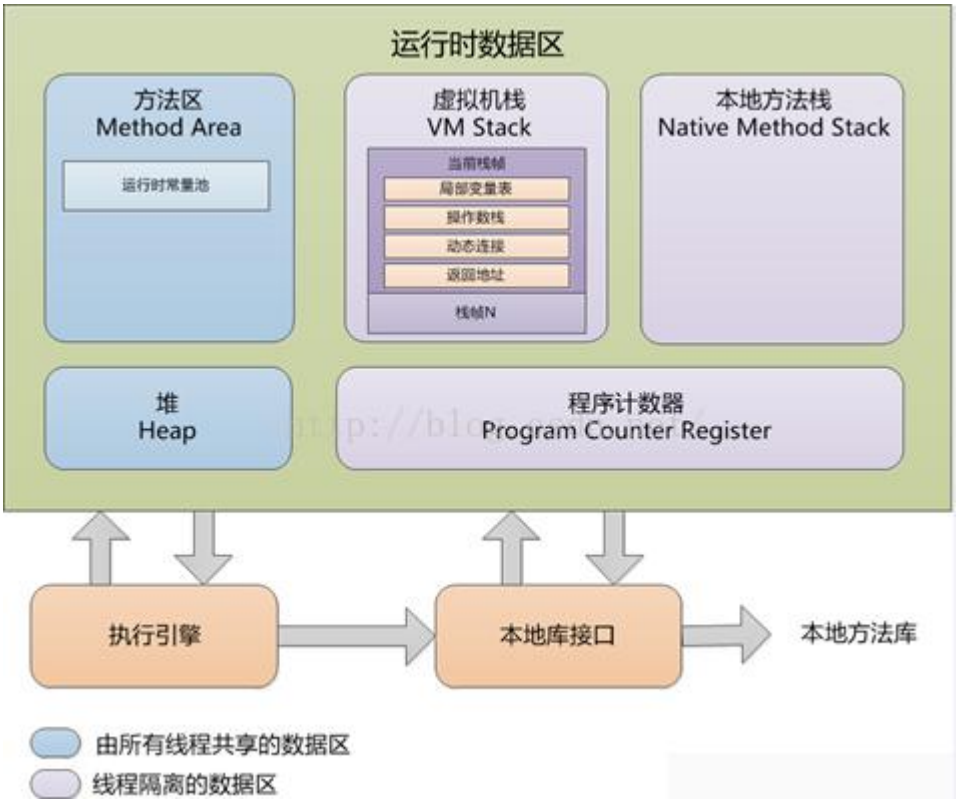
JVM 定义了若干个程序执行期间使用的数据区域。这个区域里的一些数据在 JVM 启动的时候创建，在 JVM 退出的时候销毁。而其他的数据依赖于每一个线程，在线程创建时创建，在线程退出时销毁。



程序计数器

程序计数器是一块较小的内存空间，可以看作是当前线程所执行的字节码的行号指示器。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。

由于 Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间的计数器互不影响，独立存储，我们称这类内存区域为**“线程私有”**的内存。

如果线程正在执行的是一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是 Natvie 方法，这个计数器值则为空（Undefined）。

此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。

虚拟机栈

线程私有，它的生命周期与线程相同。虚拟机栈描述的是 Java 方法执行的内存模型：**每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息**。

动画是由一帧一帧图片连续切换结果的结果而产生的，其实虚拟机的运行和动画也类似，每个在虚拟机中运行的程序也是由许多的帧的切换产生的结果，只是这些帧里面存放的是方法的局部变量，操作数栈，动态链接，方法返回地址和一些额外的附加信息组成。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

对于执行引擎来说，活动线程中，只有栈顶的栈帧是有效的，称为**当前栈帧**，这个栈帧所关联的方法称为**当前方法**。**执行引擎所运行的所有字节码指令都只针对当前栈帧进行操作**。

