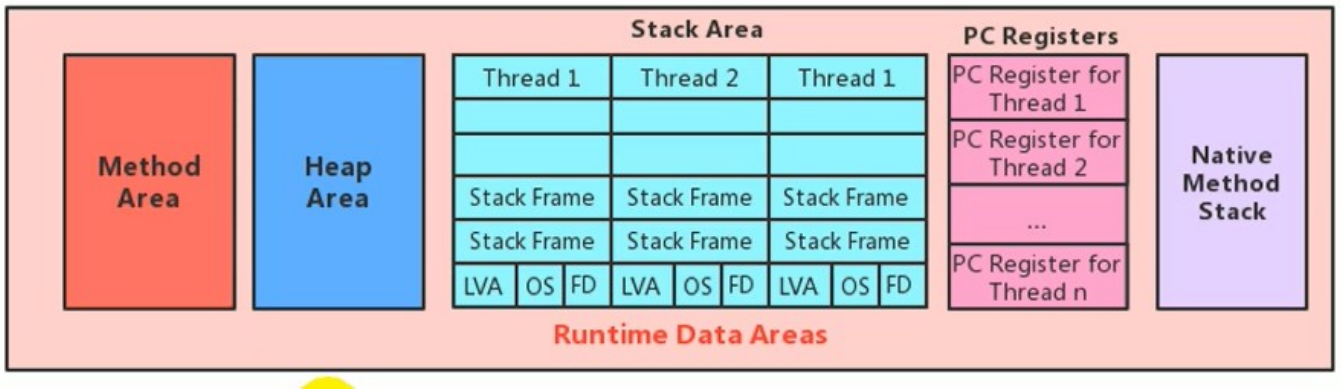
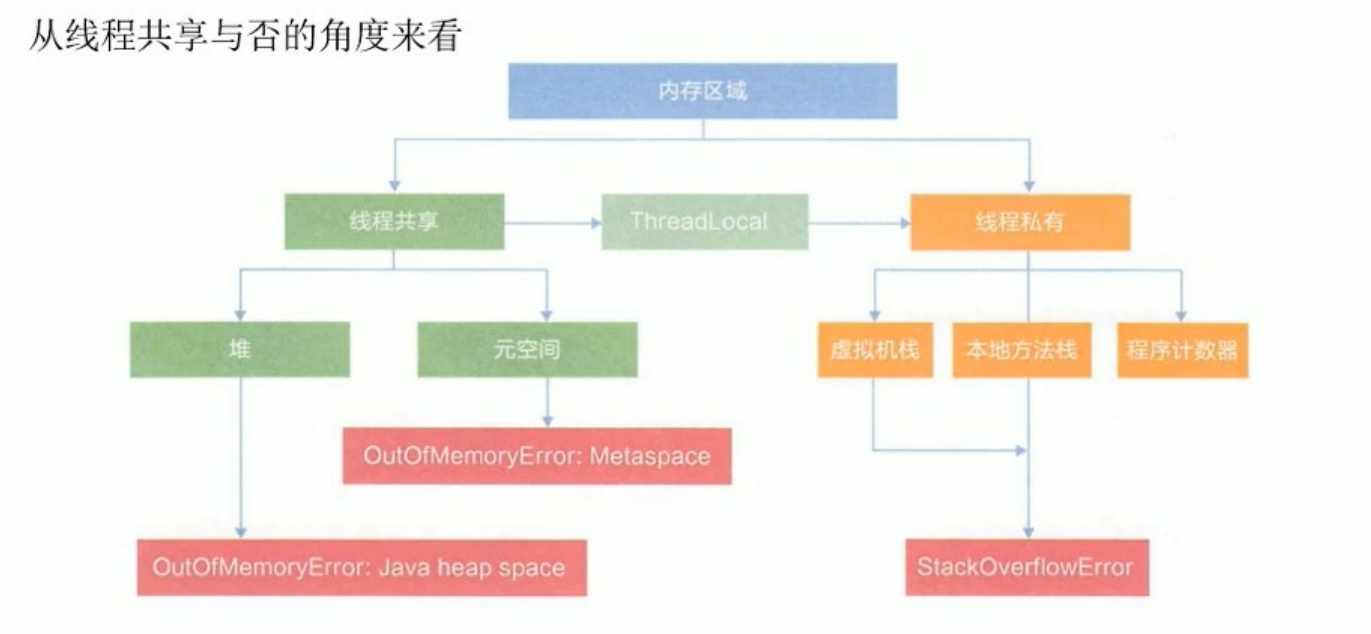
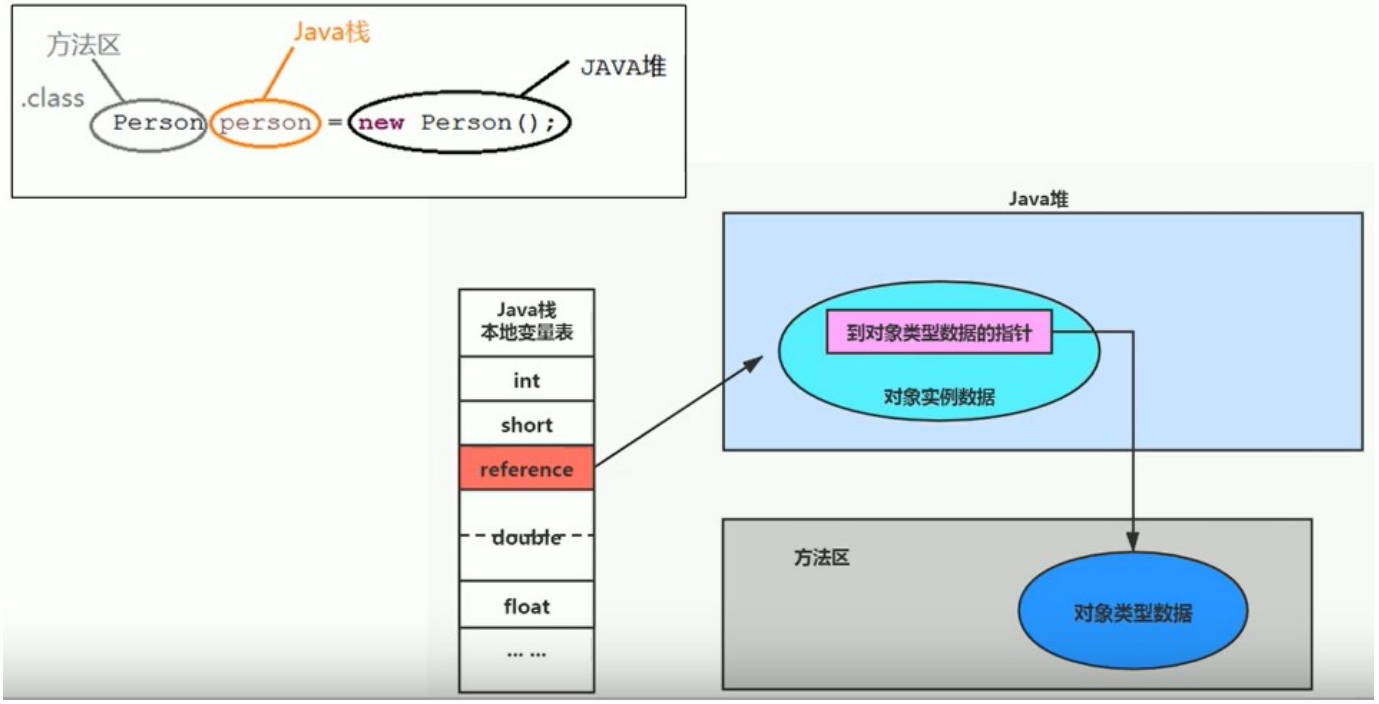
堆、栈和方法区的交互关系



内存区域的划分:



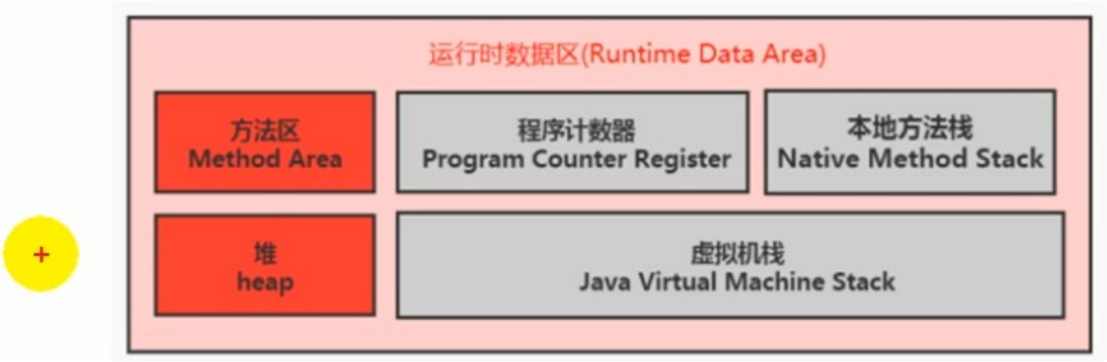
方法区存放数据类型，java栈存放数据引用 reference, java堆 存放数据



