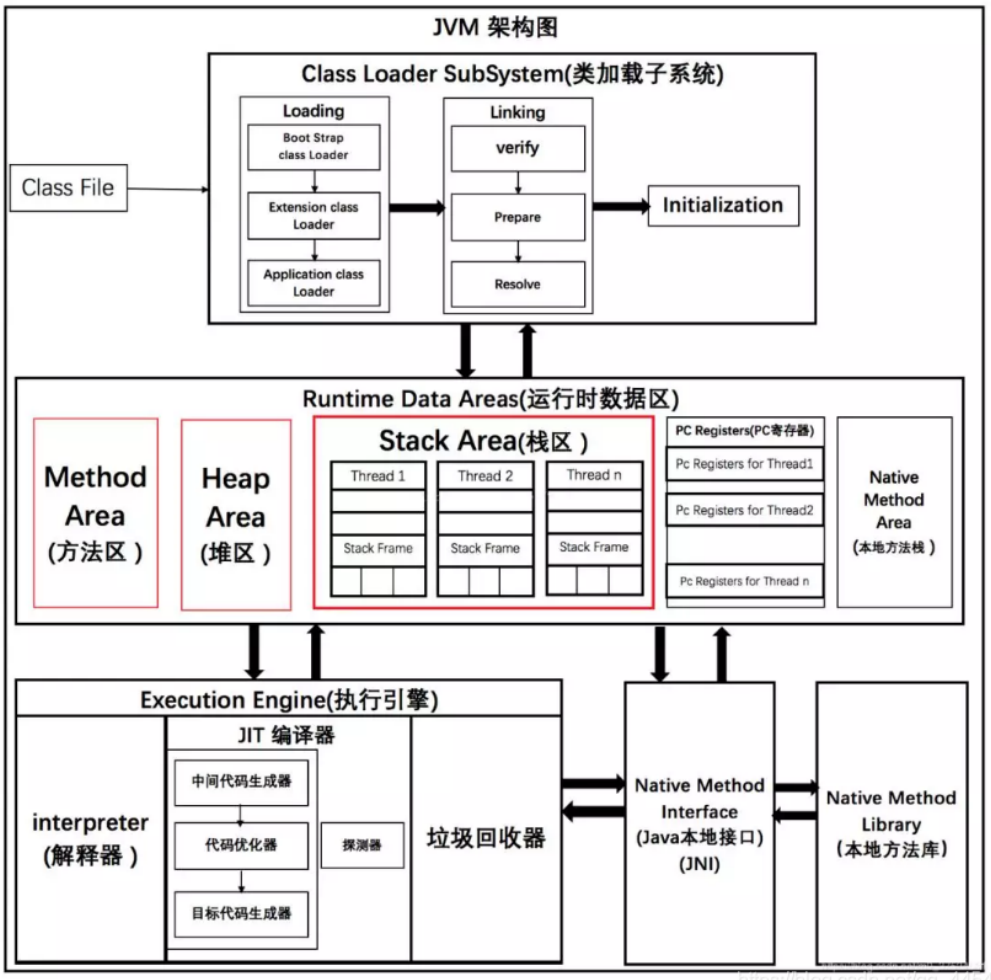
1、内存结构还是运行时数据区?

从某一角度来说，Java 虚拟机的内存结构 == 运行时数据区，在《Java 虚拟机规范》中用的是【运行时数据区】术语的，并没有内存结构这么一说法。内存结构只是听着更加贴切，更加形象，因此知道**内存结构就是运行时数据区的意思**就好了！也没必要钻牛角尖纠结这个问题。

