JAVA 的反射机制与动态代理

李海峰（QQ:61673110）-Andrew830314@163.com

运行时类型信息（RunTime Type Information，RTTI）使得你在程序运行时发现和使用类型信息。RTTI 主要用来运行时获取向上转型之后的对象到底是什么具体的类型。

1.Class 对象：

JAVA 使用 Class 对象来执行 RTTI。每个类都有一个 Class 对象，它用来创建这个类的所有对象，反过来说，每个类的所有对象都会关联同一个 Class 对象（对于数组来说，维数、类型一致的数组的 Class 对象才是相同的），每个对象的创建都依赖于 Class 对象的是否创建，Class 对象的创建发生在类加载（java.lang.ClassLoader）的时候。

java.lang.Class 类实现了 Serializable、GenericDeclaration、Type、AnnotatedElement 四个接口，分别实现了可序列化、泛型定义、类型、元数据（注解）的功能。

你可以把 Class 对象理解为一个类在内存中的接口代理（它代理了这个类的类型信息、方法签名、属性），JVM 加载一个类的时候首先创建 Class 对象，然后创建这个类的每个实例的时候都使用这个 Class 对象。

Class 只有一个私有的无参构造方法，也就是说 Class 的对象创建只有 JVM 可以完成。

如何验证同一个类的多个对象的 Class 对象是一个呢？

```
Cf1 cf1 = new Cf1();
Class clazz = Cf1.class;
System.out.println(cf1.getClass() == clazz);
```

我们知道==用来比较引用是否相等（也就是同一个引用），上面的输出语句结果是 true。那么 Class 对象是否相等是 JAVA 对象中唯一可以使用==判断的。

如何获取 Class 对象：

1.所有的引用数据类型（类-类型）的类名、基本数据类型都可以通过.class 方式获取其 Class 对象（**对于基本数据类型的封装类还可以通过.TYPE 的方式获取其 Class 对象，但要注意.TYPE 实际上获取的封装类对应的基本类型的 Class 对象的引用，那么你可以判断出 int.class==Integer.TYPE 返回 true，int.class==Integer.class 返回 false！**），通过这种方式不会初始化静态域，使用.class、.TYPE 的方式获取 Class 对象叫做类的字面常量；

2.Class 的 forName(String name)传入一个类的完整类路径也可以获得 Class 对象，但由于使用的是字符串，必须强制转换才可以获取泛型的 Class<T>的 Class 对象，并且你必须获取这个方法可能抛出的 ClassNotFoundException 异常。

2.对于引用数据类的引用（必须初始化），可以通过 Object 类继承的 getClass()方法获取这个引用的 Class 对象，由于引用已经被初始化，所以这种方式也不会初始化静态域，因为静态域已经被初始化过。另外，前面两种方式如果说是创建 Class 对象，那么这种方式应该是取得 Class 对象，因为类的实例已经被创建，那么 Class 对象也一定早就被创建。

Class 的常用方法：

- **forName(String name)**：这是一个静态方法，传入的参数是一个类的完整类路径的字符串，返回这个类的 Class 对象，前面说过 Class 对象的创建发生在类的加载时，所以这个方法会导致静态成员被调用；
- **forName(String name,boolean initialize,ClassLoader loader)**：这是上面的方

法的重载方法，initialize指定在创建Class对象时是否初始化这个类（即是否执行静态成员，由于在一次JVM的执行中，静态成员的初始化只类加载的时候执行一次，所以如果之前这个类已经被加载，那么即使initialize为true也不会再次执行静态成员的加载），loader指定使用哪个类加载器的实现类

（**Thread.*currentThread*().getContextClassLoader()**可以获取当前线程使用的类加载器）。forName(***)方法不可以获取基本数据类型的Class对象。

如果要测试initialize是否起作用，请不要在main()方法测试自身类，因为main()是静态方法，执行这个方法会导致静态域被初始化，所以你的initialize无论是true还是false，效果都是一样的。

- **asSubClass(Class superClass)**：这个方法是将父类的 class 对象作为参数传入，并将其强制转换成当前的 Class 对象（子类的 Class 对象）。

例：

```
Class<List> clazz = List.class;
Class<? extends List> subClazz = ArrayList.class.asSubclass(clazz);
System.out.println(subClazz.getCanonicalName());
```

注意红色的部分不能写成 `Class<ArrayList>`形式，因为 `asSubclass()`只知道是要转换成子类的 `Class` 对象，但不知道是哪个子类。

- **cast(Object o)**：这个方法是将传入的对象强制转换成 Class 对象所代表的类型的对象；
- **getClassLoader()**：返回创建当前 Class 对象的类加载器，默认的，应用程序获得的是 sun.misc.Launcher$AppClassLoader，Applet 程序获得的是 sun.applet.AppletClassLoader；
- **getCanonicalName()**：返回 JAVA 语言中所定义的底层类的规范化名称，如果没有规范化名称就返回 null，对于普通的引用数据类型，这个方法和 getName()方法都返回完整的类路径，对于数组（以字符串数组为例），getName()返回`[Ljava.lang.String;`，这个方法返回 `java.lang.String[]`；
- **getSimpleName()**：返回底层规范化名称的简写，也就是去掉包名；
- **getConstructors()**：返回 Class 对象的公有构造器的 java.lang.reflect.Contructor 对象数组，如果找不到匹配的构造方法，返回 NoSuchMethodExcetion 异常；
- **getConstructor(Class<?> ...parameterTypes)**：按照指定的可变参数列表，返回符合参数条件的公有构造方法的 Constructor，这里不可能是数组，因为构造方法不可能重复，如果找不到匹配的构造方法，返回 NoSuchMethodExcetion 异常；
- **getDeclaredConstructors()**：返回 Class 对象的所有构造器的 Constructor 对象，如果找不到匹配的构造方法，返回 NoSuchMethodExcetion 异常；
- **getDeclaredConstructor(Class<?> ...parameterTypes)**：按照指定的可变参数列表，返回符合参数条件的所有构造方法的 Constructor，这里不可能是数组，因为构造方法不可能重复，如果找不到匹配的构造方法，返回 NoSuchMethodExcetion 异常；
- **getFields()**：获取 Class 对象的所有公有成员属性的 java.lang.reflect.Field 数组；
- **getField(String name)**：按照字段名称获取公有字段的 Field 对象，注意 name 是区分大小写的；
- **getDeclaredFields()**：获取 Class 对象的所有成员属性的 Field 数组；
- **getDeclaredField(String name)**：按照字段名称获取所有字段的 Field 对象，注意 name 是区分大小写的；
- **getMethods()**：获取 Class 对象的公有方法（以及从父类继承的方法，但不包含构造

