深入理解JVM

> 原文链接：http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals

每个使用Java的开发者都知道Java字节码是在JRE中运行(JRE: Java 运行时环境)。JVM则是JRE中的核心组成部分，承担分析和执行Java字节码的工作，而Java程序员通常并不需要深入了解JVM运行情况就可以开发出大型应用和类库。尽管如此，如果你对JVM有足够了解，就会对Java有更好的掌握，并且能解决一些看起来简单但又尚未解决的问题。

所以，在本篇文章中，我将会介绍JVM工作原理，内部结构，Java字节码的执行及指令的执行顺序，并会介绍一些常见的JVM错误及其解决方案。最后会简单介绍下Java SE7带来的新特性。

虚拟机

JRE由Java API和JVM组成，JVM通过类加载器(Class Loader)加类Java应用，并通过Java API进行执行。

虚拟机(VM: Virtual Machine)是通过软件模拟物理机器执行程序的执行器。最初Java语言被设计为基于虚拟机器在而非物理机器，重而实现WORA(一次编写，到处运行)的目的，尽管这个目标几乎被世人所遗忘。所以，JVM可以在所有的硬件环境上执行Java字节码而无须调整Java的执行模式。

JVM的基本特性：

- **基于栈(Stack-based)的虚拟机**: 不同于Intel x86和ARM等比较流行的计算机处理器都是基于*寄存器(register)*架构，JVM是*基于栈执行的*。
- **符号引用(Symbolic reference)**: 除基本类型外的所有Java类型(类和接口)都是通过符号引用取得关联的，而非显式的基于内存地址的引用。
- **垃圾回收机制**: 类的实例通过用户代码进行显式创建，但却通过垃圾回收机制自动销毁。
- **通过明确清晰基本类型确保平台无关性**: 像C/C++等传统编程语言对于int类型数据在同平台上会有不同的字节长度。JVM却通过明确的定义基本类型的字节长度来维持代码的平台兼容性，从而做到平台无关。
- **网络字节序(Network byte order)**: Java class文件的二进制表示使用的是基于网络的字节序(network byte order)。为了在使用小端(little endian)的Intel x86平台和在使用了大端(big endian)的RISC系列平台之间保持平台无关，必须要定义一个固定的字节序。JVM选择了网络传输协议中使用的网络字节序，即基于大端(big endian)的字节序。

Sun 公司开发了Java语言，但任何人都可以在遵循JVM规范的前提下开发和提供JVM实现。所以目前业界有多种不同的JVM实现，包括Oracle Hostpot JVM和IBM JVM。Google公司使用的Dalvik VM也是一种JVM实现，尽管其并未完全遵循JVM规范。与基于栈机制的Java 虚拟机不同的是Dalvik VM是基于寄存器的，Java 字节码也被转换为Dalvik VM使用的寄存器指令集。

Java 字节码

JVM使用Java字节码—一种运行于Java(用户语言)和机器语言的中间语言，以达到WORA的目的。Java字节码是部署Java程序的最小单元。

在介绍Java 字节码之前，我们先来看一下什么是字节码。下面涉及的案例是曾在一个真实的开发场景中遇到过的情境。

现象

一个曾运行完好的程序在更新了类库后却不能再次运行，并抛出了如下异常：

```
Exception in thread "main" java.lang.NoSuchMethodError:
com.nhn.user.UserAdmin.addUser(Ljava/lang/String;)V
    at com.nhn.service.UserService.add(UserService.java:14)
    at com.nhn.service.UserService.main(UserService.java:19)
```

程序代码如下，并在更新类库之前未曾对这段代码做过变更：

```
// UserService.java
…
public void add(String userName) {
    admin.addUser(userName);
}
```

类库中更新过的代码前后对比如下：

```
// UserAdmin.java - Updated library source code
…
public User addUser(String userName) {
    User user = new User(userName);
    User prevUser = userMap.put(userName, user);
    return prevUser;
}
// UserAdmin.java - Original library source code
…
public void addUser(String userName) {
    User user = new User(userName);
    userMap.put(userName, user);
}
```

简单来说就是addUser()方法在更新之前返回void而在更新之后返回了User类型实例。而程序代码因为不关心addUser的返回值，所以在使用的过程中并未做过改变。

> 初看起来，com.mhn.user.UserAdmin.addUser()依然存在，但为什么会出现**NoSuchMethodError**？

问题分析

主要原因是程序代码在更新类库时并未重新编译代码，也就是说，虽然程序代码看起来依然是在调用addUser方法而不关心其返回值，而对编译的类文件来说，他是要明确知道调用方法的返回值类型的。

