反射详解

主要内容

反射介绍
创建对象过程
反射具体实现
反射机制的效率
章节总结

学习目标

知识点	要求
反射介绍	了解
创建对象过程	掌握
反射具体实现	掌握
反射机制的效率	掌握
章节总结	了解

一、 反射介绍

1 什么是反射

Java 反射机制是 Java 语言一个很重要的特性，它使得 Java 具有了“动态性”。在 Java 程序运行时，对于任意的一个类，我们能不能知道这个类有哪些属性和方法呢？对于任意的一个对象，我们又能不能调用它任意的方法？答案是肯定的！这种动态获取类的信息以及动态调用对象方法的功能就来自于 Java 语言的反射（Reflection）机制。

2 反射的作用

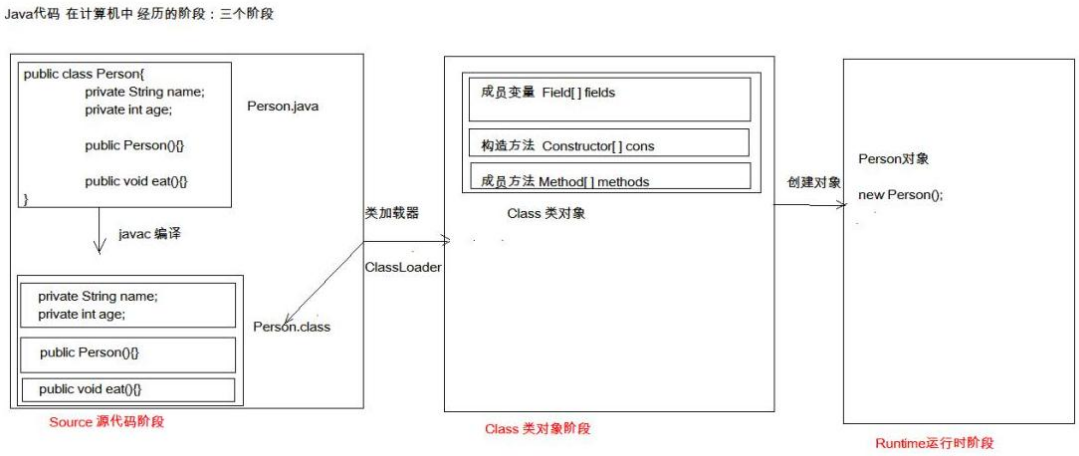
简单来说两个作用，RTTI（运行时类型识别）和 DC（动态创建）。

我们知道反射机制允许程序在运行时取得任何一个已知名称的 class 的内部信息，包括

其 modifiers(修饰符)，fields(属性)，methods(方法)等，并可于运行时改变 fields 内容或调用 methods。那么我们便可以更灵活的编写代码，代码可以在运行时装配，无需在组件之间进行源代码链接，降低代码的耦合度；还有动态代理的实现等等；但是需要注意的是反射使用不当会造成很高的资源消耗！

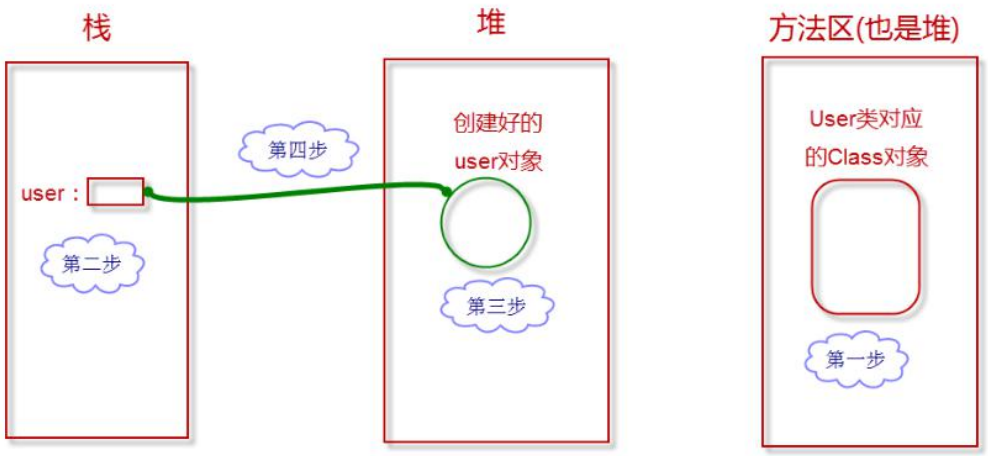
二、 创建对象过程

1 创建 Java 对象的三个阶段



2 创建对象时内存结构

Users user = new Users();



