青空b站课程

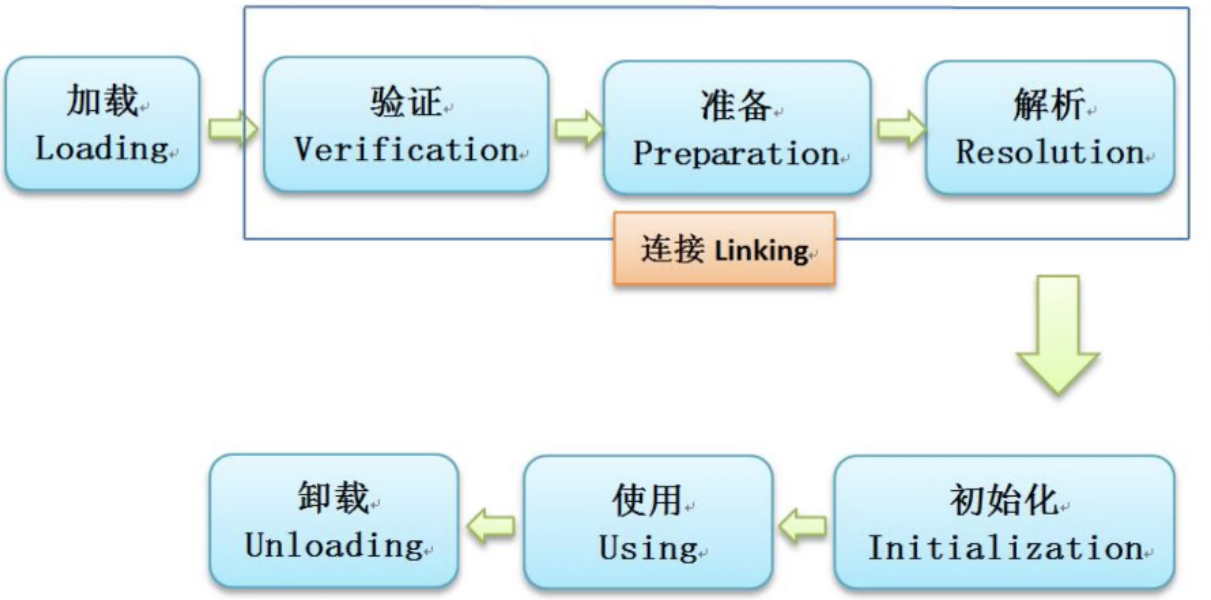
# 反射

反射就是把Java类中的各个成分映射成一个个的Java对象。即在运行状态中，对于任意一个类，都能够知道这个类所有的属性和方法，对于任意一个对象，都能调用它的任意一个方法和属性。这种动态获取信息及动态调用对象方法的功能叫Java的反射机制。

简而言之，我们可以通过反射机制，获取到类的一些属性，包括类里面有哪些字段，有哪些方法，继承自哪个类，甚至还能获取到泛型！它的权限非常高，慎重使用！

# Java类加载机制

在学习Java的反射机制之前，我们需要先了解一下类的加载机制，一个类是如何被加载和使用的：




在Java程序启动时，JVM会将一部分类（class文件）先加载（并不是所有的类都会在一开始加载），通过ClassLoader将类加载，在加载过程中，会将类的信息提取出来（存放在元空间中，JDK1.8之前存放在永久代），同时也会生成一个Class对象存放在内存（堆内存），注意此Class对象只会存在一个，与加载的类唯一对应！

为了方便各位小伙伴理解，你们就直接理解为默认情况下（仅使用默认类加载器）每个类都有且只有一个唯一的Class对象存放在JVM中，我们无论通过什么方式访问，都是始终是那一个对象。Class对象中包含我们类的一些信息，包括类里面有哪些方法、哪些变量等等。

# Class类详解

通过前面，我们了解了类的加载，同时会提取一个类的信息生成Class对象存放在内存中，而反射机制其实就是利用这些存放的类信息，来获取类的信息和操作类。那么如何获取到每个类对应的Class对象呢，我们可以通过以下方式：

```
public static void main(String[] args) throws ClassNotFoundException {
    Class<String> clazz = String.class;   //使用class关键字，通过类名获取
    Class<?> clazz2 = Class.forName("java.lang.String");   //使用Class类静态方法forName()，通过包名.类名获取，注意返回值是Class<?>
    Class<?> clazz3 = new String("cpdd").getClass();  //通过实例对象获取
}
```

