#### 之一：java虚拟机底层结构详解

**Java虚拟机**

Java虚拟机（Java Virtual Machine) 简称JVM Java虚拟机是一个想象中的机器，在实际的计算机上通过软件模拟来实现。Java虚拟机有自己想象中的硬件，如处理器、堆栈、寄存器等，还具有相应的指令系统。下面我们就来看一下这几部分比较重要的java虚拟机的结构

**JVM寄存器**

所有的CPU均包含用于保存系统状态和处理器所需信息的寄存器组。如果虚拟机定义义较多的寄存器，便可以从中得到更多的信息而不必对栈或内存进行访问，这有利于提高运行速度。然而，如果虚拟机中的寄存器比实际CPU的寄存器多，在实现虚拟机时就会占用处理器大量的时间来用常规存储器模拟寄存器，这反而会降低虚拟机的效率。针对这种情况，JVM只设置了4个最为常用的寄存器。它们是：pc程序计数器，optop操作数栈顶指针 ，frame当前执行环境指针， vars指向当前执行环境中第一个局部变量的指针， 所有寄存器均为32位。pc用于记录程序的执行。optop,frame和vars用于记录指向Java栈区的指针。

**JVM栈结构**

作为基于栈结构的计算机，Java栈是JVM存储信息的主要方法。当JVM得到一个java字节码应用程序后，便为该代码中一个类的每一个方法创建一个栈框架，以保存该方法的状态信息。每个栈框架包括以下三类信息：局部变量执行环境操作数栈 局部变量用于存储一个类的方法中所用到的局部变量。vars寄存器指向该变量表中的第一个局部变量。执行环境用于保存解释器对Java字节码进行解释过程中所需的信息。它们是：上次调用的方法、局部变量指针和操作数栈的栈顶和栈底指针。执行环境是一个执行一个方法的控制中心。例如：如果解释器要执行iadd（整数加法），首先要从frame寄存器中找到当前执行环境，而后便从执行环境中找到操作数栈，从栈顶弹出两个整数进行加法运算，最后将结果压入栈顶。　　操作数栈用于存储运算所需操作数及运算的结果。

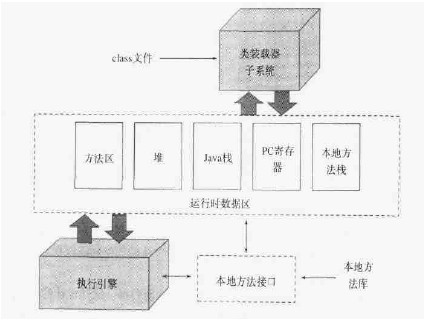
**JVM碎片回收堆**

Java类的实例所需的存储空间是在堆上分配的。解释器具体承担为类实例分配空间的工作。解释器在为一个实例分配完存储空间后，便开始记录对该实例所占用的内存区域的使用。一旦对象使用完毕，便将其回收到堆中。在Java语言中，除了new语句外没有其他方法为一对象申请和释放内存。对内存进行释放和回收的工作是由Java运行系统承担的。这允许Java运行系统的设计者自己决定碎片回收的方法。在SUN公司开发的Java解释器和Hot Java环境中，碎片回收用后台线程的方式来执行。这不但为运行系统提供了良好的性能，而且使程序设计人员摆脱了自己控制内存使用的风险。

**JVM存储区**

　　JVM有两类存储区：常量缓冲池和方法区。常量缓冲池用于存储类名称、方法和字段名称以及串常量。方法区则用于存储Java方法的字节码。对于这两种存储区域具体实现方式在JVM规格中没有明确规定。这使得Java应用程序的存储布局必须在运行过程中确定，依赖于具体平台的实现方式。JVM是为Java字节码定义的一种独立于具体平台的规格描述，是Java平台独立性的基础。目前的JVM还存在一些限制和不足，有待于进一步的完善，但无论如何，JVM的思想是成功的。对比分析：如果把Java原程序想象成我们的C++原程序，Java原程序编译后生成的字节码就相当于C++原程序编译后的80x86的机器码（二进制程序文件），JVM虚拟机相当于80x86计算机系统,Java解释器相当于80x86CPU。在80x86CPU上运行的是机器码，在Java解释器上运行的是Java字节码。　　Java解释器相当于运行Java字节码的“CPU”，但该“CPU”不是通过硬件实现的，而是用软件实现的。Java解释器实际上就是特定的平台下的一个应用程序。只要实现了特定平台下的解释器程序，Java字节码就能通过解释器程序在该平台下运行，这是Java跨平台的根本。当前，并不是在所有的平台下都有相应Java解释器程序，这也是Java并不能在所有的平台下都能运行的原因，它只能在已实现了Java解释器程序的平台下运行。

**Java虚拟机的体系结构图**



Java虚拟机从启动到结束的生命周期，当java虚拟机启动后，在如下几种情况下，Java虚拟机将结束生命周期：

1.执行了System.exit()方法

2.程序正常执行结束

3.程序在执行过程中遇到了异常或错误而异常终止

4.由于操作系统出现错误而导致Java虚拟机进程终止

**Java虚拟机的栈有三个区域**:局部变量区、运行环境区、操作数区。

**局部变量区**

每个Java方法使用一个固定大小的局部变量集。它们按照与vars寄存器的字偏移量来寻址。局部变量都是32位的。长整数和双精度浮点数占据了两个局部变量的空间,却按照第一个局部变量的索引来寻址。(例如,一个具有索引n的局部变量,如果是一个双精度浮点数,那么它实际占据了索引n和n+1所代表的存储空间)虚拟机规范并不要求在局部变量中的64位的值是64位对齐的。虚拟机提供了把局部变量中的值装载到操作数栈的指令,也提供了把操作数栈中的值写入局部变量的指令。

**运行环境区**

在运行环境中包含的信息用于动态链接,正常的方法返回以及异常捕捉。

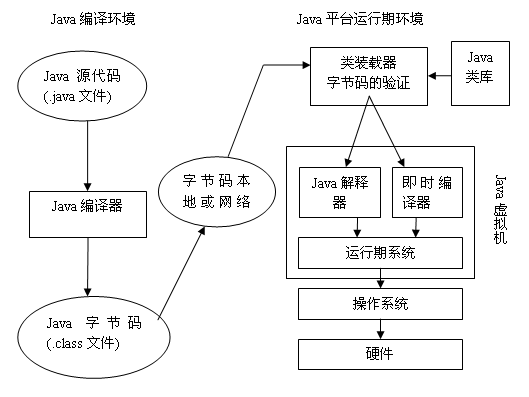
**操作数栈区**

机器指令只从操作数栈中取操作数,对它们进行操作,并把结果返回到栈中。选择栈结构的原因是:在只有少量寄存器或非通用寄存器的机器(如Intel486)上,也能够高效地模拟虚拟机的行为。操作数栈是32位的。它用于给方法传递参数,并从方法接收结果,也用于支持操作的参数,并保存操作的结果。例如,iadd指令将两个整数相加。相加的两个整数应该是操作数栈顶的两个字。这两个字是由先前的指令压进堆栈的。这两个整数将从堆栈弹出、相加,并把结果压回到操作数栈中。

每个原始数据类型都有专门的指令对它们进行必须的操作。每个操作数在栈中需要一个存储位置,除了long和double型,它们需要两个位置。操作数只能被适用于其类型的操作符所操作。例如,压入两个int类型的数,如果把它们当作是一个long类型的数则是非法的。在Sun的虚拟机实现中,这个限制由字节码验证器强制实行。但是,有少数操作(操作符dupe和swap),用于对运行时数据区进行操作时是不考虑类型的。

本地方法栈，当一个线程调用本地方法时，它就不再受到虚拟机关于结构和安全限制方面的约束，它既可以访问虚拟机的运行期数据区，也可以使用本地处理器以及任何类型的栈。例如，本地栈是一个C语言的栈，那么当C程序调用C函数时，函数的参数以某种顺序被压入栈，结果则返回给调用函数。在实现Java虚拟机时，本地方法接口使用的是C语言的模型栈，那么它的本地方法栈的调度与使用则完全与C语言的栈相同。

下图可以表示出来java程序运行的一个全过程



3  Java虚拟机的运行过程

上面对虚拟机的各个部分进行了比较详细的说明，下面通过一个具体的例子来分析它的运行过程。

虚拟机通过调用某个指定类的方法main启动，传递给main一个字符串数组参数，使指定的类被装载，同时链接该类所使用的其它的类型，并且初始化它们。例如对于程序：

class HelloApp

{

   public static void main(String[] args)

   {

       System.out.println("Hello World!");

       for (int i = 0; i < args.length; i++ )

       {

           System.out.println(args[i]);

