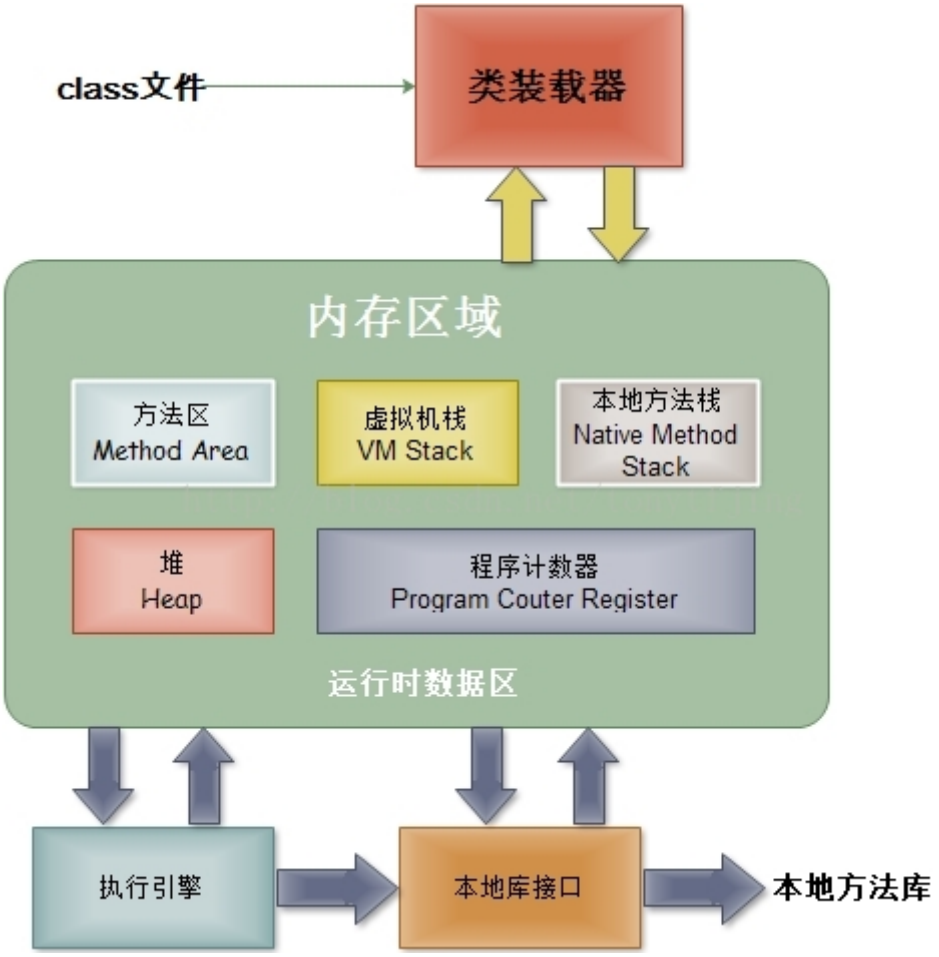


1.类加载器（ClassLoader）:在JVM启动时或者在类运行时将需要的class加载到JVM中。

2.执行引擎：负责执行class文件中包含的字节码指令。

3.内存区（运行时数据区）：是在JVM运行的时候操作所分配的内存区。

4.本地方法接口：主要是调用C或C++实现的本地方法栈返回的结果。

方法区(Method Area)：存储类结构信息，包括常量池（JDK1.7移到堆中）、静态变量、类信息和JIT编译后的机器码等数据。此外还

包含一个运行时常量池。方法区又称永久代（不会被GC回收，JDK8移除永久代），方法区移至直接内存的元数据区。元数据区：

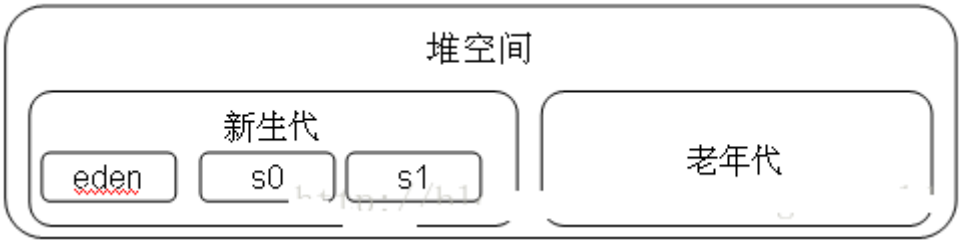
-XX:MetaspaceSize=N　　//设置Metaspace 的初始（和最小大小）。

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

ps.成员变量的基本数据类型是存在方法区的常量池里，局部变量的基本数据类型是存在栈内存的常量池里。

