<https://www.cnblogs.com/tech-bird/p/3525336.html>

# **[java中的反射](https://www.cnblogs.com/tech-bird/p/3525336.html)**

主要介绍以下几方面内容

* 理解 Class 类
* 理解 Java 的类加载机制
* 学会使用 ClassLoader 进行类加载
* 理解反射的机制
* 掌握 Constructor、Method、Field 类的用法
* 理解并掌握动态代理

# ****1.理解Class类****

　　–对象照镜子后可以得到的信息：某个类的数据成员名、方法和构造器、某个类到底实现了哪些接口。对于每个类而言，JRE 都为其保留一个不变的 Class 类型的对象。一个 Class 对象包含了特定某个类的有关信息。

　　–Class 对象只能由系统建立对象

　　–一个类在 JVM 中只会有一个Class实例

　　–每个类的实例都会记得自己是由哪个 Class 实例所生成

      1：**Class是什么？**

      Class是一个类：

[IMG_256](javascript:void(0);)

public class ReflectionTest {

@Test

public void testClass() {

Class clazz = null;

}

}

//Class的定义public final

class Class<T> implements java.io.Serializable,

java.lang.reflect.GenericDeclaration,

java.lang.reflect.Type,

java.lang.reflect.AnnotatedElement {

.....

.....

.....

}  
**//小写class表示是一个类类型，大写Class表示这个类的名称**

[IMG_257](javascript:void(0);)

      2：**Class这个类封装了什么信息？**

**Class是一个类，封装了当前对象所对应的类的信息**  
　　 一个类中有属性，方法，构造器等，比如说有一个Person类，一个Order类，一个Book类，这些都是不同的类，现在需要一个类，用来描述类，这就是Class，它应该有类名，属性，方法，构造器等。Class是用来描述类的类

　　Class类是一个对象照镜子的结果，对象可以看到自己有哪些属性，方法，构造器，实现了哪些接口等等

      3.对于每个类而言，JRE 都为其保留一个不变的 Class 类型的对象。一个 Class 对象包含了特定某个类的有关信息。   
 　  4.Class 对象只能由系统建立对象，一个类（而不是一个对象）在 JVM 中只会有一个Class实例

IMG_258 定义一个Person类

      通过Class类获取类对象

[IMG_259](javascript:void(0);)

public class ReflectionTest {

@Test

public void testClass() {

Class clazz = null;

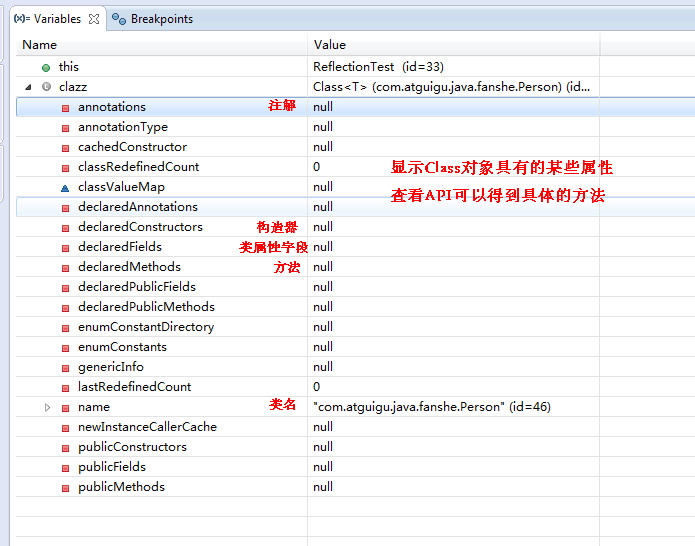
//1.得到Class对象

clazz = Person.class;

System.out.println(); //插入断点 }

}

[IMG_260](javascript:void(0);)

　　在断点处就可以看到Class对像包含的信息  


　　同样，这些属性值是可以获取的

[IMG_262](javascript:void(0);)

public class ReflectionTest {

@Test

public void testClass() {

Class clazz = null;

//1.得到Class对象

clazz = Person.class;

//2.返回字段的数组

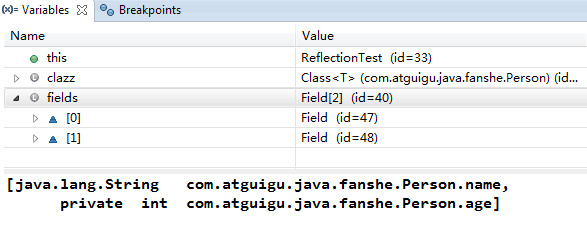
Field[] fields = clazz.getDeclaredFields();

