

# Java 平台体系——第二章 JVM (Java 虚拟机)

## 第二章 JVM (Java 虚拟机)

Java 之父 James Gosling 说过他看重的并不是 Java 语言，而是 JVM。

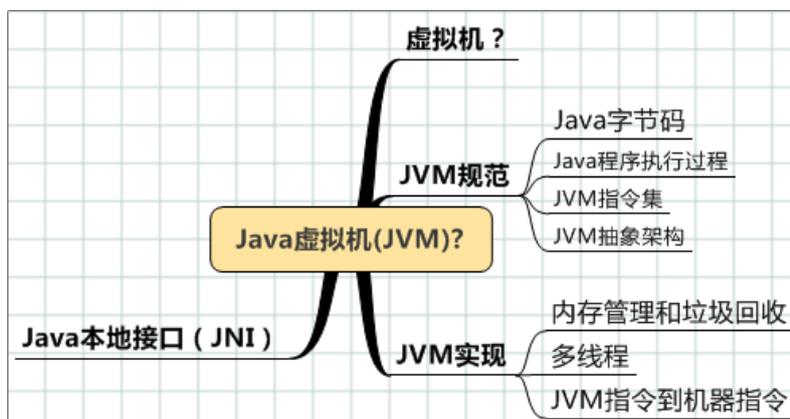
JVM 是 Java 程序能够“跨平台”运行的基础设施。前面我们说过 Java 平台不仅仅是 Java 语言，这其中很重要的一个因素就是 JVM。

可以说 JVM 是 Java 平台的核心组成部分，要成为 Java 的“大牛”，或者要成为“编程大牛”，了解 JVM 的宏观架构和核心主题是必须的。在图表 *Java SE 平台* JVM 层有如下的内容：



首先我明确一下，大家不要被“Java Hotspot Client VM”和“Java Hotspot Server VM”导致误解：JVM 就是由“Java Hotspot Client VM”和“Java Hotspot Server VM”组成。Hotspot 虚拟机（现在在 [OpenJDK](#) 下维护）仅仅是 Sun（现在被 Oracle 收购）提供的 JVM 实现。目前市面上的 JVM 有 Oracle 的 JRockit (<http://www.oracle.com/appserver/jrockit/index.html>，但 Oracle 收购 Sun 之后事情可能会变，变得方向当然是优点合并了)，IBM 的 JVM，JikesRVM (<http://jikesrvm.sourceforge.net>)，Apache Harmony (<http://harmony.apache.org>) 等。其中 JikesRVM 是大家研究和学习 JVM 的样本，其商业应用基本没有，大家别忘了去逛逛它的官方网站，相信一定会有收获，如果你是搞研究，那更应该去逛逛了，如果你熟悉 JavaScript 又想去研究 JVM，我再推荐一个比较变态的用 JavaScript 去“实现”的 JVM 吧，它就是 BicaVM (<https://github.com/nurv/BicaVM>)。

在学习虚拟机之前请先理解下图，或者至少要了解从哪几个方面去认识 JVM。



图表 1 Java 虚拟机主要内容组成

## 2.1. 什么是虚拟机?

我们常常接触的 Windows/Linux/Unix 操作系统可以说就是一个虚拟机，它为上层的应用软件提供了物理机的封装，其目的是使上层的应用软件开发更加简单和可管理，即更高效、更安全和更可靠。



“虚拟”技术目前已经成为一个热点，所谓的“云计算”、“虚拟化”等内容都是与“虚拟”技术分不开的。“虚拟”技术产生的初衷就是提高“特定资源”的利用率和利用效率。由于各自针对的资源不同，目前市面上虚拟技术产品很多。想了解更多细节，请 Google “虚拟技术”。本书后面提到的虚拟机 VirtualBox(<http://www.virtualbox.org/>)就是典型的“虚拟”技术产品。

无论虚拟技术的产品如何多样，请大家记住：虚拟技术的初衷就是提高“特定资源”的利用率和利用效率；由于针对的资源不同，有很多不同的产品，但它们一定是介于两个逻辑层之间提供对下层的封装从而提高上层应用率和应用效率。

## 2.2. JVM 规范 (JSR924)

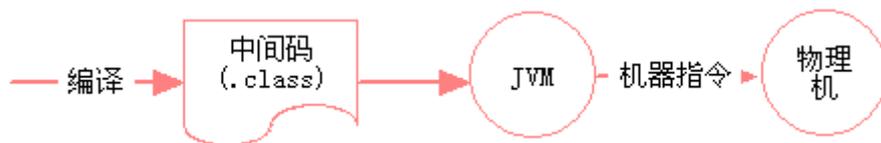
JVM 规范的提出是实现 Java “**一次编译，随处运行**”承诺的关键环节，正所谓“不成规矩，无以成方圆”，无论 JVM 的具体实现如何，但站在 JVM 之上的应用看到的都是一致的“接口”，即 JVM 规范。



JVM 规范 JSR 编号是 924, 官方地址 <http://www.jcp.org/en/jsr/detail?id=924>。随后的更新在 JSR202 <http://www.jcp.org/en/jsr/detail?id=202> 中维护。

*JCP (JAVA COMMUNITY PROCESS) 是 JAVA 的标准制定组织，由 JAVA 的重量级开发者和被授权的组织组成，JCP 维护的规范都简称 JSR，并且每个规范都会有一个编号，JAVA 几乎所有的内容都有其对应的 JSR。*

在了解 JVM 规范之前我们先从系统输入输出角度看看 JVM 的职能（抽象层面）：



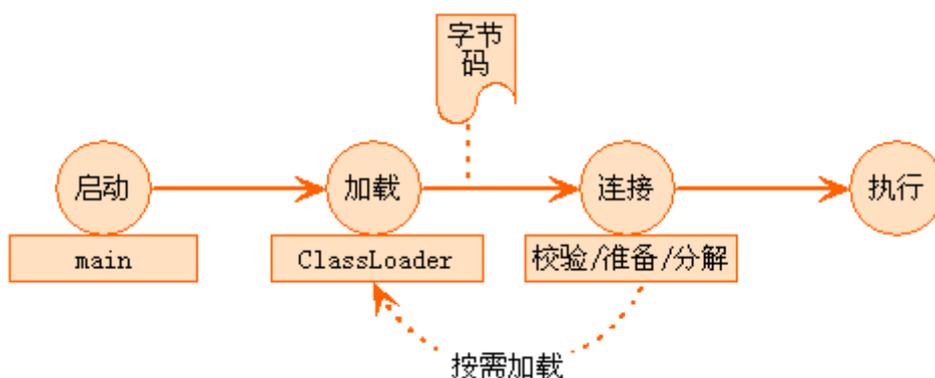
JVM 的输入是被称为 Java 字节码的中间码（在文件命名上通常以.class 为后缀，即 Java 源文件编译之后的文件），输出是机器指令集。如果我们通俗的理解 JVM 的主要职能就是负责将 Java 字节码翻译成机器指令。

*为什么不直接从 Java 源文件（在文件命名上通常以.java 为后缀）编译成机器码呢？首先这个问题的回答是：可以这么做，例如 GCJ(<http://gcc.gnu.org/java/>)、JNC (<http://jnc.mtsystems.ch>) 等工具就可以把 Java 直接编译成系统可执行程序。以前大家攻击 Java 速度和 C 有差距的主要问题就在于 JVM 这个环节。要谈引入 JVM 理由的话，我还是回到什么是虚拟机部分的内容，请读者自行思考。*

还是发扬本书的精神，给大家引入一些机器指令的相关知识：

我们经常看到的可执行程序（例如 Windows 上的 exe[PE 格式]和 Linux 上的 ELF 格式）与机器直接可执行的机器指令（例如引导盘主引导区中的程序）之间是有很大差别的，前者脱离操作系统是不能执行的，后者没有操作系统是可以执行的。换句话说，PE 和 ELF 格式需要操作系统进行翻译才能有效连接成机器指令，但最终和 CPU 打交道的还是机器指令。机器指令在语义层面通过一套被称作指令集（Instruction Set）的东东进行约定，更专业的称呼为 ISA（Instruction Set Architecture）。计算机架构（Computer Architecture）至少应该包括 ISA、微架构（Microarchitecture）和系统设计（System Design）。其中 ISA 是围绕微架构制定的，而在微架构层面各个 CPU 厂商在语义层面基本保持一致，换句话说大多数的 ISA 应该是具有相同的语义的，只是 CPU 内部的实现有所区别，但各个 CPU 之间还是有差别的，大家可以参考文章 <http://www.agner.org/optimize/microarchitecture.pdf> The microarchitecture of Intel, AMD and VIA CPUs（Intel，AMD 和 VIA 的微架构）。

### 2.2.1. Java 字节码程序的执行过程



图表 2 Java 字节码程序的执行过程

Java 程序在启动的时候首先交给 JVM（一个 JVM 的运行创建一个独立的进程）一个拥有 `public static void main(String[] args)` 函数的类，JVM 通过引导类加载器（Bootstrap ClassLoader）加载该类，加载完成之后进行校验、内存填充和结构化（即连接--校验/准备/分解），然后启用一个线程执行 `main` 函数（大多数可执行程序都有像 `main` 一样的入口函数约定，有些地方可以成为入口地址），在执行 `main` 函数的过程中碰到新的类，将会再次用 ClassLoader 加载该类，重复连接，并且继续执行 `main` 中调用的其它函数，依次反复，直到特定的终止条件发生，程序退出。

JVM 执行连接是典型的动态链接，也就是在执行过程中按需加载类，所有加载类的工作都交由 ClassLoader 来完成。在后面谈到的 **Java 字节码类文件格式** 就是 ClassLoader 加载之后交给虚拟机的字节码格式，换句话说 JVM 不管 ClassLoader 之前的类来源和格式，它只管 ClassLoader 加载之后的格式，理解这点很重要。

ClassLoader 分为系统级和用户自定义级两类，通过用户自定义类加载器，可以完成自定义的类加载方式（如何自定义请参考该章的**实战**部分）。

关于更多 Java 字节码的执行过程细节，在后面内容 **JVM 抽象架构** 中将会更进一步提到。

### 2.2.2. Java 字节码类文件格式

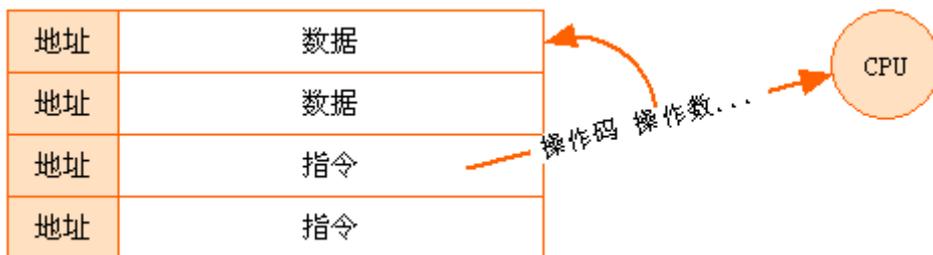
在学习该小节内容之前建议大家下载工具 JBE，<http://www.cs.ioc.ee/~ando/jbe/>，它是一个 Java 字节码编辑器，能够浏览和编辑 Java 字节码。在学习的过程中可以使用该工具亲自探索，从而加深理解。