局部变量表

局部变量表是一组变量值存储空间，**用于存放方法参数和方法内部定义的局部变量**。在 Java 程序被编译成 Class 文件时，就在方法的 Code 属性的 max_locals 数据项中确定了该方法所需要分配的最大局部变量表的容量。

局部变量表的容量以**变量槽**（Slot）为最小单位，32 位虚拟机中一个 Slot 可以存放一个 32 位以内的数据类型（boolean、byte、char、short、int、float、reference 和 returnAddress 八种）。

reference 类型虚拟机规范没有明确说明它的长度，但一般来说，虚拟机实现至少都应当能从此引用中直接或者间接地查找到对象在 Java 堆中的起始地址索引和方法区中的对象类型数据。

returnAddress 类型是为字节码指令 jsr、jsr_w 和 ret 服务的，它指向了一条字节码指令的地址。

虚拟机是使用局部变量表完成参数值到参数变量列表的传递过程的，如果是实例方法（非 static），那么局部变量表的第 0 位索引的 Slot 默认是用于传递方法所属对象实例的引用，在方法中通过 this 访问。

Slot 是可以重用的，当 Slot 中的变量超出了作用域，那么下一次分配 Slot 的时候，将会覆盖原来的数据。Slot 对对象的引用会影响 GC（要是被引用，将不会被回收）。

系统不会为局部变量赋予初始值（实例变量和类变量都会被赋予初始值）。也就是说不存在类变量那样的准备阶段。

操作数栈

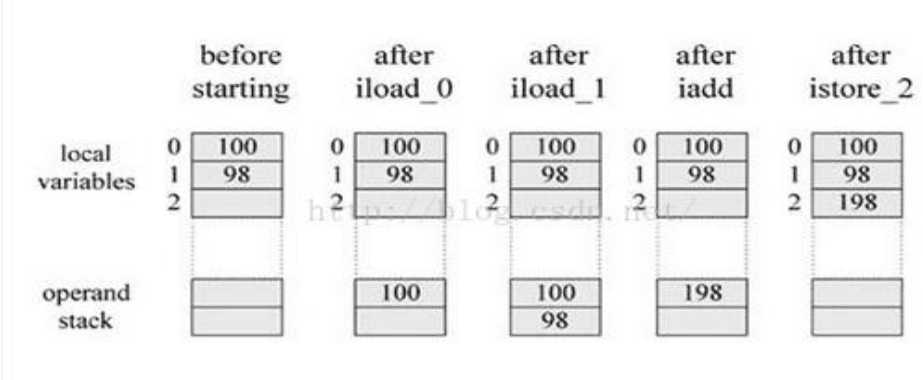
和局部变量区一样，操作数栈也是被组织成一个以字长为单位的数组。但是和前者不同的是，它不是通过索引来访问，而是通过标准的栈操作——压栈和出栈—来访问的。比如，如果某个指令把一个值压入到操作数栈中，稍后另一个指令就可以弹出这个值来使用。

虚拟机在操作数栈中存储数据的方式和在局部变量区中是一样的：如 int、long、float、double、reference 和 returnType 的存储。对于 byte、short 以及 char 类型的值在压入到操作数栈之前，也会被转换为 int。

虚拟机把操作数栈作为它的工作区——大多数指令都要从这里弹出数据，执行运算，然后把结果压回操作数栈。比如，iadd 指令就要从操作数栈中弹出两个整数，执行加法运算，其结果又压回到操作数栈中，看看下面的示例，它演示了虚拟机是如何把两个 int 类型的局部变量相加，再把结果保存到第三个局部变量的：

```
begin
iload_0    // push the int in local variable 0 ontothe stack
iload_1    //push the int in local variable 1 onto the stack
iadd       // pop two ints, add them, push result
istore_2   // pop int, store into local variable 2
end
```

在这个字节码序列里，前两个指令 iload_0 和 iload_1 将存储在局部变量中索引为 0 和 1 的整数压入操作数栈中，其后 iadd 指令从操作数栈中弹出那两个整数相加，再将结果压入操作数栈。第四条指令 istore_2 则从操作数栈中弹出结果，并把它存储到局部变量区索引为 2 的位置。下图详细表述了这个过程中局部变量和操作数栈的状态变化，图中没有使用的局部变量区和操作数栈区域以空白表示



