Java反射——Type接口详解

目录

Type的简介

java.lang.reflect.Type接口及其相关接口用于描述java中用到的所有类型，是Java的反射中很重要的组成部分。

在API文档中，Type接口的说明如下：

> Type 是 Java 编程语言中所有类型的公共高级接口。它们包括原始类型、参数化类型、数组类型、类型变量和基本类型。
>
> 从JDK1.5开始使用。

API中提到的Type的组成部分说明如下：

> - 原始类型：一般意义上的java类，由class类实现
> - 参数化类型：ParameterizedType接口的实现类
> - 数组类型：GenericArrayType接口的实现类
> - 类型变量：TypeVariable接口的实现类
> - 基本类型：int，float等java基本类型，其实也是class

不知道为什么在文档中介绍Type接口的组成时，没有包含WildcardType接口。

Type的获得

有很多场景下我们可以获得Type，比如：

1，当我们拿到一个Class，用Class. getGenericInterfaces()方法得到Type[]，也就是这个类实现接口的Type类型列表。

2，当我们拿到一个Class，用Class.getDeclaredFields()方法得到Field[]，也就是类的属性列表，然后用Field. getGenericType()方法得到这个属性的Type类型。

3，当我们拿到一个Method，用Method. getGenericParameterTypes()方法获得Type[]，也就是方法的参数类型列表。

Type的分类

Type接口包含了一个实现类(Class)和四个实现接口(TypeVariable, ParameterizedType, GenericArrayType, WildcardType)，这四个接口都有自己的实现类，但这些实现类开发都不能直接使用，只能用接口。

在不同的场景下，java会使用上面五种实现类的其中一种，来解释要描述的类型。

下面详细解释一下java是怎么在这五种实现类中选择的。

以方法的参数为例，写一段示例代码，重点关注其中的test方法：

```
public class TestReflect {

    public static void test(TestReflect p0,
                List<TestReflect> p1,
                Map<String,TestReflect> p2,
                List<String>[] p3,
                Map<String,TestReflect>[] p4,
                List<? extends TestReflect> p5,
                Map<? extends TestReflect,? super TestReflect> p6
                //T p7
                ){

    }

        public static void main(String[] args) {

                Method[] methods=TestReflect.class.getMethods();

                for(int i=0;i<methods.length;i++){
                        Method oneMethod=methods[i];

                        if(oneMethod.getName().equals("test")){
                                Type[] types=oneMethod.getGenericParameterTypes();

                                //第一个参数，TestReflect p0
                                Class type0=(Class)types[0];
                                System.out.println("type0:"+type0.getName());

                                //第二个参数，List<TestReflect> p1
                                Type type1=types[1];
                                Type[] parameterizedType1=((ParameterizedType)type1).getActualTypeArguments();
                                Class parameterizedType1_0=(Class)parameterizedType1[0];
```

```
33                  System.out.println("parameterizedType1_0:"+parameterizedType1_0.getName());34
35                                          //第三个参数，Map<String,TestReflect> p2
36                  Type type2=types[2];
37                  Type[] parameterizedType2=((ParameterizedType)type2).getActualTypeArguments();
38                  Class parameterizedType2_0=(Class)parameterizedType2[0];
39                  System.out.println("parameterizedType2_0:"+parameterizedType2_0.getName());
40                  Class parameterizedType2_1=(Class)parameterizedType2[1];
41                  System.out.println("parameterizedType2_1:"+parameterizedType2_1.getName());
42
43
44                  //第四个参数，List<String>[] p3
45                  Type type3=types[3];
46                  Type genericArrayType3=((GenericArrayType)type3).getGenericComponentType();
47                  ParameterizedType parameterizedType3=(ParameterizedType)genericArrayType3;
48                  Type[] parameterizedType3Arr=parameterizedType3.getActualTypeArguments();
49                  Class class3=(Class)parameterizedType3Arr[0];
50                  System.out.println("class3:"+class3.getName());
51
52                  //第五个参数，Map<String,TestReflect>[] p4
53                  Type type4=types[4];
54                  Type genericArrayType4=((GenericArrayType)type4).getGenericComponentType();
55                  ParameterizedType parameterizedType4=(ParameterizedType)genericArrayType4;
56                  Type[] parameterizedType4Arr=parameterizedType4.getActualTypeArguments();
57                  Class class4_0=(Class)parameterizedType4Arr[0];
58                  System.out.println("class4_0:"+class4_0.getName());
59                  Class class4_1=(Class)parameterizedType4Arr[1];
60                  System.out.println("class4_1:"+class4_1.getName());
61
62
63                  //第六个参数，List<? extends TestReflect> p5
64                  Type type5=types[5];
65                  Type[] parameterizedType5=((ParameterizedType)type5).getActualTypeArguments();
66                  Type[] parameterizedType5_0_upper=((WildcardType)parameterizedType5[0]).getUpperBounds();
67                  Type[] parameterizedType5_0_lower=((WildcardType)parameterizedType5[0]).getLowerBounds();
68
69                  //第七个参数，Map<? extends TestReflect,? super TestReflect> p6
70                  Type type6=types[6];
```

```
                                Type[] parameterizedType6=((ParameterizedType)type6).getActualTypeArguments();
                            Type[] parameterizedType6_0_upper=((WildcardType)parameterizedType6[0]).getUpperBounds();
                            Type[] parameterizedType6_0_lower=((WildcardType)parameterizedType6[0]).getLowerBounds();
                            Type[] parameterizedType6_1_upper=((WildcardType)parameterizedType6[1]).getUpperBounds();
                            Type[] parameterizedType6_1_lower=((WildcardType)parameterizedType6[1]).getLowerBounds();
                                                }


                }

        }

}
```

