## ****类加载器的概述分类：****

**类的加载时机：我们前面提到，所有的类或接口都是在*Java*虚拟机首次“主动使用”的时候才会初始化，类从*class*文件到内存经过了“加载”、“连接”、“初始化”，类是在首次主动使用的时候进行初始化，那他是什么时候加载的呢？**

**类的加载并不需要到该类或接口被首次主动使用的时候才加载，*JVM*规范允许类加载器在预料到某个类将要被使用时就预先加载它，如果在预先加载工程中遇到了*.class*文件缺失或存在错误，则类加载器必须在程序主动使用该类的时候 报告*LinkageError*错误，如果这个类一直没有被程序主动使用，则该类加载器不会报告错误！即使不存在这个类也不会报告错误！**

**这时候不是异常，是错误哦，看清楚了哦！**

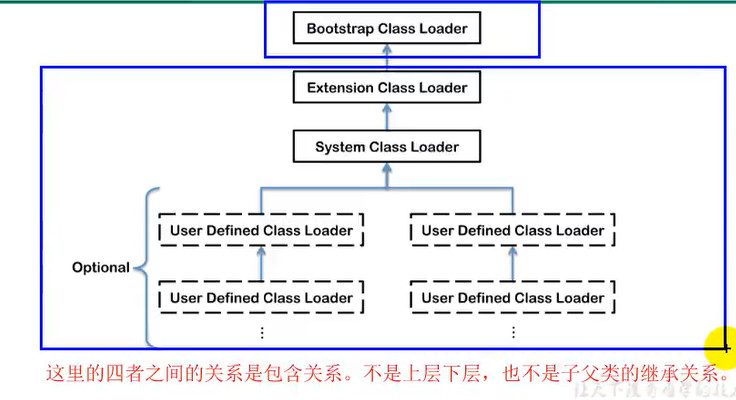
**一般我们都是和*Exception*打交道，和*Error*打交道不太多！打不是*Error*是由*JVM*；来处理的！我们来看一下*LinkageError*这个类的*API*。**

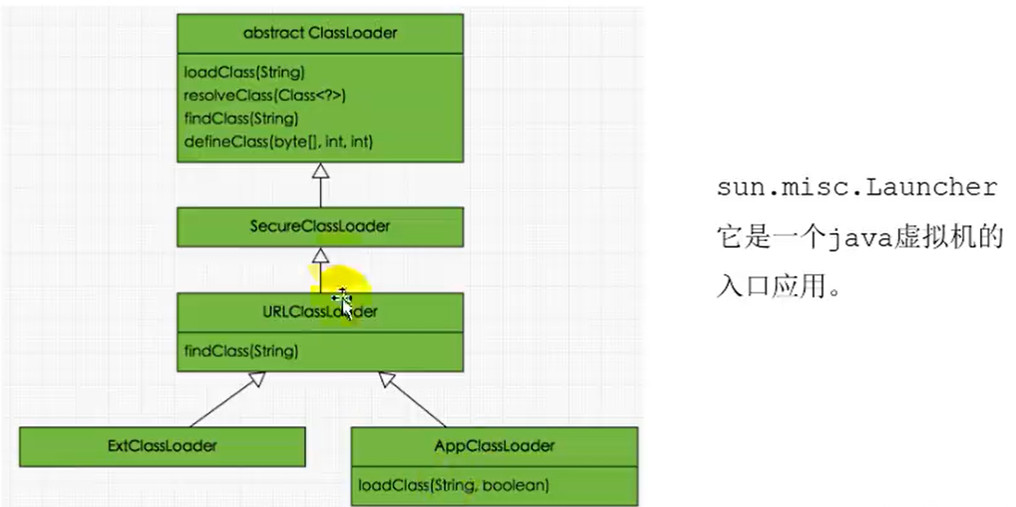
*car*调用了*car*类定义的*run*方法，而*worker*类中是没有*car*的*run*的，因此在运行的时候*JVM*把这个有一个符号引用替换为一个指针（这里指替换为真正的由*C++*底层实现的指针），而**我们*java*里面看到的就是符号引用，实际*C++*执行的才是指针，我们称之为直接引用**！