System.out.println(); //插入断点 }

}

[IMG_263](javascript:void(0);)

 　　查看fields的内容



**对象为什么需要照镜子呢？**

　　　　1. 有可能这个对象是别人传过来的

　　　　2. 有可能没有对象，只有一个全类名

　　通过反射，可以得到这个类里面的信息

## ****获取Class对象的三种方式****

**1.通过类名获取      类名.class**

**2.通过对象获取      对象名.getClass()**

**3.通过全类名获取    Class.forName(全类名)**

[IMG_265](javascript:void(0);)

public class ReflectionTest {

@Test

public void testClass() throws ClassNotFoundException {

Class clazz = null;

//1.通过类名

clazz = Person.class;

//2.通过对象名

//这种方式是用在传进来一个对象，却不知道对象类型的时候使用

Person person = new Person();

clazz = person.getClass();

//上面这个例子的意义不大，因为已经知道person类型是Person类，再这样写就没有必要了

//如果传进来是一个Object类，这种做法就是应该的

Object obj = new Person();

clazz = obj.getClass();

//3.通过全类名(会抛出异常)

//一般框架开发中这种用的比较多，因为配置文件中一般配的都是全类名，通过这种方式可以得到Class实例

String className=" com.atguigu.java.fanshe.Person";

clazz = Class.forName(className);

//字符串的例子

clazz = String.class;

clazz = "javaTest".getClass();

clazz = Class.forName("java.lang.String");

System.out.println();

}

}

[IMG_266](javascript:void(0);)

## ****Class类的常用方法****

|  |  |
| --- | --- |
| 方法名 | 功能说明 |
| static Class forName(String name) | 返回指定类名 name 的 Class 对象 |
| Object newInstance() | 调用缺省构造函数，返回该Class对象的一个实例 |
| Object newInstance(Object []args) | 调用当前格式构造函数，返回该Class对象的一个实例 |
| getName() | 返回此Class对象所表示的实体（类、接口、数组类、基本类型或void）名称 |
| Class getSuperClass() | 返回当前Class对象的父类的Class对象 |
| Class [] [getInterfaces](mk:@MSITStore:C:\\ZOL\\JDK 150.chm::/jdk150/api/java/lang/Class.html)() | 获取当前Class对象的接口 |
| ClassLoader getClassLoader() | 返回该类的类加载器 |
| Class getSuperclass() | 返回表示此Class所表示的实体的超类的Class |

**Class类的newInstance（）方法**

[IMG_267](javascript:void(0);)

public void testNewInstance() throws ClassNotFoundException, InstantiationException, IllegalAccessException{

//1.获取Class对象

String className="com.atguigu.java.fanshe.Person";

Class clazz = Class.forName(className);

//利用Class对象的newInstance方法创建一个类的实例

Object obj = clazz.newInstance();

System.out.println(obj);

}

//结果是：com.atguigu.java.fanshe.Person@2866bb78

[IMG_268](javascript:void(0);)

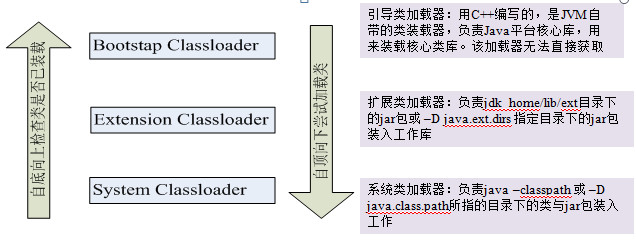
　　可以看出确实是创建了一个Person实例  
　　但是Person类有两个构造方法，到底是调用的哪一个构造方法呢

　　实际调用的是类的**无参数的构造器**。所以在我们在定义一个类的时候，定义一个有参数的构造器，作用是对属性进行初始化，还要写一个无参数的构造器，作用就是反射时候用。

**一般地、一个类若声明一个带参的构造器，同时要声明一个无参数的构造器**

# ****2.ClassLoader****

 　　类装载器是用来把类(class)装载进 JVM 的。JVM 规范定义了两种类型的类装载器：启动类装载器(bootstrap)和用户自定义装载器(user-defined class loader)。 JVM在运行时会产生3个类加载器组成的初始化加载器层次结构 ，如下图所示：



[IMG_270](javascript:void(0);)

public class ReflectionTest {

@Test

public void testClassLoader() throws ClassNotFoundException, FileNotFoundException{

