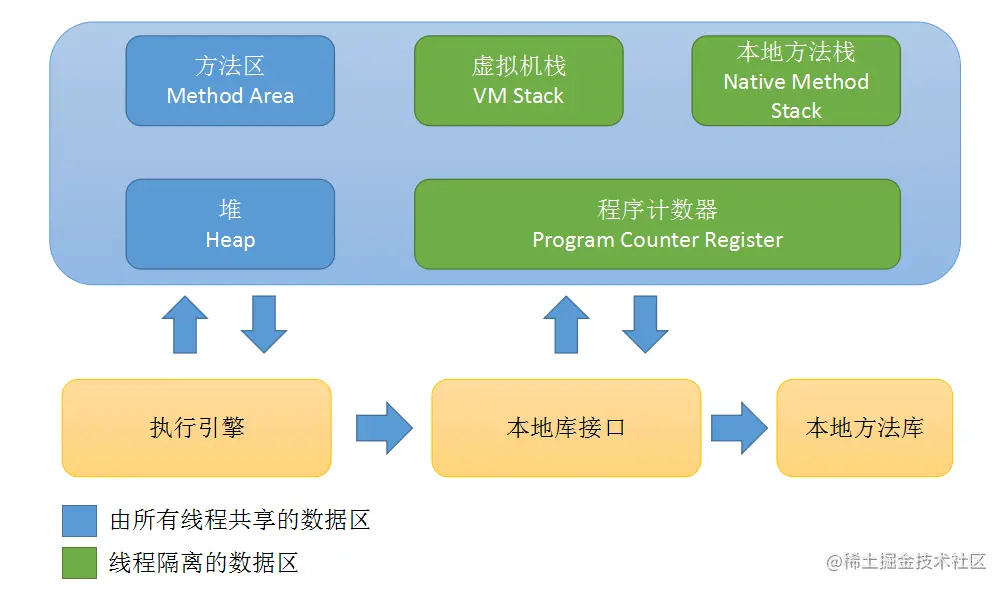
在《Java虚拟机规范》中规定了五种虚拟机运行时数据区，他们分别为：程序计数器、Java虚拟机栈、本地方法栈、本地方法区、Java堆、以及方法区。如下图所示：



程序计数器

程序计数器是用于存放下一条指令所在单元地址的一块内存，在Intel x86和Itanium微处理器中，被称作指令指针（instruction pointer，IP），显然“指令指针”这个名字更易于我们理解这个概念。在JVM中，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字节码的行号指示器。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、线程恢复等功能都需要依赖这个计数器来完成。

我们知道，多线程的情况下，线程是由CPU调度来执行指令的，在单核CPU中，某一时刻只会有一个线程在执行指令，这就意味着CPU会频繁的切换线程，而在线程切换后需要恢复到正确的执行位置，这就需要每条线程都有一个独立的程序计数器，各个线程之间计数器互不影响。

另外，关于程序计数器我们需要知道的是，Java虚拟机规范中没有针对程序计数器规定任何的OutMemoryError。至于为什么？我个人觉得应该是程序计数器所占用的内存很小，加之线程数是有限的。因此，规范中并未对其做任何约束限制。 从以上内容中我们可以总结出以下几点：

程序计数器是线程私有的，每条线程都有一个程序计数器。

程序计数器存放了下一条要执行的指令的位置

字节码解释器通过给改变计数器的值来执行指令

程序计数器是JVM内存中唯一没有规定OutMemoryError的区域

Java虚拟机栈

提到Java中的栈内存，大家应该都不陌生，甚至很多小伙伴对于栈内存说的头头是道：“栈内存是用来存储基本数据类型和对象的引用的”。但是，事实如此吗？其实，并不准确！这仅仅是Java虚拟机栈功能的一部分。

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。

栈帧是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区中的虚拟机栈的栈元素，栈帧存储了方法的变量表、操作数栈、动态连接和方法返回等信息。

而我们平时说的“栈内存”其实指的是虚拟机中的局部变量表部分。局部变量表存放了编译器可以知道的各种基本数据类型（boolean、type、char、short、int、final、long、double）、对象引用（reference类型，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它与此对象相关的位置）和returnAddress类型（指向了一条字节码指令地址）。其中长度为long和double类型的数据会占用两个局部变量空间（Slot），其余的数据类型只占用一个。而局部变量表所需要的空间是在编译期间完成分配的，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，在方法的运行期间不会改变局部变量的大小。

同时，Java虚拟机栈与程序计数器一样都是线程私有的。Java虚拟机栈的生命周期与线程相同。另外，在Java虚拟机规范中对这一区域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，就会抛出StackOverflowError异常；如果虚拟机栈可以动态扩展（当前大部分的虚拟机都可以动态扩展）如果扩展时无法申请到足够的内存则会抛出OutOfMemoryError异常。

对于以上内容，我们可以有以下几点结论：

虚拟机栈是线程私有的，生命周期与线程相同

虚拟机栈的功能不仅仅是存储基本数据类型和对象的引用

虚拟机栈描述的是Java方法执行的内存模型

Java虚拟机栈内存不足时会抛出StackOverflowError异常和OutOfMemoryError异常

本地方法栈

本地方法栈与虚拟机栈的作用是非常相似的，他们之间的区别不过是虚拟机栈为虚拟机执行Java方法（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、使用的方式与数据结构并没有强制的规定，因此虚拟机是可以自由实现的。像HotSpot虚拟机就把本地方法栈和虚拟机栈合二为一了。