方法区的理解

《Java虚拟机规范》中明确说明："尽管所有的方法区在逻辑上是属于堆的一部分，但一些简单的实现可能不会选择去进行垃圾收集或者进行压缩。" 但对于HotSpotJVM而言，方法区还有一个别名叫做Non-Heap(非堆)，目的就是要和堆分开。

所以，方法区看作是一块独立于Java堆的内存空间。

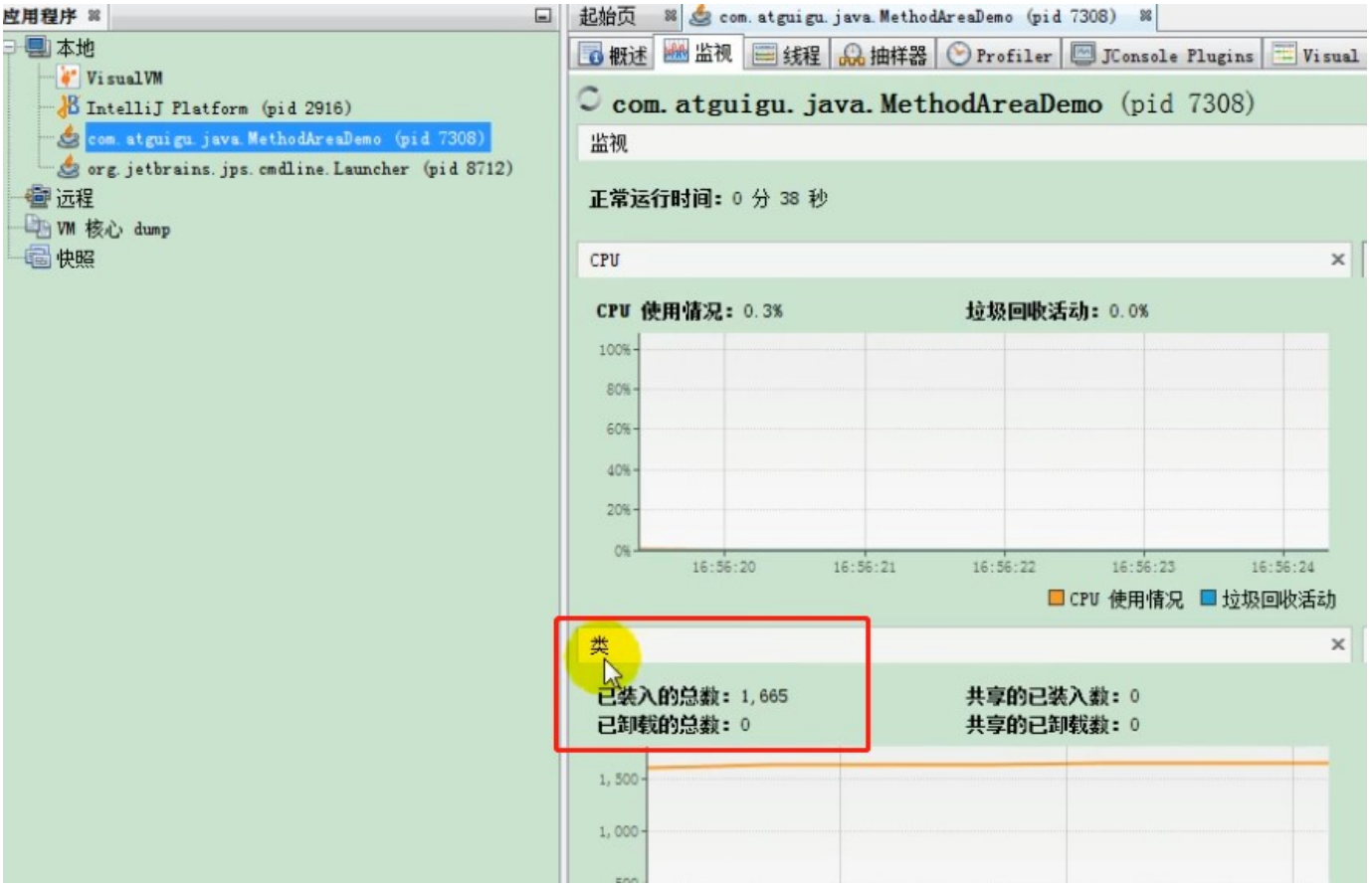


- 方法区（Method Area) 与Java堆一样，是各个线程共享的内存区域。
- 方法区在JVM启动的时候被创建，并且它的实际的物理内存空间中和Java堆区一样都可以是不连续的。
- 方法区的大小，跟堆空间一样，可以选择固定大小或者可扩展。
- 方法区的大小决定了系统可以保存多少个类，如果系统定义了太多的类，导致方法区溢出，虚拟机同样会抛出内存溢出错误：java.lang.OutOfMemoryError: PermGen

space 或者 java.lang.OutOfMemoryError : Metaspace

举例: 加载大量第三方jar包; Tomcat加载的项目过多; 大量动态生成反射类;

- 关比jvm就会释放这个区域的内存。



说明: 一个很简单的sout, 就会加载这么多类

设置方法区大小与OOM

方法区的大小不必是固定的, jvm可以根据应用的需要的动态调整;

根据官方的文档显示, 方法区可以设置成固定大小, 也可以设置成动态调整的模式;

jdk7及之前

通过 -XX:PermSize 来设置永久代初始分配空间; 默认值是20.75M

-XX: MaxPermSize 来设定永久代最大可分配空间; 32位机器默认是64M,64位机器模式是82M

当Jvm加载的类信息容量超过了这个值, 会报异常OutOfMemoryError: PermGen space

```java
/**
 *  测试设置方法区大小参数的默认值
 *
 *  jdk7及以前：
 *  -XX:PermSize=100m -XX:MaxPermSize=100m
 *
 *  jdk8及以后：
 *  -XX:MetaspaceSize=100m  -XX:MaxMetaspaceSize=100m
 * @author shkstart  shkstart@126.com
 * @create 2020  12:16
 */
public class MethodAreaDemo {
    public static void main(String[] args) {
        System.out.println("start...");
//        try {
//            Thread.sleep(1000000);
//        } catch (InterruptedException e) {
//            e.printStackTrace();
//        }

        System.out.println("end...");
    }
}
```

可以通过 jinfo -flag 查看对应参数

jdk8及以后

- 元数据区大小可以使用参数 -XX:MetaspaceSize 和 -XX:MaxMetaspaceSize指定，替代上述原有的两个参数。
- 默认值依赖于平台。windows下，-XX:MetaspaceSize是21M，-XX:MaxMetaspaceSize的值是-1，即没有限制。
- 与永久代不同，如果不指定大小，默认情况下，虚拟机会耗尽所有的可用系统内存。如果元数据区发生溢出，虚拟机一样会抛出异常OutOfMemoryError: Metaspace
- -XX:MetaspaceSize:设置初始的元空间大小。对于一个64位的服务器端JVM来说，其

默认的-XX:MetaspaceSize值为21MB。这就是初始的高水位线，一旦触及这个水位线，Full GC将会被触发并卸载没用的类（即这些类对应的类加载器不再存活）， 然后这个高水位线将会重置。新的高水位线的值取决于GC后释放了多少元空间。如果 释放的空间不足，那么在不超过MaxMetaspaceSize时，适当提高该值。如果释放 空间过多，则适当降低该值。

- 如果初始化的高水位线设置过低，上述高水位线调整情况会发生很多次; 通过垃圾回收器的日志可以观察到Full GC多次调用。为了避免频繁地GC，建议将 -XX: MetaSpaceSize设置为一个相对较高的值;

方法区内存溢出代码示例

```
import com.sun.xml.internal.ws.org.objectweb.asm.ClassWriter;
import jdk.internal.org.objectweb.asm.Opcodes;

/**
 * jdk6/7中:
 * -XX:PermSize=10m -XX:MaxPermSize=10m
 *
 * jdk8中:
 * -XX:MetaspaceSize=10m -XX:MaxMetaspaceSize=10m
 *
 * @author shkstart  shkstart@126.com
 * @create 2020  22:24
 */
public class OOMTest extends ClassLoader {
    public static void main(String[] args) {
        int j = 0;
        try {
            OOMTest test = new OOMTest();
            for (int i = 0; i < 10000; i++) {
                //创建ClassWriter对象，用于生成类的二进制字节码
                ClassWriter classWriter = new ClassWriter(0);
                //指明版本号，修饰符，类名，包名，父类，接口
                classWriter.visit(Opcodes.V1_6, Opcodes.ACC_PUBLIC, "Class" + i
                //返回byte[]
```

```
                byte[] code = classWriter.toByteArray();
                //类的加载
                test.defineClass("Class" + i, code, 0, code.length);//Class对象
                j++;
            }
        } finally {
            System.out.println(j);
        }
    }
}
```

