Java类型中ParameterizedType，GenericArrayType，TypeVariabl，WildcardType详解

(1). 和反射+泛型有关的接口类型

* java.lang.reflect.Type：java语言中所有类型的公共父接口
* java.lang.reflect.ParameterizedType
* java.lang.reflect.GenericArrayType
* java.lang.reflect.WildcardType

1. Type直接子接口

ParameterizedType，GenericArrayType，TypeVariable和WildcardType四种类型的接口

* ParameterizedType: 表示一种参数化的类型，比如Collection
* 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，比如?, ? extends Number, ? super Integer【wildcard是一个单词：就是“通配符”】

2. Type直接实现子类 :Class类

**3. java.lang.reflect.Type接口**

Type所有类型指代的有：原始类型 (raw types)【对应Class】，参数化类型 (parameterizedtypes)【对应ParameterizedType】， 数组类型 (array types)【对应GenericArrayType】，类型变量 (type variables)【对应TypeVariable】，基本数据类型(primitivetypes)【仍然对应Class】

**4. java.lang.reflect.ParameterizedType接口**

ParameterizedType接口类型的含义

表示参数化类型。比如：Map这种参数化类型

获取参数化类型<>中的实际类型

源码声明：Type[] getActualTypeArguments();

【注意】无论<>中有几层<>嵌套，这个方法仅仅脱去最外层的<>之后剩下的内容就作为这个方法的返回值。

public static  E methodIV(   ArrayList<arraylist> al1,   ArrayList al2,   ArrayList al3,   ArrayListextends Number> al4,   ArrayList al5){}

那么他的每一参数总体上看都是参数化类型的。

{1}. 对于ArrayList<arraylist>，通过getActualTypeArguments()返回之后，脱去最外层的<>之后，剩余的类型是ArrayList。因此对这个参数的返回类型是ParameterizedType。

{2}. 对于ArrayList，通过getActualTypeArguments()返回之后，脱去最外层的<>之后，剩余的类型是E。因此对这个参数的返回类型是TypeVariable。

{3}. 对于ArrayList，通过getActualTypeArguments()返回之后，脱去最外层的<>之后，剩余的类型是String。因此对这个参数的返回类型是Class。

{4}. 对于ArrayListextends Number>，通过getActualTypeArguments()返回之后，脱去最外层的<>之后，剩余的类型是? ExtendsNumber。因此对这个参数的返回类型是WildcardType。

{5}. 对于ArrayList，通过getActualTypeArguments()返回之后，脱去最外层的<>之后，剩余的类型是E[]。因此对这个参数的返回类型是GenericArrayType。

所以，可能获得各种各样类型的实际参数，所以为了统一，采用直接父类数组Type[]进行接收。

**4. java.lang.reflect. GenericArrayType接口**

GenericArrayType接口类型的含义

表示泛型数组类型。比如：void method(ArrayList[] al){…}

【注意】<>不能出现在数组的初始化中，即new数组之后不能出现<>，否则javac无法通过。但是作为引用变量或者方法的某个参数是完全可以的。

获取泛型数组中元素的类型

源码声明：Type getGenericComponentType();

【注意】无论从左向右有几个[]并列，这个方法仅仅脱去最右边的[]之后剩下的内容就作为这个方法的返回值。

为什么返回值类型是Type？

public static  E methodV(

String[] p1,

E[] p2,

ArrayList[] p3,

E[][] p4){}

{1}. 对于String[]，通过getComponentType()返回之后，脱去最右边的[]之后，剩余的类型是String。因此对这个参数的返回类型是Class

{2}. 对于E[]，通过getComponentType()返回之后，脱去最右边的[]之后，剩余的类型是E。因此对这个参数的返回类型是TypeVariable

{3}. 对于ArrayList[]，通过getComponentType()返回之后，脱去最右边的[]之后，剩余的类型是ArrayList。因此对这个参数的返回类型是ParameterizedType

{4}. 对于E[][]，通过getComponentType()返回之后，脱去最右边的[]之后，剩余的类型是E[]。因此对这个参数的返回类型是GenericArrayType

