JVM

# JVM介绍

JVM是Java Virtual Machine（Java虚拟机）的缩写，JVM是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。Java虚拟机包括一套字节码指令集、一组寄存器、一个栈、一个垃圾回收堆和一个存储方法域。 JVM屏蔽了与具体操作系统平台相关的信息，使Java程序只需生成在Java虚拟机上运行的目标代码（字节码）,就可以在多种平台上不加修改地运行。JVM在执行字节码时，实际上最终还是把字节码解释成具体平台上的机器指令执行。

Java语言的一个非常重要的特点就是与平台的无关性。而使用Java虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的目标代码。而引入Java语言虚拟机后，Java语言在不同平台上运行时不需要重新编译。Java语言使用Java虚拟机屏蔽了与具体平台相关的信息，使得Java语言编译程序只需生成在Java虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。Java虚拟机在执行字节码时，把字节码解释成具体平台上的机器指令执行。这就是Java的能够“一次编译，到处运行”的原因。

# 完成JVM环境

JVM在整个jdk中处于最底层,负责于操作系统的交互,用来屏蔽操作系统环境,提供一个完整的Java运行环境,因此也就虚拟计算机. 操作系统装入JVM是通过jdk中Java.exe来完成,通过下面4步来完成JVM环境.

**1.创建JVM装载环境和配置**

**2.装载JVM.dll**

**3.初始化JVM.dll并挂界到JNIENV(JNI调用接口)实例**

**4.调用JNIEnv实例装载并处理class类。**

一．JVM装入环境，JVM提供的方式是操作系统的动态连接文件．既然是文件那就一个装入路径的问题，Java是怎么找这个路径的呢？当你在调用Java test的时候，操作系统会在path下在你的Java.exe程序，Java.exe就通过下面一个过程来确定JVM的路径和相关的参数配置了．下面基于Windows的实现的分析．

首先查找jre路径，Java是通过GetApplicationHome api来获得当前的Java.exe绝对路径，c:\j2sdk1.4.2\_09\bin\Java.exe,那么它会截取到绝对路径c:\j2sdk1.4.2\_09\，判断c:\j2sdk1.4.2\_09\bin\Java.dll文件是否存在，如果存在就把c:\j2sdk1.4.2\_09\作为jre路径，如果不存在则判断c:\j2sdk1.4.2\_09\jre\bin\Java.dll是否存在，如果存在这c:\j2sdk1.4.2\_09\jre作为jre路径．如果不存在调用GetPublicJREHome查HKEY\_LOCAL\_MACHINE\Software\JavaSoft\Java Runtime Environment\“当前JRE版本号”\JavaHome的路径为jre路径。

然后装载JVM.cfg文件JRE路径+\lib+\ARCH（CPU构架）+\JVM.cfgARCH（CPU构架）的判断是通过Java\_md.c中GetArch函数判断的，该函数中windows平台只有两种情况：WIN64的‘ia64’，其他情况都为‘i386’。以我的为例：C:\j2sdk1.4.2\_09\jre\lib\i386\JVM.cfg.主要的内容如下：

-client KNOWN

-server KNOWN

-hotspot ALIASED\_TO -client

-classic WARN

-native ERROR

-green ERROR

在我们的jdk目录中jre\bin\server和jre\bin\client都有JVM.dll文件存在，而Java正是通过JVM.cfg配置文件来管理这些不同版本的JVM.dll的．通过文件我们可以定义目前jdk中支持那些JVM,前面部分（client）是JVM名称，后面是参数，KNOWN表示JVM存在，ALIASED\_TO表示给别的JVM取一个别名，WARN表示不存在时找一个JVM替代，ERROR表示不存在抛出异常．在运行Java XXX是，Java.exe会通过CheckJVMType来检查当前的JVM类型，Java可以通过两种参数的方式来指定具体的JVM类型，一种按照JVM.cfg文件中的JVM名称指定，第二种方法是直接指定，它们执行的方法分别是“Java -J”、“Java -XXaltJVM=”或“Java -J-XXaltJVM=”。如果是第一种参数传递方式，CheckJVMType函数会取参数‘-J’后面的JVM名称，然后从已知的JVM配置参数中查找如果找到同名的则去掉该JVM名称前的‘-’直接返回该值；而第二种方法，会直接返回“-XXaltJVM=”或“-J-XXaltJVM=”后面的JVM类型名称；如果在运行Java时未指定上面两种方法中的任一一种参数，CheckJVMType会取配置文件中第一个配置中的JVM名称，去掉名称前面的‘-’返回该值。CheckJVMType函数的这个返回值会在下面的函数中汇同jre路径组合成JVM.dll的绝对路径。如果没有指定这会使用JVM.cfg中第一个定义的JVM.可以通过set \_Java\_LAUNCHER\_DEBUG=1在控制台上测试．

