链滴

java 基础注解和反射（完结）

作者：yscxy

原文链接：https://ld246.com/article/1607333354613

来源网站：链滴

四个元注解


```
package net.yscxy.annoction;

import java.lang.annotation.*;

/**
 * @Author WangFuKun
 * @create 2020/12/4 20:20
 */
//测试元注解
```

```
public class Test02 {
    @MyAnotation
    public void test() {

    }
}

//定义一个注解
//Target表示我们的注解可以用在哪些地方
//Retention表示我们的注解在什么时候还有效runtime>class>source
//Documented表示是否将我们的注解生成在javadoc中
//Inherited子类可以继承父类
@Target(value = {ElementType.METHOD, ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
@Documented()
@Inherited()
@interface MyAnotation {

}
```

Target表示我们的注解可以用在哪些地方

Retention表示我们的注解在什么时候还有效runtime>class>source

Documented表示是否将我们的注解生成在javadoc中

Inherited子类可以继承父类

如何自定义注解

- 使用 **@interface**自定义注解时，自动继承了java.lang.annotation.Annotation接口

- **分析：**
  - ✓ @ interface用来声明一个注解，格式：public @ interface 注解名 { 定义内容 }
  - ✓ 其中的每一个方法实际上是声明了一个配置参数.
  - ✓ 方法的名称就是参数的名称.
  - ✓ 返回值类型就是参数的类型（返回值只能是基本类型,Class，String，enum）.
  - ✓ 可以通过default来声明参数的默认值
  - ✓ 如果只有一个参数成员，一般参数名为value
  - ✓ 注解元素必须要有值，我们定义注解元素时，经常使用空字符串,0作为默认值 .

```
package net.yscxy.annoction;

import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.Test;
```

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * @Author WangFuKun
 * @create 2020/12/4 21:30
 */
//自定义注解
public class Test03 {
    @Test
    @MyAnnotation2(name = "wfk", age = 10)
    public void test() {

    }
}

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation2 {
    //注解的参数：参数类型+参数名();
    String name() default "";

    int age();

    int id() default -1;//如果默认值为-1，代表不存在，如果找不到就返回-1

    String[] schools() default {"清华大学", };
}
```

反射机制

静态语言VS动态语言

静态 VS 动态语言

动态语言

- 是一类在运行时可以改变其结构的语言：例如新的函数、对象、甚至代码可以被引进，已有的函数可以被删除或是其他结构上的变化。通俗点说就是在运行时代码可以根据某些条件改变自身结构。
- 主要动态语言：Object-C、C#、JavaScript、PHP、Python等。

静态语言

- 与动态语言相对应的，运行时结构不可变的语言就是静态语言。如Java、C、C++。

- Java不是动态语言，但Java可以称之为“准动态语言”。即Java有一定的动态性，我们可以利用反射机制获得类似动态语言的特性。Java的动态性让编程的时候更加灵活!

Java Reflection

Java Reflection

- Reflection（反射）是Java被视为动态语言的关键，反射机制允许程序在执行期借助于Reflection API取得任何类的内部信息，并能直接操作任意对象的内部属性及方法。

```
Class c = Class.forName("java.lang.String")
```

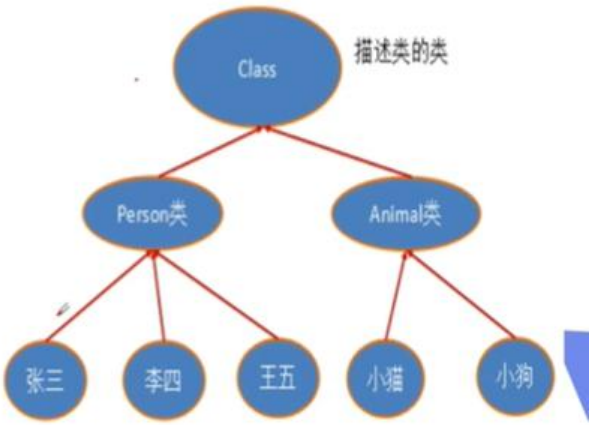
- 加载完类之后，在堆内存的方法区中就产生了一个Class类型的对象（一个类只有一个Class对象），这个对象就包含了完整的类的结构信息。我们可以通过这个对象看到类的结构。这个对象就像一面镜子，透过这个镜子看到类的结构，所以，我们形象的称之为：反射

原文链接：java 基础注解和反射（完结）

在Object类中定义了以下的方法，此方法将被所有子类继承

