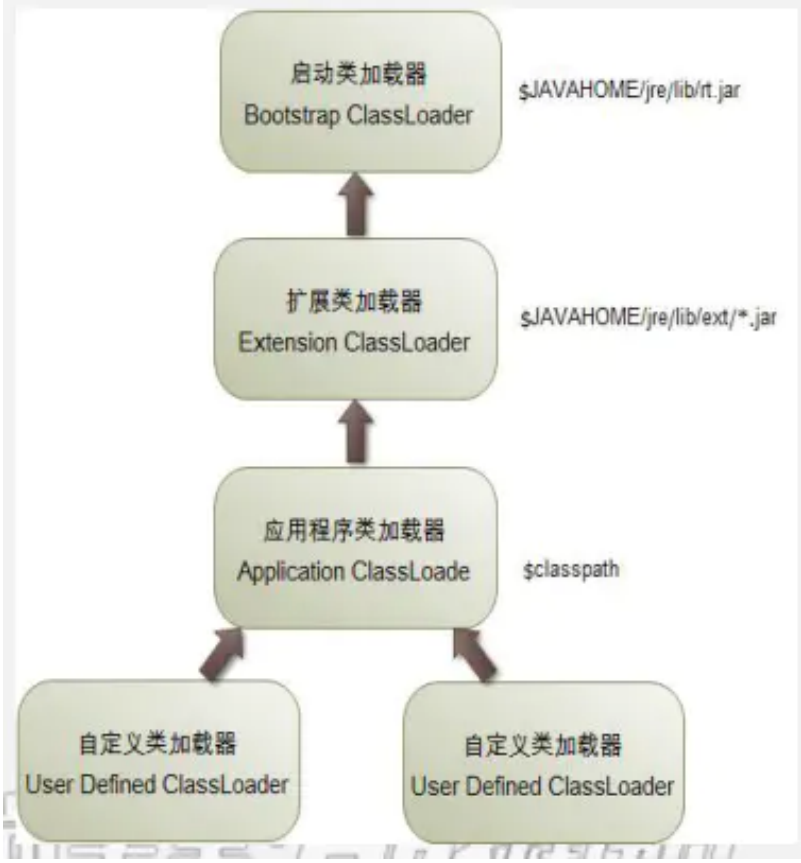
Java类加载器(ClassLoader)

类加载分类

在虚拟机提供了3种类加载器，**引导类加载器（Bootstrap ClassLoader）**、**扩展类加载器（Extension ClassLoader）**、**系统类加载器（System ClassLoader）**（也称**应用程序类加载器（Application ClassLoader）**），我们在程序里也可以自定义类加载器，下面分别介绍。



引导类加载器（Bootstrap ClassLoader）

- **使用C/C++语言实现**，在JVM内部。
- 它用来**加载 Java 的核心库**(JAVA_HOME/jre/lib/rt.jar、resources.jar或sun.boot.class.path路径下的内容)。
- **不继承** java.lang.ClassLoader，**没有父加载器**。
- 加载扩展类和应用程序类加载器。并指定他们的父类加载器。
- 出于安全考虑，Bootstrap启动类加载器**只加载包名为java、javax、sun等开头的类**。

加载内容如下

```
JAVA_HOME/jre/lib/resources.jar
JAVA_HOME/jre/lib/rt.jar
JAVA_HOME/jre/lib/sunrsasign.jar
JAVA_HOME/jre/lib/jsse.jar
JAVA_HOME/jre/lib/jce.jar
JAVA_HOME/jre/lib/charsets.jar
JAVA_HOME/jre/lib/jfr.jar
JAVA_HOME/jre/classes
```

扩展类加载器（启动类加载器，Extension ClassLoader）

- **Java语言编写**，由sun.misc.Launcher$ExtClassLoader实现。
- **派生于ClassLoader类**
- **父类加载器为引导类加载器**
- 从java.ext.dirs系统属性所指定的目录中加载类库，或从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，也会自动由扩展类加载器加载.

