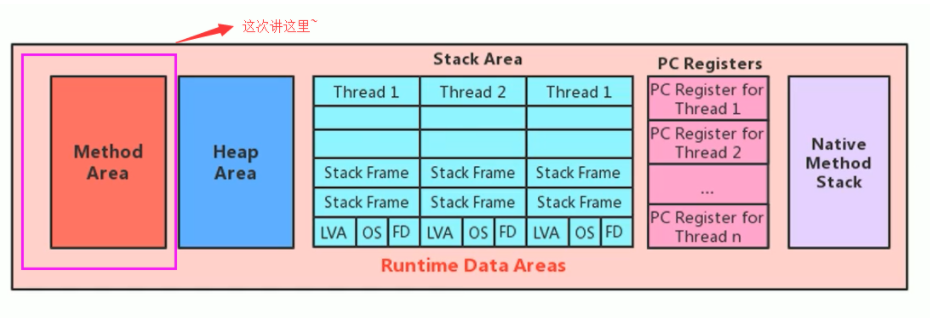
# 1.栈，堆，方法区的交互关系

从内存结构角度来看



从线程共享与否的角度来看



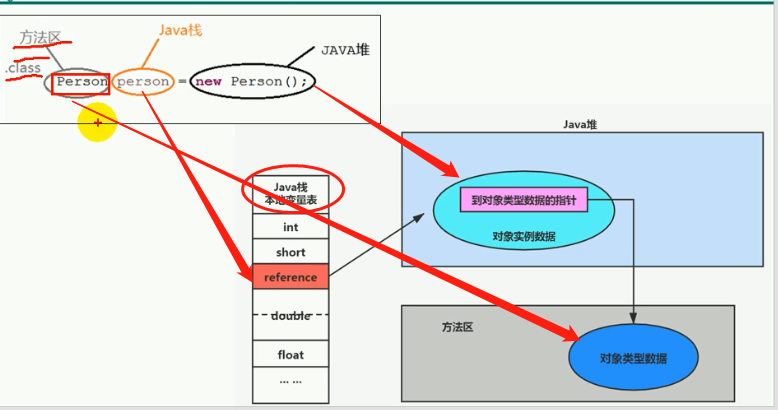
ThreadLocal：如何保证多个线程在并发环境下的安全性？典型应用就是数据库连接管理，以及独立会话管理

## 异常和GC区域总结

* 方法区： 异常 OOM:MetaSpace GC
* 堆： 异常 OOM:Java heap space MinorGC
* 虚拟机栈： StackOverflowError + OOM
* 程序计数器： 都没有

## 从创建对象的角度观察运行时数据区的分工

Person person = new Person();

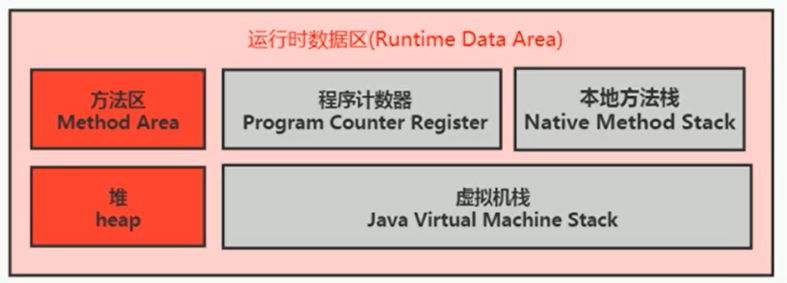


* Person是对象类型信息 存储于方法区中
* person是对象变量，如果是在方法中，则存储在栈中的局部变量表中。
* Person() 是对象实体，存储于堆中
* Person对象中，会有个指针指向方法区中Person类型数据，表名这个person对象是用方法区中的Person类new出来的。

# 方法区

## 2.1 方法区的位置

1. 《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。
2. 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。
3. 所以，方法区可以看作是一块独立于Java堆的内存空间



## 2.2 方法区的理解

**方法区主要存放的是 Class，而堆中主要存放的是实例化的对象**

1. 方法区（Method Area）与Java堆一样，是各个线程共享的内存区域
2. 多个线程同时加载统一个类时，只能有一个线程能加载该类，其他线程只能等等待该线程加载完毕，然后直接使用该类，即类只能加载一次。
3. 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，逻辑上连续。
4. 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。
5. 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：
   1. java.lang.OutofMemoryError:PermGen space

或者

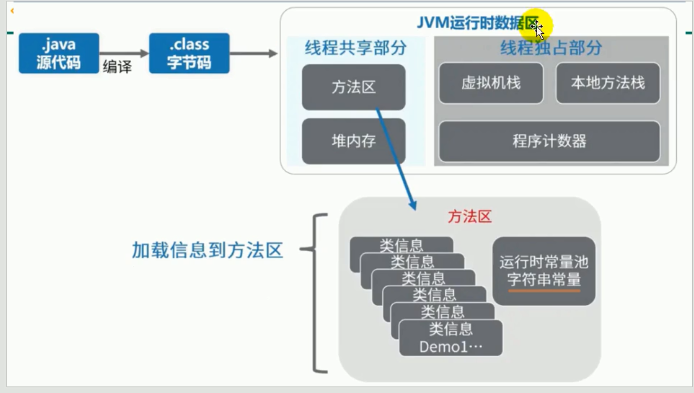
* 1. java.lang.OutOfMemoryError:Metaspace
* 举例说明方法区 OOM
  1. 加载大量的第三方的jar包
  2. Tomcat部署的工程过多（30~50个）
  3. 大量动态的生成反射类

