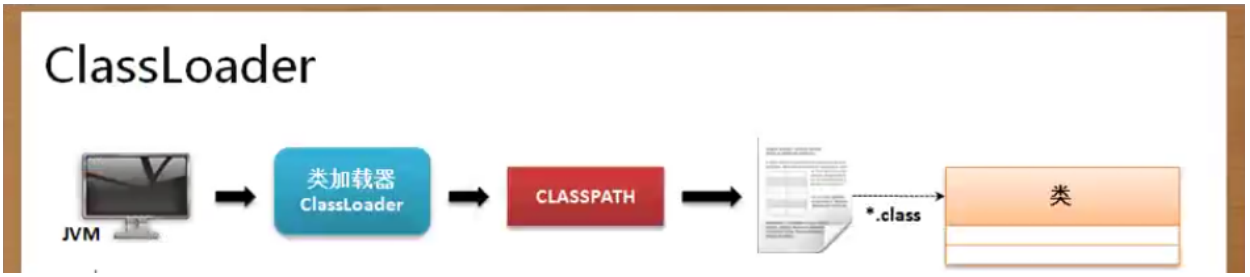
博客： https://www.cnblogs.com/HOsystem/p/14116443.html

2、具体内容

在Java语言里面提供有一个系统的环境变量：CLASSPATH，这个环境属性的作用主要是在JVM进程启动的时候进行类加载路径的定义，在JVM里面可以根据类加载器而后进行指定路径中类的加载，也就是说找到了类的加载器就意味着找到了类的来源



■系统类加载器

如果说现在要想获得类的加载器，那么一定要通过ClassLoader来获取，而要想获取ClassLoader类的对象，则必须利用Class类（反射的根源）实现，方法：public ClassLoader getClassLoader()；当获取了ClassLoader之后还可以继续获取其父类的ClassLoader类对象：public final ClassLoader getParent()；

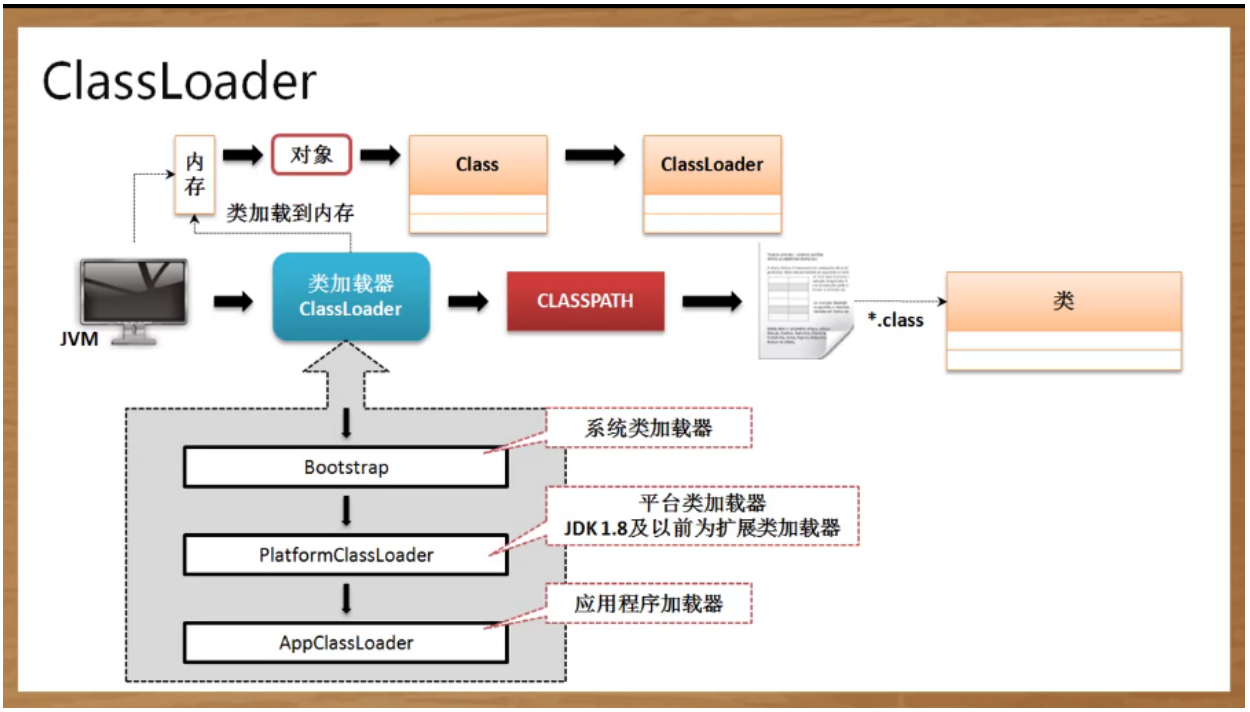
范例：观察类加载器

```
package cn.mldn.demo;
class Message {}
public class JavaAPIDemo {
    public static void main(String[] args) throws Exception {
        Class<?> clazz = Message.class ;
        System.out.println(clazz.getClassLoader());    // 获取当前类的加载器
        System.out.println(clazz.getClassLoader().getParent()); // 获取父类加载器
        System.out.println(clazz.getClassLoader().getParent().getParent());      // null
    }
}
```