java堆(Heap)：JVM中内存最大一块，存储（几乎）所有对象实例和数组。是GC（垃圾收集管理器，也称GC堆）管理的主要区域。由于现在的垃圾收集器采用分代收集算法，所以Java堆还可细分为:新生代（Eden空间，From Survivor空间，To Survivor空间）和老生代。无论哪个区域，都是用来存放对象实例。

如果当前对堆没有内存完成对象实例的创建，并且不能在进行内存扩展，则会抛出OutOfMemory异常。



ps. Java堆和方法区可以是物理上不连续的空间，只要逻辑上连续，也可扩展。

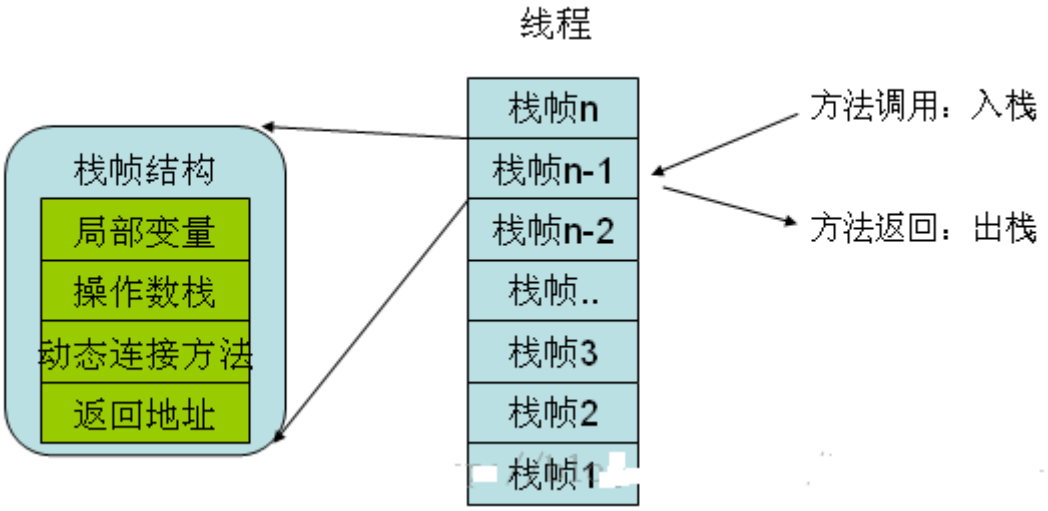
java栈(Stack)：java方法执行的内存模型。每创建一个线程时，JVM就会创建一个与之对应的java栈和程序计数器（生命周期和线程相同）。栈又包含多个栈帧（StackFrame），每执行一个方法就创建一个栈帧【存储方法的**局部变量表、操作数栈、动态链接方法、执行环境（方法返回值，方法出口）**】。每一个方法从调用直至执行完成的过程，就对应一个栈帧在java栈中入栈到出栈（不管方法返回return语句还是抛出异常）的过程。