2、运行时数据区

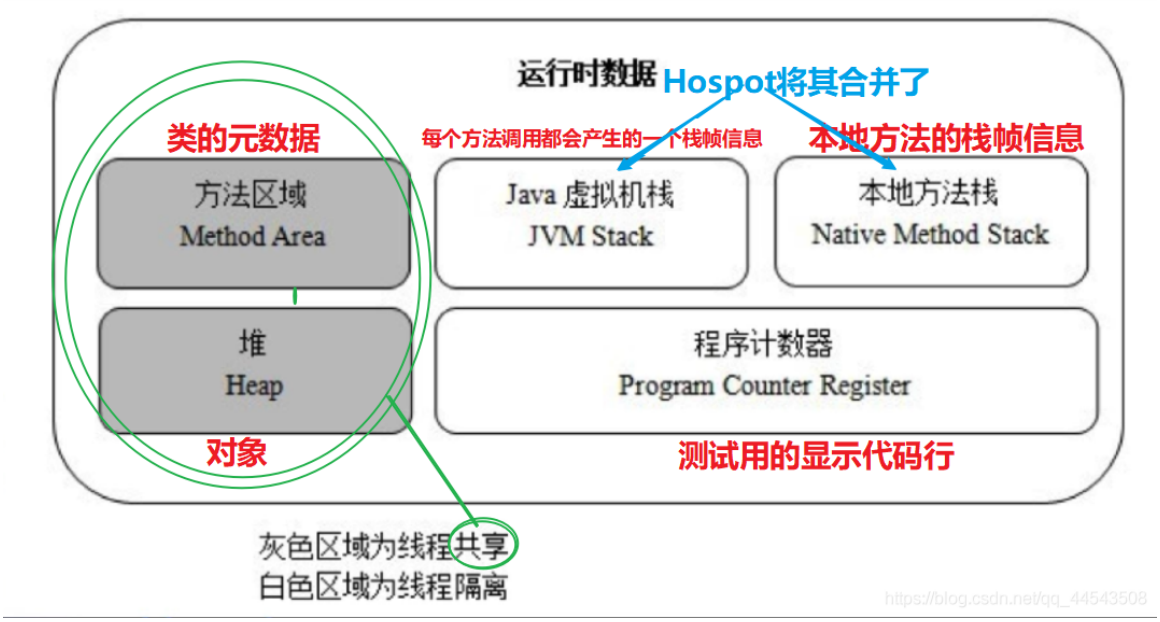
JVM被分为三个主要的子系统：类加载器子系统、运行时数据区和执行引擎 。而今天的这篇文章主要讲解其中的运行时数据区（Runtime Data Areas）



在 Java 虚拟机规范中，定义了**五种运行时数据区**，分别是 **Java 堆、方法区、虚拟机栈、本地方法区(栈)、程序计数器** ！

顺道提一句**运行时常量池也会进入方法区，也就是说方法区中就已经包括了常量池。**

特别注意其中**Java 堆和方法区是 线程共享的。其他都是 线程私有的。**



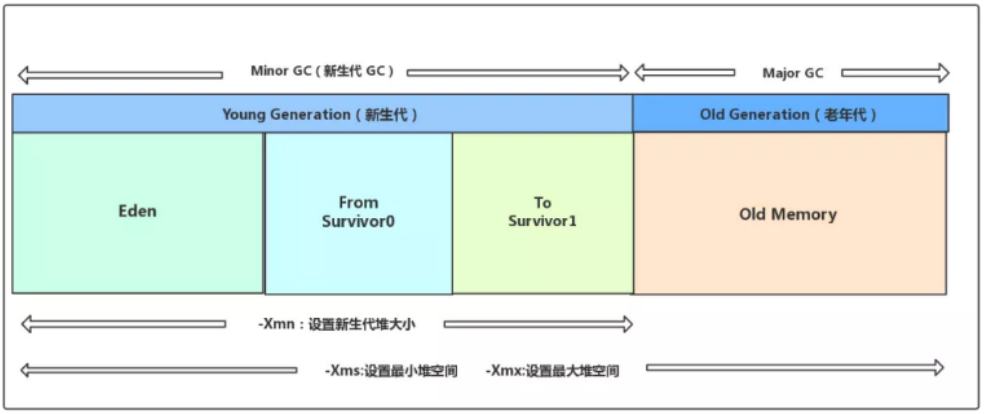
3、线程共享：Java堆、方法区

我们首先来了解了解一下线程共享的Java堆和方法区！

3.1、Java堆

Java 堆是所有线程共享的，它在虚拟机启动时就会被创建

Java 堆是内存空间占据的最大一块区域了，Java 堆是用来存放对象实例及数组，也就是说我们代码中通过 new 关键字 new 出来的对象都存放在这里。所-以这里也就成为了垃圾回收器的主要活动营地了，于是它就有了一个别名叫做 GC 堆，并且单个 JVM 进程有且仅有一个 Java 堆。根据垃圾回收器的规则，我们可以对 Java 堆进行进一步的划分，具体 Java 堆内存结构如下图所示：