可以通过下面的异常信息说明这一点：

```
java.lang.NoSuchMethodError: com.nhn.user.UserAdmin.addUser(Ljava/langString;)V
```

NoSuchMethodError 是因为 *"com.nhn.user.UserAdmin.addUser(Ljava/lang/String;)V"* 方法找不到引起的。看一下"Ljava/lang/String;"和后面的"V"。在Java字节码表示中，"*L;*"表示类的实例。所以上面的addUser方法需要一个java/lang/String对象作为参数。就这个案例中，类库中的addUser()方法的参数未发生变化，所以参数是正常的。再看一下异常信息中最后面的"V"，它表示方法的返回值类型。在Java字节码表示中，"V"意味着该方法没有返回值。所以上面的异常信息就是说需要一个java.lang.String参数且没有任何返回值的com.nhn.user.UserAdmin.addUser方法找不到。

因为程序代码是使用之前版本的类库进编译的，class文件中定义的是应该调用返回"V"类型的方法。然而，在改变类库后，返回"V"类型的方法已不存在，取而代之的是返回类型为"Lcom/nhn/user/User;"的方法。所以便发生了上面看到的NoSuchMethodError。

> 注释
>
> 因为开发者未针对新类库重新编译程序代码，所以发生了错误。尽管如此，类库提供者却也要为此负责。因为之前没有返回值的addUser()方法既然是public方法，但后面却改成了会返回user实现，这意味着方法签名发生了明显的变化。这意味了该类库不能对之前的版本进行兼容，所以类库提供者必须事前对此进行通知。

我们重新回到Java 字节码，**Java 字节码**是JVM的基本元素，JVM本身就是一个用于执行Java字节码的执行器。Java编译器并不会把像C/C++那样把高级语言转为机器语言(CPU执行指令)，而是把开发者能理解的Java语言转为JVM理解的Java字节码。因为Java字节码是平台无关的，所以它可以在安装了JVM(准确的说，是JRE环境)的任何硬件环境执行，即使它们的CPU和操作系统各不相同(所以在Windows PC机上开发和编译的class文件在不做任何调整的情况下就可以在Linux机器上执行)。编译后文件的大小与源文件大小基本一致，所以比较容易通过网络传输和执行Java字节码。

Java class文件本身是基于二进制的文件，所以我们很难直观的理解其中的指令。为了管理这些class 文件， JVM提供了javap命令来对二进制文件进行反编译。执行javap得到的是直观的java指令序列。在上面的案例中，通过对程序代码执行javap -c就可得到应用中的UserService.add()方法的指令序列，如下：

```
public void add(java.lang.String);
  Code:
   0:   aload_0
   1:   getfield        #15; //Field admin:Lcom/nhn/user/UserAdmin;
   4:   aload_1
   5:   invokevirtual   #23; //Method com/nhn/user/UserAdmin.addUser:(Ljava/lang/String;)V
   8:   return
```

在上面的Java指令中，addUser()方法是在第五行被调用，即"5: invokevirtual #23"。这句的意思是索引位置为23的方法会被调用，方法的索引位置是由javap程序标注的。**invokevirtual**是Java 字节码中最常用到的一个操作码，用于调用一个方法。另外，在Java字节码中有4个表示调用方法的操作码： *invokeinterface, _invokespecial, invokestatic, _invokevirtual* 。他们每个的含义如下：

- **invokeinterface**: 调用接口方法
- **invokespecial**: 调用初始化方法、私有方法、或父类中定义的方法
- **invokestatic**: 调用静态方法
- **invokevirtual**: 调用实例方法

Java 字节码的指令集包含操作码(OpCode)和操作数(Operand)。像invokevirtual这样的操作码需要一个2字节长度的操作数。

对上面案例中的程序代码，如果在更新类库后重新编译程序代码，然后我们再反编译字节码将看到如下结果：

```
public void add(java.lang.String);
  Code:
    0:   aload_0
    1:   getfield        #15; //Field admin:Lcom/nhn/user/UserAdmin;
    4:   aload_1
    5:   invokevirtual   #23; //Method com/nhn/user/UserAdmin.addUser:
(Ljava/lang/String;)Lcom/nhn/user/User;
    8:   pop
    9:   return
```

如上我们看到#23对应的方法变成了具有返回值类型"Lcom/nhn/user/User;"的方法。

> 在上面的反编译结果中，代码前面的数字是具有什么含义？

它是一个一字节数字，也许正因此JVM执行的代码被称为“字节码”。像 *aload_0, getfield* 和 *invokevirtual* 都被表示为一个单字节数字。(aload_0 = 0x2a, getfiled = 0xb4, invokevirtual = 0xb6)。因此Java字节码表示的最大指令码为256。