应用程序类加载器（系统类加载器，Application ClassLoader）

- **java语言编写**，由sun.misc.Launcher$AppClassLoader实现
- **派生于ClassLoader类**
- **父类加载器为扩展类加载器**
- 它**负责加载环境变量classpath或系统属性java.class.path指定路径下的类库**
- 该类加载是程序中**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

获取加载器加载内容方法如下

```
// 获取引导类加载器加载url
URL[] bootUrls = sun.misc.Launcher.getBootstrapClassPath().getURLs();
// 获取扩展类加载器加载url
URL[] extUrls = ((URLClassLoader)
ClassLoader.getSystemClassLoader().getParent()).getURLs();
// 获取系统类加载器加载url
URL[] systemUrls = ((URLClassLoader) ClassLoader.getSystemClassLoader()).getURLs();
```

自定义类加载器

为什么要自定义类加载器?

- **隔离加载类**
- 修改类加载的方式
- 扩展加载源
- **防止源码泄漏**

用户自定义类加载器实现步骤

- 通过**继承 java.lang.ClassLoader类**的方式实现自己的类加载器，以满足一些特殊的需求。
- 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类，但是在JDK1.2之后已不再建议用户去覆盖loadclass()方法，而是建议把自定义的类加载逻辑写在findclass()方法中
- 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，使自定义类加载器编写更加简洁。

ClassLoader

ClassLoader类，**它是一个抽象类**，其后所有的类加载器都继承自ClassLoader（**不包括启动类加载器**）

作用

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。

常用方法总结

- getParent()：返回该类加载器的父类加载器
- forName(String className)：通过类名称获取已经初始化的java.lang.Class类的实例
- loadClass(String name)：通过类名称加载类，返回java.lang.Class类的实例，该方法中的逻辑就是双亲委派模式的实现

  此方法负责加载指定名字的类，首先会从已加载的类中去寻找，如果没有找到；从parent ClassLoader[ExtClassLoader]中加载；如果没有加载到，则从Bootstrap ClassLoader中尝试加载(findBootstrapClassOrNull方法), 如果还是加载失败，则自己加载。如果还不能加载，则抛出异常ClassNotFoundException
- findClass(String name)：通过类名称查找类，返回java.lang.Class类的实例，需要自己实现
- findLoaderClass(String name)：通过类名称查找已经被加载的类，返回java.lang.Class类的实例
- defineClass(String name, byte[] b, int off, int len)：把字节数组b中的内容转换为一个Java类，返回java.lang.Class类的实例
- resolveClass(Class<?> c)：链接指定的一个Java类

方法源码

loadClass

```
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        // 先从缓存查找该class对象，找到就不用重新加载
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            long t0 = System.nanoTime();
```

```
                try {
                    if (parent != null) {
                        //如果找不到，则委托给父类加载器去加载
                        c = parent.loadClass(name, false);
                    } else {
                    //如果没有父类，则委托给启动加载器去加载
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    // ClassNotFoundException thrown if class not found
                    // from the non-null parent class loader
                }

                if (c == null) {
                    // If still not found, then invoke findClass in order
                    // 如果都没有找到，则通过自定义实现的findClass去查找并加载
                    c = findClass(name);

                    // this is the defining class loader; record the stats
                    sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);
                    sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
                    sun.misc.PerfCounter.getFindClasses().increment();
                }
            }
            if (resolve) {//是否需要在加载时进行解析
                resolveClass(c);
            }
            return c;
        }
    }
```

findClass

```
    //直接抛出异常
    protected Class<?> findClass(String name) throws ClassNotFoundException {
          throw new ClassNotFoundException(name);
    }
```