动态连接

虚拟机运行的时候,运行时常量池会保存大量的符号引用，这些符号引用可以看成是每个方法的间接引用。如果代表栈帧 A 的方法想调用代表栈帧 B 的方法，那么这个虚拟机的方法调用指令就会以 B 方法的符号引用作为参数，但是因为符号引用并不是直接指向代表 B 方法的内存位置，所以在调用之前还必须要将符号引用转换为直接引用，然后通过直接引用才可以访问到真正的方法。

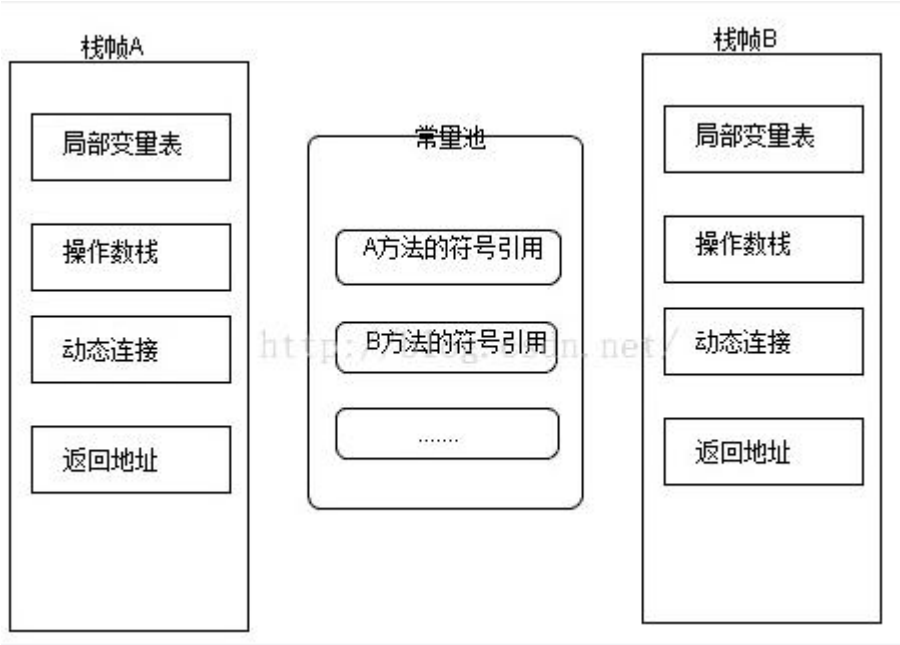
如果符号引用是在类加载阶段或者第一次使用的时候转化为直接应用，那么这种转换成为**静态解析**，如果是在运行期间转换为直接引用，那么这种转换就成为**动态连接。**

返回地址

方法的返回分为两种情况，一种是正常退出，退出后会根据方法的定义来决定是否要传返回值给上层的调用者，一种是异常导致的方法结束，这种情况是不会传返回值给上层的调用方法。

不过无论是那种方式的方法结束，在退出当前方法时都会跳转到当前方法被调用的位置，如果方法是正常退出的，则调用者的 PC 计数器的值就可以作为返回地址,，果是因为异常退出的，则是需要通过异常处理表来确定。

方法的的一次调用就对应着栈帧在虚拟机栈中的一次入栈出栈操作，因此方法退出时可能做的事情包括：恢复上层方法的局部变量表以及操作数栈，如果有返回值的话，就把返回值压入到调用者栈帧的操作数栈中，还会把 PC 计数器的值调整为方法调用入口的下一条指令。



异常

在 Java 虚拟机规范中，对虚拟机栈规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 **StackOverflowError** 异常；如果虚拟机栈可以动态扩展（当前大部分的 Java 虚拟机都可动态扩展，只不过 Java 虚拟机规范中也允许固定长度的虚拟机栈），当扩展时无法申请到足够的内存时会抛出 **OutOfMemoryError** 异常。

