Java JVM系列课程

------ 基本Java虚拟机介绍

本文档包含以下内容：

1. JVM基本概念
2. JRE/JDK/JVM的关系
3. JVM原理介绍
4. JVM体系结构概述
5. JVM工作过程

#### 前言：

随着科技的发展，人们的交流、学习等不在受制地理位置限制，借助网络能够快速通信，那你知道“网络”到底是什么吗？你有理解数据传输原理吗？本节课系统性的回答一下大家数据到底如何传输。

#### 一 、**JVM基本概念**

#### **JRE(JavaRuntimeEnvironment):Java运行环境**

也就是Java平台。所有的Java 程序都要在JRE下才能运行。

**JDK(Java Development Kit):Java开发工具**

是程序开发者用来来编译、调试java程序用的开发工具包。

**JDK的工具也是Java程序，也需要JRE才能运行。**

为了保持JDK的独立性和完整性，在JDK的安装过程中，JRE也是 安装的一部分。所以，在JDK的安装目录下有一个名为jre的目录，用于存放JRE文件。

**JVM(JavaVirtualMachine):Java虚拟机**

是JRE的一部分。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。JVM有自己完善的硬件架构，如处理器、堆栈、寄存器等，还具有相应的指令系统。Java语言最重要的特点就是跨平台运行。使用JVM就是为了支持与操作系统无关，实现跨平台。

**那么jvm是如何实现的跨平台？**

Java是可以跨平台的编程语言，那我们首先得知道什么是平台，我们把CPU处理器与操作系统的整体叫平台。

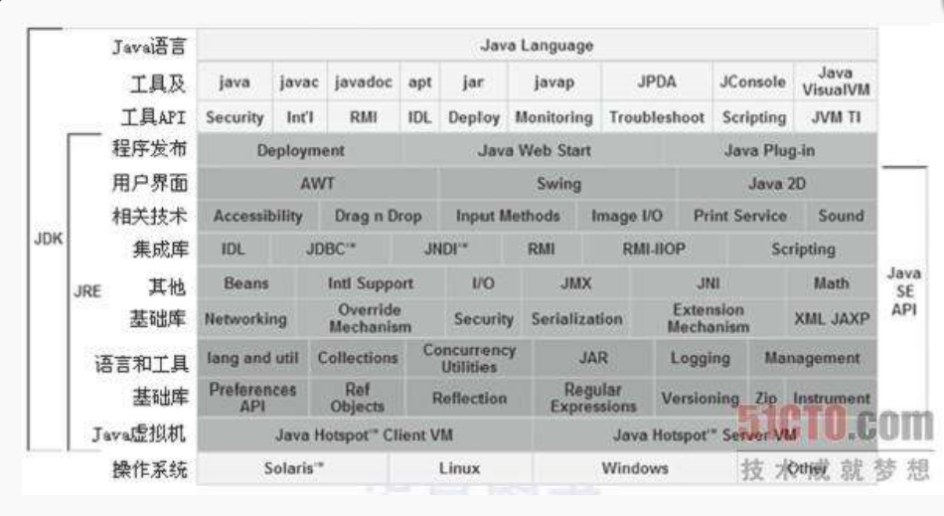
因为Java程序编译之后的代码不是能被硬件系统直接运行的代码，而是一种“中间码”——字节码。然后不同的硬件平台上安装有不同的Java虚拟机(JVM)，由JVM来把字节码再“翻译”成所对应的硬件平台能够执行的代码。因此对于Java编程者来说，不需要考虑硬件平台是什么。所以Java可以跨平台

# java类先编译成.class文件

# 通过JVM运行和操作系统无关

# 不同操作系统选择不同的JVM就OK

1. JRE/JDK/JVM的关系



JRE

　　是Java Runtime Environment，是java程序的运行环境。既然是运行，当然要包含JVM(Java Virtual Machine)，也就是虚拟机，还需要包含java类库的class文件，在jre/lib目录下打包成了jar。

