链滴

Java 注解和反射

作者：yjsf

原文链接：https://ld246.com/article/1643381489475

来源网站：链滴

Java注解与反射

什么是注解（Annotation）

注解就是在一个代码元素（类、方法、变量）上用@标识的一种方式，像注释一样，注释是给人看的但是注解是给Java编译器看的，注解会在编译阶段，被编译器解析并还原为相应的操作

在Java5.0引入

注解主要是要配合反射使用，一般先通过filed.isAnnotationPresent(MyAnnotationj.class)判断代码素是否含有该注解，然后通过field.getAnnotation(MyAnnotationj.class)来拿到注解的内容，目的为了给容器或工厂提供注入的元数据。

内置注解

Java有一些内置的注解，常用的有：

@Override

最常见的一种注解，表示打算重写超类中的方法，修饰在方法上

@Deprecated

表示该元素被废弃，不应该被使用

@SuperWarning

抑制警告信息，必须要给予参数才可以起效，@SuperWarning（"all"）,表示抑制所有警告

元注解（Meta-annotation）

在外面自定义注解时，需要要使用Java提供的四种元注解，被用来提供对其他注解类型声明

- @Target

用于描述注解的使用范围，就是自己定义的注解可以使用在什么地方

范围：

1）PACKAGE 可以在package包上使用

2）TYPE 在类、接口、枚举、Annotation上使用

3）CONSTRUCTOR 构造器上

4）FIELD 在域上

5）METHOD：用于方法

6）LOCAL_VARIABLE 局部变量上

7）PARAMETER 用于参数上

- @Retention

用于描述注解的生命周期：SOURCE 在源文件中有效；CLASS 在class文件中有效

RUNTIME 运行时有效，一般定义为运行时有效，且可以被反射机制读取

- @Document 表示注解生成在javadoc中
- @Inherited 表示子类可以继承父类的注解

自定义注解

自定义注解主要使用以上四种元注解 对自定义注解进行注解，自定义注解格式为：

```
@Target({ElementType.ANNOTATION_TYPE.METHOD,ElementType.ANNOTATION_TYPE.FIELD
) //可作用在方法和域上
@Retention(RetentionPolicy.RUNTIME) //运行时有效
@Documented //添加到JavaDoc
@Inherited //可继承
public @interface MyAnnotation{
  String name() default "";
  int age() default 0;
  String[] things()default {""};
}
```

使用@interface声明为一个接口

其中的每一个方法对应了一个注解的参数, 可以有默认值

反射Refection

静态语言和动态语言

动态语言：可以在运行时改变其结构的语言，也就是程序在运行时，可以根据某些条件，改变自身的态；如：C#，JavaScript，PHP，Python

静态语言：运行时结构不可以改变的语言就是静态语言；如Java，C、C++等

Java可以称为准动态语言，依靠着反射机制，Java实现了一定的动态特征，使编程时更灵活

反射API java.lang.Reflection

Java允许在运行时，通过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方法，

主要是操作一个类的class对象操作，通过这个class对象我们可以看到一个类的所有结构信息，就像子一样，所以叫反射

反射可以实现的功能：

1）可以判断一个对象所属的类

2）可以构造任意一个类的对象（哪怕是单例模式，也可以通过反射额外构造出对象）

3）可以在运行时，获取一个类的所有方法，域（包括私有的，但需要设置可访问性）

4）在运行时获取泛型、注解等信息（注解的主要作用就是通过反射实现的）

5）动态代理等

反射会带来更高的性能损耗，是new方式的几十倍时间

获取class对象：

三种方式：通过对象获取；通过类名获取；通过Class类的静态方法forName获取（最常用）

```
String s="";
        Class c=s.getClass();//通过对象获取class
        c=String.class;//通过类名获取class
        c=Class.forName("java.lang.String");//通过完整类名，使用Class.forName()获取class
```

通过class.getSuperClass()获取类的父类class；

一个类只有一个class对象，通过每种方式获取的对象都是同一个class；

getclass方法，class属性定义在Object类中的，所有类都具有这些属性；

具有class对象的程序元素：所有类、接口、数组、枚举、注解、基本数据类型、**void**

获取运行时类的完整结构：

可以获取的结果有：Field、Method、Constructor、Superclass、Interface、Annotation

- 获取类的Field

```
//获取类的属性：
Field field;
Field[] fields=c.getFields();//获取类的全部公开属性(包括静态的)，
//field=c.getField("hash");//通过属性名获取类的公开属性，String的hash属性为私有的，此方法获不到
field=c.getDeclaredField("hash");//获取类的私有属性，但需要设置属性的可访问性
field.setAccessible(true);//设置元素的可访问性，默认为false不可访问
```