Java 字节码类文件（.class）是 Java 编译器编译 Java 源文件（.java）产生的“目标文件”。无论使用的 Java 编译器具体如何实现，其编译之后的 Java 字节码类文件（.class）应该在任何 JVM 上运行。

记得大学教程里面的冯·诺伊曼（电脑之父啊）架构（当然现代计算机已经有了很多变化，例如非冯·诺伊曼架构系列等，但多为在冯·诺伊曼架构基础上的改进）吗？

*其实我一直在思考技术和哲学（不是技术哲学）之间的关系，当我们把技术刨根问底的时候，总是发现它是那么的哲学，难道真正的“圣人”就是一个刚出生的“婴儿”？其实在计算机运算上追求的是一种“精妙的简”（在大量采用二进制之前有采用 10 进制和 3 进制的计算机），一个要“表象万物”的东西一定要“简”，就像技术哲学中的原子假设一样可以预知未来（这不是算命吗？周易八卦？），这种“简”能够“自我复制和繁衍”（冯·诺伊曼在参与计算机研制之前搞原子弹制造）。*

冯·诺伊曼架构的存储程序提出程序是由一系列的指令和数据组成的，并且是线形地址编码存储的（**这个基本的归真思路，能帮助我们理解 C/C++，Java 等各种语言的编译、连接、执行的基本原理**）。



图表 3 冯·诺伊曼存储程序结构

上图表达的归真思路就是最终的可执行程序文件是扁平的（即线形地址编码的），程序执行的过程就是不断向 CPU 发送指令和 CPU 不断从存储设备中读取数据的过程，直到特定的终止条件发生。指令+数据更人性化地可以说是算法+数据结构（让我想起数据结构教程中的这句话，所以加进来了）。例如 Java 语言中类结构、变量、数组、引用、实例等就是“数据”，而各种操作符和控制符就是“指令”。

同样，Java 字节码类文件（.class）的格式包括两部分：数据+指令。

我们先来粗略的谈谈数据部分（即 Java 字节码类文件的格式，其中包含指令内容，但从文件格式层面来说指令也用数据来表示，正如后面内容介绍的指令=操作码+操作数）：

Java 字节码类文件由一系列的 8 位字节码流组成，16 位、32 位和 64 位都是通过 8 位字节码大数在前的方式（Big-endian，例如 16 位字符的表示前 8 位是高位，后 8 位是低位）表示。

Java 字节码类文件的具体结构如下（u2, u4 分别代表无符号两字节、四字节空间，符合 Big-endian 表示）：

头	ClassFile {
静态池（类似索引）	u4 magic;
访问控制	u2 minor_version;
类本身	u2 major_version;
父类	u2 constant_pool_count;
实现的接口	cp_info constant_pool[constant_pool_count-1];
属性	u2 access_flags;
方法	u2 this_class;
元数据	u2 super_class;
	u2 interfaces_count;
	u2 interfaces[interfaces_count];
	u2 fields_count;
	field_info fields[fields_count];
	u2 methods_count;
	method_info methods[methods_count];
	u2 attributes_count;
	attribute_info attributes[attributes_count];
	}

图表 4 Java 字节码类文件格式结构

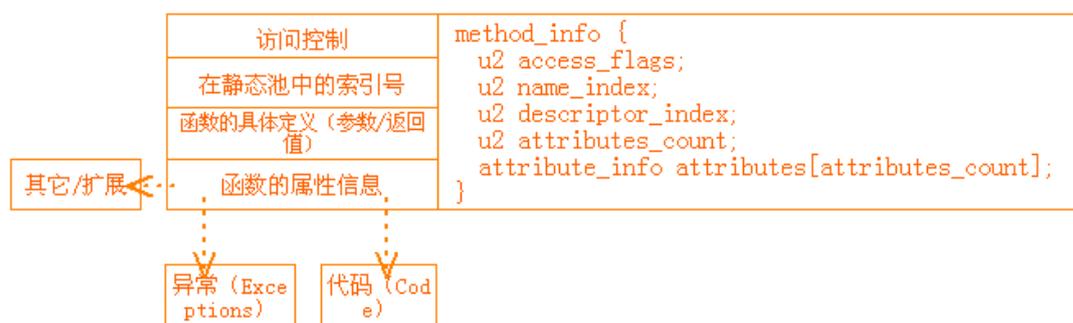
通过上面的结构我们可以了解 Java 字节码类文件的结构，我不打算逐一介绍（在 JVM 抽象架构部分会更进一步提到该内容，如果首次接触不是很理解，可以框架性的了解即可），只对重点但又难以理解的内容进行补充说明，在抓住重点和基本框架的基础上细节可以通过教程 JVM 规范第二版 [http://java.sun.com/docs/books/jvms/second\\_edition/html/VMspecTOC.doc.html](http://java.sun.com/docs/books/jvms/second_edition/html/VMspecTOC.doc.html) 查阅。

在图表 *Java 字节码类文件格式结构* 中最不容易理解但最重要的是静态池，我们可以把静态池理解为在图表 *冯·诺伊曼存储程序结构* 中提到的扁平线性地址编码的一种方法（基本每一种可行文件格式都有静态池的概念）。静态池是一个类的结构索引，其它地方对“对象”的引用可以通过索引位置来代替，我们知道在程序中一个变量可以不断地被调用，要快速获取这个变量常用的方法就是通过索引变量。这种索引我们可以直观理解为“内存地址的虚拟”。我们把它叫静态池的意思就是说这里维护着经过编译“梳理”之后的相对固定的数据索引，它是站在整个 JVM（进程）层面的共享池。

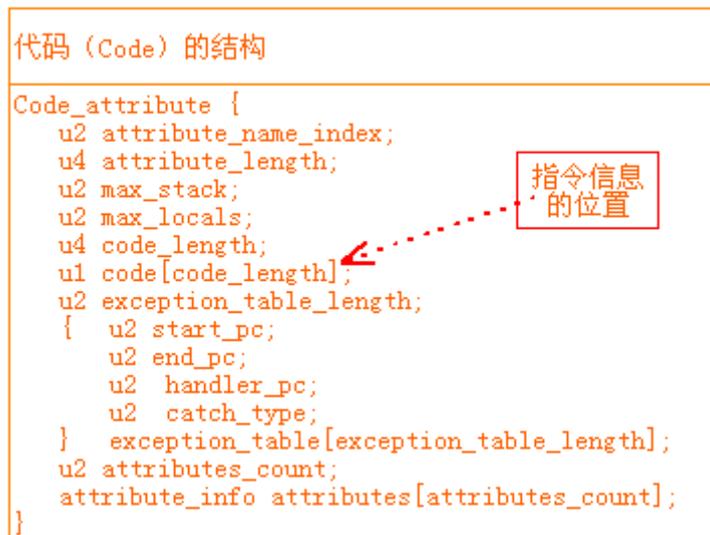
其次容易混淆的是图表 *Java 字节码类文件格式结构* 中的元数据（英文单词 Filed 和 Attribute 翻译有点让人迷惑）概念。元数据是一个给类添加额外说明信息的地方，例如把某类标识为作废等信息，在 Java 5 中提出的元数据（Metadata）都存储在这个结构中。

大家好奇的是到目前为止我们还没有看到指令在那里（应该在方法中吧？的确如此）。我们无论写任何程序，通常用方法（Method）来做某一件事情，通过嵌套或逐个调用方法来完成一系列的事情，例如 C 语言从 main 方法开始执行，Java 也从 main 方法开始执行等。那么指令一定在方法中（当然在 Java 中还有静态执行块的概念，但也以一个类的构造方法来对待，即也在方法范畴）。

那么我们先看看方法在 Java 字节码类文件中的结构（从图表 *Java 字节码类文件格式结构* 的方法中“引用”如下结构）：



图表 5 方法在 Java 字节码类文件中的结构



图表 6 Java 字节码类文件方法结构中代码的结构

好，对 Java 字节码类文件 (.class) 的格式就介绍到这里。具体细节请阅读该章的 [参考资料](#) 部分。

为了让大家深入理解基于 Java 字节码的应用，大家可以研究如下 Java 字节码的开源库（这些开源库都是在充分理解 **Java 字节码类文件格式** 的基础上编写的）：

- BCEL, <http://jakarta.apache.org/bcel/>
- ASM, <http://asm.ow2.org/>
- CGLib, <http://cglib.sourceforge.net/>

### 2.2.3. JVM 指令集

一个指令由操作码 (Opcode) 和操作数 (Operand) 组成。在 JVM 规范中对操作码给出了一个语清单和操作数的说明，即我们说的指令集。

*题外话：*

*指令集可以说在计算机世界中无处不在，而我们一般说的是 CPU 的指令集。CPU 是依靠指令来计算和控制系统的，每款 CPU 在设计时就规定了一系列与其硬件电路相配合的指令系统。现阶段的主流指令集可分为精简指令集 RISC (reduced instruction set computing) 和 CISC (complex instruction set computing)。涉及到任何特定芯片编程，了解芯片的指令集是必修课程。*

Java 指令集可以分为如下 8 大类：

- 堆栈操作 (Stack Operations)
- 算数操作 (Arithmetic Operations)
- 流程控制操作 (Control Flow)
- 导入和存回操作 (Load and Store Operations)

- 属性操作 (Field Access)
- 方法调用 (Method Invocation)
- 新建对象 (Object Allocation)
- 转换和类型检查 (Conversion and Type Checking)

其它具体细节内容，请查阅该章提供的[参考资料](#)。我下面给一个简单例子帮助大家理解指令：

Java 源代码：

```

1 public class Class1 {
2     public static void main(String[] args) {
3         int a=2;
4         int b=3;
5         int c=a+b;
6         System.out.println("c:"+c);
7     }
8 }

```

编译后方法 main 的 Java 字节码(发现 JVM 的好多指令没有操作数吧？这就是后面提到的基于堆栈的操作约定带来的操作码简化)：

```

1 0 iconst_2
2 1 istore_1
3 2 iconst_3
4 3 istore_2
5 4 iload_1
6 5 iload_2
7 6 iadd
8 7 istore_3
9 8 getstatic #16 <java/lang/System/out Ljava/io/PrintStream;>
10 11 new #22 <java/lang/StringBuilder>
11 14 dup
12 15 ldc #24 <c:>
13 17 invokespecial #26 <java/lang/StringBuilder/<init>(Ljava/lang/String;)V>
14 20 iload_3
15 21 invokevirtual #29 <java/lang/StringBuilder/append(Ljava/lang/String;)Ljava/lang/StringBuilder;>
16 24 invokevirtual #33 <java/lang/StringBuilder/toString()Ljava/lang/String;>
17 27 invokevirtual #37 <java/io/PrintStream/println(Ljava/lang/String;)V>
18 30 return

```

在上面的代码中我们看到了类似<java/lang/StringBuilder/<init>(Ljava/lang/String;)V>的信息，它们是在 Java 字节码中被称为 Descriptor 的类型表示格式，在后面内容 JNI 中本地代码调用 Java 代码中出现的名称，都是来自该约定。具体约定如下：