最后获得JVM.dll的路径，JRE路径+\bin+\JVM类型字符串+\JVM.dll就是JVM的文件路径了，但是如果在调用Java程序时用-XXaltJVM=参数指定的路径path,就直接用path+\JVM.dll文件做为JVM.dll的文件路径．

二：装载JVM.dll

通过第一步已经找到了JVM的路径，Java通过LoadJavaVM来装入JVM.dll文件．装入工作很简单就是调用Windows API函数：

LoadLibrary装载JVM.dll动态连接库．然后把JVM.dll中的导出函数JNI\_CreateJavaVM和JNI\_GetDefaultJavaVMInitArgs挂接到InvocationFunctions变量的CreateJavaVM和GetDefaultJavaVMInitArgs函数指针变量上。JVM.dll的装载工作宣告完成。

三：初始化JVM，获得本地调用接口，这样就可以在Java中调用JVM的函数了．调用InvocationFunctions－>CreateJavaVM也就是JVM中JNI\_CreateJavaVM方法获得JNIEnv结构的实例．

四：运行Java程序．

Java程序有两种方式一种是jar包，一种是class. 运行jar,Java -jar XXX.jar运行的时候，Java.exe调用GetMainClassName函数，该函数先获得JNIEnv实例然后调用Java类Java.util.jar.JarFileJNIEnv中方法getManifest()并从返回的Manifest对象中取getAttributes("Main-Class")的值即jar包中文件：META-INF/MANIFEST.MF指定的Main-Class的主类名作为运行的主类。之后main函数会调用Java.c中LoadClass方法装载该主类（使用JNIEnv实例的FindClass）。main函数直接调用Java.c中LoadClass方法装载该类。如果是执行class方法。main函数直接调用Java.c中LoadClass方法装载该类。

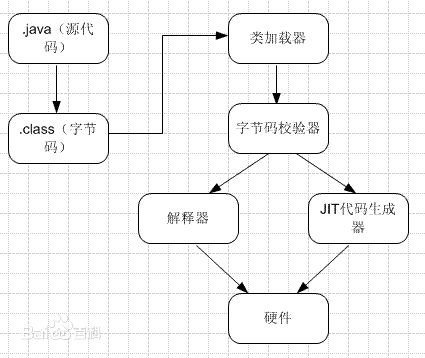
然后main函数调用JNIEnv实例的GetStaticMethodID方法查找装载的class主类中

“public static void main(String[] args)”方法，并判断该方法是否为public方法，然后调用JNIEnv实例的

CallStaticVoidMethod方法调用该Java类的main方法。

# JVM原理

JVM是java的核心和基础，在java编译器和os平台之间的虚拟处理器。它是一种利用软件方法实现的抽象的计算机基于下层的操作系统和硬件平台，可以在上面执行java的字节码程序。



java编译器只要面向JVM，生成JVM能理解的代码或字节码文件。Java源文件经编译成字节码程序，通过JVM将每一条指令翻译成不同平台机器码，通过特定平台运行。

# JVM执行程序的过程

1) 加载.class文件

2) 管理并分配内存

3) 执行垃圾收集

JRE（java运行时环境）由JVM构造的java程序的运行环，也是Java程序运行的环境，但是他同时一个操作系统的一个应用程序一个进程，因此他也有他自己的运行的生命周期，也有自己的代码和数据空间。JVM在整个jdk中处于最底层，负责于操作系统的交互，用来屏蔽操作系统环境，提供一个完整的Java运行环境，因此也就虚拟计算机。操作系统装入JVM是通过jdk中Java.exe来完成，通过下面4步来完成JVM环境：1) 创建JVM装载环境和配置 2) 装载JVM.dll 3) 初始化JVM.dll并挂界到JNIENV(JNI调用接口)实例4) 调用JNIEnv实例装载并处理class类。

