**关于Java虚拟机二三事（八）---JVM机器指令集及其执行引擎**

2018-05-12 17:56:50 [Q蛋黄酱](https://me.csdn.net/u012961566) 阅读数 487 收藏 更多

分类专栏： [Java基础知识，JVM](https://blog.csdn.net/u012961566/category_6978565.html)

版权声明：本文为博主原创文章，遵循 [CC 4.0 BY-SA](http://creativecommons.org/licenses/by-sa/4.0/" \t "_blank) 版权协议，转载请附上原文出处链接和本声明。

本文链接：<https://blog.csdn.net/u012961566/article/details/80293082>

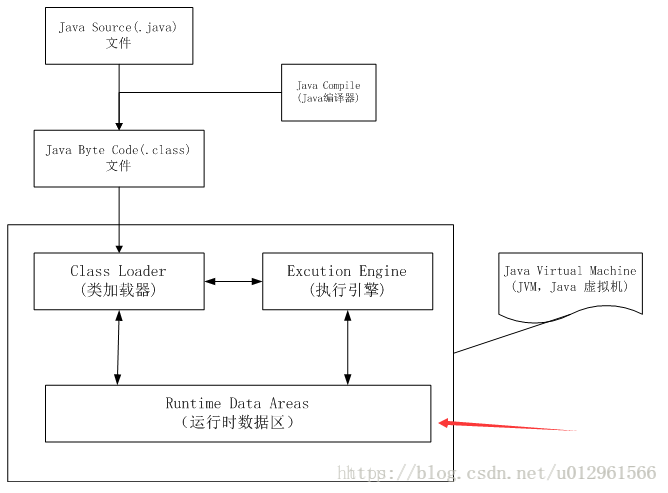
**1.前言**

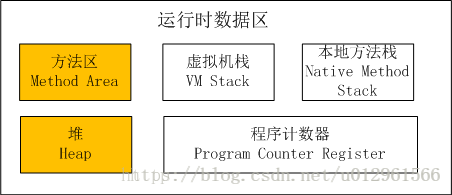
    Java虚拟机和真实的计算机一样，运行的都是二进制的机器码；而我们将.java源码编译成.class文件，class文件便是Java虚拟机能够认识的二进制机器码。Java能够识别class文件中的信息和机器指令，进而执行这些机器指令。那么，Java虚拟机是如何运行这些二进制的机器码？本文将就此问题进行讲解。

**2.Java虚拟机的运行时数据区**

    在讲解本节知识之前，先来回顾一下Java运行时数据区的结构。

    Java虚拟机的运行时数据区可分为：方法区、虚拟机栈、本地方法栈、堆、程序计数器共同组成。如下图所示。可参考[关于Java虚拟机二三事（一）---Java技术体系及JVM内存结构](https://blog.csdn.net/u012961566/article/details/78875491)。





    我们把重点放在虚拟机栈。

**3.Java虚拟机对运行时虚拟机栈的组织**

    Java虚拟机在运行时，会在内存中为每个线程分配一个虚拟机栈，来表示线程运行状态和信息，虚拟机栈中的元素称之为栈帧（JVM stack frame），每个栈帧表示对一个方法的调用信息。



**4.方法调用过程在JVM中是如何表示的**

    首先看如下例子。

1. /\*\*
2. \* JVM 原理简单用例
3. \* @author louis
4. \*
5. \*/
6. public class Bootstrap {
8. public static void main(String[] args) {
9. String name = "Louis";
10. greeting(name);
11. }
13. public static void greeting(String name)
14. {
15. System.out.println("Hello,"+name);
16. }
18. }