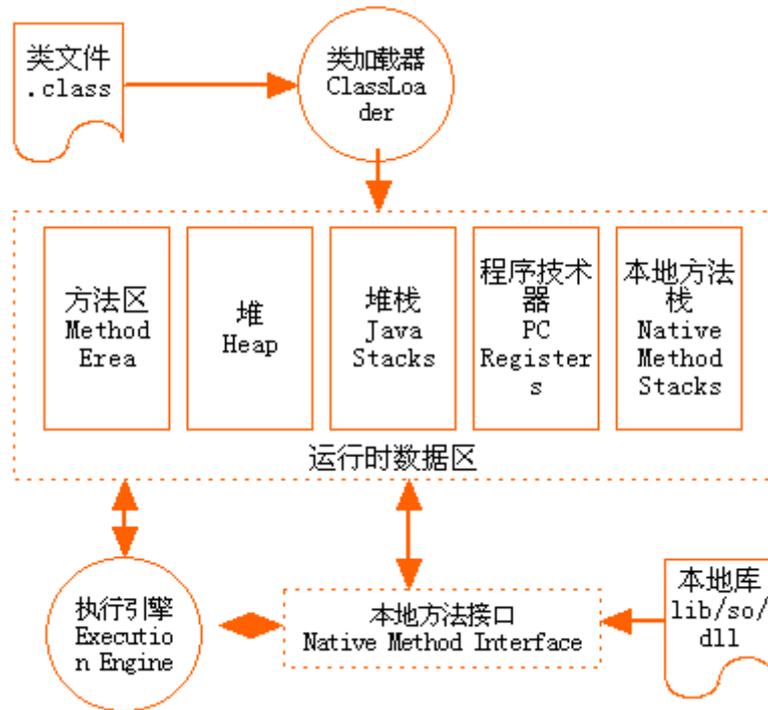
- 简单数据类型通过各自对应的符号来表示，B 表示 byte，C 表示 char，D 表示 double，F 表示 float，I 表示 int，J 表示 long，L 后面跟类全名称，再加分号;表示引用，S 表示 short，Z

表示 boolean, [表示数组; 类用反斜杠隔开的全名称, 例如 java/lang/StringBuilder。例如 String[] 可以用 [java/lang/String; 来表示, int[] 可以用 [I 来表示, int[][] 可以用 [[I 来表示。

- 方法用方法名(顺序参数类型)返回值类型表示。例如 int[] method(int a,String b) 可以表示为 method([Ljava/lang/String;)[I。

### 2.2.1. JVM 抽象架构

在学习该小节内容之前请大家先仔细理解下图所表达的内容:



图表 7 JVM 抽象架构

如上图, JVM 抽象架构的核心是运行时数据区(内存管理)的抽象架构(这里说抽象所表达的另外意思是不同的 JVM 实现可能有不同的实现策略)。当有了合理的内存管理策略, 程序的执行仅仅是指令序列的推送。

其中方法区(Method Area)是对静态类文件结构的内存维护, 不同的实现有不同的维护方法。

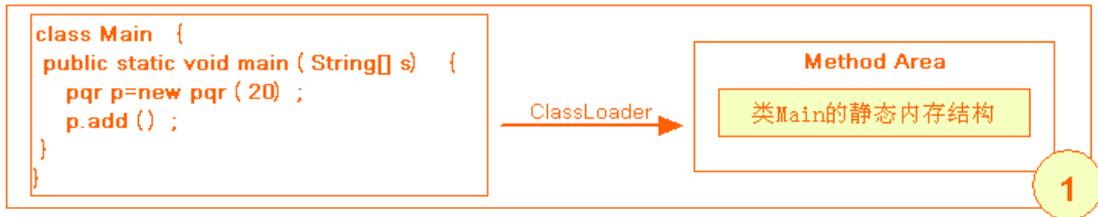
堆区(Heap)是对类实例的内存维护, 说起堆, 可以联想到“垃圾堆”, 其不可避免的就是乱, 在 **JVM 实现** 部分我们谈到的垃圾回收主要就是针对堆区的维护。

Java 堆栈(Java Stacks)是 Java 内存抽象架构的核心, 有人把 JVM 说是 **基于堆栈架构** 的原因就在于 Java 堆栈的概念。Java 堆栈要求 Java 程序执行中的每个线程都有一个独立的堆栈, 每个当前执行的方法是当前线程堆栈的一个片断(Frame)。

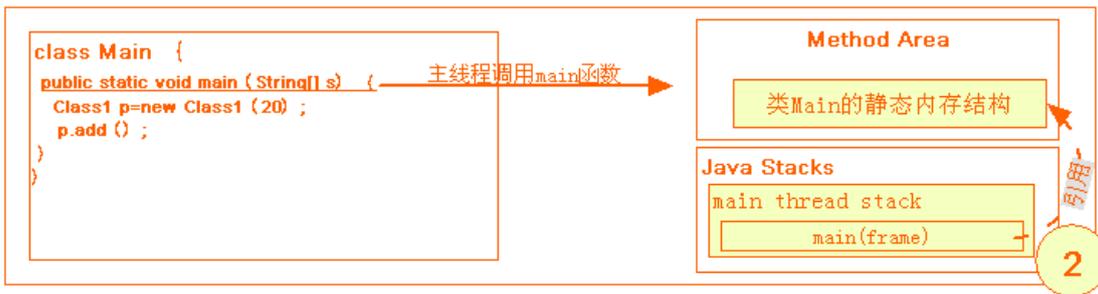
程序计数器（PC Registers），对于基于堆栈实现的 JVM，这几乎是唯一寄存器了，它用来指示当前 Java 执行引擎执行到哪条 Java 字节码，指针指向方法区的字节码。

本地调用堆栈（Native Method Stacks）是本地库的调用堆栈，用来实现 JNI（Java 本地接口，在本章 JNI（Java 本地接口）小节会介绍该内容）。

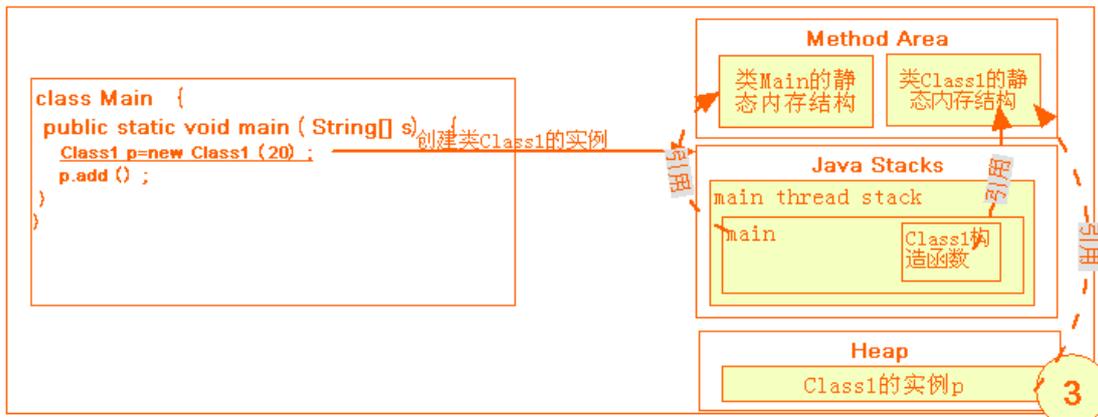
为了对抽象架构的进一步理解，我从一个具体的实例出发，通过简化图表方式帮助大家理解：



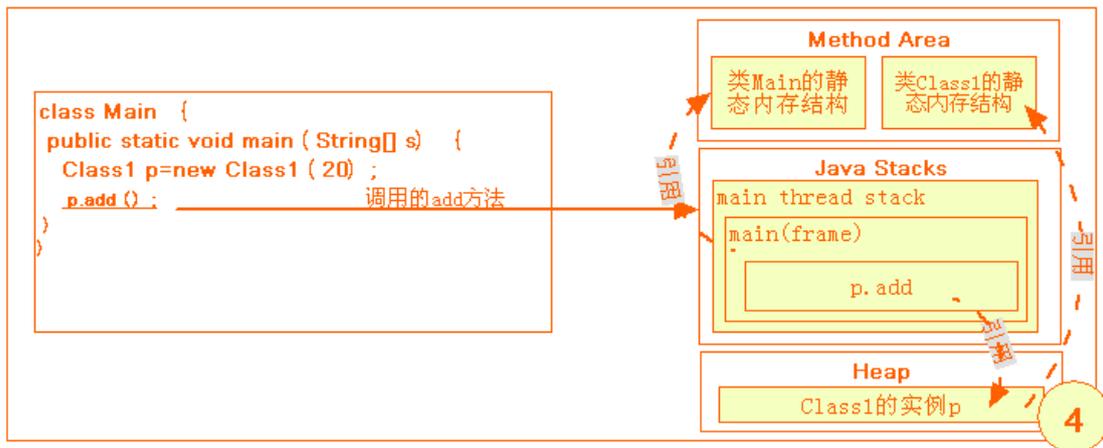
① JVM 通过引导类加载器（Bootstrap ClassLoader）加载类 Main 的字节码，通过校验之后为类在方法区建立静态内存结构。



②JVM 启动主线程执行 Main 的静态方法 main。将 main 以片断（Frame）的方式压入当前线程堆栈。



③JVM 通过类加载器加载类 Class1，并为 Class1 建立方法区内内存结构。然后调用 Class1 的构造函数，并将其压入当前线程堆栈。执行完毕构造函数之后把构造函数片断（Frame）从当前线程堆栈中移出。



④JVM 执行 Class1 的实例 p 的 add 方法，并且将其压入堆栈，当 add 方法返回，从当前线程堆栈中移出 add 片断（Frame）。

⑤主函数 main 执行完毕返回，当前线程堆栈清空，主线程执行完毕，进程直接退出。

从上面的粗略过程我们至少可以了解：

- Java 的方法区（Method Area）会随着类加载的数量不断地扩充。
- 堆（Heap）区会随着新建类实例而不断地扩充。
- 堆栈（Java Stacks）在程序执行中始终存在，不同的线程都有自己独立的执行堆栈。

下面我们再介绍一下 JVM 抽象架构中的另外一部分与我们日常编程息息相关的内容：多线程中的同步（解决并发带来数据不一致的一种方法，在 JDK5.0 之前唯一解决并发的原语就是同步）。

在前面我们已经看到堆（Heap）中的数据主要以“引用”的方式被堆栈使用，当多个线程对同一实例的数据进行操作时就可能发生数据不一致，我这里强调的主要是不一致，即超过一条指令对一个变量自身的连续引用操作（包括读和写）。

多线程中的数据同步在 JVM 规范中并没有强调具体的实现机制，从 Java 字节码类文件格式信息中我们只能看到对方法标注同步的地方和从指令集角度我们可以看到两条指令：194 (0xc2) `monitorenter` 和 195 (0xc3) `monitorexit` 来标注同步块的开始和结束。也就是我们写程序的时候在方法前面加 `synchronized` 和同步块 `synchronized(object){}` 的具体体现。至于 JVM 如何实现方法同步和块同步都由 JVM 实现者来完成。

在介绍如何同步之前，我打个大家一定能理解的“俗”比方：上厕所。

有一天你很内急，不断地跑厕所，老发现没位置，但你仍然坚持不懈地跑试图看看有没有位置。同时可能还有其他人和你一样，也内急，老跑。这时候你可能有如下选择：

- 告诉已占上位置的老哥们，那个完事了喊一下（通知 `Notify` 和等待 `Wait` 对应，但是广播性质的，人家通知了，你不一定就能抢到，因为可能等的人不止你一个）。

- 回去干干其它的事情，然后接着跑，撞运气（非阻塞策略）。
- 把老跑的人召集起来，排队等（队列阻塞等待策略 Wait）。
- 遇见领导了，让领导优先（优先级）。
- 其它...

