目录

title: 夯实Java基础系列12：深入理解Java中的反射机制 date: 2019-9-12 15:56:26 # 文章生成时间，一般不改 categories: - Java技术江湖 - Java基础 tags: - Java反射

回顾：什么是反射?

反射(Reflection)是Java 程序开发语言的特征之一，它允许运行中的 Java 程序获取自身的信息，并且可以操作类或对象的内部属性。 Oracle官方对反射的解释是

> Reflection enables Java code to discover information about the fields, methods and constructors of loaded classes, and to use reflected fields, methods, and constructors to operate on their underlying counterparts, within security restrictions.

> The API accommodates applications that need access to either the public members of a target object (based on its runtime class) or the members declared by a given class. It also allows programs to suppress default reflective access control.

> 简而言之，通过反射，我们可以在运行时获得程序或程序集中每一个类型的成员和成员的信息。

> 程序中一般的对象的类型都是在编译期就确定下来的，而Java反射机制可以动态地创建对象并调用其属性，这样的对象的类型在编译期是未知的。所以我们可以通过反射机制直接创建对象，即使这个对象的类型在编译期是未知的。

> 反射的核心是JVM在运行时才动态加载类或调用方法/访问属性，它不需要事先（写代码的时候或编译期）知道运行对象是谁。

Java反射框架主要提供以下功能：

> 1.在运行时判断任意一个对象所属的类；

> 2.在运行时构造任意一个类的对象；

> 3.在运行时判断任意一个类所具有的成员变量和方法（通过反射甚至可以调用private方法）；

> 4.在运行时调用任意一个对象的方法

> 重点：是运行时而不是编译时

反射的主要用途

> 很多人都认为反射在实际的Java开发应用中并不广泛，其实不然。

> 当我们在使用IDE(如Eclipse，IDEA)时，当我们输入一个对象或类并想调用它的属性或方法时，一按点号，编译器就会自动列出它的属性或方法，这里就会用到反射。

> 反射最重要的用途就是开发各种通用框架。

> 很多框架（比如Spring）都是配置化的（比如通过XML文件配置JavaBean,Action之类的），为了保证框架的通用性，它们可能需要根据配置文件加载不同的对象或类，调用不同的方法，这个时候就必须用到反射——运行时动态加载需要加载的对象。

> 举一个例子，在运用Struts 2框架的开发中我们一般会在struts.xml里去配置Action，比如：

```
<action name="login"
                class="org.ScZyhSoft.test.action.SimpleLoginAction"
                method="execute">
            <result>/shop/shop-index.jsp</result>
            <result name="error">login.jsp</result>
        </action>
```

配置文件与Action建立了一种映射关系，当View层发出请求时，请求会被StrutsPrepareAndExecuteFilter拦截，然后StrutsPrepareAndExecuteFilter会去动态地创建Action实例。

——比如我们请求login.action，那么StrutsPrepareAndExecuteFilter就会去解析struts.xml文件，检索action中name为login的Action，并根据class属性创建SimpleLoginAction实例，并用invoke方法来调用execute方法，这个过程离不开反射。

> 对与框架开发人员来说，反射虽小但作用非常大，它是各种容器实现的核心。而对于一般的开发者来说，不深入框架开发则用反射用的就会少一点，不过了解一下框架的底层机制有助于丰富自己的编程思想，也是很有益的。

反射的基础：关于Class类

更多关于Class类和Object类的原理和介绍请见上一节

> 1、Class是一个类，一个描述类的类（也就是描述类本身），封装了描述方法的Method，描述字段的Filed，描述构造器的Constructor等属性
>
> 2、对象照镜子后（反射）可以得到的信息：某个类的数据成员名、方法和构造器、某个类到底实现了哪些接口。
>
> 3、对于每个类而言，JRE 都为其保留一个不变的 Class 类型的对象。一个Class对象包含了特定某个类的有关信息。
>
> 4、Class 对象只能由系统建立对象
>
> 5、一个类在 JVM 中只会有一个Class实例

```
//总结一下就是，JDK有一个类叫做Class，这个类用来封装所有Java类型，包括这些类的所有信息，JVM中类信息是放在方法区的。
```

//所有类在加载后，JVM会为其在堆中创建一个Class<类名称>的对象，并且每个类只会有一个Class对象，这个类的所有对象都要通过Class<类名称>来进行实例化。

