# Java JVM 运行机制及基本原理

## 1 JVM的基础概念

JVM的中文名称叫Java虚拟机，它是由软件技术模拟出计算机运行的一个虚拟的计算机。

JVM也充当着一个翻译官的角色，我们编写出的Java程序，是不能够被操作系统所直接识别的，这时候JVM的作用就体现出来了，它负责把我们的程序翻译给系统“听”，告诉它我们的程序需要做什么操作。

我们都知道Java的程序需要经过编译后，产生.Class文件，JVM才能识别并运行它，JVM针对每个操作系统开发其对应的解释器，所以只要其操作系统有对应版本的JVM，那么这份Java编译后的代码就能够运行起来，这就是Java能一次编译，到处运行的原因。

## 2 JVM的生命周期

JVM在Java程序开始执行的时候，它才运行，程序结束的时它就停止。

一个Java程序会开启一个JVM进程，如果一台机器上运行三个程序，那么就会有三个运行中的JVM进程。

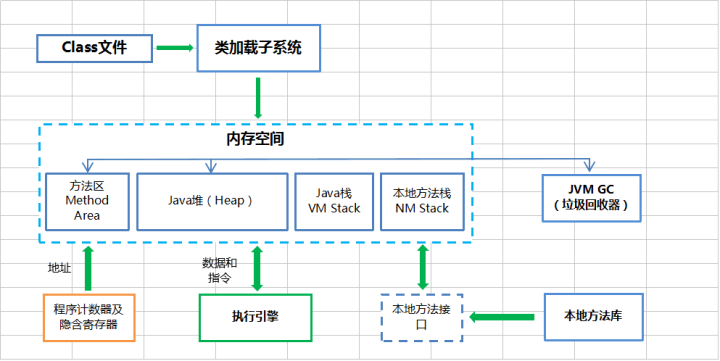
JVM中的线程分为两种：守护线程和普通线程

**守护线程**是JVM自己使用的线程，比如垃圾回收（GC）就是一个守护线程。

**普通线程**一般是Java程序的线程，只要JVM中有普通线程在执行，那么JVM就不会停止。

**PS：权限足够的话，可以调用exit()方法终止程序。**

## 3 JVM的结构体系



JVM = 类加载器 classloader

+ 执行引擎 execution engine

+ 运行时数据区域 runtime data area

## 4 JVM的启动过程

### 4.1 JVM的装入环境和配置

在学习这个之前，我们需要了解一件事情，就是JDK和JRE的区别。

JDK是面向开发人员使用的SDK，它提供了Java的开发环境和运行环境，JDK中包含了JRE。

JRE是Java的运行环境，是面向所有Java程序的使用者，包括开发者，JRE = 运行环境 = JVM。

如果安装了JDK，会发现电脑中有两套JRE，一套位于/Java/jre.../下，一套位于/Java/jdk.../jre下。那么问题来了，一台机器上有两套以上JRE，谁来决定运行那一套呢？这个任务就落到java.exe身上，java.exe的任务就是找到合适的JRE来运行java程序。java.exe按照以下的顺序来选择JRE：

1、自己目录下有没有JRE

2、父目录下有没有JRE

3、查询注册表： HKEY\_LOCAL\_MACHINE\SOFTWARE\JavaSoft\Java Runtime Environment\"当前JRE版本号"\JavaHome

这几步的主要核心是为了找到JVM的绝对路径。

**jvm.cfg**的路径为：JRE路径\lib\"CPU架构"\jvm.fig

**jvm.cfg**的内容大致如下：

-client KNOWN   
-server KNOWN   
-hotspot ALIASED\_TO -client   
-classic WARN   
-native ERROR   
-green ERROR

其中：KNOWN 表示存在 ；IGNORE 表示不存在 ；ALIASED\_TO 表示给别的JVM去一个别名；WARN 表示不存在时找一个替代 ；ERROR 表示不存在抛出异常。