注意Class类也是一个泛型类，只有第一种方法，能够直接获取到对应类型的Class对象，而以下两种方法使用了`?`通配符作为返回值，但是实际上都和第一个返回的是同一个对象：

```
Class<String> clazz = String.class;   //使用class关键字，通过类名获取
Class<?> clazz2 = Class.forName("java.lang.String");   //使用Class类静态方法forName()，通过包名.类名获取，注意返回值是Class<?>
Class<?> clazz3 = new String("cpdd").getClass();

System.out.println(clazz == clazz2);
System.out.println(clazz == clazz3);
```

通过比较，验证了我们一开始的结论，在JVM中每个类始终只存在一个Class对象，无论通过什么方法获取，都是一样的。现在我们再来看看这个问题：

```
public static void main(String[] args) {
    Class<?> clazz = int.class;   //基本数据类型有Class对象吗？
    System.out.println(clazz);
}
```

迷了，不是每个类才有Class对象吗，基本数据类型又不是类，这也行吗？实际上，基本数据类型也有对应的Class对象（反射操作可能需要用到），而且我们不仅可以通过class关键字获取，其实本质上是定义在对应的包装类中的：

```
/**
 * The {@code Class} instance representing the primitive type
 * {@code int}.
 *
 * @since   JDK1.1
 */
@SuppressWarnings("unchecked")
public static final Class<Integer>  TYPE = (Class<Integer>)
Class.getPrimitiveClass("int");

/*
 * Return the Virtual Machine's Class object for the named
 * primitive type
 */
static native Class<?> getPrimitiveClass(String name);   //C++实现，并非Java定义
```

每个包装类中（包括Void），都有一个获取原始类型Class方法，注意，getPrimitiveClass获取的是原始类型，并不是包装类型，只是可以使用包装类来表示。

```java
public static void main(String[] args) {
    Class<?> clazz = int.class;
    System.out.println(Integer.TYPE == int.class);
}
```

通过对比，我们发现实际上包装类型都有一个TYPE，其实也就是基本类型的Class，那么包装类的Class和基本类的Class一样吗？

```java
public static void main(String[] args) {
    System.out.println(Integer.TYPE == Integer.class);
}
```

我们发现，包装类型的Class对象并不是基本类型Class对象。数组类型也是一种类型，只是编程不可见，因此我们可以直接获取数组的Class对象：

```java
public static void main(String[] args) {
    Class<String[]> clazz = String[].class;
    System.out.println(clazz.getName());  //获取类名称（得到的是包名+类名的完整名称）
    System.out.println(clazz.getSimpleName());
    System.out.println(clazz.getTypeName());
    System.out.println(clazz.getClassLoader());   //获取它的类加载器
    System.out.println(clazz.cast(new Integer("10")));   //强制类型转换
}
```

下节课，我们将开始对Class对象的使用进行讲解。

## Class对象与多态

正常情况下，我们使用instanceof进行类型比较：

```java
public static void main(String[] args) {
    String str = "";
    System.out.println(str instanceof String);
}
```

它可以判断一个对象是否为此接口或是类的实现或是子类，而现在我们有了更多的方式去判断类型：

```java
public static void main(String[] args) {
    String str = "";
    System.out.println(str.getClass() == String.class);   //直接判断是否为这个类型
}
```

如果需要判断是否为子类或是接口/抽象类的实现，我们可以使用`asSubClass()`方法：

```
public static void main(String[] args) {
    Integer i = 10;
    i.getClass().asSubclass(Number.class);   //当Integer不是Number的子类时，会产生异常
}
```

通过`getSuperclass()`方法，我们可以获取到父类的Class对象：

```
public static void main(String[] args) {
    Integer i = 10;
    System.out.println(i.getClass().getSuperclass());
}
```

也可以通过`getGenericSuperclass()`获取父类的原始类型的Type：

```
public static void main(String[] args) {
    Integer i = 10;
    Type type = i.getClass().getGenericSuperclass();
    System.out.println(type);
    System.out.println(type instanceof Class);
}
```

我们发现Type实际上是Class类的父接口，但是获取到的Type的实现并不一定是Class。

同理，我们也可以像上面这样获取父接口：

```
public static void main(String[] args) {
    Integer i = 10;
    for (Class<?> anInterface : i.getClass().getInterfaces()) {
        System.out.println(anInterface.getName());
    }

    for (Type genericInterface : i.getClass().getGenericInterfaces()) {
        System.out.println(genericInterface.getTypeName());
    }
}
```

