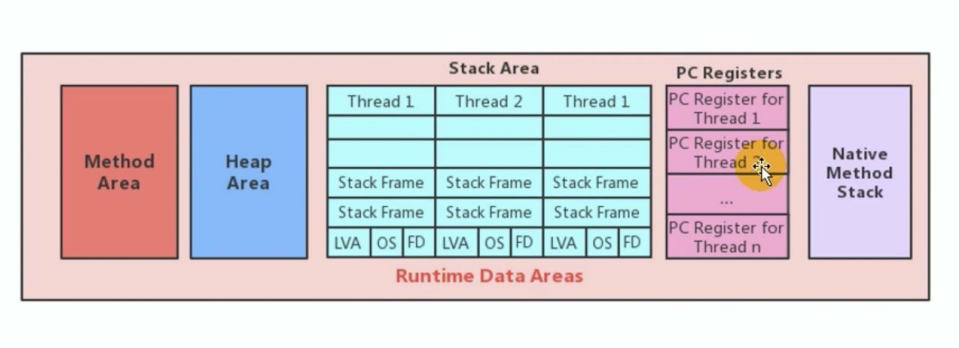
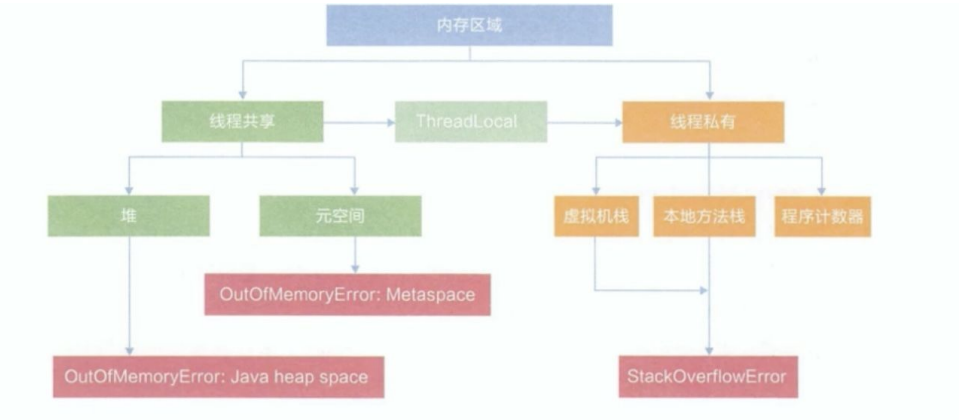
你知道 JVM 的方法区是干什么用的吗?