1. 关闭JVM就会释放这个区域的内存。

代码举例：

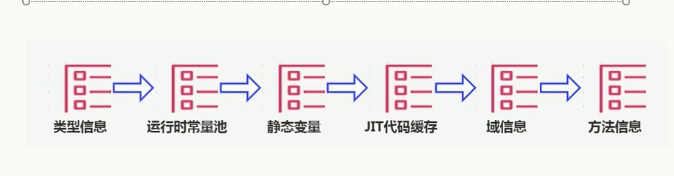
|  |
| --- |
| public class EdenSurvivorTest {  public static void main(String[] args) {  System.out.println("我只是来打个酱油~");  try {  Thread.sleep(1000000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } |

## 2.3 方法区存储的信息



《深入理解Java虚拟机》书中对方法区存储内容描述如下：

* **方法区用于存储已经经过类加载的：**
  1. 类信息：包括（接口、枚举、注解。。。）、域信息、方法信息
  2. 运行时常量池
  3. 静态变量 包括static和static final变量
  4. 即时编译器编译后的代码缓存。



### 1.类信息

#### 类的声明信息

对于每个加载过的**类型（类、接口、枚举、注**解），JVM必须在方法区中存储以下**类型信息：**

1. 此类型的**完整有效名称**（全名=包名.类名）
2. 此类型**直接父类**的完整有效名（接口或者Object类 没有父类）
3. 此类型的**修饰符**
4. 此类型**直接接口的一个有序列表**

#### 类的域信息(成员变量)

方法区存储所有域信息，以及域的声明顺序.

域信息就是域的声明信息 包括：域名、域类型、域修饰符

字节码文件反编译后查看信息：



#### 方法信息

方法区存储方法信息，以及方法的声明顺序。

方法信息：

1. 方法的声明信息：方法名、返回值类型、参数的数量和类型（按顺序）、修饰符
2. 方法的字节码、操作数栈、局部变量表及大小、异常表【这四个abstract和native方法没有，因为抽象方法没有方法体】

异常表包括每个异常处理的开始位置、结束为止、代码处理在程序计数器中的偏移地址、被捕获的异常类在常量池中的索引。

**操作数栈和局部变量表是栈桢需要的参数，这两个abstract和native方法都不需要。置于异常为什么abstract方法没有，还不确定。**

#### 方法区查看示例

演示代码：

|  |
| --- |
| import java.io.Serializable;  public class MethodInnerStrucTest extends Object implements Comparable<String>, Serializable {  *//属性* public int num = 10;  private static String *str* = "测试方法的内部结构";   *//构造器没写   //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;  }  } |