**5. java.lang.reflect. GenericArrayType接口**

TypeVariable接口类型的含义

表示类型参数或者又叫做类型变量。比如：void method(E e){}中的E就是类型变量

获取类型变量的泛型限定的上边界的类型

源码声明：Type[] getActualTypeArguments();

【注意】这里面仅仅是上边界。原因就是类型变量在定义的时候只能使用extends进行(多)边界限定。不能使用super，否则编译无法通过。同时extends给出的都是类型变量的上边界。

为什么是返回类型是数组？因为类型变量可以通过&进行多个上边界限定，因此上边界有多个，因此返回值类型是数组类型[ ]。

例如下面的方法：

public static <e extends Map& Cloneable&Serializable> E methodVI(E e){…}</e

E的第一个上边界是Map,是ParameterizedType类型

E的第二个上边界是Cloneable，是Class类型

因此，为统一，返回值的数组的元素类型就是Type

**6. java.lang.reflect.WildcardType接口**

WildcardType接口类型的含义

表示通配符类型的表达式。

比如 void printColl(ArrayListal); 中的 ? extends Number

【注意】根据上面API的注释提示：现阶段通配符表达式仅仅接受一个上边界或者下边界，这个和定义类型变量时候可以指定多个上边界是不一样。但是API说了，为了保持扩展性，这里返回值类型写成了数组形式。实际上现在返回的数组的大小就是1

获取通配符表达式对象的泛型限定的上边界的类型

源码声明：Type[] getUpperBounds();

【注意】上面说了，现阶段返回的Type[ ]中的数组大小就是1个。写成Type[ ]是为了语言的升级而进行的扩展。

例如下面的方法：

{1}. public static voidprintColl(ArrayListextends ArrayList> al){}

通配符表达式是：? extendsArrayList，这样 extends后面是?的上边界，这个上边界是ParameterizedType类型。

{2}. public static  voidprintColl(ArrayListextends E> al){}

通配符表达式是：? extends E，这样 extends后面是?的上边界，这个上边界是TypeVariable类型

{3}.public static  voidprintColl(ArrayListextends E[]> al){}

通配符表达式是：? extends E[]，这样 extends后面是?的上边界，这个上边界是GenericArrayType类型

{4}.public static  voidprintColl(ArrayListextends Number> al){}

通配符表达式是：? extends Number，这样 extends后面是?的上边界，这个上边界是Class类型

最终统一成Type作为数组的元素类型。

**7. Type及其子接口的来历**

一. 泛型出现之前的类型

没有泛型的时候，只有所谓的原始类型。此时，所有的原始类型都通过字节码文件类Class类进行抽象。Class类的一个具体对象就代表一个指定的原始类型。

二. 泛型出现之后的类型

泛型出现之后，扩充了数据类型。从只有原始类型扩充了参数化类型、类型变量类型、泛型限定的的参数化类型 (含通配符+通配符限定表达式)、泛型数组类型。

三. 与泛型有关的类型不能和原始类型统一到Class的原因

[1]. 【产生泛型擦除的原因】

本来新产生的类型+原始类型都应该统一成各自的字节码文件类型对象。但是由于泛型不是最初Java中的成分。如果真的加入了泛型，涉及到JVM指令集的修改，这是非常致命的。

[2]. 【Java中如何引入泛型】

为了使用泛型的优势又不真正引入泛型，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的麻烦。但是，一旦编译完成，所有的和泛型有关的类型全部擦除。

[3]. 【Class不能表达与泛型有关的类型】

因此，与泛型有关的参数化类型、类型变量类型、泛型限定的的参数化类型 (含通配符+通配符限定表达式)、泛型数组类型这些类型全部被打回原形，在字节码文件中全部都是泛型被擦除后的原始类型，并不存在和自身类型一致的字节码文件。所以和泛型相关的新扩充进来的类型不能被统一到Class类中。

(4). 与泛型有关的类型在Java中的表示

为了通过反射操作这些类型以迎合实际开发的需要，Java就新增了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。

