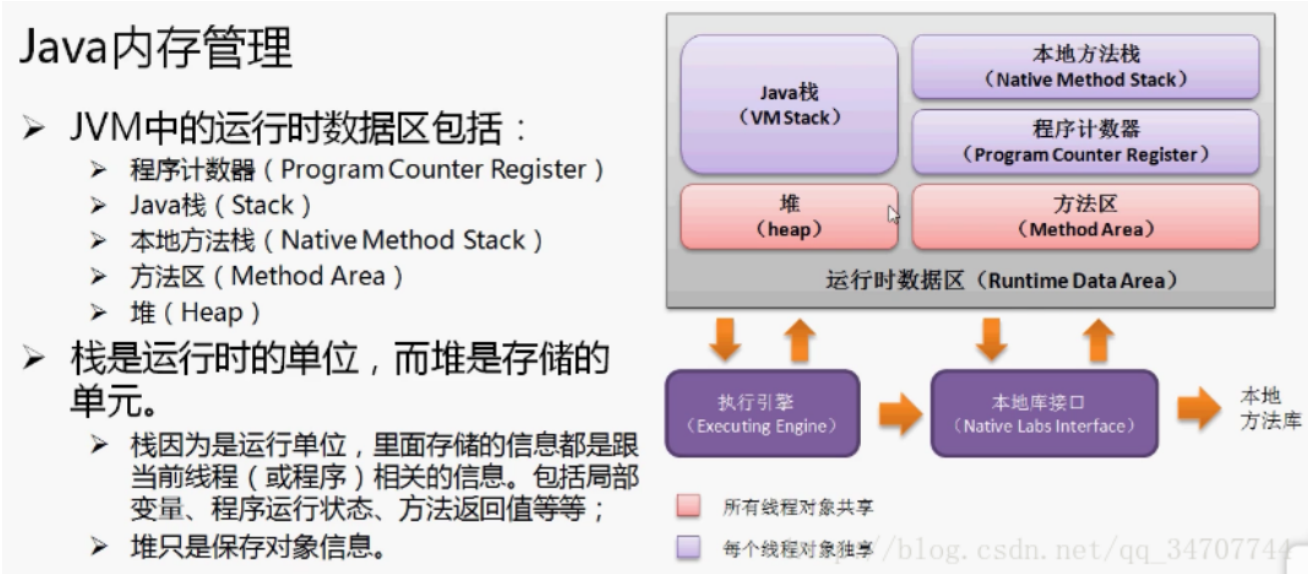
栈：先进后出（FILO） 队列：先进先出（FIFO）

栈也叫栈内存，主管Java程序的运行，是在线程创建时创建，他的生命周期是跟随线程的生命周期，线程结束栈内存也就释放，对于栈内存来说不存在垃圾回收问题，只要线程已结束该栈就over，生命周期和线程一致，是线程私有的。**8中基本数据类型+对象的引用变量+实例方法都是在函数的栈内存中分配。**

栈内包含栈帧：栈帧主要保存3类数据 **本地变量：**输入参数和输出参数以及方法内的变量 **栈操作：**记录出栈、入栈操作 **栈帧数据：**包括类文件、方法等。

每一个方法执行的同时都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至执行完毕过程，就对应着一个栈帧在虚拟机中入栈到出栈过程。栈的大小和JVM的实现有关，通常在256k~756k之间，约等于1Mb左右。





局部变量表（Local Variables）:方法的局部变量或形参，其以变量槽（solt）为最小单位，只允许保存32为长度的变量，如果超过32位则会开辟两个连续的solt(64位长度，long和double)；

操作树栈（Operand Stack）：表达式计算在栈中完成；

指向当前方法所属的类的运行时常量池的引用（Reference to runtime constant pool）：引用其他类的常量或者使用String 池中的字符串；**方法返回地址（Return Address）**：方法执行完后需要返回调用此方法的位置，所以需要再栈帧中保存方法返回地址。

注：在整个java之中存在对象池的概念，对象池是对整个常量池的一个规则破坏，因为在jvm启动时，所有的常量都已经分配好的内存空间了，但是String中的intern（）方法会打破这种限制，动态地进行常量池的内容设置。

```
public static void main(String[] args) {
    System.out.println("************************");
    m1() ;
}
//throws InterruptedException
// 错误 ： java.lang.StackOverflowError
public static void m1() {
    m1();
}
```