本地方法栈

本地方法栈（Native MethodStacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的 Native 方法服务。虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机（譬如 **Sun HotSpot 虚拟机）直接就把本地方法栈和虚拟机栈合二为一**。

与虚拟机栈一样，本地方法栈区域也会抛出 StackOverflowError 和 OutOfMemoryError 异常。

堆

堆是 Java 虚拟机所管理的内存中最大的一块。Java 堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。**此内存区域的唯一目的就是存放对象实例**，几乎所有的对象实例都在这里分配内存。但是随着 JIT 编译器的发展与逃逸分析技术的逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也渐渐变得不是那么“绝对”了。

堆是垃圾收集器管理的主要区域，因此很多时候也被称做“**GC 堆**”。

堆的大小可以通过**-Xms(**最小值)和**-Xmx**(最大值)参数设置，-Xms 为 JVM 启动时申请的最小内存，默认为操作系统物理内存的 1/64 但小于 1G，-Xmx 为 JVM 可申请的最大内存，默认为物理内存的 1/4 但小于 1G，默认当空余堆内存小于 40%时，JVM 会增大 Heap 到-Xmx 指定的大小，可通过-XX:MinHeapFreeRation=来指定这个比列；当空余堆内存大于 70%时，JVM 会减小 heap 的大小到-Xms 指定的大小，可通过 XX:MaxHeapFreeRation=来指定这个比列，对于运行系统，为避免在运行时频繁调整 Heap 的大小，通常-Xms 与-Xmx 的值设成一样。

如果从内存回收的角度看，由于现在收集器基本都是采用的分代收集算法，所以 Java 堆中还可以细分为：新生代和老年代；

新生代：程序新创建的对象都是从新生代分配内存，新生代由 **Eden Space** 和两块相同大小的 **Survivor Space**(通常又称 S0 和 S1 或 From 和 To)构成，可通过**-Xmn** 参数来指定新生代的大小，也可以通过**-XX:SurvivorRation** 来调整 Eden Space 及 SurvivorSpace 的大小。

老年代：用于存放经过多次新生代 GC 仍然存活的对象，例如缓存对象，新建的对象也有可能直接进入老年代，主要有两种情况：1、大对象，可通过启动参数设置**-XX:PretenureSizeThreshold**=1024(单位为字节，默认为 0)来代表超过多大时就不在新生代分配，而是直接在老年代分配。2、大的数组对象，且数组中无引用外部对象。

老年代所占的内存大小为-Xmx 对应的值减去-Xmn 对应的值。

如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出 OutOfMemoryError 异常。

方法区

方法区在一个 jvm 实例的内部，**类型信息**被存储在一个称为方法区的内存逻辑区中。类型信息是由类加载器在类加载时从类文件中提取出来的。类(静态)变量也存储在方法区中。

简单说方法区用来存储类型的元数据信息，一个.class 文件是类被 java 虚拟机使用之前的表现形式，一旦这个类要被使用，java 虚拟机就会对其进行装载、连接（验证、准备、解析）和初始化。而装载（后的结果就是由.class 文件转变为方法区中的一段特定的数据结构。这个数据结构会存储如下信息：

类型信息

这个类型的全限定名

这个类型的直接超类的全限定名

这个类型是类类型还是接口类型

这个类型的访问修饰符

任何直接超接口的全限定名的有序列表

字段信息

字段名

字段类型

字段的修饰符

方法信息

方法名

方法返回类型

方法参数的数量和类型（按照顺序）

方法的修饰符

其他信息

除了常量以外的所有类（静态）变量

一个指向 ClassLoader 的指针

一个指向 Class 对象的指针

常量池（常量数据以及对其他类型的符号引用）

JVM 为每个已加载的类型都维护一个**常量池**。常量池就是这个类型用到的常量的一个有序集合，包括实际的常量(string,integer,和 floating point 常量)和对类型，域和方法的符号引用。池中的数据项象数组项一样，**是通过索引访问的**。