### 4.2 装载JVM

通过第一步找到JVM的路径后，Java.exe通过LoadJavaVM来装入JVM.dll文件。

LoadLibrary装载JVM动态连接库，然后把JVM.dll中的导出函数JNI\_CreateJavaVM和JNI\_GetDefaultJavaVMIntArgs 挂接到InvocationFunction 变量的CreateJavaVM和GetDafaultJavaVMInitArgs 函数指针变量上。JVM的装载工作完成。

### 4.3 初始化JVM，获得本地调用接口

调用InvocationFunction -> CreateJavaVM也就是JVM中JNI\_CreateJavaVM方法获得JNIEnv结构的实例。

### 4.4 运行Java程序

JVM运行Java程序的方式有两种：jar包与 Class。

**运行jar 的时候**：Java.exe调用GetMainClassName函数，该函数先获得JNIEnv实例然后调用JarFileJNIEnv类中getManifest()，从其返回的Manifest对象中取getAttrebutes("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）。

**运行Class的时候**：main函数直接调用Java.c中的LoadClass方法装载该类。

**5 Class文件**

Class文件由Java编译器生成，我们创建的.Java文件在经过编译器后，会变成.Class的文件，这样才能被JVM所识别并运行。

**6 类加载子系统（Classloader）**

Classloader 把硬盘上的class 文件加载到JVM中的运行时数据区域, 但是它不负责这个类文件能否执行，而这个是 执行引擎 负责的。

**类加载子系统也可以称之为类加载器，JVM默认提供三个类加载器：**

1、BootStrap ClassLoader ：称之为启动类加载器，是最顶层的类加载器，负责加载JDK中的核心类库，如 rt.jar、resources.jar、charsets.jar等。

2、Extension ClassLoader：称之为扩展类加载器，负责加载Java的扩展类库，默认加载$JAVA\_HOME中jre/lib/\*.jar 或 -Djava.ext.dirs指定目录下的jar包。

3、App ClassLoader：称之为系统类加载器，负责加载应用程序classpath目录下所有jar和class文件。

4、除了Java默认提供的三个ClassLoader（加载器）之外，我们还可以根据自身需要自定义ClassLoader，自定义ClassLoader必须继承java.lang.ClassLoader 类。

**除了BootStrap ClassLoader 之外**的另外两个默认加载器都是继承自java.lang.ClassLoader 。BootStrap ClassLoader 不是一个普通的Java类，它底层由C++编写，已嵌入到了JVM的内核当中，当JVM启动后，BootStrap ClassLoader 也随之启动，负责加载完核心类库后，并构造Extension ClassLoader 和App ClassLoader 类加载器。

类加载器子系统不仅仅负责定位并加载类文件，它还严格按照以下步骤做了很多事情：

1、加载：寻找并导入Class文件的二进制信息

2、连接：进行验证、准备和解析

1）验证：确保导入类型的正确性

2）准备：为类型分配内存并初始化为默认值

3）解析：将字符引用解析为直接引用

3、初始化：调用Java代码，初始化类变量为指定初始值

**7 方法区（Method Area）**

在JVM中，**类型信息**和**类静态变量**都保存在方法区中，类型信息是由类加载器在类加载的过程中从类文件中提取出来的信息。

需要注意的一点是，**常量池也存放于方法区中**。

**程序中所有的线程共享一个方法区，所以访问方法区的信息必须确保线程是安全的。**如果有两个线程同时去加载一个类，那么只能有一个线程被允许去加载这个类，另一个必须等待。

在程序运行时，方法区的大小是可以改变的，程序在运行时可以扩展。

方法区也可以被垃圾回收，但条件非常严苛，必须在该类没有任何引用的情况下。

**类型信息包括：**

1、类型的全名（The fully qualified name of the type）

2、类型的父类型全名（除非没有父类型，或者父类型是java.lang.Object）（The fully qualified name of the typeís direct superclass）

3、该类型是一个类还是接口（class or an interface）（Whether or not the type is a class ）

4、类型的修饰符（public，private，protected，static，final，volatile，transient等）（The typeís modifiers）