//上面说的是JVM进行实例化的原理，当然实际上在Java写代码时只需要用 类名称就可以进行实例化了。

```
public final class Class<T> implements java.io.Serializable,
                          GenericDeclaration,
                          Type,
                          AnnotatedElement {
```

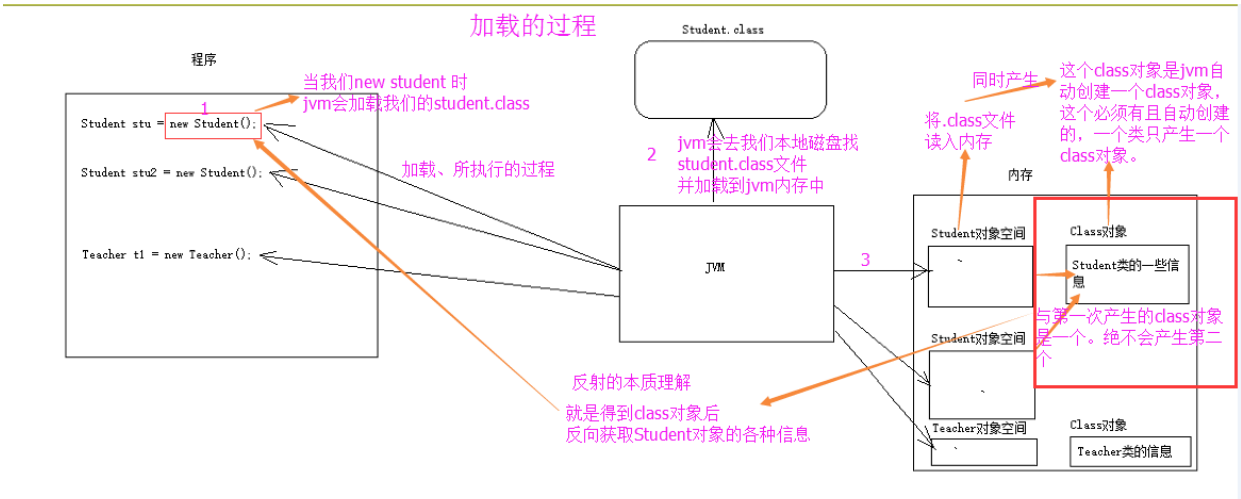
虚拟机会保持唯一一

```
            //通过类名.class获得唯一的Class对象。
            Class<UserBean> cls = UserBean.class;
            //通过integer.TYPEl来获取Class对象
            Class<Integer> inti = Integer.TYPE;
          //接口本质也是一个类，一样可以通过.class获取
            Class<User> userClass = User.class;
```

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

要想解剖一个类,必须先要获取到该类的字节码文件对象。而解剖使用的就是Class类中的方法.所以先要获取到每一个字节码文件对应的Class类型的对象.

以上的总结就是什么是反射 反射就是把java类中的各种成分映射成一个个的Java对象 例如：一个类有：成员变量、方法、构造方法、包等等信息，利用反射技术可以对一个类进行解剖，把个个组成部分映射成一个个对象。（其实：一个类中这些成员方法、构造方法、在加入类中都有一个类来描述） 如图是类的正常加载过程：反射的原理在与class对象。 熟悉一下加载的时候：Class对象的由来是将class文件读入内存，并为之创建一个Class对象。



Java为什么需要反射？反射要解决什么问题？

Java中编译类型有两种：

静态编译：在编译时确定类型，绑定对象即通过。 动态编译：运行时确定类型，绑定对象。动态编译最大限度地发挥了Java的灵活性，体现了多态的应用，可以减低类之间的耦合性。 Java反射是Java被视为动态（或准动态）语言的一个关键性质。这个机制允许程序在运行时透过Reflection APIs取得任何一个已知名称的class的内部信息，包括其modifiers（诸如public、static等）、superclass（例如Object）、实现之interfaces（例如Cloneable），也包括fields和methods的所有信息，并可于运行时改变fields内容或唤起methods。

Reflection可以在运行时加载、探知、使用编译期间完全未知的classes。即Java程序可以加载一个运行时才得知名称的class，获取其完整构造，并生成其对象实体、或对其fields设值、或唤起其methods。Class.forname("类的全名") javac静态编译期间并不会导致类加载，只有运行时执行上面代码时候，才会导致
反射（reflection）允许静态语言在运行时（runtime）检查、修改程序的结构与行为。在静态语言中，使用一个变量时，必须知道它的类型。在Java中，变量的类型信息在编译时都保存到了class文件中，这样在运行时才能保证准确无误；换句话说，程序在运行时的行为都是固定的。如果想在运行时改变，就需要反射这东西了。