是不是感觉反射功能很强大？几乎类的所有信息都可以通过反射获得。

## 创建类对象

既然我们拿到了类的定义，那么是否可以通过Class对象来创建对象、调用方法、修改变量呢？当然是可以的，那我们首先来探讨一下如何创建一个类的对象：

```
public static void main(String[] args) throws InstantiationException,
IllegalAccessException {
    Class<Student> clazz = Student.class;
    Student student = clazz.newInstance();
    student.test();
}

static class Student{
    public void test(){
        System.out.println("萨日朗");
    }
}
```

通过使用`newInstance()`方法来创建对应类型的实例，返回泛型T，注意它会抛出InstantiationException和IllegalAccessException异常，那么什么情况下会出现异常呢?

```
public static void main(String[] args) throws InstantiationException,
IllegalAccessException {
    Class<Student> clazz = Student.class;
    Student student = clazz.newInstance();
    student.test();
}

static class Student{

    public Student(String text){

    }

    public void test(){
        System.out.println("萨日朗");
    }
}
```

当类默认的构造方法被带参构造覆盖时，会出现InstantiationException异常，因为`newInstance()`只适用于默认无参构造。

```
public static void main(String[] args) throws InstantiationException,
IllegalAccessException {
    Class<Student> clazz = Student.class;
    Student student = clazz.newInstance();
    student.test();
}

static class Student{

    private Student(){}

    public void test(){
        System.out.println("萨日朗");
```

```
    }
}
```

当默认无参构造的权限不是`public`时，会出现IllegalAccessException异常，表示我们无权去调用默认构造方法。在JDK9之后，不再推荐使用`newInstance()`方法了，而是使用我们接下来要介绍到的，通过获取构造器，来实例化对象：

```
public static void main(String[] args) throws NoSuchMethodException, InvocationTargetException, InstantiationException, IllegalAccessException {
    Class<Student> clazz = Student.class;
    Student student = clazz.getConstructor(String.class).newInstance("what's up");
    student.test();
}

static class Student{

    public Student(String str){}

    public void test(){
        System.out.println("萨日朗");
    }
}
```

通过获取类的构造方法（构造器）来创建对象实例，会更加合理，我们可以使用`getConstructor()`方法来获取类的构造方法，同时我们需要向其中填入参数，也就是构造方法需要的类型，当然我们这里只演示了。那么，当访问权限不是public的时候呢？

```
public static void main(String[] args) throws NoSuchMethodException, InvocationTargetException, InstantiationException, IllegalAccessException {
    Class<Student> clazz = Student.class;
    Student student = clazz.getConstructor(String.class).newInstance("what's up");
    student.test();
}

static class Student{

    private Student(String str){}

    public void test(){
        System.out.println("萨日朗");
    }
}
```

我们发现，当访问权限不足时，会无法找到此构造方法，那么如何找到非public的构造方法呢？

```
Class<Student> clazz = Student.class;
Constructor<Student> constructor = clazz.getDeclaredConstructor(String.class);
constructor.setAccessible(true);   //修改访问权限
Student student = constructor.newInstance("what's up");
student.test();
```

使用`getDeclaredConstructor()`方法可以找到类中的非public构造方法，但是在使用之前，我们需要先修改访问权限，在修改访问权限之后，就可以使用非public方法了（这意味着，反射可以无视权限修饰符访问类的内容）

## 调用类方法

我们可以通过反射来调用类的方法（本质上还是类的实例进行调用）只是利用反射机制实现了方法的调用，我们在包下创建一个新的类：

```
package com.test;

public class Student {
    public void test(String str){
        System.out.println("萨日朗"+str);
    }
}
```

这次我们通过`forName(String)`来找到这个类并创建一个新的对象：

```
public static void main(String[] args) throws ReflectiveOperationException {
    Class<?> clazz = Class.forName("com.test.Student");
    Object instance = clazz.newInstance();   //创建出学生对象
    Method method = clazz.getMethod("test", String.class);   //通过方法名和形参类型获取类中的方法

    method.invoke(instance, "what's up");   //通过Method对象的invoke方法来调用方法
}
```

通过调用`getMethod()`方法，我们可以获取到类中所有声明为public的方法，得到一个Method对象，我们可以通过Method对象的`invoke()`方法（返回值就是方法的返回值，因为这里是void，返回值为null）来调用已经获取到的方法，注意传参。

