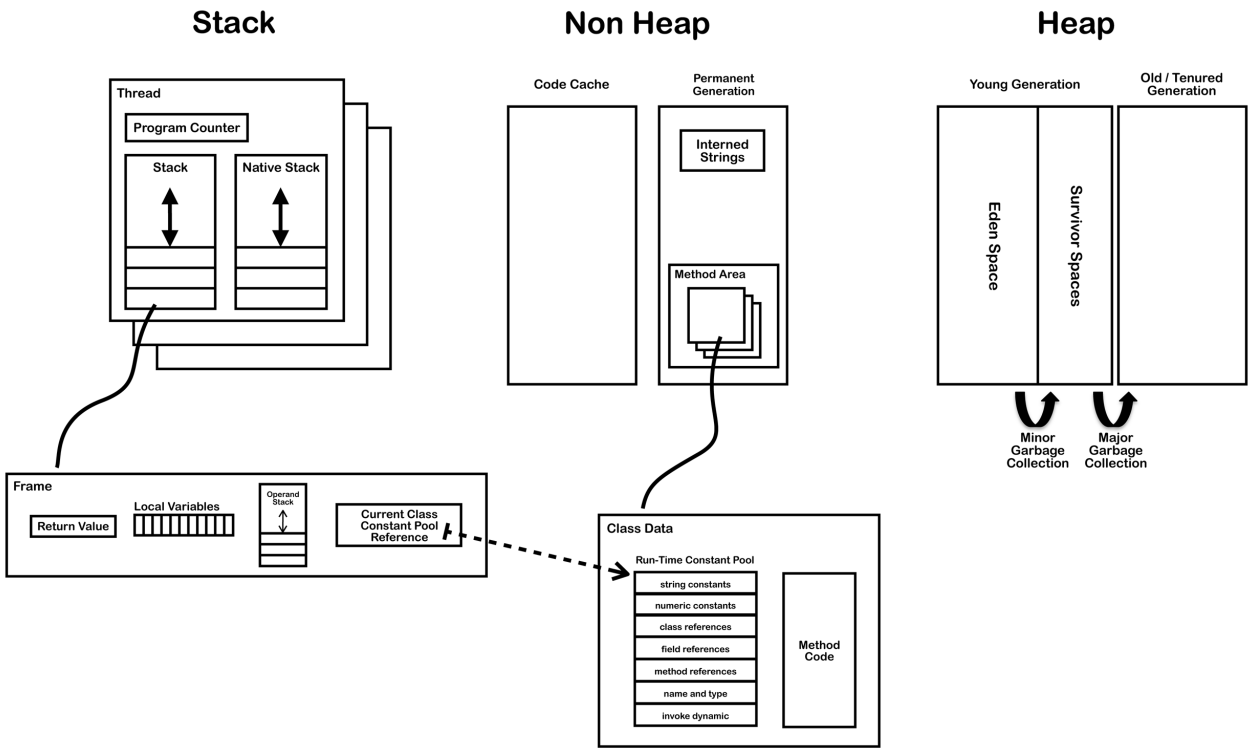
JVM 内部结构

2013/11/24

本文介绍了 Java 虚拟机 (JVM) 的内部架构。下图显示了符合Java 虚拟机规范 Java SE 7 版本的典型 JVM 的关键内部组件。



下面分两部分对该图所示的组件进行了说明。 第一部分介绍为每个线程创建的组件，第二部分介绍独立于线程创建的组件。

- 线程数
  - JVM 系统线程
  - 每线程
  - 程序计数器（PC）
  - 堆
  - 原生堆栈
  - 堆栈限制
  - 框架

线

线程是程序中执行的线程。JVM 允许应用程序同时运行多个执行线程。在 Hotspot JVM 中，Java 线程和本机操作系统线程之间存在直接映射。为 Java 线程准备好所有状态（例如线程本地存储、分配缓冲区、同步对象、堆栈和程序计数器）后，将创建本机线程。一旦 Java 线程终止，本机线程就会被回收。因此，操作系统负责调度所有线程并将它们分派到任何可用的 CPU。一旦本机线程初始化，它就会调用 Java 线程中的`run()`方法。当运行（）方法返回，处理未捕获的异常，然后本机线程确认是否由于线程终止而需要终止 JVM（即它是最后一个非守护线程）。当线程终止时，本机线程和 Java 线程的所有资源都将被释放。

JVM 系统线程

如果您使用 jconsole 或任何调试器，则可以看到有许多线程在后台运行。这些后台线程除了主线程（作为调用`public static void main(String[])`的一部分而创建）以及由主线程创建的任何线程之外运行。Hotspot JVM 中主要的后台系统线程有：

虚拟机线程	该线程等待出现需要 JVM 到达安全点的操作。这些操作必须在单独的线程上进行的原因是，它们都要求 JVM 处于安全点，不能对堆进行修改。该线程执行的操作类型是“stop-the-world”垃圾收集、线程堆栈转储、线程挂起和偏向锁定撤销。
周期性任务线程	该线程负责定时器事件（即中断），用于调度周期性操作的执行
气相色谱线程	这些线程支持 JVM 中发生的不同类型的垃圾收集活动
编译器线程	这些线程在运行时将字节代码编译为本机代码
信号调度线程	该线程接收发送到 JVM 进程的信号，并通过调用适当的 JVM 方法在 JVM 内部处理它们。

