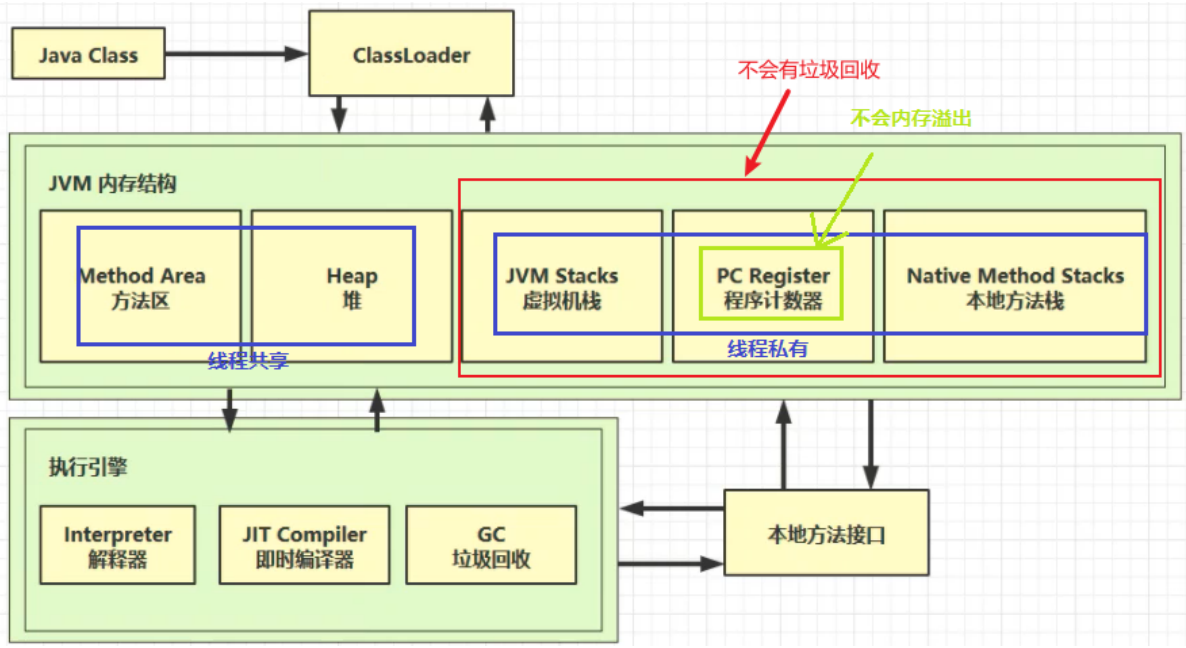
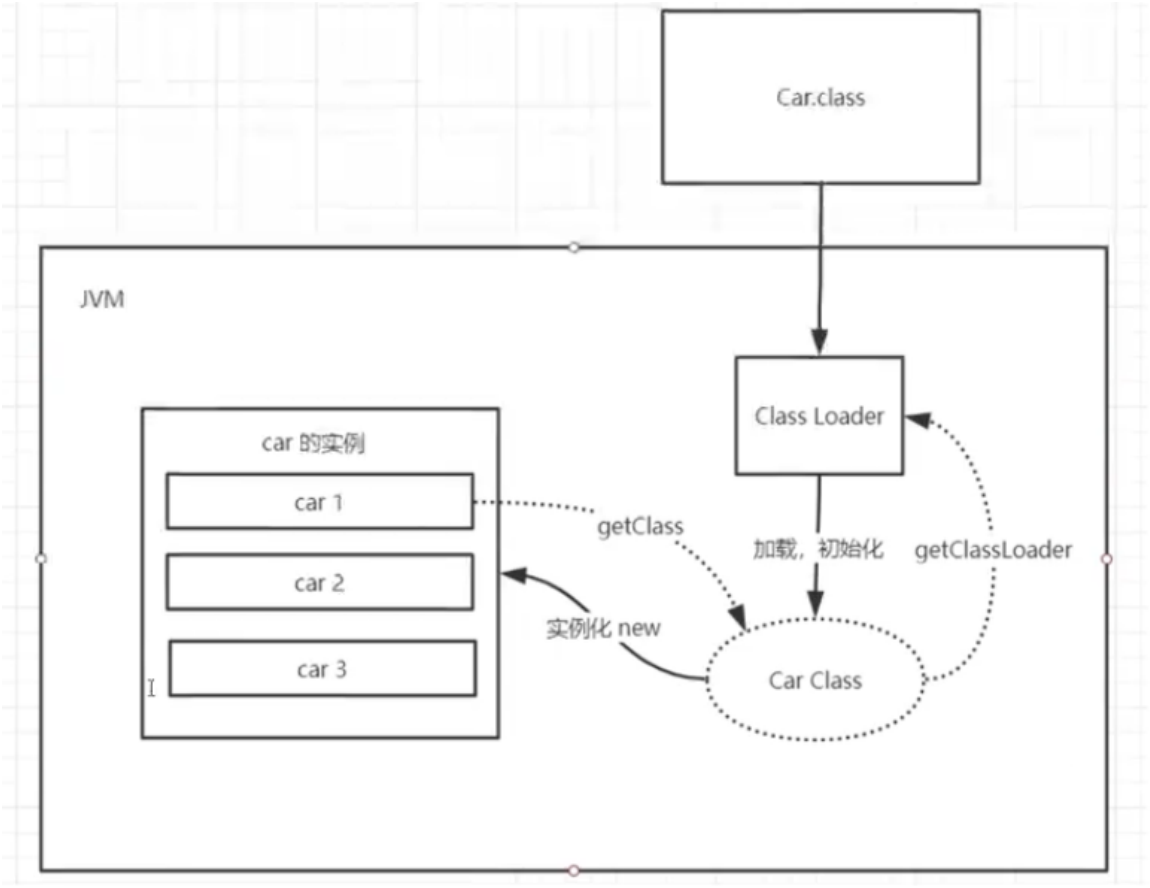
JVM

