链滴

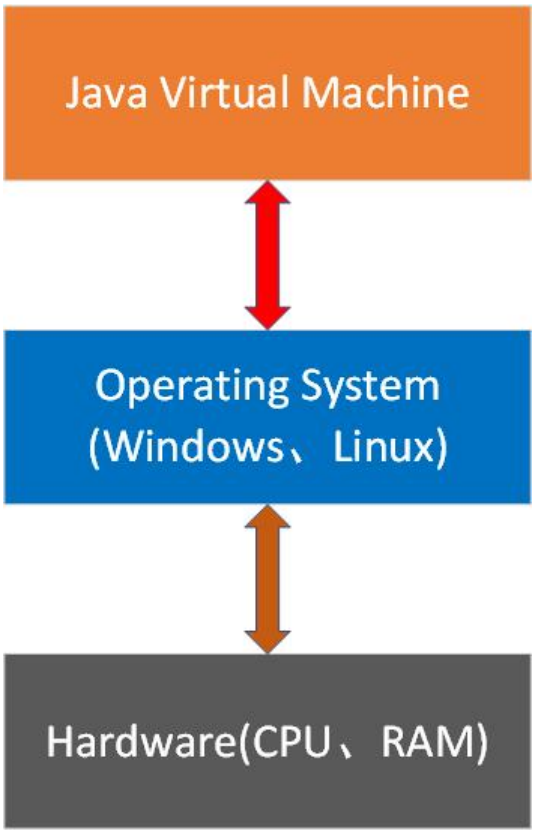
JVM 架构的基本介绍

作者：Mckaymk

原文链接：https://ld246.com/article/1575810940172

来源网站：链滴

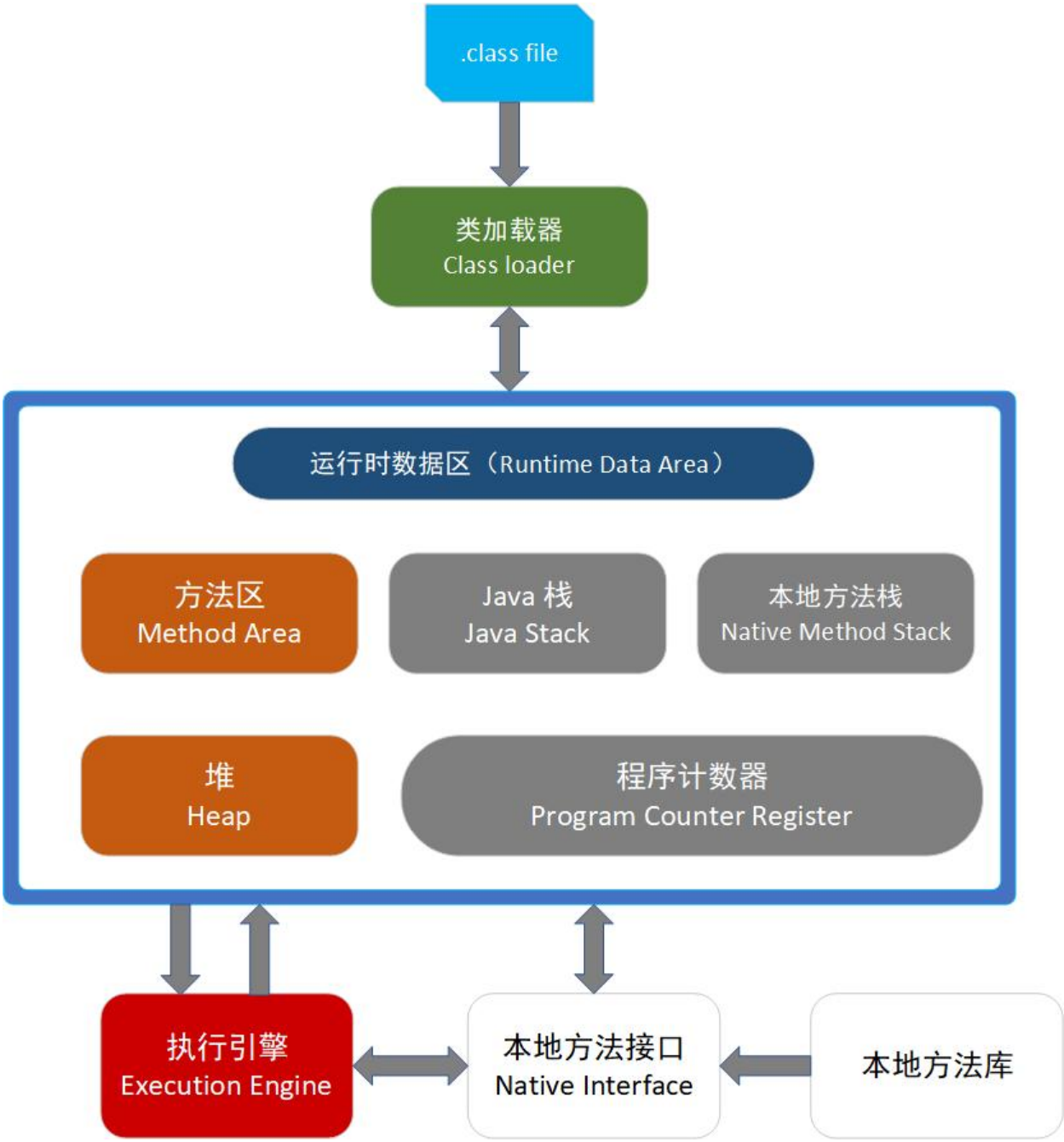
1、JVM运行环境



由上图可知，jvm是运行在操作系统之上。java程序运行在jvm之上，而jvm所提供的环境是一样的，以体现出java的一次编译，到处运行的特性。

2、JVM体系架构

原文链接：JVM 架构的基本介绍



JVM主要由上图的几大部分组成，下面将依次简单介绍。

3、类加载器（Class Loader）

类加载器主要负责加载java的class文件，class文件在文件头有一些特定的文件标识符（cafe babe等，将class文件字节码类容加载到内存中，并将这些内容转换成方法区中运行时数据结构，并且ClassLder只负责Class文件的加载，至于后面的运行则是由执行引擎（Execution Engine）决定。方法区（ethod Area）放类的基本信息（类的模版）。

虚拟机自带的三个加载器：

- 启动类加载器（BootStrap）C++，加载一些rt.jar等java运行时所需要的一些基本类，java.lang.Strng等，在jre1.8.0_211\lib\文件夹下。
- 扩展类加载器（Extension）Java，加载一些随着java更新而新增的扩展包，如javax下的包，在jre1