局部变量表存放了（局部变量的）**基本数据类型、对象引用和returnAddress类型**（指向一条字节码指令的地址）。局部变量表所需的内存空间在java编译器完成分配，当进入一个方法时这个方法需要在帧中分配多大的内存空间是完全确定的，运行期间不会改变局部变量表的大小。ps. 64为长度的long和double会占用两个局部变量空间，其他的数据类型占用一个

Java虚拟机栈可能出现两种类型的异常：

1. 若虚拟机栈内存不允许动态扩展，当线程请求的栈深度（递归多少层次，或嵌套调用多少层其他方法，-Xss参数可以设置虚拟机栈大小）超过虚拟机允许的栈深度，将抛出StackOverflowError。

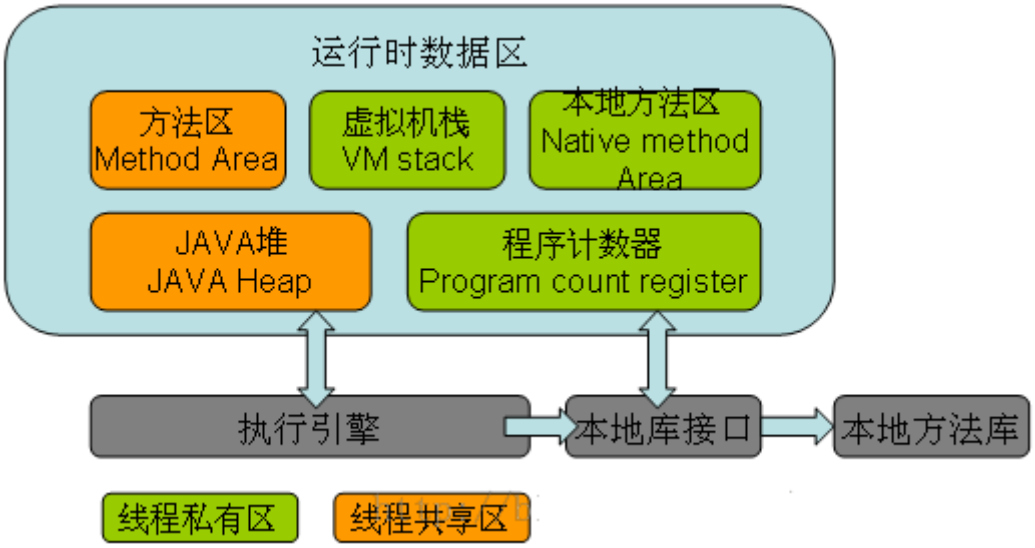
2. 若虚拟机栈空间可以动态扩展，当线程请求的栈用完了，无法再扩展时，抛出OutOfMemoryError异常。