我们发现，利用反射之后，在一个对象从构造到方法调用，没有任何一处需要引用到对象的实际类型，我们也没有导入Student类，整个过程都是反射在代替进行操作，使得整个过程被模糊了，过多的使用反射，会极大地降低后期维护性。

同构造方法一样，当出现非public方法时，我们可以通过反射来无视权限修饰符，获取非public方法并调用，现在我们将`test()`方法的权限修饰符改为private：

```java
public static void main(String[] args) throws ReflectiveOperationException {
    Class<?> clazz = Class.forName("com.test.Student");
    Object instance = clazz.newInstance();   //创建出学生对象
    Method method = clazz.getDeclaredMethod("test", String.class);   //通过方法名和形参类型获取类中的方法
    method.setAccessible(true);

    method.invoke(instance, "what's up");   //通过Method对象的invoke方法来调用方法
}
```

Method和Constructor都和Class一样，他们存储了方法的信息，包括方法的形式参数列表，返回值，方法的名称等内容，我们可以直接通过Method对象来获取这些信息：

```java
public static void main(String[] args) throws ReflectiveOperationException {
    Class<?> clazz = Class.forName("com.test.Student");
    Method method = clazz.getDeclaredMethod("test", String.class);   //通过方法名和形参类型获取类中的方法

    System.out.println(method.getName());   //获取方法名称
    System.out.println(method.getReturnType());   //获取返回值类型
}
```

当方法的参数为可变参数时，我们该如何获取方法呢？实际上，我们在之前就已经提到过，可变参数实际上就是一个数组，因此我们可以直接使用数组的class对象表示：

```java
Method method = clazz.getDeclaredMethod("test", String[].class);
```

反射非常强大，尤其是我们提到的越权访问，但是请一定谨慎使用，别人将某个方法设置为private一定有他的理由，如果实在是需要使用别人定义为private的方法，就必须确保这样做是安全的，在没有了解别人代码的整个过程就强行越权访问，可能会出现无法预知的错误。

## 修改类的属性

我们还可以通过反射访问一个类中定义的成员字段也可以修改一个类的对象中的成员字段值，通过`getField()`方法来获取一个类定义的指定字段：

```java
public static void main(String[] args) throws ReflectiveOperationException {
    Class<?> clazz = Class.forName("com.test.Student");
    Object instance = clazz.newInstance();

    Field field = clazz.getField("i");   //获取类的成员字段i
    field.set(instance, 100);   //将类实例instance的成员字段i设置为100

    Method method = clazz.getMethod("test");
    method.invoke(instance);
}
```

在得到Field之后，我们就可以直接通过`set()`方法为某个对象，设定此属性的值，比如上面，我们就为instance对象设定值为100，当访问private字段时，同样可以按照上面的操作进行越权访问：

```
public static void main(String[] args) throws ReflectiveOperationException {
    Class<?> clazz = Class.forName("com.test.Student");
    Object instance = clazz.newInstance();

    Field field = clazz.getDeclaredField("i");    //获取类的成员字段i
    field.setAccessible(true);
    field.set(instance, 100);    //将类实例instance的成员字段i设置为100

    Method method = clazz.getMethod("test");
    method.invoke(instance);
}
```

现在我们已经知道，反射几乎可以把一个类的老底都给扒出来，任何属性，任何内容，都可以被反射修改，无论权限修饰符是什么，那么，如果我的字段被标记为final呢？现在在字段`i`前面添加`final`关键字，我们再来看看效果：

```
private final int i = 10;
```

这时，当字段为final时，就修改失败了！当然，通过反射可以直接将final修饰符直接去除，去除后，就可以随意修改内容了，我们来尝试修改Integer的value值：

```
public static void main(String[] args) throws ReflectiveOperationException {
    Integer i = 10;

    Field field = Integer.class.getDeclaredField("value");

    Field modifiersField = Field.class.getDeclaredField("modifiers");  //这里要获取Field类的modifiers字段进行修改
    modifiersField.setAccessible(true);
    modifiersField.setInt(field,field.getModifiers()&~Modifier.FINAL);  //去除final标记

    field.setAccessible(true);
    field.set(i, 100);    //强行设置值

    System.out.println(i);
}
```