带上参数, 也就是不动态, 就不报错 OOM: MetaSpace

如何解决OOM

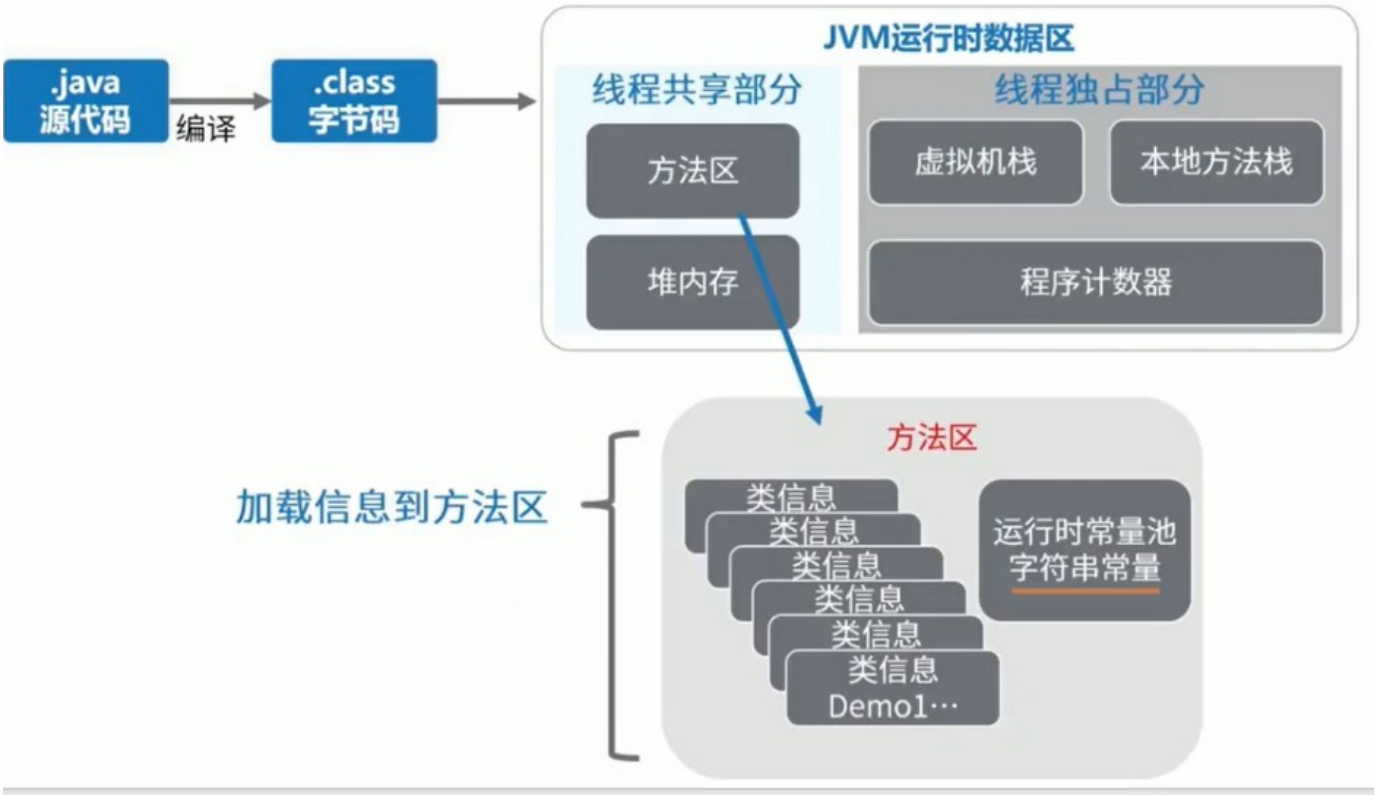
1、 要解决OOM异常或heap space的异常，一般的手段是首先通过内存映像分析工具 (如Eclipse Memory Analyzer)对dump出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的，也就是要先分清楚到底是出现了内存泄漏（Memory Leak)还是内存溢出（Memory Overflow)

2、 如果是内存泄漏，可进一步通过工具查看泄漏对象到GC Roots的引用链。于是就能找到泄漏对象是通过怎样的路径与GC Roots相关联并导致垃圾收集器无法自动回收它们的。掌握了泄漏对象的类型信息，以及GC Roots引用链的信息，就可以比较准确地定位出泄漏代码的位置。

3、 如果不存在内存泄漏，换句话说就是内存中的对象确实都还必须存活着，那就应当 检查虚拟机的堆参数（-Xmx与-Xms)，与机器物理内存对比看是否还可以调大，从代码上检查是否存在某些对象生命周期过长、持有状态时间过长的情况，尝试减少程序运行期的内存消耗。

内存泄漏, 就是栈中依旧有堆空间内对象的关联,但是对象又不再使用, 无法回收, 称之为内存泄漏;

方法区的内部结构



<<深入理解Java虚拟机>> 书中对方法区(Method Area)存储内容描述如下:

它用于存储已被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等;

类型信息

对每个加载的类型（类Class、接口interface、枚举enum、注解annotation），JVM必须在方法区中存储以下类型信息：

① 这个类型的完整有效名称（全名=包名.类名）

② 这个类型直接父类的完整有效名（对于interface或足java. lang.Object，都没有父类）

③ 这个类型的修饰符（public, abstract, final的某个子集）

④ 这个类型直接接口的一个有序列表

域(Field)信息

JVM必须在方法区中保存类型的所有域的相关信息以及域的生命顺序;

域的相关信息包括: 域名称, 域类型, 域修饰符(public, private, protected, static, final, volatile, transient的某个子集)

方法(Method)信息

JVM必须保存所有方法的以下信息，同域信息一样包括声明顺序：

- 方法名称
- 方法的返回类型（或void)
- 方法参数的数量和类型（按顺序）
- 方法的修饰符（public, private, protected, static, final, synchronized, native, abstract的一个子集）
- 方法的字节码（bytecodes)、操作数栈、局部变M表及大小（abstract和 native方法除外）
- 异常表 （abstract和native方法除外）

每个异常处理的开始位置、结束位置、代码处理在程序计数器中的偏移地址、 被捕获的异常类的常量池索引

代码示例

```
import java.io.Serializable;

/**
 * 测试方法区的内部构成
 * @author shkstart  shkstart@126.com
 * @create 2020  23:39
 */
public class MethodInnerStrucTest extends Object implements Comparable<String>,
    //属性
    public int num = 10;
    private static String str = "测试方法的内部结构";
    //构造器
```

```
    //方法
    public void test1(){
        int count = 20;
        System.out.println("count = " + count);
    }
    public static int test2(int cal){
        int result = 0;
        try {
            int value = 30;
            result = value / cal;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return result;
    }

    @Override
    public int compareTo(String o) {
        return 0;
    }
}
https://www.bilibili.com/video/BV1PJ411n7xZ?p=92
```

non-final的类变量

- 静态变量和类关联在一起, 随着类的加载而加载, 它们成为类数据在逻辑上的一部分;
- 类变量被类的所有实例共享, 即使没有类实例时你也可以访问它;

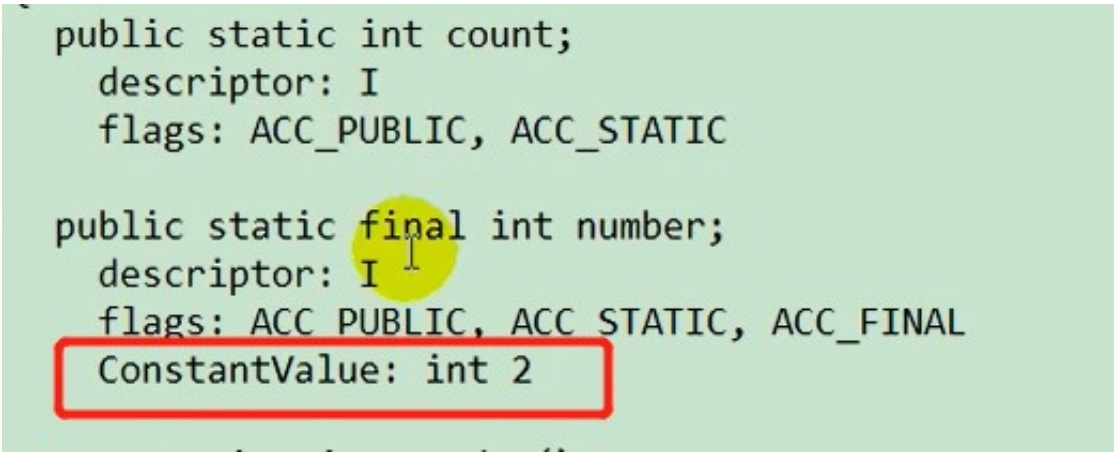
```
/**
 * non-final的类变量
 * @author shkstart  shkstart@126.com
 * @create 2020  20:37
 */
public class MethodAreaTest {
    public static void main(String[] args) {
        Order order = null;
        order.hello();
        System.out.println(order.count);
```

```
    }
}

class Order {
    public static int count = 1;
    public static final int number = 2;


    public static void hello() {
        System.out.println("hello!");
    }
}

