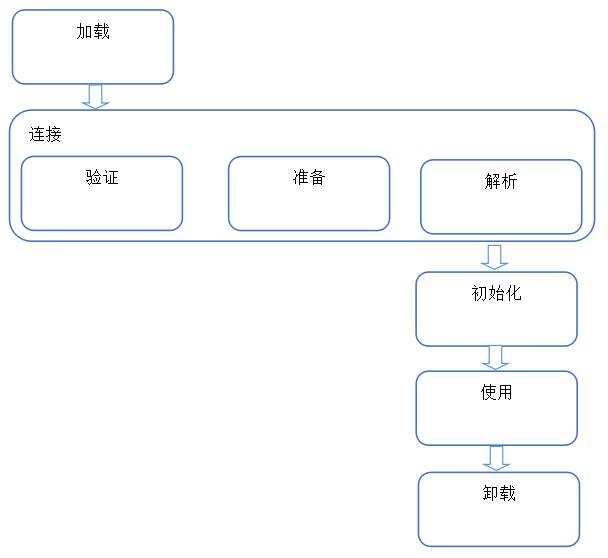
为什么研究类加载全过程？

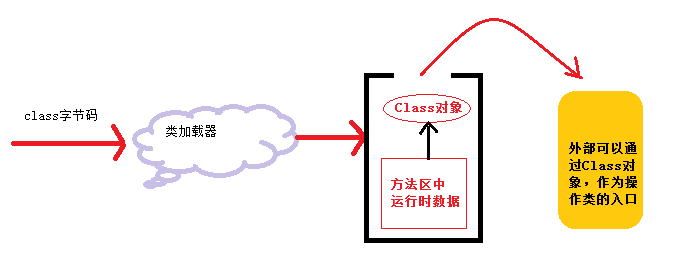
* 有助于连接JVM运行过程
* 更深入了解java动态性（解热部署，动态加载），提高程序的灵活性

**类加载机制**

* JVM把class文件加载到内存，并对数据进行校验、解析和初始化，最终形成JVM可以直接使用的java类型的全过程。



* **加载**
  + 将class文件字节码内容加载到内存中，并将这些静态数据转换成方法区中的运行时数据结构，在堆中生成一个代表这个类的java.lang.Class对象，作为方法区类数据的访问入口，这个过程需要类加载器参与。



* **链接**   将java类的二进制代码合并到JVM的运行状态之中的过程
  + **验证：**确保加载的类信息符合JVM规范，没有安全方面的问题
  + **准备：**正式为类变量（static变量）分配内存并设置类变量初始值的阶段，这些内存都将在方法去中进行分配
  + **解析：**虚拟机常量池的符号引用替换为字节引用过程
* **初始化**
  + 初始化阶段是执行类构造器<clinit>（）方法的过程。类构造器<clinit>（）方法是由编译器自动收藏类中的所有类变量的赋值动作和静态语句块(static块)中的语句合并产生
  + 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化
  + 虚拟机会保证一个类的<clinit>（）方法在多线程环境中被正确加锁和同步
  + 当范围一个Java类的静态域时，只有真正声名这个域的类才会被初始化

**例1：**

[复制代码](javascript:void(0);)

public class Demo01 {

public static void main(String[] args) {

A a = new A();

System.out.println(a.width);

}

}

class A{

public static int width=100; //静态变量，静态域 field

static{

System.out.println("静态初始化类A");

width = 300 ;