每线程

每个执行线程都有以下组件：

程序计数器（PC）

当前指令（或操作码）的地址，除非它是本机的。如果当前方法是本机方法，则 PC 未定义。所有 CPU 都有一个 PC，通常 PC 在每条指令后递增，因此保存下一条要执行的指令的地址。JVM 使用 PC 来跟踪它正在执行指令的位置，PC 实际上将指向方法区中的内存地址。

堆

每个线程都有自己的堆栈，该堆栈为在该线程上执行的每个方法保存一个帧。堆栈是后进先出（LIFO）数据结构，因此当前执行的方法位于堆栈的顶部。对于每个方法调用，都会创建一个新框架并将其添加（推送）到堆栈顶部。当方法正常返回或在方法调用期间抛出未捕获的异常时，框架将被删除（弹出）。除了压入和弹出帧

对象外，堆栈不被直接操作，因此帧对象可以在堆中分配，并且内存不需要是连续的。

原生堆栈

并非所有 JVM 都支持本机方法，但是，那些通常会创建每线程本机方法堆栈的 JVM。如果 JVM 是使用 Java 本机调用 (JNI) 的 C 链接模型实现的，则本机堆栈将是 C 堆栈。在这种情况下，本机堆栈中的参数和返回值的顺序将与典型的 C 程序相同。本机方法通常可以（取决于 JVM 实现）回调到 JVM 并调用 Java 方法。这样的 Java 原生调用将发生在堆栈上（普通 Java 堆栈）；线程将离开本机堆栈并在堆栈上创建一个新框架（普通 Java 堆栈）。

堆栈限制

堆栈可以是动态的或固定大小的。如果线程需要的堆栈大于允许的堆栈，则会抛出 StackOverflowError。如果线程需要新帧并且没有足够的内存来分配它，则会抛出 OutOfMemoryError 。

框架

对于每个方法调用，都会创建一个新框架并将其添加（推送）到堆栈顶部。当方法正常返回或在方法调用期间抛出未捕获的异常时，框架将被删除（弹出）。有关异常处理的更多详细信息，请参阅下面有关异常表的部分。

每帧包含：

- 局部变量数组
- 返回值
- 操作数栈
- 当前方法的类的运行时常量池的引用

局部变量数组

局部变量数组包含方法执行期间使用的所有变量，包括对`this` 的引用、所有方法参数和其他本地定义的变量。对于类方法（即静态方法），方法参数从零开始，但是，例如方法，零槽是为此保留的。

局部变量可以是：

- 布尔值
- 字节
- 字符
- 长的
- 短的
- 整数
- 漂浮
- 双倍的
- 参考
- 退货地址

所有类型在局部变量数组中都占用一个槽，除了`long`和`double`之外，它们都占用两个连续的槽，因为这些类型是双倍宽度（64 位而不是 32 位）。

操作数栈

在字节码指令执行期间使用操作数堆栈的方式与本机 CPU 中使用通用寄存器的方式类似。大多数 JVM 字节代码都花时间通过压入、弹出、复制、交换或执行产生或消耗值的操作来操作操作数堆栈。因此，在局部变量数组和操作数堆栈之间移动值的指令在字节码中非常频繁。例如，简单的变量初始化会产生与操作数堆栈交互的两个字节代码。

```
整数我；
```

编译为以下字节代码：

```
0 :     icont_0         // 将 0 压入操作数堆栈顶部
 1 :    istore_1        // 从操作数堆栈顶部弹出值并存储为局部变量 1
```

有关局部变量数组、操作数堆栈和运行时常量池之间交互的更多详细信息，请参阅下面的类文件结构部分。

动态链接

每个帧都包含对运行时常量池的引用。该引用指向为该帧执行的方法的类的常量池。此参考有助于支持动态链接。

C/C++ 代码通常编译为目标文件，然后将多个目标文件链接在一起以生成可用的工件，例如可执行文件或 dll。在链接阶段，每个目标文件中的符号引用被替换为相对于最终可执行文件的实际内存地址。在 Java 中，这个链接阶段是在运行时动态完成的。

当编译Java类时，所有对变量和方法的引用都作为符号引用存储在类的常量池中。符号引用是逻辑引用，而不是实际指向物理内存位置的引用。JVM 实现可以选择何时解析符号引用，这可能发生在类文件被验证、加载后，称为急切或静态解析，而这可能发生在第一次使用符号引用时，称为延迟或延迟解析。然而，JVM 必须表现得就像第一次使用每个引用时发生了解析一样，并在此时抛出任何解析错误。绑定是将符号引用所标识的字段、方法或类替换为直接引用的过程，这只发生一次，因为符号引用被完全替换。如果符号引用引用尚未解析的类，则将加载该类。每个直接引用都存储为相对于与变量或方法的运行时位置关联的存储结构的偏移量。

线程之间共享

堆

堆用于在运行时分配类实例和数组。数组和对象永远不能存储在堆栈上，因为帧的大小在创建后不会改变。帧仅存储指向堆上的对象或数组的引用。与局部变量数组（在每个帧中）中的原始变量和引用不同，对象始终存储在堆上，因此当方法结束时它们不会被删除。相反，对象仅由垃圾收集器删除。