(5). Type的引入：统一与泛型有关的类型和原始类型Class

【引入Type的原因】

为了程序的扩展性，最终引入了Type接口作为Class，ParameterizedType，GenericArrayType，TypeVariable和WildcardType这几种类型的总的父接口。这样实现了Type类型参数接受以上五种子类的实参或者返回值类型就是Type类型的参数。

【Type接口中没有方法的原因】

从上面看到，Type的出现仅仅起到了通过多态来达到程序扩展性提高的作用，没有其他的作用。因此Type接口的源码中没有任何方法。

## 测试相关

val methods = Test::class.*java*.getMethods()  
  
for (i in methods.*indices*) {  
 val oneMethod = methods[i]  
  
 if (oneMethod.getName().equals(**"test"**)) {  
 val types = oneMethod.getGenericParameterTypes()  
  
 //第一个参数，TestReflect p0  
 val type0 = types[0] as Class<\*>  
 *println*(**"type0:"** + type0.*name*)  
  
 //第二个参数，List<TestReflect> p1  
 val type1 = types[1]  
 val parameterizedType1 = (type1 as ParameterizedType).*actualTypeArguments* val parameterizedType1\_0 = parameterizedType1[0] as Class<\*>  
 *println*(**"parameterizedType1\_0:"** + parameterizedType1\_0.*name*)  
  
 //第三个参数，Map<String,TestReflect> p2  
 val type2 = types[2]  
 val parameterizedType2 = (type2 as ParameterizedType).*actualTypeArguments* val parameterizedType2\_0 = parameterizedType2[0] as Class<\*>  
 *println*(**"parameterizedType2\_0:"** + parameterizedType2\_0.*name*)  
 val parameterizedType2\_1 = parameterizedType2[1] as Class<\*>  
 *println*(**"parameterizedType2\_1:"** + parameterizedType2\_1.*name*)  
  
  
 //第四个参数，List<String>[] p3  
 val type3 = types[3]  
 val genericArrayType3 = (type3 as GenericArrayType).getGenericComponentType()  
 val parameterizedType3 = genericArrayType3 as ParameterizedType  
 val parameterizedType3Arr = parameterizedType3.*actualTypeArguments* val class3 = parameterizedType3Arr[0] as Class<\*>  
 *println*(**"class3:"** + class3.*name*)  
  
 //第五个参数，Map<String,TestReflect>[] p4  
 val type4 = types[4]  
 val genericArrayType4 = (type4 as GenericArrayType).getGenericComponentType()  
 val parameterizedType4 = genericArrayType4 as ParameterizedType  
 val parameterizedType4Arr = parameterizedType4.*actualTypeArguments* val class4\_0 = parameterizedType4Arr[0] as Class<\*>  
 *println*(**"class4\_0:"** + class4\_0.*name*)  
 val class4\_1 = parameterizedType4Arr[1] as Class<\*>  
 *println*(**"class4\_1:"** + class4\_1.*name*)  
  
  
 //第六个参数，List<? extends TestReflect> p5  
 val type5 = types[5]  
 val parameterizedType5 = (type5 as ParameterizedType).*actualTypeArguments* val parameterizedType5\_0\_upper = (parameterizedType5[0] as WildcardType).getUpperBounds()  
 val parameterizedType5\_0\_lower = (parameterizedType5[0] as WildcardType).getLowerBounds()  
  
 //第七个参数，Map<? extends TestReflect,? super TestReflect> p6  
 val type6 = types[6]  
 val parameterizedType6 = (type6 as ParameterizedType).*actualTypeArguments* val parameterizedType6\_0\_upper = (parameterizedType6[0] as WildcardType).getUpperBounds()  
 val parameterizedType6\_0\_lower = (parameterizedType6[0] as WildcardType).getLowerBounds()  
 val parameterizedType6\_1\_upper = (parameterizedType6[1] as WildcardType).getUpperBounds()  
 val parameterizedType6\_1\_lower = (parameterizedType6[1] as WildcardType).getLowerBounds()  
  
 }  
  
}

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。

---------------------

作者：lkforce

来源：CSDN

原文：https://blog.csdn.net/lkforce/article/details/82466893

版权声明：本文为博主原创文章，转载请附上博文链接！