链滴

Java 反射与注解

作者：openshell
原文链接：https://ld246.com/article/1585995404535
来源网站：链滴

反射

动态语言是一类运行时可以改变自身结构的语言，在运行中新的函数、对象、甚至代码可以被引进。

Java 是一门静态语言，但是由于 Java 拥有反射的特性，使之成为了“准动态语言”。

反射机制允许程序在运行期间借助 Reflection API 取得任何内的内部信息，并能够直接操作任意对象部属性及方法。

类加载完成后，在堆内存方法区生成一个 Class 类型的对象（一个类只有一个 Class 对象），该对象含了完整的类的结构信息

反射的优缺点

关于优缺点直接用代码说说最直观的

优点：代码简洁，提高代码复用率，增加了程序的灵活性，避免程序死。

定义一个接口

```
/**
 * 一个格式工厂，提供图片转换为jpg格式的接口
 * @author openshell
 * @since 2020-04-04
 */
interface FactoryOfFormat {
    void toJpg();
}
```

创建两个实现类：

```
/**
 * png类，实现格式工厂接口
 * @author openshell
 * @since 2020-04-04
 */
class Png implements FactoryOfFormat {
    @Override
    public void toJpg() {
        System.out.println("png format to jpg!");
    }
}

/**
 * psd类，实现格式工厂接口
 * @author openshell
 * @since 2020-04-04
 */
class Psd implements FactoryOfFormat {
    @Override
    public void toJpg() {
        System.out.println("psd format to jpg");
```

```
    }
}
```

测试类：

```
/**
 * <p>
 *创建测试类，提供两种获取实例的方法
 * 若使用getInstance每次新增加一个图片格式，就需要在该代码中新增一个判断
 * 若使用getInstanceByReflection,每次只需要传入对应的key值，通过反射可以很方便的话获取到其Cass对象，
 * 并通过newInstance方法获取到对应的实例
 * </p>
 *
 * @author openshell
 * @since 2020-04-04
 */
public class TestReflection {
    public static void main(String[] args) {
        FactoryOfFormat png = getInstance("Png");
        assert png != null;
        png.toJpg();
        FactoryOfFormat png1 = getInstanceByReflection("Png");
        png1.toJpg();
    }

    public static FactoryOfFormat getInstance(String key) {
        if (StringUtils.equals(key, "Png")) {
            return new Png();
        } else if (StringUtils.equals(key, "Psd")) {
            return new Psd();
        }
        return null;
    }

    public static FactoryOfFormat getInstanceByReflection(String key) {
        Class clazz;
        FactoryOfFormat factoryOfFormat = null;
        try {
            clazz = Class.forName("com.cqz.study.reflection." + key);
            factoryOfFormat = (FactoryOfFormat) clazz.newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return factoryOfFormat;
    }
}
```

缺点：相比直接调用性能有较大幅度的下降，但若在实际应用中只是尽使用反射，其性能可以忽略；反射破坏了Java封装的特性，会影响代码安全性

下面测试以下反射的性能影响：

1. 创建一个实体类

```
/**
 * 实体类
 */
@Data
class User{
    private Long id;
    private String name;
    private Integer age;
}
```

2. 创建测试类，编写测试方法

```
/**
 * <p>
 * test01为直接调用速度最快
 * test02通过反射调用速度最慢
 * test03关闭了Java访问权限检查，速度快于test02，慢于test01
 * </p>
 *
 * @author openshell
 * @since 2020-04-03
 */
public class PerformanceTest {
    public static void test01() {
        User user = new User();
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 1000000000; i++) {
            user.getName();
        }
        long endTime = System.currentTimeMillis();
        System.out.println("test01直接调用,耗时:" + (endTime - startTime) + "ms");

    }

    public static void test02() throws NoSuchMethodException, InvocationTargetException, Ille
alAccessException {
        User user = new User();
        Class clazz = user.getClass();
        Method getName = clazz.getMethod("getName", null);
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 1000000000; i++) {
            getName.invoke(user, null);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("test02通过反射调用,耗时:" + (endTime - startTime) + "ms");
    }

    public static void test03() throws NoSuchMethodException, InvocationTargetException, Ille
alAccessException {
        User user = new User();
        Class clazz = user.getClass();
        Method getName = clazz.getMethod("getName", null);
        getName.setAccessible(true);
        long startTime = System.currentTimeMillis();
```

```
        for (int i = 0; i < 1000000000; i++) {
            getName.invoke(user, null);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("test03通过反射调用（关闭了Java访问权限检查）,耗时:" + (endTime - sta
tTime) + "ms");
    }


    public static void main(String[] args) throws NoSuchMethodException, IllegalAccessExcepti
n, InvocationTargetException {
        test01();
        test02();
        test03();
    }
}
```

```
test01直接调用,耗时:5ms

test02通过反射调用,耗时:3346ms

test03通过反射调用（关闭了Java访问权限检查）,耗时:1936ms
```