为了支持垃圾收集，堆分为三个部分：

- **年轻一代**
    - 通常分为伊甸园和幸存者
- **老一代**（也称为终身一代）
- **永久发电**

内存管理

对象和数组永远不会显式解除分配，而是垃圾收集器自动回收它们。

通常，其工作原理如下：

1. 新的对象和数组被创建到年轻代中
2. 小型垃圾收集将在年轻一代中进行。仍然存活的对象将从伊甸区移动到幸存者空间。
3. 主要垃圾收集通常会导致应用程序线程暂停，并会在各代之间移动对象。仍然存活的对象将从年轻代移动到老（终身）代。
4. 每次收集老年代时，都会收集永久代。当其中一个已满时，它们都会被收集。

非堆内存

逻辑上被视为 JVM 机制一部分的对象不是在堆上创建的。

非堆内存包括：

- **永久生成**包含
  - 方法区
  - 驻留字符串
- **代码缓存**用于编译和存储已被 JIT 编译器编译为本机代码的方法

准时（JIT）编译

Java 字节码被解释，但是这不如直接在 JVM 主机 CPU 上执行本机代码快。为了提高性能，Oracle Hotspot VM 会查找定期执行的字节代码的“热点”区域，并将其编译为本机代码。然后，本机代码存储在非堆内存中的代码缓存中。通过这种方式，Hotspot VM 尝试选择最合适的方式来权衡编译代码所需的额外时间与执行解释代码所需的额外时间。

方法区

方法区存储每个类的信息，例如：

- **类加载器参考**
- **运行时常量池**
  - 数字常量

- 现场参考
- 方法参考
- 属性

- **现场数据**
  - 每个字段
    - 姓名
    - 类型
    - 修饰符
    - 属性

- **方法数据**
  - 每个方法
    - 姓名
    - 返回类型
    - 参数类型（按顺序）
    - 修饰符
    - 属性

- **方法代码**
  - 每个方法
    - 字节码
    - 操作数栈大小
    - 局部变量大小
    - 局部变量表
    - 异常表
      - 每个异常处理程序
        - 起点
        - 终点
        - 处理程序代码的 PC 偏移量

- 被捕获的异常类的常量池索引

所有线程共享同一个方法区，因此对方法区数据的访问以及动态链接的过程必须是线程安全的。如果两个线程尝试访问尚未加载的类上的字段或方法，则只能加载一次，并且两个线程在加载之前不得继续执行。

类文件结构

编译后的类文件由以下结构组成：

```
类文件{
    u4魔法;
    u2 次要版本;
    u2 主要版本;
    u2 常量池计数;
    cp_info contant_pool[constant_pool_count - 1];
    u2 访问标志;
    u2 这个_类;
    u2 超类;
    u2 接口数;
    u2 接口[interfaces_count];
    u2 fields_count;
    字段信息字段[字段计数];
    u2 方法计数;
    method_info 方法[methods_count];
    u2 属性计数;
    attribute_info 属性[attributes_count];
}
```

魔法、次要版本、主要版本	指定有关类的版本以及编译该类的 JDK 版本的信息。
常量池	类似于符号表，尽管它包含更多数据，下面将对此进行更详细的描述。
访问标志	提供此类的修饰符列表。
这个班	索引到`constant_pool`，提供此类的完全限定名称，即`org/jamesdbloom/foo/Bar`
超类	索引到`constant_pool`，提供对超类的符号引用，即`java/lang/Object`

接口	Constant_pool 中的索引数组提供对已实现的所有接口的符号引用。
领域	Constant_pool 中的索引数组给出了每个字段的完整描述。
方法	Constant_pool 中的索引数组给出了每个方法签名的完整描述，如果该方法不是抽象的或本机的，则也存在字节码。
属性	不同值的数组，提供有关该类的附加信息，包括带有 RetentionPolicy 的任何注释。*CLASS* 或保留策略。运行

可以使用javap命令查看已编译的 Java 类中的字节代码。

如果编译以下简单类：

```
包org.jvminternals;

公共 类简单类{

    公共 无效sayHello（）{
        System.out.println（“你好”）;
    }

}
```

