在Java反射里，我们频繁接触到Class类，它是描述类信息的类，而Class类中的forName()方法可以根据一个类的全名称得到类的Class对象，其底层原理其实是：根据ClassPath配置的路径进行类的加载。

一般情况下，我们只需要使用JDK提供的默认类加载器，不用关心类加载器。但是如果我们的类加载路径是网络，文件，这个时候我们就得了解类加载器，也就是ClassLoader的作用。

1. ClassLoader类加载器

1.1 ClassLoader概念

Class类的getClassLoader()方法：

```
// 获得当前Class对象的类加载器
public ClassLoader getClassLoader()
```

看下面的例子：

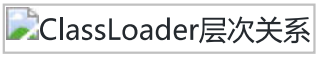
```
class Demo {
}

public class Test {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Demo.class;
        // 取得Class类对象的类加载器信息
        System.out.println(cls.getClassLoader());
        // 取得Class类对象的类加载器父加载器信息
        System.out.println(cls.getClassLoader().getParent());
        // 取得Class类对象的类加载器父加载器的父加载器信息
        System.out.println(cls.getClassLoader().getParent().getParent());
    }
}
```

JDK1.8运行结果：

我们可以看到运行结果中出现了两个名词：AppClassLoader, ExtClassLoader, 其实这就是类加载器。

JVM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"这个动作放在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。实现这个动作的代码模块称之为"类加载器"。

ClassLoader层次关系：



1.2 ClassLoader分类

ClassLoader可分为如下四种：

（1）Bootstrap

Bootstrap(启动类加载器)：这个类加载器使用C++实现，是虚拟机自身的一部分。而其他的类加载器都由Java语言实现，独立于JVM外部并且都继承于java.lang.ClassLoader。

启动类加载器无法被Java程序直接引用(C++编写)，所以在上面的例子运行结果第三行有一个 null，这其实就是Bootstrap。因为无法引用，所以打印出来是null。

BootStrap类加载器负责将存放于<Java_HOME>\lib目录中(或者被-Xbootclasspath参数指定路径中)能被虚拟机识别的类库加载到JVM内存中。

仅按照文件名识别，jvm有一个目录保存所需要的jar包名字，所以即使将起的的jar包丢进该目录，也不会识别。

（2）ExtClassLoader

ExtClassLoader(扩展类加载器)：它负责加载<Java_HOME>\lib\ext目录中，或者被java.ext.dirs系统变量指定的路径中的类库。

开发者可以直接使用扩展类加载器。

（3）AppClassLoader

AppClassLoader(应用程序类加载器):负责加载用户类路径(ClassPath)上指定的类库，如果应用程序中没有自定义自己的类加载器，则此加载器就是程序中默认的类加载器。

（4）自定义类加载器

顾名思义，自己定义的类加载器。

自定义类加载器和AppClassLoader是平级的，所以说类加载器分为三级。

2. 双亲委派模型

如果一个类加载器收到了类加载请求，它首先不会自己去尝试加载这个类，而是把这个请求委托给父类加载器去完成，每一个层次的类加载器都是如此。

因此所有的加载请求都应当传送到顶层的BootStrap加载器中，只有当父加载器反馈无法完成这个加载请求时(在自己搜索范围中没有找到此类)，子加载器才会尝试自己去加载。

类加载器的双亲委派模型从JDK1.2引入后被广泛应用于之后几乎所有的Java程序中，即编译器默认的类加载形式就是双亲委派模型。

但它并不是强制性约束，甚至可以破坏双亲委派模型来进行类加载，最典型的就是OSGI技术。

> OSGI: Java模块化技术，即热加载。自定义一个类加载器，可以在不关闭当前JVM前提下情况下，将新加入的类进行加载。而双亲委派模型不能这样。

双亲委派模式对于保证Java程序的稳定运行很重要。有一个显而易见的好处就是Java类随着它的类加载器一起具备了一种带有优先级的层次关系。

如果项目中有一个类名为 Object，包名为也为 java.lang ，那么这就会和真正的 Object 类冲突了，但是在双亲委派模型下，操作过程如下：

Java Object类 位于 java.lang ，这个包位于 rt.jar包内，rt.jar 由BootStrap 进行加载，又因为自定义的Object包由双亲委派模型进行加载，所以最终还是要由 BootStrap 进行加载，而BootStrap 搜索自己路径下的包，找到的是 Java自己的 Object，所以不会加载到自定义的 Object。

这样就保证了Java程序的稳定执行。

3. 自定义类加载器

3.1 使用默认类加载器

```
// 调用当前类加载器根据传入的类名全名称创建该类的Class对象，即类的加载
public Class<?> loadClass(String name) throws ClassNotFoundException
```

使用默认类加载器进行类的加载：

```
class Demo1 {
    @Override
    public String toString() {
        return "i am demo1";
    }
}

class Demo2 {
    @Override
    public String toString() {
        return "i am demo2";
    }
}

public class Test {
    public static void main(String[] args) throws Exception {
        // 使用加载了Demo1的默认类加载器加载Demo2
        Class<?> cls = Class.forName("class13.Demo1").getClassLoader().loadClass("class13.Demo2");
        System.out.println(cls.newInstance());
    }
}
```

运行结果：

3.2 自定义类加载器

自定义类加载器我们要用到：

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

将自定义一个类加载器将当前工程文件夹之外的类文件加载到当前工程中。

```
import java.io.ByteArrayOutputStream;
import java.io.FileInputStream;
import java.io.InputStream;

// 自定义类加载器类要继承ClassLoader类
class MyClassLoader extends ClassLoader {
    // 要加载的类名
    private String className;
    // 要加载的类的绝对路径
    private String classPath;

    public MyClassLoader(String className, String classPath) {
        super();
        this.className = className;
        this.classPath = classPath;
    }

    // 类加载器方法
    public Class<?> loadData() throws Exception {
        byte[] classData = this.loadClassData();
        return super.defineClass(this.className, classData, 0 , classData.length);
    }


    // 通过指定的文件绝对路径进行类的文件加载，实际就是对制定类二进制文件的读取
    private byte[] loadClassData() throws Exception {
        InputStream input = new FileInputStream(this.classPath);
        // 打开一个内存流
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        // 缓冲区
        byte[] data = new byte[20];
        int len = 0;
        while((len = input.read(data)) != -1) {
            bos.write(data, 0, len);
        }
        byte[] ret = bos.toByteArray();
        input.close();
        bos.close();
        return ret;
    }
}

public class Test {
    public static void main(String[] args) throws Exception {
        Class<?> cls = new MyClassLoader("Demo.class", "E:\\A\\B\\C\\Demo.class").loadData();
        System.out.println(cls);
```

```
        }
    }
```

我们通过文件I/O的方式就达到了加载项目外类文件的目的。

因为有了自定义类加载器，所以这里要完善一个概念。

比较两个类相等的前提：必须是由同一个类加载器加载的前提下才有意义。否则，即使两个类来源于同一个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，那么这两个类注定不相等。