zempty

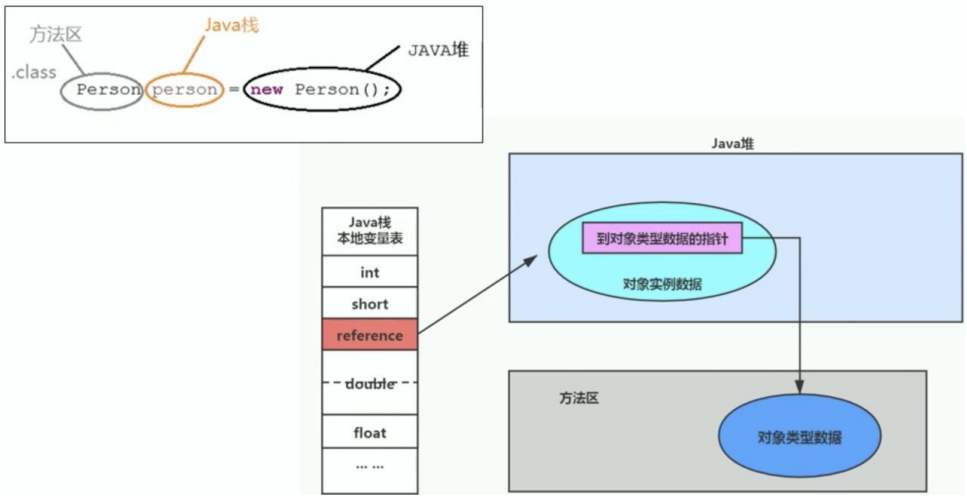
栈、堆、方法区的交互关系



运行时数据区结构图



从线程共享与否的角度来看

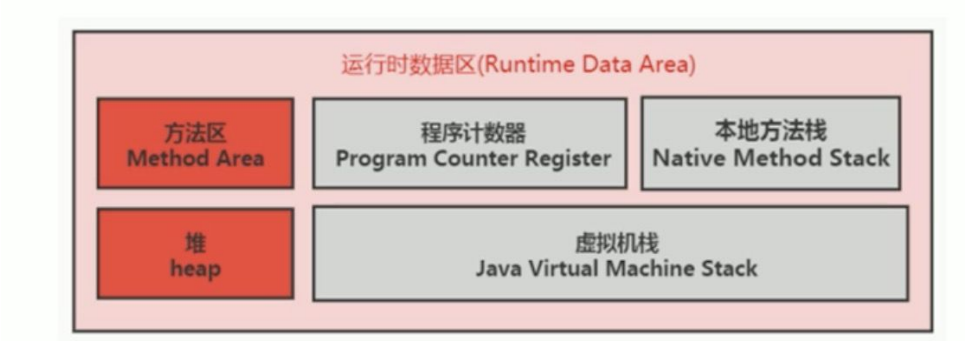


栈、堆、方法区的交互关系示意图

方法区的理解

官文的文档解释：docs.oracle.com/javase/...

《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区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**。



- **方法区的基本理解：**

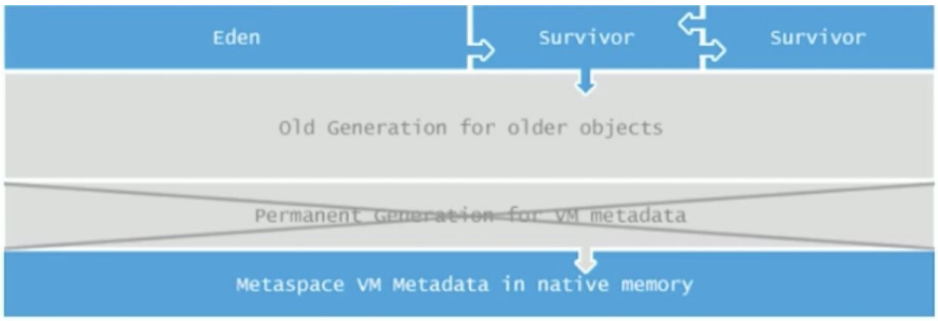
1. 方法区（Method Area) 与 Java 堆一样，是各个线程共享的内存区域。
2. 方法区在 JVM 启动的时候创建，并且它的实际的物理内存空间和 Java 堆区一样都可以是不连续的。
3. 方法区的大小，跟堆空间一样，可以选择固定大小或者可扩展。
4. 方法区的大小决定了系统可以保存多少个类，如果系统定义了太多的类，导致方法区的溢出，虚拟机同样会抛出内存溢出错误： java.lang.OutOfMemoryError:PermGen space 或者 java.lang.OutOfMemoryError:Metaspace
5. 关闭 JVM 就会释放这个内存区域。

- **Hotspot 中方法区的演进**

1. 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。
2. 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）

1. 到了 JDK 8 ,终于完全废弃了永久代的概念，改用与 JRockit,J9一样在本地内存中实现的元空间



1. 元空间的本质同永久代类似，都是对 JVM 规范中方法区的实现。不过元空间与永久代最大的区别在于：元空间不再虚拟机设置的内存当中，而是使用本地内存。
2. 永久代、元空间并不只是名字变了，内部结构也调整了。
3. 根据《 Java 虚拟机规范》的规定，如果方法区无法满足新的内存需求时，将抛出 OOM 异常。