实际上，我们在加载任何一个类时都会在方法区中建立“这个类对应的 Class 对象”，由于“Class 对象”包含了这个类的整个结构信息，所以我们可以通过这个“Class 对象”来操作这个类。

我们要使用一个类，首先要加载类；加载完类之后，在堆内存中，就产生了一个 Class 类型的对象（一个类只有一个 Class 对象），这个对象就包含了完整的类的结构信息。我们可以通过这个对象知道类的结构。这个对象就像一面镜子，透过这个镜子可以看到类的结构，所以，我们形象的称之为：反射。 因此，“Class 对象”是反射机制的核心。

三、 反射的具体实现

1 获取 Class 对象的三种方式

- 通过 getClass()方法。
- 通过.class 静态属性。
- 通过 Class 类中的静态方法 forName()。

1.1创建 Users 类

```
public class Users {
    private String username;
    private int userage;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public int getUserage() {
        return userage;
    }
```

```
    public void setUserage(int userage) {
        this.userage = userage;
    }
}
```

1.2通过 getClass()方法获取 Class 对象

```
/**
 * getClass()获取Class对象
 */
public class GetClass1 {
    public static void main(String[] args) {
          Users users  =  new Users();
          Class clazz = users.getClass();
          Class clazz2 = users.getClass();
        System.out.println(clazz);
        System.out.println(clazz.getName());
        System.out.println(clazz == clazz2);
    }
}
```

1.3通过.class 静态属性获取 Class 对象

```
/**
 * .class静态属性获取Class对象
 */
public class GetClass2 {
    public static void main(String[] args) {
        Class clazz = Users.class;
        Class clazz2 = Users.class;
        System.out.println(clazz);
        System.out.println(clazz.getName());
        System.out.println(clazz == clazz2);
    }
}
```

1.4通过 forName()获取 Class 对象

```
/**
 * 通过Class.forName("class Name")获取Class对象
 */
public class GetClass3 {
    public static void main(String[] args)throws Exception {
        Class clazz = Class.forName("com.bjsxt.Users");
        Class clazz2 = Class.forName("com.bjsxt.Users");
        System.out.println(clazz);
        System.out.println(clazz.getName());
        System.out.println(clazz == clazz2);
    }
}
```

2 获取类的构造方法

2.1方法介绍

方法名	描述
getDeclaredConstructors()	返回 Constructor 对象的一个数组，这些对象反映此 Class 对象表示的类声明的所有构造方法。
getConstructors()	返回一个包含某些 Constructor 对象的数组，这些对象反映此 Class 对象所表示的类的所有公共（public）构造方法。
getConstructor(Class<?>... parameterTypes)	返回一个 Constructor 对象，它反映此 Class 对象所表示的类的指定公共（public）构造方法。
getDeclaredConstructor(Class<?>... parameterTypes)	返回一个 Constructor 对象，该对象反映此 Class 对象所表示的类或接口的指定构造方法。

2.2方法使用

2.2.1 修改 Users 类

```
public class Users {
    private String username;
    private int userage;
    public Users(){
    }
    public Users(String username,int userage){
```

```
        this.username= username;
        this.userage=userage;
    }
    public  Users(String username){
        this.username= username;
    }
    private Users(int userage){
        this.userage = userage;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public int getUserage() {
        return userage;
    }

    public void setUserage(int userage) {
        this.userage = userage;
    }
}
```

2.2.2 获取构造方法

```
public class GetConstructor {
    public static void main(String[] args)throws Exception {
        Class clazz = Users.class;
        Constructor[]  arr = clazz.getDeclaredConstructors();
        for(Constructor c:arr){
            System.out.println(c);
        }
        System.out.println("----------------------");
        Constructor[]  arr1 = clazz.getConstructors();
        for(Constructor c:arr1){
            System.out.println(c);
        }
        System.out.println("------------------------");
        Constructor c =   clazz.getDeclaredConstructor(int.class);
```

```
        System.out.println(c);

        System.out.println("------------------------");
        Constructor c1 =   clazz.getConstructor(null);
        System.out.println(c1);
    }
}
```