       }

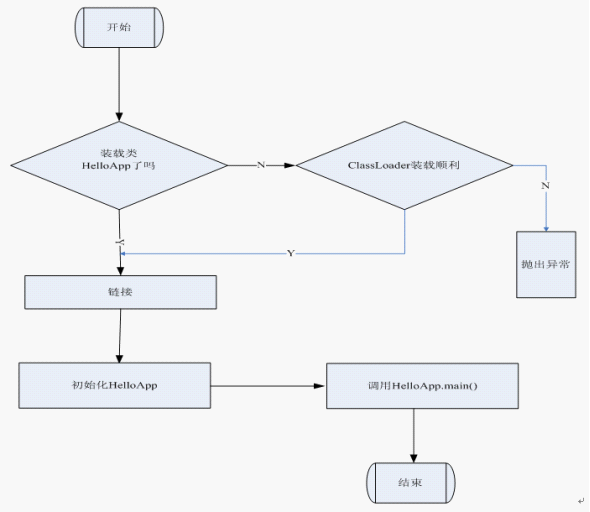
   }

}

编译后在命令行模式下键入： java HelloApp run virtual machine

将通过调用HelloApp的方法main来启动java虚拟机，传递给main一个包含三个字符串"run"、"virtual"、"machine"的数组。现在我们略述虚拟机在执行HelloApp时可能采取的步骤。

开始试图执行类HelloApp的main方法，发现该类并没有被装载，也就是说虚拟机当前不包含该类的二进制代表，于是虚拟机使用ClassLoader试图寻找这样的二进制代表。如果这个进程失败，则抛出一个异常。类被装载后同时在main方法被调用之前，必须对类HelloApp与其它类型进行链接然后初始化。链接包含三个阶段：检验，准备和解析。检验检查被装载的主类的符号和语义，准备则创建类或接口的静态域以及把这些域初始化为标准的默认值，解析负责检查主类对其它类或接口的符号引用，在这一步它是可选的。类的初始化是对类中声明的静态初始化函数和静态域的初始化构造方法的执行。一个类在初始化之前它的父类必须被初始化。整个过程如下：



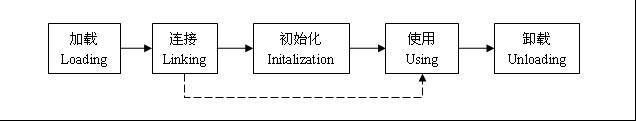
类加载器，顾名思义，类加载器（class loader）用来加载 Java 类到 Java 虚拟机中。一般来说，Java 虚拟机使用 Java 类的方式如下：Java 源程序（.java 文件）在经过 Java 编译器编译之后就被转换成 Java 字节代码（.class 文件）。类加载器负责读取 Java 字节代码，并转换成 java.lang.Class类的一个实例。每个这样的实例用来表示一个 Java 类。通过此实例的 newInstance()方法就可以创建出该类的一个对象。实际的情况可能更加复杂，比如 Java 字节代码可能是通过工具动态生成的，也可能是通过网络下载的。基本上所有的类加载器都是 java.lang.ClassLoader类的一个实例。其实我们研究类加载器主要研究的就是类的生命周期

     首先来了解一下jvm（java虚拟机）中的几个比较重要的内存区域，这几个区域在java类的生命周期中扮演着比较重要的角色：

**方法区：**在java的虚拟机中有一块专门用来存放已经加载的类信息、常量、静态变量以及方法代码的内存区域，叫做方法区。

**常量池：**常量池是方法区的一部分，主要用来存放常量和类中的符号引用等信息。

**堆区：**用于存放类的对象实例。

**栈区：**也叫java虚拟机栈，是由一个一个的栈帧组成的后进先出的栈式结构，栈桢中存放方法运行时产生的局部变量、方法出口等信息。当调用一个方法时，虚拟机栈中就会创建一个栈帧存放这些数据，当方法调用完成时，栈帧消失，如果方法中调用了其他方法，则继续在栈顶创建新的栈桢。  
  
**类的生命周期**  
       当我们编写一个java的源文件后，经过编译会生成一个后缀名为class的文件，这种文件叫做字节码文件，只有这种字节码文件才能够在java虚拟机中运行，java类的生命周期就是指一个class文件从加载到卸载的全过程。一个java类的完整的生命周期会经历加载、连接、初始化、使用、和卸载五个阶段，当然也有在加载或者连接之后没有被初始化就直接被使用的情况，这里我们主要来研究类加载器所执行的部分，也就是加载，链接和初始化。如图所示：  
                   
  
下面我先简单看一下类加载器所执行的三部分的简单介绍  
1、加载：查找并加载类的二进制数据

2、连接

   –验证：确保被加载的类的正确性

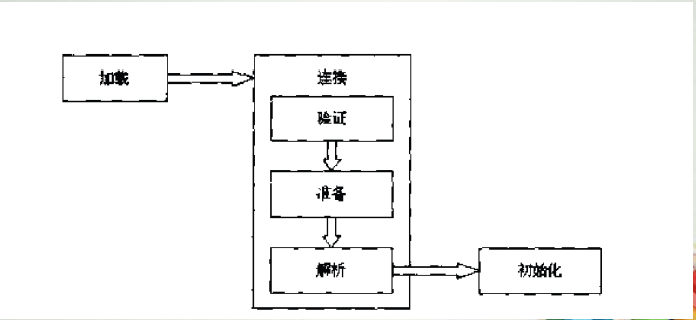
   –准备：为类的静态变量分配内存，并将其初始化为默认值

   –解析：把类中的符号引用转换为直接引用

3、初始化：为类的静态变量赋予正确的初始值

    从上边我们可以看出类的静态变量赋了两回值。这是为什么呢？原因是，在连接过程中时为静态变量赋值为默认值，也就是说，只要是你定义了静态变量，不管你开始给没给它设置，我系统都为他初始化一个默认值。到了初始化过程，系统就检查是否用户定义静态变量时有没有给设置初始化值，如果有就把静态变量设置为用户自己设置的初始化值，如果没有还是让静态变量为初始化值

类的加载、连接和初始化



#### 之二：类的生命周期（上）类的加载和连接

**类的加载**

    类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构 。这里的class对象其实就像一面镜子一样，外面是类的源程序，里面是class对象，它实时的反应了类的数据结构和信息。

**加载.class文件的方式**

1、从本地系统中直接加载

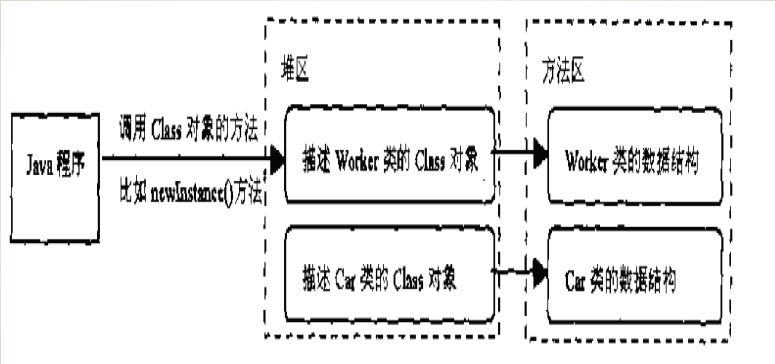
2、通过网络下载.class文件

3、从zip，jar等归档文件中加载.class文件

4、从专有数据库中提取.class文件

5、将Java源文件动态编译为.class文件

**类的加载过程**



**结论：**

1、类的加载的最终产品是位于堆区中的Class对象

2、Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口

Java虚拟机给我们提供了两种类加载器：

1、Java虚拟机自带的加载器

1）根类加载器（使用C++编写，程序员无法在Java代码中获得该类）

2）扩展加载器，使用Java代码实现

3）系统加载器（应用加载器），使用Java代码实现

2、用户自定义的类加载器

    java.lang.ClassLoader的子类

    用户可以定制类的加载方式

我们看一下API对ClassLoader的介绍：

  类加载器是负责加载类的对象。ClassLoader 类是一个抽象类。如果给定类的二进制名称，那么类加载器会试图查找或生成构成类定义的数据。一般策略是将名称转换为某个文件名，然后从文件系统读取该名称的“类文件”。每个class对象都包含一个对定义它的 ClassLoader 的引用。

我们再来看一下Class类的一个方法getClassLoader

public ClassLoader getClassLoader()

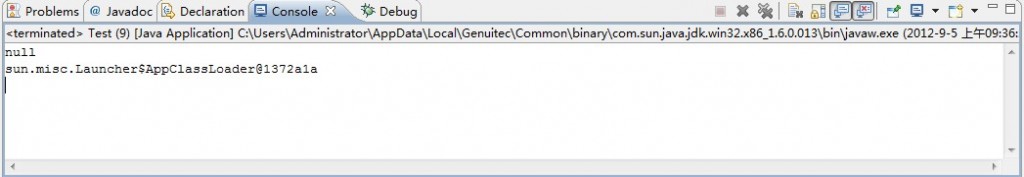
返回该类的类加载器。有些实现可能使用 null 来表示根类加载器。如果该类由根类加载器加载，则此方法在这类实现中将返回 null。

下面我们来看一个小例子来验证一下：

**[java]** view plaincopyprint?

1. **package** com.bzu.csh;
2. **public** **class** Test
3. {
4. **public** **static** **void** main(String[] args) **throws** Exception
5. {
6. Class clazz = Class.forName("java.lang.String");
7. System.out.println(clazz.getClassLoader());
8. Class clazz2 = Class.forName("com.bzu.csh.ABC");
9. System.out.println(clazz2.getClassLoader());
10. }
11. }
12. **class** ABC
13. {
14. }