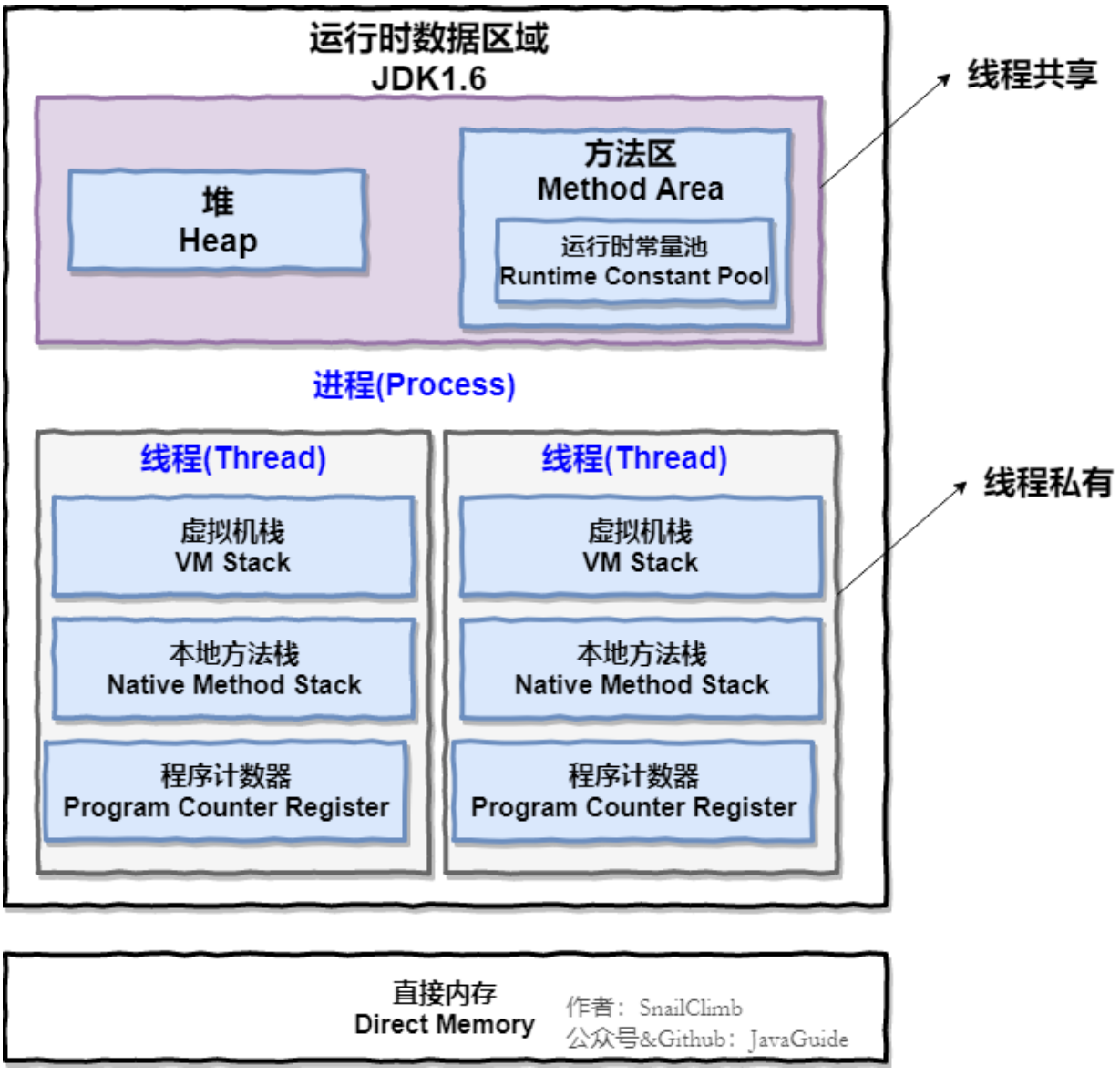
程序计数器(PC Register)：较小的区域，保存**当前线程执行的内存地址**。可以看做是当前线程所执行的**字节码的行号指示器**。字节码解释器通过改变程序计数器的值来指定下一条需要执行的指令，从而实现代码的流程控制。【另外由于JVM程序是多线程执行的（线程轮流切换），一个处理器同一时间只会执行一条线程中的指令。所以为了保证线程切换回来后，还能恢复到原先状态，每条线程就需要一个独立的计数器，记录之前中断的地方，】它的生命周期随着线程的创建而创建，随着线程的结束而死亡。

如果虚拟机正在执行的是一个Java方法，则计数器指定的是字节码指令对应的地址，如果正在执行的是一个本地方法，则计数器指定问空undefined。程序计数器区域是Java虚拟机中唯一没有定义OutOfMemory异常的区域。