获取ClassLoader的途径

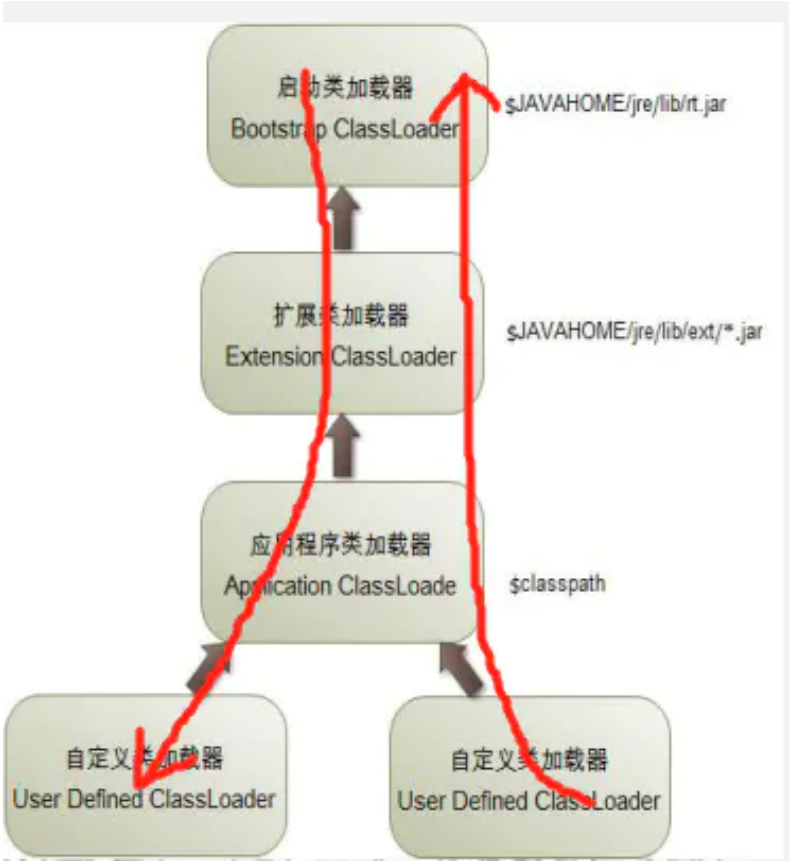
获取当前ClassLoader：class.getClassLoader()
获取当前线程上下文的ClassLoader：Thread.currentThread().getContextClassLoader()
获取系统的ClassLoader：ClassLoader.getSystemClassLoader()
获取调用者的ClassLoader：DriverManager.getCallerClassLoader()

类加载器的代理模式

代理模式即是将指定类的加载交给其他的类加载器。常用双亲委派机制。

双亲委派机制

某个特定的类加载器接收到类加载的请求时，会将加载任务委托给自己的父类，直到最高级父类**引导类加载器（bootstrap class loader）**，如果父类能够加载就加载，不能加载则返回到子类进行加载。如果都不能加载则报错。**ClassNotFoundException**



双亲委托机制是为了保证 Java 核心库的类型安全。这种机制保证不会出现用户自己能定义java.lang.Object类等的情况。例如，用户定义了java.lang.String，那么加载这个类时最高级父类会首先加载，发现核心类中也有这个类，那么就加载了核心类库，而自定义的永远都不会加载。

