JVM架构

JAVA语言的一个特点是平台无关性，使用Java虚拟机是实现这一特点的关键。JVM( Java Virtual Machine)，是一个虚拟出来的计算机，通过在实际的计算机上仿真模拟各种计算机功能来实现的。JVM包含了两个产品：

1. JRE（Java Runtime Environment），Java运行环境，面向Java程序执行者而不是开发者，仅按照JRE，那么系统只能运行Java程序。JRE是运行Java程序所必须的环境集合，包含JVM的标准实现、Java核心类库等，但是不包含开发工具（编译器和调试器）。
2. JDK（Java Development Kit），Java的开发工具包，提供Java的开发环境：编译器javac等工具，用于将java文件编译为class文件；运行环境，提供了JVM和Runtime辅助包，用于解析class文件使其得到运行。JDK是整个Java的核心，包括了Java运行环境（JRE），一堆Java工具tools.jar和Java标准类库(rt.jar)。

Java语言使用Java虚拟机屏蔽了与具体平台相关的信息，使得Java语言编译的程序之需要生成在JVM上运行的目标代码（bytecode），就可以在多种平台上不加修改的运行。在JDK中，编译器将Java文件编译成class文件，然后将class文件输入到jvm中，JVM会加载并执行类文件，JVM架构图如下所示：



图1. JVM 架构图

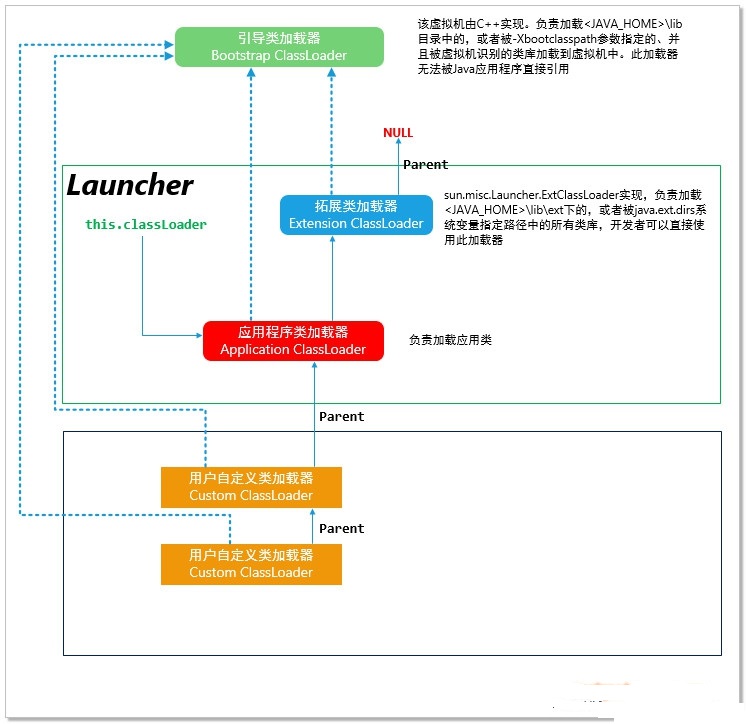
如上图所示，JVM分为三个主要子系统：

* 类加载器子系统
* 运行时数据去
* 执行引擎

下面进行详细介绍。

# 类加载器子系统(ClassLoader)

ClassLoader的作用就是将class文件加载到JVM虚拟机中，ClassLoader使用的是双亲委托模型来搜索类，每个ClassLoader都有一个父类加载器的引用（除了Bootstrap ClassLoader）。当一个ClassLoader实例需要加载某个类时，在搜索类之前，先委托给父类加载器，这个过程由上至下以此检查，如下图所示：

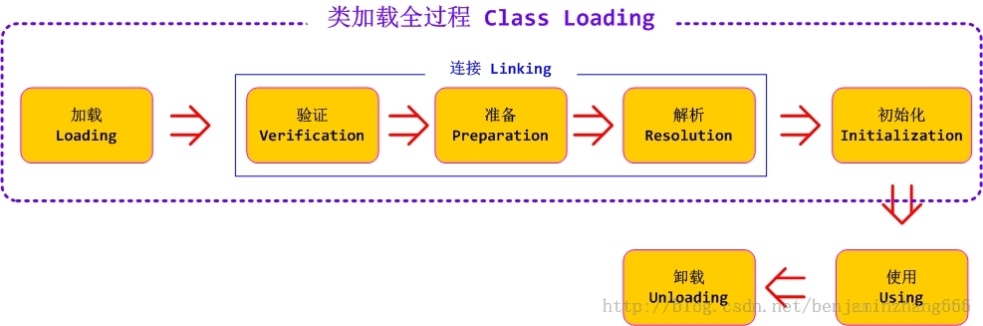


JVM中默认提供三种类加载器:

