**1. 什么是类加载机制？**

代码编译的结果从本地机器码转变成字节码，是存储格式的一小步，却是编程语言发展的一大步。

Java虚拟机把描述类的数据从Class文件加载进内存，并对数据进行校验，转换解析和初始化，最终形成可以呗虚拟机直接使用的Java类型，这就是虚拟机的类加载机制。

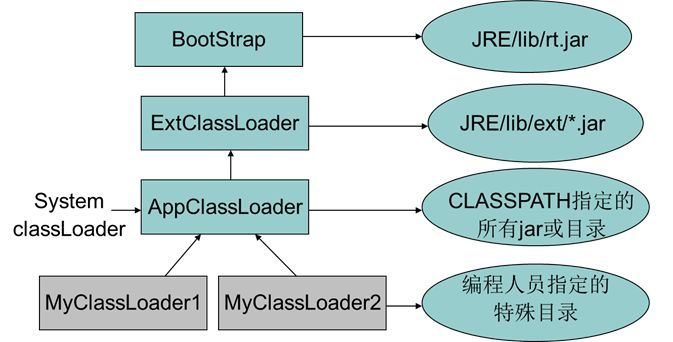
虚拟机设计团队把类加载阶段中的“**通过一个类的全限定名来获取描述此类的二进制字节流”这个动作放到Java虚拟机外部去实现**，以便让应用程序自己决定如何去获取所需要的类。实现这动作的代码模块成为“类加载器”。

**类与类加载器的关系**

类加载器虽然只用于实现类的加载动作，但它在Java程序中起到的作用却远远不限于类加载阶段。对于任意一个类，都需要由**加载他的类加载器和这个类本身一同确立其在Java虚拟机中的唯一性**，每一个类加载器，都拥有一个独立的类命名空间。这句话可以表达的更通俗一些：比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来自同一个Class文件，被同一个虚拟机加载，只要加载他们的类加载器不同，那这个两个类就必定不相等。

**2. 什么是双亲委任模型**

1. 从Java虚拟机的角度来说，只存在两种不同类加载器：一种是**启动类加载器(Bootstrap ClassLoader)**，这个类加载器使用C++语言实现（只限HotSpot），是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类加载器都由Java语言实现，独立于虚拟机外部，并且全都继承自抽象类java.lang.ClassLoader.
2. 从Java开发人员的角度来看，类加载还可以划分的更细致一些，绝大部分Java程序员都会使用以下3种系统提供的类加载器：
   * 启动类加载器（Bootstrap ClassLoader）：这个类加载器复杂将存放在 JAVA\_HOME/lib 目录中的，或者被-Xbootclasspath 参数所指定的路径种的，并且是虚拟机识别的（仅按照文件名识别，如rt.jar，名字不符合的类库即使放在lib目录下也不会重载）。
   * 扩展类加载器（Extension ClassLoader）：这个类加载器由sun.misc.Launcher$ExtClassLoader实现，它负责夹杂JAVA\_HOME/lib/ext 目录下的，或者被java.ext.dirs 系统变量所指定的路径种的所有类库。开发者可以直接使用扩展类加载器。
   * 应用程序类加载器（Application ClassLoader）：这个类加载器由sun.misc.Launcher$AppClassLoader 实现。由于这个类加载器是ClassLoader 种的getSystemClassLoader方法的返回值，所以也成为系统类加载器。它负责加载用户类路径（ClassPath）上所指定的类库。开发者可以直接使用这个类加载器，如果应用中没有定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

这些类加载器之间的关系一般如下图所示：   


图中各个类加载器之间的关系成为 **类加载器的双亲委派模型（Parents Dlegation Mode）**。**双亲委派模型要求除了顶层的启动类加载器之外，其余的类加载器都应当由自己的父类加载器加载，这里类加载器之间的父子关系一般不会以继承的关系来实现，而是都使用组合关系来复用父加载器的代码。**