如果运行，您将得到以下输出：

```
javap -v -p -s -sysinfo -constants
类/org/jvminternals/SimpleClass.class
```

```
公共 类org.jvminternals.SimpleClass
  源文件：“SimpleClass.java”
  次要版本：0
  主要版本：51
  标志：ACC_PUBLIC、ACC_SUPER
常量池：
   #1 = Methodref #6.#17          // java/lang/Object. "<init>" :()V
   #2 = Fieldref #18.#19          // java/lang/System.out:Ljava/io/PrintStre
   #3 = String #20             //   "Hello"
   #4 = Methodref #21.#22         // java/io/PrintStream.println:(Ljava/lan
   #5 = Class #23              // org/jvminternals /SimpleClass
```

```
   #6 = 类 #24              // java/lang/Object
   #7 = Utf8 <初始化>
   #8 = Utf8 ()V
   #9 = UTF8 代码
  #10 = Utf8 行号表
  #11 = Utf8 本地变量表
  #12 = utf8 这个
  #13 = Utf8 Lorg/jvminternals/SimpleClass;
  #14 = Utf8 sayHello
  #15 = Utf8 源文件
  #16 = Utf8 SimpleClass.java
  #17 = NameAndType #7:#8           //   "<init>" :()V
  #18 = Class #25              // java/lang/System
  #19 = NameAndType #26:#27         // out:Ljava/io/打印流;
  #20 = Utf8 你好
  #21 = 类 #28              // java/io/PrintStream
  #22 = NameAndType #29:#30         // println:(Ljava/lang/String;)V
  #23 = Utf8 org/jvminternals/SimpleClass
  #24 = Utf8 java/lang/对象
  #25 = Utf8 java/lang/系统
  #26 = UTF8 输出
  #27 = Utf8 Ljava/io/PrintStream;
  #28 = Utf8 java/io/PrintStream
  #29 = Utf8 打印
  #30 = Utf8 (Ljava/lang/String;)V
{
  公共org.jvminternals.SimpleClass();
    签名：()V
    标志：ACC_PUBLIC
    代码：
      堆栈=1，局部变量=1，args_size=1
        0 : aload_0
        1 : invokespecial  #1     // 方法 java/lang/Object. "<init>" :()V
        4 :返回
      行号表：
        第 3 行：0
      局部变量表：
        起始长度 槽位名称 签名
          0 5 0    这个   Lorg/jvminternals/SimpleClass;

  公共 无效sayHello();
    签名：()V
    标志：ACC_PUBLIC
    代码：
      堆栈=2，局部变量=1，args_size=1
```

```
         0 : getstatic        #2      // 字段 java/lang/System.out:Ljava/io/Pri
         3 : ldc              #3      // 字符串"Hello"
         5 : invokevirtual    #4     // 方法 java/io/PrintStream.println:(Ljava
         8 : return
       行号表：
         第 6 行：0
         第 7 行：8
       局部变量表：
         起始长度 槽位名称 签名
           0 9 0    这个   Lorg/jvminternals/SimpleClass;
}
```

该类文件显示了三个主要部分：常量池、构造函数和 sayHello 方法。

- **常量池**——它提供与符号表通常提供的信息相同的信息，下面将进行更详细的描述。
- **方法**– 每个方法包含四个区域：
  - 签名和访问标志
  - 字节码
  - LineNumberTable – 向调试器提供信息以指示哪一行对应于哪个字节码指令，例如 Java 代码中的第 6 行对应于 sayHello 方法中的字节码 0，第 7 行对应于字节码 8。
  - LocalVariableTable – 列出了框架中提供的所有局部变量，在两个示例中唯一的局部变量是this。

该类文件中使用了以下字节码操作数

`aload_0`	该操作码是一组格式为`aload_<n>`的操作码之一。它们都将对象引用加载到操作数堆栈中。<n>指的是正在访问的局部变量数组中的位置，但只能是 0、1、2 或 3。还有其他类似的操作码用于加载不是对象引用的值 iload_ <n> 、lload_ < n>、 float_<n>和dload_<n>其中 i 代表int， l 代表long， f 代表float， d 代表double。索引大于 3 的局部变量可以使用 iload 、 lload 加载， 浮动 、 dload 和 aload。这些操作码都采用单个操作数，该操作数指定要加载的局部变量的索引。

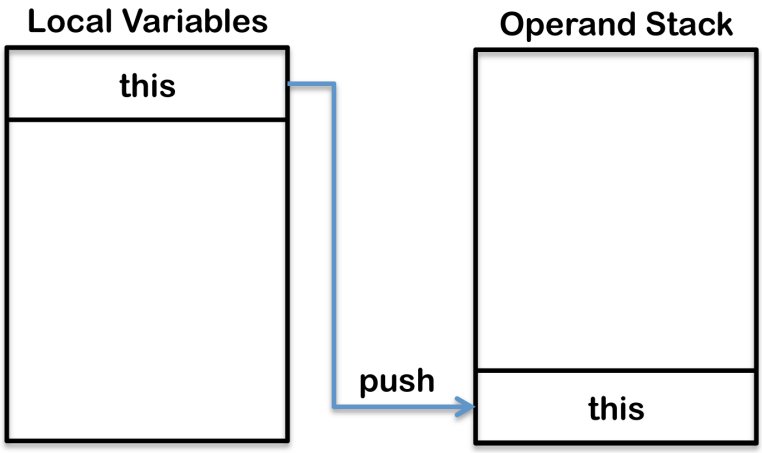
最不发达国家	该操作码用于将常量从运行时常量池推入操作数堆栈。
静态化	该操作码用于将运行时常量池中列出的静态字段中的静态值推送到操作数堆栈中。
调用特殊、调用虚拟	这些操作码位于一组调用方法的操作码中，这些方法是 invokedynamic 、invokeinterface 、 invokespecial 、invokestatic、invokevirtual。在这个类文件中，都使用了 invokespecial 和 invokevirutal ，它们之间的区别是 invokevirutal 调用基于对象的类的方法。invokespecial指令用于调用实例初始化方法以及私有方法和当前类的超类的方法。
返回	该操作码位于一组操作码 ireturn 、lreturn、 freturn、 dreturn、 areturn 和return中。这些操作码中的每一个都是类型化的 return 语句，返回不同的类型，其中 i 代表int， l 代表long， f 代表float， d 代表double， a 代表对象引用。没有前导类型字母 return 的操作码仅返回void。