方法）的 java.lang.reflect.Method 数组；

- **getMethod(String name,Class<?> ...parameterTypes)**：按照 name 指定的方法名称，parameterTypes 指定的可变数组获取公有方法（以及从父类继承的方法，但不包含构造方法）的 Method 对象，注意 name 是区分大小写的；
- **getDeclaredMethods()**：获取 Class 对象的所有方法（不包含父类继承的方法，构造方法）的 Method 数组；
- **getDeclaredMethod(String name,Class<?> ...parameterTypes)**：按照 name 指定的方法名称，parameterTypes 指定的可变数组获取所有方法（不包含父类继承的方法，构造方法）的 Method 对象，注意 name 是区分大小写的；
- **getGenericInterface()**：以 Type 数组的形式返回 Class 对象的类直接实现的包含泛型参数的接口，返回顺序是 implements 后的接口顺序；
- **getInterface()**：以 Class 数组的形式返回 Class 对象的类直接实现的接口，返回顺序是 implements 后的接口顺序；
- **getModifiers()**：以 java.lang.reflect.Modifier 的常量值的形式返回 Class 对象的类的修饰的整型值的和；
- **getGenericSuperclass()**：以 Type 数组的形式返回 Class 对象的类直接实现的包含泛型参数的超类，返回顺序是 extends 后的接口顺序；
- **getSuperclass()**：以 Class 数组的形式返回 Class 对象的类直接实现的超类，返回顺序是 extends 后的接口顺序；
- **getName()**：以 String 形式返回 Class 对象所表示的实体的完整类名，基本数据类型返回自身，数组类型（以 String 数组为例）返回`[Ljava.lang.String;`，这个方法没有 **getCanonicalName()**返回的完整，但不是所有的类型都有底层的规范化名称；
- **getPackage()**：以 java.lang.reflect.Package 形式返回 Class 对象的类所在的包，基本数据类型、数组抛出异常；
- **getResource(String url)**：这个方法使用当前 Class 对象的 ClassLoader 实体加载 url 指定的资源，返回 java.net.URL 对象。如果 url 以 \ 开头，那么就从当前的 classPath 开始定位资源，否则就从当前 Class 对象的类所在的包开始定位资源，Hibernate 要求 *.hbm.xml 必须和 PO 类在一个包下，就是利用了没有 \ 开头的 url 传入这个方法实现的；
- **getResourceAsStream(String url)**：这个方法和上面的方法一样，只不过返回的是 java.io.InputStream 的对象；
- **isAssignableFrom(Class<?> class)**：判断当前 Class 对象的类是否是参数 class 的类的超类或者接口；
- **isInstance(Object o)**：判断参数 o 是否是 Class 对象的类的实例，相当于 o instanceOf Class。

isInstance()、instanceOf 关键字检查的某个实例是否是这个类型，具有继承关系的检查能力，即使是其子类的对象，也可以返回 true。但是 Class 对象是严格的类型，所以 super.class==sub.class 是一定返回 false 的。

- **newInstance()**：这个方法将会使得 Class 对象调用类中的公有无参构造方法实例化对象，，返回一个 Object 对象，大多数框架都会使用到这个方法，例如 EJB 容器、Spring 容器都会要求受管组件提供一个默认构造方法。

注意：以上的方法并不是对每种类型都可用，对于返回数组的方法，如果不可用则返回长度为 0 的数组，对于返回其他类型的方法，如果不可用则返回 null。例如：getConstructors()

方法对于基本数据类型、数组就不可用，那么返回一个长度为 0 的 Constructor 数组。
另外，上面的方法获取类中的元素大都是只能获取 public 的元素，可以获取全部元素的大都是含有 Declared 字符。
某些方法不具有递归特性，例如只能查找本类的元素，不能查找内部类或者其父类中的元素，如果你非要这么做，需要自行递归操作，例如：调用 getSuperClass()直到其返回 null 为止。

2.JAVA 的反射机制：

前面说过 RTTI 获取某个对象的确切类型，要求在这个对象在编译时必须已知，也就是必须已经在你的代码中存在完整的声明（T t），但是如果是运行时才会知晓的对象（例如网络中传递过来的字节），RTTI 就没办法工作了。
java.lang.Class 与 java.lang.reflect.*包中的类提供了一种有别于 RTTI（编译器在编译器时打开和检查*.class 文件）的反射机制（运行时打开和检查*.class 文件）。
JAVA 的反射功能相当强大，例如前面说过的类的复用方式---组合，如果是动态组合的新类就叫聚合，反射就可以完成聚合功能。