```

全局常量

static final

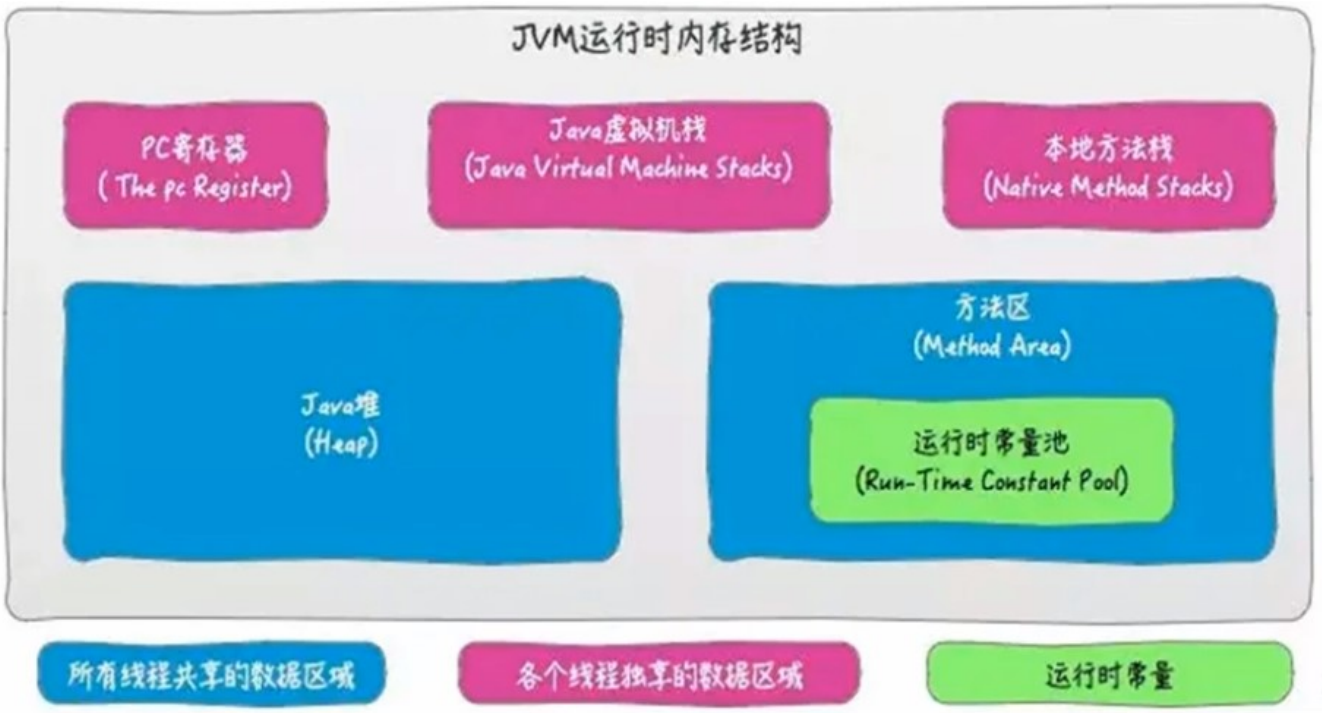
- 被声明为final的类变量的处理方法则不同, 每个全局常量在编译的时候就会被分配了;



带有 static final的常量, 在编译的时候, 就已经赋值了; 仅仅是final 只是不能变而已

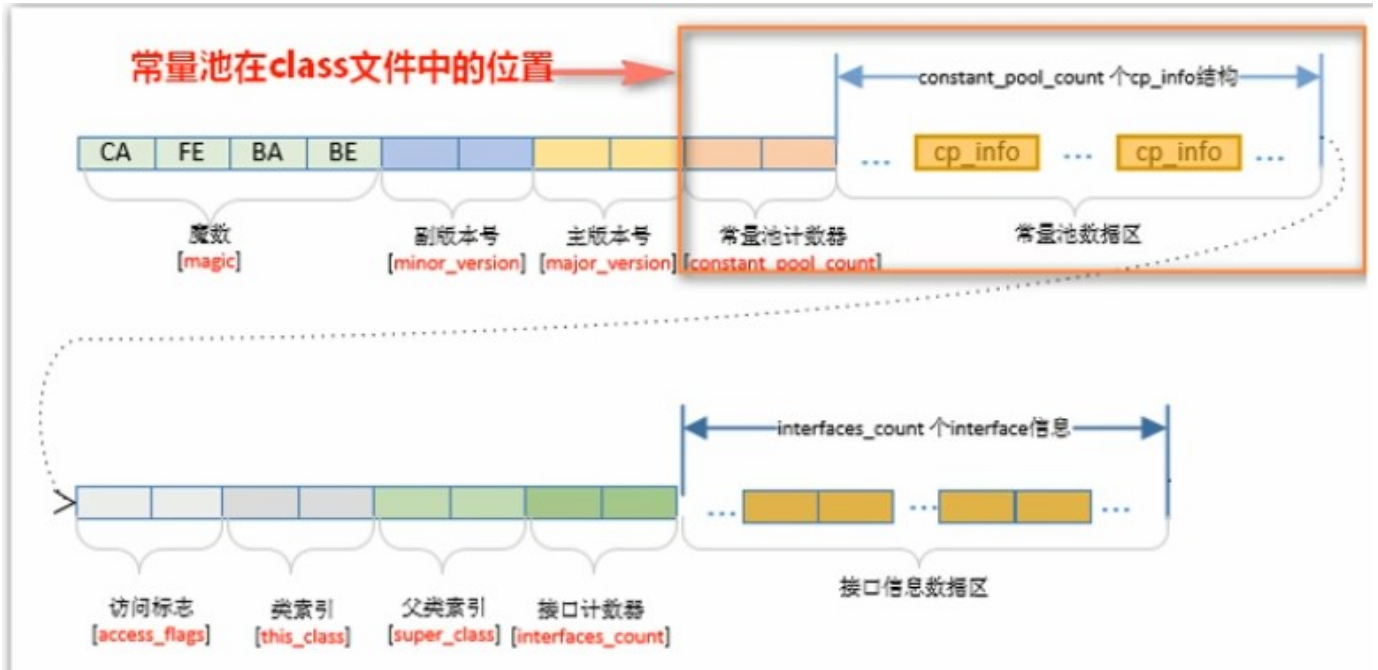
static 是在加载, prepare的时候 创建 赋值默认, 在 初始化阶段的 clinit(类构造器方法)方法进行赋值

运行时常量池和常量池



- 方法区, 内部包含了运行时常量池;
- 字节码文件, 内部包含了常量池;
- 要弄清楚方法区, 需要理解清楚ClassFile, 因为加载类的信息都在方法区;
- 要弄清楚方法区的运行时常量池, 需要理解清楚ClassFile中的常量池

常量池



- 一个有效的字节码文件中除了包含类的版本信息、字段、方法以及接口等描述信息外, 还包含一项信息那就是常量池表 (Constant Pool Table), 包括各种字面量和对类型、域

和方法的符号引用;

常量池的作用

一个java源文件中的类、接口, 编译后产生一个字节码文件;

而Java中的字节码需要数据支持,通常这种数据会很大以至于不能直接存到字节码里,

换另一种方式, 可以存到常量池, 这个字节码包含了指向常量池的引用;

在动态链接的时候会用到运行时常量池, 之前有介绍;

比如：如下的代码：

```
public class SimpleClass {
    public void sayHello() {
        System.out.println("hello");
    }
}
```

虽然只有194字节，但是里面却使用了String、System、PrintStream及Object等结构。这里代码量其实已经很小了。如果代码多，引用到的结构会更多！这里就需要常量池了！

几种在常量池内存储的数据类型包括:

- 数量值
- 字符串值
- 类引用
- 字段引用
- 方法引用

例如下面这段代码：

```
public class MethodAreaTest2 {
    public static void main(String[] args) {
        Object obj = new Object();
    }
}
```

```
Object foo = new Object();
```

将会被编译成如下字节码：

```
0:      new #2                  // Class java/lang/Object
1:      dup
2:      invokespecial #3        // Method java/ lang/Object "<init>"( ) V
```

小结

常量池, 可以看做是一张表, 虚拟机指令根据这张常量表找到要执行的类名、方法名、参数类型、字面量等类型;