我们可以发现，反射非常暴力，就连被定义为final字段的值都能强行修改，几乎能够无视一切阻拦。我们来试试看修改一些其他的类型：

```java
public static void main(String[] args) throws ReflectiveOperationException {
    List<String> i = new ArrayList<>();

    Field field = ArrayList.class.getDeclaredField("size");
    field.setAccessible(true);
    field.set(i, 10);

    i.add("测试");   //只添加一个元素
    System.out.println(i.size());  //大小直接变成11
    i.remove(10);   //瞎移除都不带报错的，淦
}
```

实际上，整个ArrayList体系由于我们的反射操作，导致被破坏，因此它已经无法正常工作了！

再次强调，在进行反射操作时，必须注意是否安全，虽然拥有了创世主的能力，但是我们不能滥用，我们只能把它当做一个不得已才去使用的工具！

## 类加载器

我们接着来介绍一下类加载器，实际上类加载器就是用于加载一个类的，但是类加载器并不是只有一个。

**思考：** 既然说Class对象和加载的类唯一对应，那如果我们手动创建一个与JDK包名一样，同时类名也保持一致，JVM会加载这个类吗？

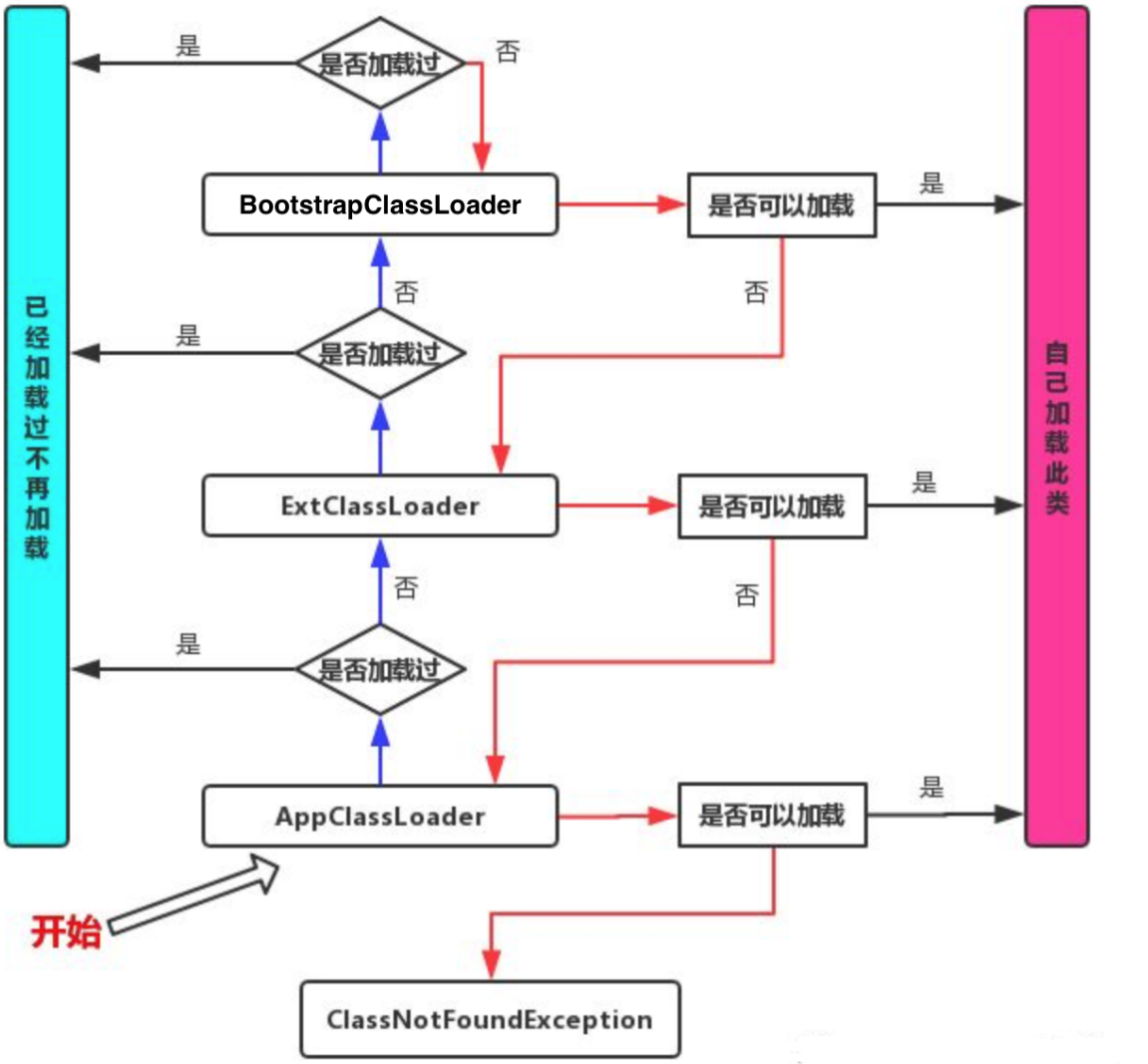
```java
package java.lang;

public class String {    //JDK提供的String类也是
    public static void main(String[] args) {
        System.out.println("我姓🐴，我叫🐴nb");
    }
}
```

