# lsy



## 提炼Java Reflection

博客分类：
- Java应用开发

JavaJVMSpringIDEASUN

反射是Java语言中很重要的一个组成部分，所以就此话题讨论的资源可谓数之不尽，日常开发也会经常使用到关于反射的Reflection API。Java5.0 Tiger出现以后，更对反射API有了新的扩展，尽管讨论的话题很多，不过我还是觉得不够全面，尤其是对泛型这一块，所以就我所知，再花力气总结一番

首先反射的入口是从Class开始的，所以如何获取Class就变得十分关键了。这里总结了几种方式：
1.通过${name}.class 语法。这里${name}可以是对象，也可以是原始数据类型，不过别忘了void.class和Void.class
2.通过${name}.TYPE 语法。这里${name}是八种原始数据的包装类和Void.TYPE
3.通过对象的getClass()方法。
4.通过Class对象的forName()方法
5.通过类Class的getSuperclass()获取父亲类Class
6.通过类Class的getEnclosingClass()获取外部类Class
7.通过类Class的getClasses()和getDeclaredClasses()获取内部类Class
下面是一张表用来说明getClasses()和getDeclaredClasses()两个方法，稍后还会用该表说明其他Reflection API

| **Member** | **Class API** | **Return type** | **Inherited members** | **Private members** |
|---|---|---|---|---|
| Class | getDeclaredClasses() | Array | N | Y |
| | getClasses() | Array | Y | N |
| Field | getDeclaredField() | Single | N | Y |
| | getField() | Single | Y | N |
| | getDeclaredFields() | Array | N | Y |
| | getFields() | Array | Y | N |
| Method | getDeclaredMethod() | Single | N | Y |
| | getMethod() | Single | Y | N |
| | getDeclaredMethods() | Array | N | Y |
| | getMethods() | Array | Y | N |
| Constructor | getDeclaredConstructor() | Single | N/A | Y |
| | getConstructor() | Single | N/A | N |
| | getDeclaredConstructors() | Array | N/A | Y |
| | getConstructors() | Array | N/A | N |

表一： 成员方法对照表
如表一所示，getClasses()拥有继承的特点，可以获取父亲级定义的内部类，而不能访问定义为private的内部类；而getDeclaredClasses()刚好相反，可以访问定义为private的内部类，却无法获取父亲级定义的内部类

成功获取了Class以后，那么就可以开始访问Field，Method和Constructor了，他们都继承自java.lang.reflect.Member。从上表已经很容易可以看出各个成员是否拥有继承特性，是否能够访问私有成员，返回类型的数量这些信息。这里需要注意一点，由于Constructor是无法被继承的，所以Constructor成员任何方法都没有继承的特性。另外Field和Method在赋值或者调用的之前需要留意是否在操作私有成员，如果是那么需要先修改可访问度，执行setAccessible(true)。还有一点就是Method成员的getDeclaredMethod()和getMethod()方法都需要两

个参数，一个是方法的名称，另外一个参数Class的数组，这里需要感谢Java5.0引入不定长参数的特点，使我们可以在某些情况下少传入一个参数，如：
假设Order类有下列方法

Java代码

```
1. public Long getId() { return id; }
```

5.0以前获取该方法的代码如下

Java代码

```
1. Method getId = Order.class.getMethod("getId", new Class[0]);
```

而5.0仅需要写

Java代码

```
1. Method getId = Order.class.getMethod("getId");
```

现在说说5.0 泛型出现之后，Java Reflection API的新特点。首先增加一个接口java.lang.reflect.Type，其下一共有4个接口继承了它，TypeVariable，ParameterizedType，GenericArrayType和WildcardType，下面逐个分析。
1.TypeVariable
我们知道泛型信息会在编译时被JVM编译时转换为定义的一个特定的类型，这减少了应用程序逐步向下检查类型的开支，避免了发生ClassCastException的危险。而TypeVariable就是用来反映在JVM编译该泛型前的信息。举个例子，假设BaseOrder类定义有如下一个方法

Java代码

```
1. public class BaseOrder<M extends Object & Serializable,N extends Comparable<N>> implements IBaseOrder {
2.     public M getManufactory(){
3.         return manufactory;
4.     }
5. }
```

这时候我们可以通过如下代码获取该泛型Type，并且经过测试该Type就是TypeVariable

Java代码

```
1. Field manufactoryField = BaseOrder.class.getDeclaredField("manufactory");
2. type = manufactoryField.getGenericType();
3. assertTrue("The type of field manufactory is an instance of TypeVariable", type instanceof TypeVariable);
4. TypeVariable tType = (TypeVariable)type;
5. assertEquals("The name of this TypeVariable is M", "M", tType.getName());
6. assertEquals("The TypeVariable bounds two type", 2, tType.getBounds().length);
7. assertEquals("One type of these bounds is Object", Object.class, tType.getBounds()[0]);
8. assertEquals("And annother si Serializable", Serializable.class, tType.getBounds()[1]);
```