JDK

　　是Java Development Kit，是java的开发工具包，里面包含了各种类库和工具，当然也包括了另外一个jre。

　　那么为什么还包含了一个jre文件呢？

　　可以看到，其中一个jre文件在jdk内部，一个jre文件在jdk外部，这就是所谓的专用JRE和公用JRE

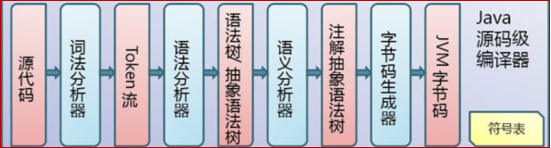
为什么有连个jre呢？

****Java代码编译和执行的整个过程****包含了以下三个重要的机制：

1）Java源码编译机制；2）类加载机制；3）类执行机制

其中，****Java源码编译****由以下三个过程组成：

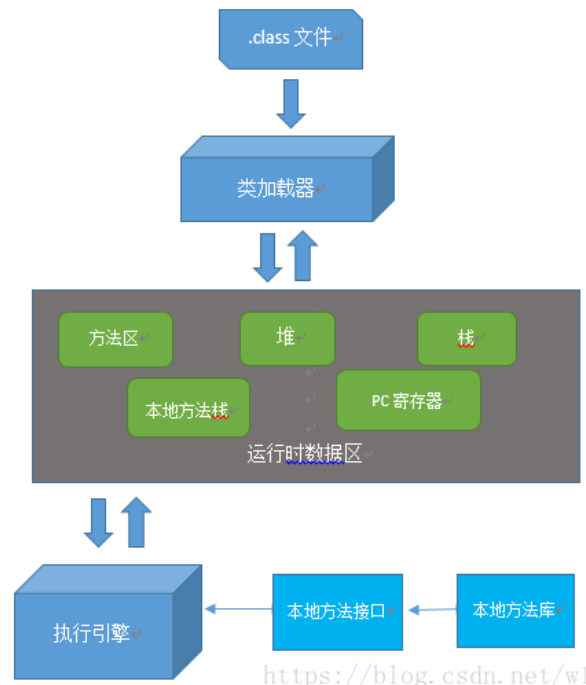
1）分析和输入到符号表；2）注解处理；3）语义分析和生成class文件



1. JVM原理介绍

体系结构：

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



**1.类装载器**

（1） Bootstrap ClassLoader（启动类加载器又称为根加载器/引导类加载器）主要加载JAVA\_HOME/jre/lib里的jar包（jdk下的jar），该目录下的所有jar包都是运行JVM时所必需的jar包。

注意：类加载器其实自身也是一个Java类，因此，自身类加载器需要被其他类加载器进行加载后方可使用，显然必须有一个类加载器的顶级父类(也就是Bootstrap ClassLoader，该类加载器是由C语言代码进行开发的)是其他类加载器的父类。关键点在于，如果一个类的类加载器是Bootstrap ClassLoader，那么该类的getClassLoader()方法返回null。（例如String）