```
public final Class getClass()
```

- 以上的方法返回值的类型是一个Class类，此类是Java反射的源头，实际上所谓反射从程序的运行结果来看也很好理解，即：可以通过对象反射求出类的名称。



获取Class类的实例的几种方案

a) 若已知具体的类，通过类的class属性获取，该方法最为安全可靠，程序性能最高。

```
Class clazz = Person.class;
```

b) 已知某个类的实例，调用该实例的getClass()方法获取Class对象

```
Class clazz = person.getClass();
```

c) 已知一个类的全类名，且该类在类路径下，可通过Class类的静态方法forName()获取，可能抛出ClassNotFoundException

```
Class clazz = Class.forName("demo01.Student");
```

d) 内置基本数据类型可以直接用类名.Type

e) 还可以利用ClassLoader我们之后讲解

```
package net.yscxy.reflection;

/**
 * @Author 野生程序员 http://yscxy.net/
 * @create 2020/12/5 10:29
 */
public class Test03 {
    public static void main(String[] args) throws ClassNotFoundException {
        Person person = new Student();
        System.out.println("这个人是->" + person.name);
        //方式一：通过对象获得
        Class c1 = person.getClass();
        System.out.println(c1);
        //方式二：forname获得
        Class c2 = Class.forName("net.yscxy.reflection.Student");
```

```
        System.out.println(c2);
        //方式三：通过类名.class获得
        Class c3 = Student.class;
        System.out.println(c3);
        //方式四：基本内置类型的包装类都有一个Type属性
        Class c4 = Integer.TYPE;
        System.out.println(c4);
        //获得父类类型
        Class c5 = c1.getSuperclass();
        System.out.println(c5);
    }
}

class Person {
    String name;

    public Person() {
    }

    public Person(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                '}';
    }
}

class Student extends Person {
    public Student() {
        this.name = "学生";
    }


}

class Teacher extends Person {
    public Teacher() {
        this.name = "老师";
    }
}
```

代码二

```
package net.yscxy.reflection;

import java.lang.annotation.ElementType;

/**
 * @Author 野生程序员 http://yscxy.net/
 * @create 2020/12/5 10:46
```

```
 */
//所有类的class
public class Test04 {
    public static void main(String[] args) {
        Class c1 = Object.class;//类
        Class c2 = Comparable.class;//接口
        Class c3 = String[].class;//一维数组
        Class c4 = int[][].class;//二维数组
        Class c5 = Override.class;//注解
        Class c6 = ElementType.class;//枚举
        Class c7 = Integer.class;//基本数据类型
        Class c8 = void.class;//void
        Class c9 = Class.class;//Class
        System.out.println(c1);
        System.out.println(c2);
        System.out.println(c3);
        System.out.println(c4);
        System.out.println(c5);
        System.out.println(c6);
        System.out.println(c7);
        System.out.println(c8);
        System.out.println(c9);
    }
}
```

什么时候会发生类的初始化

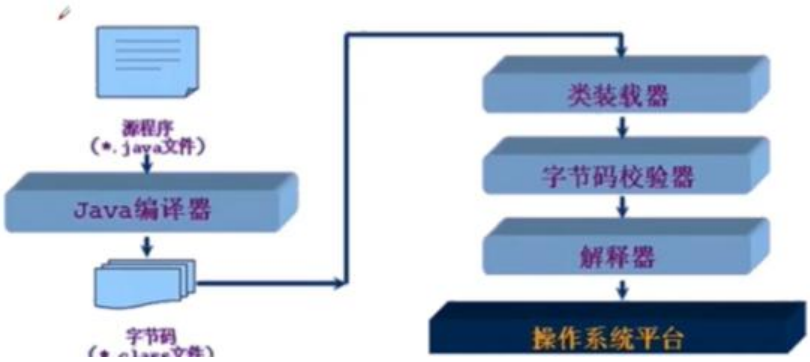
- 类的主动引用（一定会发生类的初始化）
  - 当虚拟机启动，先初始化main方法所在的类
  - new一个类的对象
  - 调用类的静态成员（除了final常量）和静态方法
  - 使用java.lang.reflect包的方法对类进行反射调用
  - 当初始化一个类，如果其父类没有被初始化，则先会初始化它的父类
- 类的被动引用（不会发生类的初始化）
  - 当访问一个静态域时，只有真正声明这个域的类才会被初始化。如：当通过子类引用父类的静态变量，不会导致子类初始化
  - 通过数组定义类引用，不会触发此类的初始化
  - 引用常量不会触发此类的初始化（常量在链接阶段就存入调用类的常量池中了）