与任何典型的字节代码一样，大多数操作数与局部变量、操作数堆栈和运行时常量池交互，如下所示。

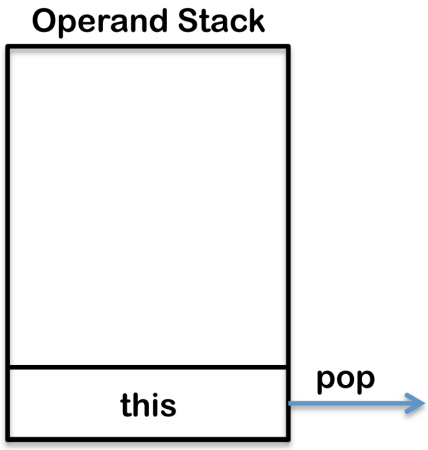
构造函数有两条指令，首先将其推入操作数堆栈，然后调用超类的构造函数，该构造函数消耗this 的值，从而将其从操作数堆栈中弹出。

SimpleClass()

0: aload_0



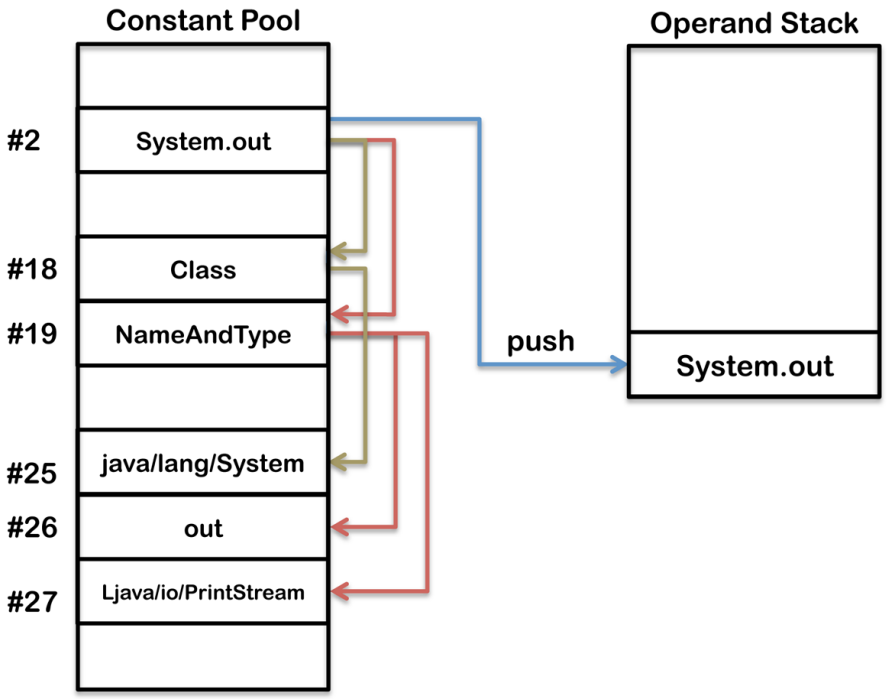
1: invokespecial #1



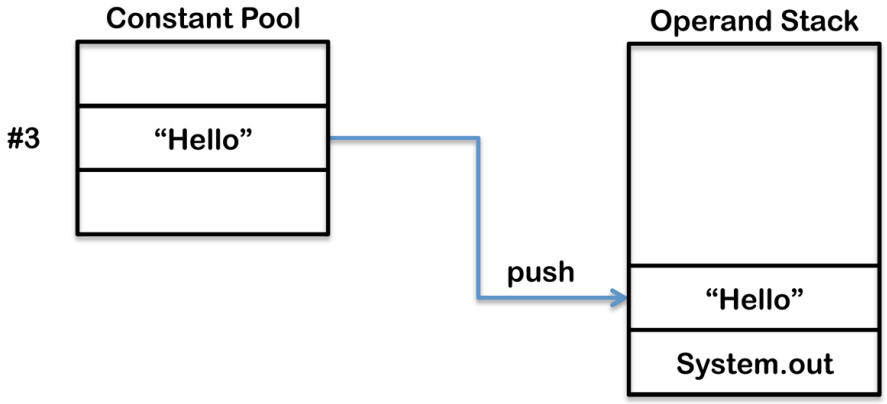
sayHello（）方法更加复杂，因为它必须使用运行时常量池将符号引用解析为实际引用，如上面更详细地解释的。第一个操作数`getstatic` 用于将对静态字段的引用从 `System` 类推送到操作数堆栈。下一个操作数`ldc`将字符串`"Hello"`压入操作数堆栈。最后一个操作数`invokevirtual`调用`System.out`的`println`方法，该方法将`"Hello"`作为参数从操作数堆栈中弹出，并为当前线程创建一个新帧。

sayHello()