像aload*0*和*aload*1这样的操作码不需要任何操作数，因此aload_0的下一个字节就是下一个指令的操作码。而像getfield和invokevirtual这样的操作码却需要一个2字节的操作数，因此第一个字节里的第二个指令getfield指令的一下指令是在第4个字节，其中跳过了2个字节。通过16进制编辑器查看字节码如下：

```
2a b4 00 0f 2b b6 00 17 57 b1
```

在Java字节码中，类实例表示为"L;"，而void表示为"V"，类似的其他类型也有各自的表示。下表列出了Java字节码中类型表示。

表1: Java字节码里的类型表示

Java 字节码	类型	描述
B	byte	单字节
C	char	Unicode字符
D	double	双精度浮点数
F	float	单精度浮点数
I	int	整型
J	long	长整型
L	引用	classname类型的实例
S	short	短整型
Z	boolean	布尔类型
[	引用	一维数组

表2: Java代码的字节码示例

java 代码	**Java** 字节码表示
double d[][][]	[[[D
Object mymethod(int i, double d, Thread t)	mymethod(I,D,Ljava/lang/Thread;)Ljava/lang/Object;

在《Java虚拟机技术规范第二版》的4.3 描述符(Descriptors)章节中有关于此的详细描述，在第6章"Java虚拟机指令集"中介绍了更多不同的指令。

类文件格式

在解释类文件格式之前，先看一个在Java Web应用中经常发生的问题。

现象

在Tomcat环境里编写和运行JSP时，JSP文件未被执行，并伴随着如下错误：

```
Servlet.service() for servlet jsp threw exception org.apache.jasper.JasperException: Unable to
compile class for JSP Generated servlet error:
The code of method _jspService(HttpServletRequest, HttpServletResponse) is exceeding the 65535
bytes limit"
```

问题分析

对于不同的Web应用容器，上面的错误信息会有些微差异，但核心信息是一致的，即65535字节的限制。这个限制是JVM定义的，用于规定**方法的定义不能大于65535个字节**。

下面我将先介绍65535个的字节限制，然后详细说明为什么要有这个限制。

Java字节码中，"goto"和"jsr"指令分别表示分支和跳转。

```
goto [branchbyte1] [branchbyte2]
jsr [branchbyte1] [branchbyte2]
```

这两个操作指令都跟着一个2字节的操作数，而2个字节能表示的最大偏移量只能是65535。然而为了支持更大范围的分支，Java字节码又分别定义了"goto_w" 和 "jsr_w" 用于接收4个字节的分支偏移量。

```
goto_w [branchbyte1] [branchbyte2] [branchbyte3] [branchbyte4]
jsr_w [branchbyte1] [branchbyte2] [branchbyte3] [branchbyte4]
```

受这两个指令所赐，分支能表示的最大偏移远远超过了65535，这么说来java 方法就不会再有65535个字节的限制了。然而，由于Java 类文件的各种其他限制，java方法的定义仍然不能够超过65535个字节的限制。下面我们通过对类文件的解释来看看java方法不能超过65535字节的其他原因。

Java类文件的大体结构如下：

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    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];}
```

上面的文件结构出自《Java虚拟机技术规范第二版》的4.1节"类文件结构"。

之前讲过的UserService.class文件的前16个字节的16进制表示如下：