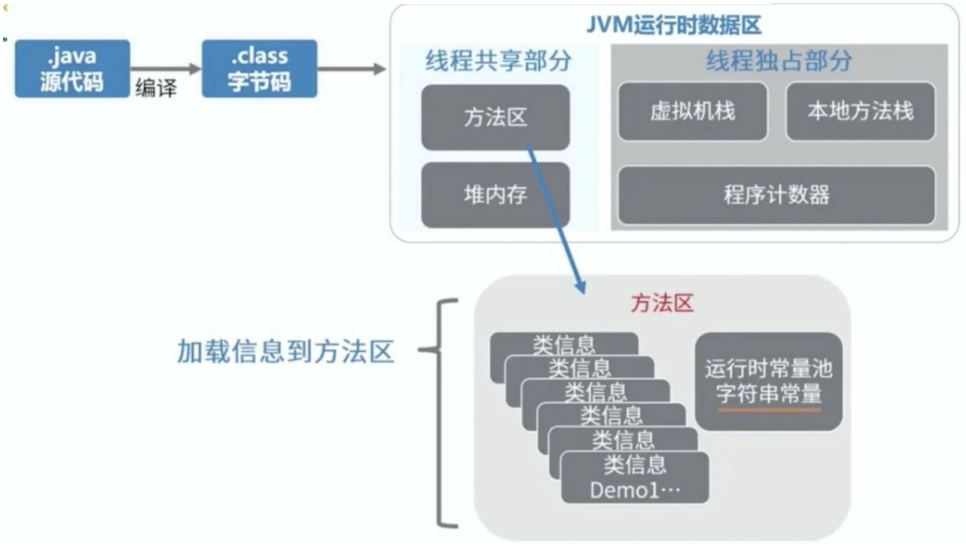
设置方法区的大小与 OOM

- **方法区的大小不必是固定的，jvm 可以根据应用的需要动态调整。**
- **jdk7 及以前：**
  - 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.

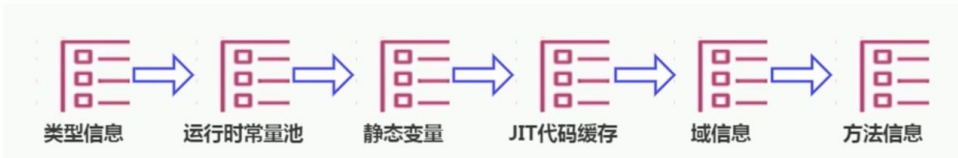
- **jdk8 及以后：**
  - 元数据大小可以使用参数 -XX:MetaspaceSize 和 -XX:MaxMetaspaceSize 指定，替代上述原有的两个参数。
  - 默认值依赖于平台。windows 下，-XX:MetaspaceSize 是 21M， -XX:MaxMetaspaceSize 的值是 -1，即没有限制。
  - 与永久代不同，如果不指定大小，默认情况下，虚拟机会耗尽所有的可用系统内存。如果元数据发生异常，虚拟机一样会抛出异常 OutOfMemoryError:Metaspace
  - -XX:MetaspaceSize 设置初始的元空间大小。对于一个 64 位的服务器 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。新的高水位线取决于 GC 释放了多少空间。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。为了避免频繁的GC,建议将 -XX:MetaSpaceSize 设置为一个相对较高的值。

- **如何解决OOM?**
  - 要解决 OOM 异常或 heap space 的异常，一般的手段首先是通过内存映射工具对 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，也就是要分清楚到底是出现了内存泄漏还是内存溢出。
  - 如果是内存泄漏，可进一步通过工具查看泄漏对象到 GC Roots 的应用链。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他们的。掌握了泄漏对象的类型信息，以及 GC Roots 引用链的信息，就可以比较准确的定位出泄漏代码的位置。
  - 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（-Xms 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。

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 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 或是 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**

1. 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声明顺序。
2. 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：

1. 方法名称
2. 方法的返回类型
3. 方法参数的数量和类型（按顺序）
4. 方法的修饰符（public ,private, protected , static ,final, synchronized, native,abstract 的一个子集）
5. 方法的字节码（bytecodes)、操作数栈、局部变量表及大小（abstract 和 native方法除外）
6. 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

- **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，他们成为类数据在逻辑上的一部分。

- **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。

- **运行时常量池 vs 常量池**
  - 方法区中，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 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。
    docs.oracle.com/javase/...