# JVM的生命周期

1.       JVM实例对应了一个独立运行的java程序它是进程级别

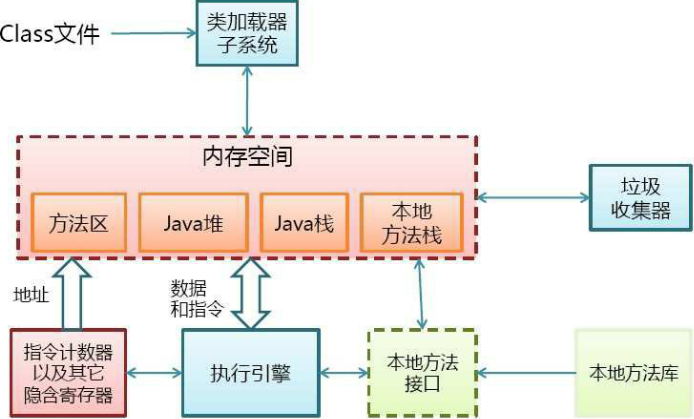
a)     启动。启动一个Java程序时，一个JVM实例就产生了，任何一个拥有public static void main(String[] args)函数的class都可以作为JVM实例运行的起点

b)     运行。main()作为该程序初始线程的起点，任何其他线程均由该线程启动。JVM内部有两种线程：守护线程和非守护线程，main()属于非守护线程，守护线程通常由JVM自己使用，java程序也可以标明自己创建的线程是守护线程

c)     消亡。当程序中的所有非守护线程都终止时，JVM才退出；若安全管理器允许，程序也可以使用Runtime类或者System.exit()来退出

2.       JVM执行引擎实例则对应了属于用户运行程序的线程它是线程级别的

# JVM的体系结构



* 类装载器（ClassLoader）（用来装载.class文件）
* 执行引擎（执行字节码，或者执行本地方法）
* 运行时数据区（方法区、堆、java栈、PC寄存器、本地方法栈）

JAVA内存控制相关

# Java堆、栈和常量池以及相关String的详细讲解

一、在JAVA中，有六个不同的地方可以存储数据：

1. 寄存器（register）。 这是最快的存储区，因为它位于不同于其他存储区的地方——处理器内部。但是寄存器的数量极其有限，所以寄存器由编译器根据需求进行分配。你不能直接控制，也不能在程序中感觉到寄存器存在的任何迹象。

------最快的存储区, 由编译器根据需求进行分配,我们在程序中无法控制.

2. 堆栈（stack）。位于通用RAM中，但通过它的“堆栈指针”可以从处理器哪里获得支持。堆栈指针若向下移动，则分配新的内存；若向上移动，则释放那些 内存。这是一种快速有效的分配存储方法，仅次于寄存器。创建程序时候，JAVA编译器必须知道存储在堆栈内所有数据的确切大小和生命周期，因为它必须生成 相应的代码，以便上下移动堆栈指针。这一约束限制了程序的灵活性，所以虽然某些JAVA数据存储在堆栈中——特别是对象引用，但是JAVA对象不存储其 中。

------存放基本类型的变量数据和对象，数组的引用，但对象本身不存放在栈中，而是存放在堆（new 出来的对象）或者常量池中（字符串常量对象存放在常量池中）

3. 堆（heap）。一种通用性的内存池（也存在于RAM中），用于存放所以的JAVA对象。堆不同于堆栈的好处是：编译器不需要知道要从堆里分配多少存储区 域，也不必知道存储的数据在堆里存活多长时间。因此，在堆里分配存储有很大的灵活性。当你需要创建一个对象的时候，只需要new写一行简单的代码，当执行 这行代码时，会自动在堆里进行存储分配。当然，为这种灵活性必须要付出相应的代码。用堆进行存储分配比用堆栈进行存储存储需要更多的时间。