```
ca fe ba be 00 00 00 32 00 28 07 00 02 01 00 1b
```

我们通过对这一段符号的分析来了解一个类文件的具体格式。

- **magic**: 类文件的前4个字节是一组魔数，是一个用于区分Java类文件的预定义值。如上所看到的，其值固定为0xCAFEBABE。也就是说一个文件的前4个字节如果是0xCAFABABE，就可以认为它是Java类文件。"CAFABABE"是与"JAVA"有关的一个有趣的魔数。
- **minor_version, major_version**: 接下来的4个字节表示类的版本号。如上所示，0x00000032表示的类版本号为50.0。由JDK 1.6编译而来的类文件的版本号是50.0，而由JDK 1.5编译而来的版本号则是49.0。JVM必须保持向后兼容，即保持对比其版本低的版本的类文件的兼容。而如果在一个低版本的JVM中运行高版本的类文件，则会出现java.lang.UnsupportedClassVersionError的发生。
- **constant_pool_count, constant_pool[]**: 紧接着版本号的是类的常量池信息。这里的信息在运行时会被分配到运行时常量池区域，后面会有对内存分配的介绍。在JVM加载类文件时，类的常量池里的信息会被分配到运行时常量池，而运行时常量池又包含在方法区内。上面UserService.class文件的constant_pool_count为0x0028，所以按照定义contant_pool数组将有(40-1)即39个元素值。
- **access_flags**: 2字节的类的修饰符信息，表示类是否为public, private, abstract或者interface。
- **this_class, super_class**: 分别表示保存在constant_pool数组中的当前类及父类信息的索引值。
- **interface_count, interfaces[]**: interface_count为保存在constant_pool数组中的当前类实现的接口数的索引值，interfaces[]即表示当前类所实现的每个接口信息。
- **fields_count, fields[]**: 类的字段数量及字段信息数组。字段信息包含字段名、类型、修饰符以及在constant_pool数组中的索引值。
- **methods_count, methods[]**: 类的方法数量及方法信息数组。方法信息包括方法名、参数的类型及个数、返回值、修饰符、在constant_pool中的索引值、方法的可执行代码以及异常信息。
- **attributes_count, attributes[]**: attribute_info有多种不同的属性，分别被field_info, method_into使用。

javap程序把class文件格式以可阅读的方式输出来。在对UserService.class文件使用"javap -verbose"命令分析时，输出内容如下：

```
Compiled from "UserService.java"

public class com.nhn.service.UserService extends java.lang.Object
  SourceFile: "UserService.java"
  minor version: 0
  major version: 50
  Constant pool:const #1 = class        #2;     //  com/nhn/service/UserService
const #2 = Asciz        com/nhn/service/UserService;
const #3 = class        #4;     //  java/lang/Object
const #4 = Asciz        java/lang/Object;
const #5 = Asciz        admin;
const #6 = Asciz        Lcom/nhn/user/UserAdmin;;// … omitted - constant pool continued …