(1.)Field：这个类用于获取类中的字段信息以及访问这些字段的能力。

- **getObject(Object o)**：返回参数 o 的对象上的 Field 表示的字段的值；
- **setObject(Object o,Object value)**：将参数 o 的对象上的 Field 表示的字段的值设置为 value；
- **getBoolean(Object o)**：获取参数 o 的对象的 Field 表示的布尔类型的字段的值；
- **setBoolean(Object o,boolean value)**：将参数 o 的对象上的 Field 表示的布尔类型的字段的值设置为 value；
  … …对于基本数据类型，都拥有自己的 getXXX()、setXXX()方法。
- **getGenericType()**：返回 Field 表示的字段声明的泛型类型的 Type 实例；
- **getModifiers()**：以整型数值和的形式返回 Field 表示的字段的修饰符；
- **getType()**：返回 Field 表示的字段的类型的 Class 对象；
- **isEnumConstants()**：如果 Field 表示的字段是枚举类型，返回 true；
- **toGenericString()**：返回描述此字段的字符串（包含泛型信息），能够包含泛型信息是与 toString()方法的唯一区别。

例：

```
public class Reflect1 {
    private int i;

    protected String s;

    public List<String> list = new ArrayList<String>();

    public Reflect1() {
        System.out.println("default");

    }
```

```
    protected Reflect1(String name) {
        System.out.println(name);
    }

    public void f() {
        System.out.println("invoke f");
    }

    String mf(int j, Object... args) {
        System.out.println("invoke mf");
        return String.valueOf(j + args.length);
    }

}
public class Rf {
    public static void main(String[] args) throws Exception {
        Class<Reflect1> clazz = Reflect1.class;
        Reflect1 rf1 = clazz.newInstance();
        // 含有Declared字符串的是获取所有的元素，否则就只能获取公有元素
        Field[] f = clazz.getDeclaredFields();
        for (Field field : f) {
            // 设置这里本不具有访问权限的元素为可访问
            field.setAccessible(true);
            // 使用基本数据类型专有的API
            if (field.getType().getCanonicalName().equals("int")
                    || field.getType().getCanonicalName().equals(
                            "java.lang.Integer")) {
                field.setInt(rf1, 9);
            }
            System.out.println("Field is " + field.getName() + "\t"
                    + field.toGenericString() + "\t" + field.get(rf1));
        }
    }
}
```

控制台输出如下语句：

```
default
Field is i   private int net.ilkj.reflect.Reflect1.i   9
Field is s   protected java.lang.String net.ilkj.reflect.Reflect1.s null
Field is list
public java.util.List<java.lang.String> net.ilkj.reflect.Reflect1.list []
```

(2.)Method：这个类用于获取类中的方法的信息以及访问这些方法的能力。

- **getDefaultValue()**：返回此方法表示的注视成员的默认值；

- **getParameterAnnotations()**：返回一个 Annotation 的二维数组，表示方法的参数列表的参数的注解；
- **getExceptionTypes()**：返回这个方法抛出的（底层）异常的 Class 数组；
- **getGenericExceptionTypes()**：返回这个方法抛出的异常的 Type 数组，包含泛型信息；
- **getParameterTypes()**：返回这个方法的参数列表的 Class 对象数组；
- **getGenericParameterTypes()**：返回这个方法的参数列表的包含泛型信息的 Type 对象数组，例如一个 List<String>的参数前面的方法会获得 `java.util.List`，而这个方法会获得 `java.util.List<java.lang.String>`；
- **getReturnType()**：获取方法返回值的类型的 Class 对象；
- **getGenericReturnType()**：获取方法返回值的类型的 Type 对象，含有泛型信息；
- **isBridge()**：如果此方法是 bridge 方法，返回 true；
- **isVarArgs()**：如果此方法的参数列表中含有可变参数，返回 true；
- **invoke(Object o,Object... args)**：使用类的对象和可变参数列表调用这个方法，这个方法返回 Object 对象，也就是类中的这个方法的返回值；
- **getTypeParameters()**：以 java.lang.reflect.TypeVariable 数组返回 Method 对象的泛型方法的类型参数，数组的顺序是泛型定义的类型的顺序；
- **toGenericString()**：返回描述此方法的字符串（包含泛型信息），能够包含泛型信息是此方法与 toString()方法的唯一区别。

例：

```
public class Reflect1<X extends Exception> {
    private int i;

    protected String s;

    public List<String> list = new ArrayList<String>();

    public Reflect1() {
        System.out.println("default");

    }

    protected Reflect1(String name) {
        System.out.println(name);
    }

    public void f() throws NumberFormatException,
            ArrayIndexOutOfBoundsException {
        System.out.println("invoke f");
    }

    String mf(List<String> list, Object... args) throws X {
        System.out.println("invoke mf");
        return String.valueOf(list.size() + args.length);
```

```
    }
}
public class Rf {
    public static void main(String[] args) throws Exception {
        Class<Reflect1> clazz = Reflect1.class;
        Reflect1 rf1 = clazz.newInstance();
        Method[] m = clazz.getDeclaredMethods();
        for (Method method : m) {
            Class[] cs = method.getParameterTypes();
            System.out.println("Method is " + method.getName());
            // 获取泛型的异常
            Type[] t = method.getGenericExceptionTypes();
            for (Type type : t) {
                System.out.println("GenericException is " + type.toString());
            }
            // 获取普通的异常
            Class[] cc = method.getExceptionTypes();
            for (Class c : cc) {
                System.out.println("Exception is " + c.getCanonicalName());
            }
            if (cs.length == 2) {
                if (cs[0].getCanonicalName().equals("java.util.List")
                        && cs[1].getCanonicalName()
                                .equals("java.lang.Object[]")) {
                    Object o = method.invoke(rf1, new ArrayList<String>(),
                            new Object[] { "2" });
                    System.out.println("The Return Value is " + o);
                }
            }
        }
    }
}
```