其实上面的选择我们都可以理解为解决并发问题的策略。但无论怎么看，占位的那些老兄就是占着位，我们形象的称作加“锁”（Lock）。大家都是为了等“锁”打开（Unlock）从而争得“锁”（Lock）而奋斗（注：解决并发带来数据不一致的策略有很多，例如假定不变先操作共享数据，回去修改的时候看看是否和操作前一样，如果一样直接修改，如果不一样，重新来过等乐观非阻塞策略，这些策略已经脱离我们说的“锁”的概念，而是解决并发带来数据不一致的另外途径，我们这里只谈 JVM 层面“锁”的抽象架构，关于其它并发解决策略，在后面[多线程编程](#)中会更进一步介绍，大家可以跳跃阅读）！

JVM 抽象架构中要求对每一个对象在运行期加一个锁标识，也就是无论我们是用方法标注同步还是用同步块，其直接的体现就是给共享对象加锁或者等待对象解锁。这种加锁和等待解锁带来的问题就是线程的等待，甚至造成你等我，我等你的死锁。

从“锁”的基本工作原理和带来问题出发，建议大家在使用同步（synchronized）的时候注意：

- 注意对需要同步的共享数据的读和写保持对应。不要出现在非同步调用中读取同步调用中写的对象数据，反之也要注意。
- 注意尽量让需要同步的共享数据集中和少，并且不要开放直接对共享数据的非同步读或写的入口，把他们控制在同步的范围内。

## 2.3. JVM 实现

学习 JVM 实现一些初学者可能感到“高不可攀”，其实说实话我也不想去了解 JVM 实现的细节，但了解 JVM 实现中一些关键问题解决思路对我们“修炼”境界还是有帮助的。

在学完 [JVM 规范](#) 之后，我们可能有如下的问题需要获得答案：

- JVM 如何实现运行时数据区的管理（即内存管理）的？
- JVM 如何实现堆（Heap）的垃圾回收？
- JVM 如何实现对多线程的支持？在 JVM 规范中很少见到多线程的影子，都是谈同步。
- JVM 如何将 JVM 指令翻译成机器指令？

下面我们就针对这些问题展开一些讨论。

### 2.3.1. 内存管理和垃圾回收

Java 的一个重要特性就是自动化内存管理，给 Java 开发者带来的直接好处就是不用为何时释放 Java 对象而担心，同时也不用担心类型混乱造成的内存非法操作（大不了抛出一个 `ClassCastException`）。

我们已经知道，Java 字节码作为中间码引入的一个很重要特性就是任何引用都是标注类型的，即强类型，任何操作指令都是建立在强类型的基础上，这带来的直接好处就是每个操作都明确知道内存的块大小。强类型为 JVM 的内存管理奠定了坚实的基础。

JVM 内存管理主要解决的问题就是如何为新建的对象有效分配内存，如何将不再被使用的对象从内存中移出，如何将内存碎片整理（C/C++是程序员负责内存管理，而在 Java 中是 JVM 负责，换句话说就是人家帮我们解决了内存管理难题。因此学习 JVM 的内存管理可以帮助我们在 C/C++编程中进行有效的内存管理，在后面内容 **Java 和 C/C++**中我们会更进一步讨论这些问题）。

内存分配通常有两种方法：连续块和非连续块，其各自的优缺点在大学数据结构教程中有所介绍。连续块内存分配的算法相对简单，例如在当前空闲内存中寻找大于等于请求块的连续内存空间，然后分派给请求块。而非连续块内存分配就复杂些，其首先要解决的问题是如何将请求块分解成小块，然后通过指针关联起来，这种非连续块的分配方法既容易造成内存碎片又容易利用内存碎片，但很多问题都是大量内存碎片造成的，还是连续的好。

内存分配的简单过程如下：

- 首先计算需要分配的内存大小（对象数据结构的压缩程度和对象的切分方法都对实际分配内存的大小有影响），然后计算 JVM 当前拥有内存的大小，进行对比，如果够，分配，如果不够，向操作系统申请内存，如果达到 JVM 规定的上界，抛出错误。
- 其次，根据具体的内存表示结构分配内存。
- 最后把分配内存的信息写入一个维护表中（主要为垃圾回收提供信息）。

垃圾回收可以看作是自动化内存管理的另外一个直观称呼，其主要负责无用数据块的清理和碎片整理。

垃圾回收的核心是垃圾回收算法，下面我们简单介绍一些常用的垃圾回收算法（准确说是介绍思想，而不是算法细节），要理解细节请阅读该章的参考资料。

在介绍这些算法之前先明确一些概念：在前面我们说过任何程序的执行都是指令驱动的，所以要在堆(Heap)中开辟内存，也是指令驱动，并且我们也说过 Java 字节码的指令都在方法(Method)中，那对下面代码你可能会疑惑（其实就是 **Java 字节码类文件格式**中的内容细节，但我们在 **Java 字节码类文件格式**中并没有介绍，而是留给感兴趣的朋友阅读参考资料）：

```

public class Class1 {
    Class1 self=new Class1();
    String var;
    public static void main(String[] args) {
        int a=2;
        int b=3;
        int c=a+b;
        System.out.println("c:"+c);
    }

    public void add(){
        synchronized(var){
            var="test";
        }
    }
}

```

其中有不在方法中的代码：

```
Class1 self=new Class1();
```

其实类似上面的操作都是在编译期被放到一个叫做<init>的方法中，该方法在类加载成功之后被JVM自行调用。用前面提到的JBE工具浏览<init>方法的字节码代码如下：

```

1 0 aload_0
2 1 invokespecial #12 <java/lang/Object/<init>()V>
3 4 aload_0
4 5 new #1 <Class1>
5 8 dup
6 9 invokespecial #14 <Class1/<init>()V>
7 12 putfield #15 <Class1/self LClass1;>
8 15 return

```

明白了吧！总结一下：在JVM中任何对象的创建都在方法中进行，也就是在方法堆栈(Java Stacks)中，同样任何对象是否活动或者被引用的源头都是在堆栈（堆栈一定是活动的）中。那么我们判断一个对象是否被引用是否应该从堆栈中开始扫描呢？当然是。那我们开始了解下面的算法吧！

### 引用计数算法（Reference Counting）

为每个对象维护一个引用计数器，每当被引用的时候计数器加1，每当引用结束时计数器减1，垃圾回收线程定时扫描维护表，发现计数器为0的，进行回收。其最明显的问题是如何解决“我引用你，你引用我”的循环引用。

### 标记-清除算法（Mark-Sweep）

该算法包括两个步骤，第一步，从堆栈中开始扫描引用对象，对遍历到的对象进行标记；第二步，对没有标记的对象进行垃圾回收。

前两种算法是解决任何管理问题的常规思路，即主动注册和被动扫描。主动注册的成本在注册和注销，被动扫描的成本在扫描，因为扫描的时候要求堆栈暂且保持一段时间的静止。

### 复制算法（Copying）

该算法首先把内存分成两个大小相等的区，只有一个区有效。然后从堆栈开始扫描引用对象，把扫描到的对象复制到另一个区，当前区被认为无效，即可以回收。

复制算法和标记-清除算法在遍历阶段思路基本一致，其优点是直接对内存进行了碎片整理；其代价在于扫描时期暂停程序运行、复制成本和内存的闲置。

### 标记-压缩算法（Mark-Compact）

类同于复制算法，只是不将内存区分成两块，而是在现有内存中重新整理和分配，这种算法的难度在如何重新分配从而达到内存整理（即压缩）。

### 增量收集算法（Incremental Collecting）

前面我们说过，扫描要求程序某一时间静止，静止的时间直接影响程序的表现，增量收集算法是从切分任务的角度来使每次的工作负荷减小，使程序保持一定的顺畅连续感。

### 分代收集算法（Generational Collecting）

增量收集算法切分任务的思想更进一步的提升就是根据对象本身的特点进行切分，然后根据不同特点的对象采用不同的收集策略。分代收集算法就是把对象根据创建的时间进行分类，对不同寿命的对象进行不同策略的回收算法。目前 JVM 的实现基本都采用分代法。

还有其它的一些算法（最好叫方法吧，不然总感觉不严谨），例如并行、同步等。

上面的算法从解决问题的思路来看基本都有借鉴和可用之处，那么明智的选择就是“站在巨人肩膀上”了，根据 JVM 的应用场景充分利用各自的优点，避免各自的缺点。这也是我们看到一些 JVM 实现提供很多参数让我们根据具体情况进行优化配置的理由了，例如 Sun 的 HotSpot 给我们提供客户机（Client）和服务器（Server）两种虚拟机模式。既然这样，我们要做得就是评估我们的应用场景了。

那么如何直观评价一个垃圾回收机对应用场景的适应性呢？我们至少应该从如下几个方面去看：

- 是否安全，不能把不需要回收的搞掉吧？
- 是否快，不能让多数时间被垃圾回收占用吧？
- 是否有效整理内存，不能让内存无限地混乱膨胀吧？
- 伸缩性如何？不能遇见单核不错，遇见多核就晕了吧？除非就是针对单核的。
- 当然还有其它。

垃圾回收就介绍到这里吧！希望把你引进门，修行还得靠个人，抓住这个思路你可以进一步做如下深入研究：

- 前面提到的算法具体是什么？你可以参考该章的参考资料，可能会帮助你，除外，你还可以用前面内容中的算法的中文或英文进行搜索，获得更多资料。
- 研究某个具体的虚拟机的垃圾回收机制，至少可以研究一下各自的垃圾回收算法组合应用思路，例如 Sun 的 HotSpot。

如果学完该节内容之后感觉空空的，请阅读该章的**实战**部分。

### 2.3.2. JVM 实现中的多线程

在 **JVM 抽象架构** 中我们谈到了多线程中解决数据不一致问题的同步锁概念，从 Java 字节码交给 JVM 的信息来看除了在方法上标注 `synchronized` 和两条同步指令（194 (0xc2) `monitorenter` 和 195 (0xc3) `monitorexit`）外，真还没有见到任何字节码可以告诉 JVM 何时启动一个线程相关的信息，但 Java 中确实是支持多线程的，这是怎么回事？

我们摘 JVM 规范中的两段关于线程的描述：

While most of the preceding discussion is concerned only with the behavior of code as executed by a single thread, the Java virtual machine can support many threads of execution at once. These threads independently execute code that operates on values and objects residing in a shared main memory.