5、所有父接口全名的列表（An ordered list of the fully qualified names of any direct superinterfaces）

6、类型的字段信息（Field information）

7、类型的方法信息（Method information）

8、所有静态类变量（非常量）信息（All class (static) variables declared in the type, except constants）

9、一个指向类加载器的引用（A reference to class ClassLoader）

10、一个指向Class类的引用（A reference to class Class）

11、基本类型的常量池（The constant pool for the type）

**方法列表（Method Tables）**

为了更高效的访问所有保存在方法区中的数据，在方法区中，除了保存上边的这些类型信息之外，还有一个为了加快存取速度而设计的数据结构：方法列表。每一个被加载的非抽象类，Java虚拟机都会为他们产生一个方法列表，这个列表中保存了这个类可能调用的所有实例方法的引用，保存那些父类中调用的方法。

**8 Java堆（JVM堆、Heap）**

1、当Java创建一个类的实例对象或者数组时，都在堆中为新的对象分配内存。

2、虚拟机中只有一个堆，程序中所有的线程都共享它。

3、堆占用的内存空间是最多的。

4、堆的存取类型为管道类型，先进先出。

5、在程序运行中，可以动态的分配堆的内存大小。

6、堆的内存资源回收是交给JVM GC进行管理的。

**9 Java栈（JVM栈、Stack）**

在Java栈中**只保存基础数据类型**（**四类八种**）和自定义对象的**引用**，**注意只是对象的引用而不是对象本身哦**，对象是保存在堆区中的。

**基本数据结构（四类）：**

**1）整型**byte 、short 、int 、long

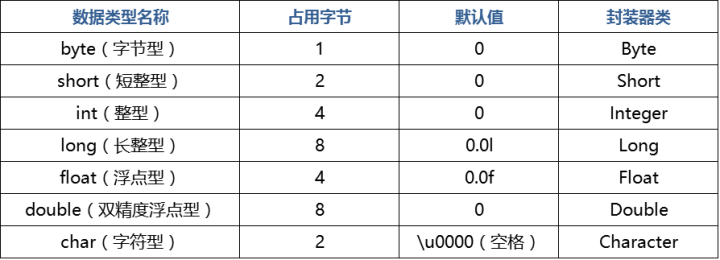
**2）浮点型**float 、 double

**3）字符型**char

**4）布尔型**boolean

**拓展知识：像String、Integer、Byte、Short、Long、Character、Boolean这六个属于包装类型，它们是存放于堆中的。**

**基本数据结构（八种）：**



1、栈的存取类型为类似于水杯，先进后出。

2、栈内的数据在超出其作用域后，会被自动释放掉，**它不由JVM GC管理**。

3、每一个线程都包含一个栈区，每个栈中的数据都是私有的，其他栈不能访问。

4、每个线程都会建立一个操作栈，每个栈又包含了若干个栈帧，每个栈帧对应着每个方法的每次调用，每个栈帧包含了三部分：

1）局部变量区（方法内基本类型变量、变量对象指针）

2）操作数栈区（存放方法执行过程中产生的中间结果）

3）运行环境区（动态连接、正确的方法返回相关信息、异常捕捉）

**10 本地方法栈**

1、本地方法栈的功能和JVM栈非常类似，用于存储本地方法的局部变量表，本地方法的操作数栈等信息。

2、栈的存取类型为类似于水杯，先进后出。

3、栈内的数据在超出其作用域后，会被自动释放掉，**它不由JVM GC管理。**

4、每一个线程都包含一个栈区，每个栈中的数据都是私有的，其他栈不能访问。

5、本地方法栈是在程序调用或JVM调用**本地方法接口（Native）**时候启用。

6、本地方法都不是使用Java语言编写的，比如使用C语言编写的本地方法，本地方法也不由JVM去运行，所以本地方法的运行不受JVM管理。

7、HotSpot VM将本地方法栈和JVM栈合并了。

**11 程序计数器**

1、**在JVM的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。**分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