看一下打印结果，一目了然：



从上面打印结果可以看出，第一个为null，也就是它用根类加载器加载的，第二个是我们自己写的类，也就是说，我们自己写的那个类用sun.misc.Launcher$AppClassLoader@1372a1a加载器加载的，我们可以看到APP，也就是应用类加载器，也就是系统加载器

还记得我们以前用过的动态代理吧，InvocationHandler，当我们利用proxy对象调用newProxyInstance建立一个代理类时，我们要给他传一个ClassLoader，也就是类加载器，如下：

**[java]** view plaincopyprint?

1. **public** **static** Object newProxyInstance(ClassLoader loader,
2. Class<?>[] interfaces,
3. InvocationHandler h)
4. **throws** IllegalArgumentException

       当时我们学习的时候，只知道这里的loader随便给他设置一个类的类加载器就可以。现在我们来想想为什么这里需要一个类加载器呢？我们知道这个newProxyInstance是动态的给我们生成一个代理类，然后根据这个代理类生成一个代理对象。动态生成这个代理类之后我们不得把他加载到内存里吗，加载到内存里我们才可以用他。用什么加载到内存里，只有类加载器，所以我们要给他指定一个类加载器。

     类加载器并不需要等到某个类被“首次主动使用”时再加载它 。JVM规范允许类加载器在预料某个类将要被使用时就预先加载它，如果在预先加载的过程中遇到了.class文件缺失或存在错误，类加载器必须在程序首次主动使用该类时才报告错误（LinkageError错误） 如果这个类一直没有被程序主动使用，那么类加载器就不会报告错误 。大家在做web开发的时候有可能会出现这种问题，比如我们在做测试的时候是用的jdk1.6，而我们在部署的时候我们用的是jdk1.5.这时候就很可能汇报LinkageError错误，版本不兼容。

**类的连接**

类被加载后，就进入连接阶段。连接就是将已经读入到内存的类的二进制数据合并到虚拟机的运行时环境中去。

**验证**：当一个类被加载之后，必须要验证一下这个类是否合法，比如这个类是不是符合字节码的格式、变量与方法是不是有重复、数据类型是不是有效、继承与实现是否合乎标准等等。总之，这个阶段的目的就是保证加载的类是能够被jvm所运行。很多人都感觉，既然这个类都通过编译加载到内存里了，那肯定就是合法的了，为什么还要验证呢，这是因为这里的验证时为了避免有人恶意编写class文件，也就是说并不是通过编译得到的class文件。所以这里验证其实是检查的class文件的内部结构是否符合字节码的要求

**准备**：准备阶段的工作就是为类的静态变量分配内存并设为jvm默认的初值，对于非静态的变量，则不会为它们分配内存。有一点需要注意，这时候，静态变量的初值为jvm默认的初值，而不是我们在程序中设定的初值。jvm默认的初值是这样的：  
基本类型（int、long、short、char、byte、boolean、float、double）的默认值为0。  
引用类型的默认值为null。  
常量的默认值为我们程序中设定的值，比如我们在程序中定义final static int a = 100，则准备阶段中a的初值就是100。

**解析：**这一阶段的任务就是把常量池中的符号引用转换为直接引用。那么什么是符号引用，什么又是直接引用呢？我们来举个例子：我们要找一个人，我们现有的信息是这个人的身份证号是1234567890。只有这个信息我们显然找不到这个人，但是通过公安局的身份系统，我们输入1234567890这个号之后，就会得到它的全部信息：比如山东省滨州市滨城区18号张三，通过这个信息我们就能找到这个人了。这里，123456790就好比是一个符号引用，而山东省滨州市滨城区18号张三就是直接引用。在内存中也是一样，比如我们要在内存中找一个类里面的一个叫做show的方法，显然是找不到。但是在解析阶段，jvm就会把show这个名字转换为指向方法区的的一块内存地址，比如c17164，通过c17164就可以找到show这个方法具体分配在内存的哪一个区域了。这里show就是符号引用，而c17164就是直接引用。在解析阶段，jvm会将所有的类或接口名、字段名、方法名转换为具体的内存地址。

#### 之三：类的生命周期(下）类的初始化

**类的初始化**:在类的生命周期执行完加载和连接之后就开始了类的初始化。在类的初始化阶段，java虚拟机执行类的初始化语句，为类的静态变量赋值，在程序中，类的初始化有两种途径：（1）在变量的声明处赋值。（2）在静态代码块处赋值，比如下面的代码，a就是第一种初始化，b就是第二种初始化

**[html]** view plaincopyprint?

1. public class Test
2. {
3. public static int a = 0;
4. public static int b ;
5. static{
6. b=2;
7. }
8. }

     静态变量的声明和静态代码块的初始化都可以看做静态变量的初始化，类的静态变量的初始化是有顺序的。顺序为类文件从上到下进行初始化，想到这，想起来**一个很无耻的面试题**，分享给大家看一下：

**[java]** view plaincopyprint?

1. **package** com.bzu.csh;
2. **class** Singleton
3. {
4. **private** **static** Singleton singleton = **new** Singleton();
5. **public** **static** **int** counter1;
6. **public** **static** **int** counter2 = 0;
7. **private** Singleton()
8. {
9. counter1++;
10. counter2++;
11. }
12. **public** **static** Singleton getInstance()
13. {
14. **return** singleton;
15. }
16. }
17. **public** **class** Test
18. {
19. **public** **static** **void** main(String[] args)
20. {
21. Singleton singleton = Singleton.getInstance();
22. System.out.println("counter1 = " + singleton.counter1);
23. System.out.println("counter2 = " + singleton.counter2);
24. }
25. }

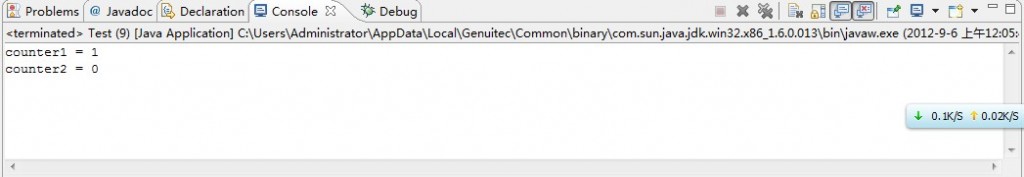
**大家先看看这里的程序会输出什么？**

不知道大家的答案是什么，如果不介意的话可以把你的答案写到评论上，看看有多少人的答案和你一样的。我先说说我刚开始的答案吧。我认为会输出：

counter1 = 1

Counter2 = 1

不知道大家的答案是不是这个，反正我的是。下面我们来看一下正确答案：



不知道你做对没有，反正我刚开始做错了。好，现在我来解释一下为什么会是这个答案。在给出解释之前，我们先来看一个概念：

Java程序对类的使用方式可分为两种

主动使用

被动使用

•所有的Java虚拟机实现必须在每个类或接口被Java程序“**首次主动使用**”时才初始化他们

**主动使用（六种）**

–创建类的实例

–访问某个类或接口的静态变量，或者对该静态变量赋值

–调用类的静态方法

–反射（如Class.forName(“com.bzu.csh.Test”)）

–初始化一个类的子类

–Java虚拟机启动时被标明为启动类的类（Java Test）

       OK，我们开始解释一下上面的答案，程序开始运行，首先执行main方法，执行main方法第一条语句，调用Singleton类的静态方法，这里调用Singleton类的静态方法就是主动使用Singleton类。所以开始加载Singleton类。在加载Singleton类的过程中，首先对静态变量赋值为默认值，

Singleton=null

counter1 = 0

Counter2 = 0

给他们赋值完默认值值之后，要进行的就是对静态变量初始化，对声明时已经赋值的变量进行初始化。我们上面提到过，初始化是从类文件从上到下赋值的。所以首先给Singleton赋值，给它赋值，就要执行它的构造方法，然后执行counter1++;counter2++;所以这里的counter1 = 1;counter2 = 1;执行完这个初始化之后，然后执行counter2的初始化，我们声明的时候给他初始化为0 了，所以counter2 的值又变为了0.初始化完之后执行输出。所以这是的

counter1 = 1

counter2 = 0

**类初始化步骤**

（1）假如一个类还没有被加载或者连接，那就先加载和连接这个类

（2）假如类存在直接的父类，并且这个父类还没有被初始化，那就先初始化直接的父类

（3）假如类中存在初始化语句，那就直接按顺序执行这些初始化语句

      在上边我们我们说了java虚拟机实现必须在每个类或接口被Java程序“**首次主动使用**”时才初始化他们，上面也举出了六种主动使用的说明。除了上述六种情形，其他使用Java类的方式都被看作是被动使用，不会导致类的初始化。程序中对子类的“主动使用”会导致父类被初始化；但对父类的“主动”使用并不会导致子类初始化（不可能说生成一个Object类的对象就导致系统中所有的子类都会被初始化）

**注：**调用ClassLoader类的loadClass方法加载一个类，并不是对类的主动使用，不会导致类的初始化。

当java虚拟机初始化一个类时，要求它的所有的父类都已经被初始化，但这条规则并不适用于接口。

      在初始化一个类时，并不会先初始化它所实现的接口

      在初始化一个接口时，并不会先初始化它的父接口

     因此，一个父接口并不会因为它的子接口或者实现类的初始化而初始化。只有当程序首次使用特定接口的静态变量时，才会导致该接口的初始化。只有当程序访问的静态变量或静态方法确实在当前类或当前接口中定义时，才可以认为是对类或接口的主动使用 。如果是调用的子类的父类属性，那么子类不会被初始化。