堆的划分

从上图可以看出Java 堆并不是单纯的一整块区域，实际上java堆是根据对象存活时间的不同，Java 堆还被分为年轻代、老年代两个区域，年轻代还被进一步划分为 Eden 区、From

Survivor 0、To Survivor 1 区。并且默认的虚拟机配置比例是Eden：from ： to = 8:1:1 。简单来说就是：

Java堆 = 老年代 + 新生代

新生代 = Eden + S0 + S1

默认Eden：from ： to = 8:1:1

仔细看过上面的 Java 堆结构图童鞋可能会发现了-Xms和-Xmn的字样，是的这个正是控制堆的JVM的参数，实际上我们是可以通过JVM参数动态控制 Java 堆中的各空间大小的，关于JVM的参数是有很多的，但是常用的也就那么几个，不多的，用的多了都会很容易记住的，下面我们来讲讲关于堆的JVM常见的参数：

-Xms： 堆容量初始大小（堆包括新生代和老年代）。 例如：-Xms 20M
-Xmx： 堆总共（最大）大小。 例如：-Xmx 30M
注意：建议将 -Xms 和 -Xmx 设为相同值，避免每次垃圾回收完成后JVM重新分配内存!
-Xmn： 新生代容量大小。例如：-Xmn 10M
-XX： SurvivorRatio 设置参数Eden、form和to的比例 【比例参数Eden、form和to默认是8：1：1】例如：-XX： SurvivorRatio=8 代表比例8：1：1

虽然没有直接设置老年代的参数，但是可以设置堆空间大小和新生代空间大小两个参数来间接控制：
老年代空间大小 = 堆空间大小 - 年轻代大空间大小
-Xmx：最大堆大小；-Xms：初始化堆大小；-Xmn：新生代堆大小；
老年代大小 = 堆空间大小 - 新生代大小

当我们的 Java 堆内没有足够的空间去完成实例分配时，并且堆也无法扩展，将会抛出我们常见的OutOfMemoryError 异常，也就是我们常说的OOM 异常。

3.2、 JVM 堆内存溢出后，其他线程是否可继续工作?

JVM 堆内存溢出后也就是OOM 异常，网上有一道非常火的面试题：JVM 堆内存溢出后，其他线程是否可继续工作？

实际上这个问题需要具体的场景分析。但是就一般情况下，发生OOM的线程都会终结（除非代码写的太烂），该线程持有的对象占用的heap都会被gc了，释放内存。因为发生OOM之前要进行gc，就算其他线程能够正常工作，也会因为频繁gc产生较大的影响。

也就是说发生OOM的线程一般情况下会死亡，也就是会被终结掉，该线程持有的对象占用的heap都会被gc了，释放内存。因为发生OOM之前要进行gc，就算其他线程能够正常工作，也

会因为频繁gc产生较大的影响。

3.3、方法区

拿HotSpot 虚拟机来说，在 JDK1.7的时候，方法区被称作为永久代， 从JDK1.8开始，Metaspace （元空间）也就是我们所谓的方法区！

方法区（Method Area）与上面讲的Java堆一样，都是各个线程共享的，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，但是它却有一个别名叫做Non-Heap（非堆），目的应该是与Java堆区分开来。

Java虚拟机规范中是这样定义方法区的：
它存储了每个类的结构信息，例如运行时常量池、字段、方法数据、构造函数和普通方法的字节码内容，还包括一些在类、实例、接口初始化时用到的特殊方法。

3.4、JDK1.8 之前的方法区

就以HotSpot 虚拟机来说，在 JDK1.8 之前，方法区也被称作为永久代，这个方法区会发生我们常见的 java.lang.OutOfMemoryError: PermGen space 异常,注意是永久代异常信息，我们也可以通过启动参数来控制方法区的大小：

-XX:PermSize 设置方法区最小空间
-XX:MaxPermSize 设置方法区最大空间

在JDK7之前的HotSpot虚拟机中，纳入字符串常量池的字符串被存储在永久代中，因此导致了一系列的性能问题和内存溢出错误。特别突出的例子就是String的intern（）方法

3.5、JDK1.8 之后的方法区

JDK8之后就没有永久代这一说法变成叫做元空间（meta space），而且将老年代与元空间剥离。元空间放置于本地的内存中，因此元空间的最大空间就是系统的内存空间了，从而不会再出现像永久代的内存溢出错误了，也不会出现泄漏的数据移到交换区这样的事情。用户可以为元空间设置一个可用空间最大值，不设置默认根据类的元数据大小动态增加元空间的容量。对于一个 64 位的服务器端 JVM 来说，其默认的 - XX:MetaspaceSize 值为21MB。也就是说默认的元空间大小是21MB。

只要类加载器还存活，其加载的类的元数据也是存活的，不会被回收掉！也就是同生共死

3.6、JDK1.8 之后的方法区为何变化如此之大?

