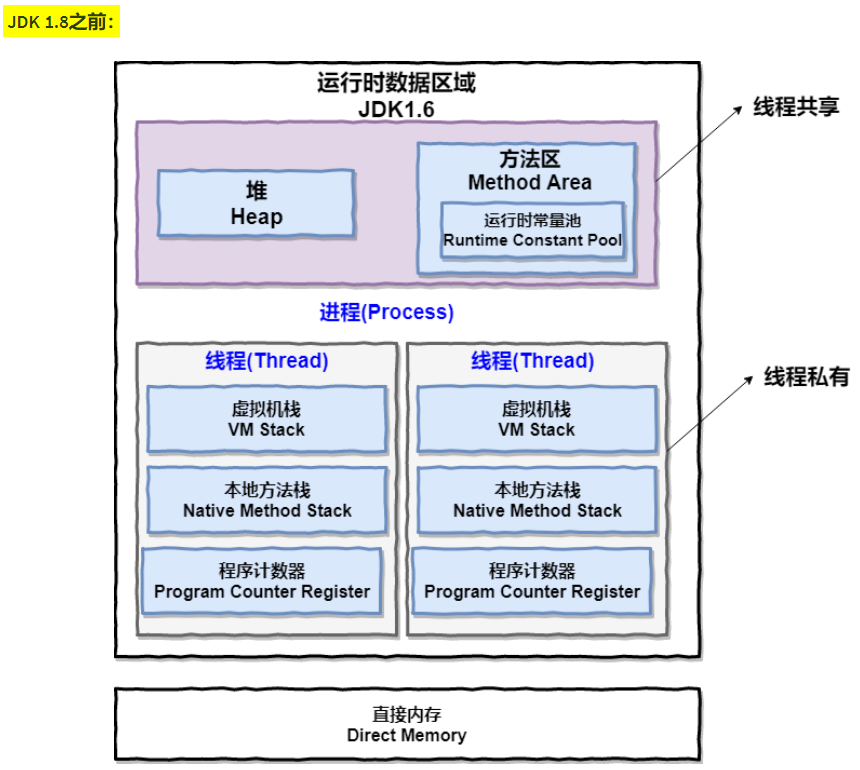
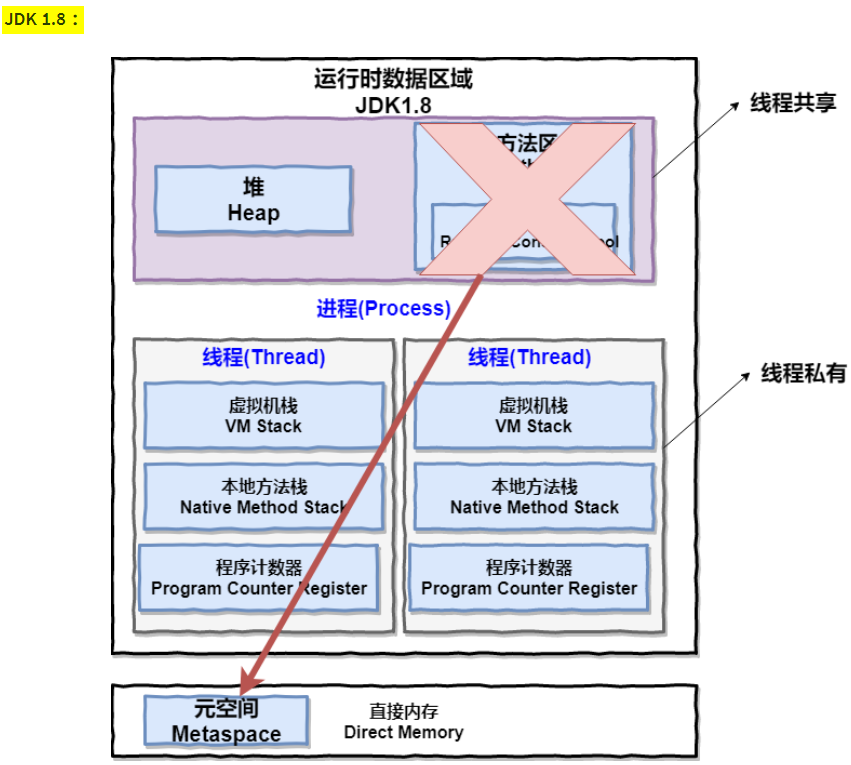
## 运行时数据区域

Java虚拟机在执行Java程序的过程中会把它管理的内存划分为若干个不同的数据区域。JDK 1.8和之前的版本略有不同。





线程私有的：

程序计数器

虚拟机栈

本地方法栈

线程共享的：

堆

方法区

直接内存（非运行时数据区的一部分）

### 2.1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。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。

另外，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，各线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