## 之四：java虚拟机的垃圾回收机制

Java语言从出现到现在，一直占据编程语言前列，他很大的一个原因就是由于java应用程序所运行的平台有关。我们大家都知道java应用程序运行在java虚拟机上。这样就大大减少了java应用程序和底层操作系统打交道的频率。这也就为java程序的跨平台提供了良好的基础。在java虚拟机中为我们提供了一个很重要的机制就是java虚拟机的自动的内存管理机制。也就是我们平时所说的垃圾回收机制，这使得开发人员不用自己来管理应用中的内存。C/C++开发人员需要通过malloc/free 和new/delete等函数来显式的分配和释放内存。这对开发人员提出了比较高的要求，容易造成内存访问错误和内存泄露等问题。今天我们就一起来看一下java虚拟机给我们提供的这个强大的功能——自动垃圾回收机制。

     我们在c/c++的程序中，他们没有java中的自动垃圾回收机制，这就需要开发人员手动的去分配和释放内存，这样就要求我们的开发人员要有一定的细心和对内存管理的经验。如果内存管理不好，很容易产生最常见的两个问题。一是“悬挂引用”，二是内存溢出。所为的悬挂引用就是一个对象引用所指向的内存区块已经被错误的回收并重新分配给新的对象了，程序如果继续使用这个引用的话会造成不可预期的结果。第二个内存溢出就很好理解了，开发人员在做开发的过程中，只显示的申请内存而忘记用完释放掉内存，这样长时间会导致内存溢出的情况。而像java这种具有自动管理内存机制的语言来说，我们开发人员只需考虑引用的运用就可以，把内存管理这块交给我们的语言运行环境来管理。。开发人员并不需要关心内存的分配和回收的底层细节。Java平台通过垃圾回收器来进行自动的内存管理。这样就大大减少了开发人员的工作量

**一、Java垃圾回收机制**

Java 的垃圾回收器要负责完成3 件任务：

1.分配内存

2.确保被引用的对象的内存不被错误回收

3.回收不再被引用的对象的内存空间。

       垃圾回收是一个复杂而且耗时的操作。如果JVM 花费过多的时间在垃圾回收上，则势必会影响应用的运行性能。一般情况下，当垃圾回收器在进行回收操作的时候，整个应用的执行是被暂时中止（stop-the-world）的。这是因为垃圾回收器需要更新应用中所有对象引用的实际内存地址。不同的硬件平台所能支持的垃圾回收方式也不同。比如在多CPU 的平台上，就可以通过并行的方式来回收垃圾。而单CPU 平台则只能串行进行。不同的应用所期望的垃圾回收方式也会有所不同。服务器端应用可能希望在应用的整个运行时间中，花在垃圾回收上的时间总数越小越好。而对于与用户交互的应用来说，则可能希望所垃圾回收所带来的应用停顿的时间间隔越小越好。对于这种情况，JVM 中提供了多种垃圾回收方法以及对应的性能调优参数，应用可以根据需要来进行定制。

**二、判断对象是否该被回收算法**

**1.引用计数算法**

给对象添加一个引用计数器，每当有一个地方引用它时，计数器值就加1，当引用失效时，计数器值就减1；任何时刻计数器值都为0时对象就表示它不可能被使用了。这个算法实现简单，但很难解决对象之间循环引用的问题，因此Java并没有用这种算法！这是很多人都误解了的地方。

**2.根搜索算法**

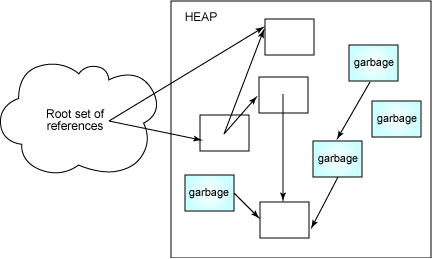
通过一系列名为“GC ROOT”的对象作为起始点，从这些结点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到GC ROOT没有任何引用链相连时，则证明这个对象是不可用的。如果对象在进行根搜索后发现没有与GC ROOT相连接的引用链，则会被第一次第标记，并看此对象是否需要执行finalize()方法（忘记finalize()这个方法吧，它可以被try-finally或其他方式代替的），当第二次被标记时，对象就会被回收。

**三、Java虚拟机基本垃圾回收算法：**

**1.标记-清除（Mark-Sweep）**

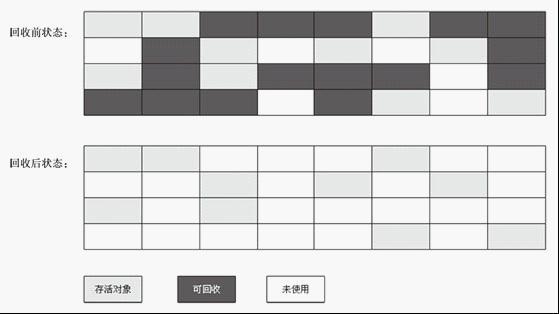
此算法执行分两阶段。第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。它停止所有工作，收集器从根开始访问每一个活跃的节点，标记它所访问的每一个节点。走过所有引用后，收集就完成了，然后就对堆进行清除（即对堆中的每一个对象进行检查），所有没有标记的对象都作为垃圾回收并返回空闲列表。下图 展示了垃圾收集之前的堆，阴影块是垃圾，因为用户程序不能到达它们：

**可到达和不可到达的对象**

****

标记-清除实现起来很简单，可以容易地回收循环的结构，并且不像引用计数那样增加编译器或者赋值函数的负担。但是它也有不足 ―― 收集暂停可能会很长，在清除阶段整个堆都是可访问的，这对于可能有页面交换的堆的虚拟内存系统有非常负面的性能影响。

标记-清除的最大问题是，每一个活跃的（即已分配的）对象，不管是不是可到达的，在清除阶段都是可以访问的。因为很多对象都可能成为垃圾，这意思着收集器花费大量精力去检查并处理垃圾。标记-清除收集器还容易使堆产生碎片，这会产生区域性问题并可以造成分配失败，即使看来有足够的自由内存可用。此算法需要暂停整个应用，同时，会产生内存碎片。



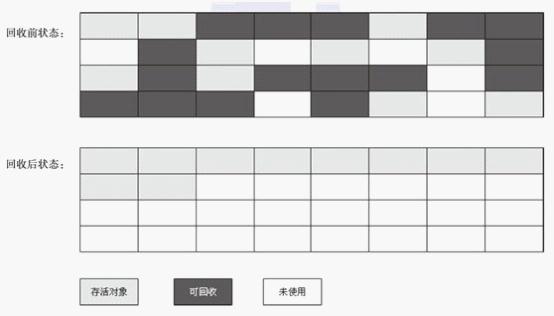
**2.复制（Copying）**

此算法把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。次算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不过出现“碎片”问题。当然，此算法的缺点也是很明显的，就是需要两倍内存空间。

****

**3.标记-整理（Mark-Compact）**

此算法结合了“标记-清除”和“复制”两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。



**4.增量收集（Incremental Collecting）**

实施垃圾回收算法，即：在应用进行的同时进行垃圾回收。不知道什么原因JDK5.0中的收集器没有使用这种算法的。

**5.分代（Generational Collecting）**

将堆分成新生代（Eden, From Survivor, To Survivor）和老年代，在新生代中使用复制算法，即Minor-GC，当一些对象经过多次的Minor-GC后还留在新生代，则会被搬移到老年代中。而老年代中使用标记-清理或标记-整理算法，即Major GC/Full GC。

-XX:PretenurseSizeThreshold=1024，则大于次参数的对象会直接分配到老年代（尽可能不要写一些“短命大对象”！）

-XX:MaxTenuringThreshold=15，在survivor空间存活15次之后，则会搬移到老年代

如果是Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代。

进行Minor GC时，虚拟机会检测之前每次晋升到老年代的平均大小是否大于老年代的剩余空间大小，如果大于，则直接进行一次Full GC。

在对垃圾收集算法进行评价时，我们可能要考虑以下所有标准：

· 暂停时间。收集器是否停止所有工作来进行垃圾收集？要停止多长时间？暂停是否有时间限制？

· 暂停的可预测性。垃圾收集暂停是否规划为在用户程序方便而不是垃圾收集器方便的时间发生？

· CPU 占用。总的可用 CPU 时间用在垃圾收集上的百分比是多少？

· 内存大小。许多垃圾收集算法需要将堆分割成独立的内存空间，其中一些空间在某些时刻对用户程序是不可访问的。这意味着堆的实际大小可能比用户程序的最大堆驻留空间要大几倍。

· 虚拟内存交互。在具有有限物理内存的系统上，一个完整的垃圾收集在垃圾收集过程中可能会错误地将非常驻页面放到内存中来进行检查。因为页面错误的成本很高，所以垃圾收集器正确管理引用的区域性 (locality) 是很必要的。

