JAVA 虚拟机概述

QFC

目录

1. 几个问题 1

2. 问题参考 2

3. JVM 的体系结构 4

4. 类加载机制，ClassLoader 子系统 5

   4.1 装载 5

   4.2 链接：6

   4.3 初始化：6

5. 执行引擎 6

6. 运行时数据区（方法区、堆、java 栈、PC 寄存器、本地方法栈）7

   6.1 线程 7

   6.2 JVM 系统线程 8

   6.3 线程独有哪些资源？8

   6.4 线程共享资源 10

By---nolan

1. 几个问题

2. Java Virtual Machine(简称 JVM)是什么?

3. JVM 有哪些版本?

3. JVM 是用什么语言实现的?

4. JDK, JRE, JVM, 编译器, 这几者之间是什么联系?

1. 问题参考

1. JVM 是一种用于计算设备的规范, 用于在实际的计算机上模拟实体计算机功能的虚拟计算机规范。JVM 提供的基于抽象规格描述的计算机模型, 为解释程序开发人员提供了很好的灵活性 (, 同时也确保 Java 代码可在符合该规范的任何系统上运行。JVM 定义了控制 Java 代码解释执行和具体实现的五种规格, 它们是:

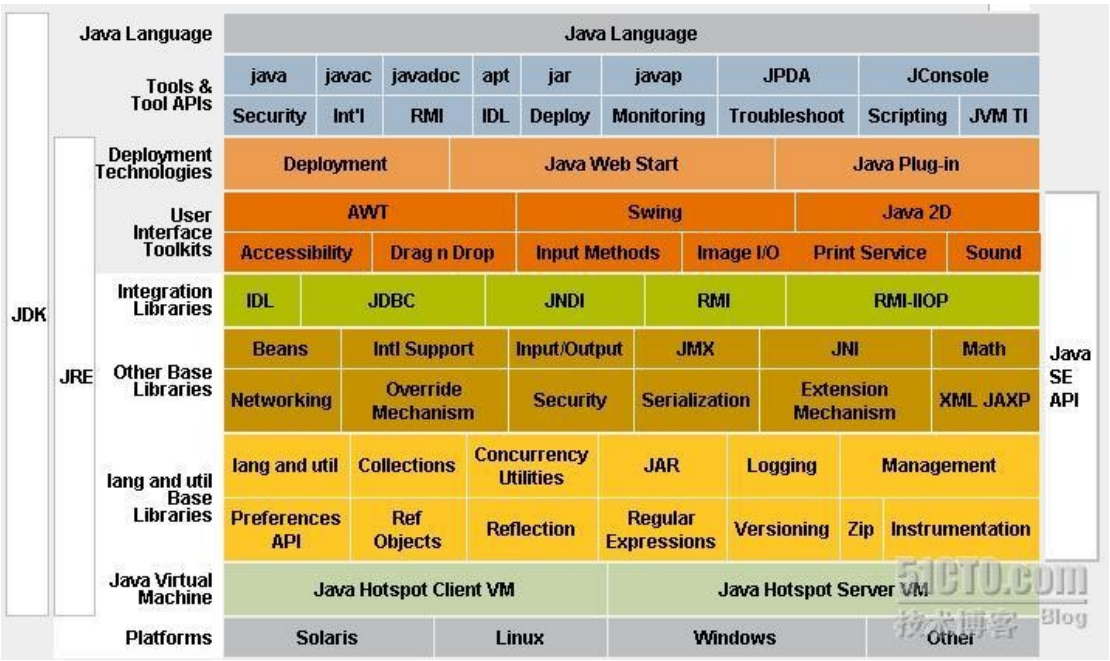
   JVM 指令系统, JVM 寄存器(PC 寄存器等), JVM 栈结构, JVM 碎片回收堆, JVM 存储区。