控制台输出如下语句：

```
default
Method is mf
GenericException is X
Exception is java.lang.Exception
invoke mf
The Return Value is 1
Method is f
GenericException is class java.lang.NumberFormatException
GenericException is class java.lang.ArrayIndexOutOfBoundsException
Exception is java.lang.NumberFormatException
Exception is java.lang.ArrayIndexOutOfBoundsException
```

注意红色的代码，你就能区分出含有 Generic 字符串的方法和不含有这个字符串的方法的区别了。

(3.)Constructor：这个类用于获取类中的构造方法的信息以及访问这些构造方法的能力。
构造方法由于比较特殊，所以单独作为一个类，但是它的方法和 Method 没有任何区别。

(4.)Member 接口：Class、Field、Method、Constructor 都实现了 Member 接口，表明他们是类的成员。

- **getDeclaringClass()**：返回此成员所属的类的 Class 对象；
- **getModifiers()**：以整型值的和的形式返回这个成员的修饰符；
- **getName()**：返回此成员的名称的字符串；
- **isSynthetic()**：如果此成员是编译器引入的，返回 true。

(5.)AccessibleObject 类：Class、Field、Method（注意没有 Constructor，可见构造方法的访问级别绝对不可以改变！）都扩展了 AccessibleObject 类，这个类提供了取消 JAVA 访问权限的限制。

- **getAnnotation(Class<T> annotationClass)**：返回这个成员上指定注解类型的注解；
- **getAnnotations()**：返回这个成员上的所有注解的 Annotation 数组；
- **getDeclaredAnnotations()**：返回直接应用在此元素上的注解的 Annotation 数组；
- **isAccessible()**：如果该成员在当前位置可以被访问，返回 true；
- **isAnnotationPresent(Class<? extends Annotation> annotationClass)**：返回指定的 annotationClass 是否存在于当前的成员；
- **setAccessible(boolean bool)**：设置此元素是否可以在当前位置访问，这个方法在外面访问类中的私有成员时，非常有用处；
- **setAccessible(AccessibleObject[] array,boolean flag)**：这是一个静态方法，你可以将一组成员包装成 AccessibleObject 数组，一并设置他们的访问性。

(6.)Modifier：这个类存放了所有修饰符的常量值，以及通过这些常量值判断是哪种修饰符。
你可以将 getModifiers()方法返回的 int 类型传入这个类的 isXXX()静态方法，判断是哪种修饰符。

(7.)Array：这个类提供了动态创建和访问数组的能力。
这个类中的方法全部是静态方法，它允许在 get()、set()方法时进行扩展转型，如果进行收缩转型，将会抛出非法的参数异常。这个类的方法都是静态方法。

- **get(Object o,int index)**：获取指定数组中的索引位置 index 上的元素；
- **set(Object o,int index,Object value)**：设置指定数组中的索引位置 index 上的元素

为 value；

getXXX()、setXXX()可以直接设置基本数据类型；

- **newInstance(Class componentType,int length)**：创建一个指定类型为 componentType 的长度为 length 的一维数组；
- **newInstance(Class componentType,int... args)**：创建一个指定类型为 componentType 的 args 指定的维数的多维数组；

例：

```
public class Rf {
    public static void main(String[] args) throws Exception {
        // 创建一个二维数组，第一维有三个元素，第二维有两个元素
        Person[][] i = (Person[][]) Array.newInstance(Person.class, 3, 2);
        Array.set(i[0], 0, new Child());
        System.out.println(Array.get(i[0], 0));
    }
}

class Person {

}

class Child extends Person {

}
```

3.动态代理：

要看清楚什么是动态代理的，首先我们来看一下静态代理的做法。无论是那种代理方式，都存在代理对象和目标对象两个模型，所谓目标对象就是我们要生成的代理对象所代理的那个对象。

(1.) **包装的模式进行静态代理：**

接口：Animal

```
public interface Animal {

    void eat(String food);

    String type();

}
```

实现类：Monkey

```
public class Monkey implements Animal {

    @Override
    public String type() {
```

```
        String type = "哺乳动物";
        System.out.println(type);
        return type;
    }

    @Override
    public void eat(String food) {
        System.out.println("The food is " + food + " !");
    }

}
```

包装类：AnimalWrapper

```
public class AnimalWrapper implements Animal {

    private Animal animal;

    // 使用构造方法包装Animal的接口，这样所有的Animal实现类都可以被这个Wrapper
包装。
    public AnimalWrapper(Animal animal) {
        this.animal = animal;
    }

    @Override
    public void eat(String food) {
        System.out.println("+++Wrapped Before!+++");
        animal.eat(food);
        System.out.println("+++Wrapped After!+++");
    }

    @Override
    public String type() {
        System.out.println("---Wrapped Before!---");
        String type = animal.type();
        System.out.println("---Wrapped After!---");
        return type;
    }
}
```

运行程序：

```
AnimalWrapper aw = new AnimalWrapper(new Monkey());
aw.eat("香蕉");
aw.type();
```

控制台输出如下语句：

```
+++Wrapped Before!+++
The food is 香蕉 !
+++Wrapped After!+++
---Wrapped Before!---
哺乳动物
---Wrapped After!---
```

这里我们完成了对 Animal 所有子类的代理，在代理方法中，你可以加入一些自己的额外的处理逻辑，就像上面的+++、---输出语句一样。那么 Spring 的前置、后置、环绕方法通知，通过这种方式可以有限的模拟出来，以 Spring 的声明式事务为例，无非就是在调用包装的目标方法之前处开启事务，在之后提交事务，这样原有的业务逻辑没有受到任何事务管理代码的侵入。