}

public A() {

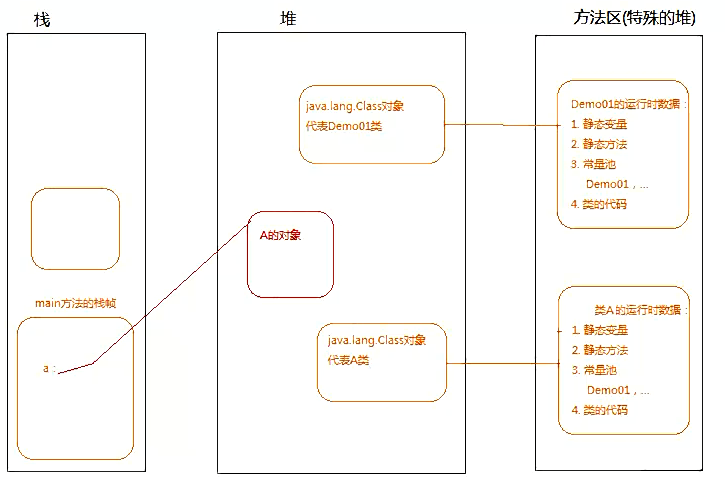
System.out.println("创建A类的对象");

}

}

[复制代码](javascript:void(0);)

**分析：**



**说明：**

内存中存在栈、堆（放创建好的对象）、方法区（实际也是一种特殊堆）

1、JVM加载Demo01时候，首先在方法区中形成Demo01类对应静态数据（类变量、类方法、代码…），同时在堆里面也会形成java.lang.Class对象（反射对象），代表Demo01类，通过对象可以访问到类二进制结构。然后加载变量A类信息，同时也会在堆里面形成a对象，代表A类。

2、main方法执行时会在栈里面形成main方法栈帧，一个方法对应一个栈帧。如果main方法调用了别的方法，会在栈里面挨个往里压，main方法里面有个局部变量A类型的a，一开始a值为null，通过new调用类A的构造器，栈里面生成A（）方法同时堆里面生成A对象，然后把A对象地址付给栈中的a,此时a拥有A对象地址。

3、当调用A.width时，调用方法区数据。

当类被引用的加载，类只会加载一次

* **类的主动引用（一定会发生类的初始化）**
  + new一个类的对象
  + 调用类的静态成员（除了final常量）和静态方法
  + 使用java.lang.reflect包的方法对类进行反射调用
  + 当虚拟机启动，java Demo01,则一定会初始化Demo01类，说白了就是先启动main方法所在的类
  + 当初始化一个类，如果其父类没有被初始化，则先初始化它父类
* **类的被动引用（不会发生类的初始化）**
  + 当访问一个静态域时，只有真正声名这个域的类才会被初始化
    - 通过子类引用父类的静态变量，不会导致子类初始化
  + 通过数组定义类的引用，不会触发此类初始化
  + 引用常量不会触发此类的初始化（常量在编译阶段就存入调用类的常量池中了）

**例2：**

[复制代码](javascript:void(0);)

public class Demo01 {

static{

System.out.println("静态初始化Demo01");

}

public static void main(String[] args) throws Exception {

System.out.println("Demo01的main方法！");

System.out.println(System.getProperty("java.class.path"));

//主动引用

// new A();

// System.out.println(A.width);

// Class.forName("com.sinosoft.test.A");

//被动引用

// System.out.println(A.MAX);

// A[] as = new A[10];

System.out.println(B.width);//B类不会被加载

}

}

class B extends A {

static {

System.out.println("静态初始化B");

}

}

class A extends A\_Father {

public static int width=100; //静态变量，静态域 field

public static final int MAX=100;

static {

System.out.println("静态初始化类A");

width=300;

}

public A(){

System.out.println("创建A类的对象");

}

}

class A\_Father extends Object {

static {

System.out.println("静态初始化A\_Father");

}

}

[复制代码](javascript:void(0);)

虚拟机把描述类的数据从class文件加载到内存，并对数据进行校验，转换分析和初始化，最终形成可以被虚拟节直接使用的JAVA类型，这就是虚拟机的类加载机制。

类从被加载到虚拟机内存到卸载出内存的生命周期包括：加载->连接(验证->准备->解析)->初始化->使用->卸载

初始化的5种情况：

1.使用new关键字实例化对象时，读取或设置一个类的静态字段，除被final修饰经编译结果放在常量池的静态字段，调用类的静态方法时。 2.使用java.lang.reflect包方法对类进行反射调用时。（Class.forName()）。 3.初始化子类时，如果父类没有初始化。 4.虚拟机启动时main方法所在的类。 5.当使用JDK1.7动态语言支持时，java.lang.invoke.MethodHandle实例解析结果为REF\_getStatic,REF\_putStatic,REF\_invokeStatic的方法句柄，且对应类没有进行初始化。