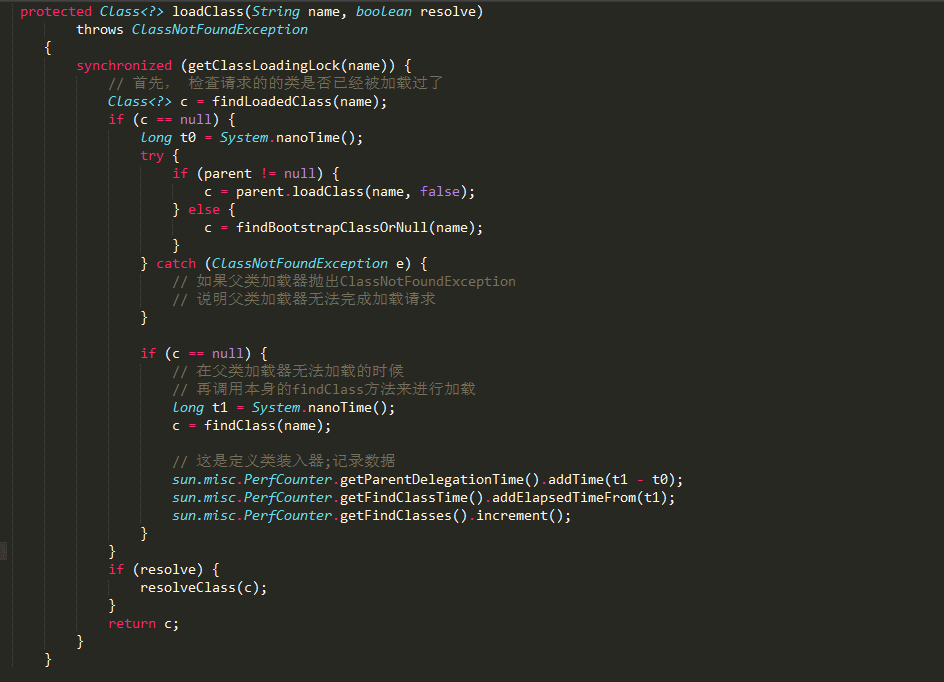
类加载器的双亲委派模型在JDK1.2 期间被引入并被广泛应用于之后的所有Java程序中，但他并不是个强制性的约束模型，而是Java设计者推荐给开发者的一种类加载器实现方式。

双亲委派模型的工作过程是：如果一个类加载器收到了类加载的请求，他首先不会自己去尝试加载这个类，而是把这个请求委派父类加载器去完成。每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个请求（他的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

**为什么要这么做呢？**

如果没有使用双亲委派模型，由各个类加载器自行加载的话，如果用户自己编写了一个称为java.lang.Object的类，并放在程序的ClassPath中，**那系统将会出现多个不同的Object类**， Java类型体系中最基础的行为就无法保证。应用程序也将会变得一片混乱。

**双亲委任模型时如何实现的？**

非常简单：所有的代码都在java.lang.ClassLoader中的loadClass方法之中，代码如下：   


逻辑清晰易懂：先检查是否已经被加载过，若没有加载则调用父加载器的loadClass方法， 如父加载器为空则默认使用启动类加载器作为父加载器。如果父类加载失败，抛出ClassNotFoundException 异常后，再调用自己的findClass方法进行加载。

**3. 如何破坏双亲委任模型？**

刚刚我们说过，双亲委任模型不是一个强制性的约束模型，而是一个建议型的类加载器实现方式。在Java的世界中大部分的类加载器都遵循者模型，但也有例外，到目前为止，双亲委派模型有过3次大规模的“被破坏”的情况。   
**第一次**：在双亲委派模型出现之前—–即JDK1.2发布之前。   
**第二次**：是这个模型自身的缺陷导致的。我们说，双亲委派模型很好的解决了各个类加载器的基础类的统一问题（越基础的类由越上层的加载器进行加载），基础类之所以称为“基础”，是因为它们总是作为被用户代码调用的API， 但没有绝对，**如果基础类调用回用户的代码**怎么办呢？