------存放所有new出来的对象。

4. 静态存储（static storage）。这里的“静态”是指“在固定的位置”。静态存储里存放程序运行时一直存在的数据。你可用关键字static来标识一个对象的特定元素是静态的，但JAVA对象本身从来不会存放在静态存储空间里。

------存放静态成员（static定义的）

5. 常量存储（constant storage）。常量值通常直接存放在程序代码内部，这样做是安全的，因为它们永远不会被改变。有时，在嵌入式系统中，常量本身会和其他部分分割离开，所以在这种情况下，可以选择将其放在ROM中

------存放字符串常量和基本类型常量（public static final）

6. 非RAM存储。如果数据完全存活于程序之外，那么它可以不受程序的任何控制，在程序没有运行时也可以存在。

------硬盘等永久存储空间 就速度来说，有如下关系：

    寄存器 >堆栈 > 堆 > 其它

    这里我们主要关心栈，堆和常量池，对于栈和常量池中的对象可以共享，对于堆中的对象不可以共享。

     栈中的数据大小和生命周期是可以确定的，当没有引用指向数据时，这个数据就会消失。堆中的对象的由垃圾回收器负责回收，因此大小和生命周期不需要确定，具有很大的灵活性。

    对于字符串：其对象的引用都是存储在栈中的，如果是编译期已经创建好(直接用双引号定义的)的就存储在常量池中，如果是运行期（new出来的）才能确定的就存储在堆中。对于equals相等的字符串，在常量池中永远只有一份，在堆中有多份。

     如以下代码： Java代码

        String s1 = "china";

        String s2 = "china";

        String s3 = "china";

        String ss1 = new String("china");

        String ss2 = new String("china");

        String ss3 = new String("china");

        这里解释一下，对于通过 new 产生一个字符串（假设为 “china” ）时，会先去常量池中查找是否已经有了 “china” 对象，如果没有则在常量池中创建一个此字符串对象，然后堆中再创建一个常量池中此 “china” 对象的拷贝对象。

        也就是有道面试题： **String s = new String(“xyz”); 产生几个对象？**

**一个或两个。**如果常量池中原来没有 “xyz”, 就是两个。如果原来的常量池中存在“xyz”时，就是一个。

        对于基础类型的变量和常量：变量和引用存储在栈中，常量存储在常量池中。

        如以下代码： Java代码

        int i1 = 9;

        int i2 = 9;

        int i3 = 9;

        public static final int INT1 = 9;

        public static final int INT2 = 9;

        public static final int INT3 = 9;

        对于成员变量和局部变量：成员变量就是方法外部，类的内部定义的变量；

     局部变量就是方法或语句块内部定义的变量。局部变量必须初始化。 形式参数是局部变量，局部变量的数据存在于栈内存中。栈内存中的局部变量随着方法的消失而消失。 成员变量存储在堆中的对象里面，由垃圾回收器负责回收。

     如以下代码： Java代码

     class BirthDate {

       private int day;

       private int month;

       private int year;

       public BirthDate(int d, int m, int y) {

                   day = d;

                   month = m;

                    year = y;

                   }

     // 省略get,set方法………

  }

 public class Test{

    public static void main(String args[]){

      int date = 9;

      Test test = new Test();

      test.change(date);

      BirthDate d1= new BirthDate(7,7,1970);

    }

  public void change1(int i){

        i = 1234;

    }

  对于以上这段代码，date为局部变量，i,d,m,y都是形参为局部变量，day，month，year为成员变量。

  下面分析一下代码执行时候的变化：

  1. main方法开始执行：int date = 9; date局部变量，基础类型，引用和值都存在栈中。

  2. Test test = new Test(); test为对象引用，存在栈中，对象(new Test())存在堆中。

  3. test.change(date); i为局部变量，引用和值存在栈中。当方法change执行完成后，i就会从栈中消失。

  4. BirthDate d1= new BirthDate(7,7,1970);  d1 为对象引用，存在栈中，对象(new BirthDate())存在堆中，其中d，m，y为局部变量存储在栈中，且它们的类型为基础类型，因此它们的数据也存储在栈中。 day,month,year为成员变量，它们存储在堆中(new BirthDate()里面)。当BirthDate构造方法执行完之后，d,m,y将从栈中消失。