运行时常量池

- 运行时常量池（Runtime Constant Pool)是方法区的一部分。
- 常量池表（Constant Pool Table) 是Class文件的一部分，**用于存放编译期生成的各种字面量与符号引用，这部分内容将在类加载后存放到方法区的运行时常最池中。**
- 运行时常量池，在加载类和接口到虚拟机后，就会创建对应的运行时常量池。
- JVM为每个已加载的类型（类或接口）都维护一个常量池。池中的数据项像数组项一样，是通过索引访问的。
- 运行时常量池中包含多种不同的常量,包括编译期就已经明确的数值字面量，也包括到运行期解析后才能够获得的方法或者字段引用。此时不再是常量池中的符号地址了，这里换为真实地址。

  运行时常量池，相对于Class文件常量池的另一重要特征是：**具备动态性。**

  例如: String.intern() 将string放入常量池
- 运行时常量池类似于传统编程语言中的符号表（symbol table)，但是它所包含的数据却比符号表要更加丰富一些。
- 当创建类或接口的运行时常量池时，如果构造运行时常池所需的内存空间超过了方法区所能提供的M大值，则JVM会抛OutOfMemoryError异常。

方法区使用举例

示例代码:

```
/**
 *
 * @create 2020  14:28
 */
public class MethodAreaDemo {
    public static void main(String[] args) {
        int x = 500;
        int y = 100;
        int a = x / y;
        int b = 50;
        System.out.println(a + b);
    }
}

```

该方法的字节码文件:

```
Classfile /D:/workspace_idea5/JVMDemo/out/production/chapter09/com/atguigu/java
  Last modified 2020-4-23; size 640 bytes
  MD5 checksum a2e8a0e034e2dd986b45d36a3401a63b
  Compiled from "MethodAreaDemo.java"
public class com.atguigu.java1.MethodAreaDemo
  minor version: 0
  major version: 51
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #5.#24         // java/lang/Object."<init>":()V
   #2 = Fieldref           #25.#26        // java/lang/System.out:Ljava/io/Prin
   #3 = Methodref          #27.#28        // java/io/PrintStream.println:(I)V
   #4 = Class              #29            // com/atguigu/java1/MethodAreaDemo
   #5 = Class              #30            // java/lang/Object
   #6 = Utf8               <init>
   #7 = Utf8               ()V
```

```
   #8 = Utf8               Code
   #9 = Utf8               LineNumberTable
  #10 = Utf8               LocalVariableTable
  #11 = Utf8               this
  #12 = Utf8               Lcom/atguigu/java1/MethodAreaDemo;
  #13 = Utf8               main
  #14 = Utf8               ([Ljava/lang/String;)V
  #15 = Utf8               args
  #16 = Utf8               [Ljava/lang/String;
  #17 = Utf8               x
  #18 = Utf8               I
  #19 = Utf8               y
  #20 = Utf8               a
  #21 = Utf8               b
  #22 = Utf8               SourceFile
  #23 = Utf8               MethodAreaDemo.java
  #24 = NameAndType        #6:#7          // "<init>":()V
  #25 = Class              #31            // java/lang/System
  #26 = NameAndType        #32:#33        // out:Ljava/io/PrintStream;
  #27 = Class              #34            // java/io/PrintStream
  #28 = NameAndType        #35:#36        // println:(I)V
  #29 = Utf8               com/atguigu/java1/MethodAreaDemo
  #30 = Utf8               java/lang/Object
  #31 = Utf8               java/lang/System
  #32 = Utf8               out
  #33 = Utf8               Ljava/io/PrintStream;
  #34 = Utf8               java/io/PrintStream
  #35 = Utf8               println
  #36 = Utf8               (I)V
{
  public com.atguigu.java1.MethodAreaDemo();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init
         4: return
      LineNumberTable:
        line 7: 0
```

```
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       5     0  this   Lcom/atguigu/java1/MethodAreaDemo;

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=5, args_size=1
         0: sipush        500
         3: istore_1
         4: bipush        100
         6: istore_2
         7: iload_1
         8: iload_2
         9: idiv
        10: istore_3
        11: bipush        50
        13: istore        4
        15: getstatic     #2                  // Field java/lang/System.out:Lja
        18: iload_3
        19: iload         4
        21: iadd
        22: invokevirtual #3                  // Method java/io/PrintStream.pri
        25: return
      LineNumberTable:
        line 9: 0
        line 10: 4
        line 11: 7
        line 12: 11
        line 13: 15
        line 14: 25
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      26     0  args   [Ljava/lang/String;
            4      22     1     x   I
            7      19     2     y   I
           11      15     3     a   I
           15      11     4     b   I
}
```

```
97 SourceFile: "MethodAreaDemo.java"
98
```

1. 程序开始执行, 首先加载 main方法的 参数 args

2. 然后将500 short sipush 压入操作数栈

3. istore_1 将500 出栈, 存到本地变量表 1的位置

4. byte bipush 将100 压入 操作数栈

5. istore_2 将100 出栈, 并放进本地变量表2的位置;

6. iload_1 (byte,short,int,boolean,char)都以int方式存在 本地变量表里, 读取LA1位置的数值, 压入操作数栈内

7. 本地变量表是数组, 操作数栈是栈, 所以500在下面, 先进后出! 这里栈也是数组实现的, 通过字节码文件,看到 长度(深度)是3

8. 做除法, 会把500 100 从栈里面取出来, 得到5 在压进栈里面; 接下来 istore_3 把5取出来,放入本地变量表3的位置;

9. 将50压入栈

10. 将50出栈, 保存到局部变量表中

11. 通过字节码解读, 发现就是调了一下 System.out; 方法压如操作数栈

12. 读取本地变量表3位置数, 并压入栈

13. 读取本地变量表4位置数, 并压入栈

14. 做了一个 相加的运算

15. 执行虚数方法, 输出

16. 结束方法调用

小结:

程序计数器, 会加载当前指令行的行数, 因为万一cpu切走, 轮询的方式, 要记录执行到哪一行了!

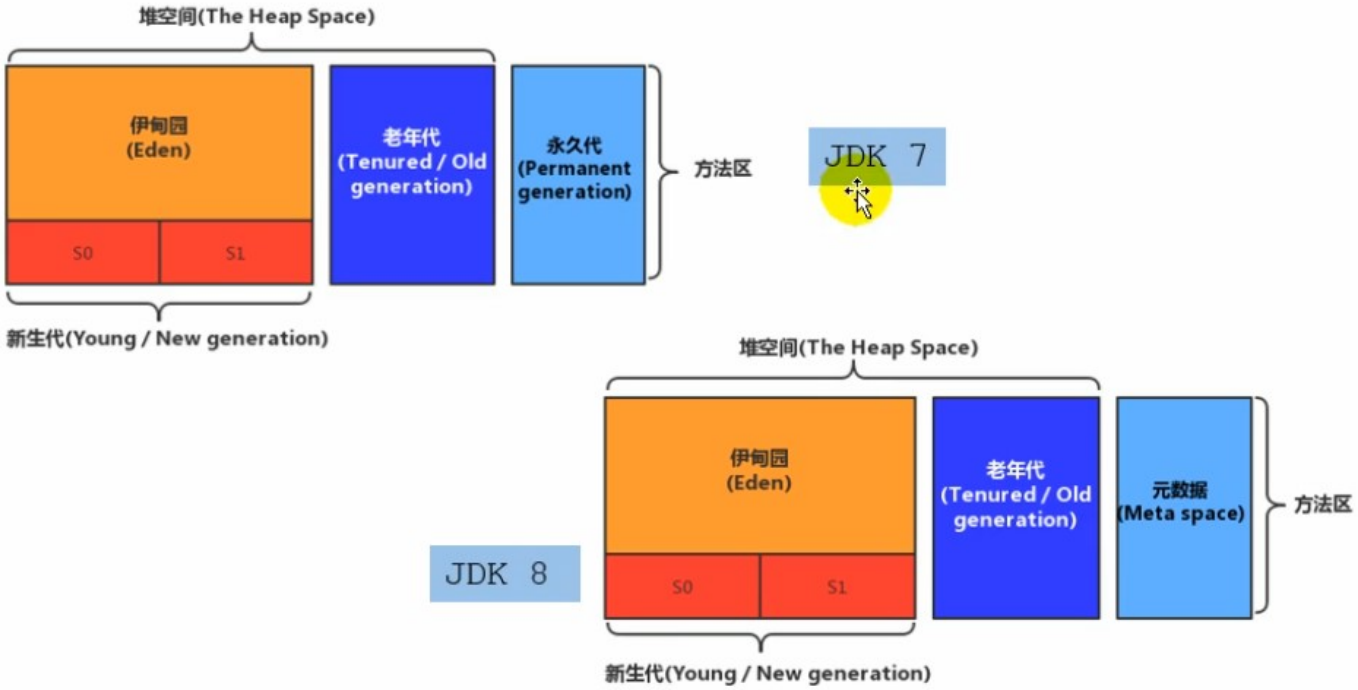
方法区的演进细节

- 在jdk7及以前，习惯上把方法区，称为永久代。jdk8开始，使用元空间取代了永久代。
- 一种独特的思路, 将方法区理解为接口, 以前的实现是永久代, 现在的实现是元空间;