2. JVM 的实现有多种, 比如 sun 公司(现在属于 oracle)的 HotSpot 版本, BEA 公司的(也被 oracle 收购了)JRockit 版本。在 jdk8 中, JRockit 版本已经被合并吸收了, 还叫 hotspot。J9 VM, J9 是 IBM 开发的一个高度模块化的 JVM, 在许多平台上, IBM J9 VM 都只能跟 IBM 产品一起使用。Zing VM, Zing VM 是一个从 Sun HoSpot VM fork 出来的一个高性能 JVM, 可以运行在 Linux/x86-64 平台上, 主要重写了 GC 部分。IKVM.NET, 直接能在.net 上运行完整的 java 程序。Android 上的 Dalvik / ART 虽然名字不叫 JVM, 但骨子里也是不折不扣的 JVM,不同的是 Dalvik 基于的是寄存器的架构。

3. 是用 C++/C 语言写的, 大部分是 C++, 少部分是 C 语言, 比如 JNI(Java 本地接口, 用于和其他语言通信)。一个主要原因是, C++的类可以很容易的实现 java 的类, 且 JVM 要同底层硬件, 操作系统打交道, C++的指针操作很受用。

   实际上, 任何语言都能够实现 JVM, 只要符合 JVM 的规范即可。Java 语言本身也可以实现 JVM, 只是需要在 JVM 上面跑, 这似乎是个哲学问题。。。不管用什么语言编写,只要能编译成目标机器代码,就能在目标机器上运行.

4. 先附上两张图



从小的开始：JVM 已经讲了，就是运行在实体计算机平台上，模拟计算机功能的虚拟机，讲的具体点就是 bin 目录下的 jvm.dll 文件；里面最重要的是执行引擎。Jre= jvm + lib：java runtime environment，Java 的运行环境，有了 jre 才能运行 java 程序，最重要的是 rt.jar，JAVA 基础类库。Jdk=jre+工具(javac,java,javadoc 等)+java 基础类库(主要如 tools.jar,dt.jar 等)。举个例子：使用 javac 编译器编译时出现下面的错误提示：

```
D:\>javac Demo.java
错误: 找不到或无法加载主类 com.sun.tools.javac.Main
```

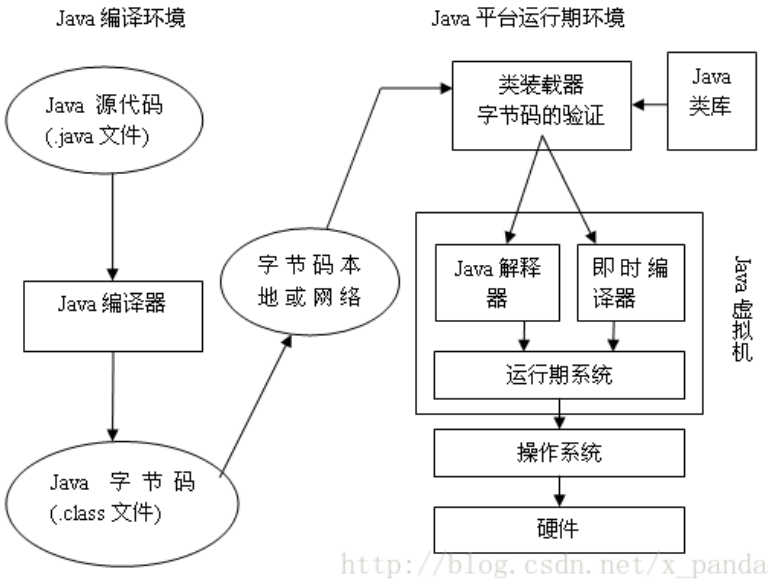
解析：输入的 javac 的命令不是去 JDK 中 bin 目录去找的 javac.exe，而是去 JDK 中 lib 目录中的 tools.jar 中 com.sun.tools.javac.Main 中执行，javac.exe 只是一个包装器（Wrapper），存在的目的只是为了让开发者免于输入过长的指命。而操作系统装入 JVM，则是通过 jdk/bin 中的 Java.exe 来完成。

总结：JDK 包含 JRE，而 JRE 包含 JVM，总的来说 JDK 是用于 java 程序的开发，而 jre 则是只能运行 class 而没有编译的功能，Eclipse、IntelliJ IDEA 等其他 IDE 有自己的编译器而不是用 JDK bin 目录中自带的 javac。

