线程隔离的数据区

由所有线程共享的数据区

本地方法库

本地库接口

执行引擎

本地方法栈 Native Method Stack

虚拟机栈 VM Stack

方法区 Method Area

**运行时数据区**

程序计数器 Program Counter Register

堆 Heap

程序计数器(Program Counter Register):是一块较小的内存区域，它可以看作时当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里，字节码解释器工作时通过改变这个计数器的值来选取下一条需要执行的字节码指令，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一个内核)都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，每条线程都需要一个独立的程序计数器，各条线程之间互不影响，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Native方法，这个计数器值则为空。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。

Java虚拟机栈(Java Virtual Machine Stacks):也是线程私有的,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，每个方法在执行的同时都会创建一个栈帧(Stack Frame)用于存储局部变量表，操作数栈，动态连接，方法出口等信息。每一个方法从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入栈到出栈的过程。

局部变量表存放了编译期可知的各种基本数据类型(boolean, byte, char, short, int, float, long double),对象引用(reference类型，它不等同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置)和returnAddress类型(指向了一条字节码指令的地址)

其中64位长度的long和double类型的数据会占有2个局部变量空间(Slot),其余的数据类型只占用1个，局部变量表所需的内存空间在编译期完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的大小。

在Java虚拟机规范中，对这个区域规定了两种异常情况，如果线程请求的栈深度大于虚拟机的深度，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虚拟机都可以动态扩展，只不过Java虚拟机规范中也允许固定长度的虚拟机栈)，如果扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

本地方法栈(Native Method Stack)与虚拟机栈非常相似，它们之间的区别不过是虚拟机栈执行Java方法(也就是字节码)服务，而本地方法栈则为虚拟机使用到的Native方法服务。在虚拟机规范中对本地方法栈中方法使用的语言，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。

Java堆(Java Heap)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。这一点在Java虚拟机规范描述是:所有的对象实例以及数组都在堆上分配，但是随着JIT编译器(把Java字节码转换成可以直接发送给处理器的指令的程序)的发展与逃逸分析技术逐渐成熟，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。

Java堆是垃圾收集器管理的主要区域，因此很多时候也被称作“GC堆”(Garbage Collected Heap)，从内存回收的角度来看，由于现在收集器基本都采用分代收集算法。所以Java堆中还可以细分为：新生代和老年代：在细致一点的有Eden空间，From Survivor空间，To Survivor空间等。从内存分配的角度来看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区(Thread Local Allocation Buffer, TLAB).不过无论如何划分，都与存放内容无关，无论哪个区域，存储的都仍然是对象实例，进一步划分的目的是为了更好地回收内存，或者更快地分配内存。

根据Java虚拟机规范的规定，Java堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘空间一样。在实现时，即可以实现成固定大小的，也可以是可扩展的，不过当前主流的虚拟机都是按照可扩展来实现的(通过-Xmx和-Xms控制)。如果在堆中没有内存完成实例分配，并且堆也无法在扩展时，将会抛出OutOfMemoryError异常。

方法区(Method Area)与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息，常量，静态常量，即时编译器编译后的代码等数据。虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做Non-Heap(非堆)，目的应该是与Java堆区分开来。

对于习惯在HotSpot虚拟机上开发，部署程序的开发者来说，很多人都更愿意把方法去称为”永久代”(Permanent Generation),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，原则上，如何实现方法区属于虚拟机实现细节，不受虚拟机规范约束，但使用永久代来实现方法区，现在看来并不是一个好主意，因为这样更容易遇到内存溢出问题(永久代有-XX:MaxPermSize的上限)，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出。

Java虚拟机规范对方法区限制非常宽松，除了和Java堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。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，一般来说，这个区域的回收“成绩”比较难以令人满意，尤其是类型的卸载，条件相当苛刻，但是这部分区域的回收确实是必要的。

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

运行时常量池(Runtime Constant Pool)是方法区的一部分，Class文件中除了有类的版本，字段，方法，接口等描述信息外，还有一项信息是常量池(Constant Pool Table),用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

Java虚拟机对Class文件每一部分(自然*也包括常量池)的*格式都有严格规定，每一个字节用于存储哪种数据都必须符合规范上的要求才会被虚拟机认可，装载和执行，但对于运行时常量池，Java虚拟机规范没有做任何细节的要求，不同的虚拟机可以按照自己的需要来实现这个内存区域。不过，一般来说，除了保存Class文件中描述的符号引用外，还会把翻译出来的直接引用也存储在运行时常量池中。

运行时常量池相对于Class文件常量池的另外一个重要特征是具备动态性，Java语言并不要求常量一定只有编译期才能产生，也就是并非预置入Class文件中常量池的内容才能进入方法区运行时常量池，运行区间也可能将新的常量池放入池中。这种特性被开发人员利用得比较多的便是String的intern()方法

在JDK6中，常量池位于PermGen(永久代)中，PermGen是一块主要用于存放已加载的类信息和字符串池的大小固定的区域。执行intern()方法时，若常量池中不存在等值的字符串，JVM就会在常量池中创建一个等值的字符串，然后返回该字符串的引用。

在JDK7中，常量池从PpermGen区域移到Java堆区，执行intern操作，如果常量池已经存在该字符串，则直接返回字符串的引用，否则复制该字符串对象的引用到常量池中并返回。

intern(）方法优点：执行速度非常快，直接使用==进行比较要比使用equals(）方法快很多；内存占用少。

当常量池无法再申请到内存时会抛出OutOfMemoryError异常。

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。

在JDK 1.4中新加入了NIO(New Input/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堆中的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。

显然，本机直接内存的分配不会受到Java堆大小的限制，但是，既然是内存，肯定还是会受到本机总内存大小以及处理器寻址空间的限制。服务器管理员在配置虚拟机参数的时，会根据实际内存设置-Xmx等参数信息，但经常忽略直接内存，使得各个内存区域总和大于物理内存限制，从而导致动态扩展时出现OutOfMemoryError异常。

对象的创建

虚拟机遇到一条new指令时，首先将去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已被加载，解析和初始化过。如果没有，那必须先执行相应的类加载过程。

在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需内存的大小在类加载完成后便可完全确定，为对象分配空间的任务等同于把一块确定大小的内存从Java堆中划分出来，假设Java堆中内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离，这种分配方式称为指针碰撞(Bump the Ppointer).