Threads are created and managed by the classes `Thread` and `ThreadGroup`. Creating a `Thread` object creates a thread, and that is the only way to create a thread. When the thread is created, it is not yet active; it begins to run when its start method is called.

翻译：

虽然在规范中我们一直在讨论单线程的执行过程，但 JVM 是可以支持多线程的，每个线程都能够独立地执行自己的代码，操作共享主内存中的数据。

线程由类 `Thread` 和 `ThreadGroup` 进行管理，通过创建 `Thread` 对象来创建一个线程，通过 `Thread` 的 `start` 方法启动线程。

总结一下：

- 首先 JVM 规范不强制每个 JVM 都必须实现多线程（Java SE 是要求的）。
- 如果来实现，请从 API 规范层面在 `Thread` 和 `ThreadGroup` 类中实现。
- 只要实现，必须要在 JVM 层面实现方法同步和同步块的语义（这句话是否怪怪的？但这是我觉得最重要的一句话）。

前面的总结我们可以更进一步的总结：只要实现 API 层面的类 Thread 和 ThreadGroup，就必须是 JVM 级别的实现，因为它必须要和 JVM 直接交流。这也是为什么该小节内容放到 **JVM 实现** 和该小节的题目为 **JVM 实现中的多线程** 的原因。

我们接着说 Thread 和 ThreadGroup 两个类，类在 JVM 规范层面来说它编译之后一定是 Java 字节码，但我们又知道 Java 字节码中根本没有线程的信息，这并不奇怪，在 JVM 抽象架构中我们提到了 JNI（Java 本地接口）的概念，其实现就是通过 JNI 的。

看看 Sun JDK（现在是 OpenJDK）对 Thread 的 start 方法实现的 Java 代码：

```
public synchronized void start() {
    if (threadStatus != 0)
        throw new IllegalThreadStateException();
    group.add(this);
    start0();
    if (stopBeforeStart) {
        stop0(throwableFromStop);
    }
}
private native void start0();
```

在 start0 前面有一个 native，明白了吗？关于 JNI 的内容请参考 **JNI（Java 本地接口）** 小节的内容。

一般情况下大多 JVM 实现不会自己来编写多线程机制，基本都是借助操作系统的多线程能力，JVM 要做的就是如何使用操作系统的多线程机制实现 JVM 规范层面的语义约束和 API 层面的多线程要求。

### 2.3.3. 从 JVM 指令到机器指令

我们前面也聊过 JVM 在规范层面引入字节码的好处，但将 Java 字节码翻译成机器指令是需要时间的。所以如何提高 Java 指令到机器指令的转化效率成为每一个 JVM 实现必须考虑的问题。如果发挥我们的想象力来理解应该有如下方法：

- 把 Java 字节码直接再次编译成本地可执行代码，执行的时候就没有 JVM 的事了。
- JVM 在运行期负责不断地解释了。

把 Java 字节码直接编译成机器码在有些应用场景中不是不可行，可以借助 GCC 的 GJC 或者其它编译器，但我们会牺牲很多 Java 的特性。用 Java 语言风格写出来的程序始终带着 Java 的风格，如果是特别庞大的 Java 项目基本是不可行的。换句话说，在这个地方我们就没有必要充当高手了，还是乖乖地研究一下 JVM 运行期的 JVM 指令解析吧。

但上面的两个方法又可以进一步的带出如下思路：

- JVM 可以在启动时一次性把所有的方法中的 Java 指令全部翻译成本地代码，这样启动的时候慢一点，但运行的时候快一点。
- JVM 一边运行一边翻译了，运行的时候慢了些。

其中第一种方法就是大家经常说的 JIT (Just In Time)。但一般一个 Java 项目不知道有多少方法要翻译，并且有些翻译也许根本用不到；还有如果是一个客户端的应用，让用户启动的等待就很吓人，体验不免有些牵强。

第二种方法是最朴素的了，还用谈？

一般情况下现在的 JVM 实现都是采用 JIT (其实.NET 也是)，但会做如下的一些优化，可以简单描述为：

- 每个方法被调用一次就给这个方法计数加 1。
- 那些方法的被调用计数越多，JVM 就优先翻译成机器码，当然少的可能在执行到的时候再翻译。

## 2.4. JNI (Java 本地接口)

当我们了解 JVM 大量的优点之后，不仅学习 Java 语言的朋友很激动，非 Java 语言的朋友可能也有些激动。例如熟悉 C 的朋友一定喜欢上 Java 的“高级”语言特性，所以考虑把一些繁琐的对性能要求不是很高的程序功能用 Java 开发，而把一些对运算性能要求高的用 C 开发（**注意**：这句话是我为了吸引大家写的，对于性能往往存在想当然的陷阱，请不要把 JNI 理解成来解决 Java 性能瓶颈的途径，即使有人这么做，我们也要建立在测试数据的基础上）。那么大家一定会遇到如何在 C 中调用 JVM 和 Java 程序，如何在 Java 中调用 C 程序。那么让我们从 JNI 开始吧。

JNI 也从规范层面有所定义，可以参考

<http://java.sun.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html>。

在学习之前我们还是发挥一下我们自己的想象。

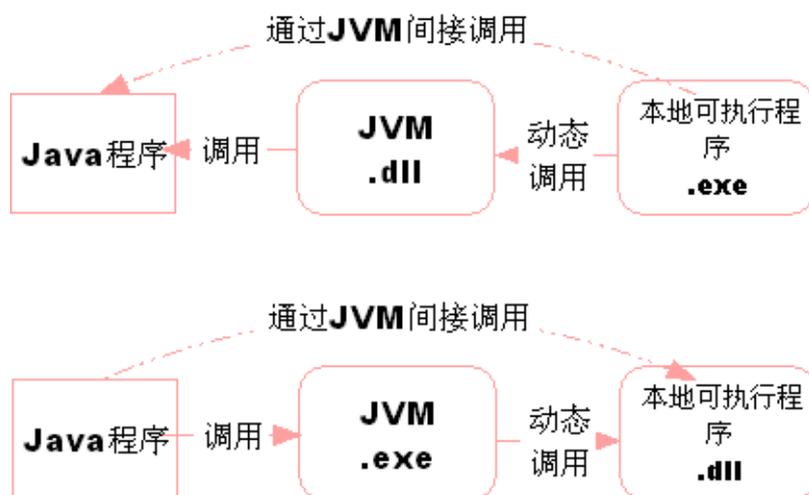
JNI 解决的问题就是程序和程序之间的互调。我们可以想象的方法有：

- 进程之间的调用。进程之间的调用可以通过管道技术和网络技术来实现（Java 也可以做到进程之间的调用）。
- 进程内调用。让不同来源的程序都在本地代码范围内在一个进程中进行调用。

我们谈的 JNI 属于进程内调用。那么让我们再想象一下进程内调用的一些条件。

大家知道静态链接(Windows 的.lib 和 Linux 的.a)和动态链接的主要目的是将可执行代码块链接到一起共同完成程序特定功能。静态链接在编译期完成，动态链接在程序执行过程中完成。动态链接技术主要解决程序在执行过程中如何通过特定的编址和寻址方式加载可执行代码(即如何将不

连续的块组成我们前面谈到的冯·诺伊曼程式），所以一般从规范层面进行约定，即我们常说的动态链接库技术，在 Windows（.dll）和 Linux（.so）上都可以采用动态链接技术来设计程序，但它们的调用方式以及程序编制方式不尽相同（从这一点我们可以明白我们为什么叫 Java 本地接口了吧？原因是这些东东与操作系统千丝万缕）。如果我们要达到进程内调用的目的，看来只能通过动态链接技术了，而动态链接的前提条件是相互调用的模块都是可执行的。



图表 8 Windows 下本地程序和 Java 程序互调

我们通过上图基本能明白 JVM 在互调中的中介作用，首先 JVM 是在操作系统上可执行的程序，所以满足和本地库互调的条件，同时 JVM 理解 Java 字节码，所以能够充当中介角色，JNI 就是针对中介制定的一套约束，从而保证 Java 和本地代码之间的沟通顺畅。

**注：**在 .NET 中与 JNI 可以类比的的就是托管和非托管代码的概念。

通过前面我们自己的基本逻辑想象，我们不难概括出 JNI 的主要内容：

- Java 程序如何让 JVM 加载本地库，并且让对本地库的调用符合 Java 语法。
- Java 中的数据类型和本地库中的数据类型如何映射。
- 本地程序如何让 JVM 调用 Java 程序，并且让对 Java 程序的调用符合本地库语法（例如 C/C++）。

### 2.4.1. Java 中使用本地库

我们从一段代码出发：

```
1 package pkg;
2 public class Cls {
3     native double f(int i, String s);
4     static {
5         System.loadLibrary("pkg_Cls");
6     }
7 }
```

上面代码通过 `native` 关键字告诉 JVM 该方法为本地方法，`System.loadLibrary` 告诉 JVM 应该导入本地库 `pkg_Cls`（在 Windows 环境下将查找 `pkg_Cls.dll`，在 Linux 环境下将查找 `pkg_Cls.so`）来填充 `native` 关键字声明的方法。`native` 和 `System.loadLibrary`（当然 Java 还可通过 `RegisterNatives` 支持静态连接，这里不详细介绍）的组合告诉 JVM 将标注 `native` 的方法和导入库中的本地方法映射起来。这样对 `Cls.f` 方法的调用将不用关注更多的本地细节，仍然站在 Java 高级特性的高度。那 JVM 是如何将 `native` 标注的方法和本地库中的方法映射的呢？答案是通过命名约定。

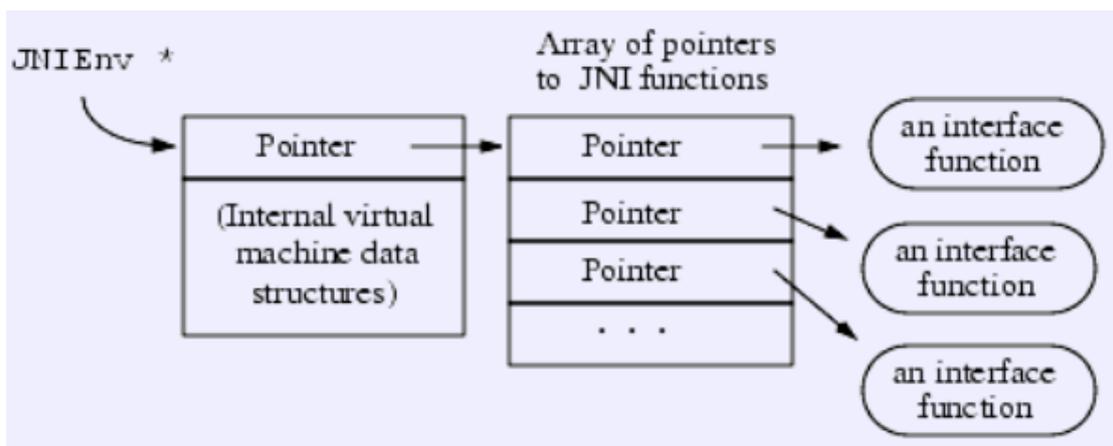
命名约定如下：

- 首先在方法前面加 `Java`。
- 然后用 `_` 分开，再加包名称。如果包名称中有非 ASCII 用 `_OXXXX` 形式的 Unicode 编码表示。
- 然后用 `_` 分开，再加上方法名称。
- 当出现 1 个以上函数名相同的时候，将用两个 `_` 将参数类型连接起来，参数类型的命名规则在 JVM 规范中有介绍，请大家参考 JVM 的规范文档。
- 其中特殊字符 `_` 用 `_1` 代替，`;` 用 `_2` 代替，`[` 用 `_3` 代替。

