本课时我们主要讲解 JVM 的内存划分以及栈上的执行过程。这块内容在面试中主要涉及以下这 3 个面试题：

- JVM 是如何进行内存区域划分的？
- JVM 如何高效进行内存管理？
- 为什么需要有元空间，它又涉及什么问题？

带着这 3 个问题，我们开始今天的学习，关于内存划分的知识我希望在本课时你能够理解就可以，不需要死记硬背，因为在后面的课时我们会经常使用到本课时学习的内容，也会结合工作中的场景具体问题具体分析，这样你可以对 JVM 的内存获得更深刻的认识。

首先，第一个问题：**JVM的内存区域是怎么高效划分的？**这也是一个高频的面试题。很多同学可能通过死记硬背的方式来应对这个问题，这样不仅对知识没有融会贯通在面试中还很容易忘记答案。

为什么要问到 JVM 的内存区域划分呢？因为 Java 引以为豪的就是它的自动内存管理机制。相比于 C++的手动内存管理、复杂难以理解的指针等，Java 程序写起来就方便的多。

然而这种呼之即来挥之即去的内存申请和释放方式，自然也有它的代价。为了管理这些快速的内存申请释放操作，就必须引入一个池子来延迟这些内存区域的回收操作。

我们常说的内存回收，就是针对这个池子的操作。我们把上面说的这个池子，叫作堆，可以暂时把它看成一个整体。

JVM 内存布局

程序想要运行，就需要数据。有了数据，就需要在内存上存储。那你可以回想一下，我们的 C++ 程序是怎么运行的？是不是也是这样？

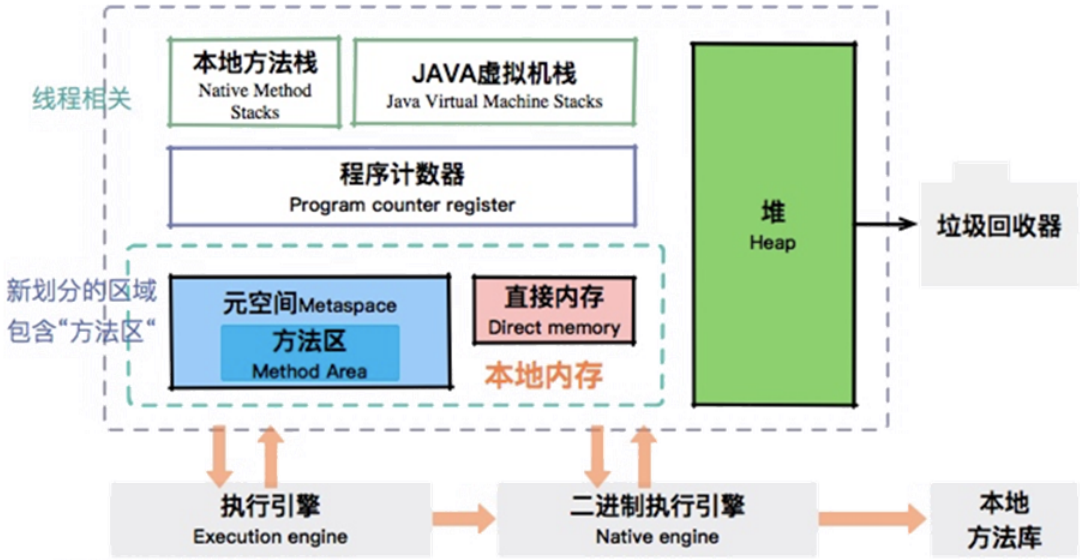
Java 程序的数据结构是非常丰富的。其中的内容，举一些例子：

- 静态成员变量
- 动态成员变量
- 区域变量
- 短小紧凑的对象声明
- 庞大复杂的内存申请

这么多不同的数据结构，到底是在什么地方存储的，它们之间又是怎么进行交互的呢？是不是经常在面试的时候被问到这些问题？

我们先看一下 JVM 的内存布局。随着 Java 的发展，内存布局一直在调整之中。比如，Java 8 及之后的版本，彻底移除了持久代，而使用 Metaspace 来进行替代。这也表示着 -XX:PermSize 和 -XX:MaxPermSize 等参数调优，已经没有了意义。但大体上，比较重要的内存区域是固定的。