获取 Class 类的方法

1. 通过类名获取

Class clazz=User.class ;

2. 通过实例获取

Class clazz = user.getClass();

3. 通过类路径获取，该方法可能抛出 ClassNotFoundException

Class clazz = Class.forName("com.cqz.study.reflection")

4. 内置基本数据类型可以直接用类名。Type

5. ClassLoader 获取

类加载器

加载 →链接 →初始化

双亲委派机制——保证安全性

```
public class ClassLoaderTest {
    public static void main(String[] args) throws ClassNotFoundException {
        //获取系统类的加载器
        ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();
        System.out.println(systemClassLoader);

        //获取系统加载器的父类加载器→扩展类加载器
        ClassLoader parent = systemClassLoader.getParent();
        System.out.println(parent);
```

```
        //获取引导类加载器，这一层用C++编写，是Jvm自带的类加载器，负责Java平台核心库，该加器无法直接获取，下面代码返回null
        ClassLoader parent1 = parent.getParent();
        System.out.println(parent1);

        //获取我们自己编写的类，用的什么加载器，可以看到是sun.misc.Launcher$AppClassLoader 18b4aac2
        Class<User> userClass = User.class;
        ClassLoader userClassLoader = userClass.getClassLoader();
        System.out.println(userClassLoader);
        //下面代码返回null，证明了JDK自带类是由引导类加载器加载的，Java无法获取到这一层
        Class<?> stringClazz = Class.forName("java.lang.Object");
        ClassLoader stringClassLoaders = stringClazz.getClassLoader();
        System.out.println(stringClassLoaders);
    }
}
```

反射操纵数据

1. 创建两个注解

```
/**
 * 应用与表名的注解
 *
 * @author openshell
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@interface TableAnnotation {
    String value();
}

/**
 * 应用于字段的注解
 *
 * @author openshell
 */
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@interface FiledAnnotation {
    String name();

    String type();

    int length();
}
```

2. 创建数据库关系映射类（使用了lombok）,并使用注解。

```
@Data
@AllArgsConstructor
@NoArgsConstructor
```

```
@ToString
@ResponseBody
@TableAnnotation("t_colleague")
class Colleague {
    @FiledAnnotation(name = "c_number", type = "bigint", length = 8)
    private Long number;
    @FiledAnnotation(name = "c_name", type = "varchar", length = 8)
    private String name;
    @FiledAnnotation(name = "c_name", type = "int", length = 4)
    private int age;
}
```

3.编写测试类

```
public class Action {

    public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldExcepti
n {
        //1.获取通过反射获取Colleague的Class对象
        Class clazz = Class.forName("com.cqz.study.reflection.Colleague");
        //     Annotation[] annotations = clazz.getDeclaredAnnotations();
        Annotation[] annotations = clazz.getAnnotations();

        //2.获取类注解
        for (Annotation annotation : annotations) {
            System.out.println(annotation);
        }
        //3. 获取类注解value的值
        TableAnnotation tableAnnotation = (TableAnnotation) clazz.getAnnotation(TableAnnotat
on.class);
        String value=tableAnnotation.value();
        System.out.println(value);

        //4.获取属性上的注解  getField(String name)只能获取public的字段，包括父类的；  getDeclar
dField(String name)只能获取自己声明的各种字段，包括public，protected，private。
        Field field = clazz.getDeclaredField("name");
        FiledAnnotation fieldAnnotation = field.getAnnotation(FiledAnnotation.class);
        System.out.println(fieldAnnotation.name());
        System.out.println(fieldAnnotation.type());
        System.out.println(fieldAnnotation.length());
    }
}
```