（2） ExtClassLoader主要加载Java核心扩展类，即JAVA\_HOME/jre/lib/ext目录下的jar文件。

（3） AppClassLoader主要加载的是开发者在应用程序中编写的类（自己写的），即 CLASSPATH路径下所有的jar文件。

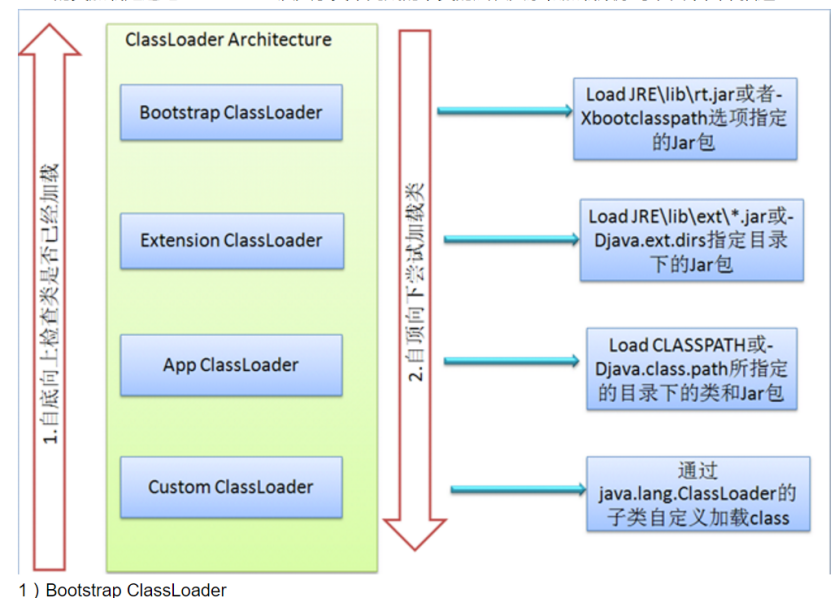
**工作流程**：

（1）当AppClassLoader加载一个class时，它首先不会自己去尝试加载这个类，而是把类加载请求委派给父类加载器ExtClassLoader去完成。

（2）当ExtClassLoader加载一个class时，它首先也不会自己去尝试加载这个类，而是把类加载请求委派给BootStrapClassLoader去完成。

（3）如果BootStrapClassLoader加载失败（例如在$JAVA\_HOME/jre/lib里未查找到该class），会使用ExtClassLoader来尝试加载；

（4）若ExtClassLoader也加载失败，则会使用AppClassLoader来加载5、如果AppClassLoader也加载失败，则会报出异常ClassNotFoundException



**这就是所谓的双亲委派模型**。

简单来说：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类， 而是把请求委托给父加载器去完成，依次向上。

**好处：**

1. 防止内存中出现多份同样的字节码(安全性角度)特别说明（类加载器在成功加载某个类之后，**会把得到的 java.lang.Class类的实例缓存起来。**） **下次再请求加载该类的时候，类加载器会直接使用缓存的类的实例，而不会尝试再次加载。**
2. 可见性机制子类加载器可以看到父类加载器加载的类。
3. 单一性机制因委托机制的关系，一个类（唯一的全限定名）只能被一个类加载器加载一次。

其实前面三点说的都是一件事。

1. 安全性，**避免用户自己编写的类动态替换Java的一些核心类**。如果不采用双亲委派模型的加载方式进行类的加载工作，那我们就可以随时使用自定义的类来动态替代Java核心API中定义的类。例如：如果黑客将“病毒代码”植入到自定义的String类当中，随后类加载器将自定义的String类加载到JVM上，那么此时就会对JVM产生意想不到“病毒攻击”。而双亲委派的这种加载方式就可以避免这种情况，因为String类已经在启动时就被引导类加载器进行了加载。

**双亲委派模型的工作过程如下：**

1. 当前类加载器从自己已经加载的类中查询是否此类已经加载，如果已经加载则返回原来已经加载的类。
2. 如果没有找到，就去委托父类加载器去加载。父类加载器也会采用同样的策略，查看自己已经加载过的类中是否包含这个类，有就返回，没有就委托其父类去加载，直到委托到启动类加载器为止。因为如果父类加载器为空了，就代表使用启动类加载器作为父加载器去加载该类。（也就是看到的String类加载器为null）
3. 如果启动类加载器加载失败，就会使用扩展类加载器来尝试加载，继续失败则会使用AppClassLoader来加载，继续失败就会抛出一个异常ClassNotFoundException。

**类装载呢分为3个过程：**

加载器加载到jvm中，接下来其实又分了好几个步骤：

加载，查找并加载类的二进制数据，在Java堆中也创建一个java.lang.Class类的对象。连接，连接又包含三块内容：验证、准备、初始化。

1. 验证，文件格式、元数据、字节码、符号引用验证；
2. 准备，为类的静态变量分配内存，并将其初始化为默认值；
3. 解析，把类中的符号引用转换为直接引用初始化，为类的静态变量赋予正确的初始值。