JVM内存区域划分

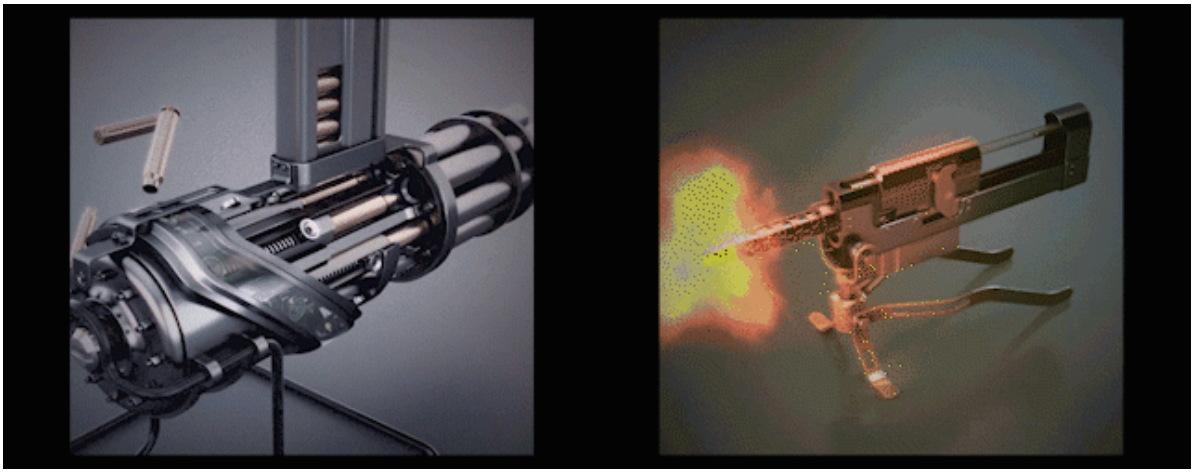


JVM 内存区域划分如图所示，从图中我们可以看出：

- JVM 堆中的数据是共享的，是占用内存最大的一块区域。
- 可以执行字节码的模块叫作执行引擎。
- 执行引擎在线程切换时怎么恢复？依靠的就是程序计数器。
- JVM 的内存划分与多线程是息息相关的。像我们程序中运行时用到的栈，以及本地方法栈，它们的维度都是线程。
- 本地内存包含元数据区和一些直接内存。

一般情况下，只要你能答出上面这些主要的区域，面试官都会满意的点头。但如果深挖下去，可能就有同学就比较头疼了。下面我们就详细看下这个过程。

虚拟机栈



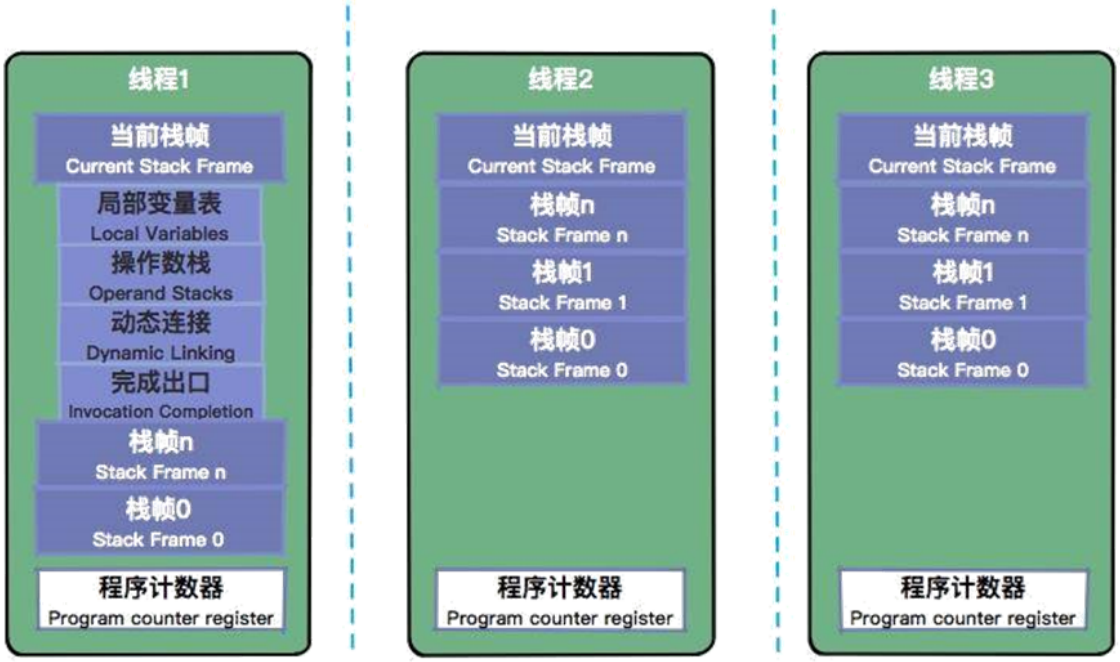
栈是什么样的数据结构？你可以想象一下子弹上膛的这个过程，后进的子弹最先射出，最上面的子弹就相当于栈顶。

我们在上面提到，Java 虚拟机栈是基于线程的。哪怕你只有一个 main() 方法，也是以线程的方式运行的。在线程的生命周期中，参与计算的数据会频繁地入栈和出栈，栈的生命周期是和线程一样的。

栈里的每条数据，就是栈帧。在每个 Java 方法被调用的时候，都会创建一个栈帧，并入栈。一旦完成相应的调用，则出栈。所有的栈帧都出栈后，线程也就结束了。每个栈帧，都包含四个区域：