{
// … omitted - method information …

public void add(java.lang.String);
  Code:
   Stack=2, Locals=2, Args_size=2
   0:   aload_0
   1:   getfield        #15; //Field admin:Lcom/nhn/user/UserAdmin;
   4:   aload_1
   5:   invokevirtual   #23; //Method com/nhn/user/UserAdmin.addUser:
(Ljava/lang/String;)Lcom/nhn/user/User;
   8:   pop
   9:   return  LineNumberTable:
   line 14: 0
   line 15: 9  LocalVariableTable:
   Start  Length  Slot  Name   Signature
   0      10      0    this       Lcom/nhn/service/UserService;
   0      10      1    userName       Ljava/lang/String; // … Omitted - Other method information
…
}
```

由于篇幅原因，上面只抽取了部分输出结果。在全部的输出信息中，会为你展示包括常量池和每个方法内容等各种信息。

方法的65535个字节的限制受到了结构体**method_info**的影响。如上面"javap -verbose"的输出所示，结构体method_info包括代码(Code)、行号表(LineNumberTable)以及本地变量表(LocalVariableTable)。其中行号表、本地变量表以及代码里的异常表(exception_table)的总长度为一个固定2字节的值。因此方法的大小不能超过行号表、本地变量表、异常表的长度，即不能超过65535个字节。

尽管很多人抱怨方法的大小限制，JVM规范也声称将会对此大小进行扩充，然而到目前为止并没有明确的进展。因为JVM技术规范里定义要把几乎整个类文件的内容都加载到方法区，因此如果方法长度将会对程序的向后兼容带来极大的挑战。

> 对于一个由**Java**编译器错误而导致的错误的类文件将发生怎样的情况？如果是在网络传输或文件复制过程中，类文件被损坏又将发生什么？

为了应对这些场景，Java类加载器的加载过程被设计为一个非常严谨的处理过程。JVM规范详细描述了这个过程。

> 注释

> 我们如何验证JVM成功执行了类文件的验证过程？如何验证不同的JVM实现是否符合JVM规范？为此，Oracle提供了专门的测试工具：TCK(Technology Compatibility Kit)。TCK通过执行大量的测试用例(包括大量通过不同方式生成的错误类文件)来验证JVM规范。只有通过TCK测试的JVM才能被称作是JVM。
>
> 类似TCK，还有一个JCP(Java Community Process; http://jcp.org)，用于验证新的Java技术规范。对于一个JCP，必须具有详细的文档，相关的实现以及提交给JSR(Java Specification Request)的TCK测试。如果用户想像JSR一样使用新的Java技术，那他必须先从RI提供者那里得到许可，或者自己直接实现它并对之进行TCK测试。

JVM 结构

Java程序的执行过程如下图所示：


图1: Java代码执行过程

类加载器把Java字节码载入到运行时数据区，执行引擎负责Java字节码的执行。

类加载

Java提供了动态加载的特性，只有在运行时第一次遇到类时才会去加载和链接，而非在编译时加载它。JVM的类加载器负责类的动态加载过程。Java类加载器的特点如下：

- **层次结构**：Java的类加载器按是父子关系的层次结构组织的。Boostrap类加载器处于层次结构的顶层，是所有类加载器的父类。
- **委派模式**：基于类加载器的层次组织结构，类加载器之间是可以进行委派的。当一个类需要被加载，会先去请求父加载器判断该类是否已经被加载。如果父类加器已加载了该类，那它就可以直接使用而无需再次加载。如果尚未加载，才需要当前类加载器来加载此类。
- **可见性限制**：子类加载器可以从父类加载器中获取类，反之则不行。
- **不能卸载**： 类加载器可以载入类却不能卸载它。但是可以通过删除类加载器的方式卸载类。

每个类加载器都有自己的空间，用于存储其加载的类信息。当类加载器需要加载一个类时，它通过FQCN)(Fully Quanlified Class Name: 全限定类名)的方式先在自己的存储空间中检测此类是否已存在。在JVM中，即便具有相同FQCN的类，如果出现在了两个不同的类加载器空间中，它们也会被认为是不同的。存在于不同的空间意味着类是由不同的加载器加载的。

下图解释了类加载器的委派模型：


图2: 类加载器的委派模型

当JVM请示类加载器加载一个类时，加载器总是按照从类加载器缓存、父类加载器以及自己加载器的顺序查找和加载类。也就是说加载器会先从缓存中判断此类是否已存在，如果不存在就请示父类加载器判断是否存在，如果直到Bootstrap类加载器都不存在该类，那么当前类加载器就会从文件系统中找到类文件进行加载。

- **Bootstrap加载器**：Bootstrap加载器在运行JVM时创建，用于加载Java APIs，包括Object类。不像其他的类加载器由Java代码实现，Bootstrap加载器是由native代码实现的。
- **扩展加载器(Extension class loader)**：扩展加载器用于加载除基本Java APIs以外扩展类。也用于加载各种安全扩展功能。
- **系统加载器(System class loader)**：如果说Bootstrap和Extension加载器用于加载JVM运行时组件，那么系统加载器加载的则是应用程序相关的类。它会加载用户指定的CLASSPATH里的类。
- **用户自定义加载器**：这个是由用户的程序代码创建的类加载器。

像Web应用服务器(WAS: Web Application Server)等框架通过使用用户自定义加载器使Web应用和企业级应用可以隔离开在各自的类加载空间独自运行。也就是说可以通过类加载器的委派模式来保证应用的独立性。不同的WAS在自定义类加载器时会有略微不同，但都不外乎使用加载器的层次结构原理。

如果一个类加载器发现了一个未加载的类，则该类的加载和链接过程如下图：

 图**3:** 类加载步骤

每一步的具体描述如下：

- **加载(Loading)**: 从文件中获取类并载入到JVM内存空间。
- **验证(Verifying)**: 验证载入的类是否符合Java语言规范和JVM规范。在类加载流程的测试过程中，这一步是最为复杂且耗时最长的部分。大部分JVM TCK的测试用例都用于检测对于给定的错误的类文件是否能得到相应的验证错误信息。
- **准备(Preparing)**: 根据内存需求准备相应的数据结构，并分别描述出类中定义的字段、方法以及实现的接口信息。
- **解析(Resolving)**: 把类常量池中所有的符号引用转为直接引用。
- **初始化(Initializing)**: 为类的变量初始化合适的值。执行静态初始化域，并为静态字段初始化相应的值。

JVM规范定义了规则，但也允许在运行时灵活处理。

运行时数据区

 图**4:** 运行时数据区结构

运行时数据区是JVM程序运行时在操作系统上分配的内存区域。运行时数据区又可细分为6个部分，即：为每个线程分别创建的**PC寄存器**、**JVM栈**、**本地方法栈**和被所有线程共用的**数据堆**、**方法区**和**运行时常量池**。

- **PC 寄存器**：每个线程都会有一个PC(Program Counter)寄存器，并跟随线程的启动而创建。PC寄存器中存有将执行的JVM指令的地址。
- **JVM 栈**：每个线程都有一个JVM栈，并跟随线程的启动而创建。其中存储的数据元素称为栈帧(Stack Frame)，JVM会把栈桢压入栈或从其中弹出。如果有任何异常，printStackTrace()方法输出的栈跟踪信息的每一行就都是一个栈帧信息。
  

图**5: JVM**栈结构