加载 加载是类加载的第一个阶段，虚拟机要完成以下三个过程：

1.通过类的全限定名获取定义此类的二进制字节流。 2.将字节流的存储结构转化为方法区的运行时结构。 3.在内存中生成一个代表该类的Class对象，作为方法区各种数据的访问入口。

验证 目的是确保class文件字节流信息符合虚拟机的要求。

准备 为static修饰的变量赋初值，例如int型默认为0，boolean默认为false。

解析 虚拟机将常量池内的符号引用替换成直接引用。

初始化 初始化是类加载的最后一个阶段，将执行类构造器< init>()方法，注意这里的方法不是构造方法。该方法将会显式调用父类构造器，接下来按照java语句顺序为类变量和静态语句块赋值。

方法调用

Java是一门面向对象的语言，它具有多态性。那么虚拟机又是如何知道运行时该调用哪一个方法？

静态分派是在编译期就决定了该调用哪一个方法而不是由虚拟机来确定，方法重载就是典型的静态分派。 动态分派是在虚拟机运行阶段才能决定调用哪一个方法，方法重写就是典型的动态分派。

动态分派的实现：当调用一个对象的方法时，会将该对象的引用压栈到操作数栈，然后字节码指令invokevirtual会去寻找该引用实际类型。如果在实际类型中找对应的方法，且访问权限足够，则直接返回该方法引用，否则会依照继承关系对父类进行查找。实际上，如果子类没有重写父类方法，则子类方法的引用会直接指向父类方法。

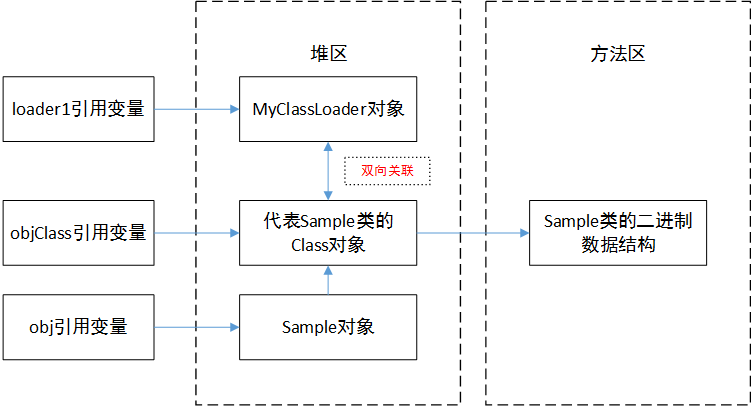
**由Java虚拟机自带的类加载器所加载的类，在虚拟机的生命周期中，始终不会被卸载。**

　　前面介绍过，Java虚拟机**自带的类加载器**包括**根类加载器**、**扩展类加载器**和**系统类加载器**。

　　Java虚拟机本身会**始终引用**这些类加载器，而这些类加载器则会**始终引用**它们所加载的类的Class对象，因此这些Class对象**始终是可触及的**。

**由用户自定义的类加载器加载的类是可以被卸载的。**

**具体的例子为：**

****

       loader1变量和obj变量间接应用代表Sample类的Class对象，而objClass变量则直接引用它。

　　如果程序运行过程中，将上图左侧三个引用变量都置为null，此时Sample对象结束生命周期，MyClassLoader对象结束生命周期，代表Sample类的Class对象也结束生命周期，Sample类在方法区内的二进制数据**被卸载**。