2.3通过构造方法创建对象

```
public class GetConstructor2 {
    public static void main(String[] args)throws Exception {
        Class clazz = Users.class;
        Constructor constructor =
clazz.getConstructor(String.class,int.class);
        Object o = constructor.newInstance("OldLu",18);
        Users users = (Users)o;

System.out.println(users.getUsername()+"\t"+users.getUserage());
    }
}
```

3 获取类的成员变量

3.1方法介绍

方法名	描述
getFields()	返回 Field 类型的一个数组,其中包含 Field 对象的所有公共(public)字段。
getDeclaredFields()	返回 Field 类型的一个数组,其中包含 Field 对象的所有字段。
getField(String fieldName)	返回一个公共成员的 Field 指定对象。
getDeclaredField(String fieldName)	返回一个 Field 指定对象。

3.2方法使用

3.2.1 修改 Users 类

```
public class Users {
    private String username;
    public int userage;
```

```
    public Users(){
    }
    public Users(String username,int userage){
        this.username= username;
        this.userage=userage;
    }
    public  Users(String username){
        this.username= username;
    }
    private Users(int userage){
        this.userage = userage;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public int getUserage() {
        return userage;
    }

    public void setUserage(int userage) {
        this.userage = userage;
    }
}
```

3.2.2 获取成员变量

```
public class GetField {
    public static void main(String[] args)throws Exception {
        Class clazz = Users.class;
        Field[] fields = clazz.getFields();
        for(Field f:fields){
            System.out.println(f);
            System.out.println(f.getName());
        }
        System.out.println("------------------------");
        Field[] fields2 = clazz.getDeclaredFields();
```

```
        for(Field f:fields2){
            System.out.println(f);
            System.out.println(f.getName());
        }
        System.out.println("------------------------");
        Field field = clazz.getField("userage");
        System.out.println(field);
        System.out.println("----------------------");
        Field field1 = clazz.getDeclaredField("username");
        System.out.println(field1);
    }
}
```

3.3操作成员变量

```
public class GetField2 {
    public static void main(String[] args)throws Exception  {
        Class clazz = Users.class;
        Field field = clazz.getField("userage");

        //对象实例化

        Object obj = clazz.newInstance();

        //为成员变量赋予新的值

        field.set(obj,18);

        //获取成员变量的值

        Object o = field.get(obj);
        System.out.println(o);


    }
}
```

4 获取类的方法

4.1方法介绍

方法名	描述
getMethods()	返回一个 Method 类型的数组,其中包含 所有公共(public)方法。
getDeclaredMethods()	返回一个 Method 类型的数组,其中包含 所有方法。
getMethod(String name, Class<?>... parameterTypes)	返回一个公共的 Method 方法对象。

getDeclaredMethod(String name, Class<?>... parameterTypes)	返回一个方法 Method 对象

4.2方法使用

4.2.1 修改 Users 类

```
public class Users {
    private String username;
    public int userage;
    public Users(){
    }
    public Users(String username,int userage){
        this.username= username;
        this.userage=userage;
    }
    public  Users(String username){
        this.username= username;
    }
    private Users(int userage){
        this.userage = userage;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public int getUserage() {
        return userage;
    }

    public void setUserage(int userage) {
        this.userage = userage;
    }
    private void suibian(){
        System.out.println("Hello Oldlu");
    }
}
```

4.2.2 获取方法

```
public class GetMethod {
    public static void main(String[] args)throws Exception{
        Class clazz = Users.class;
        Method[] methods = clazz.getMethods();
        for(Method m: methods){
            System.out.println(m);
            System.out.println(m.getName());
        }
        System.out.println("----------------------------");
        Method[] methods2 = clazz.getDeclaredMethods();
        for(Method m: methods2){
            System.out.println(m);
            System.out.println(m.getName());
        }
        System.out.println("----------------------------");
        Method method = clazz.getMethod("setUserage", int.class);
        System.out.println(method.getName());
        System.out.println("----------------------------");
        Method method1 = clazz.getDeclaredMethod("suibian");
        System.out.println(method1.getName());
    }
}
```

4.3调用方法

```
public class GetMethod2 {
    public static void main(String[] args)throws  Exception {
        Class clazz = Users.class;
        Method method = clazz.getMethod("setUsername",String.class);

        //实例化对象

        Object obj = clazz.newInstance();

        //通过 setUserName 赋值

        method.invoke(obj,"oldlu");


        //通过 getUserName 获取值

        Method method1 = clazz.getMethod("getUsername");
        Object value = method1.invoke(obj);
        System.out.println(value);
```

```
    }
}
```