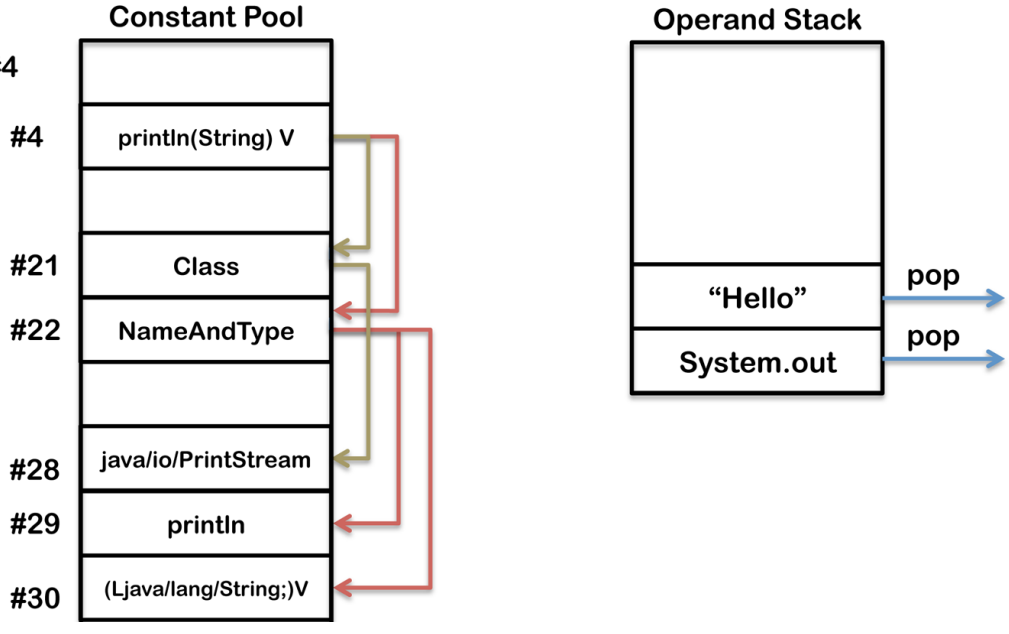
0: getstatic



3: ldc



5: invokevirtual #4



类加载器

The JVM starts up by loading an initial class using the bootstrap classloader. The class is then linked and initialized before `public static void main(String[])` is invoked. The execution of this method will in turn drive the loading, linking and initialization of additional classes and interfaces as required.

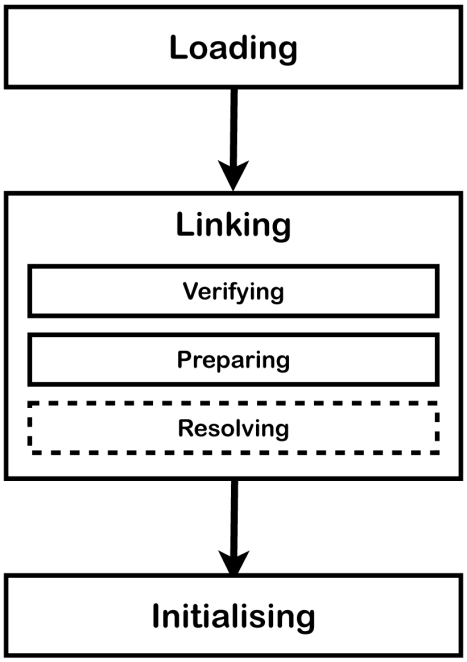
Loading is the process of finding the class file that represents the class or interface type with a particular name and reading it into a byte array. Next the bytes are parsed to confirm they represent a `Class` object and have the correct major and minor versions. Any class or interface named as a direct superclass is also loaded. Once this is completed a class or interface object is created from the binary representation.

链接是采用类或接口验证并准备类型及其直接超类和超接口的过程。链接由验证、准备和可选解决三个步骤组成。

> ***验证***是确认类或接口表示在结构上正确并遵守 Java 编程语言和 JVM 的语义要求的过程，例如执行以下检查：
>
> 1. 一致且格式正确的符号表
> 2. 最终方法/类未被覆盖
> 3. 方法尊重访问控制关键字
> 4. 方法具有正确的参数数量和类型
> 5. 字节码不会错误地操作堆栈
> 6. 变量在读取之前初始化
> 7. 变量是正确类型的值
>
> 在验证阶段执行这些检查意味着不需要在运行时执行这些检查。链接期间的验证会减慢类加载速度，但它避免了在执行字节码时多次执行这些检查的需要。
>
> ***准备工作***涉及为静态存储和 JVM 使用的任何数据结构（例如方法表）分配内存。创建静态字段并将其初始化为其默认值，但是，此阶段不会执行任何初始化程序或代码，因为这是初始化的一部分。