    当我们将Bootstrap.java编译成Bootstrap.class并运行的时候，在JVM复杂的运行逻辑中，会有以下几步：

    1. 首先JVM会将这个Bootstrap.class信息加载到内存中的方法区（Method Area）。

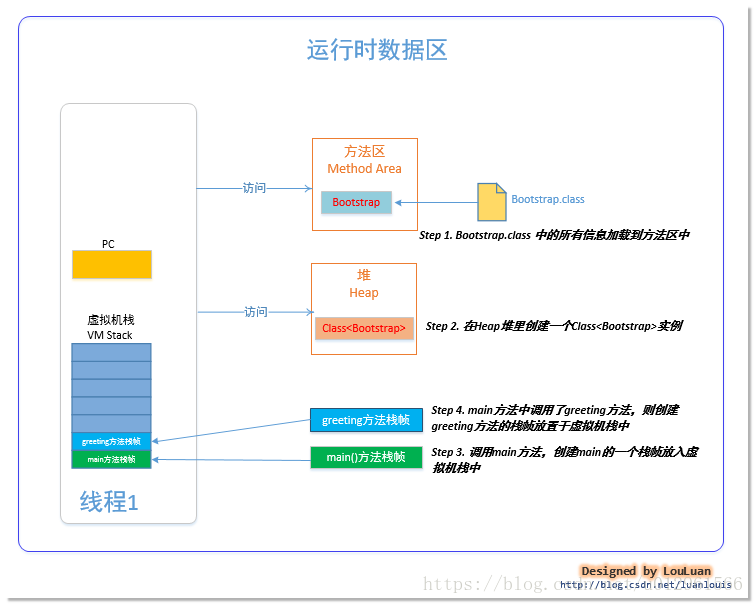
        Bootstrap.class中包含了常量池信息，方法的定义以及编译后的方法实现的二进制形式的机器指令，所有线程共享一个方法区，从中读取方法的定义和指令集。

    2. 接着，JVM会在Heap堆上为Bootstrap.class创建一个Class<Bootstrap>实例来表示Bootstrap.class的类实例。

    3. JVM开始执行Main方法，这时，JVM会为Main方法创建一个栈帧，以表示Main方法的整个执行过程。

    4. Main方法在执行的过程中，调用了greeting静态方法，则JVM会为greeting方法创建一个栈帧，推到虚拟机栈的栈顶。

    5. 当greeting方法运行完成之后，则greeting方法出栈，Main方法继续运行。



    JVM方法的调用是用栈帧来实现的，而方法的指令又是如何运行的呢？

    我们知道，class文件是JVM能够识别的二进制文件，其中通过特定的结构描述了每个方法的定义。

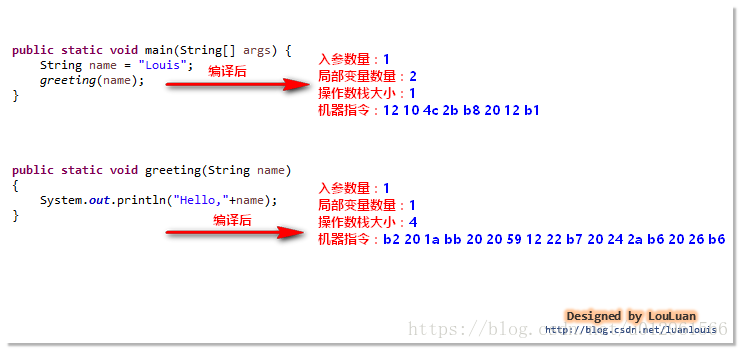
    JVM在编译Bootstrap.java的过程中，在将源代码编译成二进制机器码的同事，会判断其中每个方法的三个信息：

        1） 在运行时会使用到的局部变量的数量（作用：当JVM为方法创建栈帧的时候，在栈帧中Wie该方法创建一个局部变量表，来存储方法指令在运算时的局部变量值）

        2） 其机器指令执行时所需要的最大操作数栈的大小（当JVM为方法创建栈帧的时候，在栈帧中为方法创建一个操作数栈，保证方法内指令可以完成工作）

        3）方法的参数数量

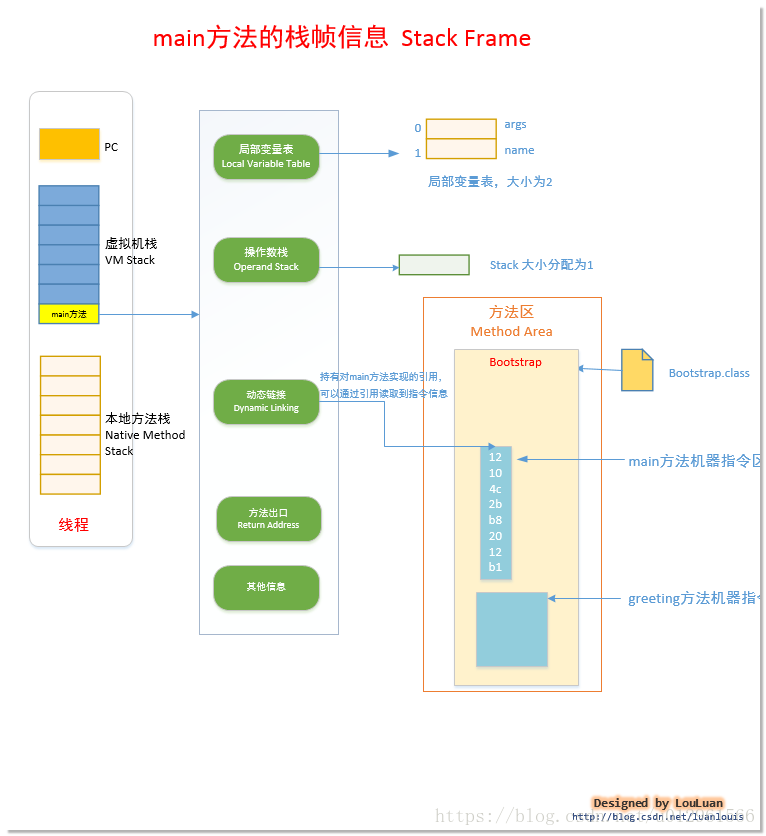
    经过编译之后，我们可以得到Main方法和greeting方法的信息如下：