我们需要关注的就是在类中定义的test方法，这个方法的7个参数基本上涵盖了Type能用到的所有类型。

所以在main方法中我们用反射得到了这个test方法，然后用method.getGenericParameterTypes()方法得到了test方法的所有参数类型，这是一个Type数组，数组中的每一个元素就是每个参数的类型，java为每一个Type选择了一个Type的实现类。

以此我们可以看到java是怎么在5种实现类中选择的。

1，Class

当需要描述的类型是：

- 普通的java类（比如String，Integer，Method等等），
- 数组，
- 自定义类（比如我们自己定义的TestReflect类），
- 8种java基本类型（比如int，float等）
- 可能还有其他的类

那么java会选择Class来作为这个Type的实现类，我们甚至可以直接把这个Type强行转换类型为Class。

这些类基本都有一个特点：**基本和泛型无关**，其他4种Type的类型，基本都是泛型的各种形态。

所以第一个参数的测试代码：

```
//第一个参数，TestReflect p0
Class type0=(Class)types[0];
System.out.println(type0.getName());
```

输出的结果是：

> type0:com.webx.TestReflect

可见第一个参数Type的实现类就是Class。

2，ParameterizedType

当需要描述的类是**泛型类**时，比如List,Map等，不论代码里写没写具体的泛型，java会选择ParameterizedType接口做为Type的实现。

真正的实现类是sun.reflect.generics.reflectiveObjects.ParameterizedTypeImpl。

ParameterizedType接口有getActualTypeArguments()方法，用于得到泛型的Type类型数组。

第二个参数的测试代码：

```
//第二个参数，List< TestReflect > p1
Type type1=types[1];
Type[] parameterizedType1=((ParameterizedType)type1).getActualTypeArguments();
Class parameterizedType1_0=(Class)parameterizedType1[0];
```

```
System.out.println(parameterizedType1_0.getName());
```

type1是List< TestReflect >，List就属于泛型类，所以java选择ParameterizedType作为type1的实现，type1可以直接转换类型为ParameterizedType。

List的泛型中只能写一个类型，所以parameterizedType1数组长度只能是1，本例中泛型是TestReflect，是一个普通类，他的Type被java用Class来实现，也就是变量parameterizedType1_0，所以代码最后输出：