实现Java反射机制的类都位于java.lang.reflect包中：

Class类：代表一个类 Field类：代表类的成员变量（类的属性） Method类：代表类的方法 Constructor类：代表类的构造方法 Array类：提供了动态创建数组，以及访问数组的元素的静态方法 一句话概括就是使用反射可以赋予jvm动态编译的能力，否则类的元数据信息只能用静态编译的方式实现，例如热加载，Tomcat的classloader等等都没法支持。

反射的基本运用

上面我们提到了反射可以用于判断任意对象所属的类，获得Class对象，构造任意一个对象以及调用一个对象。这里我们介绍一下基本反射功能的实现(反射相关的类一般都在java.lang.relfect包里)。

1、获得Class对象方法有三种

(1)使用Class类的forName静态方法:

```
 public static Class<?> forName(String className)
```
在JDBC开发中常用此方法加载数据库驱动：
要使用全类名来加载这个类，一般数据库驱动的配置信息会写在配置文件中。加载这个驱动前要先导入jar包
```java
 Class.forName(driver);
```

(2)直接获取某一个对象的class，比如:

```
//Class<?>是一个泛型表示，用于获取一个类的类型。
Class<?> klass = int.class;
Class<?> classInt = Integer.TYPE;
```

(3)调用某个对象的getClass()方法,比如:这是Object类继承的方法

```
StringBuilder str = new StringBuilder("123");
Class<?> klass = str.getClass();
```

判断是否为某个类的实例

一般地，我们用instanceof关键字来判断是否为某个类的实例。同时我们也可以借助反射中Class对象的isInstance()方法来判断是否为某个类的实例，它是一个Native方法：

==public native boolean isInstance(Object obj);== 上面的（3） klass.inInstance(str)

创建实例

通过反射来生成对象主要有两种方式。

（1）使用Class对象的newInstance()方法来创建Class对象对应类的实例。

注意：利用newInstance创建对象：调用的类必须有无参的构造器

```
//Class<?>代表任何类的一个类对象。
//使用这个类对象可以为其他类进行实例化
//因为jvm加载类以后自动在堆区生成一个对应的*.Class对象
//该对象用于让JVM对进行所有*对象实例化。
Class<?> c = String.class; 0 ldc #7 <java/lang/String>
                           2 astore_0
//Class<?> 中的 ? 是通配符，其实就是表示任意符合泛类定义条件的类，和直接使用 Class
//效果基本一致，但是这样写更加规范，在某些类型转换时可以避免不必要的 unchecked 错误。

Object str = c.newInstance();
```

（2）先通过Class对象获取指定的Constructor对象，再调用Constructor对象的newInstance()方法来创建实例。这种方法可以用指定的构造器构造类的实例。

```
//获取String所对应的Class对象
Class<?> c = String.class;
//获取String类带一个String参数的构造器
Constructor constructor = c.getConstructor(String.class);
//根据构造器创建实例
```

```
Object obj = constructor.newInstance("23333");
System.out.println(obj);
```

获取方法

获取某个Class对象的方法集合，主要有以下几个方法：

getDeclaredMethods()方法返回类或接口声明的所有方法，==包括公共、保护、默认（包）访问和私有方法，但不包括继承的方法==。

```
public Method[] getDeclaredMethods() throws SecurityException
```

getMethods()方法返回某个类的所有公用（public）方法，==包括其继承类的公用方法。==

```
public Method[] getMethods() throws SecurityException
```

getMethod方法返回一个特定的方法，其中第一个参数为方法名称，后面的参数为方法的参数对应Class的对象

```
public Method getMethod(String name, Class<?>... parameterTypes)
```

只是这样描述的话可能难以理解，我们用例子来理解这三个方法： 本文中的例子用到了以下这些类，用于反射的测试。

```
//注解类，可可用于表示方法，可以通过反射获取注解的内容。
    //Java注解的实现是很多注框架实现注解配置的基础
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Invoke {
}
```

userbean的父类personbean

```
public class PersonBean {
private String name;

int id;

public String getName() {
    return name;
}

public void setName(String name) {
```

```
        this.name = name;
    }



}
```