程序计数器主要的两个作用：

1. 字节码解释器通过改变程序计数器来一次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

注意：程序计数器是唯一一个不会出现OutOfMemoryError的内存区域，它的生命周期随着线程的创建而创建，随着线程的结束而死亡。

### 2.2Java虚拟机栈

与程序计数器一样，Java虚拟机栈也是线程私有的，它的生命周期和线程相同，描述的是Java方法执行的内存模型，每次方法调用的数据都是通过栈传递的。

Java内存可以粗糙的区分为堆内存（Heap）和堆内存（Stack），其中栈就是虚拟机栈，或者说是虚拟机栈中局部变量表部分。（实际上，Java虚拟机是由一个个栈帧组成，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法出口信息。）

局部变量表主要存放了编译器可知的各种数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference类型，它不同于对象本身，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）。

Java虚拟机栈会出现以下两种异常：

1. StackOverFlowError(栈溢出错误)：若Java虚拟机栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前Java虚拟机栈的最大深度的时候，就抛出异常；
2. OutOfMemoryError(内存溢出错误)：若Java虚拟机栈的内存大小允许动态扩展，且当线程请求栈时内存用完了，无法再动态扩展了，此时抛出异常。

Java虚拟机栈也是线程私有的，每个线程都有各自的Java虚拟机栈，而且随着线程的创建而创建，随着线程的死亡而死亡。

扩展：那么方法/函数如何调用？

Java栈可用类比数据结构中栈，Java栈中保存的主要内容是栈帧，每一次函数调用都会有一个对应的栈帧被压入Java栈，每一个函数调用结束后，都会有一个栈帧被弹出。

Java方法有两种返回方式：

1. return语句
2. 抛出异常

不管哪种返回方式都会导致栈帧被弹出。

### 2.3本地方法栈

和虚拟机栈所发挥的作用非常相似，区别是：虚拟机栈作为虚拟机执行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HotSpot虚拟机中和Java虚拟机栈合二为一。

本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、操作数栈、动态链接、出口信息。

方法执行完毕后相应的栈帧也会出栈并释放内存空间，也会出现StackOverFlowError 和 OutOfMemoryError 两种异常。

### 2.4堆

Java虚拟机所管理的内存中最大的一块，Java堆是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及数组都在这里分配内存。

Java堆就是垃圾收集器管理的主要区域，因此也被称作GC堆（Garbage Collected Heap）。从垃圾回收的角度，由于现在收集器甚至都采用分代垃圾收集算法，所以Java堆还可以细分为：新生代和老年代；再细致一点有：Eden空间、From Survivor、To Survivor空间等。进一步划分的目的是更好地回收内存，或者更快地分配内存。



大部分情况，对象都会首先在Eden区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入s0或s1，并且对象的年龄还会加1（Eden区->Survivor区后对象的初始年龄变为1），当它的年龄增加到一定程度（默认为15岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数-XX:MaxTenuringThreshold 来设置。

### 2.5方法区

方法区和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、静态变量、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，但是它却有另一个别名叫做Non-Heap（非堆），目的应该是与Java堆区分开来。

方法区也被称为永久代。

方法区和永久代的关系：

《Java虚拟机规范》只是规定了有方法区这么个概念和它的作用，并没有规定如何去实现它。那么，在不同的 JVM 上方法区的实现肯定是不同的了。 方法区和永久代的关系很像Java中接口和类的关系，类实现了接口，而永久代就是HotSpot虚拟机对虚拟机规范中方法区的一种实现方式。 也就是说，永久代是HotSpot的概念，方法区是Java虚拟机规范中的定义，是一种规范，而永久代是一种实现，一个是标准一个是实现，其他的虚拟机实现并没有永久带这一说法。