- 获取类的Method

```
//获取类的方法：
Method[] methods=c.getMethods();//获取所有公开方法
Method[] privateMethods=c.getDeclaredMethods();//获取所有方法，公开和私有的
Method method=c.getMethod("indexOf", String.class);//获取公开方法,第二个参数为可变参数，方法的参数的
// class类，空参传null即可，也表明了java通过方法名和参数唯一确定了一个方法
Method method1=c.getDeclaredMethod("checkBounds", byte[].class, int.class, int.class);//获私有方法
method1.setAccessible(true);//设置可访问性
```

- 获取类的Constuctor

```
//获取类的构造器
Constructor[] constructors=c.getConstructors();
Constructor[] constructors1=c.getDeclaredConstructors();//获取类的所有构造方法，公开和私有
```

```
，可破解单例模式 记得设置为可访问
Constructor constructor=c.getConstructor(String.class);
//构造对象 即一个String 并自定义他的hashcode
```

反射创建对象的完整代码演示，以String为例

```
String s="";
Class c=s.getClass();//通过对象获取class
c=String.class;//通过类名获取class
c=Class.forName("java.lang.String");//通过完整类名，使用Class.forName()获取class

//获取完整类名
c.getName();

//获取类的属性：
Field field;
Field[] fields=c.getFields();//获取类的全部公开属性(包括静态的)，
//field=c.getField("hash");//通过属性名获取类的公开属性，String的hash属性为私有的，此方法获不到
field=c.getDeclaredField("hash");//获取类的私有属性，但需要设置属性的可访问性
field.setAccessible(true);//设置元素的可访问性，默认为false不可访问


//获取类的方法：
Method[] methods=c.getMethods();//获取所有公开方法
Method[] privateMethods=c.getDeclaredMethods();//获取所有方法，公开和私有的
Method method=c.getMethod("indexOf", String.class);//获取公开方法,第二个参数为可变参数，方法的参数的class类，空参传null即可
// 也表明了java通过方法名和参数唯一确定了一个方法
Method method1=c.getDeclaredMethod("checkBounds", byte[].class, int.class, int.class);//获私有方法
method1.setAccessible(true);//设置可访问性

//获取类的构造器
Constructor[] constructors=c.getConstructors();
Constructor[] constructors1=c.getDeclaredConstructors();//获取类的所有构造方法，公开和私有，可破解单例模式 记得设置为可访问
Constructor constructor=c.getConstructor(String.class);
//构造对象 即一个String 并自定义他的hashcode
Object o=constructor.newInstance("hello");
field.set(o,123456);
System.out.println( o+"    "+method.invoke(o,"o"));//通过invoke执行方法，第二个可变参数为一数组
```

反射的性能

使用反射的执行速度大约是直接调用的6x10^3倍，而使用setAccessible(true)的方式可以加快反射度

反射操作泛型

java采用泛型擦除的方式引入泛型，泛型只是类似一种语法糖，在编译阶段，就会被编译器编译为正的类型，只是确保了代码的数据类型的安全性并省去了强制类型转换的问题；编译完成后，所有与泛

相关的东西全部会被擦除。

java引入了ParameterizedType,GenericArrayType, TypeVariable, WildcardType等几种类型来表不能被归一到Class类中的类型，但是又和原类型齐名的类型

ParameterizedType：表示一种参数化类型

GenericArrayType：表示一种元素类型是参数化类型或者类型变量类型的数组类型

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

通过反射操作注解

- ORM：

ORM（Object relationship Mapping）：对象关系映射

即：将对象与表结构想对应

- **类**对应**Mql中的表结构**
- **类中的属性**对应**表中的字段**
- **类的一个实例对象**对应**表的一条记录**

完成一个ORM框架

思路很简单：通过自定义一个注解，在自定义Entity上进行注解，创建实例，然后将该对象交付给我的注解处理程序 ORM 进行处理（在框架中如Spring等，是通过一个全局的扫描或类加载器来发现注并将类交付给对应的处理程序），注解处理程序先判断是否为自己要处理的注解，然后通过反射拿到名，域等信息，再拿到注解对象Annotation，获取到预设的表名，最后通过JDBC进行表创建，数据入等操作

对于域Filed或方法Method上的注解，可能是通过在获取到类时，获取到Methods对象，再获取到对的注解对象，其中的具体实现我现在也不太懂

下面给出自己实现的简单实例

我们自己定义的注解

```
@Target({ElementType.TYPE,ElementType.METHOD,ElementType.FIELD})//可注解在filed，method，Type上
@Retention(RetentionPolicy.RUNTIME)//运行时有效，不要写成Source源码级别有效
@interface MyORMAnnotation{
    String tableName() ;//为Value时，不需要指明，否则需要指明tableName="value"
}
```

实体类Entity

```
@MyORMAnnotation(tableName = "UserTestAnnotation")
class User{
    String username;
    String age;
```

```
    String money;
}
```