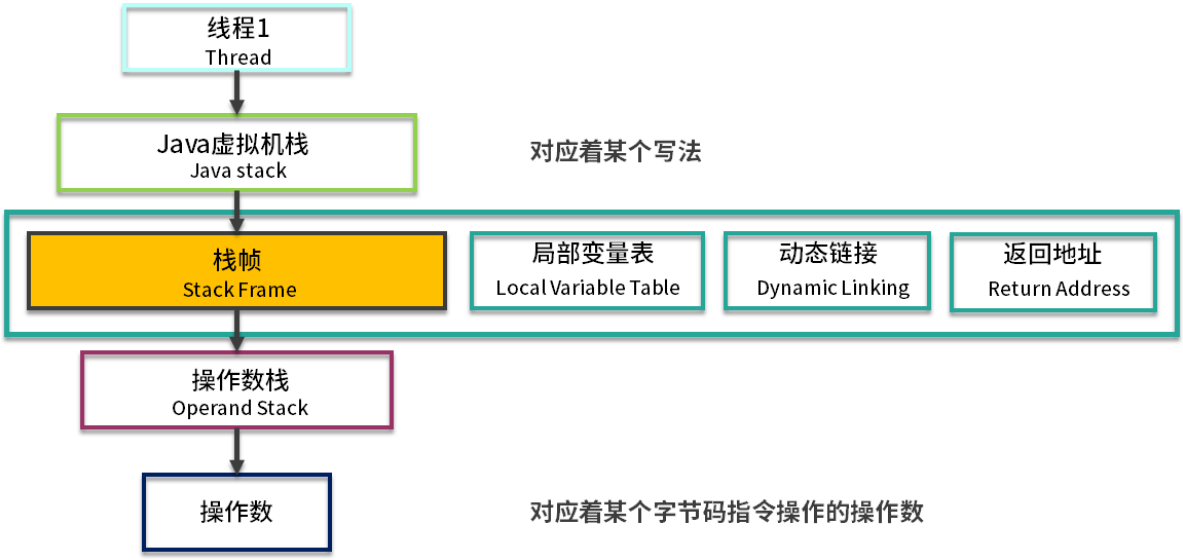
- 局部变量表
- 操作数栈
- 动态连接
- 返回地址

我们的应用程序，就是在不断操作这些内存空间中完成的。



本地方法栈是和虚拟机栈非常相似的一个区域，它服务的对象是 native 方法。你甚至可以认为虚拟机栈和本地方法栈是同一个区域，这并不影响我们对 JVM 的了解。

这里有一个比较特殊的数据类型叫作 returnAddress。因为这种类型只存在于字节码层面，所以我们平常打交道的比较少。对于 JVM 来说，程序就是存储在方法区的字节码指令，而 returnAddress 类型的值就是指向特定指令内存地址的指针。



这部分有两个比较有意思的内容，面试中说出来会让面试官眼前一亮。

- 这里有一个两层的栈。第一层是栈帧，对应着方法；第二层是方法的执行，对应着操作数。注意千万不要搞混了。
- 你可以看到，所有的字节码指令，其实都会抽象成对栈的入栈出栈操作。执行引擎只需要傻瓜式的按顺序执行，就可以保证它的正确性。

这一点很神奇，也是基础。我们接下来从线程角度看一下里面的内容。

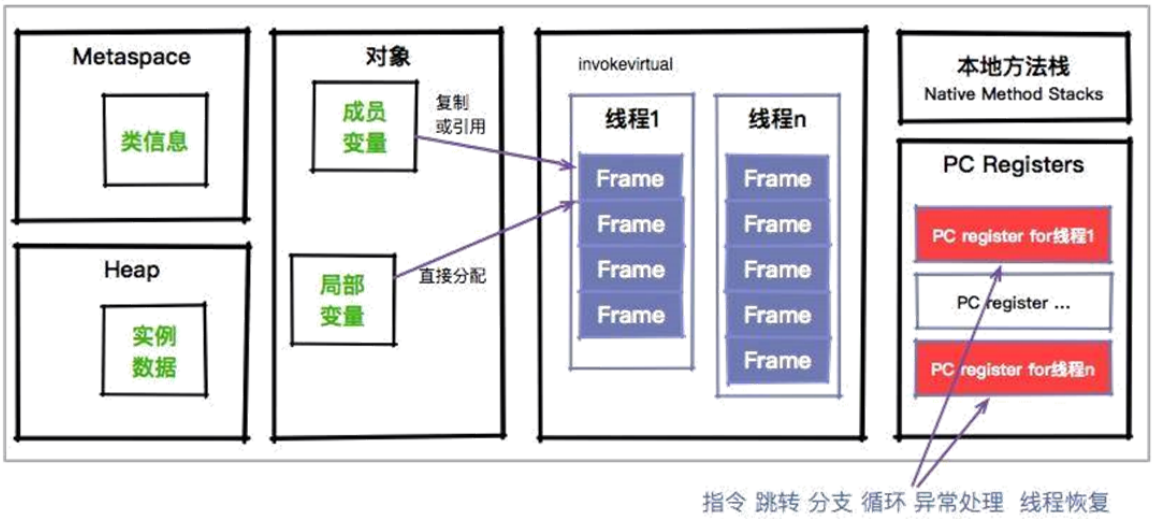
程序计数器

那么你设想一下，如果我们的程序在线程之间进行切换，凭什么能够知道这个线程已经执行到什么地方呢？

既然是线程，就代表它在获取 CPU 时间片上，是不可预知的，需要有一个地方，对线程正在运行的点位进行缓冲记录，以便在获取 CPU 时间片时能够快速恢复。

就好比你停下手中的工作，倒了杯茶，然后如何继续之前的工作？

程序计数器是一块较小的内存空间，它的作用可以看作是当前线程所执行的字节码的行号指示器。这里面存的，就是当前线程执行的进度。下面这张图，能够加深大家对这个过程的理解。