//1. 获取一个系统的类加载器(可以获取，当前这个类PeflectTest就是它加载的)

ClassLoader classLoader = ClassLoader.getSystemClassLoader();

System.out.println(classLoader);

//2. 获取系统类加载器的父类加载器（扩展类加载器，可以获取）.

classLoader = classLoader.getParent();

System.out.println(classLoader);

//3. 获取扩展类加载器的父类加载器（引导类加载器，不可获取）.

classLoader = classLoader.getParent();

System.out.println(classLoader);

//4. 测试当前类由哪个类加载器进行加载（系统类加载器）:

classLoader = Class.forName("com.atguigu.java.fanshe.ReflectionTest")

.getClassLoader();

System.out.println(classLoader);

//5. 测试 JDK 提供的 Object 类由哪个类加载器负责加载（引导类）

classLoader = Class.forName("java.lang.Object")

.getClassLoader();

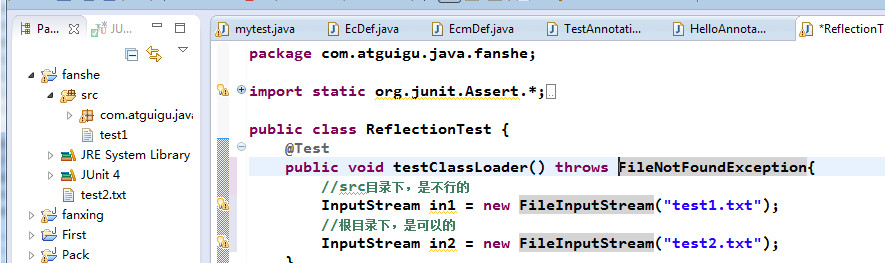
System.out.println(classLoader);

}

}//结果：//sun.misc.Launcher$AppClassLoader@5ffdfb42//sun.misc.Launcher$ExtClassLoader@1b7adb4a//null//sun.misc.Launcher$AppClassLoader@5ffdfb42//null

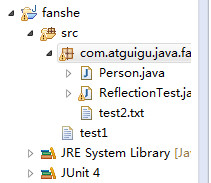
[IMG_271](javascript:void(0);)

**使用类加载器获取当前类目录下的文件**



　　首先，系统类加载器可以加载当前项目src目录下面的所有类，如果文件也放在src下面，也可以用类加载器来加载

　　调用 **getResourceAsStream** 获取类路径下的文件对应的输入流.



[IMG_274](javascript:void(0);)

、public class ReflectionTest {

@Test

public void testClassLoader() throws FileNotFoundException{

//src目录下，直接加载

InputStream in1 = null;

in1 = this.getClass().getClassLoader().getResourceAsStream("test1.txt");

//放在内部文件夹，要写全路径

InputStream in2 = null;

in2 = this.getClass().getClassLoader().getResourceAsStream("com/atguigu/java/fanshe/test2.txt");

}

}

[IMG_275](javascript:void(0);)

# ****3.反射****

## **反射概述**

     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取得任何类的內部信息，并能直接操作任意对象的内部属性及方法。

　　Java反射机制主要提供了以下功能：

* 在运行时构造任意一个类的对象
* 在运行时获取任意一个类所具有的成员变量和方法
* 在运行时调用任意一个对象的方法（属性）
* 生成动态代理

　　Class 是一个类; **一个描述类的类.**

　　封装了描述方法的 Method,

              描述字段的 Filed,

              描述构造器的 Constructor 等属性.

## **3.1如何描述方法-Method**

[IMG_276](javascript:void(0);)

public class ReflectionTest {

@Test

public void testMethod() throws Exception{

Class clazz = Class.forName("com.atguigu.java.fanshe.Person");

//  
 //1.**获取方法** // 1.1 获取取clazz对应类中的所有方法--方法数组（一）

// 不能获取private方法,且获取从父类继承来的所有方法

Method[] methods = clazz.**getMethods**();

for(Method method:methods){

System.out.print(" "+method.getName());

}

System.out.println();

//

// 1.2.获取所有方法，包括私有方法 --方法数组（二）

// 所有声明的方法，都可以获取到，且只获取当前类的方法

methods = clazz.**getDeclaredMethods**();

for(Method method:methods){

System.out.print(" "+method.getName());

}

System.out.println();

//

// 1.3.获取指定的方法

// 需要参数名称和参数列表，无参则不需要写

// 对于方法public void setName(String name) { }

Method method = clazz.**getDeclaredMethod**("setName", String.class);