8.0_211\lib\ext下的jar包中。

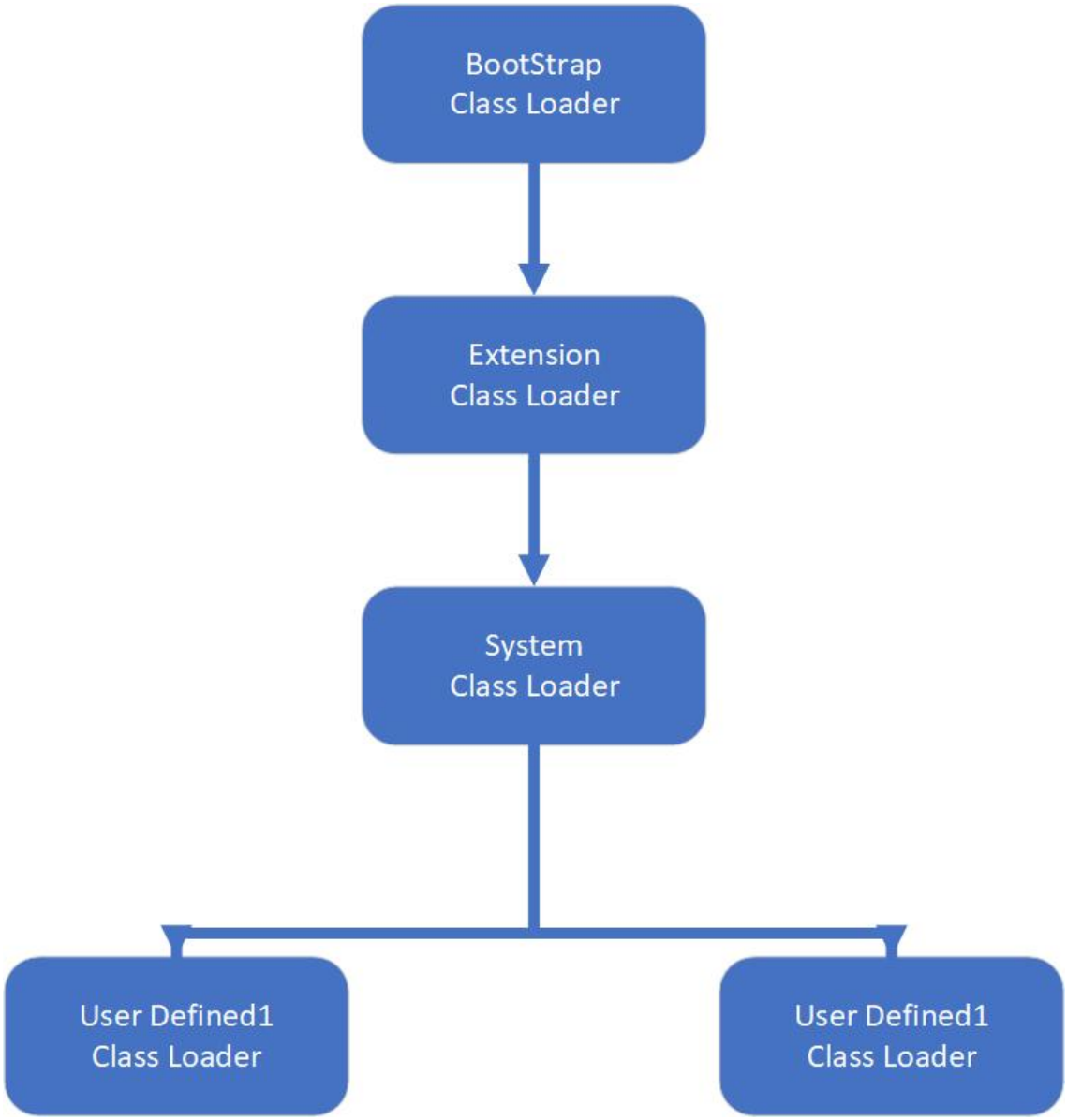
- 应用程序加载器（AppClassLoader），也叫系统加载器，加载当前应用的classpath的所有类。
- 用户自定义加载器，java.lang.ClassLoder的子类，用户可以定制类的加载方式，一般不使用。

```
/**
 * @author Mckay
 * @create 2019-12-06
 * @description 类加载器测试
 **/
public class TestDemo {
    public static void main(String[] args) {
        Object object = new Object();
        //输出null，因为Object为顶层对象，由根加载器加载，所以无法获得加载它的对象
        System.out.println(object.getClass().getClassLoader());
        TestDemo testDemo = new TestDemo();
        //输出sun.misc.Launcher$AppClassLoader@18b4aac2，因为这是用户自定义类，由应用程
类加载器加载
        System.out.println(testDemo.getClass().getClassLoader());


        //输出null，说明根加载器不是java中的加载器，由c++定义。在java中获取不到该对象
        System.out.println(testDemo.getClass().getClassLoader().getParent().getParent());
        //输出sun.misc.Launcher$ExtClassLoader@2a84aee7，说明AppClassLoade，由ExtClassLo
der扩展类加载器加载
        System.out.println(testDemo.getClass().getClassLoader().getParent());
        //输出sun.misc.Launcher$AppClassLoader@18b4aac2，说明自定义类由AppClassLoader
载器加载
        System.out.println(testDemo.getClass().getClassLoader());

    }
}
```