线程上下文类加载器

线程上下文类加载器出现的原因

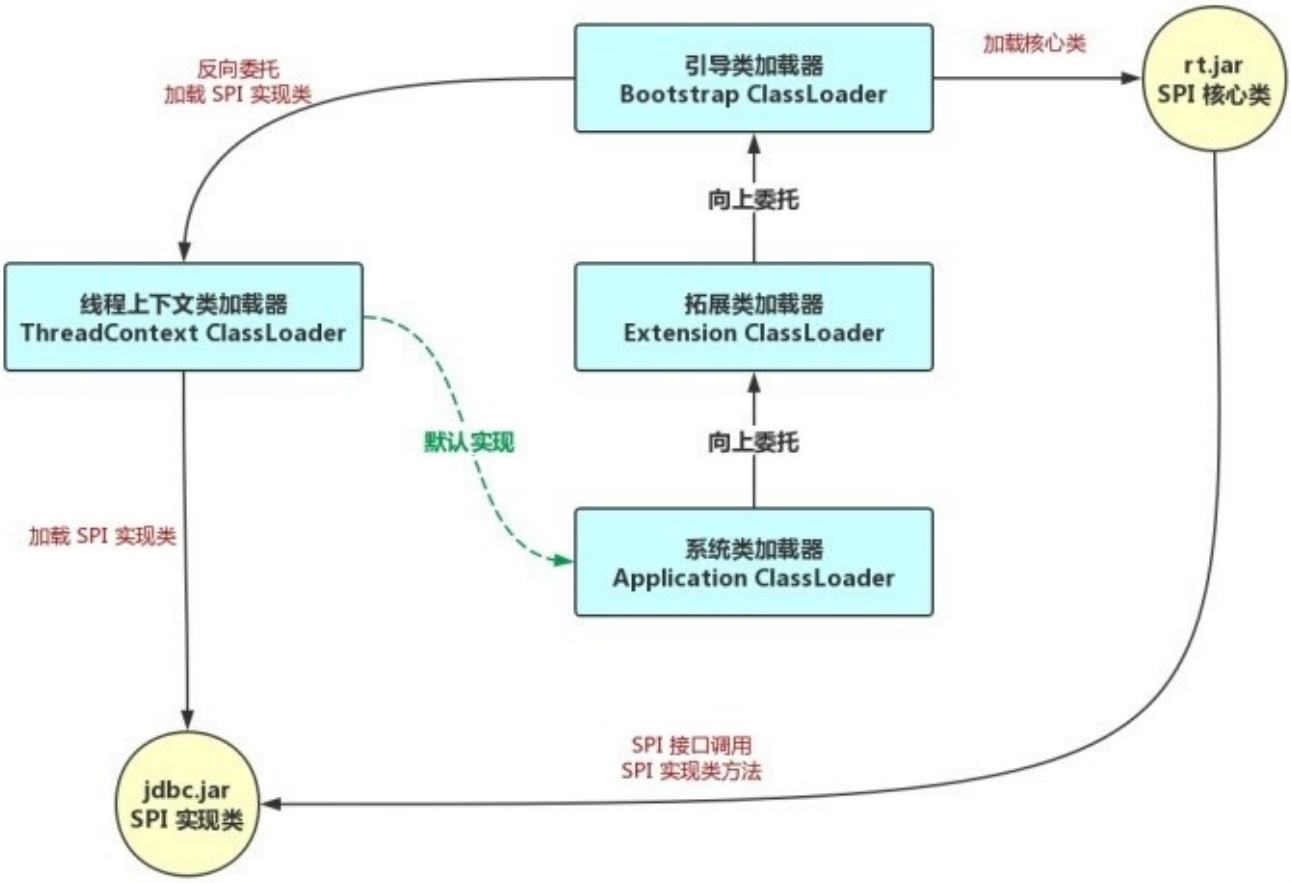
Q: 越基础的类由越上层的加载器进行加载，如果基础类又要调用回用户的代码，那该怎么办?

A: 解决方案：使用“线程上下文类加载器”

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setContextClassLoaser()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的话，那这个类加载器默认就是应用程序类加载器。

有了线程上下文类加载器，也就是父类加载器请求子类加载器去完成类加载的动作（即，父类加载器加载的类，使用线程上下文加载器去加载其无法加载的类），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。
Java中所有涉及SPI的加载动作基本上都采用这种方式，例如JNDI、JDBC、JCE、JAXB和JBI等。

线程上下文加载器流程图：



线程上下文类加载器的重要性

SPI (Service Provider Interface ———— 服务提供者接口)

父ClassLoader 可以使用当前线程 Thread.currentThread().getContextClassLoader() 所指定的 classloader 加载的类。这就改变了 父ClassLoader 不能使用 子ClassLoader 或是其他没有直接父子关系的 ClassLoader 加载的类的情况，即，改变了双亲委托模型。

线程上下文类加载器就是当前线程的 Current ClassLoader。

在双亲委托模型下，类加载器是由下至上的，即下层的类加载器会委托上层进行加载。但是对于 SPI 来说，有些接口是 Java 核心库所提供的，而 Java 核心库是由启动类加载器来加载的，而这些接口的实现却来自于不同的 jar 包（厂商提供），Java 的启动类加载器是不会加载其他来源的 jar 包，这样传统的双亲委托模型就无法满足 SPI 的要求。而通过给当前线程设置上下文类加载器，就可以由设置的上下文类加载器来实现对于接口实现类的加载。

在框架开发、底层组件开发、应用服务器、web服务器的开发，就会用到线程上下文类加载器。比如，tomcat框架，就对加载器就做了比较大的改造。
tomcat 的类加载器是首先尝试自己加载，自己加载不了才委托给它的双亲，这于传统的双亲委托模型是相反的。