```
package net.yscxy.reflection;

/**
 * @Author 野生程序员 http://yscxy.net/
 * @create 2020/12/5 11:10
 */
public class Test06 {
    static {
        System.out.println("main类被加载...");
    }
```

```
    public static void main(String[] args) throws ClassNotFoundException {
        //1.主动引用
        //Son son = new Son();
        //反射也会产生主动引用引用
        //Class.forName("net.yscxy.reflection.Son");
        //不会产生类的引用的方法
        //System.out.println(Son.b);
        //Son[] arr = new Son[5];
        //System.out.println(Son.M);
    }
}

class Father {
    static int b = 2;

    static {
        System.out.println("父类被加载...");
    }
}

class Son extends Father {
    static {
        System.out.println("子类被加载...");
    }

    static int m = 100;
    static final int M = 1;
}
```

类加载器的作用

- 类加载的作用：将class文件字节码内容加载到内存中，并将这些静态数据转换成方法区的运行时数据结构，然后在堆中生成一个代表这个类的java.lang.Class对象，作为方法区中类数据的访问入口。
- 类缓存：标准的JavaSE类加载器可以按要求查找类，但一旦某个类被加载到类加载器中，它将维持加载（缓存）一段时间。不过JVM垃圾回收机制可以回收这些Class对象



获取类的方法、字段、构造器、等等

```
package net.yscxy.reflection;

import java.io.File;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.util.Arrays;

/**
 * @Author 野生程序员 http://yscxy.net/
 * @create 2020/12/5 15:49
 */
public class Test08 {
    public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldExcepti
n, NoSuchMethodException {
        Class c1 = Class.forName("net.yscxy.reflection.User");
        //获得类的名字
        System.out.println(c1.getName());
        System.out.println(c1.getSimpleName());
        //获得类的属性
        Field[] fields = c1.getFields();//只能找到public属性
        for (int i = 0; i < fields.length; i++) {
            System.out.println(fields[i]);
        }
        Field name = c1.getDeclaredField("name");
        System.out.println(name);
        //获得类的方法
        System.out.println("==========================获得类的方法==============
============");
        Arrays.asList(c1.getMethods()).forEach(System.out::println);//获得本类以及父类的全部publi
方法
        System.out.println("==========================获得本类的所有方法==========
================");
        Arrays.asList(c1.getDeclaredMethods()).forEach(System.out::println);//获得本类的所有方法
        //获得指定方法
        System.out.println("获得指定方法");
        Method getName = c1.getMethod("getName", null);
        Method getAge = c1.getMethod("getAge", null);
        System.out.println(getName);
        System.out.println(getAge);
        System.out.println("获得指定的构造器");
        Constructor[] constructors = c1.getConstructors();
        Arrays.asList(constructors).forEach(System.out::println);
        System.out.println("++++++++++++");
        Arrays.asList(c1.getDeclaredConstructors()).forEach(System.out::println);
        System.out.println("获得指定的构造器");
        Constructor declaredConstructor = c1.getDeclaredConstructor(int.class, String.class, int.c
ass);
        System.out.println("指定：" + declaredConstructor);
    }
}
```

通过反射动态创建对象

- 创建类的对象：调用Class对象的newInstance()方法
  - 1）类必须有一个无参数的构造器。
  - 2）类的构造器的访问权限需要足够。

思考？难道没有无参的构造器就不能创建对象了吗？只要在操作的时候明确的调用类中的构造器，并将参数传递进去之后，才可以实例化操作。

- 步骤如下：

1）通过Class类的getDeclaredConstructor(Class ... parameterTypes)取得本类的指定形参类型的构造器

2）向构造器的形参中传递一个对象数组进去，里面包含了构造器中所需的各个参数。

3）通过Constructor实例化对象