类加载的双亲委派机制：如下图所示，Java的类加载一般从顶层开始查，依次向下，直到找到为止，从Bootstrap根加载器开始依次向下找，如果没找到，则抛出异常，ClassNotFoundException。这可以保证顶层类的定义不会受到应用程序类定义的干扰，从而实现了沙箱安全。




```
package java.lang;
/**
 * @author Mckay
 * @create 2019-12-06
 * @description 测试类加载机制
 **/
public class String {
    public static void main(String[] args) {
        //报错，因为顶层String类中没有main方法
        System.out.println("hello");
    }
}
```

总结：双亲委派机制指的是当一个类需要被加载时，自己不会直接去加载这个类，而是把这个请求传父类加载器去完成，每层的类加载器都会如此，所以所有的类加载请求都会被传到根类加载器（Bootstrap），而当根加载器无法完成该任务时，即无法在自己的加载路径内找到需要加载的类（Class），类加载器就会去尝试去完成，然后依次向下。这样可以优先保证最上层定义的类会被优先加载，保证定义的唯一性。

4、执行引擎（Execution Engine）

JVM的执行引擎负责执行由类加载器加载的Class字节码文件。方法的字节码流由一系列有序指令组，指令又由一个单字节的操作码 + 0个或多个操作数组成。操作码表示需要执行的操作，操作数表示作的数据，一般来源于当前栈桢中的局部变量或当前Java栈桢中操作数栈的顶部，操作数的个数，由作码决定（操作码本身就决定了它是否需要操作数，以及操作数的形式等等）。

5、本地方法区（Native Method Stack）

native：是一个关键字，只有方法声明，没有方法实现，表示该方法需要加载系统的方法，或者其他三方语言的方法，如c,c++等。

所以一个方法由native声明，则表明该方法由java之外提供，即存在于本地方法栈区，由执行引擎从地方法接口（Native Interface）中加载，放在本地方法栈区（Native Method Stack）。例如：线调用start方法后，调用的是系统的start0()方法，并且线程并不立即启动，而是进入就绪状态，等待系统CPU调度执行。

6、程序计数器（Program Counter Register）

寄存器：一个指针，指向程序下一步运行的地址，记录了方法之间的调用和执行情况。

每个线程都有一个程序计数器，是线程私有的，就是一个指针，指向方法区中的方法字节码（用来存指向下一条的地址，也即将要执行的指令代码），由执行引擎取下一条指令，是一个非常小的内存空，几乎可以忽略不记下。

在JVM中，这块内存区域很小，它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个计数器的值来选取下一条需要执行的字节码指令。如果执行的是一个Native方法，这个计数器是空。