· 缓存交互。即使在整个堆可以放到主内存中的系统上 ―― 实际上几乎所有 Java 应用程序都可以做到这一点，垃圾收集也常常会有将用户程序使用的数据冲出缓存的效果，从而影响用户程序的性能。

· 对程序区域性的影响。虽然一些人认为垃圾收集器的工作只是收回不可到达的内存，但是其他人认为垃圾收集器还应该尽量改进用户程序的引用区域性。整理收集器和复制收集器在收集过程中重新安排对象，这有可能改进区域性。

· 编译器和运行时影响。一些垃圾收集算法要求编译器或者运行时环境的重要配合，如当进行指针分配时更新引用计数。这增加了编译器的工作，因为它必须生成这些簿记指令，同时增加了运行时环境的开销，因为它必须执行这些额外的指令。这些要求对性能有什么影响呢？它是否会干扰编译时优化呢？

         不管选择什么算法，硬件和软件的发展使垃圾收集更具有实用性。20 世纪 70 和 80 年代的经验研究表明，对于大型 Lisp 程序，垃圾收集消耗 25％ 到 40％ 的运行时。垃圾收集还不能做到完全不可见，这肯定还有很长的路要走。

**四、三种垃圾回收器**

目前的收集器主要有三种：串行收集器、并行收集器、并发收集器。

**1.串行收集器**

使用单线程处理所有垃圾回收工作，因为无需多线程交互，所以效率比较高。但是，也无法使用多处理器的优势，所以此收集器适合单处理器机器。当然，此收集器也可以用在小数据量（100M左右）情况下的多处理器机器上。可以使用-XX:+UseSerialGC打开。

**2.并行收集器**

1）对年轻代进行并行垃圾回收，因此可以减少垃圾回收时间。一般在多线程多处理器机器上使用。使用-XX:+UseParallelGC.打开。并行收集器在J2SE5.0第六6更新上引入，在Java SE6.0中进行了增强--可以堆年老代进行并行收集。如果年老代不使用并发收集的话，是使用单线程进行垃圾回收，因此会制约扩展能力。使用-XX:+UseParallelOldGC打开。

2）使用-XX:ParallelGCThreads=<N>设置并行垃圾回收的线程数。此值可以设置与机器处理器数量相等。

3）此收集器可以进行如下配置：

最大垃圾回收暂停:指定垃圾回收时的最长暂停时间，通过-XX:MaxGCPauseMillis=<N>指定。<N>为毫秒.如果指定了此值的话，堆大小和垃圾回收相关参数会进行调整以达到指定值。设定此值可能会减少应用的吞吐量。

 吞吐量:吞吐量为垃圾回收时间与非垃圾回收时间的比值，通过-XX:GCTimeRatio=<N>来设定，公式为1/（1+N）。例如，-XX:GCTimeRatio=19时，表示5%的时间用于垃圾回收。默认情况为99，即1%的时间用于垃圾回收。

**3.并发收集器**

可以保证大部分工作都并发进行（应用不停止），垃圾回收只暂停很少的时间，此收集器适合对响应时间要求比较高的中、大规模应用。使用-XX:+UseConcMarkSweepGC打开。

1）并发收集器主要减少年老代的暂停时间，他在应用不停止的情况下使用独立的垃圾回收线程，跟踪可达对象。在每个年老代垃圾回收周期中，在收集初期并发收集器会对整个应用进行简短的暂停，在收集中还会再暂停一次。第二次暂停会比第一次稍长，在此过程中多个线程同时进行垃圾回收工作。

2）并发收集器使用处理器换来短暂的停顿时间。在一个N个处理器的系统上，并发收集部分使用K/N个可用处理器进行回收，一般情况下1<=K<=N/4。

3）在只有一个处理器的主机上使用并发收集器，设置为incremental mode模式也可获得较短的停顿时间。

4）浮动垃圾：由于在应用运行的同时进行垃圾回收，所以有些垃圾可能在垃圾回收进行完成时产生，这样就造成了“Floating Garbage”，这些垃圾需要在下次垃圾回收周期时才能回收掉。所以，并发收集器一般需要20%的预留空间用于这些浮动垃圾。

5）Concurrent Mode Failure：并发收集器在应用运行时进行收集，所以需要保证堆在垃圾回收的这段时间有足够的空间供程序使用，否则，垃圾回收还未完成，堆空间先满了。这种情况下将会发生“并发模式失败”，此时整个应用将会暂停，进行垃圾回收。

6）启动并发收集器：因为并发收集在应用运行时进行收集，所以必须保证收集完成之前有足够的内存空间供程序使用，否则会出现“Concurrent Mode Failure”。通过设置-XX:CMSInitiatingOccupancyFraction=<N>指定还有多少剩余堆时开始执行并发收集

**五、关于垃圾收集的几点补充**

经过上述的说明，可以发现垃圾回收有以下的几个特点：

（1）垃圾收集发生的不可预知性：由于实现了不同的垃圾收集算法和采用了不同的收集机制，所以它有可能是定时发生，有可能是当出现系统空闲CPU资源时发生，也有可能是和原始的垃圾收集一样，等到内存消耗出现极限时发生，这与垃圾收集器的选择和具体的设置都有关系。

（2）垃圾收集的精确性：主要包括2 个方面：（a）垃圾收集器能够精确标记活着的对象；（b）垃圾收集器能够精确地定位对象之间的引用关系。前者是完全地回收所有废弃对象的前提，否则就可能造成内存泄漏。而后者则是实现归并和复制等算法的必要条件。所有不可达对象都能够可靠地得到回收，所有对象都能够重新分配，允许对象的复制和对象内存的缩并，这样就有效地防止内存的支离破碎。

（3）现在有许多种不同的垃圾收集器，每种有其算法且其表现各异，既有当垃圾收集开始时就停止应用程序的运行，又有当垃圾收集开始时也允许应用程序的线程运行，还有在同一时间垃圾收集多线程运行。

（4）垃圾收集的实现和具体的JVM 以及JVM的内存模型有非常紧密的关系。不同的JVM 可能采用不同的垃圾收集，而JVM 的内存模型决定着该JVM可以采用哪些类型垃圾收集。现在，HotSpot 系列JVM中的内存系统都采用先进的面向对象的框架设计，这使得该系列JVM都可以采用最先进的垃圾收集。

（5）随着技术的发展，现代垃圾收集技术提供许多可选的垃圾收集器，而且在配置每种收集器的时候又可以设置不同的参数，这就使得根据不同的应用环境获得最优的应用性能成为可能。

#### 之五：java本地接口JNI详解

对于java程序员来说，java语言的好处和优点，我想不用我说了，大家自然会说出很多一套套的。但虽然我们作为java程序员，但我们不得不承认java语言也有一些它本身的缺点。比如在性能、和底层打交道方面都有它的缺点。所以java就提供了一些本地接口，他主要的作用就是提供一个标准的方式让java程序通过虚拟机与原生代码进行交互，这也就是我们平常常说的java本地接口（JNI——java native Interface）。它使得在 Java 虚拟机 (VM) 内部运行的 Java 代码能够与用其它编程语言（如 C、C++ 和汇编语言）编写的应用程序和库进行互操作。JNI 最重要的好处是它没有对底层 Java 虚拟机的实现施加任何限制。因此，Java虚拟机厂商可以在不影响虚拟机其它部分的情况下添加对 JNI 的支持。程序员只需编写一种版本的本地应用程序或库，就能够与所有支持 JNI 的 Java 虚拟机协同工作。我们来看一下为什么要与原生代码进行交互：

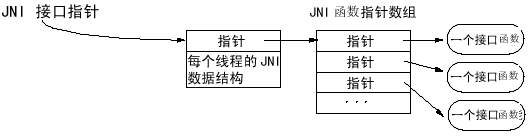
**一：提高应用程序性能**。我们知道java对于c/c++、汇编语言来说，显得比较“高级”。其实这里的高级就是简化了程序员的工作。很多底层的东西都让java虚拟机做了。但毕竟相对于直接访问底层来讲，java多了一步虚拟机的过程，所以在性能上比着这些原生语言稍微有点慢。

**二：实现一些与底层相关的功能**。Java平台提供的标准类库，还有强大的API，虽然能完成大部分功能。但有些和底层硬件打交道的功能在java API提供的类库中还是无法完成。

**三：与已有的使用原生代码编写的程序进行集成**。在于操作系统上由c或者c++等原生语言编写的软件进行集 0成的时候，可以用JNI。

**JNI 接口函数和指针**

    平台相关代码是通过调用 JNI 函数来访问 Java 虚拟机功能的。JNI 函数可通过接口指针来获得。接口指针是指针的指针，它指向一个指针数组，而指针数组中的每个元素又指向一个接口函数。每个接口函数都处在数组的某个预定偏移量中。下图说明了接口指针的组织结构。