接口user

```
    public interface User {
        public void login ();

    }
```

userBean实现user接口，继承personbean

```
    public class UserBean extends PersonBean implements User{
        @Override
        public void login() {

        }

        class B {

        }

        public String userName;
        protected int i;
        static int j;
        private int l;
        private long userId;
        public UserBean(String userName, long userId) {
            this.userName = userName;
            this.userId = userId;
        }
        public String getName() {
            return userName;
        }
        public long getId() {
            return userId;
        }
        @Invoke
        public static void staticMethod(String devName,int a) {
            System.out.printf("Hi %s, I'm a static method", devName);
        }
        @Invoke
        public void publicMethod() {
            System.out.println("I'm a public method");
        }
        @Invoke
        private void privateMethod() {
            System.out.println("I'm a private method");
```

```
    }
}
```

1 getMethods和getDeclaredMethods的区别

```
public class 动态加载类的反射 {
    public static void main(String[] args) {
        try {
            Class clazz = Class.forName("com.javase.反射.UserBean");
            for (Field field : clazz.getDeclaredFields()) {
//                field.setAccessible(true);
                System.out.println(field);
            }
            //getDeclaredMethod*()获取的是类自身声明的所有方法，包含public、
protected和private方法。
            System.out.println("------共有方法------");
//        getDeclaredMethod*()获取的是类自身声明的所有方法，包含public、
protected和private方法。
//            getMethod*()获取的是类的所有共有方法，这就包括自身的所有public方
法，和从基类继承的、从接口实现的所有public方法。
            for (Method method : clazz.getMethods()) {
                String name = method.getName();
                System.out.println(name);
                //打印出了UserBean.java的所有方法以及父类的方法
            }
            System.out.println("------独占方法------");

            for (Method method : clazz.getDeclaredMethods()) {
                String name = method.getName();
                System.out.println(name);
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

2 打印一个类的所有方法及详细信息：

```
public class 打印所有方法 {

    public static void main(String[] args) {
        Class userBeanClass = UserBean.class;
        Field[] fields = userBeanClass.getDeclaredFields();
        //注意，打印方法时无法得到局部变量的名称，因为jvm只知道它的类型
        Method[] methods = userBeanClass.getDeclaredMethods();
        for (Method method : methods) {
            //依次获得方法的修饰符，返回类型和名称，外加方法中的参数
            String methodString = Modifier.toString(method.getModifiers()) + "
" ; // private static
            methodString += method.getReturnType().getSimpleName() + " "; //
```

```
void
            methodString += method.getName() + "("; // staticMethod
            Class[] parameters = method.getParameterTypes();
            Parameter[] p = method.getParameters();

            for (Class parameter : parameters) {
                methodString += parameter.getSimpleName() + " " ; // String
            }
            methodString += ")";
            System.out.println(methodString);
        }
        //注意方法只能获取到其类型，拿不到变量名
/*        public String getName()
        public long getId()
        public static void staticMethod(String int )
        public void publicMethod()
        private void privateMethod()*/
    }
}
```

获取构造器信息

获取类构造器的用法与上述获取方法的用法类似。主要是通过Class类的getConstructor方法得到Constructor类的一个实例，而Constructor类有一个newInstance方法可以创建一个对象实例:

```
public class 打印构造方法 {
    public static void main(String[] args) {
        // constructors
        Class<?> clazz = UserBean.class;

        Class userBeanClass = UserBean.class;
        //获得所有的构造方法
        Constructor[] constructors = userBeanClass.getDeclaredConstructors();
        for (Constructor constructor : constructors) {
            String s = Modifier.toString(constructor.getModifiers()) + " ";
            s += constructor.getName() + "(";
            //构造方法的参数类型
            Class[] parameters = constructor.getParameterTypes();
            for (Class parameter : parameters) {
                s += parameter.getSimpleName() + ", ";
            }
            s += ")";
            System.out.println(s);
            //打印结果//public com.javase.反射.UserBean(String, long, )

        }
    }
}
```

获取类的成员变量（字段）信息

主要是这几个方法，在此不再赘述：

getFiled: 访问公有的成员变量 getDeclaredField：所有已声明的成员变量。但不能得到其父类的成员变量 getFileds和getDeclaredFields用法同上（参照Method）

```
public class 打印成员变量 {
    public static void main(String[] args) {
        Class userBeanClass = UserBean.class;
        //获得该类的所有成员变量，包括static private
        Field[] fields = userBeanClass.getDeclaredFields();

        for(Field field : fields) {
            //private属性即使不用下面这个语句也可以访问
//            field.setAccessible(true);