> parameterizedType1_0:com.webx.TestReflect

第三个参数的测试代码：

```
//第三个参数，Map<String,TestReflect> p2
Type type2=types[2];
Type[] parameterizedType2=((ParameterizedType)type2).getActualTypeArguments();
Class parameterizedType2_0=(Class)parameterizedType2[0];
System.out.println("parameterizedType2_0:"+parameterizedType2_0.getName());
Class parameterizedType2_1=(Class)parameterizedType2[1];
System.out.println("parameterizedType2_1:"+parameterizedType2_1.getName());
```

type2是Map<String,TestReflect>，Map属于泛型类，同样java选择ParameterizedType作为type2的实现，type2可以直接转换类型为ParameterizedType。

使用getActualTypeArguments()得到的泛型类型数组parameterizedType2有两个元素，因为Map在泛型中可以写两个类型，本例中Map的泛型类型分别是String类和TestReflect，他们的Type都会被java用Class来实现，所以最后输出的是：

> parameterizedType2_0:java.lang.String
>
> parameterizedType2_1:com.webx.TestReflect

3，GenericArrayType

当需要描述的类型是**泛型类的数组**时，比如比如List[],Map[]，type会用GenericArrayType接口作为Type的实现。

真正的实现类是sun.reflect.generics.reflectiveObjects. GenericArrayTypeImpl。

GenericArrayType接口有getGenericComponentType()方法，得到数组的组件类型的Type对象。

第四个参数的测试代码：

```
//第四个参数，List<String>[] p3
Type type3=types[3];
Type genericArrayType3=((GenericArrayType)type3).getGenericComponentType();
ParameterizedType parameterizedType3=(ParameterizedType)genericArrayType3;
Type[] parameterizedType3Arr=parameterizedType3.getActualTypeArguments();
Class class3=(Class)parameterizedType3Arr[0];
System.out.println("class3:"+class3.getName());
```

type3是List<String>[]，所以java选择GenericArrayType作为type3的实现，type3可以直接转换类型为GenericArrayType。

调用getGenericComponentType()方法，得到数组的组件类型的Type对象，也就是本例中的变量genericArrayType3，他代表的是List<String>类。

List<String>是泛型类，所以变量genericArrayType3的Type用ParameterizedType来实现，转换类型之后也就是变量parameterizedType3。

parameterizedType3.getActualTypeArguments()得到的是List<String>类型的泛型类数组，也就是数组parameterizedType3Arr。

数组parameterizedType3Arr只有一个元素，类型是String，这个Type由Class实现，就是变量class3，最后输出的是：

```
class3:java.lang.String
```

第五个参数的测试代码：

```
//第五个参数，Map<String,TestReflect>[] p4
Type type4=types[4];
Type genericArrayType4=((GenericArrayType)type4).getGenericComponentType();
ParameterizedType parameterizedType4=(ParameterizedType)genericArrayType4;
Type[] parameterizedType4Arr=parameterizedType4.getActualTypeArguments();
Class class4_0=(Class)parameterizedType4Arr[0];
System.out.println("class4_0:"+class4_0.getName());
Class class4_1=(Class)parameterizedType4Arr[1];
System.out.println("class4_1:"+class4_1.getName());
```

type4是Map<String,TestReflect>[]，所以java选择GenericArrayType作为type4的实现，type4可以直接转换类型为GenericArrayType。

调用getGenericComponentType()方法，得到数组的组件类型的Type对象，也就是本例中的变量genericArrayType4，他代表的是Map<String,TestReflect>类型。

Map<String,TestReflect>是泛型类，所以变量genericArrayType4的Type用ParameterizedType来实现，转换类型之后也就是变量parameterizedType4。

parameterizedType4.getActualTypeArguments()得到的是Map<String,TestReflect>类型的泛型类数组，也就是变量parameterizedType4Arr。

变量parameterizedType4Arr有两个元素，类型是String和TestReflect，这两个Type都由Class实现，就是变量class4_0和变量class4_1，最后输出的是：