程序执行结果：	sun.misc.Launcher$AppClassLoader@73d16e93

	sun.misc.Launcher$ExtClassLoader@15db9742 null

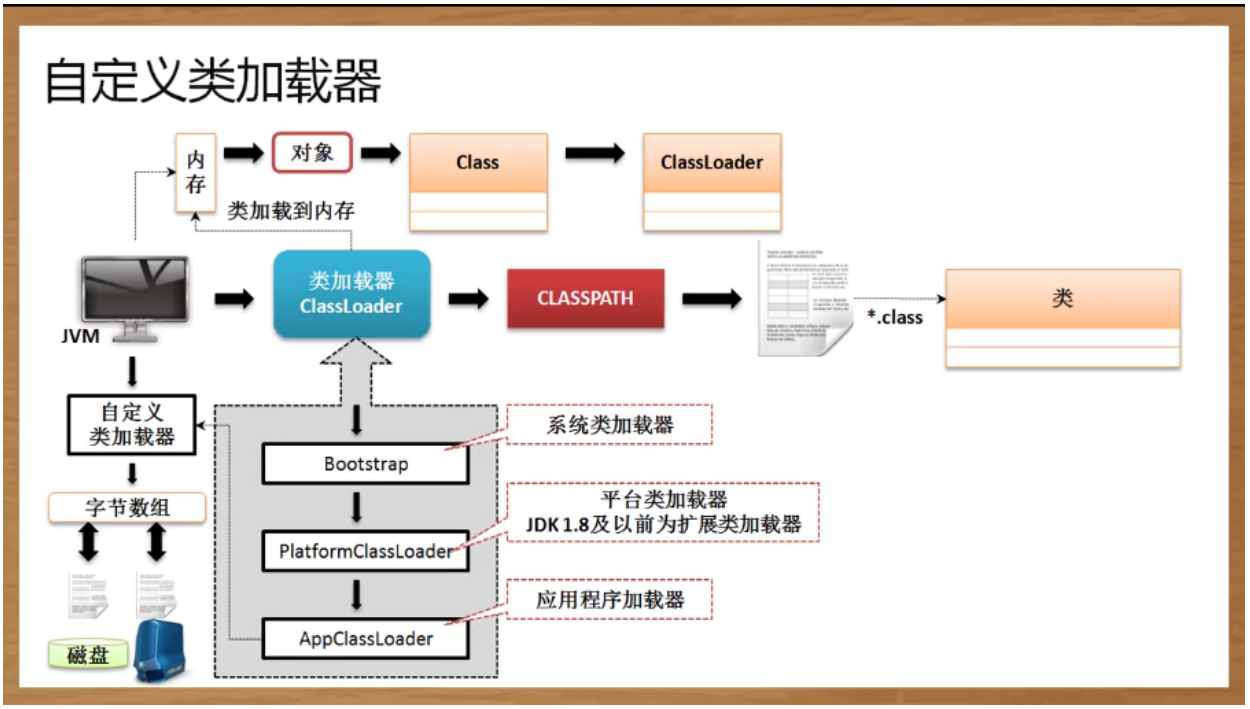
从JDK1.8之后的版本（JDK1.9、JDK1.10）提供有一个“PlatformClassLoader”类加载器，而在JDK1.8及以前的版本里面提供的加载器为“ExtClassLoader”，因为在JDK的安装目录里面提供有一个ext的目录，开发者可以将*.jar文件拷贝到此目录里面，这样就可以直接执行了，但是这样的处理开发并不安全，最初的时候也是不提倡使用的，所以从JDK.19开始将其彻底废除了，同时为了与系统类加载器和应用类加载器之间保持设计的平衡，提供有平台类加载器。