通过getName()方法可以获取该泛型定义的名称，而更为重要的是getBounds()方法，可以判断该泛型的边界。

2.ParameterizedType
这个接口就比较出名了，在过去讨论最多的问题就是GenericDao<T>中，如何获取T.class的问题了。这里再翻出来过一遍，加入上述的类BaseOrder定义不变，新定义一个Order对象，代码如下：

Java代码

```
1. public class Order
2.         extends BaseOrder<Customer, Long> implements IOrder, Serializable {
3. }
```

那么如何通过Order获取到Customer呢？

Java代码

```
1. Type genericSuperclass = Order.class.getGenericSuperclass();
2. assertTrue("Order's supper class is a type of ParameterizedType.", genericSuperclass instanceof ParameterizedType);
3. ParameterizedType pType = (ParameterizedType)genericSuperclass;
4. assertEquals("Order's supper class is BaseOrder.", BaseOrder.class, pType.getRawType());
5. Type[] arguments = pType.getActualTypeArguments();
6. assertEquals("getActualTypeArguments() method return 2 arguments.", 2, arguments.length);
7. for (Type type : arguments) {
8.     Class clazz = (Class)type;
9.     if(!(clazz.equals(Customer.class)) && !(clazz.equals(Long.class))){
10.        assertTrue(false);
11.    }
12. }
```

可以看出通过Order类的getGenericSuperclass()方法将返回一个泛型，并且它就是ParameterizedType。这个接口的getRawType()方法和getActualTypeArguments()都非常重要，getRawType()方法返回的是承载该泛型信息的对象，而getActualTypeArguments()将会返回一个实际泛型对象的数组。这里先提及一下Class对象中getGenericSuperclass()和getSuperclass()两个方法的区别，后文还有详细说明。getGenericSuperclass()方法首先会判断是否有泛型信息，有那么返回泛型的Type，没有则返回Class，方法的返回类型都是Type，这是因为Tiger中Class也实现了Type接口。将父亲按照Type接口的形式返回，而getSuperclass()直接返回父亲的Class。

3.GenericArrayType
这个接口比较好理解。如果泛型参数是一个泛型的数组，那么泛型Type就是GenericArrayType，它的getGenericComponentType()将返回被JVM编译后实际的数组对象。这里假设上文中BaseOrder有一个方法如下：

Java代码

```
1. public String[] getPayments(String[] payments, List<Product> products){
2.     return payments;
3. }
```

可以看出该方法的参数中有泛型信息，测试一下：

Java代码

```
1. Method getPayments = BaseOrder.class.getMethod("getPayments", new Class[]{String[].class, List.class});
2. types = getPayments.getGenericParameterTypes();
3. assertTrue("The first parameter of this method is GenericArrayType.", types[0] instanceof GenericArrayType);
4. GenericArrayType gType = (GenericArrayType)types[0];
5. assertEquals("The GenericArrayType's component is String.", String.class, gType.getGenericComponentType());
```

发现这个getPayments()方法中的一个参数String[] payments是一个GenericArrayType，通过getGenericComponentType()方法返回的是String.class。这是怎么回事呢？这里我们回过头去看Class对象的getGenericSuperclass()方法和getSuperclass()方法，如果把它们说成是一对的话，那么这里的getGenericParameterTypes()和getParameterTypes()就是另外一对。也就是说getGenericParameterTypes()首先判断该方法的参数中是否有泛型信息，有那么返回泛型Type的数组，没有那么直接按照Class的数组返回；而getParameterTypes()就直接按照Class的数组返回。非常相似吧，其原因就是这些成对的方法都有一个共同点就是判断是否有泛型信息，可以查看Tiger的源代码：

Java代码

```
1. public Type[] getGenericParameterTypes() {
2. (getGenericSignature() != null)
3.  return getGenericInfo().getParameterTypes();
4. e
5.  return getParameterTypes();
6. }
```

而这类成对出现的方法还很多，如Method对象定义的getGenericReturnType()和getReturnType()，getGenericExceptionTypes()和getExceptionTypes()，Field对象定义的getGenericType()和getType()。

4.WildcardType
这个接口就是获取通配符泛型的信息了。这里假设上述的BaseOrder定义有一个属性

Java代码

```
1. private Comparable<? extends Customer> comparator;
```

现在就来获取泛型?的信息，测试代码如下：

Java代码

```
1. Field comparatorField = BaseOrder.class.getDeclaredField("comparator");
2. ParameterizedType pType = (ParameterizedType)comparatorField.getGenericType();
3. type = pType.getActualTypeArguments()[0];
4. assertTrue("The type of field comparator is an instance of ParameterizedType, and the actual argument is an instance of \
5. WildcardType wType = (WildcardType)type;
6. assertEquals("The upper bound of this WildcardType is Customer.", Customer.class, wType.getUpperBounds()[0]);
```