2.2.2、堆 -- 存储（线程共享）

2.2.2.0、堆结构概览

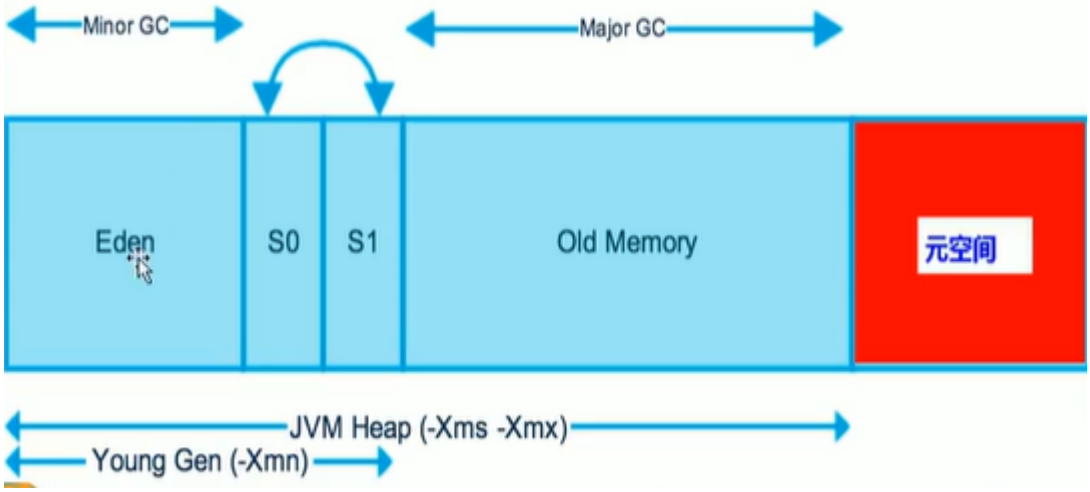
Heap堆（Java7之前）

一个JVM实例只存在一个堆内存，堆内存的大小是可以调节的。类加载器读取了类文件后，需要把类、方法、常变量放到堆内存中，保存所有引用类型的真实信息，以方便执行器执行。

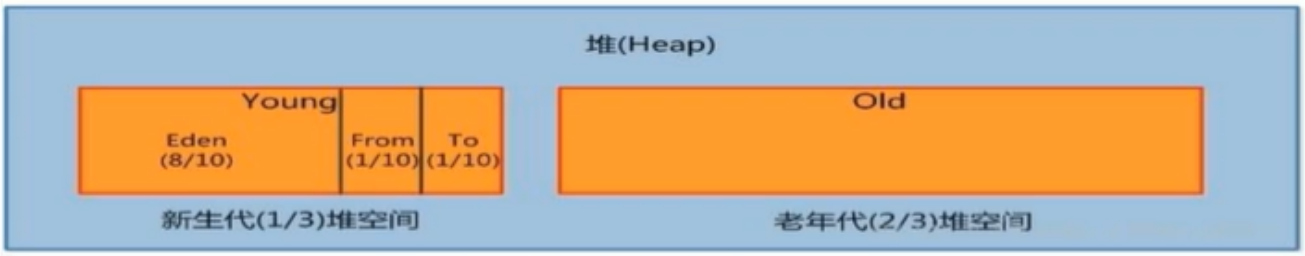
堆内存逻辑上分为三部分：新生+养老+永久

Java8

JDK 1.8之后将最初的永久代取消了，由元空间取代。



Java 堆从 GC 的角度还可以细分为: **新生代**(*Eden 区、 From Survivor 区*和 *To Survivor 区*)和**老年代**。



MinorGC的过程（复制->清空->互换）

1：eden、SurvivorFrom 复制到 SurvivorTo，年龄+1

首先，当Eden区满的时候会触发第一次GC,把还活着的对象拷贝到SurvivorFrom区，当Eden区再次触发GC的时候会扫描Eden区和From区域,对这两个区域进行垃圾回收，经过这次回收后还存活的对象,则直接复制到To区域（如果有对象的年龄已经达到了老年的标准，则赋值到老年代区），同时把这些对象的年龄+1

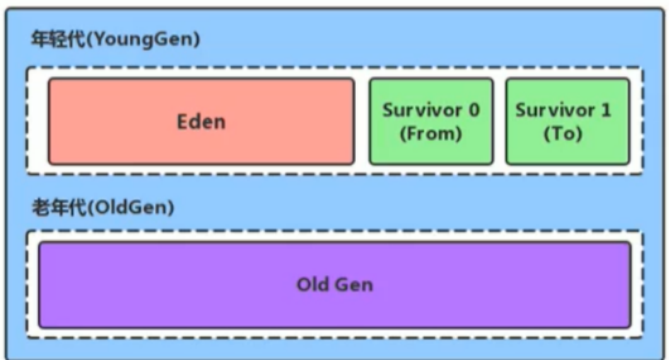
2：清空 eden、SurvivorFrom

然后，清空Eden和SurvivorFrom中的对象，也即复制之后有交换，谁空谁是to

3：SurvivorTo和 SurvivorFrom 互换

最后，SurvivorTo和SurvivorFrom互换，原SurvivorTo成为下一次GC时的SurvivorFrom区。部分对象会在From和To区域中复制来复制去,如此交换15次(由JVM参数MaxTenuringThreshold决定,这个参数默认是15),最终如果还是存活,就存入到老年代