```
- 栈帧：在JVM中一旦有方法执行，JVM就会为之创建一个栈帧，并把其添加到当前线程的JVM栈中。当方法运行结束时，栈帧也会相应的从JVM栈中移除。栈帧中存放着对本地变量数组、操作数栈以及属于当前运行方法的运行时常量池的引用。本地变量数组和操作数栈的大小在编译时就已确定，所以属在运行时属于方法的栈帧大小是固定的。
- 本地变量数组：本地变量数组的索引从0开始计数，其位置存储着对方法所属类实例的引用。从索引位置1开始的保存的是传递给该方法的参数。其后存储的就是真正的方法的本地变量了。
- 操作数栈：是方法的实际运行空间。每个方法变换操作数栈和本地变量数组，并把调用其它方法的结果从栈中弹或压入。在编译时，编译器就能计算出操作数栈所需的内存窖，因此操作数栈的大小在编译时也是确定的。
```

- **本地方法栈**：为非Java编写的本地代程定义的栈空间。也就是说它基本上是用于通过JNI(Java Native Interface)方式调用和执行的C/C++代码。根据具体情况，C栈或C++栈将会被创建。
- **方法区**：方法区是被所有线程共用的内存空间，在JVM启动时创建。它存储了运行时常量池、字段和方法信息、静态变量以及被JVM载入的所有类和接口的方法的字节码。不同的JVM提供者在实现方法区时会通常有不同的形式。在Oracle的Hotspot JVM里方法区被称为Permanent Area(永久区)或Permanent

Generation(PermGen， 永久代)。JVM规范并对方法区的垃圾回收未做强制限定，因此对于JVM实现者来说，方法区的垃圾回收是可选操作。

- **运行时常量池**：一个存储了类文件格式中的常量池表的内存空间。这部分空间虽然存在于方法区内，但却在JVM操作中扮演着举足轻重的角色，因此JVM规范单独把这一部分拿出来描述。除了每个类或接口中定义的常量，它还包含了所有对方法和字段的引用。因此当需要一个方法或字段时，JVM通过运行时常量池中的信息从内存空间中来查找其相应的实际地址。
- **数据堆**：堆中存储着所有的类实例或对象，并且也是垃圾回收的目标场所。当涉及到JVM性能优化时，通常也会提及到数据堆空间的大小设置。JVM提供者可以决定划分堆空间或者不执行垃圾回收。

我们再回到先前讨论的反编译过的字节码中：