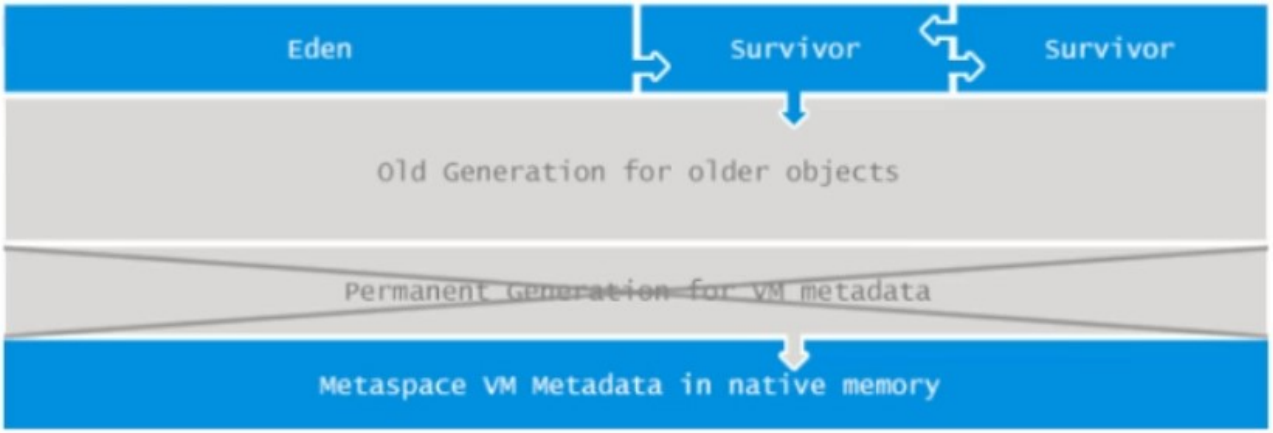
```
In JDK 8, classes metadata is now stored in the native heap and this space is c
```

- 本质上，方法区和永久代并不等价。仅是对hotspot而言的。《Java虚拟机规范》对如何实现方法区, 不做统一要求。例如：BEA JRockit/ IBM J9中不存在永久代 的概念。

现在来看，当年使用永久代，不是好的idea。导致Java程序更容易OOM (超过-XX:MaxPermSize上限)



- 而到了JDK 8, 终于完全废弃了永久代的概念, 该用于JRocket、J9一样在本地内存中实现的元空间(Metaspace)来代替



- 元空间的本质和永久带类似, 都是对JVM规范中方法区的实现。不过元空间与永久代最大的区别在于: 元空间不在虚拟机设置的内存中, 而是使用本地内存;
- 永久代、元空间二者并不只是名字变了，内部结构也调整了。
- 根据<<Java虚拟机规范>>如果方法区无法满足新的内存分配需求时,将抛出OOM异常;

明确

1. 首先明确: 只有HotSpot才有永久代;

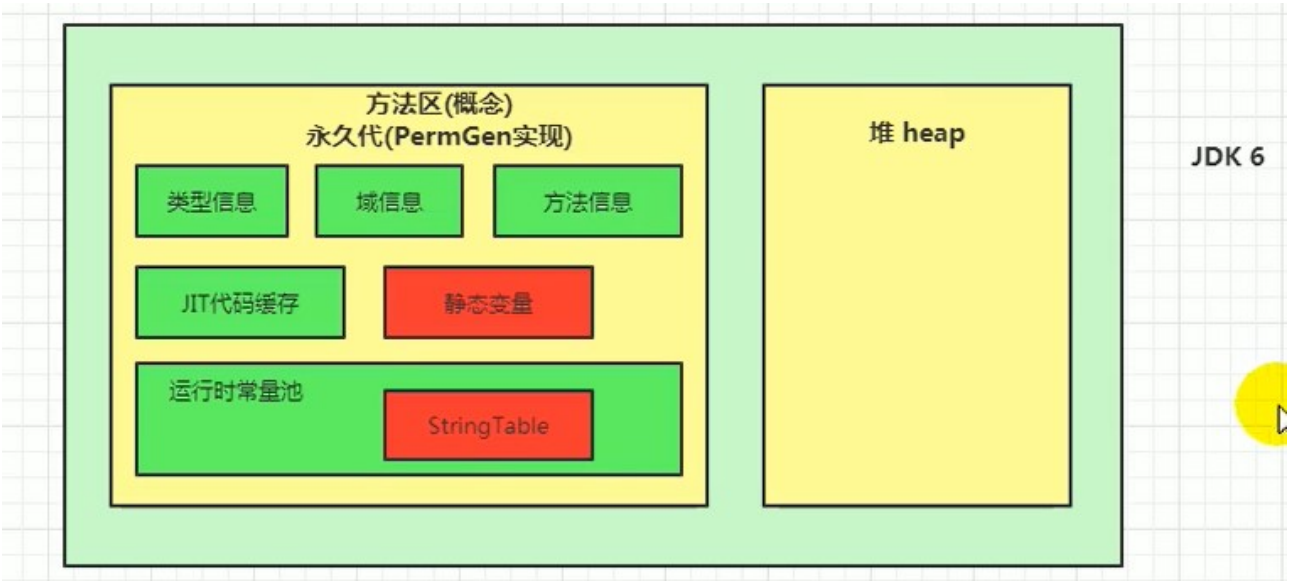
BEA, JRockit, IBM, J9等来说, 是不存在永久代的概念的;

原则上如何实现方法区属于虚拟机实现细节, 不受 <<Java虚拟机规范>>管束, 并不要求统一;

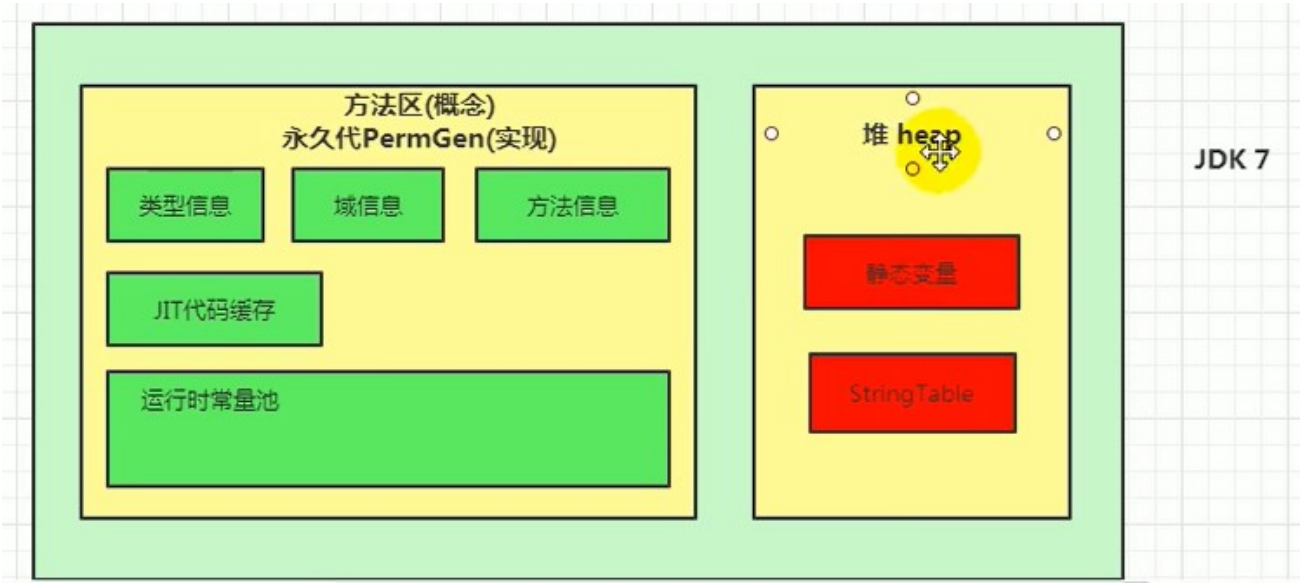
2. HotSpot中方法区的变化

jdk1.6及之前	有永久代(permanent generation)，静态变量存放在永久代上
jdk1.7	有永久代，但已经逐步“去永久代”，字符串常量池、静态变量移除，保存在堆中
jdk1.8及之后	无永久代，类型信息、字段、方法、常量保存在本地内存的元空间，但字符串常量池、静态变量仍在堆