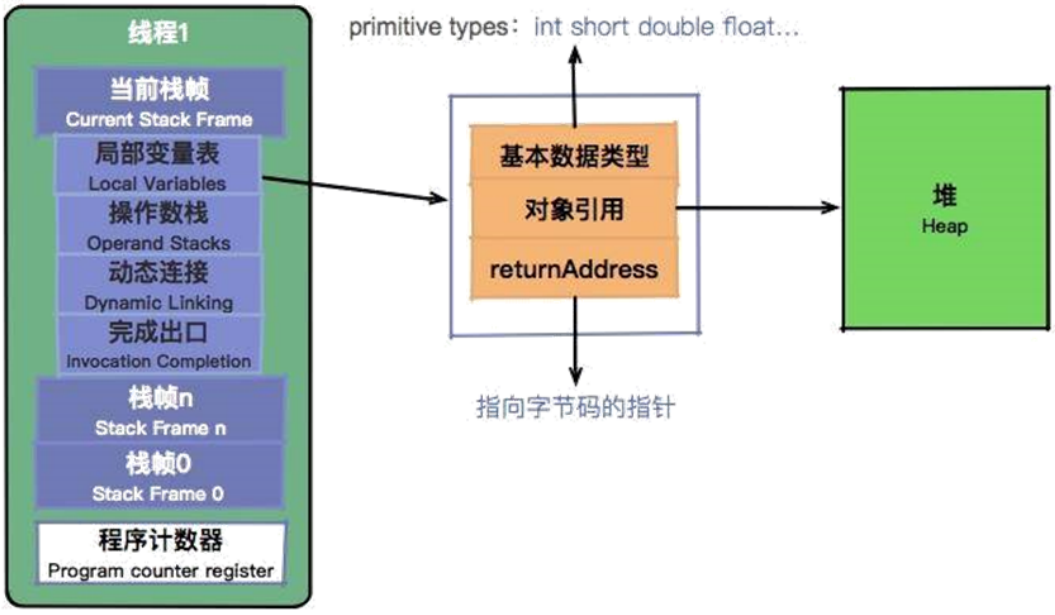


可以看到，程序计数器也是因为线程而产生的，与虚拟机栈配合完成计算操作。程序计数器还存储了当前正在运行的流程，包括正在执行的指令、跳转、分支、循环、异常处理等。

我们可以看一下程序计数器里面的具体内容。下面这张图，就是使用 javap 命令输出的字节码。大家可以看到在每个 opcode 前面，都有一个序号。就是图中红框中的偏移地址，你可以认为它们是程序计数器的内容。