- 存储在JVM中的Java对象可以被划分为两类：
  - 一类是生命周期较短的瞬时对象，这类对象的创建和消亡都非常迅速
  - 另外一类对象的生命周期却非常长，在某些极端的情况下还能够与JVM的生命周期保持一致。
- Java堆区进一步细分的话，可以划分为年轻代（YoungGen）和老年代（OldGen）
- 其中年轻代又可以划分为Eden空间、Survivor0空间和Survivor1空间（有时也叫做from区、to区）。



2.2.2.1、年轻代 （1/3占整个堆大小）

伊甸区：new 出来的对象存放地 幸存一区：伊甸区第一次gc后存活对象通过复制算法移动到from区，同时为下一次gc做角色转换（to）准备 幸存二区：角色转换为from区，同时为下一次gc做角色转换准备

注：大小比例：8：1：1

2.2.2.2、老年代（2/3占整个堆大小）

在年轻代中年龄超过15的存放到年老代，年老代满了则会进行full GC

2.2.2.3、元空间

即方法区的一个实现

2.2.3、方法区 （线程共享）

供各线程共享的运行时内存区域，**他存储了每一个类的结构信息**，例运行时常量池、字段和方法数据、构造函数和普通方法的字节码内容。**注：实例变量存储在堆内存中，和方法区无关**

方法区是规范，在不同的虚拟机里面有不同的实现，典型案例Java7的永久代和Java8的元空间

```
方法区 f = new 永久代 () ;
方法区 f = new 元空间 () ;
```

2.2.4、程序计数器（pc寄存器）（线程私有）

总结：类似于课程表，排班值日表

每个线程都有一个程序计数器，是线程私有得，就是一个指针，指向方法区中得方法字节码（用来存储指向下一条指令得地址，即将要执行得指令代码），由执行引擎读取下一条指令，是一块非常小得内存空间，几乎可以忽略不计。

这块内存区域很小，他是当前线程所执行的字节码的行号指示器，字节码解释器通过改变这个计数器的值来选取下一条需要执行的字节码指令。

如果执行的是native方法，那么这个计数器为空

用来完成分支、循环、跳转、异常处理、线程恢复等基础功能，不会发生内存溢出（OutOfMemory=OOM）错误。

2.2.5、本地方法栈（线程私有）

对应着本地接口，具体做法是Native Method Stack 中登记native方法，在Execution Engine执行时加载本地方法库。

2.3、本地接口

native：关键字修饰得方法 本地接口得作用是融合不同得编程语言为Java所用，他的初衷是融合c/c++程序，Java诞生得时候是c/c++横行得时候，要想立足必须要调用c/c++程序，于是就在内存中专门开辟了一块区域处理标记为native得代码，他的具体做法是native method Stack中登记native方法，在Execution Engine执行得时候加载native libraries。**注：目前除非是与硬件有关得应用，企业应用中特别少见。**