> class4_0:java.lang.String
>
> class4_1:com.webx.TestReflect

4，WildcardType

当需要描述的类型是泛型类，而且泛型类中的泛型被定义为**(? extends xxx)或者(? super xxx)**这种类型，比如List<? extends TestReflect>，这个类型首先将由ParameterizedType实现，当调用ParameterizedType的getActualTypeArguments()方法后得到的Type就由WildcardType实现。

真正的实现类是sun.reflect.generics.reflectiveObjects. WildcardTypeImpl。

WildcardType接口有getUpperBounds()方法，得到的是类型的上边界的Type数组，实际上就是类型的直接父类，也就是extends后面的类型。显然在当前java的设定中，这个数组只可能有一个元素，因为java现在只能extends一个类。如果实在没写extends，那他的直接父类就是Object。

WildcardType接口有getLowerBounds()方法，得到的是类型的下边界的Type数组，有super关键字时可能会用到，经测试不会得到类型的子类，而是只得到super关键字后面的类型，如果没写super关键字，则返回空数组。

第六个参数的测试代码：

```
//第六个参数，List<? extends TestReflect> p5
Type type5=types[5];
Type[] parameterizedType5=((ParameterizedType)type5).getActualTypeArguments();
Type[] parameterizedType5_0_upper=((WildcardType)parameterizedType5[0]).getUpperBounds();
Type[] parameterizedType5_0_lower=((WildcardType)parameterizedType5[0]).getLowerBounds();
```

type5代表List<? extends TestReflect>，用ParameterizedType实现。

调用getActualTypeArguments()方法后，得到只有一个元素的Type数组，这个元素就代表(? extends TestReflect)

把这个Type元素转成WildcardType后，可以调用getUpperBounds()和getLowerBounds()方法得到上边界和下边界，在本例中的上边界就是变量parameterizedType5_0_upper，只有一个元素，该元素代表TestReflect类型，下边界是变量parameterizedType5_0_lower，是个空数组。

第七个参数的测试代码：

```
//第七个参数，Map<? extends TestReflect,? super TestReflect> p6
Type type6=types[6];
Type[] parameterizedType6=((ParameterizedType)type6).getActualTypeArguments();
Type[] parameterizedType6_0_upper=((WildcardType)parameterizedType6[0]).getUpperBounds();
Type[] parameterizedType6_0_lower=((WildcardType)parameterizedType6[0]).getLowerBounds();
Type[] parameterizedType6_1_upper=((WildcardType)parameterizedType6[1]).getUpperBounds();
Type[] parameterizedType6_1_lower=((WildcardType)parameterizedType6[1]).getLowerBounds();
```

type6代表Map<? extends TestReflect,? super TestReflect>，用ParameterizedType实现。

调用getActualTypeArguments()方法后，得到有两个元素的Type数组，两个元素分别代表(? extends TestReflect)和(? super TestReflect)

把这两个Type元素转成WildcardType后，可以调用getUpperBounds()和getLowerBounds()方法得到上边界和下边界。

在本例中第一个WildcardType的上边界就是变量parameterizedType6_0_upper，只有一个元素，该元素代表TestReflect类型，下边界是变量parameterizedType6_0_lower，是个空数组。

在本例中第二个WildcardType的上边界就是变量parameterizedType6_1_upper，只有一个元素，该元素代表Object类型，下边界是变量parameterizedType6_1_lower，只有一个元素，该元素代表TestReflect类型。

5，TypeVariable

Type的最后一种实现形式是TypeVariable接口，这种实现形式是在**泛型类**中使用的。

比如我们定义一个泛型类TestReflect<T>，并在类中定义方法oneMethod(T para)，那么当调用method.getGenericParameterTypes()方法得到的Type数组，数组的元素就是由TypeVariable接口实现的。

真正的实现类是sun.reflect.generics.reflectiveObjects. TypeVariableImpl。

以上就是关于Type接口的详细介绍。