这种方式的静态代理，缺点就是当 Animal 接口中增加了新的方法，那么包装类中也必须增加这些新的方法。

(2.) 继承的模式进行静态代理：

继承类：MyMonkey

```
public class MyMonkey extends Monkey {

    @Override
    public void eat(String food) {
        System.out.println("+++Wrapped Before!+++");
        super.eat(food);
        System.out.println("+++Wrapped After!+++");
    }

    @Override
    public String type() {
        System.out.println("---Wrapped Before!---");
        String type = super.type();
        System.out.println("---Wrapped After!---");
        return type;
    }

}
```

这个例子很容易看懂，我们采用继承的方式对 MyMonkey 中的方法进行代理，运行效果与包装的模式效果是一样的。
但这种方式的缺点更明显，那就是不能实现对 Animal 所有子类的代理，与包装的模式相比，大大缩小了代理范围。

(3.) 基于 Proxy 的动态代理：

JAVA 自带的动态代理是基于 java.lang.reflect.Proxy、java.lang.reflect.InvocationHandler 两个类来完成的，使用 JAVA 反射机制。

Proxy 类中的几个方法都是静态的，通常，你可以使用如下两种模式创建代理对象：

①

Object proxy = Proxy.*newProxyInstance*(定义代理对象的类加载器**,**
要代理的目标对象的归属接口数组**,**回调接口**InvocationHandler);**

②

Class proxyClass=Proxy.getProxyClass(定义代理对象的类加载器**,**
要代理的目标对象的归属接口数组**);**

Object proxy = proxyClass.getConstructor(
new Class[] { InvocationHandler.class }).newInstance(
回调接口**InvocationHandler);**

第一种方式更加直接简便，并且隐藏了代理$Proxy0 对象的结构。

JDK 的动态代理会动态的创建一个$Proxy0 的类，这个类继承了 Proxy 并且实现了要代理的目标对象的接口，但你不要试图在 JDK 中查找这个类，因为它是动态生成的。$Proxy0 的结构大致如下所示：

public final class $Proxy0 extends Proxy implements 目标对象的接口**1,**接口**2,...{**

//构造方法

Public $Proxy0(InvocationHandler h){

... ...

}

}

从上面的类结构，你就可以理解为什么第二种创建代理对象的方法为什么要那么写了。

下面我们看一个具体的实例：

接口 1：Mammal（哺乳动物）

```
public interface Mammal {

    void eat(String food);

    String type();

}
```

接口 2：Primate（灵长类动物）

```
public interface Primate {
```

```
    void think();

}
```

实现类：Monkey

```
public class Monkey implements Mammal, Primate {

    @Override
    public String type() {
        String type = "哺乳动物";
        System.out.println(type);
        return type;
    }

    @Override
    public void eat(String food) {
        System.out.println("The food is " + food + " !");
    }

    @Override
    public void think() {
        System.out.println("思考! ");
    }

}
```

回调类：MyInvocationHandler

```
public class MyInvocationHandler implements InvocationHandler {

    private Object obj;

    public MyInvocationHandler(Object obj) {
        this.obj = obj;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        System.out.println("Invoke method Before!");
        Object returnObject = method.invoke(obj, args);
        System.out.println("Invoke method After!");
        return returnObject;
    }

}
```

注意：这里我们使用构造方法将要代理的目标对象传入回调接口，当然你也可以用其他的方式，但无论如何，一个代理对象应该是与一个回调接口对应的。

运行程序：

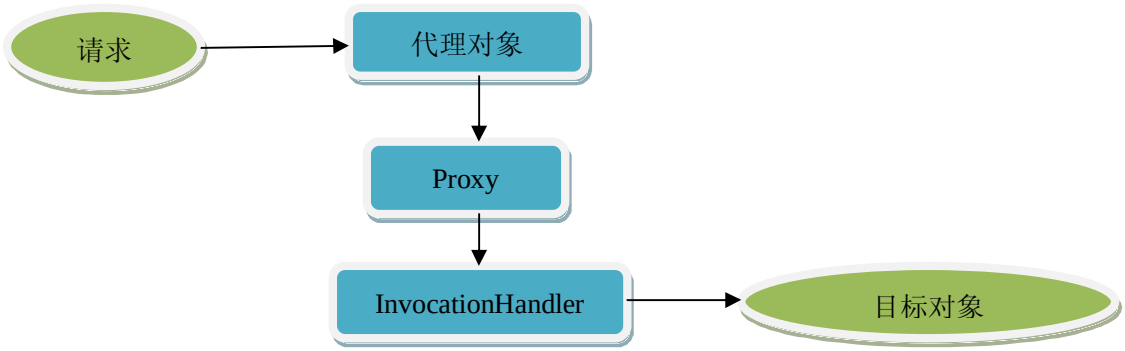
```
// 第一种创建动态代理的方法
// Object proxy = Proxy.newProxyInstance(Monkey.class.getClassLoader(),
// Monkey.class.getInterfaces(), new MyInvocationHandler(
// new Monkey()));

// 第二种创建动态代理的方法
Class<?> proxyClass = Proxy.getProxyClass(
              Monkey.class.getClassLoader(),
              Monkey.class.getInterfaces());
Object proxy = proxyClass.getConstructor(
              new Class[] { InvocationHandler.class }).newInstance(
              new MyInvocationHandler(new Monkey()));

Mammal mammal = (Mammal) proxy;
mammal.eat("香蕉");
mammal.type();

Primate primate = (Primate) proxy;
primate.think();
```

控制台输出：

```
Invoke method Before!
The food is 香蕉 !
Invoke method After!
Invoke method Before!
哺乳动物
Invoke method After!
Invoke method Before!
思考!
Invoke method After!
```