这不是没有可能的。一个典型的例子就是JNDI服务，JNDI现在已经是Java的标准服务，它的代码由启动类加载器去加载（在JDK1.3时就放进去的rt.jar）,但它需要调用由独立厂商实现并部署在应用程序的ClassPath下的JNDI接口提供者（SPI， Service Provider Interface）的代码，但启动类加载器不可能“认识“这些代码啊。因为这些类不在rt.jar中，但是启动类加载器又需要加载。怎么办呢？

为了解决这个问题，Java设计团队只好引入了一个不太优雅的设计：**线程上下文类加载器（Thread Context ClassLoader）**。这个类加载器可以通过java.lang.Thread类的**setContextClassLoader**方法进行设置。如果创建线程时还未设置，它将会从父线程中继承一个，如果在应用程序的全局范围内都没有设置过多的话，**那这个类加载器默认使用应用程序类加载器**。

嘿嘿，有了线程上下文加载器，JNDI服务使用这个线程上下文加载器去加载所需要的SPI代码，也就是**父类加载器请求子类加载器去完成类加载的动作**，这种行为实际上就是打通了双亲委派模型的层次结构来逆向使用类加载器，实际上已经违背了双亲委派模型的一般性原则。但这无可奈何，Java中所有涉及SPI的加载动作基本胜都采用这种方式。例如JNDI，**JDBC**，JCE，JAXB，JBI等。

**第三次**：为了实现热插拔，热部署，模块化，意思是添加一个功能或减去一个功能不用重启，只需要把这模块连同类加载器一起换掉就实现了代码的热替换。

书中还说到：

Java 程序中基本有一个共识：OSGI对类加载器的使用时值得学习的，弄懂了OSGI的实现，就可以算是掌握了类加载器的精髓。

牛逼啊！！！

现在，我们已经基本明白了Java默认的类加载的作用了原理，也知道双亲委派模型。说了这么多，差点把我们的tomcat给忘了，我们的题目是Tomcat 加载器为何违背双亲委派模型？下面就好好说说我们的tomcat的类加载器。

**4. Tomcat 的类加载器是怎么设计的？**

首先，我们来问个问题：

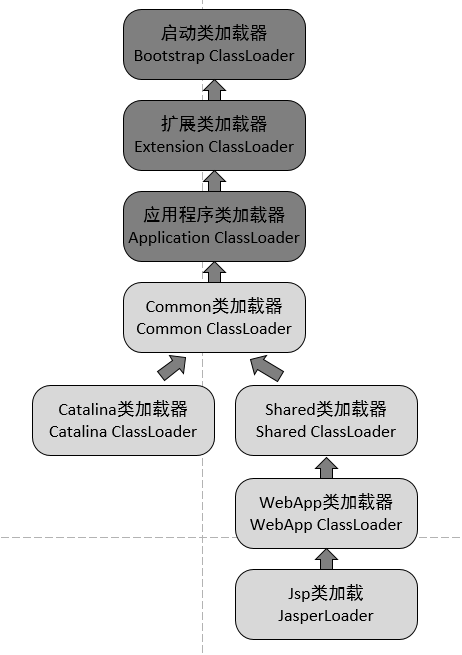
**Tomcat 如果使用默认的类加载机制行不行？**

我们思考一下：Tomcat是个web容器， 那么它要解决什么问题：   
1. **一个web容器可能需要部署两个应用程序，不同的应用程序可能会依赖同一个第三方类库的不同版本，不能要求同一个类库在同一个服务器只有一份，因此要保证每个应用程序的类库都是独立的，保证相互隔离。**2. 部署在同一个web容器中相同的类库**相同的版本可以共享**。否则，如果服务器有10个应用程序，那么要有10份相同的类库加载进虚拟机，这是扯淡的。   
3. web容器也有**自己依赖的类库，不能于应用程序的类库混淆**。基于安全考虑，应该让容器的类库和程序的类库隔离开来。   
4. **web容器要支持jsp的修改，**我们知道，jsp 文件最终也是要编译成class文件才能在虚拟机中运行，但程序运行后修改jsp已经是司空见惯的事情，否则要你何用？ 所以，**web容器需要支持 jsp 修改后不用重启。**