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

关于本地虚拟机栈，我们有如下总结：

本地方法栈为虚拟机使用到的Native方法服务

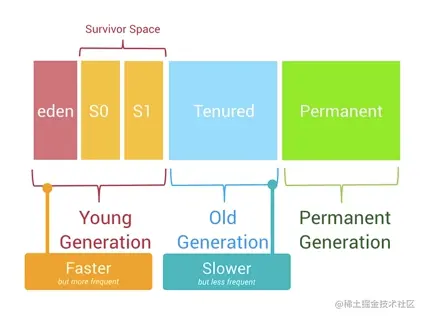
某些虚拟机本地方法栈与虚拟机栈合二为一

本地方法栈区域会抛出StackOverflowError和OutOfMemoryError

Java堆

我们所创建的对象实例几乎都是存放在堆内存中。因此，Java堆是虚拟机所管理的最大的一块内存区域。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。Java虚拟机规范中对堆内存有这样的描述：所有的对象实例以及数组都要在堆上分配，但是随着JIT编译器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，所有的对象都在堆上分配也不是那么绝对了。

对于Java堆内存我们还可以进行一个细分：新生代（Young Generation）、老年代（Old Generation）；而新生代则又可以细分为Eden空间、From Survivor空间、To Survivor。如下图所示（注：图片中Permanent Generation即永久代不属于堆内存，而是方法区的一部分）：



而Java中的垃圾回收就是针对Java堆上不同的区域进行扫描回收。

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缓冲区（Thread Local Allocation Buffer，TLAB).

但是，无论怎么划分，堆上所存放的内容都是对象的实例。进一步划分只不过是为了更好的回收内存或者更快的分配内存。

另外，Java虚拟机规范中规定，Java堆可以处于物理上不连续的内存空间中。如果虚拟机在为实例分配内存时没有足够的空间，并且堆也无法在扩展，则会抛出OutOfMemoryError异常。

关于Java堆我们可做如下总结：

堆内存是被所有线程共享的一块内存区域

Java中创建的对象几乎都存放在堆内存中

垃圾回收机制会对堆内存进行扫描和垃圾对象的回收

堆内存不足时会抛出OutOfMemoryError异常

方法区

方法区是用来存储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。方法区与堆内存一样是所有线程共享的一块区域。Java虚拟机对于方法区的限制非常宽松，因此也就导致了不同的虚拟机上方法区有不同的表现。我们以HotSpot虚拟机为例：方法区在JDK1.7之前是一块单独的区域，很多人喜欢把它成为“永久代”（Permanent Generation），但本质上两者并不等价，仅仅是因为HotSpot虚拟机的设计团队把GC分代收集扩展到了方法区，或者说使用永久代来实现了方法区。这样HotSpot的垃圾收集器就可以向管理Java堆一样管理这部分内存。但是对于其它虚拟机（如BEA JRockit、IBM J9等）来说其实是不存在永久代的概念的。用永久代来实现方法区并不是一个好主意，因为这样更容易出现内存溢出的问题。而HotSpot的团队显然也意识到了这点问题，因此，在JDK1.7中将方法区中的字符串常量池移到了堆内存中，并在JDK1.8中完全废除了“永久代”，使用元空间替代了永久代。

另外，根据Java虚拟机规范规定，当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。

关于方法区有如下总结：

方法区与堆内存一样是所有线程共享的一块区域

方法区是用来存储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

不同的虚拟机对于方法区的实现不同

HotSpot虚拟机在JDK1.7中将字符串常量池移到了堆内存，并在JDK1.8中用元空间去掉了“永久代”。

方法区内存不足时会抛出OutOfMemoryError异常

Java中的常量池

关于常量池Java中可以分为三种，分别为：全局字符串常量池、Class文件常量池、运行时常量池。很多不了解的开发者容易将这三块区域混淆。本节内容我们就来详细的堆这三种常量池进行分析。

1.字符串常量池（String Pool）

我们知道，字符串属于引用数据类型，但是可以说字符串是Java中使用频繁的一种数据类型。因此，为了节省程序内存，提高性能，Java的设计者开辟了一块叫字符串常量池的区域，用来存储一些全局的字符串。字符串常量池是所有类公用的一块空间，在一个虚拟机中只有一块常量池区域。在类加载完成，经过验证，准备阶段之后在堆中生成字符串对象实例，然后将该字符串对象实例的引用值存到字符串常量池中（这里描述指的是JDK7及以后的HotSpot虚拟机）。 在HotSpot虚拟机中字符串常量池是通过一个StringTable类来实现的。它是一个哈希表，里面存的是字符串引用（JDK6及之前版本存放的是字符串对象）。 我们在上一节中提到，在JDK7以前，字符串常量池在方法区（永久代）中，此时常量池中存放的是字符串对象。而在JDK7中，字符串常量池从方法区迁移到了堆内存，同时将字符串对象存到了堆内存，只在字符串常量池中存入了字符串对象的引用。

2.Class文件常量池（Class Constant Pool）

我们知道，Java文件在通过javac编译后会生成Class文件，Class文件中除了存放类的版本、字段、方法、接口等描述信息外，还有一项是常量池（Constants Pool Table），常量池用于存放编译期间生产的各种字面量和符号引用。

3.运行时常量池（Runtime Constant Pool）

运行时常量池是方法区的一部分，虚拟机会在类加载后将存Class文件常量池的内容加载到运行时常量池。因此，每一个类都会有一个运行时常量池。运行时常量池相对于Class文件常量池有具备动态性。因为Java语言并不要求常量一定在编译期间才能生成。也就是并非预置入Class文件常量池中的内容才能进入运行时常量池，运行期间也可以将新的常量放入常量池中，例如String类中的intern()方法， 由于运行时常量池属于方法区，因此其内存限制自然与方法区无异，当常量池无法再申请到内存时会抛出OutMemoryError异常。