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=6, args_size=1
       0: new           #3                  // class A
       3: dup
       4: invokespecial #4                  // Method "<init>":()V
       7: astore_1
       8: ldc2_w        #5                  // long 654321
      11: lstore_2
      12: aload_1
      13: lload_2
      14: invokevirtual #7                  // Method fig:(J)J
      17: lstore        4
      19: getstatic     #8                  // Field java/lang/System.out:Ljava/io/PrintStream;
      22: lload         4
      24: invokevirtual #9                  // Method java/io/PrintStream.println:(J)V
      27: return
    LineNumberTable:
      line 9: 0
      line 10: 8
      line 12: 12
      line 14: 19
      line 15: 27
```

堆



堆是 JVM 上最大的内存区域，我们申请的几乎所有的对象，都是在这里存储的。我们常说的垃圾回收，操作的对象就是堆。

堆空间一般是程序启动时，就申请了，但是并不一定会全部使用。

随着对象的频繁创建，堆空间占用的越来越多，就需要不定期的对不再使用的对象进行回收。这个在 Java 中，就叫作 GC（Garbage Collection）。

由于对象的大小不一，在长时间运行后，堆空间会被许多细小的碎片占满，造成空间浪费。所以，仅仅销毁对象是不够的，还需要堆空间整理。这个过程非常的复杂，我们会在后面有专门的课时进行介绍。

那一个对象创建的时候，到底是在堆上分配，还是在栈上分配呢？这和两个方面有关：对象的类型和在 Java 类中存在的位置。

Java 的对象可以分为基本数据类型和普通对象。

对于普通对象来说，JVM 会首先在堆上创建对象，然后在其他地方使用的其实是它的引用。比如，把这个引用保存在虚拟机栈的局部变量表中。

对于基本数据类型来说（byte、short、int、long、float、double、char)，有两种情况。

我们上面提到，每个线程拥有一个虚拟机栈。当你在方法体内声明了基本数据类型的对象，它就会在栈上直接分配。其他情况，都是在堆上分配。

注意，像 int[] 数组这样的内容，是在堆上分配的。数组并不是基本数据类型。

这就是 JVM 的基本的内存分配策略。而堆是所有线程共享的，如果是多个线程访问，会涉及数据同步问题。这同样是个大话题，我们在这里先留下一个悬念。

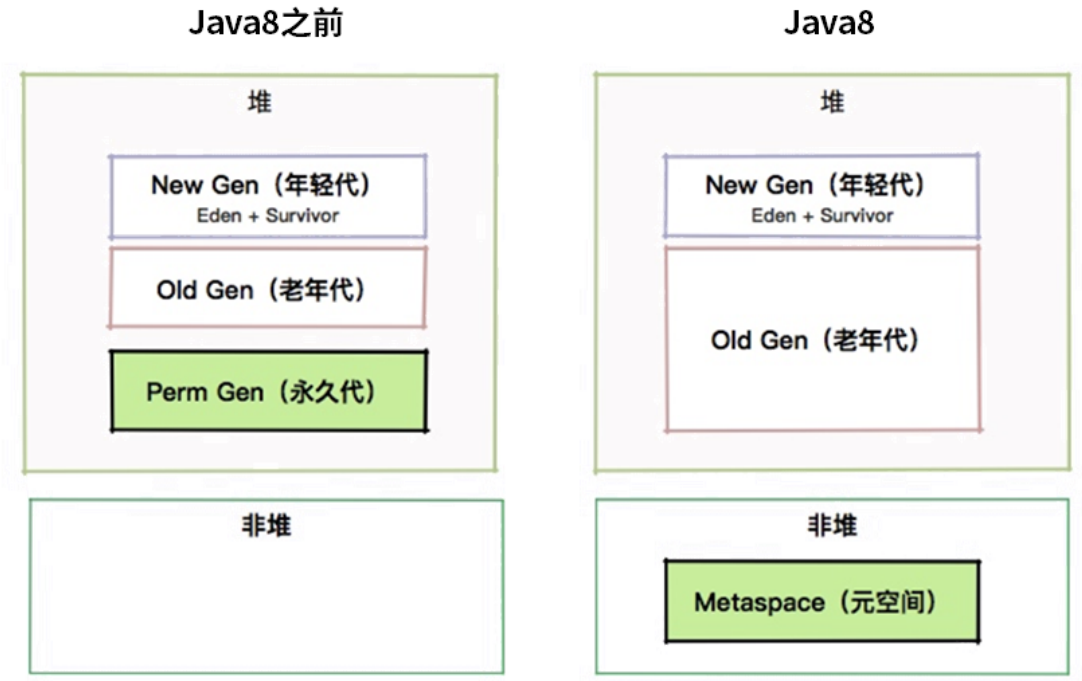
元空间

关于元空间，我们还是以一个非常高频的面试题开始：“为什么有 Metaspace 区域？它有什么问题？”

说到这里，你应该回想一下类与对象的区别。对象是一个活生生的个体，可以参与到程序的运行中；类更像是一个模版，定义了一系列属性和操作。那么你可以设想一下。我们前面生成的 A.class，是放在 JVM 的哪个区域的？

想要问答这个问题，就不得不提下 Java 的历史。在 Java 8 之前，这些类的信息是放在一个叫 Perm 区的内存里面的。更早版本，甚至 String.intern 相关的运行时常量池也放在这里。这个区域有大小限制，很容易造成 JVM 内存溢出，从而造成 JVM 崩溃。

Perm 区在 Java 8 中已经被彻底废除，取而代之的是 Metaspace。原来的 Perm 区是在堆上的，现在的元空间是在非堆上的，这是背景。关于它们的对比，可以看下这张图。



然后，元空间的好处也是它的坏处。使用非堆可以使用操作系统的内存，JVM 不会再出现方法区的内存溢出；但是，无限制的使用会造成操作系统的死亡。所以，一般也会使用参数 -XX:MaxMetaspaceSize 来控制大小。

方法区，作为一个概念，依然存在。它的物理存储的容器，就是 Metaspace。我们将在后面的课时中，再次遇到它。现在，你只需要了解到，这个区域存储的内容，包括：类的信息、常量池、方法数据、方法代码就可以了。

小结

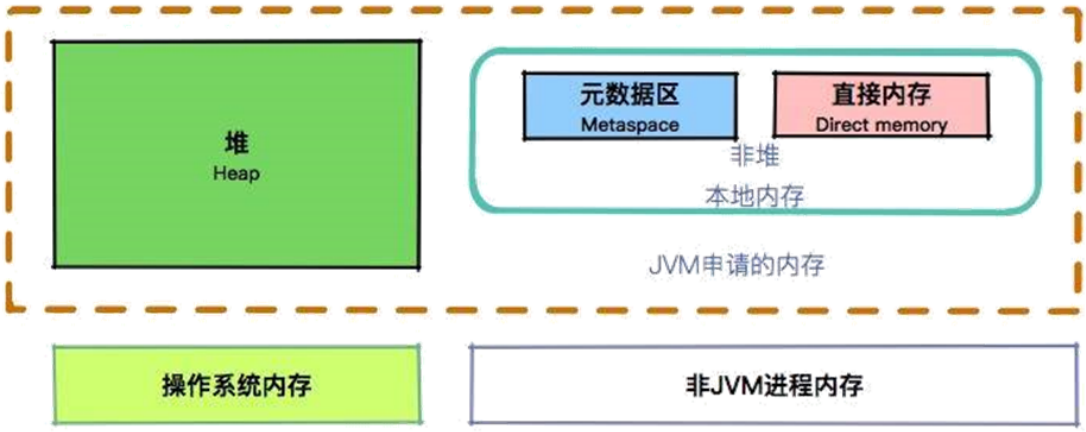
好了，到这里本课时的基本内容就讲完了，针对这块的内容在面试中还经常会遇到下面这两个问题。

- 我们常说的字符串常量，存放在哪呢？