        JVM运行Main方法的过程

        1. 为Main方法创建栈帧

            JVM解析Main方法，发现其局部变量的数量为2，操作数栈的数量为1，则会为main方法创建一个栈帧，并将其加入虚拟机栈中。



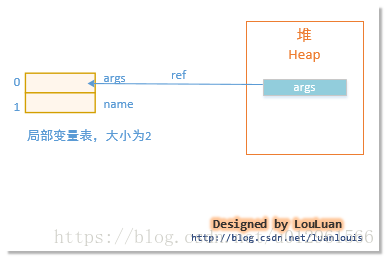
    2. 完成栈帧初始化

        main栈帧创建完成之后，会将栈帧push到虚拟机栈中，现在有两步重要的事情要做：

        1）**计算pc值**。PC是指令计数器，其内部的值决定了JVM下一步应该知晓哪一个机器指令，而机器指令存放在方法区，我们需要让PC的值指向方法区的main方法上

        初始化PC = main方法在方法区指令的地址 + 0

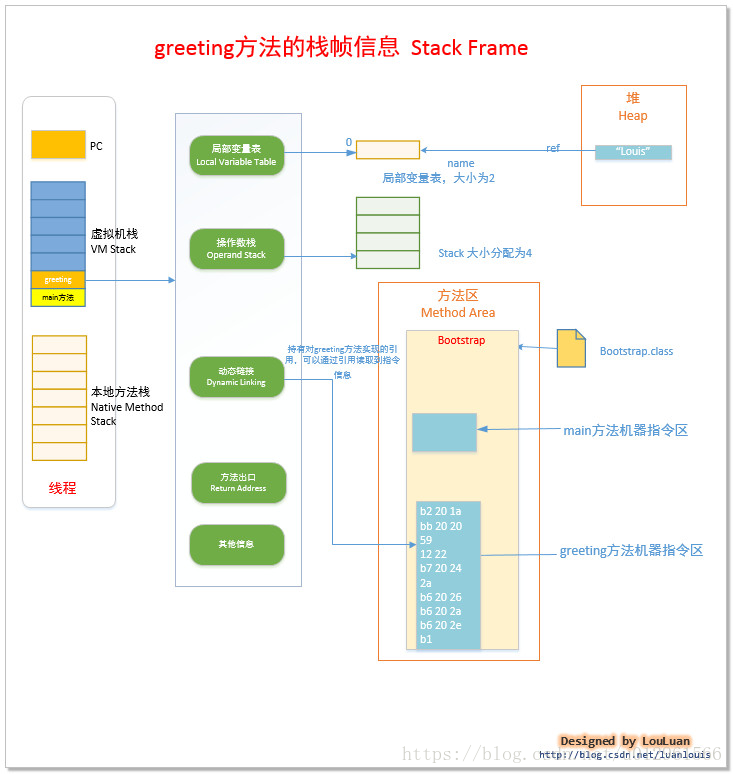
        2）**局部变量的初始化**。main方法有个入参（String[] args），JVM已经在main所在的栈帧的局部变量表中为其空出来了一个slot，我们需要将args的引用值初始化到局部变量表中。