* 启动类加载器（Bootstrap ClassLoader），是Java类加载层次中最顶层的类加载，用C++实现的类加载器，是虚拟机的一部分，主要加载JDK中核心类库，如rt.jar,resources.jar等，完全由JVM控制，开发者无法访问
* 扩展类加载器（Extension ClassLoader），负责加载{JRE\_HOME}/lib/ext目录或者被java.ext.dirs系统变量指定的路径
* 应用程序加载器（App ClassLoader），负责加载classPath路径上指定的类库，如果程序中没有定义类加载器，一般作为默认的类加载器

用户还可以根据需要定义自己的ClassLoader，这些自定义的ClassLoader都必须继承java.lang.ClassLoader类，也包括Java提供的Extension ClassLoader和App ClassLoader在内，但是Bootstrap ClassLoader不继承ClassLoader，一般用户定义ClassLoader时覆盖其中的findClass方法。

JVM把class文件加载到内存，并对数据进行校验、解析和初始化，最终形成JVM可以直接使用的Java对象组件的全过程如下所示：



主要步骤如下：

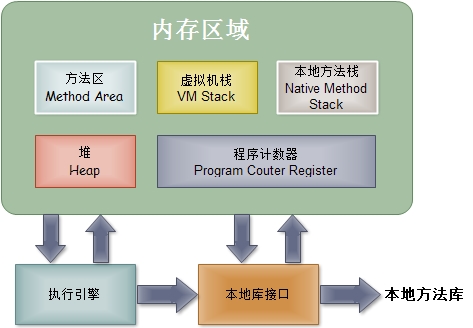
* 装载(Loading)，查找和导入class文件
* 链接(Linking)，执行校验、准备和解析步骤，解析步骤可以选择：

1. 校验，检查载入Class文件数据的正确性
2. 准备，给类的静态变量分配存储空间
3. 解析，将符号引用转成直接引用

* 初始化，对类的静态变量、静态代码块执行初始化工作。

# 运行时数据区

运行时数据区分为5个主要组件：

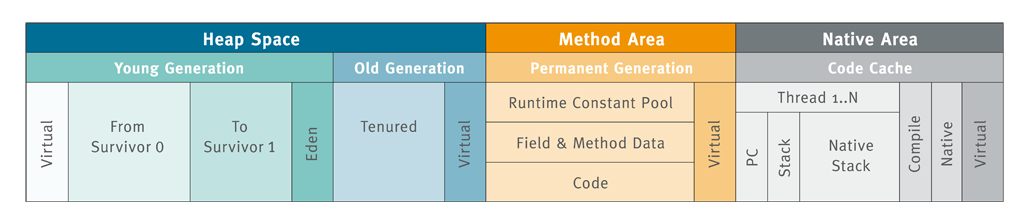


1. 方法区， 所有的类级别的数据都存储在这里，包括静态变量，每个JVM只有一个方法区，并且它是共享资源。类加载器将数据加载到这个数据区中。
2. 堆区域，所有对象及其对应的实例便利和数组都存储在这里，每个JVM只有一个堆区域，由于方法和堆区域共享多个线程的内存，因此所存储的数据非线程安全。
3. 堆栈区，JVM为每个线程都会创建一个单独的运行时栈，对于每个方法调用，将在堆栈存储器中产生一个条目，成为堆栈帧。所有局部变量将在堆栈内存中创建，堆栈区域是线程安全的，不是共享资源。堆栈帧分为三个子元素：

* 局部变量数组，与方法相关，涉及局部变量以及将在此存储的相应值得多少
* 操作数堆栈，如果需要执行任何中间操作，那么操作数堆栈充当工作空间来执行操作
* 帧数据，对应于方法的所有符号存储在此处，在任何异常的情况下，捕捉块信息都被保持在帧数据中。