解析是一个可选阶段，涉及通过加载引用的类或接口来检查符号引用并检查引用是否正确。如果此时这没有发生，则可以将符号引用的解析推迟到字节码指令使用它们之前。

类或接口的初始化包括执行类或接口初始化方法`<clinit>`



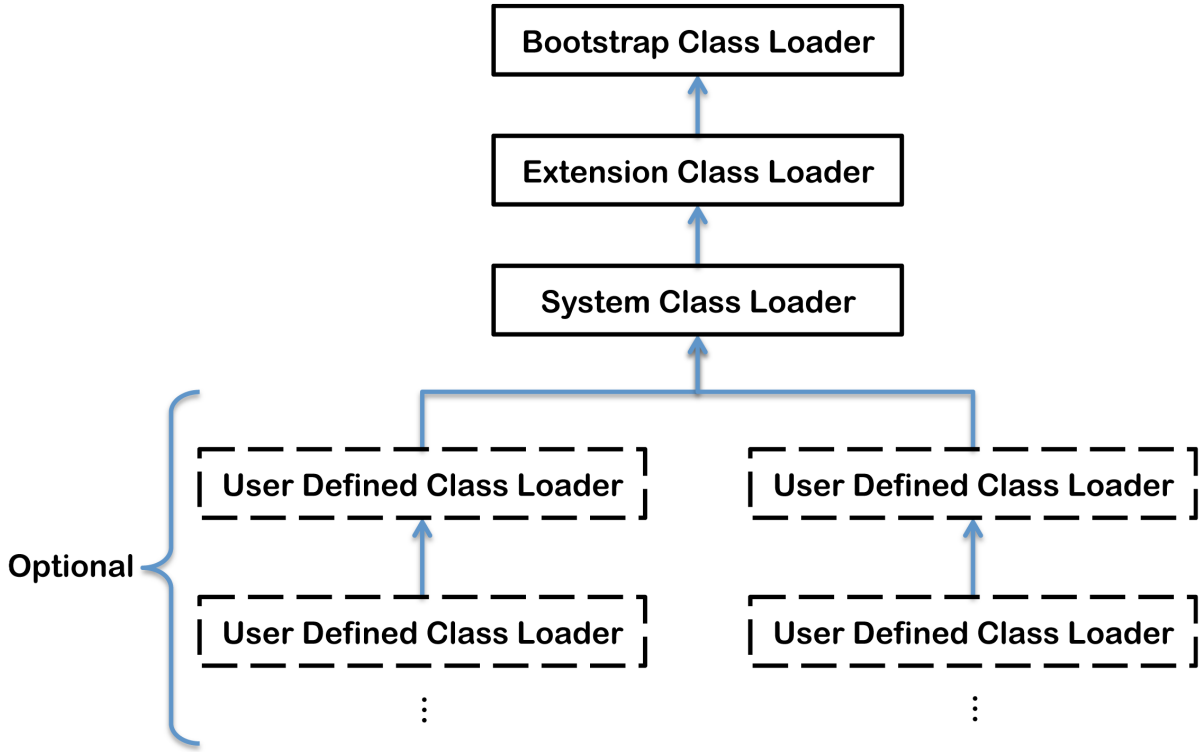
在 JVM 中，有多个具有不同角色的类加载器。每个类加载器都委托给其父类加载器（加载它的），但**引导类加载器**除外，它是顶级类加载器。

Bootstrap 类加载器通常作为本机代码实现，因为它在 JVM 加载时很早就被实例化。引导类加载器负责加载基本的 Java API，例如包括 rt.jar。它只加载在启动类路径上找到的具有更高信任级别的类；因此，它跳过了为普通类完成的大部分验证。

扩展类加载器从标准 Java 扩展 API（例如安全扩展函数）加载类。

系统类加载器是默认的应用程序类加载器，它从类路径加载应用程序类。

用户定义的类加载器也可以用于加载应用程序类。用户定义的类加载器用于许多特殊原因，包括运行时重新加载类或分离不同组的加载类，这些通常是 Web 服务器（例如 Tomcat）所需的。



更快的类加载

HotSpot JMV 从 5.0 版本开始引入了名为类数据共享 (CDS) 的功能。在 JVM 的安装过程中，安装程序会将一组关键 JVM 类（例如 rt.jar）加载到内存映射共享存档中。CDS 减少了加载这些类所需的时间，提高了 JVM 启动速度，并允许这些类在 JVM 的不同实例之间共享，从而减少了内存占用。

方法区在哪里

Java 虚拟机规范 Java SE 7 版本明确指出：“虽然方法区逻辑上是堆的一部分，但简单的实现可能会选择不进行垃圾收集或压缩它。”与此相反，Oracle JVM 的 jconsole 将方法区域（和代码缓存）显示为非堆。OpenJDK代码显示 CodeCache 是 VM 中与 ObjectHeap 不同的一个字段。

类加载器参考

加载的所有类都包含对加载它们的类加载器的引用。反过来，类加载器还包含对其已加载的所有类的引用。

运行时常量池

JVM 维护每个类型的常量池，这是一种类似于符号表的运行时数据结构，尽管它包含更多数据。Java中的字节码需要数据，通常这些数据太大而无法直接存储在字节码中，而是存储在常量池中，并且字节码包含对常量池的引用。运行时常量池用于动态链接，如上所述

常量池中存储了几种类型的数据，包括

- 数字文字
- 字符串文字
- 类参考
- 现场参考
- 方法参考

例如下面的代码：

```
对象 foo = new Object();
```

用字节码写成如下：

```
0 :     new   #2                  // 类 java/lang/Object
1 :     dup
2 :     invokespecial   #3      // 方法 java/lang/Object "<init>" ( ) V
```

新操作码（操作数代码）后面跟着#2操作数。该操作数是常量池的索引，因此引用常量池中的第二个条目。第二个条目是类引用，该条目又引用常量池中的另一个条目，其中包含类名称作为常量 UTF8 字符串，其值为 `// Class java/lang/Object`。然后可以使用此符号链接来查找 java.lang.Object 的类。新的操作码创建一个类实例并初始化其变量。然后将对新类实例的引用添加到操作数堆栈中。骗子_然后，操作码在操作数堆栈上创建顶部引用的额外副本，并将其添加到操作数堆栈的顶部。最后，第 2 行由invokespecial调用实例初始化方法。该操作数还包含对常量池的引用。初始化方法使用（弹出）操作数池中的顶部引用作为该方法的参数。最后有一个对已创建并初始化的新对象的引用。

如果编译以下简单类：

```
包org.jvminternals;

公共 类简单类{

    公共 无效sayHello（）{
        System.out.println( “你好” );
    }

}
```