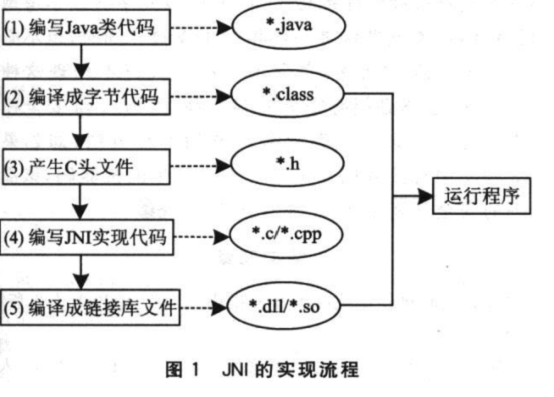
          JNI 接口的组织类似于 C++ 虚拟函数表或 COM 接口。使用接口表而不使用硬性编入的函数表的好处是使 JNI 名字空间与平台相关代码分开。虚拟机可以很容易地提供多个版本的 JNI 函数表。例如，虚拟机可支持以下两个 JNI 函数表：

· 一个表对非法参数进行全面检查，适用于调试程序；

· 另一个表只进行 JNI 规范所要求的最小程度的检查，因此效率较高。

JNI 接口指针只在当前线程中有效。因此，本地方法不能将接口指针从一个线程传递到另一个线程中。实现 JNI 的虚拟机可将本地线程的数据分配和储存在 JNI 接口指针所指向的区域中。

本地方法将JNI 接口指针当作参数来接受。虚拟机在从相同的 Java 线程中对本地方法进行多次调用时，保证传递给该本地方法的接口指针是相同的。但是，一个本地方法可被不同的 Java 线程所调用，因此可以接受不同的 JNI 接口指针。



**(1)编写Java类代码**

   其中，需要JNI实现的方法应当用native关键字声明。在该类中，用System．1oadLibrary()方法加载需要的动态链接库。关键代码如下：  
   ／／Compute.java  
   public class Compute{  
   public native double sqrt(double  params);  
   static{  
   ／／调用动态链接库  
   System．loadLibrary(“compute”)；  
   }  
     
**(2)编译成字节代码**

   在这个过程中，由于采用了native关键字声明，Java编译器会忽视没有代码体的JNI方法部分。  
    
**(3)生成相关JNI方法的头文件**

   这个过程的实现一般是通过利用jlavah-jni  \* class生成的(-jni可以省略），也可以手工生成该文件；但是由于 Java 虚拟机是根据一定的命名规范完成对JNI方法的调用，所以手工编写头文件需要特别小心。  
   上述文件产生的头文件部分代码如下：  
   ／／Compute.h  
   extern“C”{  
   JNIEXPORT jdoubleJNICALL Java\_Compute\_comp(JNI-Env \*, jobject, jdoubleArray)；

   JNI函数名称分为三部分：首先是Java关键字，供Java虚拟机识别；然后是调用者类名称(全限定的类名，其中用下划线代替名称分隔符)；最后是对应的方法名称，各段名称之间用下划线分割。

   JNI函数的参数也由三部分组成：首先是JNIEnv \*,是一个指向JNI运行环境的指针；第二个参数随本地方法是静态还是非静态而有所不同一一非静态本地方法的第二个参数是对对象的引用，而静态本地方法的第二个参数是对其Java类的引用；其余的参数对应通常Java方法的参数，参数类型需要根据一定规则进行映射。

**(4)编写相应方法的实现代码**

   在编码过程中，需要注意变量的长度问题，例如Java的整型变量长度为32位，而C语言为16位，所以要仔细核对变量类型映射表，防止在传值过程中出现问题。

  **(5)将JNI实现代码编译成动态链接库**

   编译过程是利用C／C++编译器实现的，在windows平台上，编译和连接的结果是动态链接库DLL文件。当要使用生成的动态链接库时，调用者类中需要显式调用该链接库dll文件。  
   经过上述处理，基本上完成了一个包含本地化方法的Java类的开发。

**附录：将Jav类型映射到本地 C 类型**

|  |  |  |
| --- | --- | --- |
| 基本类型和本地等效类型 | | |
| Java 类型 | 本地类型 | 说明 |
| boolean | jboolean | 无符号，8 位 |
| byte | jbyte | 无符号，8 位 |
| char | jchar | 无符号，16 位 |
| short | jshort | 有符号，16 位 |
| int | jint | 有符号，32 位 |
| long | jlong | 有符号，64 位 |
| float | jfloat | 32 位 |
| double | jdouble | 64 位 |
| void | void | N/A |

为了使用方便，特提供以下定义。

   #define JNI\_FALSE  0

   #define JNI\_TRUE   1

jsize 整数类型用于描述主要指数和大小：

typedef jint jsize;

**故障排除**

当使用 JNI 从 Java 程序访问本机代码时，您会遇到许多问题。您会遇到的三个最常见的错误是：

1）无法找到动态链接。它所产生的错误消息是：java.lang.UnsatisfiedLinkError。这通常指无法找到共享库，或者无法找到共享库内特定的本机方法。

2）无法找到共享库文件。当用 System.loadLibrary(String libname) 方法（参数是文件名）装入库文件时，请确保文件名拼写正确以及没有指定扩展名。还有，确保库文件的位置在类路径中，从而确保 JVM 可以访问该库文件。

3）无法找到具有指定说明的方法。确保您的 C/C++ 函数实现拥有与头文件中的函数说明相同的说明。

**结束语**

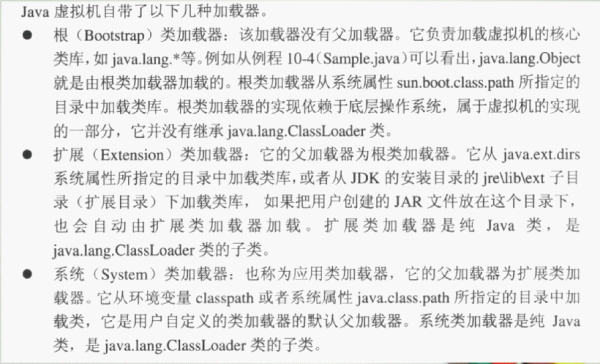
           从 Java 调用 C 或 C++ 本机代码（虽然不简单）是 Java 平台中一种良好集成的功能。虽然 JNI 支持 C 和 C++，但 C++ 接口更清晰一些并且通常比 C 接口更可取。正如您已经看到的，调用 C 或 C++ 本机代码需要赋予函数特殊的名称，并创建共享库文件。当利用现有代码库时，更改代码通常是不可取的。要避免这一点，在 C++ 中，通常创建代理代码或代理类，它们有专门的 JNI 所需的命名函数。然后，这些函数可以调用底层库函数，这些库函数的说明和实现保持不变。

#### 之六：类加载的父亲委托机制

我们都知道。类加载器用来把类加载到java虚拟机。从JDK2.0开始，类的加载过程采用父亲委托机制。JVM的ClassLoader采用的是树形结构，除了根类加载器以外，每个ClassLoader都会有且仅有一个父类加载器，用户自定义的ClassLoader默认的父类加载器是系统类加载器，当然你可以自己指定需要用个ClassLoader的实例，我们来看他们的父子关系：



     父类委托机制中，当一个java程序请求加载器loader1加载Hello类时，loader1首先委托自己的父亲加载器加载hello类，若父亲加载器能加载，则由附加器完成加载人物，否则才由加载器loader1本身加载Hello类。下面我们来再次看一下java虚拟机自带的几个加载器：



      除了java虚拟机自带的加载器之外，我们用户自己也可以自定义自己的类加载器，根据自己的需要。。Java提供了抽象类java.lang.ClassLoder，所有用户自定义的类加载器都要继承这个classloader类。

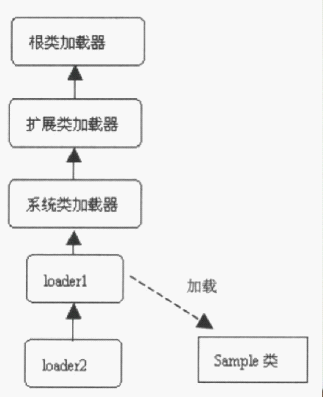
**注：**加载器之间的父子关系实际上指的是加载器对象之间的包装关系，而不是类之间的继承关系。一对父子加载器可能是同一个加载器类的两个实例，也可能不是。在子加载器对象中包装了一个父加载器对象.当生成一个自定义的类加载器实例时，如果没有指定它的父加载器，那么系统类加载器就将成为该类加载器的父加载器。如果在构造方法中指定父类加载器那么父类加载器就是指定的加载器。证明如下：



**[java]** view plaincopyprint?

1. ClassLoader loader1 = **new** MyClassLoader();
2. //参数loader1将作为loader2的父加载器
3. ClassLoader loader2 = **new** MyClassLoader(loader1);

当Java虚拟机要加载一个类时，到底该派哪个类加载器去加载呢 ？我们看下图：



       Loader1和loader2是我们自己定义的两个类加载器，loader1和loader2是父子关系。现在我们想让loader2这个类加载器加载我们自己写的一个Sample类：loader2.loadclass（“sample”），我们来分析一下看看到底应该用哪一个类加载器去加载。当这段代码被执行时，loader2首先到自己的命名空间去查找Sample类是否已经被加载，如果被加载就直接返回这个类的class对象的引用。如果Sample类还没有被加载，loader2首先请求loader1代为加载，loader1再请求系统类加载器代为加载，系统类加载器再请求扩展类加载器，扩展类加载器再请求根类加载器，若根类加载器和扩展类加载器都不能加载，则系统类加载器尝试加载，若能加载，则将Sample类所对应的Class对象的引用返回给loader1，loader1在将引用返回给loader2，从而成功将Sample类加载到虚拟机。若系统类加载器不能加载Sample类，则loader1尝试加载Sample了哦，若loader1不能加载，则loader2尝试，若所有的类加载都不能加载，则抛出ClassNotFoundException异常。