由于常量池，在 Java 7 之后，放到了堆中，我们创建的字符串，将会在堆上分配。

- 堆、非堆、本地内存，有什么关系？

关于它们的关系，我们可以看一张图。在我的感觉里，堆是软绵绵的，松散而有弹性；而非堆是冰冷生硬的，内存非常紧凑。



大家都知道，JVM 在运行时，会从操作系统申请大块的堆内内存，进行数据的存储。但是，堆外内存也就是申请后操作系统剩余的内存，也会有部分受到 JVM 的控制。比较典型的就是一些 native 关键词修饰的方法，以及对内存的申请和处理。

在 Linux 机器上，使用 top 或者 ps 命令，在大多数情况下，能够看到 RSS 段（实际的内存占用），是大于给 JVM 分配的堆内存的。

如果你申请了一台系统内存为 2GB 的主机，可能 JVM 能用的就只有 1GB，这便是一个限制。

总结

JVM 的运行时区域是栈，而存储区域是堆。很多变量，其实在编译期就已经固定了。.class 文件的字节码，由于助记符的作用，理解起来并不是那么吃力，我们将在课程最后几个课时，从字节码层面看一下多线程的特性。

JVM 的运行时特性，以及字节码，是比较偏底层的知识。本课时属于初步介绍，有些部分并未深入讲解。希望你应该能够在脑海里建立一个 Java 程序怎么运行的概念，以便我们在后面的课时中，提到相应的内存区域时，有个整体的印象。

精选评论

****5241：**

> 您好，文中有个地方没理解，您说“由于常量池，在 Java 7 之后，放到了堆中，我们创建的字符串，将会在堆上分配”。
>
> 但是您上文也说了，JAVA8开始，metasapce是非堆区域，而且文中也提到了该区域包含的内容是类的信息、常量池、方法数据、方法代码。
> 那么java字符串常量，就不应该在堆上创建了啊。
> 劳烦您解释下，添麻烦了，谢谢。

讲师回复：

> 你好，JVM中存在多个常量池。把第一个改成字符串常量池就比较好理解了。
> 1、字符串常量池，已经移动到堆上（jdk8之前是perm区），也就是执行intern方法后存的地方。
> 2、类文件常量池，constant_pool，是每个类每个接口所拥有的，第四节字节码中“#n”的那些都是。这部分数据在方法区，也就是元数据区。而运行时常量池是在类加载后的一个内存区域，它们都在元空间。

**宾：

> 大家都知道，JVM 在运行时，会从操作系统申请大块的堆内内存，进行数据的存储。但是，堆外内存也就是申请后操作系统剩余的内存，也会有部分受到 JVM 的控制。比较典型的就是一些 native 关键词修饰的方法，以及对内存的申请和处理
>
> 这句话是jvm去申请了一块操作系统的堆内内存，那图上怎么jvm申请的内存包括了堆内存和非堆内存，有点疑惑。就是jvm申请的内存其实有这两个？

讲师回复：

> 可以这样理解：
> 操作系统有8G。-Xmx分配了4G（堆内内存），Metaspace使用了256M（堆外内存）
> 剩下的 8G-4G-256M ，就是操作系统剩下的本地内存。具体有没有可能变成堆外内存，要看情况。
> 比如：
> （1）netty的direct buffer使用了额外的120MB内存，那么现在JVM占用的堆外内存就有256M+120M
> （2）使用了jni或者jna，直接申请了内存2GB，那么现在JVM占用的堆外内存就有256M+120M+2GB
> （3）网络socket连接等，占用了操作系统的50MB内存

这个时候，留给操作系统的就只剩下了：8GB-4GB-256M-120M-2GB-50M。具体“堆和堆外”一共用了多少，可以top命令，看RSS段。

**山：

> 您好，文中说除了基本类型，其他都是在堆上分配的。
>
> 之前粗略在哪个地方看到过jvm会判断对象是否存在线程逃逸，如果不存在就直接在栈上分配对象。这种在栈上创建对象的情况是怎样的呢。

讲师回复：

> 你说的没错，这种情况在第6小节已聊到了。不过它是分层编译的优化手段，所以我们在后面JIT小节还会碰到它。

*吴：

> 老师您好，这里对于虚拟机栈有点不太理解，原文：这里有一个两层的栈。第一层是栈帧，对应着方法；第二层是方法的执行，对应着操作数栈；这里是说栈帧是说具体的Java方法，而真正的调用，是在栈帧中里面还建了一个操作数栈对吗？为什么要这么做呀？辛苦老师指导。
> 😄😄😄

讲师回复：

> 线程方法栈（栈）->栈帧（元素）=>方法级别的操作。
> 栈帧里的操作数栈（栈）->操作数（元素）=> 字节码指令级的操作。
> 主管的功能不同，层次也不同。

****峰：**

> 内容很多，每一篇文章都需要反复观看。

编辑回复：

> 温故而知新，相信你每一次都会得到满满的收获

****青：**

> 线程方法对应着栈帧，方法运行对应着对操作数栈的操作。虚拟机栈保存着多个栈帧，每个栈帧包含局部变量表，操作数栈，方法返回地址

****飞：**

> 老师，jvm在运行过程中栈的分配是在那个区域，每个栈的大小可以使用xss设置，但是这个栈是在那个地方分配的？不在堆，不在堆外内存，具体在哪里

讲师回复：

> 属于堆外内存。通过top命令查看RES列包含其中。使用pmap命令可获取详细分布。