JVM总结构图



> 虚拟机栈、程序计数器、本地方法栈都是线程私有的，方法区、堆是线程共享的

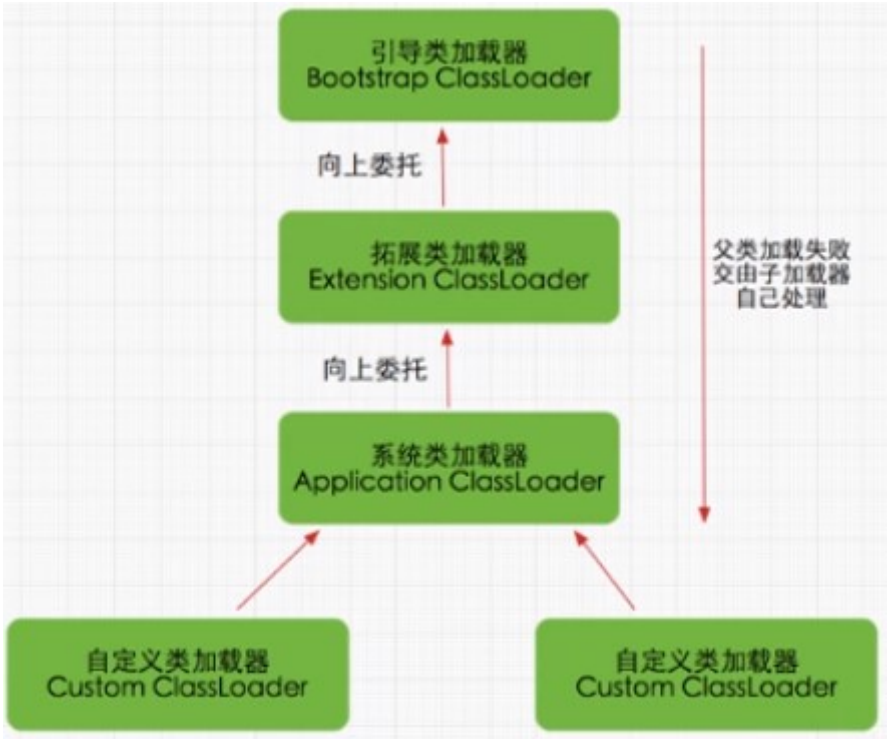
类加载器



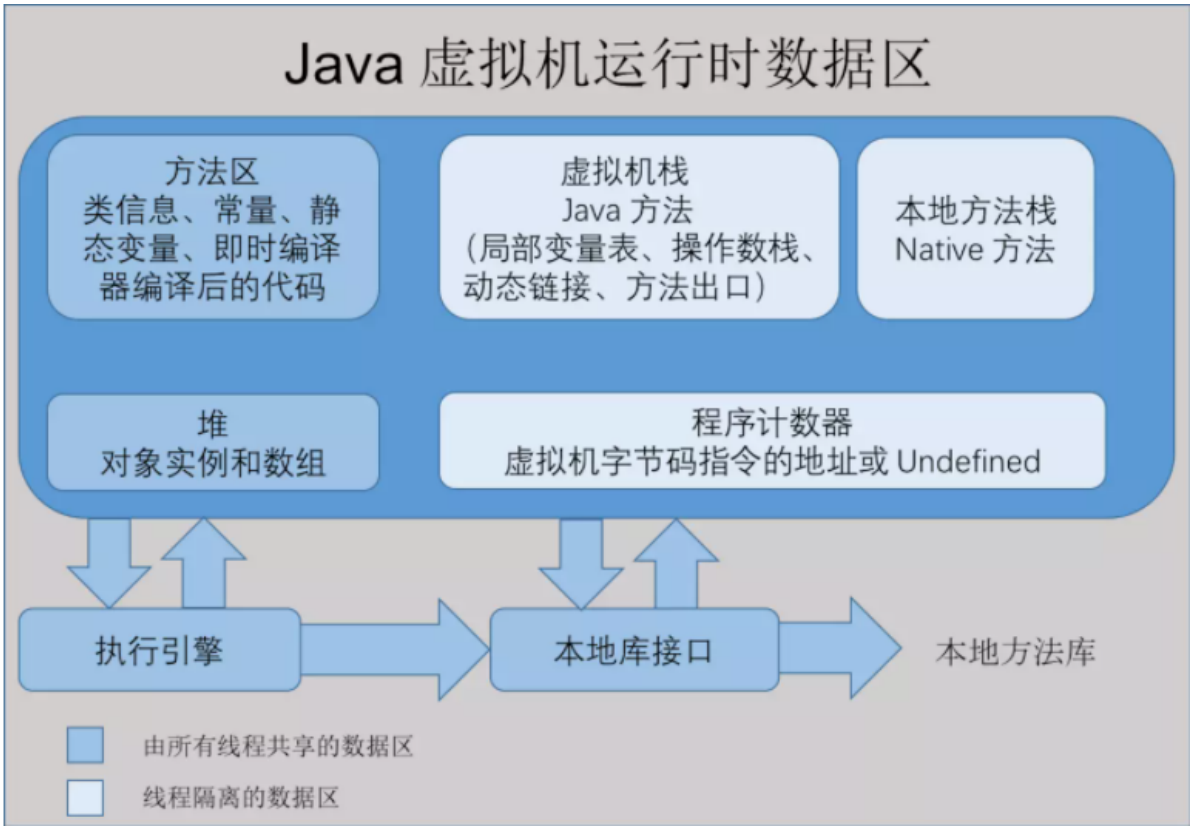

```
public class Car {
    public static void main(String[] args) {
```

```
        //类时模板，对象时具体的。具体的对象可以拥有相同的类模板
        Car car1 = new Car();
        Car car2 = new Car();
        Car car3 = new Car();
        System.out.println(car1.hashCode());
        System.out.println(car2.hashCode());
        System.out.println(car3.hashCode());
        //获取类class模板
        Class<? extends Car> aClass1 = car1.getClass();
        Class<? extends Car> aClass2 = car2.getClass();
        Class<? extends Car> aClass3 = car3.getClass();
        System.out.println(aClass1.hashCode());
        System.out.println(aClass2.hashCode());
        System.out.println(aClass3.hashCode());
        /*结果：三个实例对象的地址值不一样，但模板地址值一样
        621009875
        1265094477
        2125039532
        856419764
        856419764
        856419764*/
        ClassLoader classLoader = aClass1.getClassLoader();
        System.out.println(classLoader);// AppClassLoader
        System.out.println(classLoader.getParent()); // ExtClassLoader
        System.out.println(classLoader.getParent().getParent()); //null,Java程序抓不到。这是本地方法接口，由C或C++写的
    }
}
```