**定义类加载器**：如果某个类加载器能够加载一个类，那么该类加载器就称作：定义类加载器；定义类加载器及其所有子加载器都称作：**初始类加载器**

     父委托机制的优点就是能够提高软件系统的安全性。因为在词机制下，用户自定义的类加载器不可能加载本应该由父加载器加载的可靠类，从而防止不可靠的恶意代码代替由父类加载器加载的可靠类，从而防止不可靠的甚至恶意的代码代替由父类加载器加载的可靠代码。如，java.lang.Object类总是由根类加载器加载的，其他任何用户自定义的类加载器都不可能加载含有恶意代码的java.lang.Object类。

    命名空间，其实这里所说的命名空间就是我们java中常用的package，每个类加载器都有自己的命名空间，命名空间由该加载器及所有父加载器所加载的类的组成。在同一个命名空间中，不会出现类的完整名字（包括类的包名）相同的两个雷；在不同的命名空间中，有可能会出现类的完整名字（包括类的包名）相同的两个类。

 由同一类加载器加载的属于相同包的类组成了运行时包。决定两个类是不是属于同一个运行时包，不仅要看他们的包名称是否相同，还要看定义类加载器是否相同。只有属于同一运行时包的类之间才能相互访问可见（默认访问级别）的类和成员。假设用户自定义了一个类java.lang.TestCase并由用于自定义的类加载器加载，由于java.lang.TestCase和核心类库java.lang.\*由不同的类加载器加载，他们属于不同的运行时包，所以java.lang.TestCase不能访问核心库java.lang包中的包可见成员。

#### 之七：深入源码看java类加载器ClassLoader

ClassLoader类加载器是负责加载类的对象。ClassLoader 类是一个抽象类。如果给定类的二进制名称（即为包名加类名的全称），那么类加载器会试图查找或生成构成类定义的数据。一般策略是将名称转换为某个文件名，然后从文件系统读取该名称的“类文件”。java.lang.ClassLoader类的基本职责就是根据一个指定的类的名称，找到或者生成其对应的字节代码，然后从这些字节代码中定义出一个 Java 类，即 java.lang.Class类的一个实例。除此之外，ClassLoader还负责加载 Java 应用所需的资源，如图像文件和配置文件等。 作为所有类加载器的基类，ClassLoader的内部实现机制还是值得我们细细研究一下的。所以今天我就带领大家一起来看一下Classloader的内部实现源码。

首先我们来看一下Classloader类的两个构造方法。



       从上边的帮助文档中我们可以发现，在创建一个classloader的实例时我们可以显示的指出他的父加载器，也可以不指定，不指定的时候他的默认的父加载器是系统加载器。我们来看一下源码的实现：

**[java]** view plaincopyprint?

1. /\*\*
2. \* Creates a new class loader using the <tt>ClassLoader</tt> returned by
3. \* the method {@link #getSystemClassLoader()
4. \* <tt>getSystemClassLoader()</tt>} as the parent class loader.
6. \*/
7. **protected** ClassLoader() {
8. SecurityManager security = System.getSecurityManager();
9. **if** (security != **null**) {
10. security.checkCreateClassLoader();
11. }
12. **this**.parent = getSystemClassLoader();
13. initialized = **true**;
14. }

**[java]** view plaincopyprint?

2. /\*\*
3. \* Creates a new class loader using the specified parent class loader for
4. \*/
5. **protected** ClassLoader(ClassLoader parent) {
6. SecurityManager security = System.getSecurityManager();
7. **if** (security != **null**) {
8. security.checkCreateClassLoader();
9. }
10. **this**.parent = parent;
11. initialized = **true**;
12. }

     从上面源码我们可以发现，classloader中一定有一个parent的属性来指定当前loader的附加器。其源码也证明了我们刚才上面的说法。

    看完构造方法的实现，下一步我们来看一下ClassLoader中一个特别重要的方法loadclass方法，这个方法就是用来加载类的。这个方法在jdk中有个重载的方法，但是其实是一个样的。一个带有是否链接类的方法，一个不带。下面我们就看一下这个方法的源码：

**[java]** view plaincopyprint?

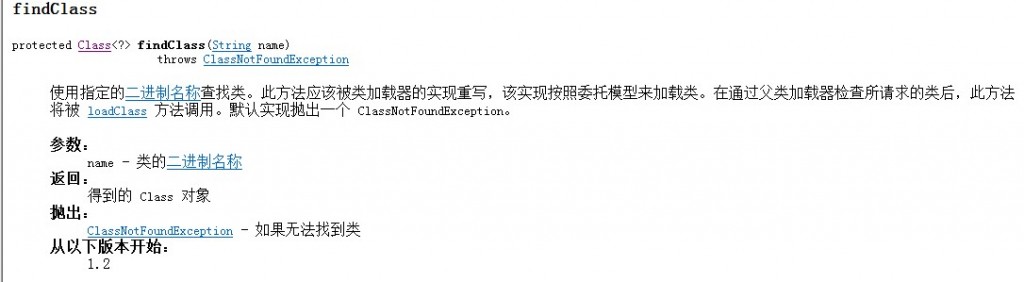
1. **public** Class<?> loadClass(String name) **throws** ClassNotFoundException {
2. **return** loadClass(name, **false**);
3. }
4. /\*\*
5. \* Loads the class with the specified <a href="#name">binary name</a>.  The
6. \* default implementation of this method searches for classes in the
8. \* @throws  ClassNotFoundException
9. \*          If the class could not be found
10. \*/
11. **protected** **synchronized** Class<?> loadClass(String name, **boolean** resolve)
12. **throws** ClassNotFoundException
13. {
14. // First, check if the class has already been loaded
15. Class c = findLoadedClass(name);
16. **if** (c == **null**) {
17. **try** {
18. **if** (parent != **null**) {
19. c = parent.loadClass(name, **false**);
20. } **else** {
21. c = findBootstrapClass0(name);
22. }
23. } **catch** (ClassNotFoundException e) {
24. // If still not found, then invoke findClass in order
25. // to find the class.
26. c = findClass(name);
27. }
28. }
29. **if** (resolve) {
30. resolveClass(c);
31. }
32. **return** c;
33. }

关于这个方法的源码解释，我感觉我没必要解释什么，因为在API中已经有详细的解释了。我在解释肯定也不如他解释的好，所以我们来看一下API是怎么解释的吧：



       虽然jdk已经解释的很清楚了，但是有一点我还是要在补充一下。从上面的源码我们可以看出loadClass方法是一个递归的方法，一直往上找，一直找到根类加载器为止，然后让类加载器去加载这个类。至于跟加载器是怎么实现的我们就不得而知了。因为跟类加载加载类时一个本地方法，他是用c++写的。我们无法看到他的源码。这样验证了在前面我们所说的类加载器的父类委托机制。

下面我们来看一下findClass方法，我们在上面的代码中发现。在loadClass方法中有调用这个findclass方法，下面我们首先来看一下API对这个方法的介绍：



      从上面介绍我们可以看出，这个方法主要是来查找我们的类文件的。我们在自定义我们自己的类加载器的时候应该重写这个方法，因为jdk中对这个方法基本没有实现什么，这就需要我们自己来重写这个方法，用我们自己的所定义的方法去查找类文件。不信。你可以看一下他的源码实现：

**[java]** view plaincopyprint?

1. **protected** Class<?> findClass(String name) **throws** ClassNotFoundException {
2. **throw** **new** ClassNotFoundException(name);
3. }

暂时先介绍这些吧。其他的方法基本都差不多，最近感觉博客越来越难写，越来越吃力。越来越发现底层理论的缺乏。希望通过自己的努力，可以慢慢改变这个现状。

#### 之八：开发自己的类加载器

在大多数情况下，系统默认提供的类加载器实现已经可以满足需求。但是在某些情况下，您还是需要为应用开发出自己的类加载器。比如您的应用通过网络来传输 Java 类的字节代码，为了保证安全性，这些字节代码经过了加密处理。这个时候您就需要自己的类加载器来从某个网络地址上读取加密后的字节代码，接着进行解密和验证，最后定义出要在 Java 虚拟机中运行的类来。下面将通过两个具体的实例来说明类加载器的开发。

    要创建自定义的类加载器只需要扩展java.lang.ClassLoader类就可以，然后覆盖它的findClass(String name)方法即可。该方法根据参数指定的类的名字，去找对应的class文件。然后返回class对应的对象。下面我们就根据我们自定义的类加载器的源码来具体详解一下这个自定义的步骤：