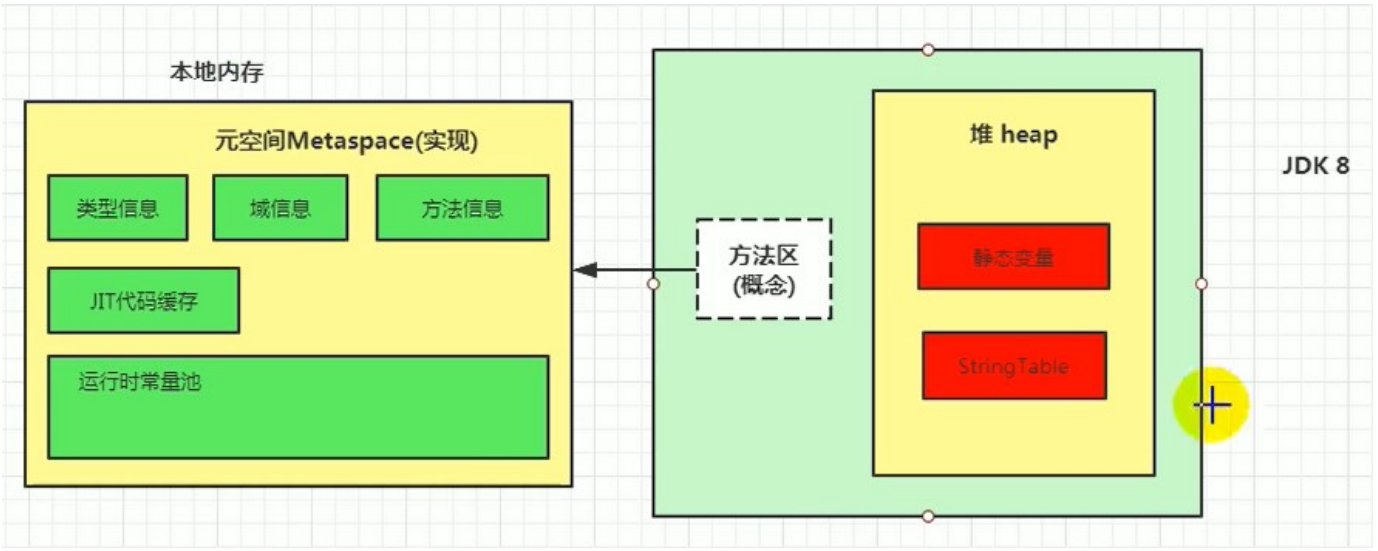
JDK 6



JDK 7 此时 方法区, 还是用的 虚拟机的内存;



JDK 8　使用本地内存, 不再用虚拟内存;



永久代被元空间替换原因

- 随着Java8的到来，HotSpot VM中再也见不到永久代了。但是这并不意味着类的元数据信息消失了。这些数据被移到了一个与堆不相连的本地内存区域，这个 区域叫做元空间（Metaspace)。
- 由于类的元数据分配在本地内存中，元空间的最大可分配空间就是系统可用内存空间。
- 这项改动是很有必要的，原因有：

1)为永久代设置空间大小是很难确定的

在某些场景下，如果动态加载类过多，容易产生Perm区的OOM。比如某个实际Web工程中，因为功能点比较多，在运行过程中，要不断动态加载很多类，经常出现致命错误。

"Exception in thread *dubbo client x.x connector1
java.lang.OutOtTviemoryError: PermGen space"

而元空间和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。

2)对永久代进行调优是很困难的。**不好分析**

> The proposed implementation will allocate class meta-data in native memory and move interned Strings and class statics to the Java heap. Hotspot will explicitly allocate and free the native memory for the class meta-data. Allocation of new class meta-data would be limited by the amount of available native memory rather than fixed by the value of -XX:MaxPermSize, whether the default or specified on the command line.

官方文档很明确说明, 字符串常量池 和 静态变量 在 java heap 中;

StringTable移入堆中原因

jdk7中将StringTable放到了堆空间; 因为永久代的回收效率很低, 在full gc的时候才会触发;

而full gc是老年代的空间不足, 永久代不足时才会触发;

这就导致了StringTable回收效率不高; 而我们开发中会有大量的字符串被创建, 回收效率低, 导致永久代内存不足;

放到堆里, 能及时回收内存;

静态变量证明在堆中

```
/**
 * 结论：
 * 静态引用对应的对象实体始终都存在堆空间
 *
 * jdk7：
 * -Xms200m -Xmx200m -XX:PermSize=300m -XX:MaxPermSize=300m -XX:+PrintGCDetails
 * jdk 8：
 * -Xms200m -Xmx200m -XX:MetaspaceSize=300m -XX:MaxMetaspaceSize=300m -XX:+Prin
 * @author shkstart  shkstart@126.com
 * @create 2020  21:20
```

```
 */
public class StaticFieldTest {
    private static byte[] arr = new byte[1024 * 1024 * 100];//100MB

    public static void main(String[] args) {
        System.out.println(StaticFieldTest.arr);

//        try {
//            Thread.sleep(1000000);
//        } catch (InterruptedException e) {
//            e.printStackTrace();
//        }
    }
}
```

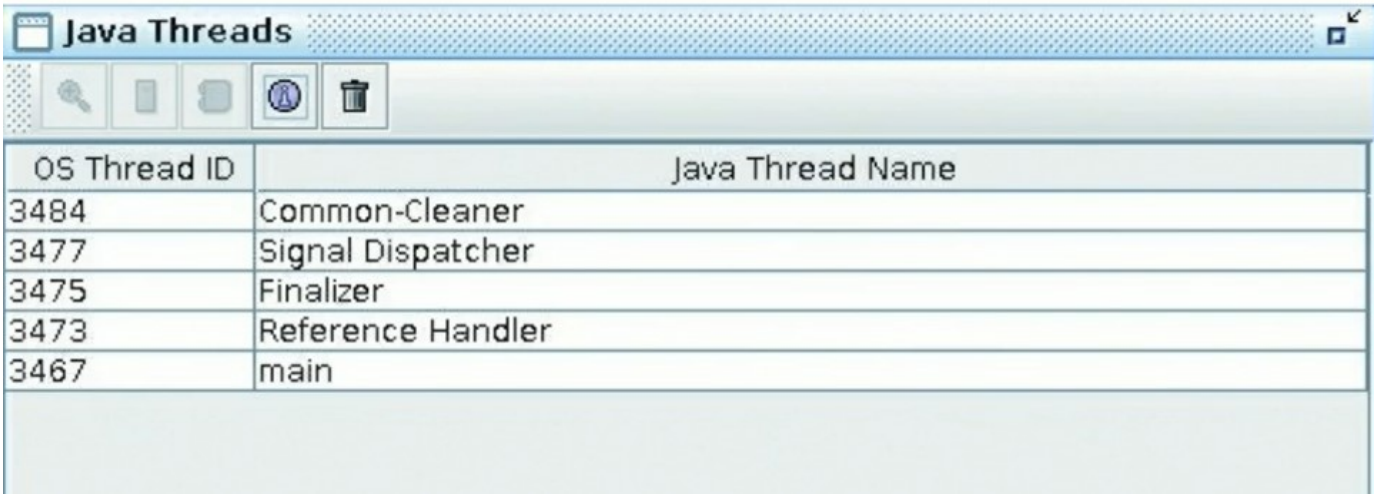
```
[B@1540e19d
Heap
 PSYoungGen      total 59904K, used 5184K [0x00000000fbd80000, 0x0000000100000000, 0x0000000100000000)
  eden space 51712K, 10% used [0x00000000fbd80000,0x00000000fc2903a0,0x00000000ff000000)
  from space 8192K, 0% used [0x00000000ff800000,0x00000000ff800000,0x0000000100000000)
  to   space 8192K, 0% used [0x00000000ff000000,0x00000000ff000000,0x00000000ff800000)
 ParOldGen       total 136704K, used 102400K [0x00000000f3800000, 0x00000000fbd80000, 0x00000000fbd80000)
  object space 136704K, 74% used [0x00000000f3800000,0x00000000f9c00010,0x00000000fbd80000)
 Metaspace       used 3497K, capacity 4498K, committed 4864K, reserved 1056768K
  class space    used 387K, capacity 390K, committed 512K, reserved 1048576K

Process finished with exit code 0
```

new的对象, 始终都放在堆里面! 我们要证明的是 指向对象的 静态变量存放位置, 所以看下面深入案例

```
/**
 * 《深入理解Java虚拟机》中的案例:
 * staticObj、instanceObj、localObj存放在哪里?
 * @author shkstart  shkstart@126.com
 * @create 2020  11:39
 */
public class StaticObjTest {
    static class Test {
        static ObjectHolder staticObj = new ObjectHolder();
        ObjectHolder instanceObj = new ObjectHolder();

```

```
12          void foo() {
13              ObjectHolder localObj = new ObjectHolder();
14              System.out.println("done");
15          }
16      }
17
18      private static class ObjectHolder {
19      }
20
21      public static void main(String[] args) {
22          Test test = new StaticObjTest.Test();
23          test.foo();
24      }
25  }
```