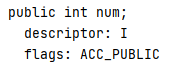
字节码文件反编译后查看信息：

* 类信息

在运行时方法区中，类信息中记录了哪个加载器加载了该类，同时类加载器也记录了它加载了哪些类

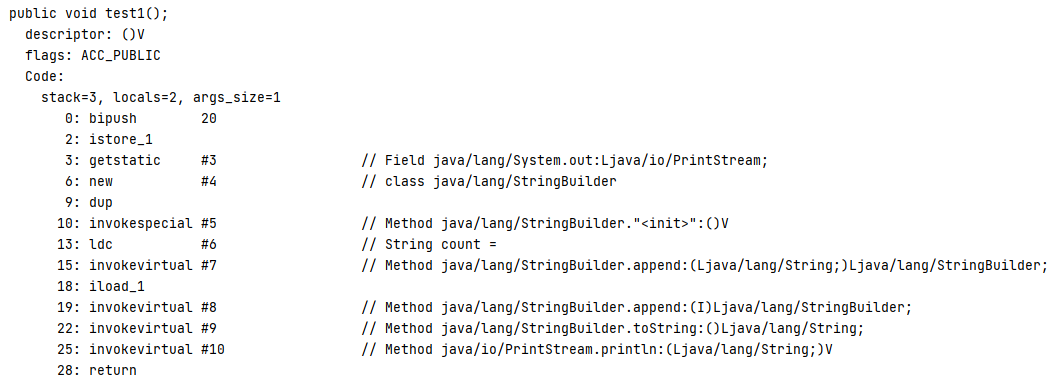


* 域信息

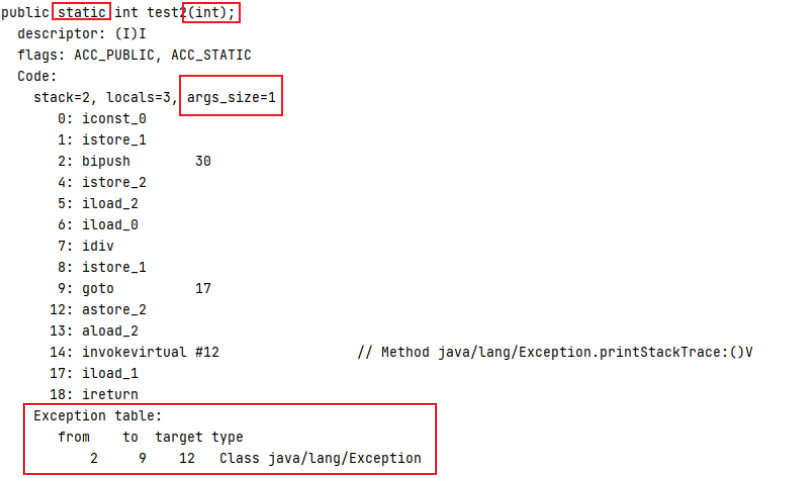


* 方法信息

test1()方法



test2()方法



test1非静态无参，test2为静态有一个参数，但是显示参数列表个数都是1，这是因为对于非静态方法来说，他会默认存储一个this，在局部变量表中的0号槽位。

如果有参数，那么this不计入这个args\_size，虽然依然存在。

异常列表；

### 2.Non-final类变量和final类变量

特点:

1. 静态变量和类关联在一起，随着类的加载而加载，他们成为类数据在逻辑上的一部分
2. 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
3. 类变量与实例无关，即使实例对象为null，调用类变量和类方法仍然可以。因为我们只要知道这个实例对象的类型就可以了。

|  |
| --- |
| 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!");  }  }  hello!  1 |
| 这个java代码会编译成两个.class文件：    查看Order反编译文件： |

* Static final 变量在编译期间就分配赋值了
* 可以看出反编译的文件中 static final在编译期间的链接阶段就已经初始化赋值了。但是非final的静态变量不会初始化赋值。

### [3.运行时常量池](#_运行时常量池)

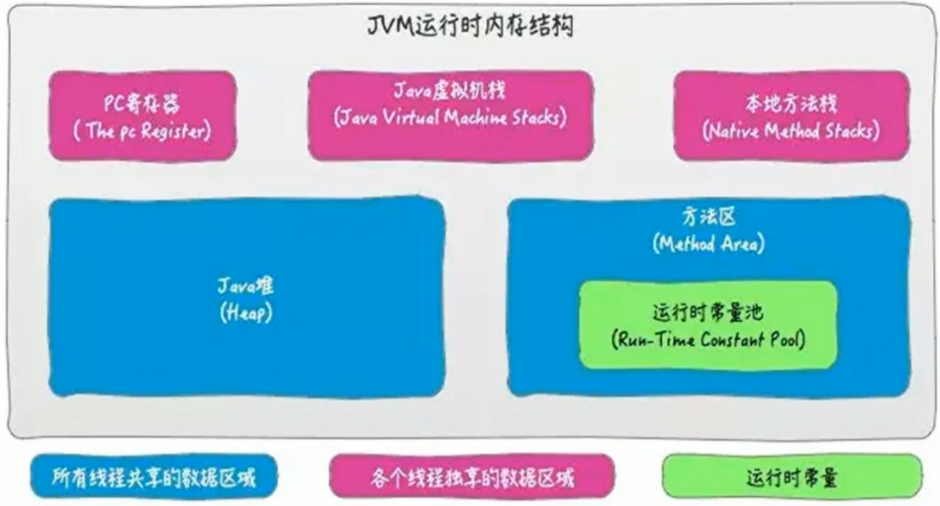
### 4.JIT编译后的热点代码

## 2.4 运行时常量池VS常量池

官方文档

https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html

1. 方法区是内存空间，内部包含了运行时常量池
2. 字节码文件是磁盘文件，内部包含了常量池
3. 要弄清楚方法区的运行时常量池，因为方法区中运行时常量池是由Class文件加载而来的，所以得先弄明白Class文件中常量池。



### 常量池

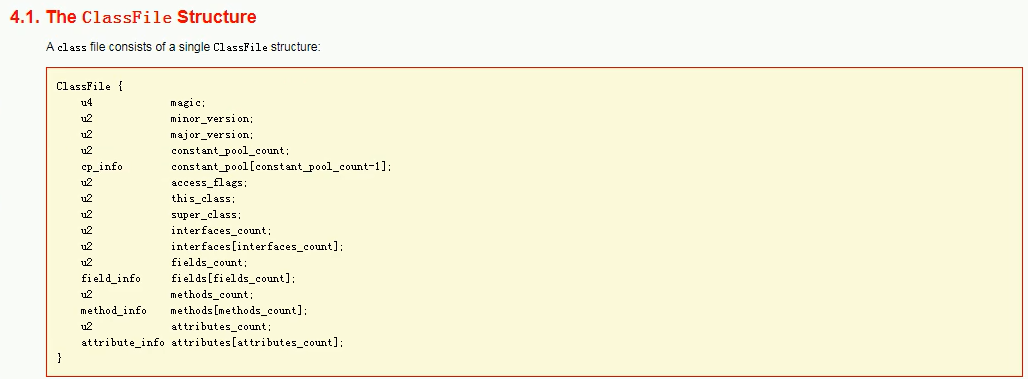
#### Class文件中常量池的理解

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符信息外

还包含一项信息就是**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



#### Class文件结构



#### 常量池保存的信息

* 一个有效的字节码文件中除了包含类信息和常量池表
* 常量池中存储的数据类型



这几种数据类型可以归为两类：字面量和符号引用

字面量：就是基本数据类型的值和String类型的值

符号引用：可以看作是名称

|  |
| --- |
| 常量池中的符号引用实例：  有三种：Method、Class、Field |

**为什么Class文件中要有常量池表？常量池表为啥存储的是字面量和符号引用？**

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；而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