- **为什么需要常量池?**

一个java源文件中类、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，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，应用到的结构会更多！这里就需要常量池了！

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。

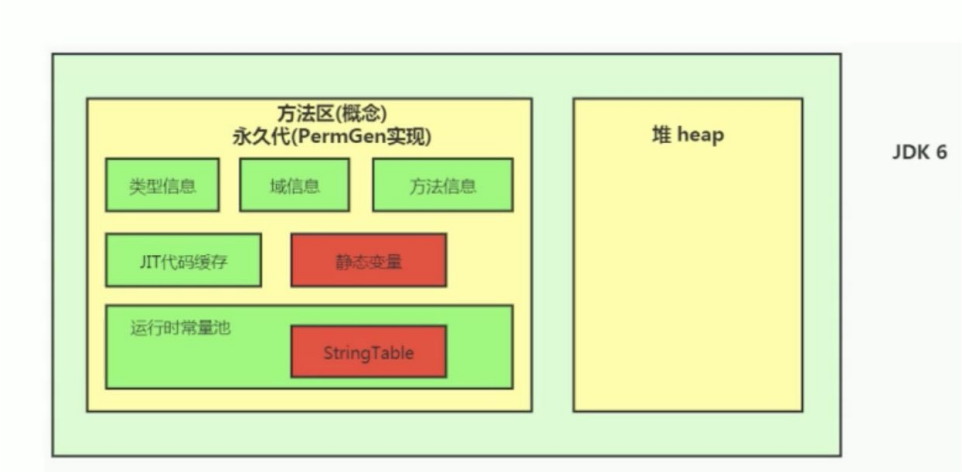
- **运行时常量池**
  - 运行时常量池（Runtime Constant Pool) 是方法区的一部分。
  - 常量池表（Constant Pool Table) 是 Class 文件的一部分，用于存放编译期生成的各种字面量与符号应用，这部分内容将在类加载后存放到方法区的运行时常量池中。
  - 在加载类和接口到虚拟机后，就会创建对应的运行时常量池。
  - JVM 为每个已加载到类型（类或接口）都维护了一个常量池。池中的数据项像数组项一样，是通过索引访问的。
  - 运行时常量池中包含多种不同的常量，包括编译器就已经明确的数值字面量，也包括到运行期解析后才能够获得的方法或者字段引用。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。

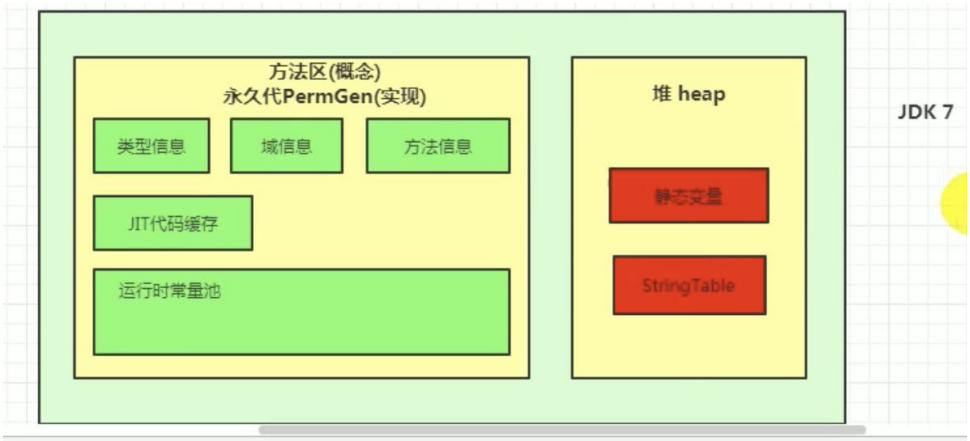
- 运行时常量池类似于传统编程语言中的符号表（symbol table),但是它所包含的数据却比符号表要更加丰富一些。
- 当创建类或者接口的运行时常量池时，如果构造运行时常量池所需的内存空间超过了方法区所提供的最大值，则 JVM 会抛 OutOfMemoryError 异常。