你可以看到动态代理成功了，在目标对象的方法调用前后都输出了我们打印的语句。其实 Spring 中对接口的动态代理，进而做诸如声明式事务的 AOP 操作也是如此，只不过代码会更加复杂。

我们用下面的图说明上面的执行过程：



我们看到目标对象的方法调用被 Proxy 拦截，在 InvocationHandler 中的回调方法中通过反射调用。这种动态代理的方式实现了对类的方法的运行时修改。

JDK 的动态代理有个缺点，那就是不能对类进行代理，只能对接口进行代理，想象一下我们的 Monkey 如果没有实现任何接口，那么将无法使用这种方式进行动态代理（实际上是因为$Proxy0 这个类继承了 Proxy，JAVA 的继承不允许出现多个父类）。但准确的说这个问题不应该是缺点，因为良好的系统，每一个类都是应该有一个接口的。

从上面知道$Proxy0 是动态代理对象的所属类型，但由于这个类型根本不存在，我们如何鉴别一个对象是一个普通的对象还是动态代理对象呢？Proxy 类中提供了 isProxyClass(Class c)方法鉴别与此。

下面我们介绍一下 InvocationHandler 这个接口，它只有一个方法 invoke()需要实现，这个方法会在目标对象的方法调用的时候被激活，你可以在这里控制目标对象的方法的调用，在调用前后插入一些其他操作（譬如：鉴权、日志、事务管理等）。Invoke()方法的后两个参数很好理解，一个是调用的方法的 Method 对象，另一个是方法的参数，第一个参数有些需要注意的地方，这个 proxy 参数就是我们使用 Proxy 的静态方法创建的动态代理对象，也就是$Proxy0 的实例（这点你可以在 Eclipse 的断点调试中看到 proxy 的所属类型确实是$Proxy0）。由于$Proxy0 在 JDK 中不是静态存在的，因此你不可以把第一个参数 Object proxy 强制转换为$Proxy0 类型，因为你根本就无法从 Classpath 中导入$Proxy0。那么我们可以把 proxy 转为目标对象的接口吗？因为$Proxy0 是实现了目标对象的所有的接口的，答案是可以的。但实际上这样做的意义不大，因为你会发现转换为目标对象的接口之后，你调用接口中的任何一个方法，都会导致 invoke()的调用陷入死循环而导致堆栈溢出。如下所示：

```
public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        Mammal mammal=(Mammal)proxy;
        mammal.type();
        … …
}
```

这是因为目标对象的大部分的方法都被代理了，你在 invoke()通过代理对象转换之后的接口调用目标对象的方法，依然是走的代理对象，也就是说当 mammal.type()方法被激活时会立即导致 invoke()的调用，然后再次调用 mammal.type()方法，… …从而使方法调用进入死循环，就像无尽的递归调用。

那么 invoke()方法的第一个参数到底干什么用的呢？其实一般情况下这个参数都用不到，除

非你想获得代理对象的类信息描述，因为它的 getClass()方法的调用不会陷入死循环。如下所示：

```
Class<?> c = proxy.getClass();
Method[] methods = c.getDeclaredMethods();
for (Method m : methods) {
    System.out.println(m.getName());
}
```

这里我们可以获得代理对象的所有的方法的名字，你会看到控制台输出如下信息：

```
eat
think
type
equals
toString
hashCode
```

我们看到 proxy 确实动态的把目标对象的所有的接口中的方法都集中到了自己的身上。

这里还要注意一个问题，那就是从 Object 身上继承的方法 hashCode()等的调用也会导致陷入死循环，为什么 getClass()不会呢？因为 getClass()方法是 final 的，不可以被覆盖，所以也就不会被 Proxy 代理。但不要认为 Proxy 不可以对 final 的方法进行动态代理，因为 Proxy 面向的是 Monkey 的接口，而不是 Monkey 本身，所以即便是 Monkey 在实现 Mammal、Primate 接口的时候，把方法都变为 final 的，也不会影响到 Proxy 的动态代理。

(4.) 基于 CGLIB 的动态代理：

CGLIB 是一个开源的动态代理框架，它的出现补充了 JDK 自带的 Proxy 不能对类实现动态代理的问题。CGLIB 是如何突破限制，对类也能动态代理的呢？这是因为 CGLIB 内部使用了另一个字节码框架 ASM，类似的字节码框架还有 Javassist、BCEL 等，但 ASM 被认为是性能最好的一个。但这类字节码框架要求你对 JAVA 的 Class 文件的结构、指令集都比较了解，CGLIB 对外屏蔽了这些细节问题。由于 CGLIB 使用 ASM 直接操作字节码，因此效率要比 Proxy 高，但这里所说的效率是指代理对象的性能，在创建代理对象时，Proxy 是要比 CGLIB 效率高的。

下面我们简单看一个 CGLIB 完成动态代理的例子。

目标类：Monkey

```
public class Monkey {

    public String type() {
        String type = "哺乳动物";
        System.out.println(type);
        return type;
    }

    public final void eat(String food) {
        System.out.println("The food is " + food + " !");
```

```
    }

    public void think() {
        System.out.println("思考！");
    }

}
```

我们看到这个 Monkey 类有两点变化，第一点是没有实现任何接口，第二点是 eat()方法是 final 的。

回调接口：

```
import java.lang.reflect.Method;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

public class MyMethodInterceptor implements MethodInterceptor {

    @Override
    public Object intercept(Object obj, Method method, Object[] args,
            MethodProxy proxy) throws Throwable {
        System.out.println("********************");
        Object o = proxy.invokeSuper(obj, args);
        System.out.println("++++++++++++++++++");
        return o;
    }

}
```