比如：如下的代码：

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



虽然上述代码只有194字节，但是里面却使用了String、System、PrintStream及Object等结构。如果不使用常量池，就需要将用到的类信息、方法信息等记录在当前的字节码文件中，造成文件臃肿。所以我们将所需用到的结构信息记录在常量池中，并通过引用的方式，来加载、调用所需的结构。这里的代码量其实很少了，如果代码多的话，引用的结构将会更多，这里就需要用到常量池了。

#### 常量池总结

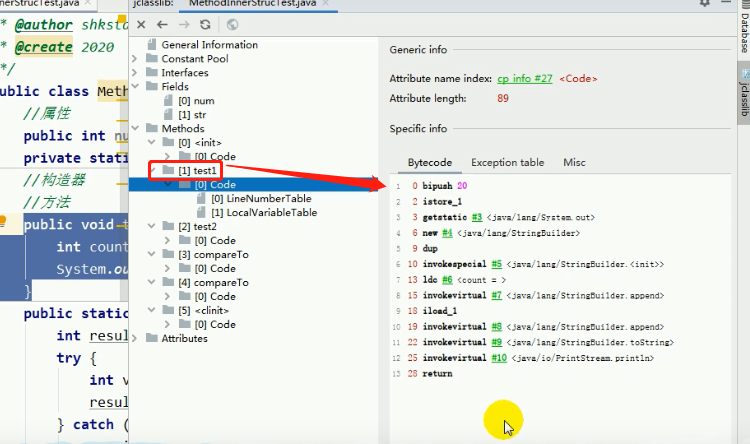
常量池可以看做是Class文件中的一张表，虚拟机指令根据这张常量表找到要执行的类名、方法名、参数类型、字面量等类型

只有真正在运行的时候，符号引用替换为真实的引用，这个过程发生在类加载的连接中的解析阶段，动态链接。

**常量池存储的作用就是存储实际运行时所需要的原材料的信息。**

#### class文件观察实例

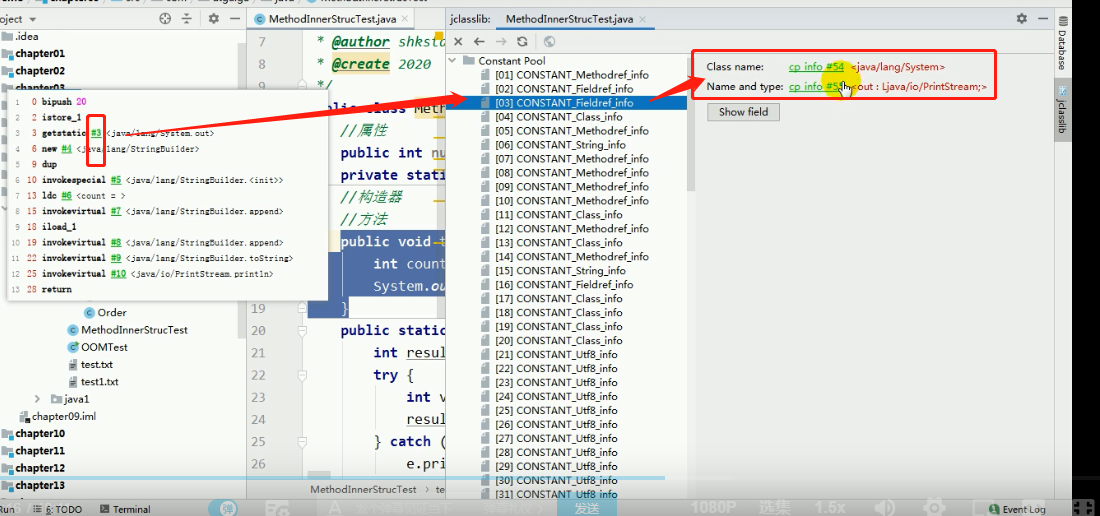
一个方法对应的字节码：



右边是该方法的字节码（ByteCode）:

其中 **#num** 都是符号引用，具体对应到常量池中。

下面观察常量池Constant Pool：



可以看出符号引用存储了一些原料信息。

### 运行时常量池

* **运行时常量池是方法区的一部分**
* **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，并且在类加载后存放到方法区中的运行时常量池中。

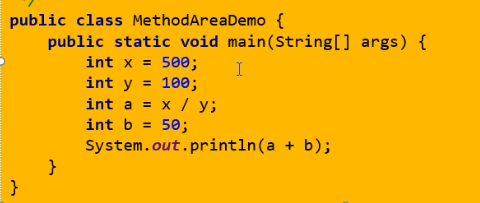
* JVM为每个已加载的类型（类或接口）都维护一个常量池。池中的数据项像数组项一样，是通过索引访问的。
* **运行时常量池中存储的不仅仅是编译期间就已经明确的数值字面量，也包括后面运行期间解析后才能获得的方法或者字段的引用（这是多态的原理）。**这个时候，就不再是常量池中的符号引用了，而是真实地址。