　　当再次有需要时，会检查Sample类的Class对象是否存在，**如果存在会直接使用，不再重新加载**；如果不存在Sample类会被重新加载，在Java虚拟机的堆区会生成一个新的代表Sample类的Class实例(可以通过哈希码查看是否是同一个实例)。

 当Java编译器编译好.class文件之后，我们需要使用JVM来运行这个class文件。那么最开始的工作就是要把字节码从磁盘输入到内存中，这个过程我们叫做【**加载**】。加载完成之后，我们就可以进行一系列的运行前准备工作了，比如： 为类静态变量开辟空间，将常量池存放在方法区内存中并实现常量池地址解析，初始化类静态变量等等。这篇文章我们要好好谈谈JVM是如何加载class文件的？  
   
**1、JVM加载类的过程**  
      当我们使用命令来执行某一个Java程序(比如Test.class)的时候：java Test  
      (1) java.exe 会帮助我们找到 JRE ，接着找到位于 JRE 内部的 jvm.dll ，这才是真正的 Java 虚拟机器 , 最后加载动态库，激活 Java 虚拟机器。  
      (2) 虚拟机器激活以后，会先做一些初始化的动作，比如说读取系统参数等。一旦初始化动作完成之后，就会产生第一个**类装载器**―― Bootstrap Loader(**启动类装载器** ) 。  
      (3) Bootstrap Loader 所做的初始工作中，除了一些基本的初始化动作之外，最重要的就是加载 Launcher.java 之中的 ExtClassLoader(**扩展类装载器)**，并设定其 Parent 为 null ，代表其父加载器为 BootstrapLoader 。  
      (4) 然后 Bootstrap Loader 再要求加载 Launcher.java 之中的 AppClassLoader(**用户自定义类装载器**) ，并设定其 Parent 为之前产生的 ExtClassLoader 实体。这两个加载器都是以静态类的形式存在的。  
      这里要请大家注意的是， Launcher$ExtClassLoader.class 与 Launcher$AppClassLoader.class 都是由 Bootstrap Loader 所加载，所以 Parent 和由哪个类加载器加载没有关系。   
      初学者对这个过程很难理解，我们将在下面详细的讲讲类装载器和"Parent"是什么。  
   
   
**2、类装载器体系结构**  
   
      JVM加载class文件必须通过一个叫做类装载器的程序，它的作用就是从磁盘文件中将要运行代码的字节码流加载进内存(JVM管理的方法区)中。下面是几个比较重要的概念：   
   
      **(1) 启动类装载器 ：**每个Java虚拟机实现都必须有一个启动类装载器。它只负责在系统类(核心Java API的class文件)的安装路径中查找要装入的类。这个装载器的实现由C++ 所撰写而成，是JVM实现的一部分。  
  
      **(2) 扩展类装载器和自定义类装载器 ：**负责除核心Java API以外的其它class文件的装载。例如、用于安装或下载标准扩展的class文件,在类路径中发现的类库的class文件,用于应用程序运行的class文件等等。这里有一点需要注意：自定义类装载器并非由应用程序员自己实现，它也是JVM  
   
      **(3) 命名空间:**Java虚拟机为每一个类装载器维护一个唯一标识的命名空间。一个Java程序可以多次装载具有同一个全限定名的多个类。 Java虚拟机要确定这"多个类"的唯一性，因此,当多个类 装载器都装载了同名的类时，为了唯一地标识这个类,还要在类名前加上装载该类的类装载器的标识(指出了类所位于的命名空间)。下图显示了两个类装载器有关的命名空间，显然，不同的类装载器允许装载相同的类Volcano。   
      命名空间有助于安全的实现，因为你可以有效地在装入了不同命名空间的类之间设置一个防护罩。在Java虚拟机中，在同一个命名空间内的类可以直接进行交 互，而不同的命名空间中的类甚至不能察觉彼此的存在，除非显式地提供了允许它们进行交互的机制。一旦加载后，如果一个恶意的类被赋予权限访问其他虚拟机加 载的当前类，它就可以潜在地知道一些它不应该知道的信息，或者干扰程序的正常运行。  
   