例如上面的类 `Cls` 中的 `f` 方法将映射为：`Java_pkg_Cls_f`。

OK，这就是方法名称映射的约定。

我们再回到一个方法（很多人叫函数），一个方法一般有返回值、方法名称和参数三个要素，在 JNI 的一篇“最佳实践”文档（可以参考 <http://www.ibm.com/developerworks/cn/java/j-jni/>）中建议不要让本地代码反向和 Java 代码互动太多，即一般情况下 Java 调用本地库不建议本地代码反向调用 Java 代码，本地代码仅仅对方法参数进行理解 and 操作即可。因此，我们的重点是解释在本地代码中如何操作参数、数据类型映射。数据类型映射请参考小节数据类型映射，这里主要说一下本地代码如何操作参数，先看下图：



前面我们说过，本地代码对 Java 代码的操作必须要经过 JVM，特别是对引用对象的操作，上图中的 `JNIEnv` 就是 JVM 提供给本地代码的一个虚拟机 Struct，即对引用对象的操作必须要经过 `JNIEnv`。把这一点记在脑子里，在该章的**实战**部分类似 `(*env)->Method` 的调用你就明白什么意思。

其实 Java 调用本地代码的核心框架内容就这么多，在框架的指导下大家可以借助该章提供的参考资料和该章实战部分开始动手了。

### 2.4.2. 数据类型映射

如果你已经有了 JDK，请在 JDK 目录中找到 `include/jni.h`，它里面定义了所有 JNI 的数据类型。当然这需要你稍微懂一点 C/C++ 的知识，对 C/C++ 不了解但又感兴趣的朋友可以恶补一下相关知识，这对我们理解 JNI 是很有帮助的（在 **前言** 部分提供过一个学习 C 的老掉牙教程）。

这块我不打算详细进行阐述，只要搞清楚值数据类型和引用数据类型的区别即可，同时在调用时注意 `j` 开头的 C/C++ 数据类型具体对应 C/C++ 的什么数据类型即可（可以通过 JNI 规范和 `jni.h` 获得对应关系），这是核心。在该章的 **实战** 部分我会举个例子加深大家的理解（你可以跳着阅读 **实战** 中的例子）。

### 2.4.3. 本地程序如何调用虚拟机和 Java 代码

这是一个有趣的话题，但在我们前面阐述的基本思路框架下，我们理解这个并不难，除了一个反向的 JVM 建立之外，和前面说的内容基本是一致的。

用 C/C++ 写程序的朋友只要知道本地程序调用虚拟机和 Java 代码都是通过动态链接技术先启动虚拟机，然后通过虚拟机调用 Java 代码便可动手写程序（当然也可以静态连接，唯一的前提条件就是要理解 `jni.h` 中定义的数据结构和每个方法的使用）。在该章的 **实战** 部分会给大家具体的例子。

## 2.5. 实战

说了那么多，大家可能晕晕的，我自己也晕晕的。下面我们来动动手吧！

**注意：**我们在每一章都有一个实战环节，大家在阅读一些基本知识的时候可以同时参考实战，带着实际应用的需要去学习基本知识是一件很愉快的事情。

实战前的开发包和工具准备：

- 从 <http://java.sun.com/javase/downloads/widget/jdk6.jsp> 下载 Sun 的 JDK。具体安装过程比较简单，请自行解决。
- 弄个文本编辑器来编写 Java 代码，或者下载 Java 的 IDE，推荐从 <http://www.eclipse.org/downloads/> 下载 Eclipse IDE for Java Developers。安装过程简单，请自行解决。
- 从 <http://www.cs.ioc.ee/~ando/jbe> 下载 Java 字节码编辑工具 JBE。
- 从 <http://www.codeblocks.org/downloads/binaries> 下载 `codeblocks-*mingw-setup.exe` 作为 C/C++ 开发工具，其中自带了 Mingw 的 GCC 编译工具。如果你有 Visual Studio 那就不用下了。

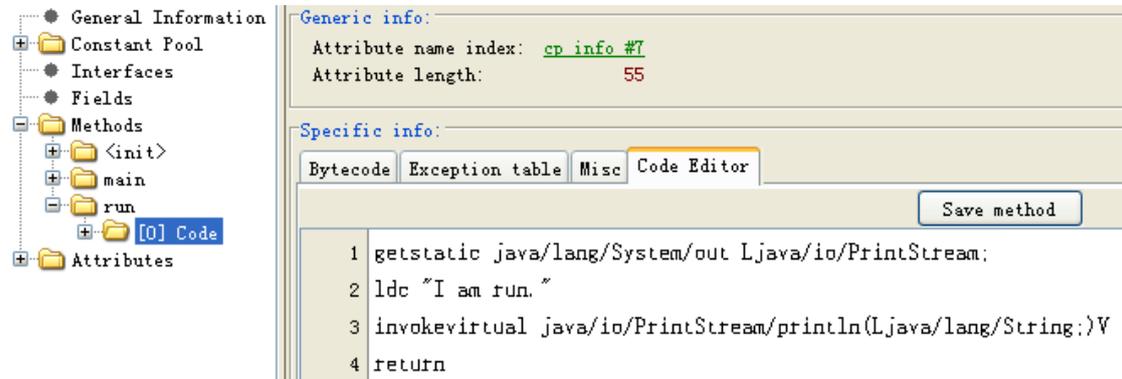
### 2.5.1. 用 JBE 修改 Java 字节码

如下图，我们把左边的 Class1 代码编译后的.class 文件修改成右边源代码的输出结果（用 AOP 的说法就是拦截方法 run，在 run 运行之前做件事情）：

```
1 public class Class1{
2     public static void main(String[] args) {
3         Class1 clazz=new Class1();
4         clazz.run();
5     }
6     public void run() {
7         System.out.println("I am run.");
8     }
9 }

1 public class Class1{
2     public static void main(String[] args) {
3         Class1 clazz=new Class1();
4         clazz.run();
5     }
6     public void run() {
7         System.out.println("Before call run.");
8         System.out.println("I am run.");
9     }
10 }
```

先把源文件编译，用 JBE 打开编译后的 Class1.class 文件，找到 Methods-run-[0]Code，如下图：



把 run 方法的字节码修改为下图内容，保存：

```
1 getstatic java/lang/System/out Ljava/io/PrintStream;
2 ldc "Before run call."
3 invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
4 getstatic java/lang/System/out Ljava/io/PrintStream;
5 ldc "I am run."
6 invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
7 return
```

OK，我们的任务完成了。

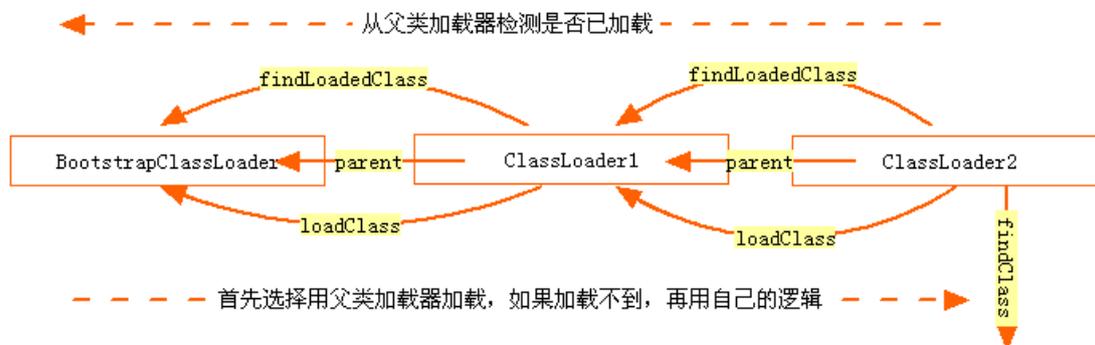
**借题发挥：** 我们经常会遇到需要搞明白编译后的 Java 字节码具体干什么但又没有源文件，我们有两种选择：

- 阅读字节码，阅读懂的前提条件是了解 **Java 字节码类文件格式**和 **JVM 指令集**。
- 用 Java 字节码反编译器，即让编译器帮你做了读懂字节码的工作。推荐一个命令行反编译器 JAD(<http://www.varanekas.com/jad>)。

### 2.5.2. ClassLoader 的应用

在 JVM 规范抽象架构中我们提到 `ClassLoader` 负责加载 Java 字节码交给 JVM，为了给下面应用更进一步提供基础，我还是快速的介绍一下 `ClassLoader` 的工作机制。

`ClassLoader` 常规工作机制如下：



图表 9 `ClassLoader` 常规工作机制

对上图进一步解释：

- 每个 `java.lang.ClassLoader` 的子类都允许指定一个 `parent`（父加载），如果不指定将直接使用系统级 `ClassLoader`。
- 每个 `java.lang.ClassLoader` 的子类在 `loadClass` 的时候都应该先检测自己的父加载器是否已经加载了要求加载的类，如果加载直接返回。
- 如果没有加载，首先应该用自己父加载器的 `loadClass` 去加载，如果父加载器还没有加载成功，就直接调用自己的 `findClass` 方法来查找类。
- 如果自己的 `findClass` 还加载不成功，抛出 `ClassNotFoundException`。

一般情况下建议不要破坏这种树形加载机制，因为它维护比较完整的类加载器树，被认为是类加载策略的“最佳实践”，要自定义请直接重写 `findClass` 方法即可。我们现实中常碰到的类加载问题（例如加载的类并不是你想要加载的；在同一个应用场景中同一个类多次被加载；在一些 `Java EE` 应用服务器中没有充分共享应该共享的类库等）都是对常规加载机制的理解不到位或破坏了常规类加载机制造成的。但当碰到一些特殊需求时（例如在一个 `JVM` 进程中一个类有不同的版本需要在父加载器和本身加载器中并存时），还是可以改变常规加载策略的，通常通过直接重写 `loadClass` 方法来完成。

好，我们回到 `ClassLoader` 的应用！通常对 `ClassLoader` 的应用有：

- 热部署。
- 软件保护。
- 动态生成 Java 字节码。
- 非常规的字节码加载。

下面我仅仅以 Tomcat6 的 WEB 应用 ClassLoader 为例介绍热部署，其它的留给大家通过过互联网资源去研究。

## 热部署 (Hot Deployment)

热部署其实并没有什么神秘的，最差的热部署可以通过自动重启 JVM<sup>©</sup>。但我们一般意义上说的热部署指的是在不影响当前 JVM 中其它应用的前提下只对需要重新部署的程序进行局部更换。

