JVM的类加载ClassLoader机制

做Java开发，对于**ClassLoader的机制**是必须要熟悉的基础知识，本文针对Java ClassLoader的机制做一个简要的总结。因为不同的JVM的实现不同，本文所描述的内容均只限于**Hotspot JVM**。

本文将会从**JDK默认的提供的ClassLoader**，**双亲委托模型**，如何**自定义ClassLoader**以及**Java中打破双亲委托机制的场景**四个方面入手去讨论和总结一下。

# ClassLoader的介绍

## 作用

ClassLoader主要对**类的请求**提供服务，当JVM需要某类时，它根据名称向**ClassLoader**要求这个类，然后由ClassLoader返回这个类的class对象。

ClassLoader负责载入系统的所有Resources（Class，文件，来自网络的字节流 等），通过ClassLoader从而将资源载入JVM。每个class都有一个reference，指向自己的ClassLoader。Class.getClassLoader()。array的ClassLoader就是其元素的ClassLoader，若是基本数据类型，则这个array没有ClassLoader 。

## ClassLoader的重要方法介绍

**loadClass**是 ClassLoader 的入口点；

**defineClass**方法是 ClassLoader 的主要诀窍。该方法接受由**原始字节组成的数组**并把它转换成 **Class 对象**。原始数组包含如从**文件系统或网络**装入的数据。 defineClass是final修饰，不允许被覆盖。

**findSystemClass** 方法从本地文件系统装入文件。它在本地文件系统中寻找类文件，如果存在，就使用 defineClass 将原始字节转换成 Class 对象，以将该文件转换成类。**当运行 Java 应用程序时，这是 JVM 正常装入类的缺省机制**。

**resolveClass**可以不完全地（不带解析）装入类，也可以完全地（带解析）装入类。当编写我们自己的 loadClass 时，可以调用 resolveClass，这取决于 loadClass 的 resolve 参数的值 。

**findLoadedClass** 充当一个缓存：当请求 loadClass 装入类时，它调用该方法来查看 ClassLoader 是否已装入这个类，这样可以避免重新装入已存在类所造成的麻烦。应首先调用该方法 。

## 一般load方法过程如下：

### 调用 findLoadedClass 来查看是否存在已装入的类；

### 如果没有，那么采用某种方式来获取原始字节。（通过IO从文件系统，来自网络的字节流等）

### 如果已有原始字节，调用 defineClass 将它们转换成 Class 对象。

### 如果没有原始字节，然后调用 findSystemClass 查看是否从本地文件系统获取类。

### 如果 resolve 参数是 true，那么调用 resolveClass 解析 Class 对象。

### 如果还没有类，返回 ClassNotFoundException。

### 否则，将类返回给调用程序。

# JDK默认ClassLoader

JDK 默认提供了如下几种ClassLoader

## Bootstrp loader

Bootstrp加载器是用**C++语言**写的，它是在Java虚拟机启动后初始化的，它主要负责加载**%JAVA\_HOME%/jre/lib,-Xbootclasspath**参数指定的路径以及**%JAVA\_HOME%/jre/classes**中的类。

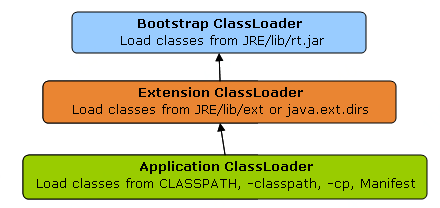
## ExtClassLoader

Bootstrp loader加载**ExtClassLoader**,并且将ExtClassLoader的父加载器设置为Bootstrp loader.ExtClassLoader是用Java写的，具体来说就是 **sun.misc.Launcher$ExtClassLoader**，ExtClassLoader主要加载%JAVA\_HOME%/jre/lib/ext，此路径下的所有classes目录以及java.ext.dirs系统变量指定的路径中类库。

## AppClassLoader

Bootstrp loader加载完ExtClassLoader后，就会加载**AppClassLoader**,并且将AppClassLoader的父加载器指定为 **ExtClassLoader**。AppClassLoader也是用Java写成的，它的实现类是 sun.misc.Launcher$AppClassLoader，另外我们知道ClassLoader中有个**getSystemClassLoader**方法,**此方法返回的正是AppclassLoader。**AppClassLoader主要负责加载classpath所指定的位置的类或者是jar文档，**它也是Java程序默认的类加载器**。