双亲委派机制（安全）



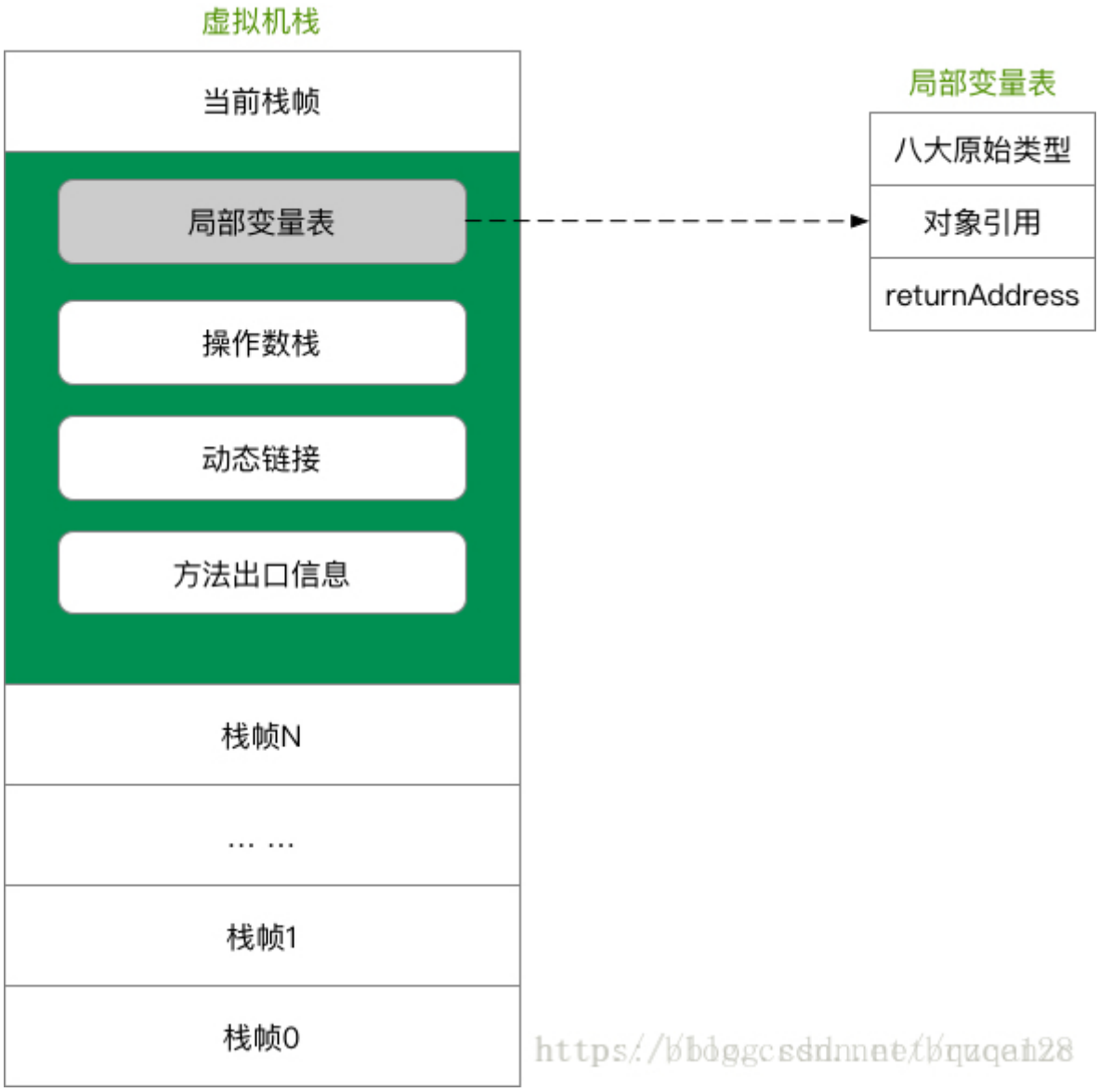
APP-->Ext-->BootStrap



程序计数器 PC Register

- 作用：记住下一条jvm指令的执行地址
- 特点：
    1. 是线程私有的
    2. 不会存在内存溢出

虚拟机栈 JVM Stacks



- 定义：线程运行时所需要的内存成为虚拟机栈
- 每个栈由多个栈帧组成，对应着每次方法调用时所占的内存
- 每个线程只能有一个活动栈帧，对应着正在执行的那个方法

栈:栈内存，主管程序的运行，生命周期和线程同步;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!

栈里存放的是:8大基本类型 + 对象引用 + 实例的方法

栈运行原理:栈帧

栈溢出:

1. 栈帧过多（递归）
2. 栈帧过大，一个栈帧比栈还大

栈内存溢出实例:

```
public class Car {
    public static void main(String[] args) {
        new Car().a();
    }
    public void a() {
        b();
    }
    public void b() {
        a();
    }
}
```

对象的实例化

声明：