注解处理程序

```
public class MyORM {
    MyJDBCDriver myJDBCDriver =new MyJDBCDriver();
    public boolean myORMTranslate(Object obj) throws SQLException, IllegalAccessException {
        Class clazz=obj.getClass();
        boolean isMyORMAnnotation =clazz.isAnnotationPresent(MyORMAnnotation.class);//
断指定注解是否存在于该class上
        Annotation annotation=clazz.getAnnotation(MyORMAnnotation.class);//获取指定注解
        Annotation[] annotations=clazz.getAnnotations();//获取全部注解
        Annotation[] annotations1=clazz.getDeclaredAnnotations();//获取私有注解
        if (!isMyORMAnnotation){
            return false;
        }
        Field[] fields = clazz.getDeclaredFields();//获取所有的属性
        String[] types=new String[fields.length];
        String[] tablefileds=new String[fields.length];//所有的属性名
        String[] tablefiledsValue=new String[fields.length];//该对象的属性名所对应的值
        for (int i = 0; i < fields.length; i++) {
            fields[i].setAccessible(true);
            tablefileds[i]=fields[i].getName();
            tablefiledsValue[i]= (String) fields[i].get((User)obj);
            types[i]=fields[i].getType().getName();
        }
        MyORMAnnotation myORMAnnotation=(MyORMAnnotation)annotation;
        String tableName=myORMAnnotation.tableName();
        if (!myJDBCDriver.isTableExit(tableName)){
            myJDBCDriver.creatTable(tableName,types,tablefileds);
        }
        return myJDBCDriver.insertDate(tableName,tablefileds,tablefiledsValue);
    }
}
```

JDBCDriver的实现

```
public class MyJDBCDriver {
    Connection connection;
    public boolean connect(){
        Connection con;
        //jdbc驱动
        String driver="com.mysql.cj.jdbc.Driver";
        //这里我的数据库是cxxt
        String url="jdbc:mysql://119.23.62.68:3306/mytest?&useSSL=false&serverTimezone=UT
";
        String user="root";
        String password="root";
        try {
            //注册JDBC驱动程序
            Class.forName(driver);
            //建立连接
```

```
            con = DriverManager.getConnection(url, user, password);
            if (!con.isClosed()) {
                System.out.println("数据库连接成功");
            }
            connection=con;
        } catch (ClassNotFoundException e) {
            System.out.println("数据库驱动没有安装");

        } catch (SQLException e) {
            e.printStackTrace();
            System.out.println("数据库连接失败");
        }

        return true;
    }
    public boolean isTableExit(String tableName) throws SQLException {
        Statement statement= connection.createStatement();
        String SQL="SELECT table_name FROM information_schema.TABLES WHERE table_name
='"+tableName+"';";
        System.out.println(SQL);
        ResultSet resultSet=statement.executeQuery(SQL);
        while (resultSet.next()){
            return true;
        }
        return false;
    }

    public boolean creatTable(String tableName, String[] types, String[] fields) throws SQLExcep
ion {
        String cmd="";
        for (int i = 0; i < fields.length; i++) {
            cmd=cmd+fields[i]+"  VARCHAR(20) ,";
        }
        cmd=cmd.substring(0,cmd.length()-1);
        String SQL="create TABLE "+tableName +
                " ( ID int  not null," +   cmd+ " );";
        Statement statement= connection.createStatement();
        System.out.println(SQL);
        return statement.execute(SQL);
    }

    //INSERT INTO table_name (列1, 列2,...) VALUES (值1, 值2,....)
    public boolean insertDate(String tableName,String[] keys,String[] values) throws SQLExcept
on {
        String cmd=" (";//考虑使用StringBuilder更高效
        String ID=String.valueOf(System.currentTimeMillis()).substring(5);
        cmd=cmd+"ID ,";
        for (int i = 0; i < keys.length; i++) {
            cmd=cmd+keys[i]+",";
        }
        cmd=cmd.substring(0,cmd.length()-1)+") VALUES (";
        cmd=cmd+ID+",";
        for (int i = 0; i <values.length ; i++) {
            cmd=cmd+"'"+values[i]+"',";
```

```
        }
        cmd=cmd.substring(0,cmd.length()-1);
        String SQL="INSERT into "+tableName +    cmd+ " );";
        Statement statement= connection.createStatement();
        System.out.println(SQL);
        return statement.execute(SQL);
    }
}
```

测试代码

```
    public static void main(String[] args) throws SQLException, IllegalAccessException {
        User user=new User();
        user.age="10";
        user.username="小红";
        user.money="$12.5";
        MyORM myORM=new MyORM();
        myORM.myJDBCDriver.connect();
        myORM.myORMTranslate(user);
    }
```

测试结果是能进行插入，不附截图了