当你获取了类加载器之后就可以利用类加载器来实现类的反射加载处理。

■自定义类加载器

清楚了类加载器的功能之后就可以根据自身的需要来实现自定义的类加载器，但是千万要记住一点，自定义的类加载器其加载的顺序是在所有系统类加载器的最后。系统类中的类加载器都是根据CLASSPATH路径进行类加载的，而如果有了自定义类的加载器，就可以由开发者任意指派类的加载位置。



1、随意编写一个程序类，并且将这个类保存在磁盘上

```
package cn.mldn.util;
public class Message {
	public void send() {
		System.out.println("www.mldn.cn");
	}
}
```

2、将此类直接拷贝到D盘上进行编译处理，并且不打包：java Message.java，此时并没有进行打包处理，所以这个类无法通过CLASSPATH正常加载。

3、自定义一个类加载器，并且继承自ClassLoader类，在ClassLoader类里面为用户提供有一个字节转换为类结构的方法：

·定义类：protected final Class<?> defineClass(String name,byte[] b,int off,int len) throws ClassFormatError;

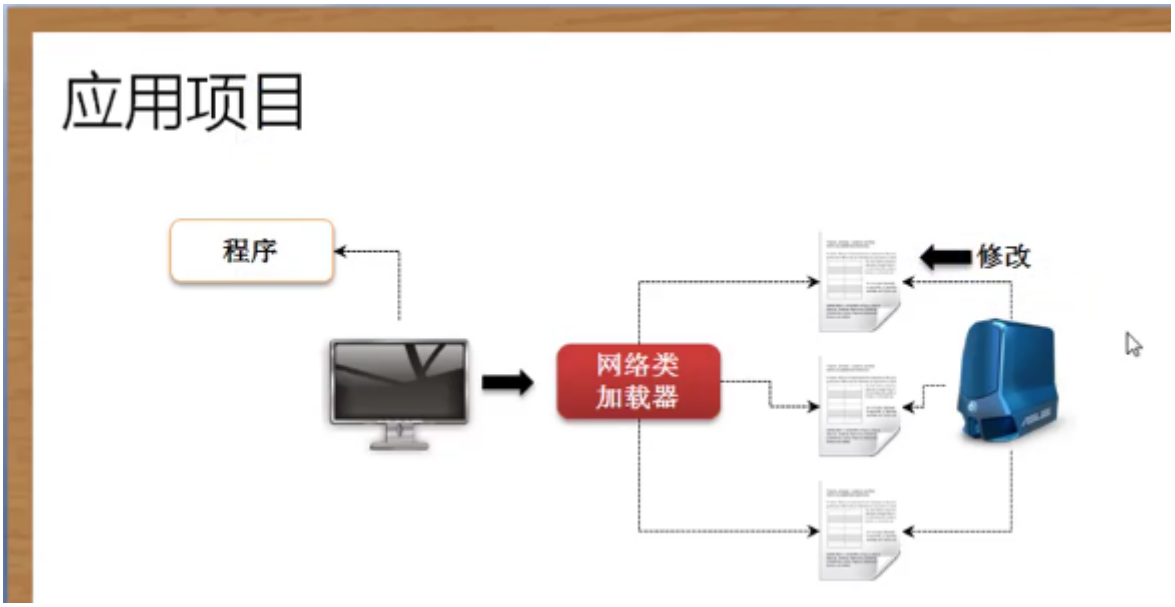
```
package cn.mldn.util;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
public class MLDNClassLoader extends ClassLoader {
	private static final String MESSAGE_CLASS_PATH = "D:" + File.separator +
"Message.class" ;
	/**
	 * 进行指定类的加载
	 * @param className 类的完整名称“包.类”
	 * @return 返回一个指定类的Class对象
	 * @throws Exception 如果类文件不存在则无法加载
	 */
	public Class<?> loadData(String className) throws Exception {
		byte [] data = this.loadClassData() ; // 读取二进制数据文件
```

```
            if (data != null) {      // 读取到了
                return super.defineClass(className, data, 0, data.length) ;
            }
            return null ;
        }
        private byte [] loadClassData() throws Exception {// 通过文件进行类的加载
            InputStream input = null ;
            ByteArrayOutputStream bos = null ; // 将数据加载到内存之中
            byte data [] = null ;
            try {
                bos = new ByteArrayOutputStream() ;   // 实例化内存流
                input = new FileInputStream(new File(MESSAGE_CLASS_PATH)) ; // 文件流加
载
                input.transferTo(bos) ; // 读取数据
                data = bos.toByteArray() ; // 将所有读取到的字节数据取出