运行程序：

```
import net.sf.cglib.proxy.Enhancer;

public class Cglib {

    public static void main(String[] args) {
        Monkey monkey = (Monkey) Enhancer.create(Monkey.class,
                new MyMethodInterceptor());
        monkey.eat("香蕉");
        monkey.type();
        monkey.think();
    }

}
```

控制台输出：

```
The food is 香蕉 !
******************
```
哺乳动物
```
++++++++++++++++++
******************
```
思考!
```
++++++++++++++++++
```
你会发现 eat()方法没有被代理，因为在它的前后没有输出 MethodInterceptor 中的打印语句。这是因为 CGLIB 动态代理的原理是使用 ASM 动态生成目标对象的子类，final 方法不能被子类覆盖，自然也就不能被动态代理，这也是 CGLIB 的一个缺点。

我们看到 CGLIB 进行动态代理的编写过程与 Proxy 没什么太大的不同，Enhancer 是 CGLIB 的入口，通过它创建代理对象，同时为代理对象分配一个 net.sf.cglib.proxy.Callback 回调接口，用于执行回调。我们常用的是 MethodInterceptor 接口，这个接口继承自 Callback 接口，用于执行方法拦截。

MethodInterceptor 接口中的 intercept()方法中的参数分别为代理对象、被调用的方法的 Method 对象，方法的参数、CGLIB 提供的方法代理对象，一般来说调用目标方法时我们使用最后一个参数，而不是 JAVA 反射的第二个参数，因为 CGLIB 使用 ASM 的字节码操作，代理对象的执行效率比反射机制更高。

(4-1.)关于 Enhancer：

这个类的静态方法 create()可以用于创建代理对象，CGLIB 使用动态生成子类的方式完成动态代理，那么默认情况下，子类会继承父类的无参构造进行实例化，如果你想调用父类的其他构造器，可以使用 create(Class[] argumentsType,Object[] arguments)这个重载方法分别指定构造方法的参数类型、传递参数。

但是一般情况下，我们会创建 Enhancer 的实例来完成动态代理，而不是使用静态方法 create()，因为使用 Enhancer 的实例，你可以获取更多的功能。使用 Enhancer 实例的代码如下所示：

```
Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(Monkey.class);
enhancer.setCallback(new MyMethodInterceptor());
Monkey monkey = (Monkey) enhancer.create();
monkey.eat("香蕉");
monkey.type();
monkey.think();
```

通过 Enhancer 的实例你可以设置是否使用缓存、生成策略等。

(4-2.)关于 Callback：

除了 MethodInterceptor 以外，CGLIB 还提供了一些内置的回调处理。

- net.sf.cglib.proxy.FixedValue

为提高性能，FixedValue 回调对强制某一特别方法返回固定值是有用的。

- net.sf.cglib.proxy.NoOp

NoOp 回调把对方法调用直接委派到这个方法在父类中的实现，相当于不进行代理。

- net.sf.cglib.proxy.LazyLoader

当实际的对象需要延迟装载时，可以使用 LazyLoader 回调。一旦实际对象被装载，它将被每一个调用代理对象的方法使用。

(4-3.)关于 CallbackFilter：

Enhancer 的 setCallbacks(Callback[] callbacks)方法可以为代理对象设置一组回调器，你可以配合 CallbackFilter 为不同的方法使用不同的回调器。

```
Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(Monkey.class);
enhancer.setCallbacks(new Callback[] { new MyMethodInterceptor(),
                NoOp.INSTANCE });
enhancer.setCallbackFilter(new CallbackFilter() {

    @Override
    public int accept(Method arg0) {
        // 方法type使用回调组中的第二个回调器
        if (arg0.getName().equals("type"))
            return 1;
        else
            return 0;
    }

});
Monkey monkey = (Monkey) enhancer.create();
monkey.eat("香蕉");
monkey.type();
monkey.think();
```

这里我们指定 type()方法使用第二个回调器（也就是什么也不做的 NoOp，相当于不对 type()方法进行代理），其余的使用第一个回调器。这也就是说 CallbackFilter 的 accept()方法返回的是回调器的索引值。

CGLIB 被 Hibernate、Spring 等很多开源框架在内部使用，用于完成对类的动态代理，Spring 中的很多 XML 配置属性的 proxy-target-class，默认都为 false，其含义就是默认不启用对目标类的动态代理，而是对接口进行动态代理。某些情况下，如果你想对 Struts2 的 Action 或者 Spring MVC 的 Controller 进行动态代理，你会发现默认 Spring 会报告找不到$Proxy0 的 xxx 方法，这是因为一般我们都不会给控制层写一个接口，而是直接在实现类中写请求方法，这样 JDK 自带的 Proxy 是找不到这些方法的，因为他们不在接口中，此时你就要设置 proxy-target-class=”true”，并引入 CGLIB、ASM 的类库，Spring 的动态代理就可以正常工作了。

5.Spring 的 AOP 编程：

Spring 的 AOP 底层通过动态代理（接口代理使用 Proxy、类代理使用 CGLIB）来做支持，有了前面的知识，Spring 的 AOP 就比较好理解了，就是在运行时通过动态代理，动态的将某段代码织入到你的程序，从而在不影响原有的业务代码时增加了新的功能。

AOP 涉及到如下几个概念：

- **切面 Aspect：**切面就是一个关注点的模块化，譬如：事务管理、日志记录、权限管理都是所谓的切面。
- **连接点 Joinpoint：**程序执行时的某个特定的点，在 Spring 中就是一个方法的执行。
- **通知 Advice：**通知就是在切面的某个连接点上执行的操作，也就是事务管理、日志记录等的代码。
- **切入点 Pointcut：**切入点是指描述某一类特定的连接点，也就是说指定某一类要织入通知的方法。
- **目标对象 Target：**就是被 AOP 动态代理的目标对象。