```
package net.yscxy.reflection;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

/**
 * @Author 野生程序员 http://yscxy.net/
 * @create 2020/12/6 8:43
 */
//通过反射动态创建对象
public class Test09 {
    public static void main(String[] args) throws ClassNotFoundException, IllegalAccessExcepti
n, InstantiationException, NoSuchMethodException, InvocationTargetException, NoSuchField
xception {
        Class<?> c1 = Class.forName("net.yscxy.reflection.User");
        //构造对象
        User user = (User) c1.newInstance();
        System.out.println(user);
        //通过构造器创建对象
        Constructor<?> declaredConstructor = c1.getDeclaredConstructor(int.class, String.class,
nt.class);
        User user13 = (User) declaredConstructor.newInstance(1, "测试", 1);
        System.out.println(user13);
        //通过反射获取一个方法,invoke激活的意思
        User user3 = (User) c1.newInstance();
        Method setName = c1.getDeclaredMethod("setName", String.class);
        setName.invoke(user3, "野生程序员");
        System.out.println(user3);
        //通过反射操作属性
        User user4 = (User) c1.newInstance();
        Field name = c1.getDeclaredField("name");
        //设置属性是否可以被访问，也就是取消他的安全监测，不能直接访问私有属性，只有关掉之后
能访问
        name.setAccessible(true);
        name.set(user4, "野生程序员2");
        System.out.println(user4);
```

```
    }
}
```

三种方式的额性能检测

```
package net.yscxy.reflection;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

/**
 * @Author 野生程序员 http://yscxy.net/
 * @create 2020/12/7 16:07
 */
//分析反射性能问题
public class Test10 {
    public static void main(String[] args) throws NoSuchMethodException, IllegalAccessExcepti
on, InvocationTargetException {
        test01();
        test02();
        test3();
    }

    //普通方式调用
    public static void test01() {
        User user = new User();
        long starTime = System.currentTimeMillis();
        for (int i = 0; i < 1000000000; i++) {
            user.getName();
        }
        long endTime = System.currentTimeMillis();
        System.out.println("普通方式耗费时间->" + (endTime - starTime));
    }

    //反射方式调用
    public static void test02() throws NoSuchMethodException, InvocationTargetException, Ille
alAccessException {
        User user = new User();
        Class c1 = user.getClass();
        Method getName = c1.getDeclaredMethod("getName", null);
        long starTime = System.currentTimeMillis();
        for (int i = 0; i < 1000000000; i++) {
            getName.invoke(user, null);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("反射方式耗费时间->" + (endTime - starTime));
    }

    //反射方式调用 并且关闭安全检测
    public static void test3() throws NoSuchMethodException, InvocationTargetException, Illega
AccessException {
        User user = new User();
        Class c1 = user.getClass();
```

```
        Method getName = c1.getDeclaredMethod("getName", null);
        getName.setAccessible(true);
        long starTime = System.currentTimeMillis();
        for (int i = 0; i < 1000000000; i++) {
            getName.invoke(user, null);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("反射并且关闭安全检查方式耗费时间->" + (endTime - starTime));
    }
}
```

运行结果

普通方式耗费时间->5

反射方式耗费时间->4928

反射并且关闭安全检查方式耗费时间->2171

通过反射操作泛型

- Java采用泛型擦除的机制来引入泛型，Java中的泛型仅仅是给编译器javac使用的,确保数据的安全性和免去强制类型转换问题，但是，一旦编译完成，所有和泛型有关的类型全部擦除
- 为了通过反射操作这些类型，Java新增了 ParameterizedType，GenericArrayType，TypeVariable 和 WildcardType 几种类型来代表不能被归一到Class类中的类型但是又和原始类型齐名的类型.
- ParameterizedType：表示一种参数化类型,比如Collection<String>
- GenericArrayType：表示一种元素类型是参数化类型或者类型变量的数组类型
- TypeVariable：是各种类型变量的公共父接口
- WildcardType：代表一种通配符类型表达式