            //因为类的私有域在反射中默认可访问，所以flag默认为true。
            String fieldString = "";
            fieldString += Modifier.toString(field.getModifiers()) + " "; //
`private`
            fieldString += field.getType().getSimpleName() + " "; // `String`
            fieldString += field.getName(); // `userName`
            fieldString += ";";
            System.out.println(fieldString);

            //打印结果
//            public String userName;
//            protected int i;
//            static int j;
//            private int l;
//            private long userId;
        }

    }
}
```

调用方法

当我们从类中获取了一个方法后，我们就可以用invoke()方法来调用这个方法。invoke方法的原型为:

```
public Object invoke(Object obj, Object... args)
        throws IllegalAccessException, IllegalArgumentException,
           InvocationTargetException

public class 使用反射调用方法 {
    public static void main(String[] args) throws InvocationTargetException,
IllegalAccessException, InstantiationException, NoSuchMethodException {
```

```
        Class userBeanClass = UserBean.class;
        //获取该类所有的方法，包括静态方法，实例方法。
        //此处也包括了私有方法，只不过私有方法在用invoke访问之前要设置访问权限
        //也就是使用setAccessible使方法可访问，否则会抛出异常
//        // IllegalAccessException的解释是
//         * An IllegalAccessException is thrown when an application tries
// * to reflectively create an instance (other than an array),
// * set or get a field, or invoke a method, but the currently
// * executing method does not have access to the definition of
// * the specified class, field, method or constructor.

//        getDeclaredMethod*()获取的是类自身声明的所有方法，包含public、protected和private方法。
//            getMethod*()获取的是类的所有共有方法，这就包括自身的所有public方法，和从基类继承的、从接口实现的所有public方法。

        //就是说，当这个类，域或者方法被设为私有访问，使用反射调用但是却没有权限时会抛出异常。
        Method[] methods = userBeanClass.getDeclaredMethods(); // 获取所有成员方法
        for (Method method : methods) {
            //反射可以获取方法上的注解，通过注解来进行判断
            if (method.isAnnotationPresent(Invoke.class)) { // 判断是否被 @Invoke 修饰
                //判断方法的修饰符是是static
                if (Modifier.isStatic(method.getModifiers())) { // 如果是 static 方法
                    //反射调用该方法
                    //类方法static方法可以直接调用，不必先实例化
                    method.invoke(null, "wingjay",2); // 直接调用，并传入需要的参数 devName
                } else {
                    //如果不是类方法，需要先获得一个实例再调用方法
                    //传入构造方法需要的变量类型
                    Class[] params = {String.class, long.class};
                    //获取该类指定类型的构造方法
                    //如果没有这种类型的方法会报错
                    Constructor constructor = userBeanClass.getDeclaredConstructor(params); // 获取参数格式为 String,long 的构造函数
                    //通过构造方法的实例来进行实例化
                    Object userBean = constructor.newInstance("wingjay", 11); // 利用构造函数进行实例化，得到 Object
                    if (Modifier.isPrivate(method.getModifiers())) {
                        method.setAccessible(true); // 如果是 private 的方法，需要获取其调用权限
//                          Set the {@code accessible} flag for this object to
//     * the indicated boolean value.  A value of {@code true} indicates that
//     * the reflected object should suppress Java language access
//     * checking when it is used.  A value of {@code false} indicates
//                                  * that the reflected object should enforce Java language access checks.
                    //通过该方法可以设置其可见或者不可见，不仅可以用于方法
                    //后面例子会介绍将其用于成员变量
                                        //打印结果
```

```
//            I'm a public method
// Hi wingjay, I'm a static methodI'm a private method
                    }
                    method.invoke(userBean); // 调用 method，无须参数
```

```
                }
            }
        }
    }
}
```

利用反射创建数组

数组在Java里是比较特殊的一种类型，它可以赋值给一个Object Reference。下面我们看一看利用反射创建数组的例子：

```
public class 用反射创建数组 {
    public static void main(String[] args) {
        Class<?> cls = null;
        try {
            cls = Class.forName("java.lang.String");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        Object array = Array.newInstance(cls,25);
        //往数组里添加内容
        Array.set(array,0,"hello");
        Array.set(array,1,"Java");
        Array.set(array,2,"fuck");
        Array.set(array,3,"Scala");
        Array.set(array,4,"Clojure");
        //获取某一项的内容
        System.out.println(Array.get(array,3));
        //Scala
    }