   
**3、双亲委托模型**  
   
      用户自定义类装载器经常依赖其他类装载器——至少依赖于虚拟机启动时创建的启动类装载器—来帮助它实现一些类装载请求:.在版本1.2前，非启动类装载器 必须显式地求助于其他类装载器，类装载器可以请求另一个用户自定义的类装载器来装载一个类，这个请求是通过对被请求的用户自定义类装载器调用 loadClass()来实现的。除此以外，类装载器也可以通过调用findSystemClass()来请求启动类装载器来装载类，这是类 ClassLoader中的一个静态方法。  
      在版本1.2中，类装载器请求另一个类装载器来装载类型的过程被形式化，称为双亲委派模式 。  
      从版本1.2开始、除启动类装载器以外的每一个类装载器，都有一个“双亲”类装载器 ，在某个特定的类装载器试图以常用方式装载类型以前，它会先默认地将这个任务“委派”给它的双亲——清求它的双亲来装载这个类型。这个双亲再依次请求它自 己的双亲来装载这个类型。这个委派的过程一直向上继续，直到达到启动类装载器，通常启动类装载器是委派链中的最后一个类装载器。如果一个类装载器的双亲类 装载器有能力来装载这个类型。则这个类装载器返回这个类型。否则，这个类装载器试图自己来装载这个类。  
   
     当Java虚拟机开始运行时,在应用程序开始启动以前,它至少创建一个用户自定义装载器,也可能创建多个.所有这些装载器被连接在一个Parent-Child的委托链中,在这个链的顶端是启动类装载器。  
   
    例如：假设你写了一个应用程序,在虚拟机上运行它.虚拟机在启动时实例化了两个用户自定义类装载器:一个"扩展类装载器",一个"类路径类装载器".这些类装载器和启动类装载器一起联入一个Parent-Child委托链中,如下图所示.  
   
  
       上图所示类路径类装载器的Parent是扩展类装载器, 扩展类装载器的Parent是启动类装载器.在图2中,类路径类装载器就被实例为系统类装载器.假设你的程序实例化它的网络类装载器,它就指明了系统类装载器作为它的Parent.  
下面的例程说明了类装载器的父子关系.

**Java代码  收藏代码**

1. **package** test;
2. **import** java.net.URL;
3. **import** java.net.URLClassLoader;
4. **public** **class** ClassLoaderTest {
5. **private** **static** **int** count = -1;
6. **public** **static** **void** testClassLoader(Object obj) {
7. **if** (count < 0 && obj == **null**) {
8. System.out.println("Input object is NULL";
9. **return**;
10. }
11. ClassLoader cl = **null**;
12. **if** (obj != **null** && !(obj **instanceof** ClassLoader)) {
13. cl = obj.getClass().getClassLoader();
14. } **else** **if** (obj != **null**) {
15. cl = (ClassLoader) obj;
16. }
17. count++;
18. String parent = "";
19. **for** (**int** i = 0; i < count; i++) {
20. parent += "Parent ";
21. }
22. **if** (cl != **null**) {
23. System.out.println(
24. parent + "ClassLoader name = " + cl.getClass().getName());
25. testClassLoader(cl.getParent());
26. } **else** {
27. System.out.println(
28. parent + "ClassLoader name = BootstrapClassLoader";
29. count = -1;
30. }
31. }
32. **public** **static** **void** main(String[] args) {
33. URL[] urls = **new** URL[1];
34. URLClassLoader urlLoader = **new** URLClassLoader(urls);
35. ClassLoaderTest.testClassLoader(urlLoader);
36. }
37. }

以上例程的输出为:  
ClassLoader name = java.net.URLClassLoader  
Parent ClassLoader name = sun.misc.Launcher＄AppClassLoader  
Parent Parent ClassLoader name = sun.misc.Launcher＄ExtClassLoader  
Parent Parent Parent ClassLoader name = BootstrapClassLoader  
   