2、JVM的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，为了各条线程之间的切换后计数器能恢复到正确的执行位置，所以**每条线程都会有一个独立的程序计数器**。

3、程序计数器仅占很小的一块内存空间。

4、**当线程正在执行一个Java方法，程序计数器记录的是正在执行的JVM字节码指令的地址。如果正在执行的是一个Natvie（本地方法），那么这个计数器的值则为空（Underfined）。**

5、程序计数器这个内存区域是唯一一个在JVM规范中没有规定任何OutOfMemoryError（内存不足错误）的区域。

**12 JVM执行引擎**

Java虚拟机相当于一台虚拟的“物理机”，这两种机器都有代码执行能力，其区别主要是物理机的执行引擎是直接建立在处理器、硬件、指令集和操作系统层面上的。而JVM的执行引擎是自己实现的，因此程序员可以自行制定指令集和执行引擎的结构体系，因此**能够执行那些不被硬件直接支持的指令集格式。**

1）在JVM规范中制定了虚拟机字节码执行引擎的概念模型，这个模型称之为JVM执行引擎的统一外观。

2）JVM实现中，可能会有两种的执行方式：解释执行（通过解释器执行）和编译执行（通过即时编译器产生本地代码）。

3）有些虚拟机只采用一种执行方式，有些则可能同时采用两种，甚至有可能包含几个不同级别的编译器执行引擎。

**4）输入的是字节码文件、处理过程是等效字节码解析过程、输出的是执行结果**。在这三点上每个JVM执行引擎都是一致的。

**13 本地方法接口（JNI）**

JNI是Java Native Interface的缩写，它提供了若干的API实现了Java和其他语言的通信（主要是C和C++）。

**JNI的适用场景**

当我们有一些旧的库，已经使用C语言编写好了，如果要移植到Java上来，非常浪费时间，而JNI可以支持Java程序与C语言编写的库进行交互，这样就不必要进行移植了。或者是与硬件、操作系统进行交互、提高程序的性能等，都可以使用JNI。需要注意的一点是需要保证本地代码能工作在任何Java虚拟机环境。

**JNI的副作用**：一旦使用JNI，Java程序将丢失了Java平台的两个优点

1、**程序不再跨平台**，要想跨平台，必须在不同的系统环境下程序编译配置本地语言部分。

2、**程序不再是绝对安全的**，本地代码的使用不当可能会导致整个程序崩溃。一个通用规则是，调用本地方法应该集中在少数的几个类当中，这样就降低了Java和其他语言之间的耦合。

**14 JVM GC（垃圾回收机制）**

**15 JVM 常量池**

**JVM常量池也称之为运行时常量池，它是方法区（Method Area）的一部分。用于存放编译期间生成的各种字面量和符号引用。**运行时常量池不要求一定只有在编译器产生的才能进入，运行期间也可以将新的常量放入池中，这种特性被开发人员利用比较多的就是String.intern()方法。