其实热部署中采用 ClassLoader 主要是解决热部署中类重新加载的问题，而不是用 ClassLoader 来完全实现热部署。

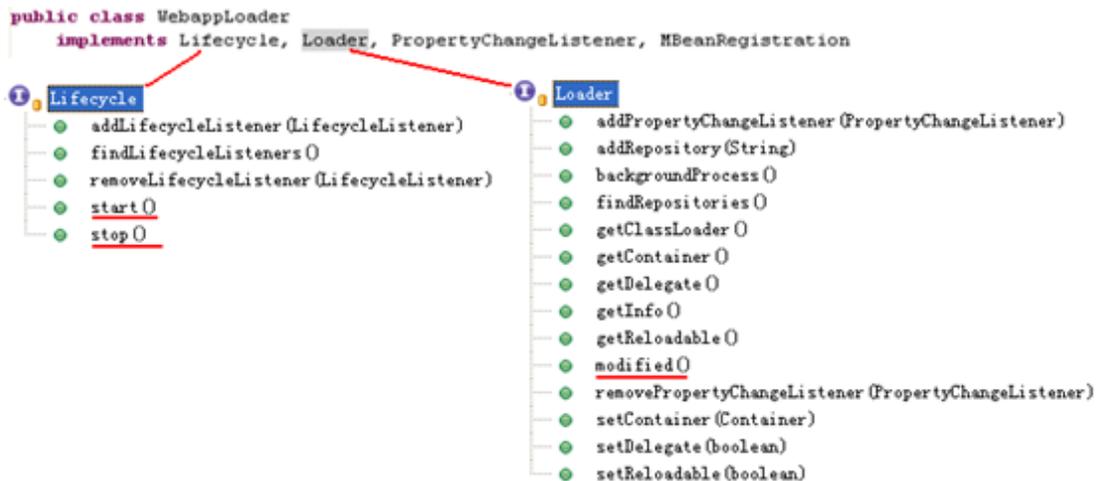
我们以 Tomcat6 的 Web 应用 (WAR) 热部署为例看看其实现的基本原理。

在我剪辑代码说思路的时候，大家可以通过在线浏览源代码

(<http://svn.apache.org/repos/asf/tomcat/tc6.0.x/trunk>) 了解详细内容，当然你可以 Check out 到你的 IDE (例如 Eclipse, 需要安装 subclipse)。

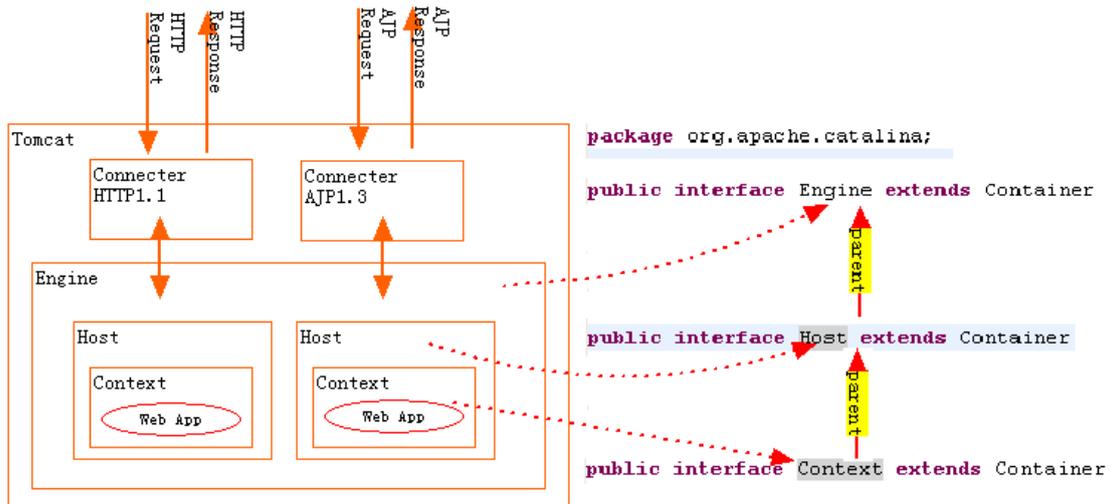
**注：**在大家读代码时看到 SecurityManager 相关内容的时候可以跳过，仅仅带着它是进行安全管理理解即可，在后面内容 **Java 安全** 部分我们会详细介绍该内容。

Tomcat6 的 Web 应用加载实现在包 org.apache.catalina.loader 中，主要通过类 WebappLoader 来实现。



我们谈 Tomcat 的 Web 应用热部署，就是谈一个 WAR 的变化驱动 Tomcat 如何重新加载 Web 应用，那至少应该有一个监听变化的线程吧？要搞清楚那个线程做到对 Web 应用变化的扫描，我们还不得不提一下 Tomcat 的内部架构 (参阅

<http://tomcat.apache.org/tomcat-6.0-doc/architecture/overview.html>) :



图表 10 Tomcat 的内部架构

从上图我们可以看到一个 Web 应用肯定有一个 Context 维护它，一个 Context 肯定有一个 Host 维护它，一个 Host 肯定有一个 Engine 维护它。那么监听 Context 变化的线程启动可能在 Host 和 Engine 里面，因为在 Context 级别启动一个线程做热部署这件事发生频率不是很高的事情实在有点浪费。那么我们先看看 Host 的标准实现类 `org.apache.catalina.core.StandardHost` 的 `start` 方法：

```

package org.apache.catalina.core;

public class StandardHost

    public synchronized void start() throws LifecycleException {
        .....
        super.start();
    }

public class ContainerBase

    public synchronized void start() throws LifecycleException {
        .....
        threadStart();
        .....
    }

    protected void threadStart() {

        if (thread != null)
            return;
        if (backgroundProcessorDelay <= 0)
            return;

        threadDone = false;
        String threadName = "ContainerBackgroundProcessor[" + toString() + "]";
        thread = new Thread(new ContainerBackgroundProcessor(), threadName);
        thread.setDaemon(true);
        thread.start();
    }

```

好，我们发现线程启动是在 StandardHost（Host 的实现）中，追踪 ContainerBackgroundProcessor 类即可发现 WebappLoader 中如下的代码：

```
public void backgroundProcess() {
    if (reloadable && modified()) {
        try {
            Thread.currentThread().setContextClassLoader
                (WebappLoader.class.getClassLoader());
            if (container instanceof StandardContext) {
                ((StandardContext) container).reload();
            }
        } finally {
            if (container.getLoader() != null) {
                Thread.currentThread().setContextClassLoader
                    (container.getLoader().getClassLoader());
            }
        }
    } else {
        closeJARs(false);
    }
}
```

请注意红色下划线的代码，其中代码 `WebappLoader.class.getClassLoader` 默认返回的是 `org.apache.catalina.loader.WebappClassLoader`（注意：这里没有新建一个 `ClassLoader`，而是直接使用同一个 `ClassLoader`，网上很多关于热部署的示例都是新建一个 `ClassLoader`，这样操作简单，但是不科学的，重新发明轮子有什么意义？很浪费！）。在此我们发现热部署就是重新设置了一下 `ClassLoader` 然后重新加载（`reload`）`Context`。其实事情就这么简单，我前面说过热部署没有什么神秘的，但我也说过如何让变的东西重新加载，不变的东西不重新加载才是“高明”呢！那我们还是来看看 `WebappClassLoader` 吧，从中学习一些“高明”的东西。

在研究 `WebappClassLoader` 之前我们自己先想想如何在不新建一个 `ClassLoader` 的前提下完成类的重新加载：

- 在重新加载之前清除当前 `ClassLoader` 维护的所有已加载类的信息，这是最简单的方法。
- 在监听变化的时候，把变化的信息进行缓存，在重新加载之前清除变化部分的已加载类信息，这种方法很复杂，需要程序的模块化程度相当高。

Tomcat 的 `WebappClassLoader` 采用的是第一种策略，在需要热部署之前会调用 `WebappClassLoader` 的 `stop` 方法，如下图：

```

public class WebappClassLoader
    extends URLClassLoader
    implements Reloader, Lifecycle

public void stop() throws LifecycleException {
    clearReferences();
    started = false;
    int length = files.length;
    for (int i = 0; i < length; i++) {files[i] = null;}
    length = jarFiles.length;
    for (int i = 0; i < length; i++) {
        try {
            if (jarFiles[i] != null) {jarFiles[i].close();}
        } catch (IOException e) {}
        jarFiles[i] = null;
    }
    notFoundResources.clear();
    resourceEntries.clear();
    resources = null;
    repositories = null;
    repositoryURLs = null;
    files = null;
    jarFiles = null;
    jarRealFiles = null;
    jarPath = null;
    jarNames = null;
    lastModifiedDates = null;
    paths = null;
    hasExternalRepositories = false;
    parent = null;

    permissionList.clear();
    loaderPC.clear();
    if (loaderDir != null) {
        deleteDir(loaderDir);
    }
}

```

在 WebappClassLoader 的 stop 方法中两个地方很重要：

- clearReferences(), 清除引用。
- resourceEntries.clear(), 清除已加载类的缓存。

清除已加载类的信息包括清除对已加载类的相关引用和清除已加载类的缓存两个主要环节。

我们前面说过，重写 ClassLoader 的 loadClass 方法是实现自定义类加载器的直接入口，下面我们看看 WebappClassLoader 的 loadClass 方法：

```

public Class loadClass(String name, boolean resolve)
    throws ClassNotFoundException

```

loadClass 方法很重要的几点，我列出来，其它请大家直接浏览源代码：

- loadClass 方法的操作在一个同步块( synchronized (name.intern()))中。这几乎是 我们重写类加载器必须的。
- 首先通过代码 clazz = findLoadedClass0(name)检索是否已经加载，而

findLoadedClass0 使自己重写的。这和我们前面提到的常规加载机制有所区别。请各位直接浏览 findLoadedClass0 的代码，其中很重要的操作就是看看 resourceEntries 中是否已经有已经加载的类，如果你能和前面清除部分提到的 resourceEntries.clear()对应上的话，就好理解其加载机制。

- 如果没有缓冲，再调用 `clazz = findLoadedClass(name)` 看是否缓存。这步操作才回到常规类加载机制中的已加载类查找，其实按照逻辑，这一步仅仅把非 Web 应用程序中的类交给更上一层的类加载器来管理。
- 第三步，如果前两步都没有找到类，首先调用 `clazz = system.loadClass(name)`，通过系统类加载器直接加载。这个很好玩，为什么？原因很简单！首先我们自己写的 Web 应用类库通常情况下不会被系统类加载器找见，换句话说系统类加载器一定加载不到我们的类，如果加载到，那就有问题了！但先用系统类加载器恰恰解决了我们“胡乱”往我们的 Web 应用中扔其它系统包的问题，例如有些人直接把编译期用到的 `javax.servlet-api.jar` 扔到自己部署的 Web 应用中。通过这点，也提醒大家不要乱扔包。
- 第四步，如果设置代理属性为 `true`，直接用 `WebappClassLoader` 的父类加载器。这个也很好理解，例如我们想把 Tomcat 拿来直接嵌入到我们的应用程序，这时我们希望 Tomcat 先用调用它的应用程序类加载还是有必要的。
- 第五步，调用 `WebappClassLoader` 重写的 `findClass`，顺着 `findClass` 可以找见首先调用方法 `findClassInternal`，这一步才是真正 Tomcat 自定义如何加载和缓冲 Web 应用类的地方。请大家自己阅读 `findClassInternal`，主要关注 `defineClass` 和 `definePackage`。
- 第六步，等于一个迫不得已默认类加载，即交给父类加载器或者系统类加载器。