JVM支持两种类加载器：

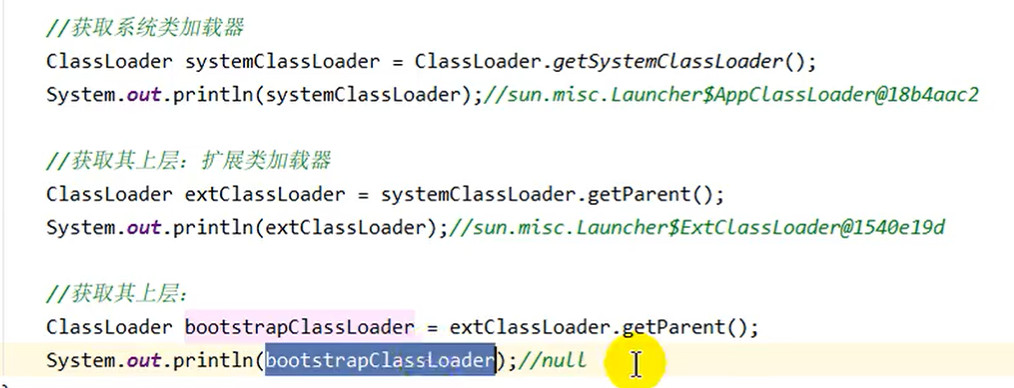
1、引导类加载器（Bootstrap ClassLoader）**--使用的是C与C++进行实现的**：是虚拟机自带的加载器，我们是拿不到它的实例的。

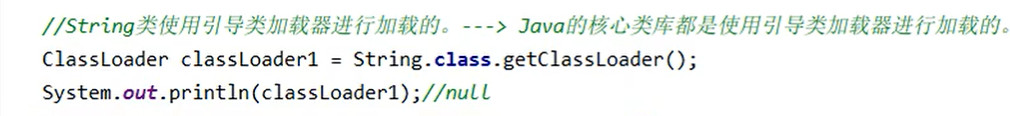
2、自定义加载器（User-Defined ClassLoader）**--使用的是Java实现的**：一般指得是程序中由开发人员自定义的一类类加载器，但是Java虚拟机规范却没有这么定义，而是**将所有派生于抽象类CLassLoader的类加载器都划分为自定义类加载器。（Java虚拟机规范的定义更加广义，故扩展类加载器于系统类加载器都是属于自定义类加载器）**

0



**（再次强调：不是继承关系，是上下层的关系，是包含关系）**





**最常见的类加载器（3个）：引导类加载器、扩展类加载器、系统类加载器**

## 各种类加载器的具体介绍：

（[深入剖析Classloader(二)--根类加载器，扩展类加载器与系统类加载器\_zhouyong0的专栏-CSDN博客\_扩展类加载器加载哪些类](https://blog.csdn.net/zhouyong0/article/details/7933625)这篇文章见解挺独道的）

**JVM自带的加载器：（从下方的加载器可知：每一类加载器其实都是由自己负责的块或者层次）**

**（再强调：**

**引导类加载器会加载扩展类加载器，**

**扩展类加载器会加载系统类加载器，**

**系统类加载器会加载用户自定义类-用户自定义加载器也是用户自定义类**

**他们之间不是继承关系，仅仅只是因为被上者加载，*也说明不同的类加载器所处块或者说层次是不同的***

**）**

1、**引导类加载器**（根加载器、启动类加载器BootStrap ClassLoader，可以说它就是JVM的一部分）（负责Java核心块）

特点：

1.该类加载器是使用C/C++编写实现的，嵌套在JVM内部。

2.它用来加载Java的核心库（JAVA\_HOME/jre/lib/jrt,jar、resource.jar、sun.boot.class.path路径下的内容，用于提供JVM自身需要的类。）

3.并不继承自java.lang.ClassLoader，没有父加载器。

4.加载扩展类和应用程序类加载器（系统类加载器），并指定为他们的父类加载器。（扩展类加载器与系统类加载器其实也是Java类，所以也需要加载，而他们就是使用引导类加载器加载的）

5.处于安全考虑，BootStrap启动类加载器只加载包含名为java、javax、sun等开头的类。

2、**扩展类加载器**（Extension ClassLoader，加载核心类外的扩展类）（负责扩展块）

1.Java语言编写，由sum.misc.Launcher$ExtClassLoader实现。

2.派生于ClassLoader类

3.父类加载器为启动类加载器

4.从java.ext.dirs系统属性所指定的目录中加载类库，或者从JDK的安装目录的jre/lib/ext子目录（扩展目录）下加载类库。如果用户创建的JAR放在此目录下，也会自动扩展类加载器加载。