**加载、链接、初始化**

1. **加载**

（1）通过一个类的全限定名（包名与类名）来**获取定义此类的二进制字节流**（字节码）（Class文件）。而获取的方式，可以通过jar包、war包、网络中获取、JSP文件生成等方式。

（2）**将这个字节流所代表的静态存储结构转化为方法区的运行时[数据结构](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "http://blog.csdn.net/zhangliangzi/article/details/_blank)**。这里只是转化了数据结构，并未合并数据。（方法区就是用来存放已被加载的类信息，常量，静态变量，编译后的代码的运行时内存区域）

（3）在堆区生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。（反射机制的入口）

1. **链接**
2. **验证**：验证被加载后的类是否有正确的结构，类数据是否会符合虚拟机的要求，确保不会危害虚拟机安全。

**提问：都需要验证哪些东西？**

**··深入理解java虚拟机P178**

**（2）准备**：为类的静态变量（static filed）在方法区分配内存，并赋默认初值（0值或null值）。**如static int a = 100; 静态变量a就会在准备阶段被赋默认值0。**

对于一般的成员变量是在类实例化时候，随对象一起分配在堆内存中。

另外，**静态常量（static final filed）会在准备阶段赋程序设定的初值，如static final int a = 666;**

静态常量a就会在准备阶段被直接赋值为666，对于静态变量，这个操作是在初始化阶段进行的。

1. **解析**：将类的二进制数据中的符号引用换为直接引用。

什么是符号引用？什么是直接引用？

1. **初始化**

类初始化是类加载的最后一步，前面的类加载过程中，除了加载阶段用户可以通过自定义的类加载器参与，其他阶段都完全由虚拟机主导和控制。

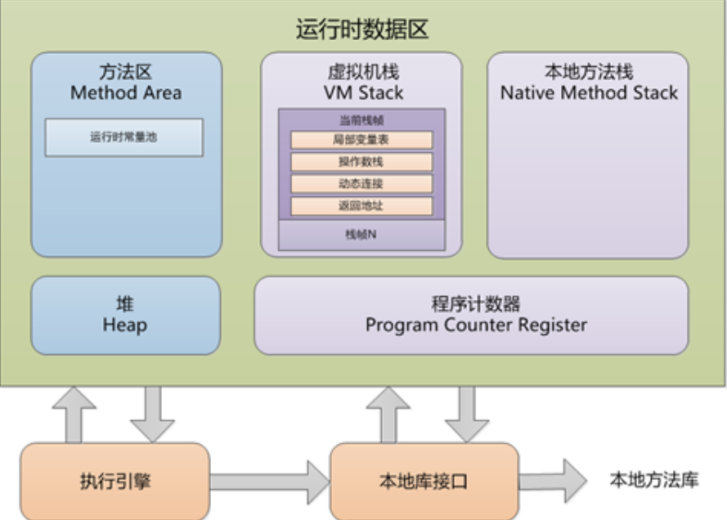
**到了初始化阶段才真正执行Java代码。**

初始化阶段是执行类构造器<clinit>方法的过程，<clinit>方法由类变量的赋值动作和静态语句块按照在源文件出现的顺序合并而成，该合并操作由编译器完成。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **private** **static** **int** value = 100;  **static** **int** a = 100;  **static** **int** b = 100;  **static** **int** c;    **static** {          c = a + b;          System.out.println("it only run once");      } |