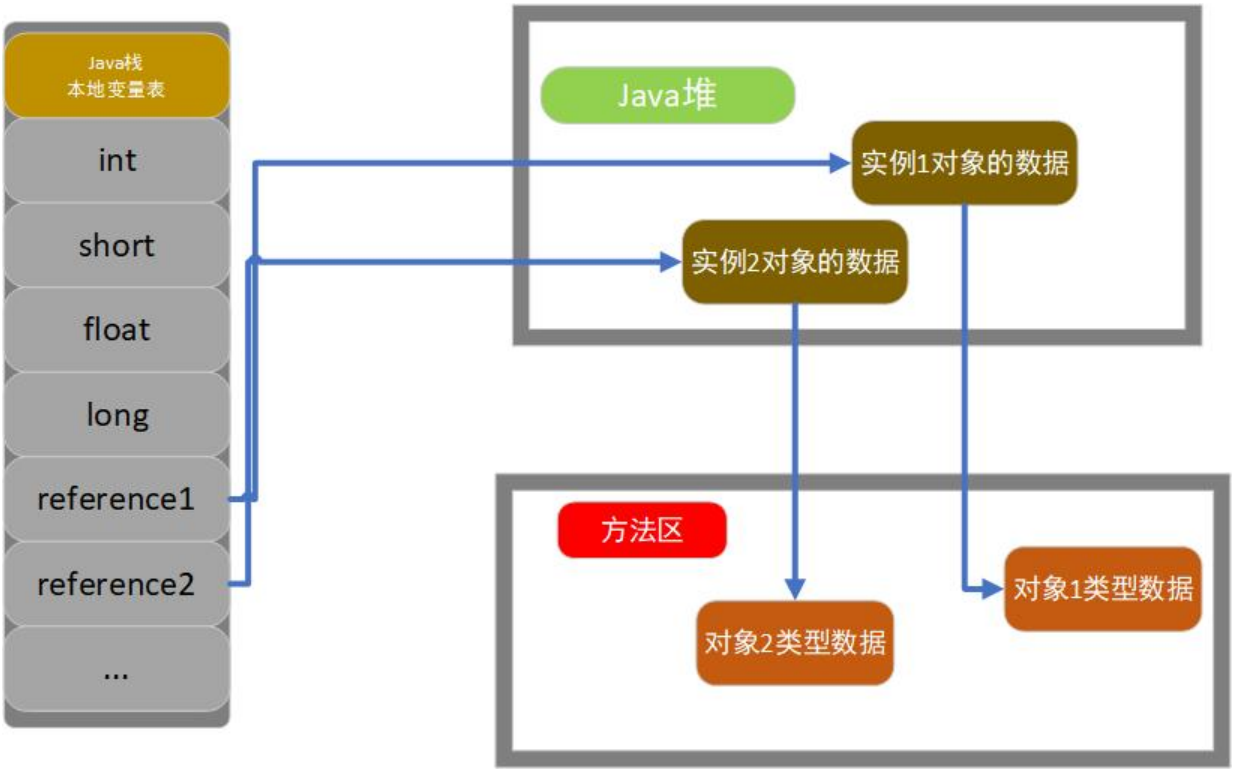
可以完成分支、循环、跳转、异常处理、线程恢复等基础功能，不会发生内存溢出（OutOfMemory错误。

7、Java栈（Java Stack）

为线程私有，没有垃圾回收机制。栈也叫栈内存，主管java程序的运行，是在线程创建时创建，它的命周期是跟随线程的生命周期，线程结束栈内存也就释放了，对于栈来说，不存在垃圾回收问题，只线程结束，线程对应的栈也会全部被销毁。生命周期和线程一致，为线程私有。java方法放入栈内被为栈帧。栈的存储内容：8种基本类型的变量+对象的引用变量+实例方法都是在函数的栈内存中分配栈中的数据都是以栈帧（Stack Frame）的格式存在，栈帧是一个内存区快，是一个数据集，是一个关方法（Method）和运行期数据的数据集。

栈帧中主要保存三类数据：

- 本地变量（Local Variables）：输入参数和输出参数以及方法内的变量；
- 栈操作（Operand Stack）：记录出栈、入栈的操作；
- 栈帧数据（Frame Data）：包括类文件、方法等等。



以上为栈、堆和方法区三者之间的关系。HotSpot是使用指针的方式来访问对象。Java堆中会存放访类元数据的地址，reference存储的就是直接的对象的地址。

8、堆（Heap）

以上为java堆中的逻辑逻辑分区，需要注意的是，最后一块，在java7中是永久区，在java8被改成元间。

一个JVM实例只存在一个堆内存，堆内存的大小是可以调节的。类加载器读取了类文件后，需要把类方法、常变量放到堆内存中，保存所有引用类型的真实信息，以方便执行器去执行。

以上是对JVM的基本介绍，如有错误，欢迎指正，后面随着学习在对各个部分进行更深入的了解。

原文链接：JVM 架构的基本介绍