2.4、执行器

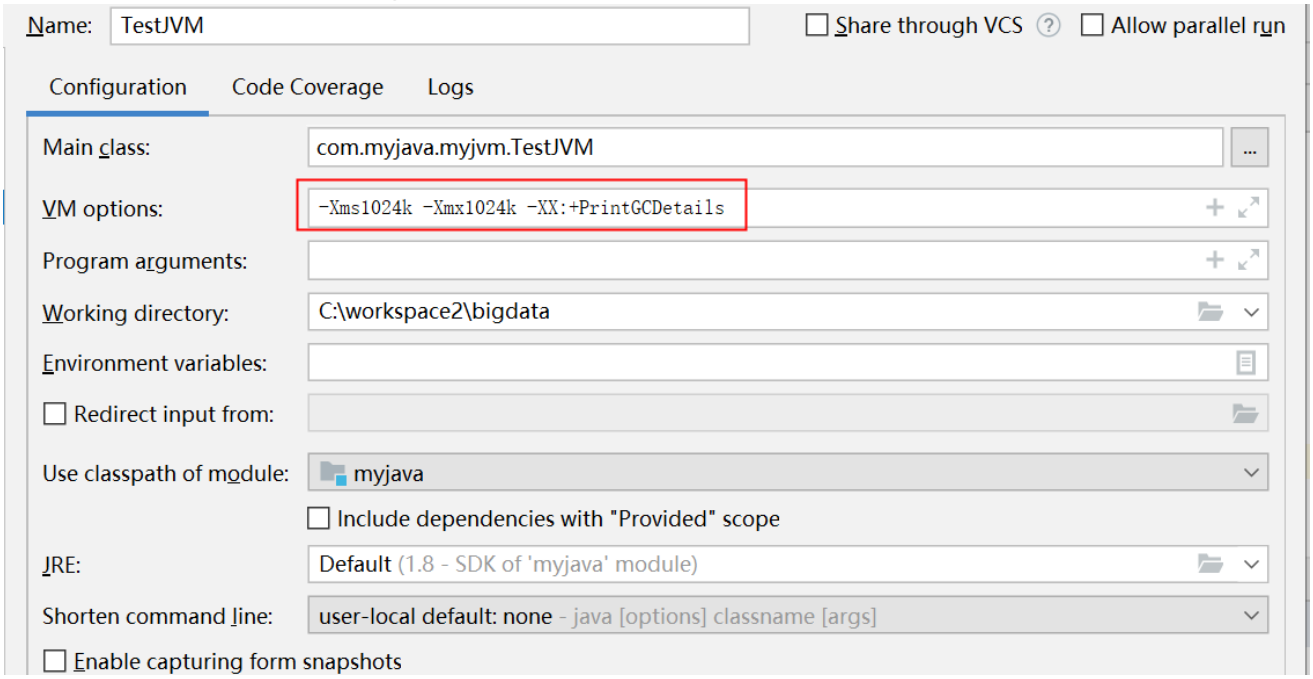
负责解释命令，提交操作系统执行

3、JVM参数

3.1、堆参数

-Xms：设置堆空间（年轻代+年老代）的初始内存大小 **-Xmx：**设置堆空间（年轻代+年老代）的最大内存大小

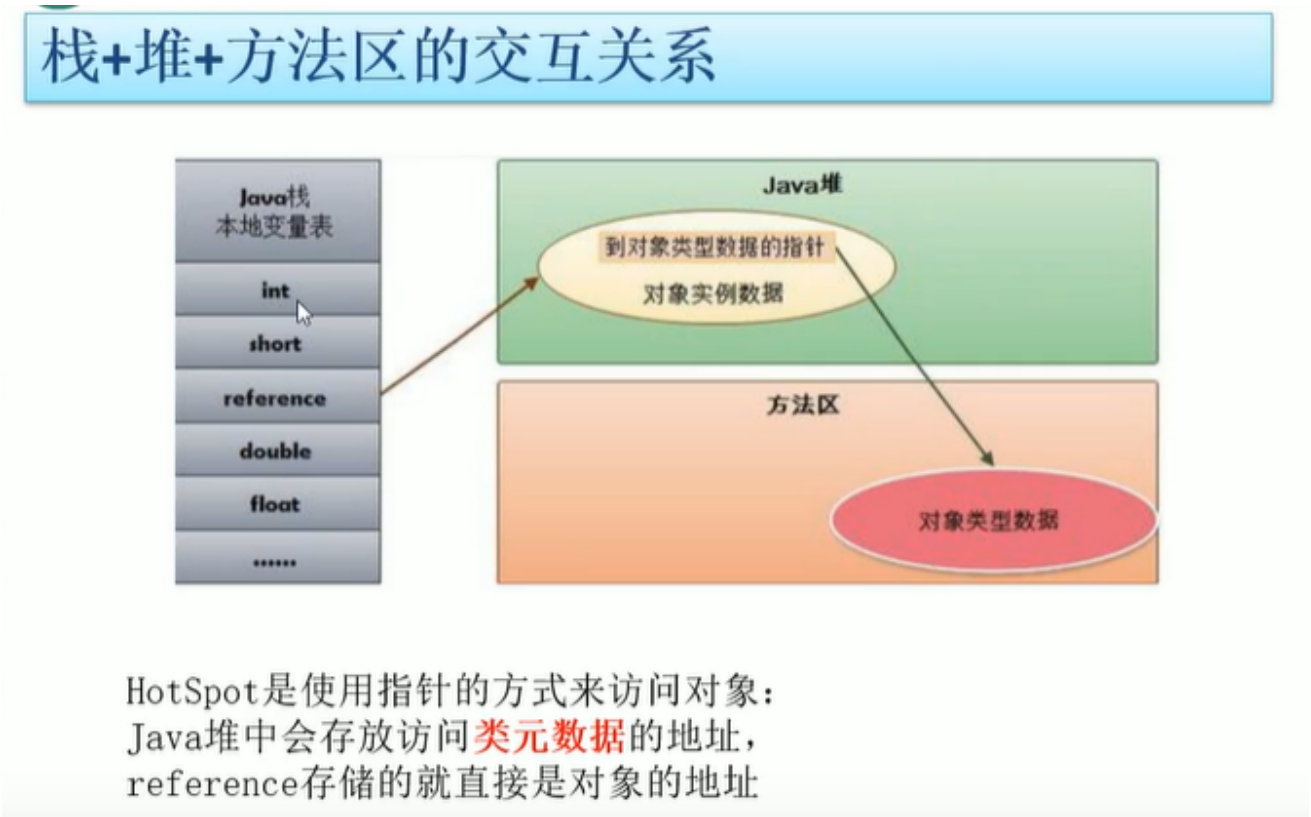
-XX:+PrintGCDetails：输出详细gc处理日志



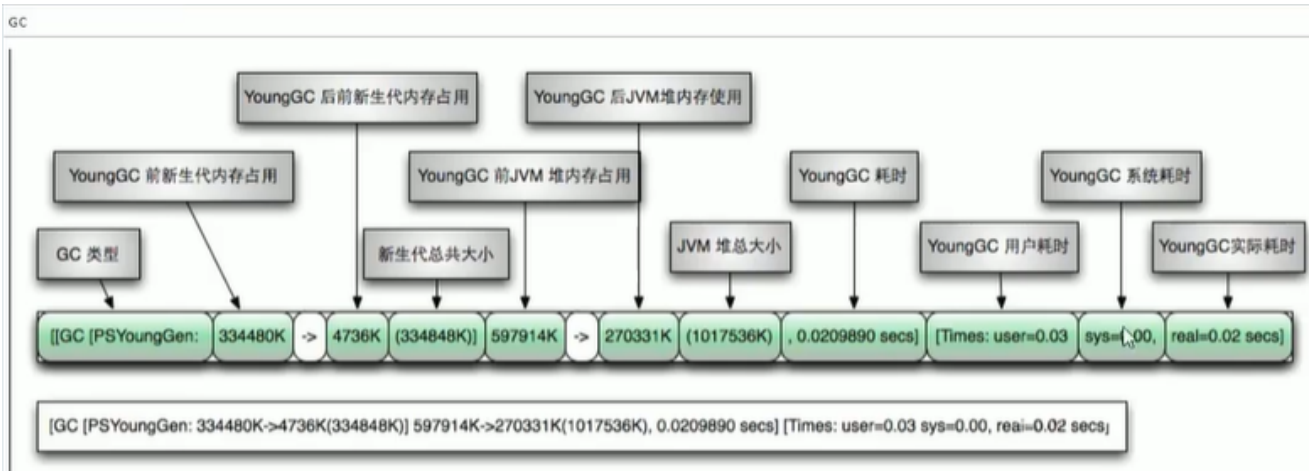
OOM时导出堆到文件: -Xms1m -Xmx8m -XX:+HeapDumpOnOutOfMemoryError

4、JVM原理（数据交互）

4.1、堆+栈+方法区的交互关系



5、GC日志信息



6、JVM涉及GC算法

6.1、引用计数法

6.2、复制算法

当对象在 Eden (包括一个 Survivor 区域，这里假设是 from 区域) 出生后，在经过一次 Minor GC 后，如果对象还存活，并且能够被另外一块 Survivor 区域所容纳(上面已经假设为 from 区域，这里应为 to 区域，即 to 区域有足够的内存空间来存储 Eden 和 from 区域中存活的对象)，则使用复制算法将这些仍然还存活的对象复制到另外一块 Survivor 区域 (即 to 区域) 中，然后清理所使用过的 Eden 以及 Survivor 区域 (即 from 区域)，并且将这些对象的年龄设置为1，以后对象在 Survivor 区每熬过一次 Minor GC，就将对象的年龄 + 1，当对象的年龄达到某个值时 (默认是 15 岁，通过-XX:MaxTenuringThreshold 来设定参数)，这些对象就会成为老年代。

注：劣势，要求存活率低，浪费一半内存

6.3、标记清除算法

标记一遍，清理一遍，期间应用暂停，速度慢，易产生内存碎片 **注：针对年老代**

6.4、标记整理算法

标记一遍，移动一遍

注：针对年老代