1、<clinit>方法对于类或接口不是必须的，如果一个类中没有静态代码块，也没有静态变量的赋值操作，那么编译器不会生成<clinit>；  
2、<clinit>方法与实例构造器不同，不需要显式的调用父类的<clinit>方法，虚拟机会保证父类的<clinit>优先执行；  
3、为了防止多次执行<clinit>，虚拟机会确保<clinit>方法在多线程环境下被正确的加锁同步执行，如果有多个线程同时初始化一个类，那么只有一个线程能够执行<clinit>方法，其它线程进行阻塞等待，直到<clinit>执行完成。  
4、注意：执行接口的<clinit>方法不需要先执行父接口的<clinit>，只有使用父接口中定义的变量时，才会执行。

**运行时数据区域**



**PC 寄存器 线程私有 存放正在执行的字节码地址**

程序计数器是一块较小的区域，它的作用可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的模型里，字节码指示器就是通过改变程序计数器的值来指定下一条需要执行的指令。

**分支、循环、跳转、异常处理、线程恢复**等功能都需要依赖这个计数器来完

主要作用有两个：字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

每一条 Java虚拟机线程都有自己的 PC（ Program Counter）寄存器 。那 PC 寄存器就保存 Java 虚拟机正在执行的字节码指令的地址，如果该方法是 native 的，那 PC 寄存器的值是 undefined。 PC 寄存器的容量至少应当能保存一个 returnAddress 类型的数据或者一个与平台相关的本地指针的值

注意：程序计数器是唯一一个不会出现OutOfMemoryError的内存区域，它的生命周期随着线程的创建而创建，随着线程的结束而死亡。

Java**虚拟机栈**

**存放基本类型的数据和对象的引用，但对象本身不存放在栈中，而是存放在堆中**

局部变量表 主要存放了编译器可知的各种数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference类型，它不同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置）。

描述的是JAVA方法执行的内存模型：每个方法执行的时候都会创建一个栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每个方法的被调用直至执行完成的过程，就对应着一个栈帧在虚拟机中从入栈到出栈的过程。所以也是线程私有的。

是一个线程的执行区域， 它保存着一个线程中的方法的调用状态， 也可以说， 一个Java线程的运行状态， 都由一个Java栈来保存。 在这个栈中， 每一方法对应一个栈帧，请注意区分栈帧和栈这两个概念。栈指的是整个线程的执行栈， 栈帧是栈中的一个单位， 每个方法对应一个栈帧。JVM会对Java栈执行两种操作： 压栈和出栈。 这两种操作在执行时都是以帧（栈帧）为单位的。 当调用了一个新的方法， 就会压入一个栈帧， 当一个方法调用完成， 就会弹出这个方法的栈帧， 回到调用者的栈帧。

**动态链接：**

**方法出口：**

**操作数栈：**

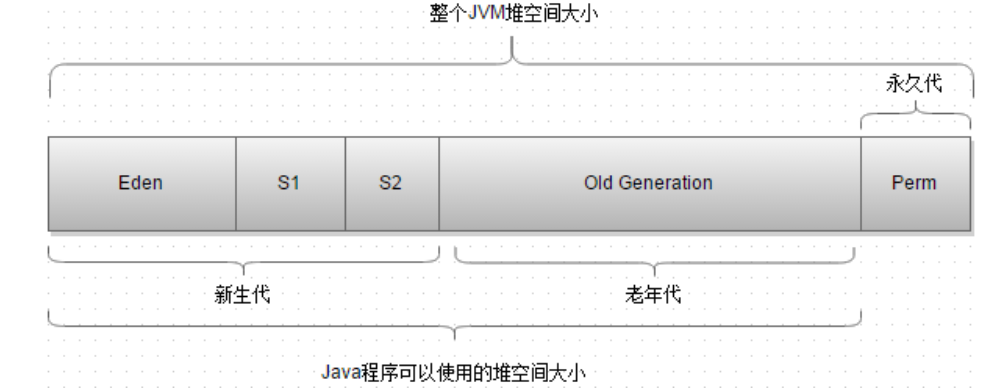
**Java 虚拟机栈会出现两种异常：StackOverFlowError 和 OutOfMemoryError。**StackOverFlowError：