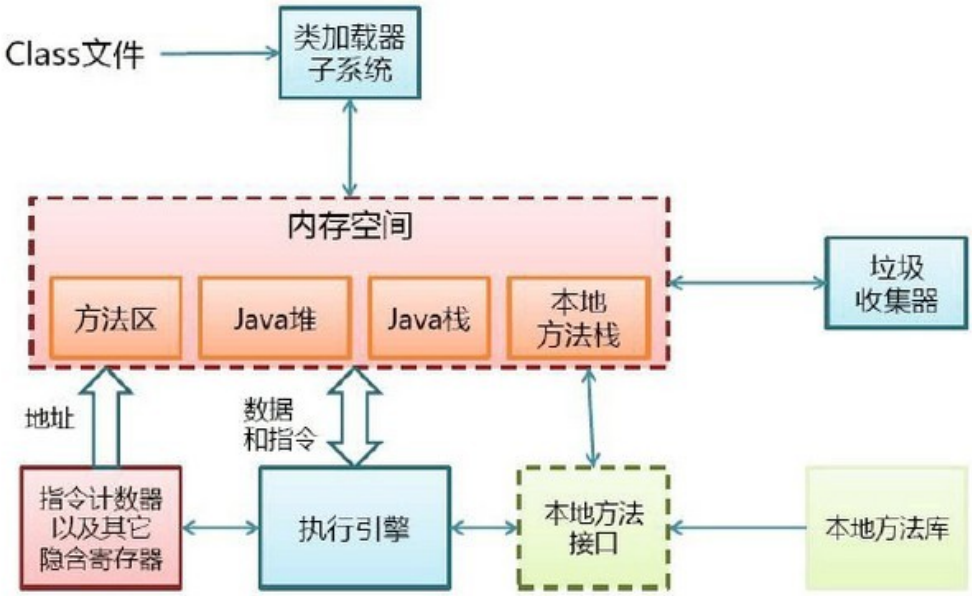
进入正题:

2．JVM 的体系结构

先看看 java 程序的从编译到运行整个过程:

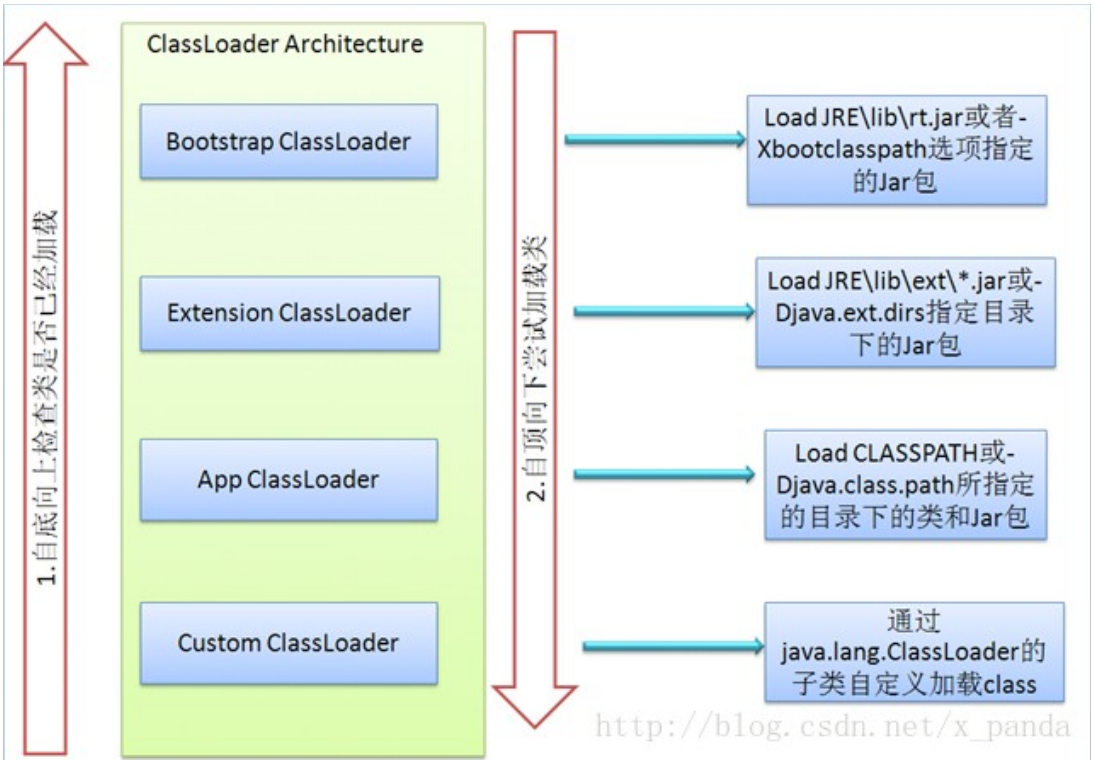


再给出 JVM 自身的体系结构



从上图可以看到：JVM 结构主要包括两个子系统和两个组件。两个子系统分别是 Classloader 子系统和 Execution engine(执行引擎)子系统；两个组件分别是 Runtime data area(运行时数据区域)组件和 Native interface(本地接口)组件。

3．类加载机制，ClassLoader 子系统



3.1 装载

Java 提供了动态的装载特性；它会在运行时的第一次引用到一个 class 的时候对它进行装载和链接，而不是在编译期进行。JVM 的类装载器负责动态装载。Java 类装载器有如下几个特点：

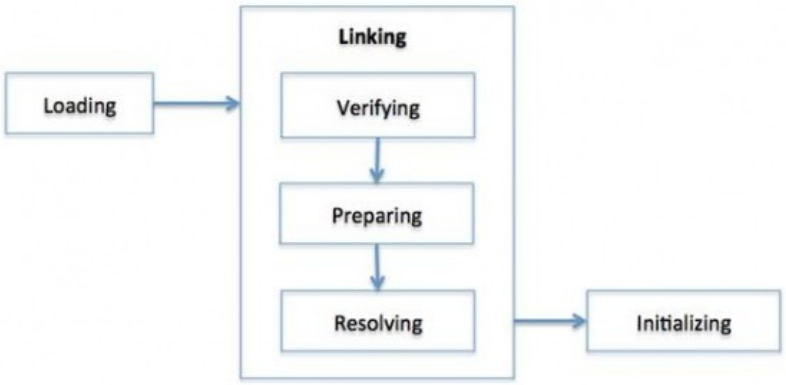
•层级结构：Java 里的类装载器被组织成了有父子关系的层级结构。Bootstrap 类装载器是所有装载器的父亲。
•代理模式：基于层级结构，类的装载可以在装载器之间进行代理。当装载器装载一个类时，首先会检查它是否在父装载器中进行装载了。如果上层的装载器已经装载了这个类，这个类会被直接使用。反之，类装载器会请求装载这个类。
•可见性限制：一个子装载器可以查找父装载器中的类，但是一个父装载器不能查找子装载器里的类。

•不允许卸载：类装载器可以装载一个类但是不可以卸载它，不过可以删除当前的类装载器，然后创建一个新的类装载器。

每个类装载器都有一个自己的命名空间用来保存已装载的类。当一个类装载器装载一个类时，它会通过保存在命名空间里的类全局限定名(Fully Qualified Class Name)进行搜索来检测这个类是否已经被加载了。如果两个类的全局限定名是一样的，但是如果命名空间不一样的话，那么它们还是不同的类。不同的命名空间表示 class 被不同的类装载器装载。

当一个类装载器（class loader）被请求装载类时，它首先按照顺序在上层装载器、父装载器以及自身的装载器的缓存里检查这个类是否已经存在。简单来说，就是在缓存里查看这个类是否已经被自己装载过了，如果没有的话，继续查找父类的缓存，直到在 bootstrap 类装载器里也没有找到的话，再从上到下尝试加载这些类，找不到就会抛出常见的 classNotFoundException，原因主要有：1）这个类没有被编译，或者更新后没有被重新编译；2）这个类没有放在 classpath 路径下，3）反射时的类名写错了。同样的 NoSuchMethodError 也有可能是这些原因.