堆外内存包含：（1）元空间 （2）CodeCache （3）本地内存；本地内存包括：（1）网络内存 （2）线程内存 （3）JNI内存 （4）直接内存[direct memory]。

这里你就可以认为是线程内存。因为JVM本质上就是一个c语言程序。

****0742：**

> 老师您好，对于操作数栈 我可不可以理解为 该方法的一条条指令。程序计数器是用来标识，该指令执行的位置。

***怀：**

> 老师，能不能详细说一下操作数栈，动态连接，完成出口这三个区域在方法执行的过程中起到了什么作用。returnAddress 类型的值就是指向特定指令内存地址的指针，这句话的意思我可以理解成指向了调用了上一个栈帧形成的字节码指令吗

讲师回复：

> JVM的运行是基于栈的，所以汇编、C语言等基于栈的特征都是相似的。当方法执行完毕，比如递归、嵌套等，程序需要知道下一条指令的执行地址。这个地址就可以将执行引擎指向到具体的某条指令，继续运行。

****用户1491：**

> 老师您好，字符串常量池，已经移动到堆上（jdk8之前是perm区），
>
> jdk8之前 的 perm区 不就是Perm Gen所在的位置吗？图上Perm Gen在堆内部，那不就是【字符串常量池一直位于堆上吗？】难道perm区不在堆内部吗？

讲师回复：

> 你这么说也没错。我觉得你只需要记忆jdk8之后的就可以了，因为这个常量池经过了多次变更。
> <jdk7: 处于perm区，属于堆，但空间单独管理
> =jdk7 处于堆，此堆非彼堆，空间上限也不同 :)
> jdk8: perm区没了，它又不在元空间，也只能说堆了

****文：**

> 对于基本类型的包装类也是在栈上分配吗？

讲师回复：

> 包装类属于引用类型，它们不属于基本类型，所以是在堆上分配的。

****飞：**

> "程序计数器还存储了当前正在运行的流程，包括正在执行的指令、跳转、分支、循环、异常处理等。"这句话给我带来了理解上的困扰，不明白程序计数器里到底都存储了些什么。记录了整个运行流程？占用内存又非常小，在遇到非常复杂的方法时如何做到不会内存溢出呢？在查阅了其他的资料之后，看到的描述是“分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成”。或许这句话更好理解一些

讲师回复：

> 程序计数器就是个指针，指明了下一条要执行的指令。跳转、分钟、异常、循环等，都会修改这个指针，它是相当于物理PC（寄存器）的一种模拟。

***靖：**

> 方法区就是元空间吗，这两个概念是相同的吗

讲师回复：

> 元空间包含方法区，是包含关系。概念不等同。

LeonardoEzio：

> 静态方法调用会产生栈帧吗？

讲师回复：

> 所有的方法调用都会产生栈帧，和是否静态，是否是native无关

***风：**

> ”静态变量在堆上。只有在执行方法的栈帧中，才存在是否是堆和栈的问题。“老师，静态变量是不是与类对象一起都在方法区？JDK8以后，方法区是不是在元空间？为什么说是在堆上？

讲师回复：

> 引用和数据是两回事。举一个最浅显的例子：加入你声明了一个Map类型的静态变量，在程序运行时，向里面存放了1w条记录。显然，这些记录是存放在Map中的，但它们的实际存储地址却是在堆上。

**东：

> 周志明的书里说：在jdk1.7之后，字符串常量池和运行时常亮池都移动到了堆中，而不是元空间吧（我看到评论第一条您说是在元空间中），元空间在jdk1.8之后取代了永久代，用以存放类的元数据，它存储的位置是本地内存。

讲师回复：

> 字符串常量池在堆上是没有异议了，争论在运行时常量池上，现在社区也没有统一的答案。每一个类都有一个运行时常量池，根据Java规范，它是属于方法区的：https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html#jvms-5.1，但实际上也有一部分数据通过指针指向到堆上。可以把它认为是一个混合体，但在逻辑上，是属于方法区的。

**波：

> String Pool 这个老师可以讲述一下吗?

讲师回复：

> JDK8的字符串常量池在堆中，目的是通过复用减少内存的开销，增加访问速度。可通过JVM参数-XX:+PrintStringTableStatistics查看具体使用情况。更多扩展请参考String的intern()方法。

**辉：

> 类的成员变量 static int a = 2; 这个a是在栈在还是堆上?看文章写到基本数据类型除了在方法体申明的,其余都在堆上.那么a在堆上?但是看到老师回复静态变量。基本类型的直接在方法区 所以冲突了?

讲师回复：

> 静态变量在堆上。只有在执行方法的栈帧中，才存在是否是堆和栈的问题。

**的小单同学：

> 老师，操作系统分配给每个进程的内存是有限制的，比如给jvm进程分配了2G，那您说的这些堆外内存可以申请的内存空间会受这个进程内存的限制吗，还是说只有一部分会受到，望老师看到能解答一下，非常感谢！！！！

讲师回复：

> 你分配的2G，只是堆内内存；堆外内存不受限制。具体参见下面这篇文章：http://xjjdog.cn/1590651201157 8.html