        接着JVM开始读取PC指向的机器指令。如上图所示，main方法的指令序列：**12 10 4c 2b b8 20 12 b1** ，通过JVM虚拟机指令集规范，可以将这个指令序列解析成以下Java汇编语言:

|  |  |  |  |
| --- | --- | --- | --- |
| **机器指令** | **汇编语言** | **解释** | **对栈帧的影响** |
| 0x12 0x10 | ldc #16 | 将常量池中第16个常量池项引用推到操作数栈栈顶。 常量池第16项是CONSTANT\_UTF-8\_INFO项，表示”Louis”字符串 | 这里写图片描述 |
| 0x4c | astore\_1 | 操作数栈的栈顶元素出栈，将栈顶元素的值赋给index=1 的局部变量表元素上。  这里等价于：name = “Louis”. | 这里写图片描述 |
| 0x2b | aload\_1 | 将局部变量表中index=1的元素的值推到操作数栈栈顶 | 这里写图片描述 |
| 0xb8 0x20 0x12 | invokestatic #18 | **0xb8**表示机器指令invokestatic,操作数是0x20 << 8| 0x12 = 18，操作数18表示指向常量池第18项，该项是main方法的符号引用： org/louis/jvm/codeset/Bootstrap.greeting:(Ljava/lang/String;)V  当JVM执行这条语句的时候，会做以下几件事： **a).方法符号引用校验。**会校验这个方法的符号引用，按照这个符号规则 在常量池中查找是否有这个方法的定义，如果找到了此方法的定义，则表示解析成功。如果是方法greeting:(Ljava/lang/String;)V没有找到，JVM会抛出错误NoSuchMethodError **b).为新的方法调用创建新的栈帧。**然后JVM会为此方法greeting创建一个新的栈帧(VM stack)，并根据greeting中操作数栈的大小和局部变量的数量分别创建相应大小的操作数栈；然后将此栈帧推到虚拟机栈的栈顶。 **c).更新PC指令计数器的值。**将当前PC程序计数器的值记录到greeting栈帧中，当greeting执行完成后，以便恢复PC值。更新PC的值，使下一条执行的指令地址指向greeting方法的指令开始部分。 这条语句会使当前的main方法执行暂停，使JVM进入对greeting方法的执行当中当greeting方法执行完成后，才会恢复PC程序计数器的值指向当前下一条指令。 |  |
| 0xb1 | return | 返回 |  |
|  |  |  |  |

    当main方法调用greeting（）时，JVM会为greeting方法创建一个栈帧，用一下表示对greeting方法的调用，具体栈帧信息如下。



具体的greeting方法的机器码表示的含义如下图所示：

|  |  |  |  |
| --- | --- | --- | --- |
| **机器指令** | **汇编语言** | **解释** | **常量池引用** |
| b2 20 1a | getstatic     #26 | 获取指定类的静态域，并将其值压入栈顶. 将常量池中的第26个符号引用推到操作数栈中： | #26： // Field java/lang/System.out:Ljava/io/PrintStream; |
| bb 20 20 | new           #32 | 创建一个对象，并将其引用值压入栈顶。 创建一个java/lang/StringBuider实例,将其压入栈顶。 | #32: // class java/lang/StringBuilder |
| 59 | dup | 复制操作数栈栈顶的值，并插入到栈顶 |  |
| 12 22 | ldc           #34 | 从运行时常量池中提取数据推入操作数栈 将“Hello” String引用复制到 操作数栈中 | #34: // String Hello, |
| b7 20 24 | invokespecial #36 | 调用超类构造方法，实例初始化方法，私有方法。 此处调用StringBuilder(String)构造方法，并将结果推到栈顶 | #36:  // Method java/lang/StringBuilder."<init>":(Ljava/lang/String;)V |
| 2a | aload\_0 | 将第一个局部变量的引用推到栈顶。 当前局部变量表的第一个局部变量引用是 ：“Louis”，即将Louis推到栈顶 |  |
| b6 20 26 | invokevirtual #38 | 调用超类构造方法，实例初始化方法，私有方法。 StringBuilder实例的 append(String ) 方法，表示： "Hello,"+"Louis". | // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder; |
| b6 20 2a | invokevirtual #42 | 调用超类构造方法，实例初始化方法，私有方法。 调用StringBuilder实例的toString()方法，结果保留在栈顶。 | // Method java/lang/StringBuilder.toString:()Ljava/lang/String; |
| b6 20 2e | invokevirtual #46 | 调用超类构造方法，实例初始化方法，私有方法。 调用System.out.println(String)方法 | // Method java/io/PrintStream.println:(Ljava/lang/String;)V |
| b1 | return | 结束返回 |  |
|  |  |  |  |