```
class Student {
    String name = "Alice";//显示初始化
    int age = 18; //显示初始化
    public Student() {
        name = "Bob"; //构造方法初始化
        age = 24 ;  //构造方法初始化
    }
}
```

调用:

```
Student s = new Student();
```

编译期间，会生成Student.class字节码文件。

初始化时：

1、类加载器ClassLoader，加载Student.class字节码到内存；

2、在栈里面为变量s申请一个空间，用来声明s；

3、new的时候，在堆内开辟空间。然后，开始进行默认初始化，String类型默认给null，int类型默认给0等。默认初始化后，开始进行显示初始化，比如成员变量里name默认值为Alice，所以这时会初始化name为Alice。

3、执行Student()构造方法，构造方法进栈，进行构造方法初始化

4、执行构造方法初始化，构造方法执行完毕后出栈，把堆内的对象物理地址，复制给栈内的s，也就是s存放的是对象的引用（物理地址）。

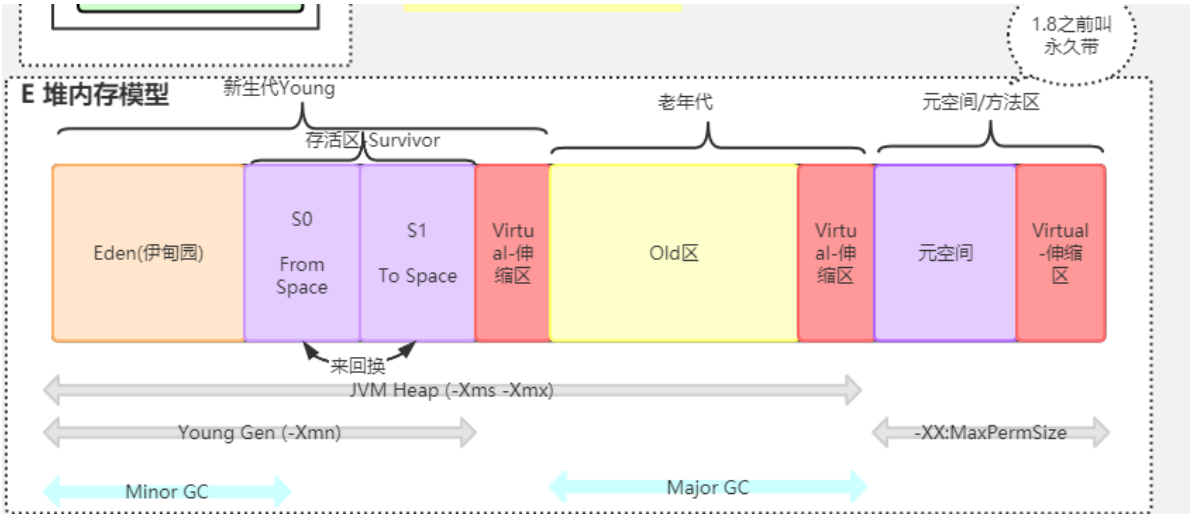
5、执行Student里面的方法时，方法进栈，方法里隐式的this指向堆内存空间s