生成的类文件中的常量池如下所示：

```
常量池：
   #1 = Methodref #6.#17          // java/lang/Object. "<init>" :()V
   #2 = Fieldref #18.#19          // java/lang/System.out:Ljava/io/PrintStre
   #3 = String #20              //   "Hello"
   #4 = Methodref #21.#22         // java/io/PrintStream.println:(Ljava/lan
   #5 = Class #23              // org/jvminternals /SimpleClass
   #6 = 类 #24              // java/lang/Object
   #7 = Utf8 <初始化>
   #8 = Utf8 ()V
   #9 = UTF8 代码
  #10 = Utf8 行号表
  #11 = Utf8 本地变量表
  #12 = utf8 这个
  #13 = Utf8 Lorg/jvminternals/SimpleClass;
  #14 = Utf8 sayHello
  #15 = Utf8 源文件
  #16 = Utf8 SimpleClass.java
  #17 = NameAndType #7:#8           //   "<init>" :()V
  #18 = Class #25              // java/lang/System
  #19 = NameAndType #26:#27         // out:Ljava/io/打印流;
  #20 = Utf8 你好
  #21 = 类 #28              // java/io/PrintStream
  #22 = NameAndType #29:#30         // println:(Ljava/lang/String;)V
  #23 = Utf8 org/jvminternals/SimpleClass
  #24 = Utf8 java/lang/对象
  #25 = Utf8 java/lang/系统
  #26 = UTF8 输出
  #27 = Utf8 Ljava/io/PrintStream;
  #28 = Utf8 java/io/PrintStream
```

```
#29 = Utf8 打印
#30 = Utf8 (Ljava/lang/String;)V
```

常量池包含以下类型：

整数	4 字节 int 常量
长的	8 字节长的常量
漂浮	4 字节浮点常量
双倍的	8 字节双精度常量
细绳	一个字符串常量，指向常量池中另一个包含实际字节的 Utf8 条目
UTF8	表示 Utf8 编码字符序列的字节流
班级	一个类常量，指向常量池中的另一个 Utf8 条目，其中包含内部 JVM 格式的完全限定类名（由动态链接过程使用）
名称和类型	冒号分隔一对值，每个值都指向常量池中的其他条目。第一个值（冒号之前）指向作为方法或字段名称的 Utf8 字符串条目。第二个值指向表示类型的 Utf8 条目，对于字段来说，这是完全限定的类名，对于方法来说，这是每个参数一个完全限定类名的列表。
字段引用、方法引用、接口方法引用	一对点分隔的值，每个值都指向常量池中的其他条目。第一个值（点之前）指向类条目。第二个值指向 NameAndType 条目。

异常表

异常表存储每个异常处理程序的信息，例如：

- 起点
- 终点
- 处理程序代码的 PC 偏移量
- 被捕获的异常类的常量池索引

如果方法定义了 try-catch 或 try-finally 异常处理程序，则将创建异常表。它包含每个异常处理程序或finally块的信息，包括处理程序应用的范围、正在处理的异常类型以及处理程序代码的位置。

当抛出异常时，JVM 在当前方法中查找匹配的处理程序，如果没有找到，该方法会突然结束并弹出当前堆栈帧，并在调用方法（新的当前帧）中重新抛出异常。如果在弹出所有帧之前未找到异常处理程序，则线程将终止。如果在最后一个非守护线程中抛出异常（例如，如果该线程是主线程），这也可能导致 JVM 本身终止。

最后，异常处理程序匹配所有类型的异常，因此每当抛出异常时总是执行。在没有抛出异常的情况下，finally 块仍然在方法末尾执行，这是通过在执行 return 语句之前立即跳转到finally 处理程序代码来实现的。

符号表

除了每个类型的运行时常量池之外，Hotspot JVM 还在永久代中保存了一个符号表。符号表是一个将符号指针映射到符号的哈希表（即`Hashtable<Symbol*, Symbol>`），并包含指向所有符号的指针，包括每个类中运行时常量池中保存的符号。

引用计数用于控制何时从符号表中删除符号。例如，当卸载一个类时，其运行时常量池中保存的所有符号的引用计数都会减少。当符号表中符号的引用计数变为零时，符号表就知道该符号不再被引用，并且该符号将从符号表中卸载。对于符号表和字符串表（见下文），所有条目都以规范化形式保存，以提高效率并确保每个条目仅出现一次。

内部字符串（字符串表）

Java 语言规范要求包含相同 Unicode 代码点序列的相同字符串文字必须引用相同的 String 实例。此外，如果在 String 实例上调用`String.intern()`，则必须返回一个引用，该引用与字符串是文字时的引用返回相同。因此，以下内容成立：

```
("j" + "v" + "m").intern() == "jvm"
```

在Hotspot JVM中，interned字符串保存在字符串表中，字符串表是一个将对象指针映射到符号的Hashtable（即`Hashtable<oop, Symbol>`），并保存在永久代

中。对于符号表（见上文）和字符串表，所有条目都以规范化形式保存，以提高效率并确保每个条目仅出现一次。

字符串文字由编译器自动保留，并在加载类时添加到符号表中。此外，可以通过调用`String.intern()`显式地实习 String 类的实例。当调用`String.intern()`时，如果符号表已包含该字符串，则返回对此的引用，如果没有，则将该字符串添加到字符串表中并返回其引用。

詹姆斯·D·布鲁姆