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

这个过程表示为 运行时常量池的动态性。 比如String的itern()方法

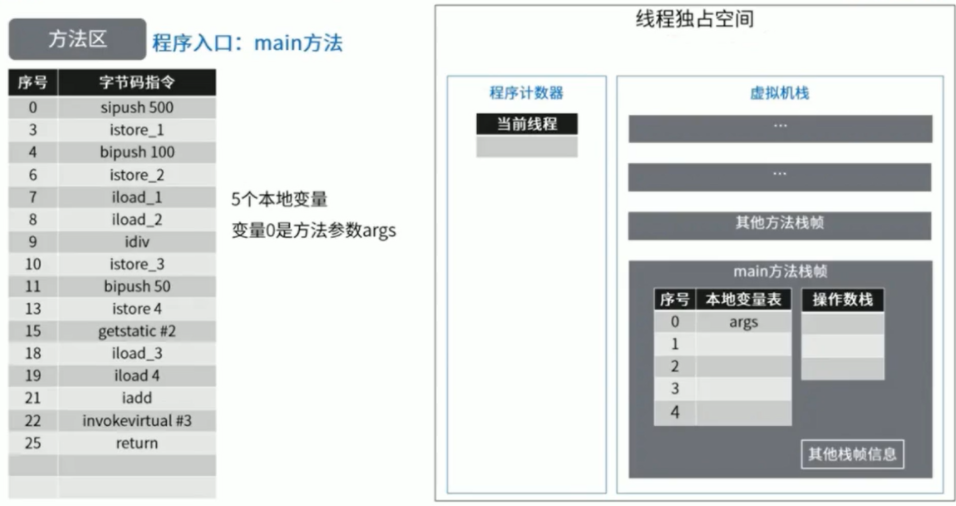
* 运行时常量池类似于传统编程语言中的符号表（symbol table），但是它所包含的数据却比符号表要更加丰富一些。
* 当创建类或接口的运行时常量池时，如果构造运行时常量池所需的内存空间超过了方法区所能提供的最大值，则JVM会抛OutofMemoryError异常。

## 2.5方法区使用举例



**图解字节码指令执行流程：**

1. 初始状态：



1. int x = 500;



然后操作数 500 从操作数栈中取出，存储到局部变量表中索引为 1 的位置



1. int y = 100;

同样：100压入操作数栈，然后从栈中取出放进本地变量表

然后操作数 100 从操作数栈中取出，存储到局部变量表中索引为 2 的位置





1. a = x/y;







10 是将操作数栈顶元素5放进本地变量表

1. int b = 50;





1. System.out.println(a + b);







关于【符号引用 --> 直接饮用】的理解

* 上面代码调用 System.out.println() 方法时，首先需要看看 System 类有没有加载，再看看 PrintStream 类有没有加载
* 如果没有加载，则执行加载，执行时，将常量池中的符号引用（字面量）转换为直接引用（真正的地址值）

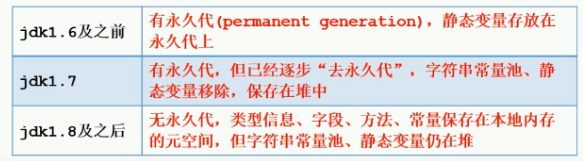
关于程序计数器的说明

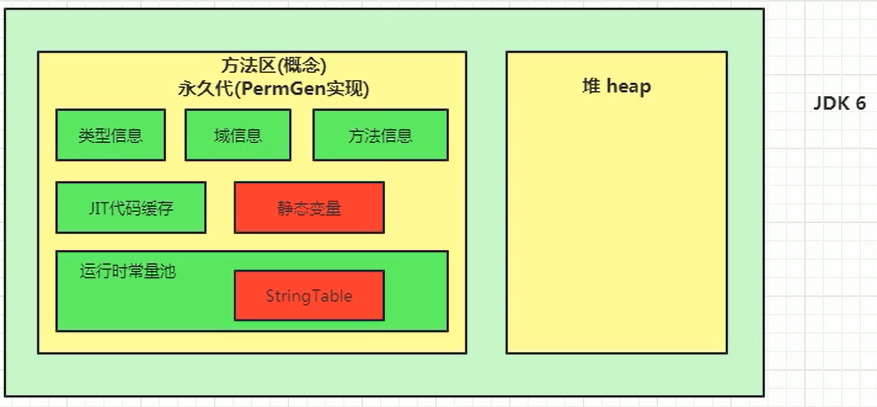
程序计数器始终计算的都是当前代码运行的位置，目的是为了方便记录方法调用后能够正常返回，或者是进行了CPU切换后，也能回来到原来的代码进行执行。