本地方法栈 Native Method Stacks

- 作用：为本地方法运行提供内存空间

```
Object.java
202      * exception at run time.
203      *
204      * @return     a clone of this instance.
205      * @throws  CloneNotSupportedException  if the object's class does not
206      *               support the {@code Cloneable} interface. Subclasses
207      *               that override the {@code clone} method can also
208      *               throw this exception to indicate that an instance cannot
209      *               be cloned.
210      * @see java.lang.Cloneable
211      */
212     protected native Object clone() throws CloneNotSupportedException;
213
214     /**
215      * Returns a string representation of the object. In general, the
216      * {@code toString} method returns a string that
217      * "textually represents" this object. The result should
218      * be a concise but informative representation that is easy for a
```

堆 Heap



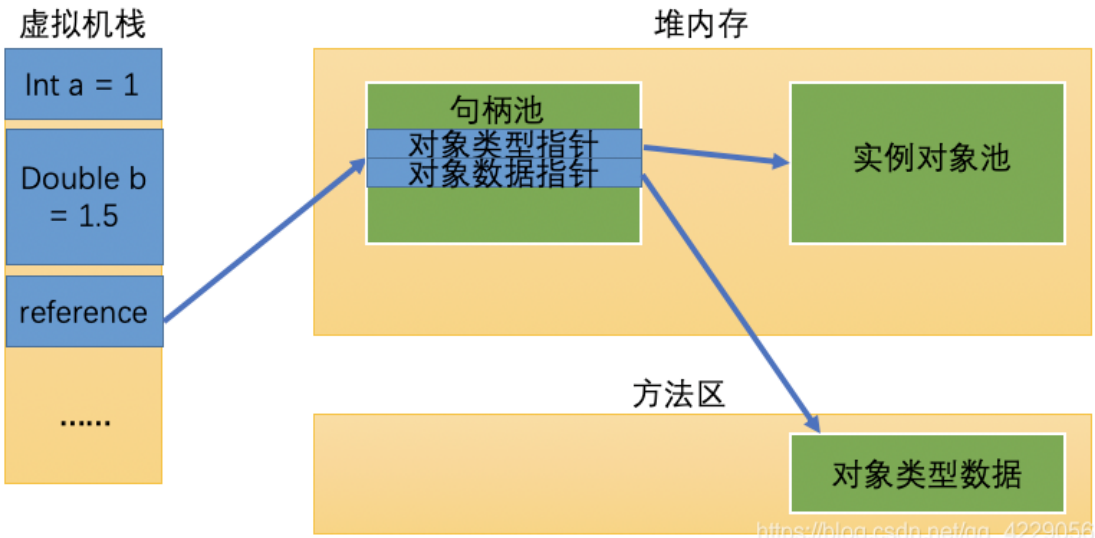
- 通过new关键字，创建对象都会使用堆内存
- 特点

    1. 它是线程共享的，堆中的对象都要考虑线程安全问题
    2. 有垃圾回收机制

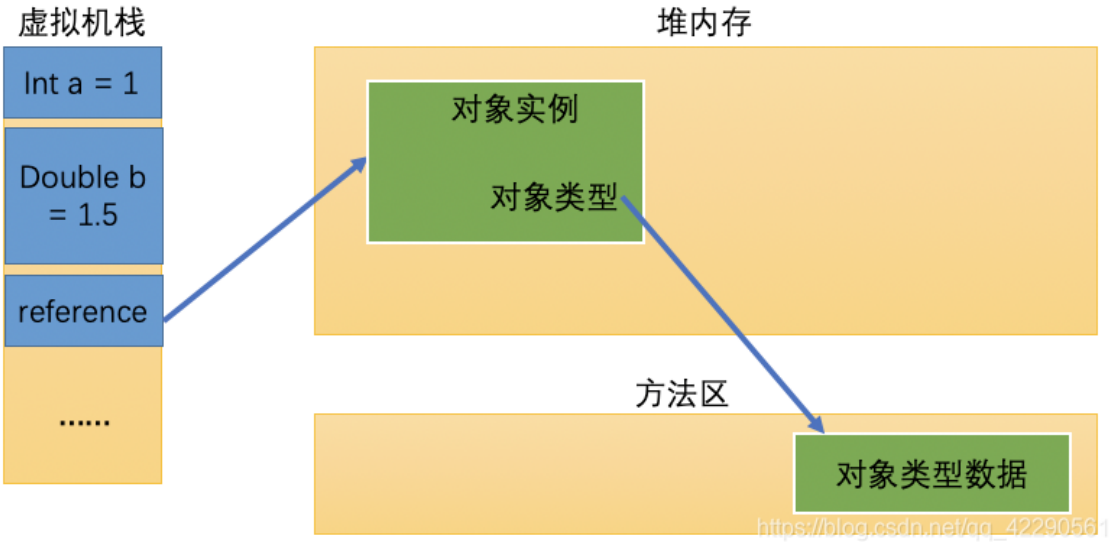
不断的new对象就会造成堆内存溢出

对象的引用:

- 句柄



- 直接引用



方法区 Method Area

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(static)、常量(finale)、类信息(构造方法、接口定义)(Class模板)、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

- 方法区里存放着类的版本、字段、方法、接口和常量池（存储字面量和符号引用）

  符号引用包括：1、类的权限定名；2、字段名和属性；3、方法名和属性。

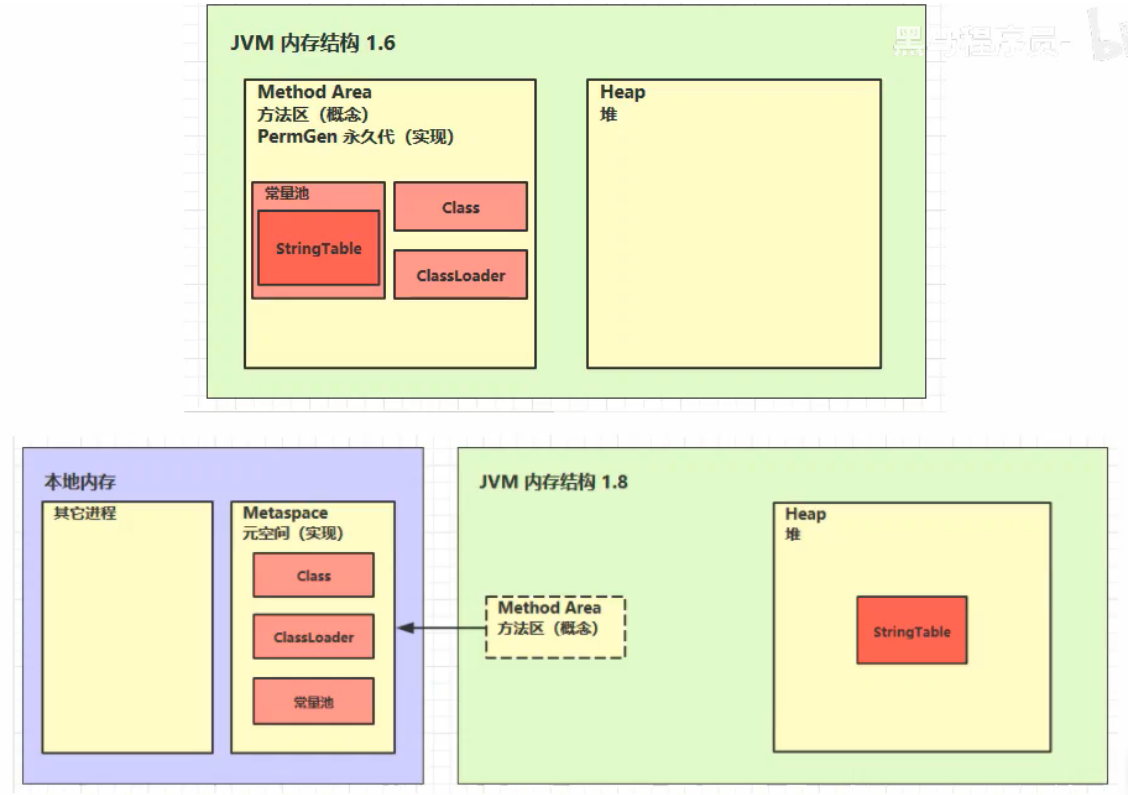
Method Area (方法区)