如果类装载器查找到一个没有装载的类，它会按照下图的流程来装载和链接这个类：



每个阶段的描述如下：

•Loading：类的信息从文件中获取并且载入到 JVM 的内存里。

•Verifying:检查读入的结构是否符合 Java 语言规范以及 JVM 规范的描述。这是类装载中最复杂的过程，并且花费的时间也是最长的。并且 JVM TCK 工具的大部分场景的用例也用来测试在装载错误的类的时候是否会出现错误。

•Preparing:分配一个结构用来存储类信息，这个结构中包含了类中定义的成员变量，方法和接口的信息。

•Resolving:把这个类的常量池中的所有的符号引用改变成直接引用。

•Initializing:把类中的变量初始化成合适的值。执行静态初始化程序，把静态变量初始化成指定的值。

JVM 规范定义了上面的几个任务，不过它允许具体执行的时候能够有些灵活的变动。

四个级别的类加载器

①Bootstrap ClassLoader

负责加载$JAVA_HOME 中 jre/lib/rt.jar 里所有的 class，由 C++实现，不是 ClassLoader 子类

②Extension ClassLoader

负责加载 java 平台中扩展功能的一些 jar 包，包括$JAVA_HOME 中 jre/lib/*.jar 或-Djava.ext.dirs 指定目录下的 jar 包

③App ClassLoader

负责加载 classpath 中指定的 jar 包及工程目录中的 class

④Custom ClassLoader

属于应用程序根据自身需要自定义的 ClassLoader，如 tomcat 会根据 j2ee 规范自行实现 ClassLoader。

JVM 采用三个元素来标识一个被加载了的类：类名+包名+ClassLoader 实例 ID，装载后，放在方法区（也称作非堆区）。

3.2 链接：

链接过程负责，1).对二进制字节码的格式进行校验、2).初始化装载类中的静态变量、3).解析类中调用的接口、类。

3.3 初始化：

初始化过程即为执行类中的静态初始化代码、构造器代码以及静态属性的初始化，在四种情况下初始化过程会被触发执行：调用了 new；反射调用了类中的方法；子类调用了初始化；JVM 启动过程中指定的初始化类。