5 获取类的其他信息

```
public class GetClassInfo {
    public static void main(String[] args) {
        Class clazz = Users.class;

        //获取类名

        String className = clazz.getName();
        System.out.println(className);

        //获取包名

        Package p = clazz.getPackage();
        System.out.println(p.getName());

        //获取超类

        Class superClass = clazz.getSuperclass();
        System.out.println(superClass.getName());

        //获取该类实现的所有接口

        Class[] interfaces = clazz.getInterfaces();
        for(Class inter:interfaces){
            System.out.println(inter.getName());
        }
    }
}
```

6 反射应用案例

需求：根据给定的方法名顺序来决定方法的执行顺序。

```
public class ReflectDemo {
    public void method1(){
        System.out.println("Method1.......");
    }
    public void method2(){
        System.out.println("Method2.......");
    }
```

```
    public void method3(){
        System.out.println("Method3.......");
    }
}
```

```
public class Test {
    public static void main(String[] args)throws Exception {
        ReflectDemo rd = new ReflectDemo();
        if(args != null && args.length > 0){

            //获取 ReflectDemo 的 Class 对象

            Class clazz = rd.getClass();

            //通过反射获取 ReflectDemo 下的所有方法

            Method[] methods = clazz.getMethods();
            for(String str :args){
                for(int i=0;i<methods.length;i++){
                    if(str.equalsIgnoreCase(methods[i].getName())){
                        methods[i].invoke(rd);
                        break;
                    }
                }
            }
        }else{
            rd.method1();
            rd.method2();
            rd.method3();
        }
    }
}
```

四、 反射机制的效率

由于 Java 反射是要解析字节码，将内存中的对象进行解析，包括了一些动态类型，而 JVM 无法对这些代码进行优化。因此，反射操作的效率要比那些非反射操作低得多！

接下来我们做个简单的测试来直接感受一下反射的效率。

1 反射机制的效率测试

```
public class Test2 {
    public static void main(String[] args) {
        try {

            //反射耗时

            Class clazz = Class.forName("com.bjsxt.Users");
            Users users = (Users) clazz.newInstance();
            long reflectStart = System.currentTimeMillis();
            Method method = clazz.getMethod("setUsername",
String.class);
            for(int i=0;i<100000000;i++){
                method.invoke(users,"oldlu");
            }
            long reflectEnd = System.currentTimeMillis();

            //非反射方式的耗时

            long start = System.currentTimeMillis();
            Users u = new Users();
            for(int i=0;i<100000000;i++){
                u.setUsername("oldlu");
            }
            long end = System.currentTimeMillis();

            System.out.println("反射执行时间："+(reflectEnd -
reflectStart));

            System.out.println("普通方式执行时间："+(end - start));

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

2 setAccessible 方法

setAccessible()方法：

setAccessible 是启用和禁用访问安全检查的开关。值为 true 则指示反射的对象在使用时应该取消 Java 语言访问检查。值为 false 则指示反射的对象应该实施 Java 语言访问检

查;默认值为 false。

由于 JDK 的安全检查耗时较多.所以通过 setAccessible(true)的方式关闭安全检查就可以达到提升反射速度的目的。

```
public class Test3 {
    public static void main(String[] args)throws Exception {
        Users users = new Users();
        Class clazz = users.getClass();
        Field field = clazz.getDeclaredField("username");

        //忽略安全检查

        field.setAccessible(true);
        field.set(users,"oldlu");
        Object object = field.get(users);
        System.out.println(object);
        System.out.println("------------------------------");
        Method method = clazz.getDeclaredMethod("suibian");
        method.setAccessible(true);
        method.invoke(users);
    }
}
```

五、 本章总结

- Java 反射机制是 Java 语言一个很重要的特性，它使得 Java 具有了“动态性”。
- 反射机制的优点：
  - 更灵活。
  - 更开放。
- 反射机制的缺点：
  - 降低程序执行的效率。
  - 增加代码维护的困难。
- 获取 Class 类的对象的三种方式：

- 运用 getClass()。
- 运用.class 语法。
- 运用 Class.forName()（最常被使用）。

❑ 反射机制的常见操作

- 动态加载类、动态获取类的信息（属性、方法、构造器）。
- 动态构造对象。
- 动态调用类和对象的任意方法。
- 动态调用和处理属性。
- 获取泛型信息。
- 处理注解。