虚拟机已加载的类信息

Class1	Class2	Class3.....n
1、类型信息	1、类型信息	
2、类型的常量池	2、类型的常量池	
3、字段信息	3、字段信息	
4、方法信息	4、方法信息	
5、类变量	5、类变量	
6、指向类加载器的引用	6、指向类加载器的引用	
7、指向Class实例的引用	7、指向Class实例的引用	
8、方法表	8、方法表	

运行时常量池

- 方法区在jdk1.6和1.7以后的区别



> Method Area都只是一种概念，Method Area在1.6以前是在jvm中，而1.7以后是在本地内存（操作系统内存）中。

> 移除永久代的工作从JDK1.7就开始了。JDK1.7中，存储在永久代的部分数据就已经转移到了Java Heap或者是 Native Heap。但永久代仍存在于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；类的静态变量(class statics)转移到了java heap。元空间本质和永久代类似，都是对JVM规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此默认情况下元空间的大小仅受本地内存限制。

StringTable特性:

String s1 = "a";

String s2 = "b";

1. 常量池中的字符串仅是符号，第一次用到时才变为对象
2. 利用串池的机制，来避免重复创建字符串对象
3. 字符串变量拼接的原理是StringBuilder (1.8) s1 + s2
4. 字符串常量拼接的原理是编译期优化 "a" + "b"
5. 可以使用intern方法，主动将串池中还没有的字符串对象放入串池

```
public class Main {
    public static void main(String[] args) {
        String s1 = new String("a") + new String("b"); //相当于new String("ab"),new出来的对象都放在堆中
        //此时串池中是["a", "b"]两个字符串
        //此时堆中是new String("a"),new String("b"),new String("ab")
        String s2 = s1.intern();//将这个字符串对象放入串池。如果串池有该字符串对象，则不放入且返回串池中的对象，如果没有，则放入并返回串池中的对象。并返回串池中的这个对象
        //此时串池中是["a", "b", "ab"]
        //刚才new出来的s1也被放入了串池
        System.out.println(s2 == "ab");//true
        System.out.println(s1 == "ab");//true
        System.out.println(s1 == s2);//true
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        String x = "ab";
        String s1 = new String("a") + new String("b");
        String s2 = s1.intern(); //不会将s1放入串池，因为串池中有“ab”字符串，直接返回串池中的“ab”就行
        System.out.println(x == s2); //true
        System.out.println(x == s1); //false
    }
}
```

StringBuilder是线程不安全的，但是它只是个**局部变量**，局部变量存储在**虚拟机栈**，**虚拟机栈**是线程隔离的，所以不会有线程安全问题