4. 执行引擎

在执行方法时 JVM 提供了四种指令来执行：

（1）invokestatic：调用类的 static 方法

（2）invokevirtual：调用对象实例的方法

（3）invokeinterface：将属性定义为接口来进行调用。

（4）invokespecial：JVM 对于初始化对象（Java 构造器的方法为：<init>）以及调用对象实例中的私有方法时。

主要的执行技术有：

解释，即时编译，自适应优化执行

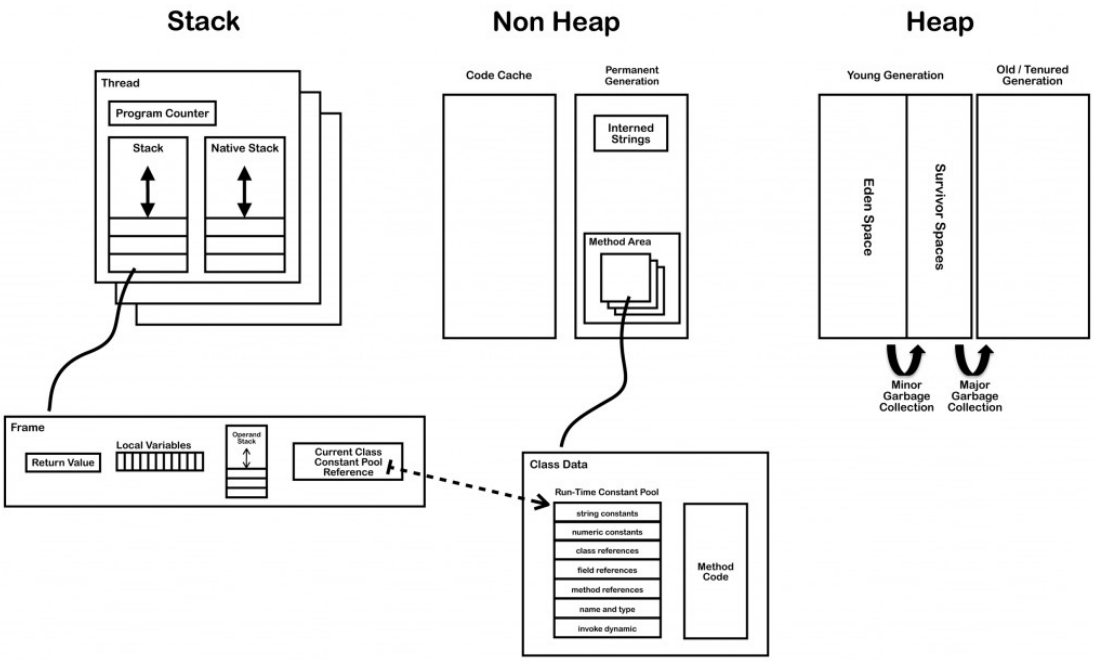
（1）解释属于第一代 JVM；

（2）即时编译 JIT(Just In Time)属于第二代 JVM；

（3）自适应优化：为了提高性能，Oracle Hotspot 虚拟机会找到执行最频繁的字节码片段并把它们编译成原生机器码。编译出的原生机器码被存储在非堆内存的代码缓存中。通过这种方法，Hotspot 虚拟机将权衡下面两种时间消耗：将字节码编译成本地代码需要的额外时间和解释执行字节码消耗更多的时间。

5．运行时数据区（方法区、堆、java 栈、PC 寄存器、本地方法栈）

这里要从线程的角度去解读运行时数据区，下图是 jdk1.7 的运行时数据区的内容结构图：



5.1 线程

线程是一个程序里的运行单元。JVM 允许一个应用有多个线程并行的执行。在 Hotspot JVM 里，每个线程都与操作系统的本地线程直接映射。在一个 Java 线程准备好了所有的状态后（比如线程本地存储，缓存分配，同步的对象，栈以及程序计数器），这时一个操作系统中的本地线程也同时创建。当 Java 线程终止后，本地线程也会回收。操作系统因此负责所有线程的安排调度到任何一个可用的 CPU 上。一旦本地线程初始化成功，它就会调用 Java 线程中的 run()方法。当 run()方法返回，或发生了未捕获异常，Java 线程终止，本地线程就会决定是否 JVM 也应该被终止(是否是最后一个非守护线程）。当线程终止后，本地线程和 Java 线程持有的资源都会被释放。

5.2 JVM 系统线程

JVM 系统线程不包括调用 public static void main(String[])的 main 线程以及所有这个 main 线程自己创建的线程。这些主要的后台系统线程在 Hotspot JVM 里主要是以下几个：

- 周期任务线程：这种线程是时间周期事件的体现(比如中断)，他们一般用于周期性操作的调度执行。
- GC 线程：这种线程对在 JVM 里不同种类的垃圾收集行为提供了支持。
- 编译线程：这种线程在运行时会将字节码编译成到本地代码。
- 信号调度线程：这种线程接收信号并发送给 JVM，在它内部通过调用适当的方法进行处理。

5.3 线程独有哪些资源?

1）程序计数器(PC)

PC 中存储着当前线程下一个指令（或操作码）的地址。如果当前方法是 native 方法，那么 PC 的值为 undefined。在计算机中， CPU 都有一个 PC，典型状态下，每执行一条指令 PC 都会自增。同样的，JVM 用 PC 来跟踪指令执行的位置，PC 将实际上是指向方法区（Method Area）的一个内存地址。

2）java 栈(stack)

每个线程拥有自己的栈，栈包含每个方法执行的栈帧。栈是一个后进先出（LIFO）的数据结构，因此当前执行的方法在栈的顶部。每次方法调用时，一个新的栈帧创建并压栈到栈顶。

当方法正常返回或抛出未捕获的异常时，栈帧就会出栈。除了栈帧的压栈和出栈，栈不能被直接操作。所以可以在堆上分配栈帧，并且不需要连续内存。

栈可以是动态分配也可以固定大小。如果线程请求一个超过允许范围的空间，就会抛出一个 StackOverflowError。如果线程需要一个新的栈帧，但是没有足够的内存可以分配，就会抛出一个 OutOfMemoryError。

3）本地方法栈(Native stack)(可选)

并非所有的 JVM 实现都支持本地（native）方法，那些提供支持的 JVM 一般都会为每个线程创建本地方法栈。如果 JVM 用 C-linkage 模型实现 JNI（Java Native Invocation），那么本地栈就是一个 C 的栈。在这种情况下，本地方法栈的参数顺序、返回值和典型的 C 程序相同。本地方法一般来说可以（依赖 JVM 的实现）反过来调用 JVM 中的 Java 方法。这种 native 方法调用 Java 会发生在栈（一般是 Java 栈）上；线程将离开本地方法栈，并在 Java 栈上开辟一个新的栈帧.考虑到代码的可移植性，非必须情况下，不建议使用本地方法。