## 2.6方法区的演进(☆☆☆)

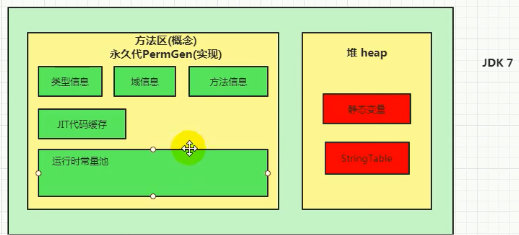
永久代和元空间都只是方法区的实现方式

1. 只有HotSpot才有永久代。
2. Jdk1.6 及之前，有永久代，静态变量在永久代上
3. Jdk1.7 有永久代，但是已经逐步“去永久代”，1.7已经把字符串常量池和静态变量从方法区中移除了，而是保存到堆中了。
4. Jdk1.8及之后就没有永久代了，而是用元空间实现了方法区。元空间在直接内存，而不是JVM内存了，类信息、常量(字面量)都保存在元空间中，但是静态变量和字符串常量池还在堆中。





注意：字符串常量池原本是在运行时常量池的。



注意: 这里的静态变量在jdk1.7中从方法区中转移到了堆内存中，这个是**静态变量，并非对象实体。**

### 永久代为什么要被元空间替代？

首先要明确二者的本质区别：永久代存储于JVM内存，元空间是JVM外的内存；

方法区中存储的数据是类的元数据+常量+符号引用+JIT编译后的热点代码

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 connector' java.lang.OutOfMemoryError: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。

1. 对永久代进行调优是很困难的。

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**不再用的类型**，方法区的调优主要是为了降低Full GC

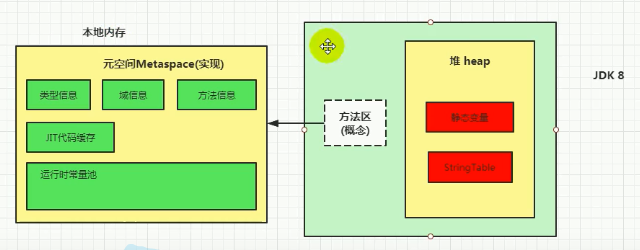
有些人认为方法区（如HotSpot虚拟机中的元空间或者永久代）是没有垃圾收集行为的，其实不然。《Java虚拟机规范》对方法区的约束是非常宽松的，提到过可以不要求虚拟机在方法区中实现垃圾收集。事实上也确实有未实现或未能完整实现方法区类型卸载的收集器存在（如JDK11时期的ZGC收集器就不支持类卸载）。

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

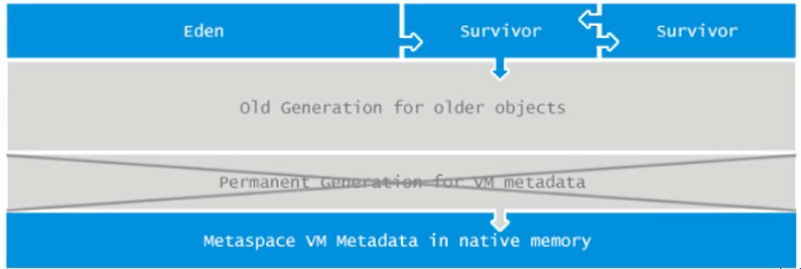
思考1：为什么在JDK1.7中要将静态变量和String Table（字符串常量池）转移到堆中？

**因为永久代回收效率低，只能在full GC才回触发永久代的垃圾回收。**

full GC是由老年代空间不足、永久代不足时(达到初始化大小)才会触发，这就导致字符串常量池回收效率低，在实际开发中会有大量的字符串被创建，回收效率低会导致永久代内存不足，容易产生OOM，而放进堆中能够及时回收内存。



JVM内存结构的变化：



思考2：为什么JDK1.8要用元空间替换永久代？

其实元空间和永久代的本质区别是位置不同，元空间位于堆外内存，永久代放在堆内，那么HotSpot中永久代的设计思想相当于将一切都纳入JVM自己的内存空间来进行管理。

而JVM加载的class的总数，方法的大小等都很难确定，因此对永久代大小的指定难以确定。在某些场景下，如果动态加载类过多，容易产生Perm区(超过-XX:MaxPermsize上限)的OOM。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

### 静态变量在哪里？

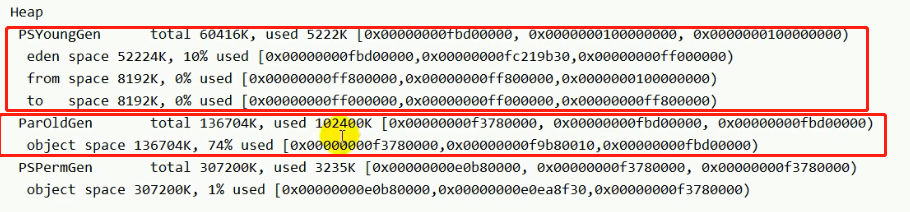
#### Demo1: 查看new的静态对象存放在哪

|  |
| --- |
| *//-Xms200m -Xmx200m -XX:MetaspaceSize=300m -XX:MaxMetaspaceSize=300m -XX:+PrintGCDetails* public class StaticFieldTest {  private static byte[] *arr* = new byte[1024 \* 1024 \* 100];*//100MB* public static void main(String[] args) {  System.*out*.println(StaticFieldTest.*arr*);  }  } |