(5-1.)启用注解风格的 AOP：

要启用注解风格的 AOP，需要配置以下的内容：

```
<aop:aspectj-autoproxy/>
```

这个元素的 proxy-target-class 属性默认为 false，也就是使用 Proxy 接口动态代理，当然你可以指定为 true，强制使用 CGLIB 类动态代理。

目标对象 Target：

```
@Service("myService")
public class MyServiceImpl implements IMyService {

    @Override
    public User m1(User u, String id) {
        System.out.println("m1 executed!");
        if (id == null || "".equals(id))
            return u;
        else
            return new User("1", "Hello World!");
    }

    @Override
    public User m2(User u, String id) {
        System.out.println("m2 executed!");
        if (id == null || "".equals(id))
            return u;
        else
            return new User("2", "Hello World!");
    }

}
```

这里我们省略接口 IMyService 的源码，MyServiceImpl 就是一个 Spring 的业务层组件，没什么特别的，我们要完成的任务是使用 Spring AOP 动态的在 m1、m2 方法中插入一些其他的逻辑。

切面 Aspect：

```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

// 使用@Aspect 注解的类， Spring 将会把它当作一个特殊的 Bean（一个切面），也就是不对这个类本身进行动态代理。
@Component("aspectDemo")
@Aspect
public class AspectDemo {

    // 定义切入点表达式
    private final static String EXP = "execution (* *.m1(..)) || execution (* *.m2(..))";

    //前置方法通知
    // @Before的value属性对哪些切入点织入通知
    // 这个方法必须是返回值为void，无参或者只有一个连接点类型的参数，从这个JoinPoint参数上可以获取到被代理的对象的相关信息。
    @Before(EXP)
    public void before(JoinPoint point) {
        System.out.println("Before " + point.getSignature().getName());
    }

    // 后置方法通知
    @After(EXP)
    public void after(JoinPoint point) {
        System.out.println("After " + point.getSignature().getName());
    }

    // 后置方法返回值通知
    // @AfterReturning(EXP)
    // 默认情况下，@AfterReturning与@After没什么不同，但你可以使用下面的方式，定义argNames参数，表示被注解的方法的一个参数，然后returning指定返回值使用这个参
```

```
数，实际上就是Spring会把方法的返回值传递给你指定的参数obj。
    @AfterReturning(pointcut = EXP, returning = "obj", argNames = "obj")
    public void afterReturning(JoinPoint point, Object obj) {
        System.out.println("After Returning " + obj);
    }

    // 异常通知
    @AfterThrowing(pointcut = EXP, throwing = "e", argNames = "e")
    public void afterThrowing(JoinPoint point, Exception e) {
        System.out.println("After Throwing " + e);
    }

    // 环绕方法通知，环绕方法通知要注意必须给出调用之后的返回值，否则被代理的方法会停止调用并返回null，除非你真的打算这么做。
    // 只有环绕通知才可以使用JoinPoint的子类ProceedingJoinPoint，这个连接点类型可以调用代理的方法，并获取、改变返回值。
    @Around(EXP)
    public Object around(ProceedingJoinPoint point) throws Throwable {
        System.out.println("Around Before " +
                point.getSignature().getName());
        // 调用目标对象的方法并获取返回值
        Object o = point.proceed(point.getArgs());
        System.out.println("Around After " +
                point.getSignature().getName());
        return o;
    }

}
```

上面的这段代码展示了 Spring 的四种通知策略，在你调用 IMyService 的 m1 或者 m2 方法时，通过控制台输出的语句，即可看出这四种通知的含义。另外，从名字上也很好理解，Around 就是环绕通知，也就是在目标对象的方法执行前后各插入一些逻辑进行包围（环绕）。

控制台输出如下所示：

```
Around Before m1
Before m1
m1 executed!
Around After m1
After Returning net.ilkj.service.model.User@9934d4
After m1
```

红色的文字是目标方法的调用，在其前后就是 AOP 的通知调用。

(5-2.)启用 XML 风格的 AOP：

使用 XML 风格的 AOP 也很简单，如下所示：

```
<aop:config>
```

```
    <aop:pointcut id="pointcut" expression="表达式" />
    <aop:aspect ref="一个类似于AspectDemo的类，只不过不使用@Aspect对其注解">
        <aop:before method="AspectDemo类中的拦截方法的名称"
                    pointcut-ref="pointcut" />
    </aop:aspect>
</aop:config>
```

联合注解风格的 AOP，上面的 XML 片段很好理解，你可以在 aop:config 中定义多个 aop:pointcut，也就是指定多个切入点，然后在 aop:aspect 的 ref 中指定切面，在 aop:before 等子元素中指定通知的类型及调用的切面中的方法。

(5-3.)切入点表达式：

前面的例子中看到了切入点表达式形如下面的格式：

```
execution (modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern (param-pattern)
        throws-pattern?)
```

这是 Spring 中使用最多的切入点表达式。?表示可以不配置，也就是说只有方法的名字 name-pattern、方法的参数 param-pattern 是必须的。对于 param-pattern 之外的其余部分，可以使用*作为通配符，表示任意，例如：`execution (* *.m1(..))`就是执行任意返回值、任意类中的 m1 方法时进行拦截。

参数的统配稍微复杂一些，其中(..)表示参数为 0 个或者多个，且类型任意；(*)表示任意类型的一个参数；(*,String)表示第一个参数为任意类型，第二个参数为 String 类型；什么都不写表示无参数。

多个表达式可以使用&&、||、!进行运算，因为表达式返回的是布尔值。