1. PC寄存器，每个线程都有单独的PC寄存器，用于保存当前执行的指令地址，一旦指令执行，PC寄存器将更新到下一条指令
2. 本地方法堆栈，本地方法堆栈保存本地方法信息，对于每个线程，将创建一个单独的本地方法堆栈

对于JVM，方法区和堆区域是内存分配和管理的主要区域，内存模型如下图：



其中Perm区域为永久代（方法区），GC不会在住程序运行期间对这个区域进行清理，在JDK 1.8之后，取而代之的是元数据区Metaspace。

所有的对象在实例化后的整个运行周期内，都被存放在堆内存中，JVM中为了对短周期和长周期对象进行管理，采用分代的策略：

* **年轻代**是所有新对象生成的地方。当年轻代内存空间被用完后，会触发垃圾回收，这个垃圾回收称为Minor GC，这个区域内存相对会比较小，Minor GC会比较频繁。年轻代由分为1个Eden Space和2个Suvivor Space，默认比例为8:1。
* **Eden Space**，在大多数情况下，对象在新生代Eden区中分配，当Eden区没有足够的区间区分配时，JVM发起一次Minor GC，将Eden区和其中一块Survivor 0区内尚存活的对象放入另一块Survivor 1区域。Eden Space作为使用空间。
* **Survivor**，对于GC来讲，Survivor相当于保留空间。如果没有Survivor，Edit区每进行一次Minor GC，存活的对象就会被送到老年代，老年代很快被填满，触发Major GC(Full GC)，这个过程执行时间较长，影响程序的执行和相应速度。Survivor存在的意义在于减少Full GC发生。Survivor区又分为From Suvivor 0和To Suvivor 1两个区，存在两个Suvivor区的意义在于解决内存碎片化的问题

垃圾回收机制是Java语言的一个优势，在JVM中垃圾回收由GC(守护进程)来实现，其根据内存使用情况自动运行。由于GC需要消耗一定资源和时间，采用分代的方式进行对象收集，按照新生代、旧生代对对象进行收集，尽可能的缩短GC对应用造成的暂停。GC对垃圾的回收分为：

1. **Partitial GC，不收集整个GC堆得模式**

* Minor GC（Young GC），当Eden区没有足够的区间区分配时，JVM发起一次Minor GC，将Eden区和其中一块Survivor 0区内尚存活的对象放入另一块Survivor 1区域
* Old GC，当年老代内存被占满时进行Old GC，清理老生代，只有CMS的Concurrent Collection是这个模式。
* Mixed GC，收集整个young gen以及部分old gen GC，目前只有G1有这个模式。

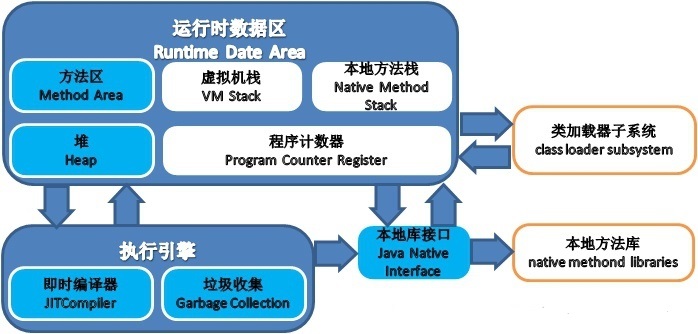
1. **Full GC，对整个堆进行GC，当出现以下情况触发Full GC**

* System.gc方法调用
* 老生代空间不足
* 永生区空间不足
* CMS GC时出现promotion failed和concurrent mode failure一次。Concurrent Mode Failure时在执行CMS GC的过程中同时又对象要放入老生代，而老生代空间不足造成的。
* 堆中分配很大的对象，该对象直接放入老生代，但是老生代没有足够的连续空间放置该对象

Major GC通常和Full GC是等价的，收集整个GC堆.

# 执行引擎

分配给运行时数据区的字节码由执行引擎(解释器)执行，执行引擎读取字节码并逐个执行，原理图如下所示:



Java程序经过编译器编译成字节码，而不是编译成某个特定处理器硬件平台对应的指令代码,执行引擎的作用如上图所示从运行时数据区读取字节码，然后转换成机器指令，通过本地库接口调用物理机器。执行引擎的核心JITCompiler(Just-in-time Compiler)，编译器编译整个字节码并将其更改为本地代码，整个本地代码将直接用于重复的调用，主要过程如下：

1. 中间代码生成器，生成中间代码
2. 代码优化器，负责优化上面生成的中间代码
3. 目标代码生成器，负责生成机器代码或本地代码
4. 分析器，一个特殊组件，负责查找热点，即该方法是否被多次调用

以上过程涉及到底层调用，不再详述。

# 参考文献

JVM架构：http://www.codeceo.com/article/jvm-architecture-explained.html