综上所述，它们之间的关系可以通过下图形象的描述：



双亲委派模型自从JDK1.2以后，ClassLoader做了改进，使用了双亲委派模型，所有系统中的**ClassLoader**组成一棵树，ClassLoader在载入类库时先让Parent寻找，Parent找不到才自己找。

JVM 在运行时会产生三个ClassLoader：**Bootstrap** ClassLoader、**Extension** ClassLoader和 **App** ClassLoader。其中，Bootstrap ClassLoader是用C++编写的，在Java中看不到它，是null。它用来加载核心类库，就是在lib下的类库，Extension ClassLoader加载lib/ext下的类库，**App** ClassLoader加载 Classpath里的类库，三者的关系为:App ClassLoader的Parent是Extension ClassLoader，而 Extension ClassLoader的Parent为Bootstrap ClassLoader。加载一个类时，首先BootStrap进行寻找，找不到再由Extension ClassLoader寻找，最后才是App ClassLoader。

# 获得ClassLoader的几种方法

可以通过如下3种方法得到ClassLoader

### this.getClass.getClassLoader(); // 使用当前类的ClassLoader

### Thread.currentThread().getContextClassLoader(); // 使用当前线程的ClassLoader

### ClassLoader.getSystemClassLoader(); // 使 用系统ClassLoader，即系统的入口点所使用的ClassLoader。

（注意，system ClassLoader与根 **ClassLoader**并不一样。JVM下system ClassLoader通常为App **ClassLoader**）

# 双亲委托模型

Java中ClassLoader的加载采用了**双亲委托机制**，采用双亲委托机制加载类的时候采用如下的几个步骤：

当前ClassLoader首先从自己已经加载的类中查询是否此类已经加载，如果已经加载则直接返回原来已经加载的类。每个类加载器都有自己的加载缓存，当一个类被加载了以后就会放入缓存，等下次加载的时候就可以直接返回了。当前classLoader的缓存中没有找到被加载的类的时候，**委托父类加载器去加载**，父类加载器采用同样的策略，首先查看自己的缓存，然后委托**父类的父类**去加载，一直到Bootstrp ClassLoader.

当所有的父类加载器都没有加载的时候，再由当前的类加载器加载，并将其放入它自己的缓存中，以便下次有加载请求的时候直接返回。

说到这里大家可能会想，**Java为什么要采用这样的委托机制**？

理解这个问题，我们引入另外一个关于Classloader的概念“**命名空间**”， 它是指要确定某一个类，需要**类的全限定名以及加载此类的ClassLoader来共同确定**。**也就是说即使两个类的全限定名是相同的，但是因为不同的 ClassLoader加载了此类，那么在JVM中它是不同的类。**明白了命名空间以后，我们再来看看委托模型。采用了委托模型以后加大了不同的 ClassLoader的交互能力，比如上面说的，我们JDK本生提供的类库，比如hashmap,linkedlist等等，这些类由bootstrp 类加载器加载了以后，无论你程序中有多少个类加载器，那么这些类其实都是可以共享的，这样就避免了不同的类加载器加载了**同样名字的不同类以后造成混乱**。

将 ClassLoader设计成双亲委派模型的一个重要原因是出于**安全考虑**，比如在Applet中，如果编写了一个java.lang.String类并具有破坏性。假如不采用这种委托机制，就会将这个具有破坏性的String加载到了用户机器上，导致破坏用户安全。但采用这种委托机制则不会出现这种情况。因为要加载java.lang.String类时，系统最终会由Bootstrap进行加载，这个具有破坏性的String永远没有机会加载。

# 线程上下文类加载器ContextClassLoader

Thread的**getContextClassLoader**

**ClassLoader** getContextClassLoader()

Returns the **context ClassLoader** for this Thread.

void **setContextClassLoader**(ClassLoader cl)

Sets the context ClassLoader for this Thread.