由“用于存放编译期间生成的各种字面量和符号引用”这句话可见，常量池中存储的是对象的引用而不是对象的本身。

**常量池的好处：**

常量池是为了避免频繁的创建和销毁对象而影响系统性能，它也实现了对象的共享。例如字符串常量池：在编译阶段就把所有字符串文字放到一个常量池中。

1）节省内存空间：常量池中如果有对应的字符串，那么则返回该对象的引用，从而不必再次创建一个新对象。

2）节省运行时间：比较字符串时，==比equals()快。对于两个引用变量，==判断引用是否相等，也就可以判断实际值是否相等。

**PS：此处双等号（==）的含义**

1）基本数据类型之间使用双等号，比较的是数值。

2）复合数据类型（类）之间使用双等号，比较的是对象的引用地址是否相等。

**八种基本类型的包装类和String包装类与常量池**

**1、Byte、Short、Integer、Long、Character、Boolean这6种基本类型的包装类都各自实现了自己的常量池**

1）Byte、Short、Integer、Long、Character这5种包装类都默认创建了数值[-128 , 127]的缓存数据；

2）当对这5个类型的数据不在这个区间内的时候，将会去创建新的对象，**并且不会将这些新的对象放入常量池中。**

**2、Float 和Double 这2中基本类型的包装类没有实现常量池**

**3、String包装类实现了常量池**

**1）String.intern()方法：**对于使用了new 创建的字符串对象，如果想要将这个对象引用到字符串常量池，可以使用intern() 方法；

**2）对于常量池中str1和str2：**str3=str1+str2依然在常量池中。（）

**3）对于String str1=”\*\*\*”;如果常量池中不存在”\*\*\*”时**：则在堆中创建一个相应的对象，将创建的对象的引用存放到常量池中，同时将引用返回给变量 **str 1**。对于**String str2=”\*\*\*”;中str2==str1是True的。**

**PS：只有2）和3）情况下字符串对象引用自动加入字符串常量池中。**

## Java 基本数据类型 - 四类八种

**一、Java四大数据类型分类**

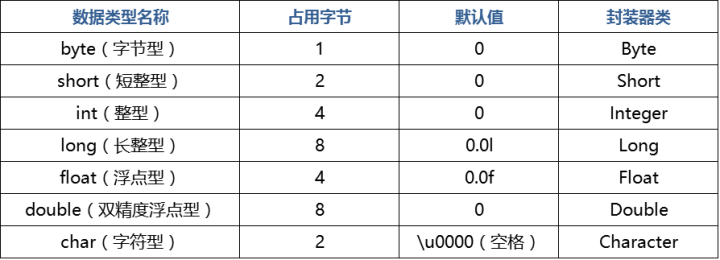
**1、整型**byte 、short 、int 、long

**2、浮点型**float 、 double

**3、字符型**char

**4、布尔型**boolean

**二、八种基本数据类型**



**三、数据类型详细介绍**

整型（byte、short、int、long）

虽然byte、short、int、long 数据类型都是表示整数的，但是它们的取值范围可不一样。

byte 的取值范围：-128～127（-2的7次方到2的7次方-1）

short 的取值范围：-32768～32767（-2的15次方到2的15次方-1）

int 的取值范围：-2147483648～2147483647（-2的31次方到2的31次方-1）

long 的取值范围：-9223372036854774808～9223372036854774807（-2的63次方到2的63次方-1）

由上可以看出 byte、short 的取值范围比较小，而long的取值范围时最大的，所以占用的空间也是最多的。int 取值范围基本上可以满足我们的日常计算需求了，所以 int 也是我们使用的最多的一个整型类型。

浮点型（float、double）

float 和 double 都是表示浮点型的数据类型，它们之间的区别在于精确度的不同。

float（单精度浮点型）取值范围：3.402823e+38～1.401298e-45（e+38 表示乘以10的38次方，而e-45 表示乘以10的负45次方）

double（双精度浮点型）取值范围：1.797693e+308～4.9000000e-324（同上）

double 类型比float 类型存储范围更大，精度更高。

通常的浮点型数据在不声明的情况下都是double型的，如果要表示一个数据时float 型的，可以在数据后面加上 "F" 。

浮点型的数据是不能完全精确的，有时候在计算时可能出现小数点最后几位出现浮动，这时正常的。

字符型（char）

char 有以下的初始化方式：

char ch = 'a'; // 可以是汉字，因为是Unicode编码

char ch = 1010; // 可以是十进制数、八进制数、十六进制数等等。

char ch = '\0'; // 可以用字符编码来初始化，如：'\0' 表示结束符，它的ascll码是0，这句话的意思和 ch = 0 是一个意思。

Java是用unicode 来表示字符，“中” 这个中文字符的unicode 就是两个字节。

String.getBytes(encoding) 方法获取的是指定编码的byte数组表示。

通常gbk / gb2312 是两个字节，utf-8 是3个字节。

如果不指定encoding 则获取系统默认encoding 。

布尔型（boolean）

boolean 没有什么好说的，它的取值就两个：true 、false 。

**四、基本类型之间的转换**

将一种类型的值赋值给另一种类型是很常见的。在Java中，boolean 类型与其他7中类型的数据都不能进行转换，这一点很明确。但对于其他7种数据类型，它们之间都可以进行转换，只是可能会存在精度损失或其他一些变化。