做这个改变呢也许主要是基于以下两点原因：

1、由于 永久代（PermGen）内存经常会溢出，引发恼人的 java.lang.OutOfMemoryError: PermGen，因此 JVM 的开发者希望这一块内存可以更灵活地被管理，不要再经常出现这样的OOM错误。

2、移除 永久代（PermGen）可以促进 HotSpot JVM 与 JRockit VM 的融合，因为 JRockit 没有永久代。

还有需要注意一点的是永久代的移除并不代表自定义的类加载器泄露问题就解决了。因此，你还必须监控你的内存消耗情况，因为一旦发生泄漏，会占用你的大量本地内存，并且还可能导致交换区交换更加糟糕。

4、线程私有：程序计数器、Java 虚拟机栈、本地方法栈

Java 堆以及方法区的数据是共享的，但是有一些部分则是线程私有的。线程私有部分可以分为：程序计数器、Java 虚拟机栈、本地方法栈三大部分。

4.1、Java 虚拟机栈（JVM Stacks)

1、 Java 虚拟机的每一条线程都有自己私有的 Java 虚拟机栈，这个 Java 虚拟机栈跟线程同时创建，所以它跟线程有相同的生命周期。

2、Java 虚拟机栈描述的是 Java 方法执行的内存模型：每一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、动态链接、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 Java 虚拟机栈中的入栈到出栈的过程。

3、局部变量表存放了编译期可知的各种基本数据类型、对象引用和 returnAddress 类型。

1. 基本类型：八种基本类型
2. 对象引用：reference 类型，它不等同于对象本身，根据不同的虚拟机实现，它可能是一个指向对象起始地址的引用指针，也可能指向一个代表对象的句柄(存放对象的引用)或者其他与此对象相关的位置。
3. 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 类型的数据会占用 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间完成分配，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。

4、Java 虚拟机栈既允许被实现成固定的大小，也允许根据计算动态来扩展和收缩，如果采用固定大小的话，每一个线程的 Java 虚拟机栈容量可以在线程创建的时候独立选定。在 Java 虚拟机栈中会发生两种异常，这个在虚拟机规范中有指出：

- 如果线程请求分配的栈容量超过 Java 虚拟机栈允许的最大容量，Java 虚拟机将会抛出 StackOverflowError 异常；也就是栈溢出错误！方法递归调用产生 StackOverflowError 异常这种结果。
- 如果 Java 虚拟机栈可以动态扩展，并且在尝试扩展的时候无法申请到足够的内存或者在创建新的线程时没有足够的内存去创建对应的 Java 虚拟机栈，那么虚拟机将会抛出 OutOfMemoryError 异常。也就是OOM内存溢出错误！(线程启动过多)

当然，可以通过参数 -Xss 去调整JVM栈的大小！

4.2、本地方法栈（Native Method Stacks)

和虚拟栈相似，只不过它服务于Native方法，线程私有。当 Java 虚拟机使用其他语言（例如 C 语言）来实现指令集解释器时，也会使用到本地方法栈。如果 Java 虚拟机不支持 natvie 方法，并且自己也不依赖传统栈的话，可以无需支持本地方法栈。

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

HotSpot虚拟机直接就把本地方法栈和虚拟机栈合二为一。

4.3、程序计数器

当前线程所执行的字节码的行号指示器，用于记录正在执行的虚拟机字节指令地址，线程私有。

需要特别注意的是，程序计数器是唯一一个在Java虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。

方法区：

1、存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码数据等，回收目标主要是常量池的回收和类型的卸载，各线程共享

2、方法区在JDK1.7的时候叫做永久代，到JDK1.8之后废弃了永久代改为元空间（meta space）