4）栈帧

每次方法调用都会新建一个新的栈帧并把它压栈到栈顶。当方法正常返回或者调用过程中抛出未捕获的异常时，栈帧将出栈。

每个栈帧包含：

- 局部变量数组
- 返回值
- 操作数栈
- 类当前方法的运行时常量池引用

5）局部变量数组

局部变量数组包含了方法执行过程中的所有变量，包括 this 引用、所有方法参数、其他局部变量。对于类方法（也就是静态方法），方法参数从下标 0 开始，对于对象方法，位置 0 保留为 this。

除了 long 和 double 类型以外，所有的变量类型都占用局部变量数组的一个位置，同样的，对象只存储对象的引用(指针)，占一个字节。long 和 double 需要占用局部变量数组两个连续的位置，因为它们是 64 位双精度，其它类型都是 32 位单精度。

6）操作数栈(几个寄存器)

操作数栈在执行字节码指令过程中被用到，这种方式类似于原生 CPU 寄存器。大部分 JVM 字节码把时间花费在操作数栈的操作上：入栈、出栈、复制、交换、产生消费变量的操作。因此，局部变量数组和操作数栈之间的交换变量指令操作通过字节码频繁执行。比如，一个简单的变量初始化语句将产生两条跟操作数栈交互的字节码。

```
int i;
```

被编译成下面的字节码

```
0:    iconst_0    // Push 0 to top of the operand stack
1:    istore_1    // Pop value from top of operand stack and store as local variable 1
```

字节码

7）动态链接

每个栈帧都有一个运行时常量池的引用。这个引用指向栈帧当前运行方法所在类的常量池。通过这个引用支持动态链接（dynamic linking）

5.4线程共享资源

1）Java 堆

堆被用来在运行时分配类实例、数组。不能在栈上存储数组和对象。因为栈帧被设计为创建以后无法调整大小。栈帧只存储指向堆中对象或数组的引用。与局部变量数组（每个栈帧中的）中的原始类型和引用类型不同，对象总是存储在堆上以便在方法结束时不会被移除。对象只能由垃圾回收器移除。

为了支持垃圾回收机制，堆被分为了下面三个区域：

- 新生代（经常被分为 Eden 和 Survivor）
- 老年代
- 永久代

对象和数组永远不会显式回收，而是由垃圾回收器自动回收。通常，过程是这样的：

- 新的对象和数组被创建并放入老年代。
- Minor 垃圾回收将发生在新生代。依旧存活的对象将从 eden 区移到 survivor 区。
- Major 垃圾回收一般会导致应用进程暂停，它将在三个区内移动对象。仍然存活的对象将被从新生代移动到老年代。
- 每次进行老年代回收时也会进行永久代回收。它们之中任何一个变满时，都会进行回收。

2）非堆内存

包括：方法区+驻留字符串+代码缓存(用于编译和存储那些被 JIT 编译器编译成原生代码的方法)

方法区存储的内容包括：

- Classloader 引用
- 运行时常量池

  数值型常量，字段引用，方法引用，属性
- 字段数据(域)

  针对每个字段的信息，字段名，类型，修饰符，属性（Attribute），方法数据
- 方法数据

  方法名，返回值类型，参数类型（按顺序），修饰符，属性
- 方法代码

  字节码，操作数栈大小，局部变量大小，局部变量表
- 异常表

  开始点，结束点，异常处理代码的程序计数器（PC）偏移量，被捕获的异常类对应的常量池下标

所有线程共享同一个方法区，因此访问方法区数据的和动态链接的进程必须线程安全。如果两个线程试图访问一个还未加载的类的字段或方法，必须只加载一次，而且两个线程必须等它加载完毕才能继续执行。

参考链接：http://www.cnblogs.com/dingyingsi/p/3760447.html

http://ifeve.com/jvm-internals/

http://www.open-open.com/lib/view/open1408453806147.html

http://www.csdn.net/article/2012-12-05/2812509-Java-JVM