转换分为自动转换和强制转换：

自动转换（隐式）：无需任何操作。

强制转换（显式）：需使用转换操作符（type）。

将6种数据类型按下面顺序排列一下：

double > float > long > int > short > byte

如果从小转换到大，那么可以直接转换，而从大到小，或char 和其他6种数据类型转换，则必须使用强制转换。

**1、自动转换**

自动转换时发生扩宽（widening conversion）。因为较大的类型（如int）要保存较小的类型（如byte），内存总是足够的，不需要强制转换。如果将字面值保存到byte、short、char、long的时候，也会自动进行类型转换。注意区别，此时从int（没有带L的整型字面值为int）到byte/short/char也是自动完成的，虽然它们都比int小。在自动类型转化中，除了以下几种情况可能会导致精度损失以外，其他的转换都不能出现精度损失。

》int--> float

》long--> float

》long--> double

》float -->double without strictfp

除了可能的精度损失外，自动转换不会出现任何运行时（run-time）异常。

**2、强制类型转换**

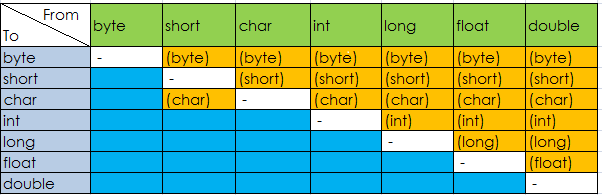
如果要把大的转成小的，或者在short与char之间进行转换，就必须强制转换，也被称作缩小转换（narrowing conversion）,因为必须显式地使数值更小以适应目标类型。强制转换采用转换操作符（）。严格地说，将byte转为char不属于narrowing conversion），因为从byte到char的过程其实是byte-->int-->char，所以widening和narrowing都有。强制转换除了可能的精度损失外，还可能使模（overall magnitude）发生变化。强制转换格式如下：

**(**target**-**type**)** value**;**

如果整数的值超出了byte所能表示的范围，结果将对byte类型的范围取余数。例如a=256超出了byte的[-128,127]的范围，所以将257除以byte的范围（256）取余数得到b=1；需要注意的是，当a=200时，此时除了256取余数应该为-56，而不是200。

将浮点类型赋给整数类型的时候，会发生截尾（truncation）。也就是把小数的部分去掉，只留下整数部分。此时如果整数超出目标类型范围，一样将对目标类型的范围取余数。

7种基本类型转换总结如下图：



3、赋值及表达式中的类型转换

3.1、字面值赋值

在使用字面值对整数赋值的过程中，可以将int literal赋值给byte short char int，只要不超出范围。这个过程中的类型转换时自动完成的，但是如果你试图将long literal赋给byte，即使没有超出范围，也必须进行强制类型转换。例如 byte b = 10L；是错的，要进行强制转换。

3.2、表达式中的自动类型提升

除了赋值以外，表达式计算过程中也可能发生一些类型转换。在表达式中，类型提升规则如下：

· 所有byte/short/char都被提升为int。

· 如果有一个操作数为long，整个表达式提升为long。float和double情况也一样。

拓展知识点：Java是面向对象语言，其概念为一切皆为对象，但基本数据类型算是个例外哦，基本数据类型大多是面向机器底层的类型，它是 “值” 而不是一个对象，它存放于“栈”中而不是存放于“堆”中，但Java一切皆为对象的概念不是说说而已，它为每一个基本数据类型都做了相应的包装类，我们日常使用中大多情况下都会使用着这些包装类：  
boolean Boolean  
char Character  
byte Byte  
short Short  
int Integer  
long Long  
float Float  
double Double  
String（字符串）  
包装类就是一个对象，它存放于“堆”中。