再看看我们的问题：Tomcat 如果使用默认的类加载机制行不行？   
答案是不行的。为什么？我们看，第一个问题，如果使用默认的类加载器机制，那么是**无法加载两个相同类库的不同版本**的，默认的累加器是不管你是什么版本的，只在乎你的全限定类名，并且只有一份。第二个问题，默认的类加载器是能够实现的，因为**他的职责就是保证唯一性**。第三个问题和第一个问题一样。我们再看第四个问题，我们想我们要怎么实现jsp文件的热修改（楼主起的名字），**jsp 文件其实也就是class文件，那么如果修改了，但类名还是一样，类加载器会直接取方法区中已经存在的，修改后的jsp是不会重新加载的。**那么怎么办呢？我们可以直接卸载掉这jsp文件的类加载器，所以你应该想到了，**每个jsp文件对应一个唯一的类加载器，当一个jsp文件修改了，就直接卸载这个jsp类加载器。**重新创建类加载器，重新加载jsp文件。

**Tomcat 如何实现自己独特的类加载机制？**

所以，Tomcat 是怎么实现的呢？牛逼的Tomcat团队已经设计好了。我们看看他们的设计图：



我们看到，前面3个类加载和默认的一致，**CommonClassLoader、CatalinaClassLoader、SharedClassLoader和WebappClassLoade**r则是**Tomcat自己定义的类加载器**，它们分别加载/common/\*、/server/\*、/shared/\*（在tomcat 6之后已经合并到根目录下的lib目录下）和/WebApp/WEB-INF/\*中的Java类库。其中WebApp类加载器和Jsp类加载器通常会存在多个实例，**每一个Web应用程序**对应一个WebApp类加载器，每一个JSP文件对应一个Jsp类加载器。

* commonLoader：Tomcat最基本的类加载器，加载路径中的class可以被Tomcat容器本身以及各个Webapp访问；
* catalinaLoader：Tomcat容器私有的类加载器，加载路径中的class对于Webapp不可见；
* sharedLoader：各个Webapp共享的类加载器，加载路径中的class对于所有Webapp可见，但是对于Tomcat容器不可见；
* WebappClassLoader：各个Webapp私有的类加载器，加载路径中的class只对当前Webapp可见；

从图中的委派关系中可以看出：

**CommonClassLoader能加载的类都可以被Catalina ClassLoader和SharedClassLoader使用，从而实现了公有类库的共用，**而CatalinaClassLoader和Shared ClassLoader自己能加载的类则与对方相互隔离。

**WebAppClassLoader可以使用SharedClassLoader加载到的类，但各个WebAppClassLoader实例之间相互隔离。**

而JasperLoader的加载范围仅仅是这个JSP文件所编译出来的那一个.Class文件，它出现的目的就是为了被丢弃：当**Web容器检测到JSP文件被修改时，会替换掉目前的JasperLoader的实例，并通过再建立一个新的Jsp类加载器来实现JSP文件的HotSwap功能**。

好了，至此，我们已经知道了tomcat为什么要这么设计，以及是如何设计的，那么，tomcat 违背了java 推荐的双亲委派模型了吗？答案是：违背了。 我们前面说过：

双亲委派模型要求除了顶层的启动类加载器之外，其余的类加载器都应当由自己的父类加载器加载。

很显然，tomcat 不是这样实现，tomcat 为了实现隔离性，没有遵守这个约定，**每个webappClassLoader加载自己的目录下的class文件，不会传递给父类加载器。**

**我们扩展出一个问题：如果tomcat 的 Common ClassLoader 想加载 WebApp ClassLoader 中的类，该怎么办？**

看了前面的关于破坏双亲委派模型的内容，我们心里有数了，我们可以使用**线程上下文类加载器实现**，使用线程上下文加载器，可以让父类加载器请求子类加载器去完成类加载的动作。牛逼吧。