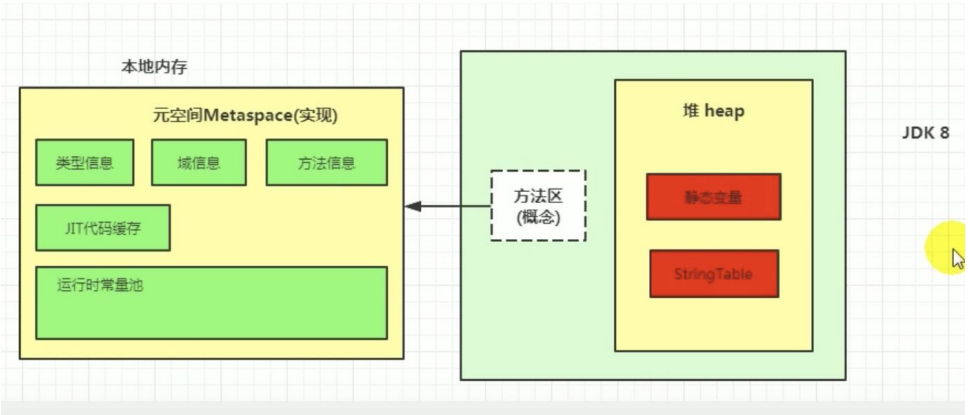
方法区的演进细节

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：

- jdk 1.8 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、静态变量仍在堆中







- **永久代为什么要被元空间替换?**

详细信息可参考文档：openjdk.java.net/jeps/1...

1. 随着 Java 8 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也消失了。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).
2. 由于类的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。
3. 这项改动是很有必要的，原因有：

- 对永久代进行调优是很困难的。

- **StringTable 为什么要调整?**

jdk7 中将 StringTable 放到了堆空间中。因为永久代的回收率很低，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 StringTable 回收率不高。而我们开发中会有大量的字符串被创建，回收率低，导致永久代内存不足。放到堆里，能及时回收内存。

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有确实是必要的。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。

方法区的收集主要回收两部分内容：常量池中的废弃的常量和不再使用的类型。

方法区内常量池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 Java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，包括下面三类常量：

1. 类和接口的权限定名
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，之前常量池中的常量没有被任何地方引用，就可以回收。回收废弃常量与回收 java 堆中的对象非常类似。

判定一个常量是否“废弃”还是相对简单的，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。需要同事满足下面三个条件：

1. 该类的所有实例都已经被回收了，也就是 java 堆中不存在该类及其任何派生子类的实例。
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如 OSGI、JSP 的重加载等，否则通常是很难达成的。
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 虚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”，而不是和对象一样，没有引用了就必然回收。关于是否要对类型进行回收，HotSpot 虚拟机提供了 -Xnoclassgcc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压力。