首先我们获取到comparator这个属性，通过它的getGenericType()方法我们拿到了一个Type，可以看出它是一个ParameterizedType，而ParameterizedType的actual argument就是我们需要的WildcardType，这个接口有两个主要的方法，getLowerBounds()获取该通配符泛型的下界信息，相反getUpperBounds()方法获取该通配符泛型的上界信息。

说完了这四个接口，我们再回过头来看看Method对象，它还定义有一个方法getTypeParameters()，这又是干什么的呢？我们知道泛型是不能出现在静态的成员，静态的方法，或者静态的初始化器的逻辑中的。如下列代码都是错误的：

Java代码

```
1. //error
2. private static T customer;
3.
4. //error
5. public static T getCustomer(){
6.     return customer;
7. }
8.
9. //error
10. static {
11.     customerHolder = new HashSet<T>();
12. }
```

不过静态的方法的参数却是可以被泛化的，如：

Java代码

```
1. public static <B extends BusinessType, S extends Serializable> Customer getSpcialCustomer(List<B> types, S serial){
2.     return new Customer();
3. }
```

这个方法我们就可以通过getTypeParameters()方法来获取它的参数化信息，代码如下：

Java代码

```
1. Method getSpcialCustomer = SalePolicy.class.getMethod("getSpcialCustomer", new Class[]
   {List.class,Serializable.class});
2. types = getSpcialCustomer.getTypeParameters();
3. assertEquals("The method declared two TypeVariable.", 2, types.length);
4. assertEquals("One of the TypeVariable is B.", "B", ((TypeVariable)types[0]).getName());
5. assertEquals("And another is S.", "S", ((TypeVariable)types[1]).getName());
```

最后，说一下Annotation，成员可以通过isAnnotationPresent(annotationClass)和getAnnotation(annotationClass)方法来判断是否被某个Annotation标注，不过需要注意的时该annotation自身必须被标注为@Retention(RetentionPolicy.RUNTIME)。

晕，一写就那么晚了，抓紧睡觉去。如果本文写的有错误的地方请指正，以免误人子弟。

PS：附件是一个较为全面的测试代码

- summary-reflection.rar (15.6 KB)
- 描述: 测试代码
- 下载次数: 109



第一个Oracle Java Procedure | 设置参数解决内存溢出

- 2008-07-28 03:31
- 浏览 2378
- 评论(1)
- 论坛回复 / 浏览 (1 / 2270)
- 分类:编程语言
- 相关推荐

评论

1 楼 lsy 2008-07-30
反射虽然功能强大，可以让应用程序不知情的情况下改变对象的行为和状态。不过也正因为如此，所以如果稍有疏忽可能会造成一些莫名其妙的问题。因此如果反射的目标应用于完全符合JavaBeanhttp://java.sun.com/docs/books/tutorial/javabeans/规范的对象时，那么JavaBean已经提供了一套完整的机制来处理。BeanUtils和Spring也都是这样做的，在应用时我们也不妨借鉴借鉴:idea:

发表评论

您还没有登录,请您登录后再发表评论

lsy

- 浏览: 60041 次
- 性别:
- 来自: 深圳
- 我现在离线

最近访客 更多访客>>


jybbh


haha1903


Angel_Night


berdy

文章分类

- 全部博客 (33)
- 默认类别 (0)

- Java应用开发 (13)
- 随笔 (0)
- 数据库技术 (7)
- 中间件技术 (6)
- SOA/ESB应用 (5)
- 云计算 (1)

社区版块

- 我的资讯 (0)
- 我的论坛 (139)
- 我的问答 (3)

存档分类

- 2013-04 (1)
- 2013-01 (13)
- 2012-07 (2)
- 更多存档...

最新评论

- 老竹枝： 为什么不直接写成下面这样？还省了一个子查询<pre na ...
  小记：使用Oracle rownum分页
- snoopy3384： 明了，易懂
  Oracle 行列转换
- lsy： 反射虽然功能强大，可以让应用程序不知情的情况下改变对象的行为和 ...
  提炼Java Reflection
- kujioon： 10:41:51,140 WARN [JmxKernelAbs ...
  EJB3.0学习之路 让第一个Stateless Session Bean跑起来
- laobai： 首先谢谢前面两位了，我也遇到了cachalot的问题，异常和他 ...
  EJB3.0学习之路 让第一个Stateless Session Bean跑起来

---