自定义的类加载器：

**[java]** view plaincopyprint?

1. **package** com.bzu.csh.test;
2. **import** java.io.ByteArrayOutputStream;
3. **import** java.io.File;
4. **import** java.io.FileInputStream;
5. **import** java.io.InputStream;
6. /\*\*
7. \*
8. \* @author 曹胜欢
9. \* @version 1.0
10. \*
11. \*/
12. **public** **class** MyClassLoader **extends** ClassLoader {
13. **private** String name; // 类加载器的名字
14. **private** String path = "d:\\"; // 加载类的路径
15. **private** **final** String fileType = ".class"; // class文件的扩展名
16. **public** MyClassLoader(String name) {
17. **super**(); // 让系统类加载器成为该类加载器的父加载器
18. **this**.name = name;
19. }
20. **public** MyClassLoader(ClassLoader parent, String name) {
21. **super**(parent); // 显式指定该类加载器的父加载器
22. **this**.name = name;
23. }
24. @Override
25. **public** String toString() {
26. **return** **this**.name;
27. }
28. **public** String getPath() {
29. **return** path;
30. }
31. **public** **void** setPath(String path) {
32. **this**.path = path;
33. }
34. /\*\*
35. \* @param 类文件的名字
36. \* @return 类文件中类的class对象
37. \*
38. \* 在这里我们并不需要去显示的调用这里的findclass方法，在上篇文章中，我们通过查看
39. \* loadclass的源码可以发现，她是在loadclass中被调用的，所以这里我们只需重写这个方法，
40. \* 让它根据我们的想法去查找类文件就ok，他会自动被调用
41. \*
42. \*
43. \* defineClass()将一个 byte 数组转换为 Class 类的实例。必须分析 Class，然后才能使用它
44. \* 参数：
45. \* name - 所需要的类的二进制名称，如果不知道此名称，则该参数为 null
46. \* b - 组成类数据的字节。off 与 off+len-1 之间的字节应该具有《Java Virtual Machine Specification》定义的有效类文件的格式。
47. \* off - 类数据的 b 中的起始偏移量
48. \* len - 类数据的长度
49. \*/
50. @Override
51. **public** Class<?> findClass(String name) **throws** ClassNotFoundException {
52. **byte**[] data = **this**.loadClassData(name);//获得类文件的字节数组
53. **return** **this**.defineClass(name, data, 0, data.length);//
54. }
55. /\*\*
56. \*
57. \* @param 类文件的名字
58. \* @return 类文件的 字节数组
59. \* 通过类文件的名字获得类文件的字节数组，其实主要就是用
60. \* 输入输出流实现。
61. \*/
62. **private** **byte**[] loadClassData(String name) {
63. InputStream is = **null**;
64. **byte**[] data = **null**;
65. ByteArrayOutputStream baos = **null**;
66. **try** {
67. **this**.name = **this**.name.replace(".", "\\");
68. is = **new** FileInputStream(**new** File(path + name + fileType));
69. baos = **new** ByteArrayOutputStream();
70. **int** ch = 0;
71. **while** (-1 != (ch = is.read())) {
72. baos.write(ch);
73. }
74. data = baos.toByteArray();
75. } **catch** (Exception ex) {
76. ex.printStackTrace();
77. } **finally** {
78. **try** {
79. is.close();
80. baos.close();
81. } **catch** (Exception ex) {
82. ex.printStackTrace();
83. }
84. }
85. **return** data;
86. }

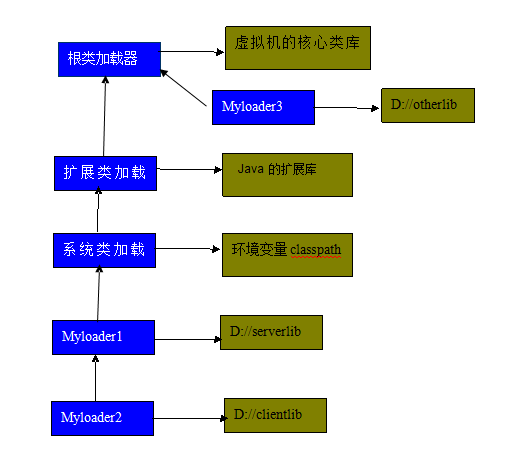
      我想上面的注释中已经足够让大家明白这个自定义类加载器的原理了。在这我在重复的从上到下的再说一遍，加深一下大家的理解。首先在构造方法中，我们可以通过构造方法给类加载器起一个名字，也可以显示的指定他的父累加器器，如果没有显示的指出父类加载器的话他默认的就是系统类加载器。由于我们继承了ClassLoader类，所以它自动继承了父类的loadclass方法。我们以前看过loadclass的源码知道，它调用了findclass方法去查找类文件。所以在这里我们重写了ClassLoader的findclass方法。在这个方法中首先调用loadClassData方法，通过类文件的名字获得类文件的字节数组，其实主要就是用输入输出流实现。然后调用defineClass()将一个 字节 数组转换为 Class 类的实例。有时候我们手动生成的二进制码的class文件被加密了。所以在我们在利用我们自定义的类加载器的时候还要写一个解密的方法进行解密，这里我们就不实现了。

我们实现了自定义类加载器，下一步我们来看一下我们怎么来应用我们这个自定义的类加载器：

**[java]** view plaincopyprint?

1. **public** **static** **void** main(String[] args) **throws** Exception {
2. //创建一个loader1类加载器，设置他的加载路径为d:\\serverlib\\，设置默认父加载器为系统类加载器
3. MyClassLoader loader1 = **new** MyClassLoader("loader1");
4. loader1.setPath("d:\\myapp\\serverlib\\");
5. //创建一个loader2类加载器，设置他的加载路径为d:\\clientlib\\，并设置父加载器为loader1
6. MyClassLoader loader2 = **new** MyClassLoader(loader1, "loader2");
7. loader2.setPath("d:\\myapp\\clientlib\\");
8. //创建一个loader3类加载器，设置他的加载路径为d:\\otherlib\\，并设置父加载器为根类加载器
9. MyClassLoader loader3 = **new** MyClassLoader(**null**, "loader3");
10. loader3.setPath("d:\\myapp\\otherlib\\");
11. test(loader2);
12. System.out.println("----------");
13. test(loader3);
14. }
15. **public** **static** **void** test(ClassLoader loader) **throws** Exception {
16. Class clazz = loader.loadClass("com.bzu.csh.test.Sample");
17. Object object = clazz.newInstance();
18. }

                                                                         类加载器结构图



                     （PS:突然发现WPS自带的画图工具也挺好用，虽然有点难看，哈哈）

          当执行这段代码的时候。首先让loader2去加载Sample类文件，当然我们在执行这段代码的前提时在各个默认加载器中已经有我们Sample的class文件。Loader2首先让父加载器是loader1去加载，然后loader1会让系统类加载器去加载，系统类加载器会让扩展类加载器加载，扩展类加载器会让根类加载器加载，由于系统类加载器，扩展类加载器，根类加载器的默认路径中都没有我们要的sample类，所以loader2的默认路径有sample这个类，也就是说loader2会去加载这个sample类。当执行test（loader3)的时候，由于loader3的默认父加载器是根类加载器，并且根类加载前默认路径没有对应的sample.class文件，所以，直接的loader3类加载器就去加载这个类。

       最后要说明的一点是，自定义类加载不光只能从我们本地加载到class文件，我们也可以加载网络，即基本的场景是：Java 字节代码（.class）文件存放在服务器上，客户端通过网络的方式获取字节代码并执行。当有版本更新的时候，只需要替换掉服务器上保存的文件即可。通过类加载器可以比较简单的实现这种需求。其实他的实现和本地差不多，基本上就是geclassdata方法改变了一些。下面我们来具体看一下：

**[java]** view plaincopyprint?

1. **private** **byte**[] getClassData(String className) {
2. String path = classNameToPath(className);
3. **try** {
4. URL url = **new** URL(path);
5. InputStream ins = url.openStream();
6. ByteArrayOutputStream baos = **new** ByteArrayOutputStream();
7. **int** bufferSize = 4096;
8. **byte**[] buffer = **new** **byte**[bufferSize];
9. **int** bytesNumRead = 0;
10. **while** ((bytesNumRead = ins.read(buffer)) != -1) {
11. baos.write(buffer, 0, bytesNumRead);
12. }
13. **return** baos.toByteArray();
14. } **catch** (Exception e) {
15. e.printStackTrace();
16. }
17. **return** **null**;
18. }

      在通过网络加载了某个版本的类之后，一般有两种做法来使用它。第一种做法是使用 Java 反射 API。另外一种做法是使用接口。需要注意的是，并不能直接在客户端代码中引用从服务器上下载的类，因为我们写的类加载器被加载所用的类加载器和我们加载的网络类不是同一个类加载器，所以客户端代码的类加载器找不到这些类。使用 Java 反射 API 可以直接调用 Java 类的方法。而使用接口的做法则是把接口的类放在客户端中，从服务器上加载实现此接口的不同版本的类。在客户端通过相同的接口来使用这些实现类。

**不同类加载器的命名空间关系：**

1.同一个命名空间内的类是相互可见的。

2.子加载器的命名空间包含所有父加载器的命名空间。因此子加载器加载的类能看见父加载器加载的类。例如系统类加载器加载的类能看见根类加载器加载的类。

3.由父加载器加载的类不能看见子加载器加载的类。

4.如果两个加载器之间没有直接或间接的父子关系，那么它们各自加载的类相互不可见。

5.当两个不同命名空间内的类相互不可见时，可以采用Java的反射机制来访问实例的属性和方法。