使用JHSDB工具进行分析，这里细节略掉。



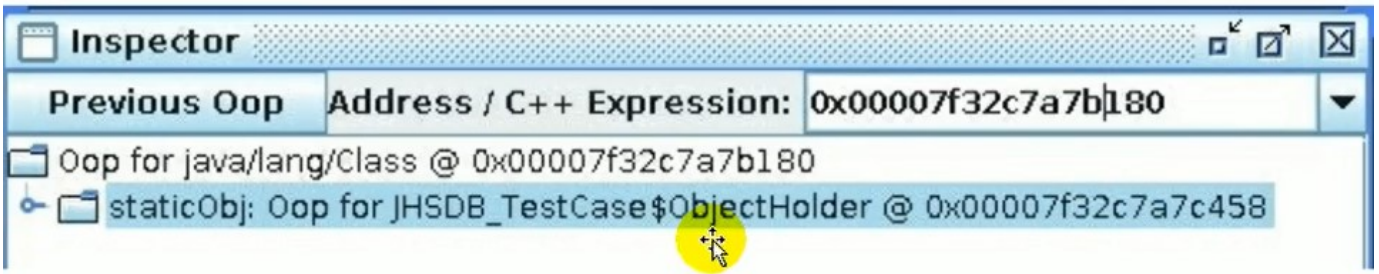
instanceObj 成员变量也存储在 堆中，对象也在堆中，但是变量的引用存在栈中

staticObj随着Test的类型信息存放在方法区，instanceObj随着Test的对象实例存放在Java堆，localObject则是存放在foo()方法栈帧的局部变量表中。

```
hsdb>scanoops 0x00007f32c7800000 0x00007f32c7b50000 JHSDB_TestCase$ObjectHolder
0x00007f32c7a7c458 JHSDB_TestCase$ObjectHolder
0x00007f32c7a7c480 JHSDB_TestCase$ObjectHolder
0x00007f32c7a7c490 JHSDB_TestCase$ObjectHolder
```

测试发现：三个对象的数据在内存中的地址都落在Eden区范围内，所以结论：只要是对象实例必然会在Java堆中分配。

接着，找到了一个引用该staticObj对象的地方，是在一个java.lang.Class的实例里，并且给出了这个实例的地址，通过Inspector查看该对象实例，可以清楚看到这确实是一个java.lang.Class类型的对象实例，里面有一个名为staticObj的实例字段：



从《Java虚拟机规范》所定义的概念模型来看，所有Class相关的信息都应该存放在方法区之中，但方法区该如何实现，《Java虚拟机规范》并未做出规定，这就成了一件允许不同虚拟机自己灵活把握的事情。JDK 7及其以后版本的HotSpot虚拟机选择把静态变量与类型在Java语言一端的映射Class对象存放在一起，存储于Java堆之中，从我们的实验中也明确验证了这一点

方法区的垃圾回收

有些人认为方法区（如Hotspot虚拟机中的元空间或者永久代）是没有垃圾收集行为的，其实不然。《Java虚拟机规范》对方法区的约束是非常宽松的，提到过可以不要求虚拟机在方法区中实现垃圾收集。事实上也确实有未实现或未能完整实现方法区类型卸载的收集器存在（如JDK 11时期的ZGC收集器就不支持类卸载）。

一般来说这个区域的回收效果比较难令人满意，尤其是类型的卸载，条件相当苛刻。但是这部分区域的回收有时又确实是必要的。以前Sun公司的Bug列表中，曾出现过的若干个严重的Bug就是由于低版本的HotSpot虚拟机对此区域未完全回收而导致内存泄漏。

方法区的垃圾收集主要回收两部分内容：常量池中废弃的常量和不再使用的类型。

- 先来说说方法区内常量池之中主要存放的两大类常量：字面量和符号引用。

  字面量比较接近]ava语言层次的常量概念，如: 文本字符串、被声明为 final 的常量值等。

  而符号引用则属于编译原理方面的概念，包括下而三类常量：

  > 1、类和接口的全限定名

  > 2、字段的名称和描述符

  > 3、方法的名称和描述符

- Hotspot虚拟机对常量池的回收策略是很明确的，只要常量池中的常量没有 被任何地方

引用，就可以被回收。

- 回收废弃常量与回收]ava堆中的对象非常类似。

类的卸载，条件苛刻，效果还不好

- 判定一个常量是否“废弃”还是相对简单，而要判定一个类型是否属于“不再被使用的类”的条件就比较苛刻了。
  需要同时满足下面三个条件：

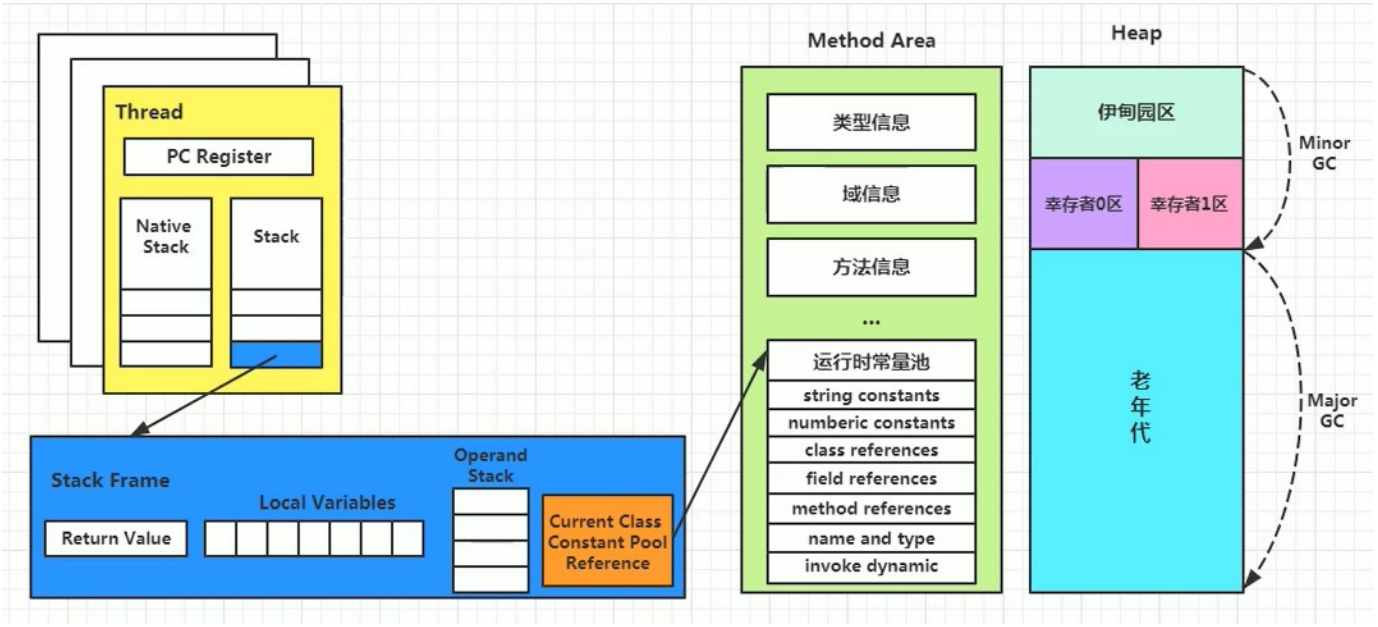
>该类所有的实例都已经被回收，也就是]ava堆中不存在该类及其任何派生子类的实例。

>加载该类的类加载器已经被回收，这个条件除非是经过精心设计的可替换类加载器的场景，如OSGi、JSP的重加载等，否则通常是很难达成的。

>该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

- Java虚拟机被允许对满足上述三个条件的无用类进行回收，这取说的仅仅是 “被允许”，而并不是和对象一样，没有引用了就必然会回收。关于是否要对类型进行回收，HotSpot虚拟机提供了 -Xnoclassgc参数进行控制，
  还可以使用-verbose: class以及 -XX:+TraceClass-Loading、-XX:+TraceClassUnLoading查看类加载和卸载信息
- 在大量使用反射、动态代理、CGLib等字节码框架，动态生成JSP以及OSGi这类频繁自定义类加载器的场景中，**通常都需要java虚拟机具备类型卸载的能力，以保证不会对方法区造成过大的内存压力。**

总结



常见问题，用来复习

```
百度
三面：说一下JVM内存模型吧，有哪些区？分别干什么的？

蚂蚁：
Java8的内存分代改进
JVm 内存分哪几个区，每个区的作用是什么？
1面：JVM内存分步/内存结构？栈和堆的区别？堆的结构？为什么两个survivior区？
2面：Eden和Survivor的比例分配

小米：
jvm内存分区，为什么要有新生代和老年代

字节跳动：
2面：Java的内存分区
2面：讲讲jvm运行时数据库区
什么时候对象会进入老年代？


京东：
JVM的内存结构，Eden和Survivor比例
JVM内存为什么要分成新生代,老年代,持久代（我们就认为1.7之前的永久代）；新生代为什么要分
```

天猫:
1面: Jvm内存模型以及分区, 需要详细到每个区放什么;
1面: Jvm内存模型, java8做了什么修改

拼多多:
jvm 内存分哪几个区, 每个区的作用是什么

美团:
java内存分配
jvm的永久代中会发生垃圾回收嘛
1面: jvm内存分区, 为什么要有新生代和老年代;