线 程中的ClassLoader每个运行中的线程都有一个成员contextClassLoader，用来在运行时动态地载入其它类，可以使用方法 Thread.currentThread().setContextClassLoader(...);更改当前线程的 contextClassLoader，来改变其载入类的行为；也可以通过方法 Thread.currentThread().getContextClassLoader()来获得当前线程的ClassLoader。

实际上，在Java应用中所有程序都运行在线程里，如果在程序中没有手工设置过ClassLoader，对于一般的java类如下两种方法获得的ClassLoader通常都是同一个 。

this.getClass.getClassLoader()；

Thread.currentThread().getContextClassLoader()；

方 法一得到的Classloader是不变(静态)的，表明类的载入者是谁；方法二得到的Classloader是动态的，谁执行（某个线程），就是那个执行者的 Classloader。对于单例模式的类，静态类等，载入一次后，这个实例会被很多程序（线程）调用，对于这些类，载入的Classloader和执行线程的Classloader通常都不同。

# 自定义类加载器ClassLoader

## 自定义类加载器的应用

**安全性**。类进入JVM之前先经过ClassLoader，所以可以在这边检查是否有正确的数字签名等；

**加密**。java字节码很容易被反编译，通过定制ClassLoader使得字节码先加密防止别人下载后反编译，这里的ClassLoader相当于一个动态的解码器；

**归档**。可能为了节省网络资源，对自己的代码做一些特殊的归档，然后用定制的ClassLoader来解档；

**自展开程序**。把java应用程序编译成单个可执行类文件，这个文件包含压缩的和加密的类文件数据，同时有一个固定的ClassLoader，当程序运行时它在内存中完全自行解开，无需先安装；

**动态生成**。可以生成应用其他还未生成类的类，实时创建整个类并可在任何时刻引入JVM 。

## 如何自定义ClassLoader

Java除了上面所说的默认提供的**classloader**以外，它还容许应用程序可以自定义classloader，那么要想**自定义classloader**我们需要通过**继承java.lang.ClassLoader**来实现,接下来我们就来看看再自定义Classloader的时候，我们需要注意的几个重要的方法：

### loadClass 方法

loadClass method declare

public Class<?> loadClass(String name)  throws ClassNotFoundException

上面是loadClass方法的原型声明，上面所说的**双亲委托机制的实现其实就实在此方法中实现的**。下面我们就来看看此方法的代码来看看它到底如何实现双亲委托的。

loadClass method implement

public Class<?> **loadClass**(String name) throws ClassNotFoundException {

return loadClass(name, false);

}

从上面可以看出**loadClass**方法调用了loadClass(name,false)方法，那么接下来我们再来看看另外一个loadClass方法的实现。

Class loadClass(String name, boolean resolve)

protected synchronized Class<?> **loadClass**(String name, boolean resolve) throws ClassNotFoundException {

// First, check if the class has already been loaded

Class c = findLoadedClass(name);

//检查class是否已经被加载过了

if (c == null) {

try {

if (parent != null) {

c = parent.loadClass(name, false); //如果没有被加载，且指定了父类加载器，则委托父加载器加载。

} else {

c = findBootstrapClass0(name);//如果没有父类加载器，则委托bootstrap加载器加载 }

} catch (ClassNotFoundException e) {

// If still not found, then invoke findClass in order

// to find the class.

c = **findClass(name);//如果父类加载没有加载到，则通过自己的findClass来加载。**

}

}

if (resolve) {

resolveClass(c);

}

return c;

}

上面的代码，我加了注释通过注释可以清晰看出**loadClass的双亲委托机制**是如何工作的。 这里我们需要注意一点就是public Class<?> loadClass(String name) throws ClassNotFoundException**没有被标记为final**，也就意味着我们是可以override这个方法的，也就是说双亲委托机制是可以打破的。另外上面注意到有个findClass方法，接下来我们就来说说这个方法到底是搞末子的。

### findClass

我们查看**java.lang.ClassLoader**的源代码，我们发现findClass的实现如下：

protected Class<?> **findClass**(String name) throws **ClassNotFoundException** {

throw new ClassNotFoundException(name);