```
package net.yscxy.reflection;

import java.lang.reflect.Method;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
import java.util.List;
import java.util.Map;

/**
 * @Author 野生程序员 http://yscxy.net/
 * @create 2020/12/7 16:24
 */
//通过反射获取泛型
public class Test11 {
    public void test01(Map<String, User> map, List<User> list) {
        System.out.println("test01");
```

```
    }

    public Map<String, User> test02() {
        System.out.println("test02");
        return null;
    }

    public static void main(String[] args) throws NoSuchMethodException {
        //获取参数的泛型类型
        Method method = Test11.class.getMethod("test01", Map.class, List.class);
        Type[] genericExceptionTypes = method.getGenericParameterTypes();
        for (Type ge : genericExceptionTypes) {
            System.out.println(ge);
            //instanceof 严格来说是Java中的一个双目运算符，用来测试一个对象是否为一个类的实例
            if (ge instanceof ParameterizedType) {
                System.out.println("开始打印...");
                Type[] actualTypeArguments = ((ParameterizedType) ge).getActualTypeArguments()

                for (Type actualTypeArgument : actualTypeArguments) {
                    System.out.println(actualTypeArgument);
                }
            }
        }
        System.out.println("获取返回值的泛型的参数类型...");
        method = Test11.class.getMethod("test02", null);
        Type genericReturnType = method.getGenericReturnType();
        System.out.println(genericReturnType);
        if (genericReturnType instanceof ParameterizedType) {
            Type[] actualTypeArguments = ((ParameterizedType) genericReturnType).getActualTy
eArguments();
            for (Type actualTypeArgument : actualTypeArguments) {
                System.out.println(actualTypeArgument);
            }
        }
    }
}
```

运行结果

```
java.util.Map<java.lang.String, net.yscxy.reflection.User>
开始打印...
class java.lang.String
class net.yscxy.reflection.User
java.util.List<net.yscxy.reflection.User>
开始打印...
class net.yscxy.reflection.User
获取返回值的泛型的参数类型...
java.util.Map<java.lang.String, net.yscxy.reflection.User>
class java.lang.String
class net.yscxy.reflection.User
```

反射操作注解

了解一下什么叫ORM

◆ Object relationship Mapping --> 对象关系映射

```
class Student{
    int id;
    String name;
    int age;
}
```

id	name	age
001	秦疆	3
002	老沈	30

◆ 类和表结构对应
◆ 属性和字段对应
◆ 对象和记录对应

◆ 要求：利用注解和反射完成类和表结构的映射关系

```
package net.yscxy.reflection;

import java.lang.annotation.*;
import java.lang.reflect.Field;

/**
 * @Author 野生程序员 http://yscxy.net/
 * @create 2020/12/7 16:59
 */
public class Test12 {
    public static void main(String[] args) throws NoSuchFieldException {
        Class c1 = Student2.class;
        //通过反射获取注解
        Annotation[] annotations = c1.getAnnotations();
        for (Annotation a : annotations) {
            System.out.println(a);
        }
        //获取注解的value的值
        TableY tableY = (TableY) c1.getAnnotation(TableY.class);
        System.out.println(tableY.value());
        //获得类指定的注解
        Field f = c1.getDeclaredField("name");
        FiledY annotation = f.getAnnotation(FiledY.class);
        System.out.println(annotation.columnName());
        System.out.println(annotation.type());
        System.out.println(annotation.length());

    }
}

//数据
@TableY("du_student")
class Student2 {
```

```
    @FiledY(columnName = "db_id", type = "int", length = 10)
    private int id;
    @FiledY(columnName = "db_age", type = "int", length = 10)
    private int age;
    @FiledY(columnName = "db_name", type = "varchar", length = 3)
    private String name;

    public Student2() {
    }

    public Student2(int id, int age, String name) {
        this.id = id;
        this.age = age;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Student2{" +
                "id=" + id +
                ", age=" + age +
                ", name='" + name + '\'' +
                '}';
    }
}

//类名注解
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@interface TableY {
```

```
    String value();
}

//属性的注解
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@interface FiledY {
    String columnName();

    String type();

    int length();
}
```

运行结果

@net.yscxy.reflection.TableY(value=du_student)

du_student

db_name

carchar

3