}
```

其中的Array类为java.lang.reflect.Array类。我们通过Array.newInstance()创建数组对象，它的原型是:

```
public static Object newInstance(Class<?> componentType, int length)
        throws NegativeArraySizeException {
        return newArray(componentType, length);
    }
```

而newArray()方法是一个Native方法，它在Hotspot JVM里的具体实现我们后边再研究，这里先把源码贴出来

```
private static native Object newArray(Class<?> componentType, int length)
        throws NegativeArraySizeException;
```

Java反射常见面试题

什么是反射?

反射是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为 Java 语言的反射机制。

哪里用到反射机制?

JDBC中，利用反射动态加载了数据库驱动程序。 Web服务器中利用反射调用了Sevlet的服务方法。 Eclispe等开发工具利用反射动态刨析对象的类型与结构，动态提示对象的属性和方法。 很多框架都用到反射机制，注入属性，调用方法，如Spring。

ArrayList的自定义序列化过程中，ObjectOutPutStream通过arraylist对象，通过反射动态判断arraylist是否实现了writeobject方法，如果实现了通过反射执行arraylist的自定义序列化方法

什么叫对象序列化，什么是反序列化，实现对象序列化需要做哪些工作?

对象序列化，将对象中的数据编码为字节序列的过程。 反序列化；将对象的编码字节重新反向解码为对象的过程。 JAVA提供了API实现了对象的序列化和反序列化的功能，使用这些API时需要遵守如下约定： 被序列化的对象类型需要实现序列化接口，此接口是标志接口，没有声明任何的抽象方法，JAVA编译器识别这个接口，自动的为这个类添加序列化和反序列化方法。 为了保持序列化过程的稳定，建议在类中添加序列化版本号。 不想让字段放在硬盘上就加transient 以下情况需要使用 Java 序列化： 想把的内存中的对象状态保存到一个文件中或者数据库中时候； 想用套接字在网络上传送对象的时候；想通过RMI（远程方法调用）传输对象的时候。

反射机制的优缺点?

优点：可以动态执行，在运行期间根据业务功能动态执行方法、访问属性，最大限度发挥了java的灵活性。 缺点：对性能有影响，这类操作总是慢于直接执行java代码。

动态代理是什么？有哪些应用?

动态代理是运行时动态生成代理类。 动态代理的应用有 Spring AOP数据查询、测试框架的后端 mock、rpc，Java注解对象获取等。

怎么实现动态代理?

JDK 原生动态代理和 cglib 动态代理。 JDK 原生动态代理是基于接口实现的，而 cglib 是基于继承当前类的子类实现的。

Java反射机制的作用

在运行时判断任意一个对象所属的类 在运行时构造任意一个类的对象 在运行时判断任意一个类所具有的成员变量和方法 在运行时调用任意一个对象的方法

如何使用Java的反射?

通过一个全限类名创建一个对象

Class.forName("全限类名"); 例如：com.mysql.jdbc.Driver Driver类已经被加载到 jvm中，并且完成了类的初始化工作就行了 类名.class; 获取Class<？> clz 对象 对象.getClass();

获取构造器对象，通过构造器new出一个对象

Clazz.getConstructor([String.class]); Con.newInstance([参数]); 通过class对象创建一个实例对象（就相当与new类名（）无参构造器) Cls.newInstance();

通过class对象获得一个属性对象

Field c=cls.getFields()：获得某个类的所有的公共（public）的字段，包括父类中的字段。 Field c=cls.getDeclaredFields()：获得某个类的所有声明的字段，即包括public、private和proteced，但是不包括父类的声明字段

通过class对象获得一个方法对象

Cls.getMethod("方法名",class……parameaType);（只能获取公共的）
Cls.getDeclareMethod("方法名");（获取任意修饰的方法，不能执行私有）
M.setAccessible(true);（让私有的方法可以执行） 让方法执行 1）. Method.invoke(obj实例对象,obj可变参数);-----（是有返回值的）

参考文章

http://www.cnblogs.com/peida/archive/2013/04/26/3038503.html
http://www.cnblogs.com/whoislcj/p/5671622.html
https://blog.csdn.net/grandgrandpa/article/details/84832343
http://blog.csdn.net/lylwo317/article/details/52163304
https://blog.csdn.net/qq_37875585/article/details/89340495

微信公众号