}

我们可以看出**此方法默认的实现是直接抛出异常，其实这个方法就是留给我们应用程序来override的**。那么具体的实现就看你的实现逻辑了，你可以从磁盘读取，也可以从网络上获取class文件的字节流，**获取class二进制**以后就可以交给defineClass来实现进一步的加载。defineClass我们再下面再来描述。 ok，通过上面的分析，我们可以得出如下结论：

我们在**写自己的ClassLoader的时候，如果想遵循双亲委托机制，则只需要override findClass**，内部调用defineClass实现。

### defineClass

我们首先还是来看看defineClass的源码：

protected **final** Class<?> **defineClass**(String name, byte[] b, int off, int len)

throws ClassFormatError{

return **defineClass**(name, b, off, len, null);

}

从上面的代码我们看出此方法被定义为了final，这也就意味着**此方法不能被Override**，**其实这也是JVM留给我们的唯一的入口，通过这个唯一的入口，jvm保证了类文件必须符合Java虚拟机规范规定的类的定义**。此方法最后会调用**native的方法**来实现真正的类的加载工作。

## 思考下面一个问题：

假如我们自己写了一个**java.lang.String**的类，我们是否可以替换调JDK本身的类？

答案是**否定的**。我们不能实现。为什么呢？我看很多网上解释是说**双亲委托机制解决**这个问题，其实不是非常的准确。因为双亲委托机制是可以打破的，你完全可以自己写一个classLoader来加载自己写的java.lang.String类，但是你会发现也不会加载成功，具体就是因为针对java.\*开头的类，Java的实现中已经保证了**必须由bootstrp来加载**。

# 不遵循“双亲委托机制”的场景

上面说了**双亲委托机制**主要是**为了实现不同的ClassLoader之间加载的类的交互问题**，**被大家公用的类**就交由**父加载器**去加载，但是J**ava中确实也存在父类加载器加载的类需要用到子加载器加载的类的情况**。下面我们就来说说这种情况的发生。

Java中有一个***SPI(Service Provider Interface)标准***,使用了SPI的库，比如JDBC，JNDI等，我们都知道**JDBC需要第三方提供的驱动**才可以，而驱动的jar包是放在我们应用程序本身的classpath的，而jdbc 本身的api是jdk提供的一部分，它已经被B**ootstrp**加载了，那第三方厂商提供的实现类怎么加载呢？这里面JAVA引入了**线程上下文类加载ContextClassLoader**的概念，**线程类加载器默认会从父线程继承，如果没有指定的话，默认就是系统类加载器（AppClassLoader），这样的话当加载第三方驱动的时候，就可以通过线程的上下文类加载器来加载**。

另外为了实现更灵活的类加载器OSGI以及一些Java app server也打破了双亲委托机制。

# Tomcat的ClassLoader机制

委托模型还带来了一些问题，在某些情况下会产生混淆，如下是Tomcat的ClassLoader结构图:

Bootstrap

|

System

|

Common

/

Catalina Shared

/

Webapp1 Webapp2 ...

由 Common 类加载器装入的类决不能（根据名称）直接访问由 **Web 应用程序装入的类**。使这些类联系在一起的唯一方法是通过使用这两个类集都可见的接口。在这个例子中，就是包含由 Java servlet 实现的 javax.servlet.Servlet。

如果在lib或者lib/ext等类库有与应用中同样的类，**那么应用中的类将无法被载入**。通常在jdk新版本出现有类库移动时会出现问题，例如最初我们使用自己的xml解析器，而在jdk1.4 中xml解析器变成标准类库，load的优先级也高于我们自己的xml解析器，我们自己的xml解析器永远无法找到，将可能导致我们的应用无法运行。

相同的类，不同的ClassLoader，将导致**ClassCastException**异常 。

Web应用中的ClassLoader回到上面的例子，在Tomcat 里，WebApp的ClassLoader的工作原理有点不同，它先试图自己载入类（在**ContextPath/WEB-INF/...中载入类**），如果无法载入，再请求父ClassLoader完成。

由此可得：

对于WEB APP线程，它的contextClassLoader是**WebAppClassLoader**

对于Tomcat Server线程，它的contextClassLoader是**CatalinaClassLoade**