**6862：

> 老是你好，希望得到回到，JDK1.7之前Hotspot针对方法区的实现也就是永久代和堆共享一片物理空间连续的内存是吗？-Xmx代表的堆最大内存是不是也包含了永久代的大小，他们只是逻辑上隔离，也即是非堆的说法，而JDK1.8中属于完全隔离，因为元空间使用的是本地内存

讲师回复：

> 对1.8的理解是对的，1.7的理解不完全对，永久代的MaxPermSize和堆空间的Xmx，控制的是不同的区域，最大堆大小和永久代大小的设置是分开的。

**里的梦幻：

> 老师你好，全局变量存放在哪里?

讲师回复：

> 静态变量。基本类型的直接在方法区，但是其他对象则是在堆上，比如你申请了一个5MB的数组

****3430：**

> 请问老师非堆就是方法区吗

讲师回复：

> 堆外内存包含：（1）元空间 （2）CodeCache （3）本地内存；本地内存包括：（1）网络内存 （2）线程内存 （3）JNI内存 （4）直接内存[direct memory]。更多可以搜我之前的一篇文章：一图解千愁，jvm内存从来没有这么简单过

****宇：**

> 请问虚拟机栈的数据一定是线程安全的吗？

讲师回复：

> 答案是肯定的，在同一虚拟机栈的数据不需要做同步。建议了解一下“线程封闭”这个概念。主要提到了Ad-hoc、栈封闭、ThreadLocal等。

****4003：**

> 1、字符串常量池，已经移动到堆上（jdk8之前是perm区），也就是执行intern方法后存的地方。？
>
> jdk1.7字符串常量池是在永久代吗？我看周志明的书上写的“在目前已经发布的JDK1.7的HotSpot中，已经把原来放在永久代的字符串常量池移除。"请问能解释一下吗？我有些疑惑

讲师回复：

> 你是在疑惑1.7还是1.8移动的么？常量池确实是1.7移动的。永久带是1.8废掉的。

****用户1271：**

> 您好，上文中提到jdk8之后的版本，废弃了perm gen 取而代之的是产生了非堆区matespace，这个区是方法区的物理存储形式，常量池也存在其中，，下文又说常量池是在堆中创建，请问常量池的存储形式具体是在哪个区，还是我的理解有误呢，

讲师回复：

> 你好，JVM中存在多个常量池。把第一个改成字符串常量池就比较好理解了。
> 1、字符串常量池，已经移动到堆上（jdk8之前是perm区），也就是执行intern方法后存的地方。
> 2、类文件常量池，con...

****绪：**

> 非堆指的就是方法区吧，看您图中在堆中还画了一个永久带（Perm），非堆和永久带是不是有点重复？在我理解是一回事吧？

讲师回复：

> 你概念好像搞混了。图中有两种情况，Java8之前的Perm是属于堆的，包括里面的方法区；课程默认是Java8及其以后，没有Perm，此时方法区是在metaspace非堆。另外，非堆也不仅仅只是方法区，更详细的排查可以参考第13课时。

****富：**

> 你好，运行时常量池和字符串常量池什么区别？string常量池是在堆中还是元空间？

讲师回复：

> 参见其他回答：JVM中存在多个常量池。1、字符串常量池，已经移动到堆上（jdk8之前是perm区），也就是执行intern方法后存的地方。2、类文件常量池，constant_pool，是每个类每个接口所拥有的，第四节字节码中“#n”的那些都是。这部分数据在方法区，也就是元数据区。而运行时常量池是在类加载后的一个内存区域，它们都在元空间。

****民：**

> 还想请教老师一个问题，就是我在main主线程中新建了两个线程，其中一个线程发生了oom，但是我发现发生oom后，有问题的那个线程把占用的内存都释放了，其他的线程也没受到影响，继续运行。
> 我的疑惑是：我们分配的堆空间不是对整个进程有效吗？为什么其中一个线程发生了oom，内存会释放掉呢？且不影响其他线程呢？

编辑回复：

> 你说的这种情况确实存在，少量线程模拟下可以复现，因为GC线程和用户线程是并行执行的，线程溢出的空间能够被及时释放。但是系统一般都是高速运行，有很多线程在运行和并行申请内存，在实际中很难复现，都是雪崩式直接退出。还有很多情况是GC线程疯狂运转，直到系统异常，也就是我们后面说的GC线程占用cpu 100%

****民：**

> 老师请教一个问题，应该是道面试题：
>
> 创建一个100M的数组，程序OOM，但是分析日志发现 堆内存还大于100M，造成这个问题有哪些情况？
> 这个问题和咱们这篇文章的最后有点像，老师说用top命令观察 RSS段（实际占用内存）一般是大于分配的堆内存的，文中说“堆外内存也就是申请后操作系统剩余的内存，也会有部分受到 JVM 的控制。比较典型的就是一些 native 关键词修饰的方法，以及对内存的申请和处理”，这块没怎么看明白，自己也解释不通上面的那道题，希望老师帮忙指点下

讲师回复：

> 这个OOM描述不太清楚，OOM在会发生在很多区域。第101、13小节具体讲解了堆外内存的排查，希望对你有所帮助。另外，一些JVM配置参数也会造成此种状况，比如CMS的预留空间大小。