我们发现，会出现以下报错：

```
错误: 在类 java.lang.String 中找不到 main 方法, 请将 main 方法定义为:
   public static void main(String[] args)
```

但是我们明明在自己写的String类中定义了main方法啊，为什么会找不到此方法呢？实际上这是ClassLoader的`双亲委派机制`在保护Java程序的正常运行：




实际上类最开始是由BootstarpClassLoader进行加载，BootstarpClassLoader用于加载JDK提供的类，而我们自己编写的类实际上是AppClassLoader加载的，只有BootstarpClassLoader都没有加载的类，才会让AppClassLoader来加载，因此我们自己编写的同名包同名类不会被加载，而实际要去启动的是真正的String类，也就自然找不到`main`方法了。

```
public class Main {
    public static void main(String[] args) {
        System.out.println(Main.class.getClassLoader());    //查看当前类的类加载器
        System.out.println(Main.class.getClassLoader().getParent());  //父加载器
        System.out.println(Main.class.getClassLoader().getParent().getParent());
 //爷爷加载器
        System.out.println(String.class.getClassLoader());    //String类的加载器
    }
}
```

由于BootstarpClassLoader是C++编写的，我们在Java中是获取不到的。

既然通过ClassLoader就可以加载类，那么我们可以自己手动将class文件加载到VM中吗？先写好我们定义的类：

```java
package com.test;

public class Test {
    public String text;

    public void test(String str){
        System.out.println(text+" > 我是测试方法! "+str);
    }
}
```

通过javac命令，手动编译一个.class文件：

```
nagocoler@NagodeMacBook-Pro HelloWorld % javac src/main/java/com/test/Test.java
```

编译后，得到一个class文件，我们把它放到根目录下，然后编写一个我们自己的ClassLoader，因为普通的ClassLoader无法加载二进制文件，因此我们编写一个自定义的来让它支持：

```java
//定义一个自己的ClassLoader
static class MyClassLoader extends ClassLoader{
    public Class<?> defineClass(String name, byte[] b){
        return defineClass(name, b, 0, b.length);   //调用protected方法，支持载入外部class文件
    }
}

public static void main(String[] args) throws IOException {
    MyClassLoader classLoader = new MyClassLoader();
    FileInputStream stream = new FileInputStream("Test.class");
    byte[] bytes = new byte[stream.available()];
    stream.read(bytes);
    Class<?> clazz = classLoader.defineClass("com.test.Test", bytes);   //类名必须和我们定义的保持一致
    System.out.println(clazz.getName());   //成功加载外部class文件
}
```

现在，我们就将此class文件读取并解析为Class了，现在我们就可以对此类进行操作了（注意，我们无法在代码中直接使用此类型，因为它是我们直接加载的），我们来试试看创建一个此类的对象并调用其方法：

```java
try {
    Object obj = clazz.newInstance();
    Method method = clazz.getMethod("test", String.class);   //获取我们定义的test(String str)方法
    method.invoke(obj, "哥们这瓜多少钱一斤？");
}catch (Exception e){
    e.printStackTrace();
}
```

我们来试试看修改成员字段之后，再来调用此方法：

```
try {
    Object obj = clazz.newInstance();
    Field field = clazz.getField("text");   //获取成员变量 String text;
    field.set(obj, "华强");
    Method method = clazz.getMethod("test", String.class);   //获取我们定义的test(String str)方法
    method.invoke(obj, "哥们这瓜多少钱一斤？");
}catch (Exception e){
    e.printStackTrace();
}
```

通过这种方式，我们就可以实现外部加载甚至是网络加载一个类，只需要把类文件传递即可，这样就无需再将代码写在本地，而是动态进行传递，不仅可以一定程度上防止源代码被反编译（只是一定程度上，想破解你代码有的是方法），而且在更多情况下，我们还可以对byte[]进行加密，保证在传输过程中的安全性。