```
public void add(java.lang.String);
  Code:
    0:   aload_0
    1:   getfield        #15; //Field admin:Lcom/nhn/user/UserAdmin;
    4:   aload_1
    5:   invokevirtual   #23; //Method com/nhn/user/UserAdmin.addUser:
(Ljava/lang/String;)Lcom/nhn/user/User;
    8:   pop
    9:   return
```

比较一下上面反编译过的字节码和我们常见的基于x86架构的机器码的区别，虽然它们着相似的格式、操作码，但有一个明显的区别：Java字节码中没有寄存器名称、内存地址或者操作数的偏移位置。正如前所述，JVM使用的是栈模型，因此它并不需要x86架构中使用的寄存器。因为JVM自己管理内存，所以Java字节码中使用像15、23这样的索引值而非直接的内存地址。上面的15和23指向的是当前类的常量池中的位置(即UserService类)。也就是JVM为每个类创建一个常量池，并在常量池中存储真实对象的引用。

上面每行代码的解释如下：

- **aload_0**: 把本地变量数组的0号元素添加到操作数栈。本地变量数组的0号元素始终是 *this* ，即当前类实例对象的引用。
- **getfield #15**: 在当前类的常量池中，把15号元素添加到操作数栈。上面的15号元素是UserAdmin admin字段。因为admin是一个类实例对象，因此其引用被加入到操作数栈。
- **aload_1**: 把本地变量数组的1号元素添加到操作数栈中。本地变量数组中从第1个位置开始的元素存储着方法的参数。因此调用add()方法传入的String userName参数的引用将会添加到操作数栈。
- **invokevirtual #23**: 调用当前类常量池中的第23号元素所引用的方法，同时被aload_1和getField #15操作添加到操作数栈中的引用信息将被传给方法调用。当方法调用完成后，其结果将被添加到操作数栈。
- **pop**: 把通过invokevirtual方法调用得到的结果从操作数栈中弹出。在前面讲述中使用之前类库时没有返回值，也就不需要把结果从操作数栈中弹出了。
- **return**: 方法完成。

下图将帮忙容易理解上面的文字解释：


图6: 从运行时数据区加载Java字节码示例

作为示例，上面的方法中本地变量数组中的值未曾有任何改变，所以上图中我们只看到操作数栈的变化。实际上，在大多数场景中本地变量数组也是被发生变化的。数据通过加载指令(aload, iload)和存储指令(astore, istore)在本地变量数组和操作数栈之间发生变化和移动。

在本章节我们对运行时常量池和JVM栈作了清晰的介绍。在JVM运行时，每个类的实例被分配到数据堆上，类信息(包括User, UserAdmin, UserService, String)等被存储在方法区。

执行引擎

JVM通过类加载器把字节码载入运行时数据区是由执行引擎执行的。执行引擎以指令为单位读入Java字节码，就像CPU一个接一个的执行机器命令一样。每个字节码命令包含一字节的操作码和可选的操作数。执行引擎读取一个指令并执行相应的操作数，然后去读取并执行下一条指令。

尽管如此，Java字节码还是以一种可以理解的语言编写的，而不像那些机器直接执行的无法读懂的语言。所以JVM的执行引擎必须要把字节码转换为能被机器执行的语言指令。执行引擎有两种常用的方法来完成这一工作：

- **解释器(Interpreter)**：读取、解释并逐一执行每一条字节码指令。因为解释器逐一解释和执行指令，因此它能够快速的解释每一个字节码，但对解释结果的执行速度较慢。所有的解释性语言都有类似的缺点。叫做字节码的语言人本质上就像一个解释器一样运行。
- **即时编译器(JIT: Just-In-Time)**：即时编译器的引入用来弥补解释器的不足。执行引擎先以解释器的方式运行，然后在合适的时机，即时编译器把整修字节码编译成本地代码。然后执行引擎就不再解释方法的执行而是通过使用本地代码直接执行。执行本地代码较逐一解释执行每条指令在速度上有较大的提升，并且通过对本地代码的缓存，编译后的代码能具有更快的执行速度。