每个类的这些元数据，无论是在构建这个类的实例还是调用这个类某个对象的方法，都会访问方法区的这些元数据。

构建一个对象时，JVM 会在堆中给对象分配空间，这些空间用来存储当前对象实例属性以及其父类的实例属性（而这些属性信息都是从方法区获得），注意，这里并不是仅仅为当前对象的实例属性分配空间，还需要给父类的实例属性分配，到此其实我们就可以回答第一个问题了，即实例化父类的某个子类时，JVM 也会同时构建父类的一个对象。从另外一个角度也可以印证这个问题：调用当前类的构造方法时，首先会调用其父类的构造方法直到 Object，而构造方法的调用意味着实例的创建，所以子类实例化时，父类肯定也会被实例化。

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。**这些变量只与类相关，所以在方法区中**，它们成为类数据在逻辑上的一部分。在 JVM 使用一个类之前，它必须在方法区中为每个 non-final 类变量分配空间。

方法区主要有以下几个特点：

1、方法区是线程安全的。由于所有的线程都共享方法区，所以，方法区里的数据访问必须被设计成线程安全的。例如，假如同时有两个线程都企图访问方法区中的同一个类，而这个类还没有被装入 JVM，那么只允许一个线程去装载它，而其它线程必须等待

2、方法区的大小不必是固定的，JVM 可根据应用需要动态调整。同时，方法区也不一定是连续的，方法区可以在一个堆(甚至是 JVM 自己的堆)中自由分配。

3、方法区也可被垃圾收集，当某个类不在被使用(不可触及)时，JVM 将卸载这个类，进行垃圾收集

可以通过**-XX:PermSize** 和 **-XX:MaxPermSize** 参数限制方法区的大小。

对于习惯在 HotSpot 虚拟机上开发和部署程序的开发者来说，很多人愿意把方法区称为**“永久代”**（PermanentGeneration），本质上两者并不等价，仅仅是因为 HotSpot 虚拟机的设计团队选择把 GC 分代收集扩展至方法区，或者说使用永久代来实现方法区而已。对于其他虚拟机（如 BEA JRockit、IBM J9 等）来说是不存在永久代的概念的。

相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入了方法区就如永久代的名字一样“永久”存在了。这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载。

当方法区无法满足内存分配需求时，将抛出 OutOfMemoryError 异常。

总结

名称	特征	作用	配置参数	异常
程序计数器	占用内存小，线程私有， 生命周期与线程相同	大致为字节码行号指示器	无	无
虚拟机栈	线程私有，生命周期与线程相同，使用连续的内存空间	Java 方法执行的内存模型，存储局部变量表、操作栈、动态链接、方法出口等信息	-Xss	StackOverflowError OutOfMemoryError
java 堆	线程共享，生命周期与虚拟机相同，可以不使用连续的内存地址	保存对象实例，所有对象实例（包括数组）都要在堆上分配	-Xms -Xmx -Xmn	OutOfMemoryError
方法区	线程共享，生命周期与虚拟机相同，可以不使用连续的内存地址	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等数据	-XX:PermSize: 16M -XX:MaxPermSize 64M	OutOfMemoryError
运行时常量池	方法区的一部分，具有动态性	存放字面量及符号引用		

直接内存

直接内存（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是 Java 虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用，而且也可能导致 OutOfMemoryError 异常出现，所以我们放到这里一起讲解。

在 JDK 1.4 中新加入了 NIO(NewInput/Output)类，引入了一种基于通道(Channel)与缓冲区（Buffer）的 I/O 方式，它可以使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆里面的 DirectByteBuffer 对象作为这块内存的引用

进行操作。这样能在一些场景中显著提高性能，因为避免了在 Java 堆和 Native 堆中来回复制数据。

堆与栈的对比

经常有人把 Java 内存区分为堆内存（Heap）和栈内存（Stack），这种分法比较粗糙，Java 内存区域的划分实际上远比这复杂。这种划分方式的流行只能说明大多数程序员最关注的、与对象内存分配关系最密切的内存区域是这两块。