  5.main方法执行完之后，date变量，test，d1引用将从栈中消失，new Test(),new BirthDate()将等待垃圾回收。

# 内存控制心中有数

## 两个读取内存信息的函数

1. Runtime.getRuntime().maxMemory();

得到虚拟机可以控制的最大内存数量。

1. Runtime.getRuntime().totalMemory();

得到虚拟机当前已经使用的内存数量。

**public** **static** **void** main(String[] args)

{

String s = "abcdefghijklmnop";

System.***out***.print("maxMemory:");

System.***out***.println(Runtime.*getRuntime*().maxMemory()/1024/1024 + "M");

System.***out***.print("totalMemory:");

System.***out***.println(Runtime.*getRuntime*().totalMemory()/1024/1024 + "M");

**for**(**int** i = 0; i < 19; i++) {

s = s + s;

}

System.***out***.print("totalMemory:");

System.***out***.println(Runtime.*getRuntime*().totalMemory()/1024/1024 + "M");

}

## 设置堆栈和内存大小

1．设置Eclipse内存使用情况

修改eclipse根目录下的eclipse.ini文件

-vmargs //虚拟机设置

-Xms40m

-Xmx256m

-XX:PermSize=128M //非堆内存设置

-XX:MaxPermSize=256M

2．JVM内存设置

打开eclipse window－preferences－Java －Installed JREs －Edit －Default VM Arguments

在VM自变量中输入：-Xmx128m -Xms64m -Xmn32m -Xss16m

3．Tomcat内存设置

打开Tomcat根目录下的bin文件夹，编辑catalina.bat

修改为：set JAVA\_OPTS= -Xms256m -Xmx512m

4. 在CMD中设置

java -Xmx1024m -classpath ……

# 内存控制效率优化的启示

## 启示1：String和StringBuffer的不同之处

使用String：

**public** **class** MemoryTest {

**public** **static** **void** main(String[] args) {

String s = "abcdefghijklmnop";

System.***out***.print("maxMemory:");

System.***out***.println(Runtime.*getRuntime*().maxMemory()/1024/1024 + "M");

System.***out***.print("循环前totalMemory:");

System.***out***.println(Runtime.*getRuntime*().totalMemory()/1024/1024 + "M");

**int** count = 0;

**while**(**true**) {

**try** {

s += s;

count++;

} **catch** (Error o) {

System.***out***.println("循环次数:" + count);

System.***out***.println("String实际字节数:" + s.length()/1024/1024 + "M");

System.***out***.print("循环后，已占用内存:");

System.***out***.println(Runtime.*getRuntime*().totalMemory()/1024/1024 + "M");

System.***out***.println("Catch到的错误：" + o);

**break**;

}

}

}

}

运行结果：

maxMemory:245M

循环前totalMemory:60M

循环次数:21

String实际字节数:32M

循环后，已占用内存:255M

Catch到的错误：java.lang.OutOfMemoryError: Java heap space

使用StringBuffer：

**package** lee;

**public** **class** MemoryTest {

**public** **static** **void** main(String[] args) {

**StringBuffer s = new StringBuffer("abcdefghijklmnop");**

System.***out***.print("maxMemory:");

System.***out***.println(Runtime.*getRuntime*().maxMemory()/1024/1024 + "M");

System.***out***.print("循环前totalMemory:");

System.***out***.println(Runtime.*getRuntime*().totalMemory()/1024/1024 + "M");

**int** count = 0;

**while**(**true**) {

**try** {

**s.append(s);**

count++;

} **catch** (Error o) {

System.***out***.println("循环次数:" + count);

System.***out***.println("String实际字节数:" + s.length()/1024/1024 + "M");

System.***out***.print("循环后，已占用内存:");

System.***out***.println(Runtime.*getRuntime*().totalMemory()/1024/1024 + "M");

System.***out***.println("Catch到的错误：" + o);

**break**;

}

}

}

}

运行结果：

maxMemory:245M

循环前totalMemory:60M

循环次数:22

String实际字节数:64M

循环后，已占用内存:252M

Catch到的错误：java.lang.OutOfMemoryError: Java heap space

## 启示2：用-Xmx参数来提高内存可控制量

参照设置堆栈和内存大小

## 启示3：二维数组比一维数组占用更多的内存空间