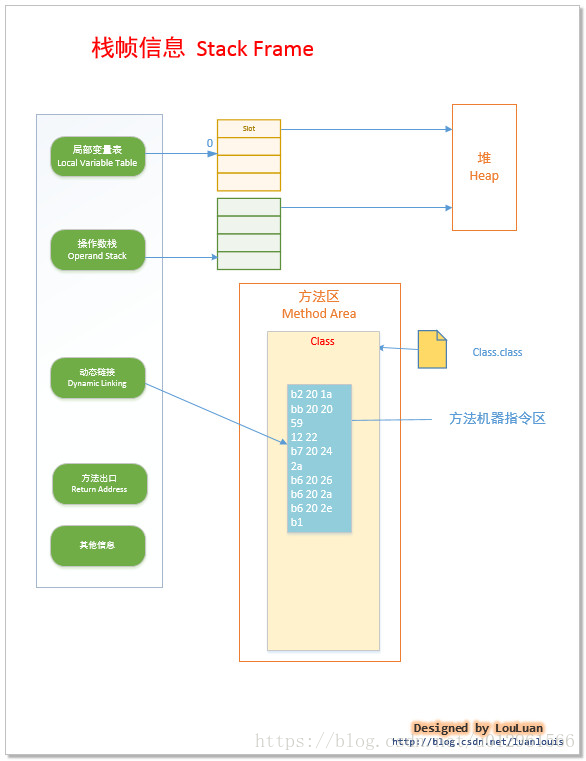
**5.JVM对一个方法执行的基本策略**

    一般的，对于Java方法的执行，在JVM在其某一特定线程的虚拟机栈中会为方法分配**一个局部变量表、一个操作数栈、用以存储方法运行过程中的中间值存储**。

    由于JVM的指令是基于栈的，即大部分的指令的执行，都伴随着操作数的出栈和入栈。所以在学习JVM的机器指令的时候，一定要注意：

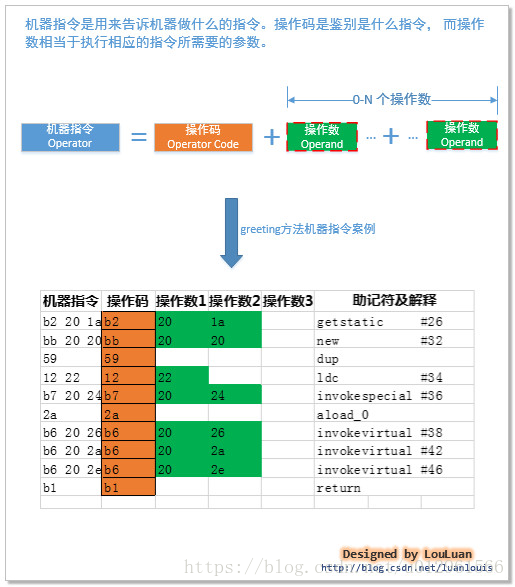
        每个机器指令的执行，对操作数栈和局部变量的影响，充分地了解了这个机制，你就可以非常顺畅地读懂class文件中的二进制机器指令了。

    如下是栈帧信息的简化图，在分析JVM指令时，脑海中对栈帧有个清晰的认识：



**6.机器指令的格式**

    所谓的机器指令，就是只有机器才能够认识的二进制代码，一个机器指令分为两部分组成



            注：

                a).  如上图所示JVM虚拟机的操作码是由一个字节组成的，也就是说对于JVM虚拟机而言，其指令的数量最多为 2^8,即 256个;

                b). 上图中的操作码如:b2,bb,59....等等都是表示某一特定的机器指令，为了方便我们识别，其分别有相应的助记符：getstatic,new,dup.... 这样方便我们理解。

**7.机器指令的执行模式——基于操作数栈的模式**

    对于传统的物理机而言，大部分的机器指令的设计都是寄存器模型的，物理机内设置若干个寄存器，用以存储机器指令运行过程中的值，寄存器的数量和支持的指令的数量决定了这个机器的处理能力。

    但Java虚拟机的设计机制并不是这样，Java虚拟机使用操作数栈来存储机器指令的运算过程中的值，所有的操作数的操作，都要遵循出栈和入栈规则，所以在《Java虚拟机规范》中，有很多机器指令都是关于出栈和入栈操作 。