1. JDK 1.8之前永久代还没有被彻底移除的时候通常通过下面这些参数来调节方法区大小。

-XX:PermSize=N

//方法区(永久代)初始大小

-XX:MaxPermSize=N

//方法区(永久代)最大大小,超过这个值将会抛出OutOfMemoryError异常：java.lang.OutOfMemoryError: PermGen

相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入方法区后就“永久存在”了。

1. JDK 1.8的时候，方法区（HotSpot的永久代）被彻底移除了（JDK 1.7就已经开始了），取而代之的是元空间，元空间使用的是直接内存。下面是一些常用的参数：

-XX:MetaspaceSize=N //设置Metaspace的初始（和最小大小）

-XX:MaxMetaspaceSize=N //设置Metaspace的最大大小

与永久代很大的不同就是，如果不指定大小的话，随着更多类的创建，虚拟机会耗尽所有可用的系统内存。

为什么要将永久代（PermGen）替换成元空间（MetaSpace）呢？

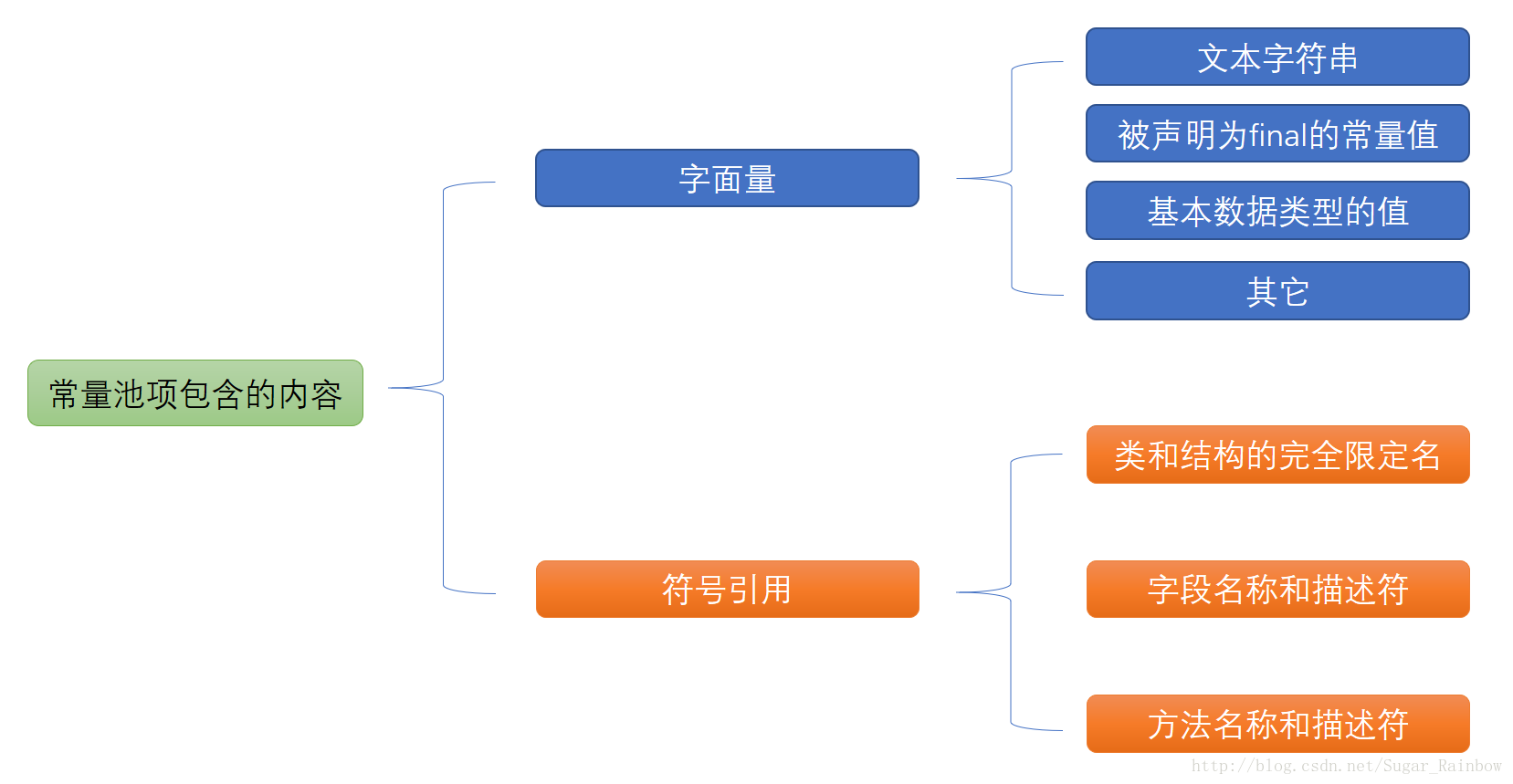
整个永久代有一个JVM本身设置固定大小上线，无法进行调整，而元空间使用的是直接内存，受本机可用内存的限制，且永远不会得到java.lang.OutOfMemoryError。你可以使用-XX：MaxMetaspaceSize标志设置最大元空间大小，默认值为unlimited，这意味着它只受系统内存的限制。-XX：MetaspaceSize调整标志定义元空间的初始大小如果未指定此标志，则Metaspace将根据运行时的应用程序需求动态地重新调整大小。当然这只是其中一个原因，还有很多底层的原因，这里就不提了。

### 2.6运行时常量池

运行时常量池是方法区的一部分。Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有常量池信息（存放编译器生成的各种字面量和符号引用）。

既然运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出OutOfMemoryError异常。

JDK 1.7及之后版本的JVM已经将运行时常量池从方法区中移了出来，在Java堆（Heap）中开辟了一块区域存放运行时常量池。



### 2.7直接内存

直接内存并不是虚拟机运行时数据区的一部分，也不是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OutOfMemory异常出现。

JDK 1.4中新加入的NIO（New Input/Output）类，引入了一种基于通道（Channel）与缓存区（Buffer）的I/O方式，它可以直接使用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。这样就能在一些场景中显著提高性能，因为避免了在Java堆和Native堆之间来回复制数据。

本机直接内存的分配不会受到Java堆的限制，但是，既然是内存就会受到本机总内存大小以及处理器寻址空间的限制。