到此为止，就等于基本介绍完了 Tomcat 的热部署的基本实现。总结一下实现热部署的关键环节：

- 首先所有的需要热部署的模块架构必须应该有明确的生命周期，例如 `start` 和 `stop`。并且在 `start` 期间应该有明确的显式类加载器去加载模块类（这个很重要，不要写的代码没有模块的感觉还要热部署，那是给自己找麻烦）。
- 其次，要动态监听模块的更新，需要一个低层线程来不断地检查。
- 再次，自定义类加载器，自己维护模块类是否已经加载的判断和缓存。
- 最后，在执行重新加载的时候记住要先停止 (`stop`) 当前的模块应用然后清除引用和缓存（至于如何维护当前模块的运行状态那是另外一件事情，但你还是要考虑），然后重新开始模块 (`start`)。

**借题发挥：** 熟练掌握了这块，你可以实战很多有趣的东西，例如你可以写一个插件模式的聊天服务器，并且支持插件的热部署；当然你可以去金蝶碰碰是否可以加入他们的应用服务器团队。

好，先说到这里。

### 2.5.3. Java 调用本地代码例子

- 首先编写 Java 代码，如下：

```

1 package pkg;
2 public class Cls {
3     public native int intMethod(int n);
4     public native boolean booleanMethod(boolean bool);
5     public native String stringMethod(String text);
6     public native int intArrayMethod(int[] intArray);
7     static {
8         System.loadLibrary("pkg_Cls");
9     }
10    public static void main(String[] args) {
11        Cls sample = new Cls();
12        int square = sample.intMethod(5);
13        boolean bool = sample.booleanMethod(true);
14        String text = sample.stringMethod("JAVA");
15        int sum = sample.intArrayMethod(new int[] { 1, 1, 2, 3, 5, 8, 13 });
16        System.out.println("intMethod: " + square);
17        System.out.println("booleanMethod: " + bool);
18        System.out.println("stringMethod: " + text);
19        System.out.println("intArrayMethod: " + sum);
20    }
21 }

```

- 编译完之后，我们用 JDK 自带的 javah(命令: javah pkg.Cls)来生成 C 头文件。将会生成类似如下的头文件（这部分内容我写作的假设是大家了解 C/C++编程和相应的动态链接库技术，所以我不会作更多的解释，我的初衷是让了解 C/C++的朋友快速了解 JNI）：

```

1  #include <jni.h>
2  #ifndef _Included_pkg_Cls
3  #define _Included_pkg_Cls
4  #ifdef __cplusplus
5  extern "C" {
6  #endif
7  /*
8   * Class:      pkg_Cls
9   * Method:     intMethod
10  * Signature:  (I)I
11  */
12  JNIEXPORT jint JNICALL Java_pkg_Cls_intMethod
13  (JNIEnv *, jobject, jint);
14
15  /*
16  * Class:      pkg_Cls
17  * Method:     booleanMethod
18  * Signature:  (Z)Z
19  */
20  JNIEXPORT jboolean JNICALL Java_pkg_Cls_booleanMethod
21  (JNIEnv *, jobject, jboolean);
22
23  /*
24  * Class:      pkg_Cls
25  * Method:     stringMethod
26  * Signature:  (Ljava/lang/String;)Ljava/lang/String;
27  */
28  JNIEXPORT jstring JNICALL Java_pkg_Cls_stringMethod
29  (JNIEnv *, jobject, jstring);
30
31  /*
32  * Class:      pkg_Cls
33  * Method:     intArrayMethod
34  * Signature:  ([I)I
35  */
36  JNIEXPORT jint JNICALL Java_pkg_Cls_intArrayMethod
37  (JNIEnv *, jobject, jintArray);
38
39  #ifdef __cplusplus
40  }
41  #endif
42  #endif

```

- 编写本地代码（我这里用 C）。代码如下：

```

1  #include "pkg_Cls.h"
2  #include <string.h>
3
4  JNIEXPORT jint JNICALL Java_pkg_Cls_intMethod
5  (JNIEnv *env, jobject obj, jint num) {
6      return num * num;
7  }
8
9  JNIEXPORT jboolean JNICALL Java_pkg_Cls_booleanMethod
10 (JNIEnv *env, jobject obj, jboolean boolean) {
11     return !boolean;
12 }
13
14 JNIEXPORT jstring JNICALL Java_pkg_Cls_stringMethod
15 (JNIEnv *env, jobject obj, jstring string) {
16     const char *str = (*env)->GetStringUTFChars(env, string, 0);
17     char cap[128];
18     strcpy(cap, str);
19     (*env)->ReleaseStringUTFChars(env, string, str);
20     return (*env)->NewStringUTF(env, strupr(cap));
21 }
22
23 JNIEXPORT jint JNICALL Java_pkg_Cls_intArrayMethod
24 (JNIEnv *env, jobject obj, jintArray array) {
25     int i, sum=0;
26     jsize len = (*env)->GetArrayLength(env, array);
27     jint *body = (*env)->GetIntArrayElements(env, array, 0);
28     for (i=0; i<len; i++){
29         sum += body[i];
30     }
31     (*env)->ReleaseIntArrayElements(env, array, body, 0);
32     return sum;
33 }

```

- 将上面代码在 Windows 下编译成 pkg\_Cls.dll，并且将 pkg\_Cls.dll 放到运行 Java 程序的根目录。这块最容易出问题的是导出函数名称是否符合约定，例如在编译 `__stdcall` 和 `__cdecl` 声明的函数时将会有不同的命名约定，从而导致不符合 JNI 的命名约定，所以可能会出现对应不上问题，解决此问题的方法可以将 `jni_md.h` 中加入如声明 `#define JNICALL __cdecl`。
- 运行 Java 类 `pkg.Cls`。

OK，这算是一个完整的演示 Java 如何调用本地库的流程。

#### 2.5.4. 本地库调用 JVM 和 Java 代码例子

- 先来编写一个 Java 类：

```

1 package pkg;
2 public class Cls2 {
3     public static int intMethod(int param){
4         return param*param;
5     }
6     public boolean booleanMethod(boolean param){
7         return !param;
8     }
9 }

```

- 调用的 C 程序。

```
1 #include "jni.h"
2 #include <stdio.h>
3 int main(){
4     JavaVMOption options[1]; /*启动JVM的选项*/
5     JNIEnv *env; /*从JVM实例中获取环境，用JNIEnv的方法和Java调用本地代码中本地代码调用的一致*/
6     JavaVM *jvm;
7     JavaVMInitArgs vm_args;
8     long status;
9     jclass cls;
10    jobject obj;
11    jmethodID mid;
12    jint square;
13    jboolean n;
14    options[0].optionString = "-Djava.class.path="; /*假设把编译后的class放到当前运行目录*/
15    memset(&vm_args, 0, sizeof(vm_args));
16    vm_args.version = JNI_VERSION_1_6;
17    vm_args.nOptions = 1;
18    vm_args.options = options;
19    status = JNI_CreateJavaVM(&jvm, (void*)&env, &vm_args);
20    if (status != JNI_ERR){
21        cls = (*env)->FindClass(env, "pkg/Cls2"); /*在类路径中查找该类，并且创建类实例*/
22        if(cls !=NULL){
23            mid = (*env)->GetStaticMethodID(env, cls, "intMethod", "(I)I"); /*(I)I方法的参数和返回值标识*/
24            if(mid !=NULL){
25                square = (*env)->CallStaticIntMethod(env, cls, mid, 5);
26                printf("Result of intMethod: %d\n", square);
27            }
28            mid = (*env)->GetMethodID(env, cls, "<init>", "()V"); /*获取空的构造函数*/
29            if(mid !=NULL){
30                obj = (*env)->NewObject(env, cls, mid); /*创建pkg.Cls2的对象*/
31                if(obj !=NULL){
32                    mid = (*env)->GetMethodID(env, cls, "booleanMethod", "(Z)Z");
33                    if(mid !=NULL){
34                        n = (*env)->CallBooleanMethod(env, obj, mid, JNI_TRUE); /*调用对象的方法*/
35                        printf("Result of booleanMethod: %d\n", n);
36                    }
37                    (*env)->DeleteLocalRef(env, obj);
38                }
39            }
40            (*env)->DeleteLocalRef(env, cls);
41        }
42        (*jvm)->DestroyJavaVM(jvm);
43        return 0;
44    }
45    else return -1;
46 }
```

- 编译运行。编译的时候可以用 JDK 提供的 lib/jvm.lib 进行静态连接。

OK，就到这里！用到的各个函数，请参考 <http://java.sun.com/docs/books/jni/html/jniTOC.html>。

## 2.6. 本章小结

- JVM 和其它任何虚拟机一样，都是完成资源的封装，从而提高资源的利用率和可管理性。
- JVM 是 Java 平台体系的核心基础设施。Java 语言编写的程序在 JVM 上被解释和执行。
- JVM 在 JSR-924 中进行了规范，在 JSR-202 中维护更新。JVM 规范使 JVM 的具体实现和建立在 JVM 之上的应用之间耦合性降低。
- JVM 规范主要从 Java 字节码规范、基本的执行过程、JVM 指令集和抽象架构四个层面进行描述。
- JVM 实现中的关键问题是内存管理和垃圾回收、JVM 指令到机器指令翻译方法、多线程支持。

## 2.7. 参考资料

- [http://java.sun.com/docs/books/jvms/second\\_edition/html/VMSpecTOC.doc.html](http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html) JVM 规范第二版，教程性质的文档，。

- <http://jikesrvm.org/Publications> Jikesrvm 是比较有影响力的开源 JVM，在其官方网站上提供了大量关于 JVM 的论文、讲稿（英文资源）。
- [http://www.infoq.com/cn/news/2010/04/cliff\\_click\\_gc\\_pauses](http://www.infoq.com/cn/news/2010/04/cliff_click_gc_pauses) 在堆增大的同时确保垃圾回收停顿时间短暂——专访 Cliff Click 博士。
- <http://www.ibm.com/developerworks/cn/java/j-lo-harmony1/>  
<http://www.ibm.com/developerworks/cn/java/j-lo-harmony2/> Apache Harmony 项目简介（中文文章）。
- <http://java.sun.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html> Java Native Interface Specification.
- <http://java.sun.com/docs/books/jni/html/jniTOC.html> The Java Native Interface Programmer's Guide and Specification.
- <http://www.ibm.com/developerworks/cn/java/j-jni/> 使用 Java Native Interface 的最佳实践。