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                if (input != null) {
                    input.close();
                }
                if (bos != null) {
                    bos.close();
                }
            }
            return data ;
        }
}
```

4、编写测试类实现类加载控制

```
package cn.mldn.demo;
import java.lang.reflect.Method;
import cn.mldn.util.MLDNClassLoader;
class Message {}
public class JavaAPIDemo {
        public static void main(String[] args) throws Exception {
            MLDNClassLoader classLoader = new MLDNClassLoader() ; // 实例化自定义类加载
器
            Class<?> cls = classLoader.loadData("cn.mldn.util.Message") ; // 进行类的加载
            Object obj = cls.getDeclaredConstructor().newInstance() ;
            Method method = cls.getDeclaredMethod("send") ;
            method.invoke(obj) ;
        }
}
```

如果在以后结合到网络程序开发的话，就可以通过一个远程的服务器来确定类的功能。



5、观察当前的Message类的加载器的情况

```
package cn.mldn.demo;
import java.lang.reflect.Method;
import cn.mldn.util.MLDNClassLoader;
class Message {}
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		MLDNClassLoader classLoader = new MLDNClassLoader() ; // 实例化自定义类加载
器
		Class<?> cls = classLoader.loadData("cn.mldn.util.Message") ; // 进行类的加载
		System.out.println(clazz.getClassLoader());
		System.out.println(clazz.getClassLoader().getParent());
		System.out.println(clazz.getClassLoader().getParent().getParent());
	}
}
```

程序执行结果	cn.mldn.util.MLDNClassLoader@6979e8cb jdk.internal.loader.ClassLoaders$AppClassLoader@6659c656 jdk.internal.loader.ClassLoaders$PlatformClassLoader@763d9750

如果说现在定义了一个类，这个类的名字为：java.lang.String，兵器利用了自定义的类加载器进行加载处理，这个类将不会被加载，Java之中针对于类加载器提供有双亲加载机制，如果现在要加载的程序类时由系统提供的类则会由系统类进行加载，如果现在开发者定义的类与系统类名称相同，那么为了保证系统的安全性绝对不会加载。