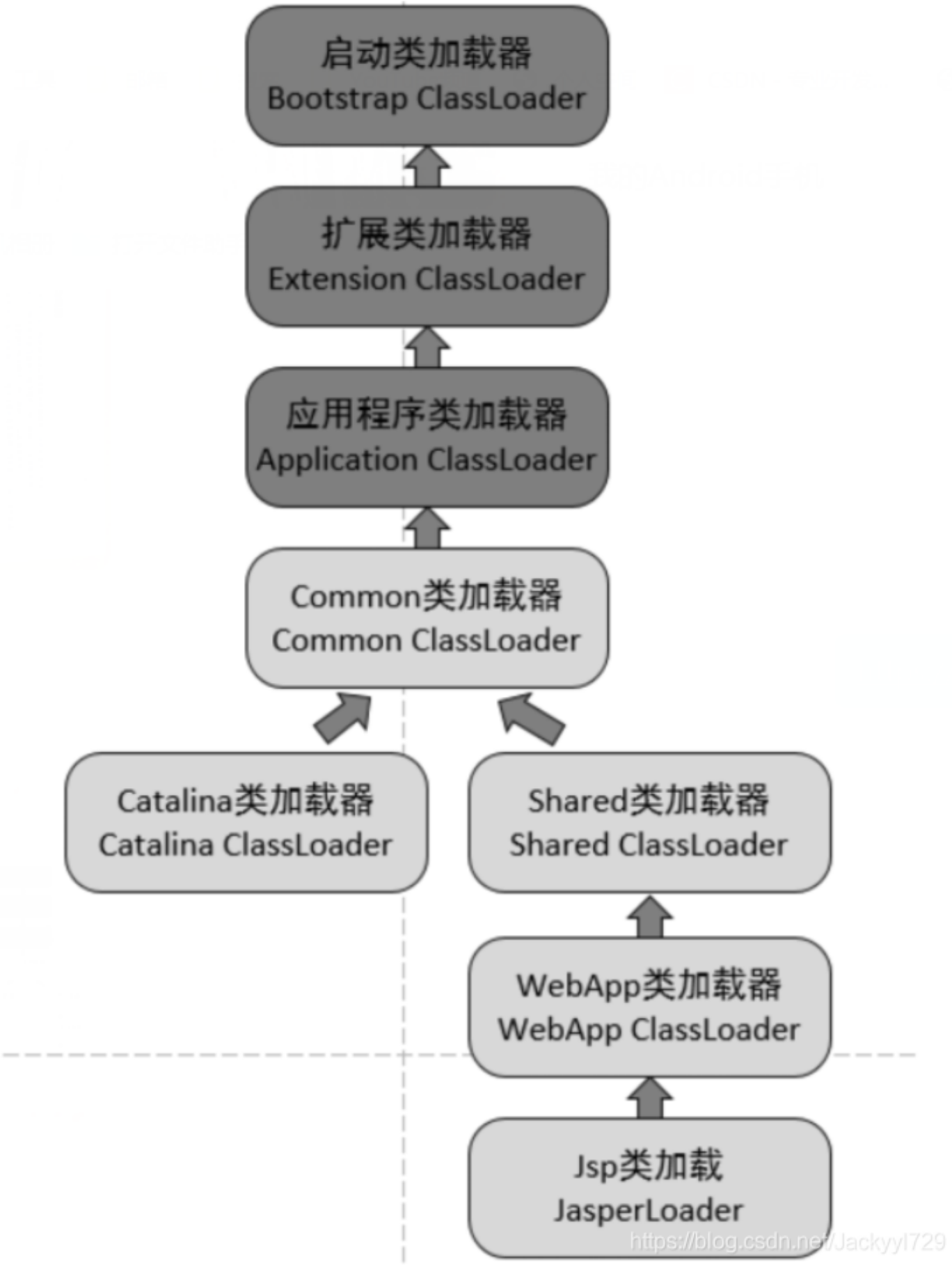
Q：其实只要是能加载目标类的任何加载器都可以实现打破委托模式的目的，那么为什么一定通过线程上下文类加载来实现了？

A：因为任何的 Java 代码都是运行某个线程上的，因此将这个打破委托模式的类加载器放在线程中是最合适的。

Tomcat的类加载机制

tomca通过重写 ClassLoader 的两个方法：findClass 和 loadClass，打破了双亲委派机制。

Tomcat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；

加载机制

1. 先从JVM的BootStrapClassLoader中加载。
2. 从 JVM 扩展类加载器 ExtClassLoader 去加载。
3. 加载Web应用下`/WEB-INF/classes`中的类（WebApp ClassLoader）。
4. 加载Web应用下`/WEB-INF/lib/*.jap`中的jar包中的类（WebApp ClassLoader）。
5. 加载上面定义的System路径下面的类。
6. 加载上面定义的Common路径下面的类。
7. 加载上面定义的Shared路径下面的类。