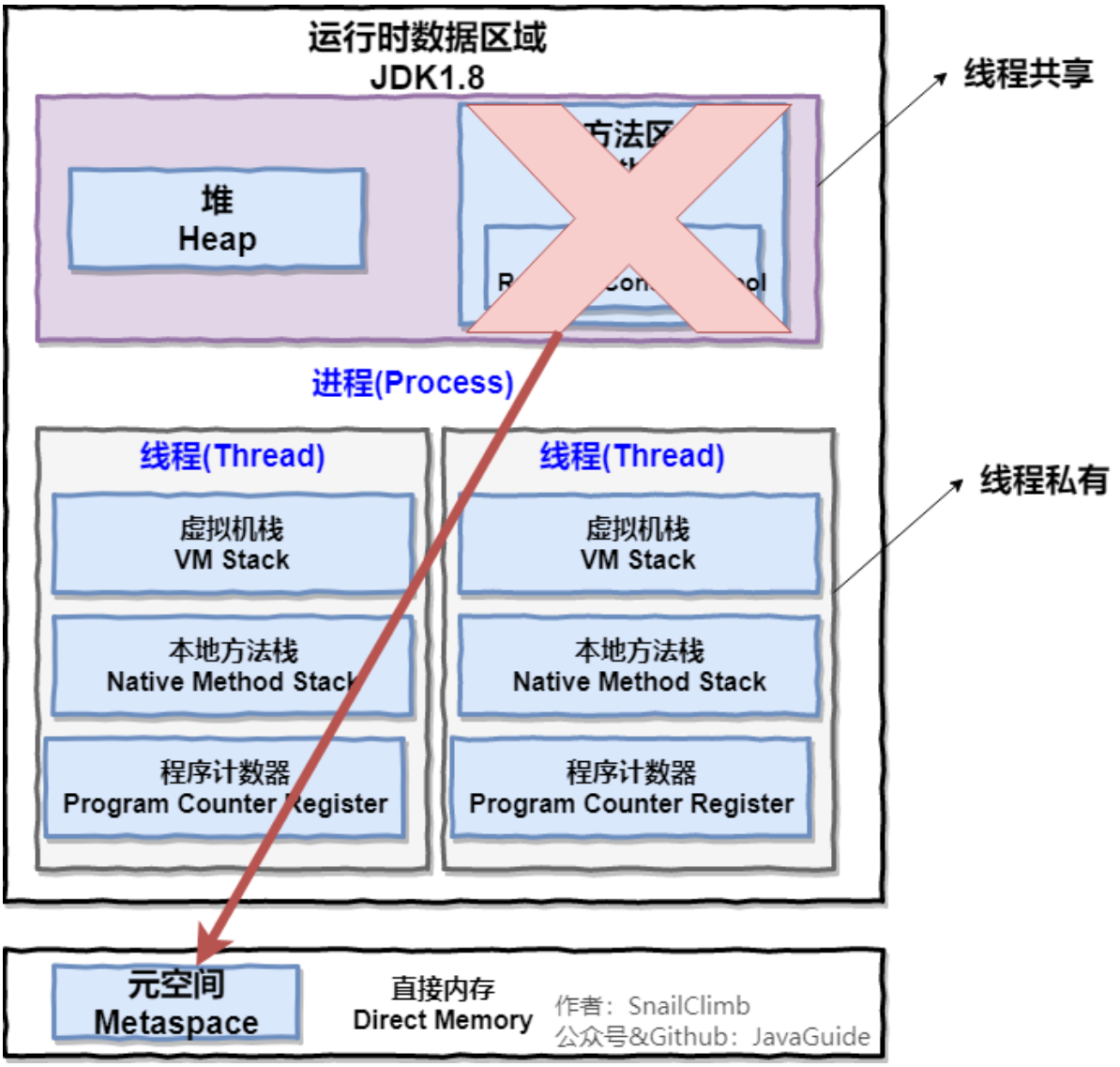
本地方法栈(Native Method Stack)：类似java栈，区别在于Java虚拟机栈执行的是Java代码（字节码），本地方法栈中执行的是C实现的本地方法（Native）的服务。*本地方法栈中也会抛出StackOverflowError和OutOfMemory异常。*



JDK 1.8 之前：



JDK 1.8 ：



直接内存

直接内存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。这部分内存被频繁地使用，也可能出现 OutOfMemoryError 异常。

JDK1.4 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，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收到 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。