这里创建了一个数组，是个类对象。

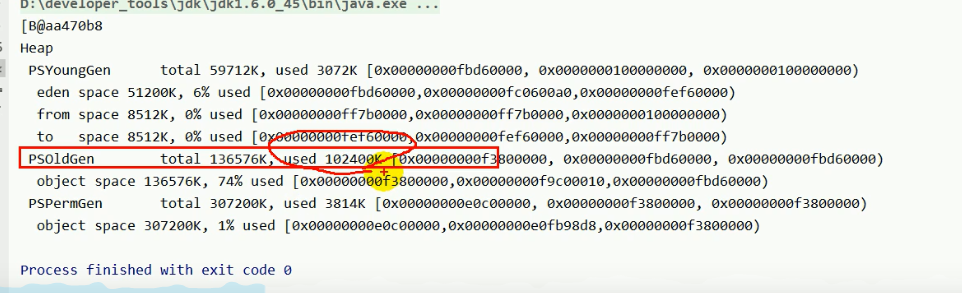
查看各个jdk版本的运行时数据区：

**Jdk6:**



放在老年代中了。

JDK7:



也放在老年代中。

JDK8：



仍然放在老年代中。

结论：  
 静态变量在jdk6/7存在与永久代中，在jdk8存在于堆中  
 静态引用对应的对象实体始终都存在堆空间

#### Demo2: 变量存储在哪里？

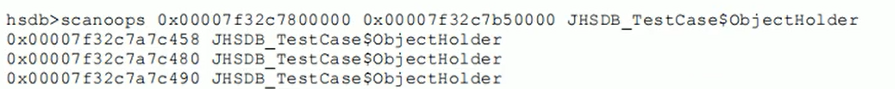
|  |
| --- |
| */\*\*  \* 《深入理解Java虚拟机》中的案例：  \* staticObj、instanceObj、localObj存放在哪里？  \*  \* @author shkstart shkstart@126.com  \* @create 2020 11:39  \*/* public class StaticObjTest {  static class Test {  static ObjectHolder *staticObj* = new ObjectHolder();  ObjectHolder instanceObj = new ObjectHolder();   void foo() {  ObjectHolder localObj = new ObjectHolder();  System.*out*.println("done");  }  }   private static class ObjectHolder {  }   public static void main(String[] args) {  Test test = new StaticObjTest.Test();  test.foo();  } } |

staticObj是个静态变量，引用ObjectHolder对象。

InstanceObj是个普通成员变量，引用ObjectHolder对象

localObj是个局部变量。

1. **首先观察对象都放在哪里：**

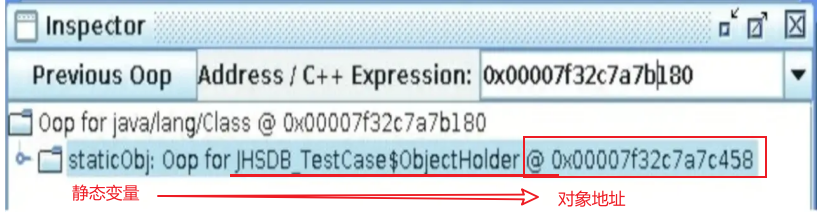


测试发现：三个对象的数据在内存中的地址都落在Eden区范围内，所以结论：只要是对象实例必然会在Java堆中分配。

1. **观察三种变量所存储的位置：**

**staticObj变量随着Test的类型信息存放在方法区**;instance0bj成员变量随着Test的对象实例存放在Java堆;localobject局部变量则是存放在foo()方法栈帧的局部变量表中。

1. 实例变量（也就是成员变量）的生命周期是跟随对象的，属于对象的内容。而对象实例化之后，存放在堆中，所有成员变量也会存在堆中。
2. 静态变量，也叫类变量，属于类型的信息，是存放在方法区中的。静态域存储于定义类型的Class对象中，Class对象就是类模板，在方法区中
3. 局部变量是属于方法（对应栈帧）的，也就存在栈中，即栈帧当中的局部变量表当中。

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。**JDK7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**

**结论：**

1. **静态变量存储在Class对象中也就是堆中！！！**
2. **成员变量随着对象存储在堆中**
3. **局部变量存储在栈桢中。**
4. **所有对象都在堆中。**

#### Class对象和类的元数据

<https://www.zhihu.com/question/59174759>

JDK7以上版本，静态域存储于定义类型的Class对象中，Class对象如同堆中其他对象一样，存在于GC堆中。

java.lang.Class对象，它们从来都是”普通”Java对象，跟其它Java对象一样存在普通的Java堆（GC堆的一部分）里。

类的元数据（元数据并不是类的Class对象！Class对象是加载的最终产品，类的方法代码，变量名，方法名，访问权限，返回值等等都是在方法区的）才是存在方法区的。

## 2.7方法区大小的配置 & OOM

方法区的大小不必是固定的，JVM可以根据应用的需要动态调整。

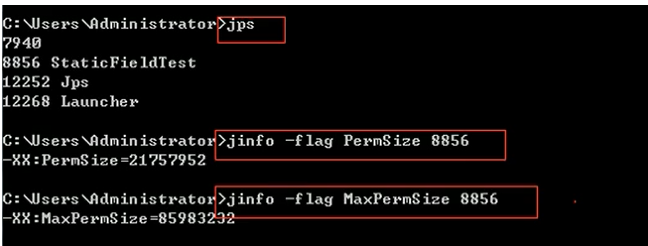
### 方法区大小设置

#### jdk7之前 永久代

* -XX:PermSize 设置初始空间大小 默认为20.75M
* -XX：MaxPermSize 设置方法区最大空间 32位操作系统默认为64M，64位默认为82M