   
类装载器请求过程  
以上例程1为例.将main方法改为:  
        ClassLoaderTest tc = new ClassLoaderTest();  
        ClassLoaderTest.testClassLoader(tc);  
输出为:  
ClassLoader name = sun.misc.Launcher＄AppClassLoader  
Parent ClassLoader name = sun.misc.Launcher＄ExtClassLoader  
Parent Parent ClassLoader name = BootstrapClassLoader  
   
     程序运行过程中，类路径类装载器发出一个装载ClassLoaderTest类的请求， 类路径类装载器必须首先询问它的Parent---扩展类装载器 ---来查找并装载这个类,同样扩展类装载器首先询问启动类装载器。由于ClassLoaderTest不是 Java API(JAVA\_HOME\jre\lib)中的类，也不在已安装扩展路径(JAVA\_HOME\jre\lib\ext)上，这两类装载器 都将返回而不会提供一个名为ClassLoaderTest的已装载类给类路径类装载器。类路径类装载器只能以它自己的方式来装载 ClassLoaderTest，它会从当前类路径上下载这个类。这样，ClassLoaderTest就可以在应用程序后面的执行中发挥作用。  
      在上例中，ClassLoaderTest类的testClassLoader方法被首次调用，该方法引用了Java API中的类 java.lang.String。Java虚拟机会请求装载ClassLoaderTest类的类路径类装载器来装载 java.lang.String。就像前面一样，类路径类装载器首先将请求传递给它的Parent类装载器，然后这个请求一路被委托到启动类装载器。但 是，启动类装载器可以将java.lang.String类返回给类路径类装载器，因为它可以找到这个类,这样扩展类装载器就不必在已安装扩展路径中查找 这个类，类路径类装载器也不必在类路径中查找这个类。扩展类装载器和类路径类装载器仅需要返回由启动类装载器返回的类java.lang.String。 从这一刻开始，不管何时ClassLoaderTest类引用了名为java.lang.String的类，虚拟机就可以直接使用这个 java.lang.String类了。

**4、一个经典的实例说明**

我们看看下面的代码：

**Java代码  收藏代码**

1. **package** java.lang;
3. **public** **class** String {
4. **public** **static** **void** main(String[] args){
6. }
7. }

       大家发现什么不同了吗？对了，我们写了一个与JDK中String一模一样的类，连包java.lang都一样，唯一不同的是我们自定义的String类有一个main函数。我们来运行一下：

                     java.lang.NoSuchMethodError: main  
                     Exception in thread "main"

这是为什么? 我们的String类不是明明有main方法吗？

其实联系我们上面讲到的双亲委托模型，我们就能解释这个问题了。

      运行这段代码，JVM会首先创建一个自定义类加载器，不妨叫做AppClassLoader，并把这个加载器链接到委托链中：AppClassLoader -> ExtClassLoader -> BootstrapLoader。

      然后AppClassLoader会将加载java.lang.String的请求委托给ExtClassLoader，而 ExtClassLoader又会委托给最后的启动类加载器BootstrapLoader。

      启动类加载器BootstrapLoader只能加载JAVA\_HOME\jre\lib中的class类(即J2SE API)，问题是标准API中确实有一个java.lang.String(注意，这个类和我们自定义的类是完全两个类)。BootstrapLoader以为找到了这个类，毫不犹豫的加载了j2se api中的java.lang.String。

      最后出现上面的加载错误(注意不是异常，是错误，JVM退出)，因为API中的String类是没有main方法的。

结论：我们当然可以自定义一个和API完全一样的类，但是由于双亲委托模型，使得我们不可能加载上我们自定义的这样一个类。所以J2SE规范中希望我们自定义的包有自己唯一的特色(网络域名)。还有一点，这种加载器原理使得JVM更加安全的运行程序，因为黑客很难随意的替代掉API中的代码了。