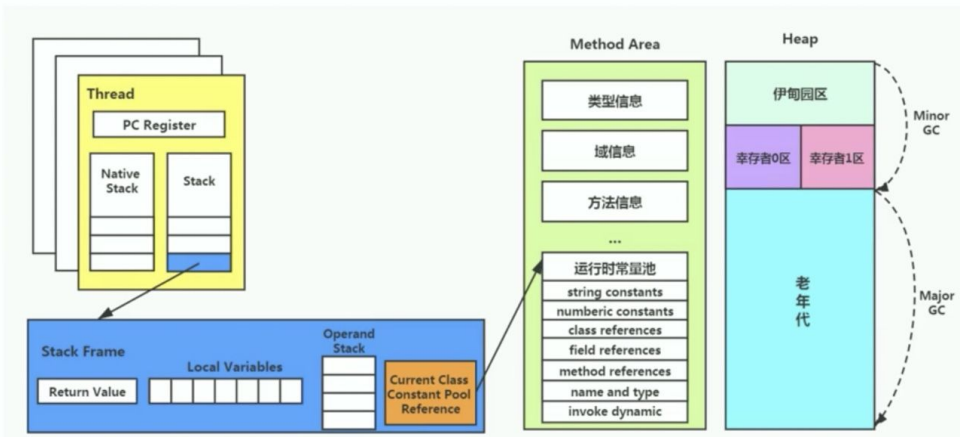
总结

zempty：JVM 系列分享- 类加载器篇
1 赞同 · 0 评论　文章

zempty：JVM 系列分享-虚拟机栈
1 赞同 · 0 评论 文章

zempty：JVM 系列分享 - 堆空间
3 赞同 · 0 评论 文章

JVM 运行时数据区： PC 寄存器，jvm 虚拟机栈，本地堆，方法区大致如下图：



编辑于 2020-08-09 23:04

javase Java 虚拟机（JVM） Java

文章被以下专栏收录

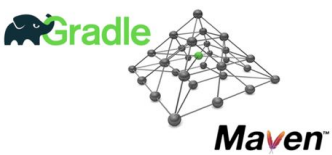
zempty 笔记
码农，终身学习者

推荐阅读

Java包管理的那些事2——神奇字节码在哪里

上一篇文章链接： Java包管理的那些事1——IDE背后发生了什么上回我们只说了两个重点： 强大的IDE屏蔽了JVM的细节，给开发者造成了一种错觉：写好Java代码之后直接就可以在JVM中运行。IDE偷...

blindpirate



Java包管理的那些事1——IDE背后发生了什么

blindpirate

Java运
点整理

1. Java语
面向对象
由 javac
字节码文
2. Java 编
见： Tha

Thas

写下你的评论...

晚熟的人 2020-08-08

你好为啥大量的jsp就会导致方法区溢出啊

1

zempty (作者) 回复 **晚熟的人** 2020-08-09

您好，jsp 实质上是 java , jsp 是要翻译成 java 进行处理的， 一个 .jsp 的文件就对应一个 .java 的文件，大量的 jsp 也就意味着大量的 java 文件，方法区存储着 java 类的信息，jdk 1.8 以后方法区存储在本地内存之中（大小可调），如果存在大量的 java 文件，超出容量范围就会 OOM（java.lang.OutOfMemoryError:Metaspace）。

赞

george 回复 **zempty** (作者) 2021-06-26

一个jsp本质上是一个servlet对吧?

赞

来自China的神秘人 2021-09-24

请教2个问题

1.类数据到底指哪些信息啊？类的静态变量，静态方法，普通方法，构造函数这些吗？还包括什么吗?

2.如果类数据包括静态方法，普通方法，也就是说类加载的时候普通方法和静态方法都被加载到方法区内，那为什么 static synchronized void test()获取的是类锁，synchronized void test()这种获取的就是实例锁。按道理说两种方法都在类加载时已经加载到方法区了，为啥只有只有静态方法获取的是类锁。

我疑惑的点就是方法被加载到方法区，是怎么个意思。变量被加载我还能理解(申请个内存，地址啥的)，那方法被加载后是个什么状态呢

赞

RONIN 回复 **来自China的神秘人** 01-08

我的理解是，类数据就是标记这个类的类型是什么类所处的位置还有它的父类的位置的信息等。至于方法的话存储的是方法返回的是什么类型、访问的权限信息、字节码中就是存储一些栈中的局部变量表中使用的大小、操作数栈的大小，还有一些符号引用，用于引用常量池中的符号作为直接引用找到对应的位置。有啥疑惑或者不对的地方一起讨论🤔

赞

RONIN 回复 **来自China的神秘人** 01-08

堆就会去存储一些对象的实例信息

赞

Ceeeeeeeeeeeeeb 2021-09-23

所以实例方法也是存储在方法区当中的吗?

赞

风中的雪糕 2020-09-02

以前6的时候Spring启动时也有可能导致内存溢出

赞