然而，即时编译器在编译代码时比逐一解释和执行每条指令更耗时，所以如果代码只会被执行一次，解释执行可能会具有更好的性能。所以JVM通过检查方法的执行频率，然后只对达到一定频率的方法才会做即时编译。

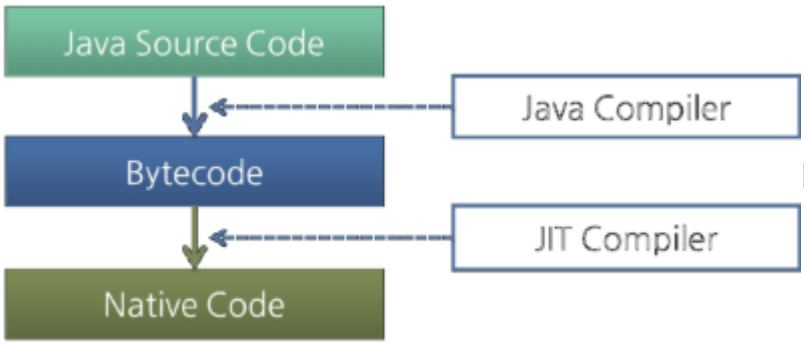

图7: Java编译器和即时编译器

JVM规范中并未强行约束执行引擎如何运行。所以不同的JVM在实现各种的执行引擎时通过各种技术手段并引入多种即时编译器来提升性能。

大部分的即时编译器运行流程如下图：

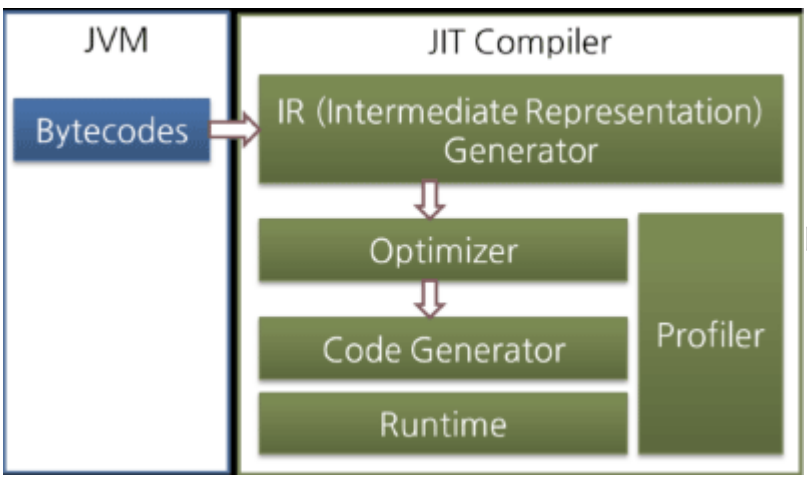

图8: 即时编译器

即时编译器先把字节码转为一种中间形式的表达式(IR: Itermediate Representation)，并对之进行优化，然后再把这种表达式转为本地代码。

Oracel Hotspot VM使用的即时编译器称为Hotspot编译器。之所以称为Hotspot是因为Hotspot Compiler会根据分析找到具有更高编译优先级的热点代码，然后所这些热点代码转为本地代码。如果一个被编译过的方法不再被频繁调用，也即不再是热点代码，Hotspot VM会把这些本地代码从缓存中删除并对其再次使用解释器模式执行。Hotspot VM有Server VM和Client VM之后，它们所使用的即时编译器也有所不同。

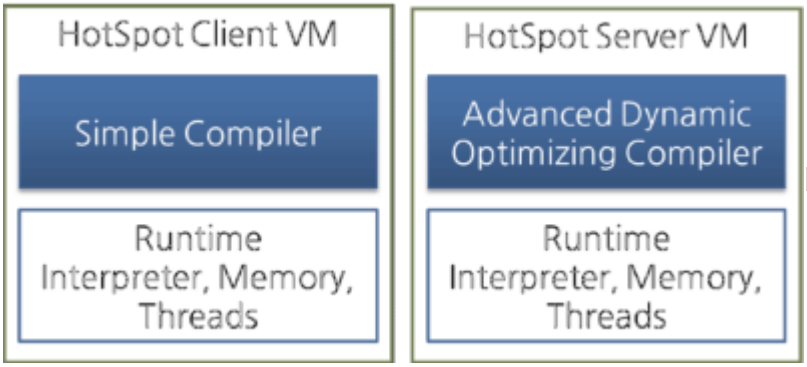

图9: Hotspot ClientVM 和Server VM

Client VM和Server VM使用相同的运行时环境，如上图所示，它们的区别在于使用了不同的即时编译器。Server VM通过使用多种更为复杂的性能优化技术从而具有更好的表现。

IBM VM在他的IBM JDK6中引入了AOT(Ahead-Of-Time) 编译器技术。通过此种技术使得多个JVM之间能通过共享缓存分享已编译的本地代码。也就是说通过AOT编译器编译的代码能被其他JVM直接使用而无须再次编译。另外IBM JVM通过使用AOT编译器把代码预编译为JXE(Java Executable)文件格式从而提供了一种快速执行代码的方式。

大多数的Java性能提升都是通过优化执行引擎的性能实现的。像即时编译等各种优化技术被不断的引入，从而使得JVM性能得到了持续的优化和提升。老旧的JVM与最新的JVM之间最大的差异其实就来自于执行引擎的提升。

Hotspot编译器从Java 1.3开始便引入到了Oracle Hotspot VM中，而即时编译器从Android 2.2开始便被引入到了Android Dalvik VM中。

> 注释
>
> 像其他使用了像字节码一样的中间层语言的编译语言，VM在执行中间层字节码时也像JVM执行字节码一样，引入了即时编译等技术来提高VM的执行效率。像Microsoft的.Net语言，其运行时的VM叫做CLR(Common Language Runtime)。CLR执行一种类似字节码的语言CIL(Common Intermediate Language)。CLR同时提供了AOT编译器和即时编译器。因为如果使用C#或VB.NET编写程序，编译器会把源码编译成CIL，CLR通过使用即时编译器来执行CIL。CLR也有垃圾回收，并且和JVM一样也是以基于栈的方式运行。

结束语

虽然使用Java并不需要了解Java是如何被创造出来的，并且很多程序员在并没有深入研究JVM的情况下依然开发出了很多伟大的应用和类库。但是如果能够了解JVM，就能对Java 有更多深入的提高，并在解决文中案例问题场景时有所帮助。

除了上文所述，JVM还有很多特性和技术细节。JVM技术规范为JVM开发者提供了灵活的规范空间，以帮忙开发者能使用多种技术手段创造出具有更好性能的JVM实现。另外虽然垃圾回收手术已被很多具有类似VM能力的编程语言作为常用的性能提升的新手段，但因有很多对其详细介绍的资料，所以这里没有深入讲解。

作者：Se Hoon Park，消息平台开发团队，NHN公司。