若Java虚拟机栈的内存大小不允许动态扩展， 那么当线程请求栈的深度超过当前Java虚拟机栈的最大深度的时候，就抛出StackOverFlowError异常。OutOfMemoryError：

若Java虚拟机栈的内存大小允许动态扩展，且当线程请求栈时内存用完了， 无法再动态扩展了，此时抛出OutOfMemoryError异常。Java 虚拟机栈也是线程私有的，每个线程都有各自的Java虚拟机栈，而且随着线程的创建而创建，随着线程的死亡而死亡。

**Java 堆**  
 在 Java 虚拟机中，堆（ Heap）是可供各条线程共享的运行时内存区域，也是供所有类实例和数组对象分配内存的区域

垃圾回收器收集的区域因此也被称作是GC堆。



永久带说明：Jdk1.6及之前：常量池分配在永久代 。Jdk1.7：有，但已经逐步“去永久代” 。Jdk1.8及之后：无(java.lang.OutOfMemoryError: PermGen space,这种错误将不会出现在JDK1.8中)。在 JDK 1.8中移除整个永久代，取而代之的是一个叫元空间（Metaspace）的区域（永久代使用的是JVM的堆内存空间，而元空间使用的是物理内存，直接受到本机的物理内存限制）。

**本地方法栈**

和虚拟机栈所发挥的作用非常相似，区别是：虚拟机栈为虚拟机执行 Java 方法 （也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。

本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、操作数栈、动态链接、出口信息。方法执行完毕后相应的栈帧也会出栈并释放内存空间，也会出现 StackOverFlowError 和 OutOfMemoryError 两种异常。

**方法区**

方法区与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。**虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分**，但是它却有一个别名叫做 Non-Heap（非堆），目的应该是与 Java 堆区分开来。

HotSpot 虚拟机中方法区也常被称为 “永久代”，本质上两者并不等价。仅仅是因为 HotSpot 虚拟机设计团队用永久代来实现方法区而已，这样 HotSpot 虚拟机的垃圾收集器就可以像管理 Java 堆一样管理这部分内存了。但是这并不是一个好主意，因为这样更容易遇到内存溢出问题。

相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入方法区后就“永久存在”了。

**运行时常量池**

运行时常量池是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有常量池信息（用于存放编译期生成的各种字面量和符号引用）既然运行时常量池时方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常。

提问常量池与String的关系？

什么是字面量呢？

JDK1.7及之后版本的 JVM 已经将运行时常量池从方法区中移了出来，在 Java 堆（Heap）中开辟了一块区域存放运行时常量池。

NIO的Buffer提供了一个可以不经过JVM内存直接访问系统物理内存的类——DirectBuffer。 DirectBuffer类继承自ByteBuffer，但和普通的ByteBuffer不同，普通的ByteBuffer仍在JVM堆上分配内存，其最大内存受到最大堆内存的限制；而DirectBuffer直接分配在物理内存中，并不占用堆空间，其可申请的最大内存受操作系统限制。直接内存是除Java虚拟机之外的内存,但有可能被Java使用操作直接内存:在NIO中引入了一种基于通道和缓存的IO方式,他可以调用本地方法的直接分配Java虚拟机之外的内存,然后通过一个存储在堆中的DirectByteBuffer对象直接操作该内存,而无需将外部内存中数据复制到堆中再进行操作,从而提高数据操作的效率,直接内存的大小不受Java虚拟机,也会抛出OutOfMemoryError异常直接内存和堆内存比较直接内存申请空间耗费更高的性能直接内存读取IO的性能优于普通的堆内存直接内存的作用链:本地IO-->直接内存-->本地IO堆内存的作用链:本地IO-->直接内存-->非直接内存-->直接内存--->本地IO服务器管理员在配置虚拟机参数时,会根据实际内存设置 -Xmx等参数信息,但经常忽略直接内存,使得各个内存区域总和大于物理内存,从而导致动态扩展时出现OutOFMemoryError