loadClass

```
public Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException {

    synchronized (getClassLoadingLock(name)) {

        Class<?> clazz = null;

        //1. 先在本地cache查找该类是否已经加载过
        clazz = findLoadedClass0(name);
        if (clazz != null) {
            if (resolve)
                resolveClass(clazz);
            return clazz;
        }

        //2. 从系统类加载器的cache中查找是否加载过
        clazz = findLoadedClass(name);
        if (clazz != null) {
            if (resolve)
                resolveClass(clazz);
            return clazz;
        }

        // 3. 尝试用ExtClassLoader类加载器类加载，为什么?
        ClassLoader javaseLoader = getJavaseClassLoader();
        try {
            clazz = javaseLoader.loadClass(name);
            if (clazz != null) {
                if (resolve)
                    resolveClass(clazz);
                return clazz;
            }
```

```
        } catch (ClassNotFoundException e) {
            // Ignore
        }

        // 4. 尝试在本地目录搜索class并加载
        try {
            clazz = findClass(name);
            if (clazz != null) {
                if (resolve)
                    resolveClass(clazz);
                return clazz;
            }
        } catch (ClassNotFoundException e) {
            // Ignore
        }

        // 5. 尝试用系统类加载器(也就是AppClassLoader)来加载
            try {
                clazz = Class.forName(name, false, parent);
                if (clazz != null) {
                    if (resolve)
                        resolveClass(clazz);
                    return clazz;
                }
            } catch (ClassNotFoundException e) {
                // Ignore
            }
       }

    //6. 上述过程都加载失败，抛出异常
    throw new ClassNotFoundException(name);
}
/*
loadClass 方法稍微复杂一点，主要有六个步骤：
1、先在本地 Cache 查找该类是否已经加载过，也就是说 Tomcat 的类加载器是否已经加载过这个类。
2、如果 Tomcat 类加载器没有加载过这个类，再看看系统类加载器是否加载过。
3、如果都没有，就让 ExtClassLoader 去加载，这一步比较关键，目的防止 Web 应用自己的类覆盖 JRE 的核心类。因为 Tomcat 需要打破双亲委托机制，假如 Web 应用里自定义了一个叫 Object 的类，如果先加载这个 Object 类，就会覆盖 JRE 里面的那个 Object 类，这就是为什么 Tomcat 的类加载器会优先尝试用 ExtClassLoader 去加载，因为 ExtClassLoader 会委托给 BootstrapClassLoader 去加载，BootstrapClassLoader 发现自己已经加载了 Object 类，直接返回给 Tomcat 的类加载器，这样 Tomcat 的类加载器就不会去加载 Web 应用下的 Object 类了，也就避免了覆盖 JRE 核心类的问题。
4、如果 ExtClassLoader 加载器加载失败，也就是说 JRE 核心类中没有这类，那么就在本地 Web 应用目录下查找并加载。
5、如果本地目录下没有这个类，说明不是 Web 应用自己定义的类，那么由系统类加载器去加载。这里请你注意，Web 应用是通过Class.forName调用交给系统类加载器的，因为Class.forName的默认加载器就是系统类加载器。
6、如果上述加载过程全部失败，抛出 ClassNotFound 异常。
*/
```

findClass 方法

```
public Class<?> findClass(String name) throws ClassNotFoundException {
    ...

    Class<?> clazz = null;
    try {
            //1. 先在Web应用目录下查找类
            clazz = findClassInternal(name);
    }  catch (RuntimeException e) {
           throw e;
       }

    if (clazz == null) {
    try {
            //2. 如果在本地目录没有找到，交给父加载器去查找
            clazz = super.findClass(name);
    }  catch (RuntimeException e) {
           throw e;
       }

    //3. 如果父类也没找到，抛出ClassNotFoundException
    if (clazz == null) {
        throw new ClassNotFoundException(name);
     }

    return clazz;
}

/*
在 findClass 方法里，主要有三个步骤：
1、先在 Web 应用本地目录下查找要加载的类。
2、如果没有找到，交给父加载器去查找，它的父加载器就是上面提到的系统类加载器 AppClassLoader。
3、如何父加载器也没找到这个类，抛出 ClassNotFound 异常。
*/
```