3、**应用程序类加载器**（系统类加载器AppClassLoader）（负责自定义块）

1.Java语言编写，由sun.misc.Launcher$AppClassLoader实现

2.派生于ClassLoader类

3.父类加载器为扩展类加载器

4.**它负责加载环境变量classpath(自定义的类、第三方Jar包)或系统属性java.class.path指定路径下的类库**

5.**该类加载器是程序中默认的类加载器，一般来说，Java应用的类都是由它来完成加载**

6,通过Class Loader#getSystemClassLoader()方法可以获取到该类加载器。

看看引导类加载器能加载的路径：



看下扩展类加载器能加载的路径：



**用户自定义类加载器：**

1.在Java日常应用程序开发中，类的加载几乎是由前三种类加载器相互配合执行的，在必要时，我们还可以自定义类加载器，来定制类的加载方式。

2.为什么需要自定义加载器（在什么情况下）？

（[为什么要自定义类加载器？\_NSX-Truth-CSDN博客\_为什么自定义类加载器](https://blog.csdn.net/nsx_truth/article/details/109144191)）

1.**隔离加载类**：在某个应用中需要使用中间件，这个中间件有自己的依赖的jar包，在同一个工程里面，如果引用多个框架的话，有可能会出现某些类的路径一样、类名也相同，这样就会出现类的冲突了，这个时候就需要做一个类的仲裁，像现在主流的容器类的框架一样，它们都会自定义类的加载器，实现不同的中间件隔离，避免类的冲突

2.**修改类加载的方式**：在整个类的加载过程中，bootstrap引导类加载器是一定被使用的，用来加载系统需要的核心API，除了bootstrap引导类加载器之外，其他的类加载器也不是必须的，我们可以根据实际情况中修改类的加载方式，具体要用的时候我们再引用

3.**扩展加载源**：加载的类除了可以在网络、本地物理磁盘、jar包去加载之外，我们还可以考虑通过数据库、电视机的机顶盒等等来扩展加载源

4.**防止源码泄露**：当有了字节码文件或者没有反编译的手段，java代码是很容易被编译和篡改，所以，为了防止编译和篡改，我们可以对字节码文件进行加密，当我们需要运行这个字节码文件时候，我们需要解密来还原成内存中的类，而这个解密的操作，就需要自定义类的加载器来实现

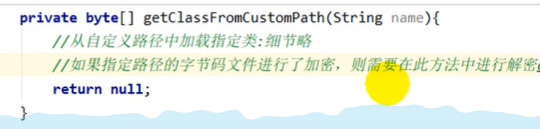
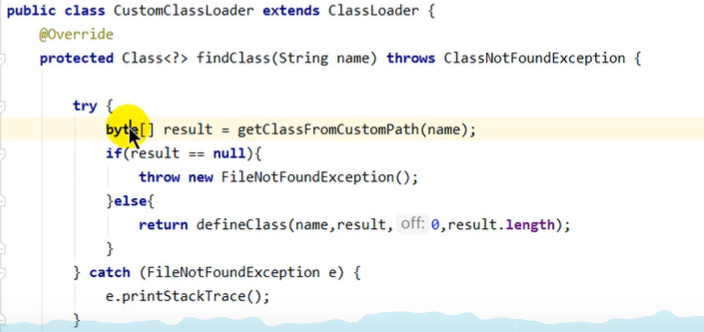
**用户自定义加载器实现步骤：**

1、开发人员可以通过继承抽象java.lang.ClassLoader类的方式，实现自己的类加载器，以满足一些特殊需求。

2、在JDK1.2之前，在自定义类加载器时，总会去继承ClassLoader类并重写loadClass()方法，从而实现自定义的类加载器，但是JDK1.2以后已不再建议重写loadClass()，而是建议把自定义的类加载逻辑卸载findClass()方法中

3、在编写自定义类加载器时，如果没有太过于复杂的需求，可以直接继承URLClassLoader类，这样就可以避免自己去编写findClass()方法及其获取字节码流的方式，使得自定义类加载器编写更加简洁。

示例：（将指定路径的Class文件以二进制形式读入内存）



## 关于ClassLoader

1、ClassLoader类，他是一个抽象类，其后所有的了加载器都继承自ClassLoader(除了启动类加载器)



几种获取ClassLoader(加载器)的方式：

1、获取当前类的ClassLoader：clazz（类名 ）.getClassLoader();

2、获取当前线程上下文的ClassLoader:Thread.currentThread().getContextClassLoader();

3、获取系统的ClassLoader：ClassLoader.getSystemClassLoader();（获取系统类加载器）

4、获取调用者的ClassLoader：DriverManager.getCallerClassLoader()；