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

* 查看永久代大小



#### Jdk8及以后 元空间

* -XX:MetaspaceSize 设置初始化元空间大小 windows 默认21M
* -XX:MaxMetaspaceSize -1（无限制）

**初始化大小相当于一个水位线，当方法区内存触及到水位线时会FULL GC**，卸载无用的类，然后这个初始值会被重置，新的水位线值取决于GC后释放了多大的空间，如果释放的空间不多，那么在不超过MaxMetaspaceSize情况下，适当提高该值，释放空间过多的时候，适当降低该值。

初始值过低会导致Full GC、调整初始值多次，应该避免这种情况。建议将初始值设置为一个较大的值。

##### 元空间配置示例

|  |
| --- |
| 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...");  }  }  JVM 参数：-XX:MetaspaceSize=100m -XX:MaxMetaspaceSize=100m |

### OOM

解决OOM问题，首先要确定是内存中的对象是否是必要的，也就是弄明白是内存泄漏还是内存溢出。

* Case1: 内存泄漏

什么叫内存泄漏：泄露对象的定义：该对象已经是闲置的资源，但是由于堆当中的闲置对象由于引用链的引用关系无法被回收

如何解决: 通过工具查看泄漏对象到GC Roots 的引用链，找到泄漏对象是通过怎样的路径与GCRoots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。

* Case2:内存溢出

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

解决方法：

1. 从虚拟机堆参数来解决：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
2. 从代码解决： 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。

## 2.8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JDK11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。

* **方法区垃圾回收主要回收两个部分：常量池中废弃的常量和不再使用的类型。**

### 1.常量池中废奔的常量

* + **常量池中的常量指的是：**
    - **字面量：基本数据类型的值、字符串的值、final值**
    - **符号引用：类和接口、字段、方法的名称**
  + **常量池的回收策略：**
    - **只要常量池中的常量没有被任何地方引用，就可以被回收。（似堆中的对象）**

### 2.常量池中不再使用的类型

* **不再被使用的类型的回收：**

必须同时满足以下三个条件，该类型可以叫做不再被使用

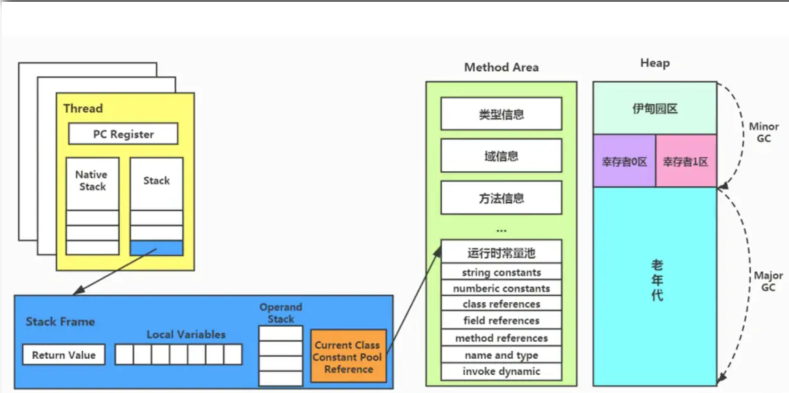
1. **该类的所有实例都被回收**，也就是Java堆中不存在该类及其任何子类的实例
2. **加载该类的类加载器已经被回收**，这个条件除非是经过精心设计的可替换类加载器的场景。
3. **该类对应的java.lang.Class对象没有在任何地方被引用**，无法在任何地方通过反射访问该类的方法。

Java虛拟机被允许对满足上述三个条件的无用类进行回收，这里说的仅仅是“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。

# 运行时数据区总结

* 线程私有结构：程序计数器、虚拟机栈、本地方法栈
* 每个虚拟机栈由由具体的栈帧组成，在栈帧的动态链接中，保存至对方法的引用
* 方法区在 JDK7 之前，使用永久代实现，在 JDK8 之后，使用元空间实现
* Minor GC 针对于新生区，Major GC 针对于老年区，Full GC 针对于整个堆空间和方法区



# 面试题

百度

三面：说一下JVM内存模型吧，有哪些区？分别干什么的？

蚂蚁金服：

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？  
一面： JVM内存分布/内存结构？栈和堆的区别？堆的结构？为什么两个survivor区？  
二面： Eden和Survior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：

二面： Java的内存分区  
二面：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。新生代中为什么要分为Eden和Survivor。

天猫：

一面： Jvm内存模型以及分区，需要详细到每个区放什么。  
一面： JVM的内存模型，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吗？  
一面： jvm内存分区，为什么要有新生代和老年代？