堆很灵活，但是不安全。对于对象，我们要动态地创建、销毁，不能说后创建的对象没有销毁，先前创建的对象就不能销毁，那样的话我们的程序就寸步难行，所以 Java 中用堆来存储对象。而一旦堆中的对象被销毁，我们继续引用这个对象的话，就会出现著名的 NullPointerException，这就是堆的缺点——错误的引用逻辑只有在运行时才会被发现。

栈不灵活，但是很严格，是安全的，易于管理。因为只要上面的引用没有销毁，下面引用就一定还在，在大部分程序中，都是先定义的变量、引用先进栈，后定义的后进栈，同时，区块内部的变量、引用在进入区块时压栈，区块结束时出栈，理解了这种机制，我们就可以很方便地理解各种编程语言的作用域的概念了，同时这也是栈的优点——错误的引用逻辑在编译时就可以被发现。

栈--主要存放引用和基本数据类型。

堆--用来存放 new 出来的对象实例。

内存溢出和内存泄漏

内存溢出 **out of memory**，是指程序在申请内存时，没有足够的内存空间供其使用，出现 out of memory；比如申请了一个 integer，但给它存了 long 才能存下的数，那就是内存溢出。

内存泄露 **memory leak**，是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。

memory leak 会最终会导致 out ofmemory。

Java 堆内存的 OutOfMemoryError 异常是实际应用中最常见的内存溢出异常情况。出现 Java 堆内存溢出时，异常堆栈信息“java.lang.OutOfMemoryError”会跟着进一步提示“Java heapspace”。

要解决这个区域的异常，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）。

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。

如果不存在泄漏，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。

内存分配过程

1、JVM 会试图为相关 Java 对象在 Eden Space 中初始化一块内存区域。

2、当 Eden 空间足够时，内存申请结束；否则到下一步。

3、JVM 试图释放在 Eden 中所有不活跃的对象（这属于 1 或更高级的垃圾回收）。释放后若 Eden 空间仍然不足以放入新对象，则试图将部分 Eden 中活跃对象放入 Survivor 区。

4、Survivor 区被用来作为 Eden 及 Old 的中间交换区域，当 Old 区空间足够时，Survivor 区的对象会被移到 Old 区，否则会被保留在 Survivor 区。

5、当 Old 区空间不够时，JVM 会在 Old 区进行完全的垃圾收集（0 级）。

6、完全垃圾收集后，若 Survivor 及 Old 区仍然无法存放从 Eden 复制过来的部分对象，导致 JVM 无法在 Eden 区为新对象创建内存区域，则出现“outofmemory”错误。

对象访问

对象访问在 Java 语言中无处不在，是最普通的程序行为，但即使是最简单的访问，也会却涉及 Java 栈、Java 堆、方法区这三个最重要内存区域之间的关联关系，如下面的这句代码：

```
Object obj = newObject();
```

假设这句代码出现在方法体中，那“Object obj”这部分的语义将会反映到 Java 栈的本地变量表中，作为一个 reference 类型数据出现。而“new Object()”这部分的语义将会反映到 Java 堆中，形成一块存储了 Object 类型所有实例数据值（Instance Data，对象中各个实例字段的数据）的结构化内存，根据具体类型以及虚拟机实现的对象内存布局（Object Memory Layout）的不同，这块内存的长度是不固定的。另外，在 Java 堆中还必须包含能查找到此对象类型数据（如对象类型、父类、实现的接口、方法等）的地址信息，这些类型数据则存储在方法区中。

由于 reference 类型在 Java 虚拟机规范里面只规定了一个指向对象的引用，并没有定义这个引用应该通过哪种方式去定位，以及访问到 Java 堆中的对象的具体位置，因此不同虚拟机实现的对象访问方式会有所不同，主流的访问方式有两种：**使用句柄**和**直接指针**。

如果使用句柄访问方式，Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据和类型数据各自的具体地址信息。

---------- 作者：冰河 winner 来源：CSDN