System.out.println(method);

// 而对于方法public void setAge(int age) { }

method = clazz.getDeclaredMethod("setAge", Integer.class);

System.out.println(method);

// 这样写是获取不到的，如果方法的参数类型是int型

// 如果方法用于反射，**那么要么int类型写成Integer**： public void setAge(Integer age) { }  
　　　　 // 要么获取方法的参数写成int.class

//

//2.**执行方法**

// invoke第一个参数表示执行哪个对象的方法，剩下的参数是执行方法时需要传入的参数

Object obje = clazz.newInstance();

method.**invoke**(obje,2);  
　　　　//如果一个方法是私有方法，第三步是可以获取到的，但是这一步却不能执行  
　**//私有方法的执行，必须在调用invoke之前加上一句method.setAccessible（true）;**  
 }

}

[IMG_277](javascript:void(0);)

　　主要用到的两个方法

[IMG_278](javascript:void(0);)

/\*\*

\* @param name the name of the method

\* @param parameterTypes the list of parameters

\* @return the {@code Method} object that matches the specified

\*/

public Method getMethod(String name, Class<?>... parameterTypes){

}

/\*\*

\* @param obj the object the underlying method is invoked from

\* @param args the arguments used for the method call

\* @return the result of dispatching the method represented by

\*/

public Object invoke(Object obj, Object... args){

}

[IMG_279](javascript:void(0);)

**自定义工具方法**

　　自定义一个方法

**把类对象和类方法名作为参数，执行方法**

**把全类名和方法名作为参数，执行方法**

　　比如Person里有一个方法

public void test(String name,Integer age){

System.out.println("调用成功");

}

　　那么我们自定义一个方法  
     1. **把类对象和类方法名作为参数，执行方法**

[IMG_280](javascript:void(0);)

/\*\*

\*

\* @param obj: 方法执行的那个对象.

\* @param methodName: 类的一个方法的方法名. 该方法也可能是私有方法.

\* @param args: 调用该方法需要传入的参数

\* @return: 调用方法后的返回值

\*

\*/

public Object invoke(Object obj, String methodName, Object ... args) throws Exception{

//1. 获取 Method 对象

// **因为getMethod的参数为Class列表类型，所以要把参数args转化为对应的Class类型。**

Class [] parameterTypes = new Class[args.length];

for(int i = 0; i < args.length; i++){

parameterTypes[i] = args[i].getClass();

System.out.println(parameterTypes[i]);

}

Method method = obj.getClass().getDeclaredMethod(methodName, parameterTypes);

//**如果使用getDeclaredMethod，就不能获取父类方法，如果使用getMethod，就不能获取私有方法**  
　　　　  
　　　　　//  
　　　　　//2. 执行 Method 方法

//3. 返回方法的返回值

return method.invoke(obj, args);

}

[IMG_281](javascript:void(0);)

　　调用：

@Test

public void testInvoke() throws Exception{

Object obj = new Person();

invoke(obj, "test", "wang", 1);

}

　　这样就通过对象名，方法名，方法参数执行了该方法

　　**2.把全类名和方法名作为参数，执行方法**

[IMG_282](javascript:void(0);)

/\*\*

\* @param className: 某个类的全类名

\* @param methodName: 类的一个方法的方法名. 该方法也可能是私有方法.

\* @param args: 调用该方法需要传入的参数

\* @return: 调用方法后的返回值

\*/

public Object invoke(String className, String methodName, Object ... args){

Object obj = null;

try {

obj = Class.forName(className).newInstance();

//调用上一个方法

return invoke(obj, methodName, args);

}catch(Exception e) {

e.printStackTrace();

}

return null;

}

[IMG_283](javascript:void(0);)

　　调用

@Test

public void testInvoke() throws Exception{

invoke("com.atguigu.java.fanshe.Person",

"test", "zhagn", 12);

}

　　使用系统方法（前提是此类有一个无参的构造器（查看API））

@Test

public void testInvoke() throws Exception{

Object result =

invoke("java.text.SimpleDateFormat", "format", new Date());

System.out.println(result);

}

**这种反射实现的主要功能是可配置和低耦合。只需要类名和方法名，而不需要一个类对象就可以执行一个方法。如果我们把全类名和方法名放在一个配置文件中，就可以根据调用配置文件来执行方法**

**如何获取父类定义的（私有）方法**

　　前面说一般使用getDeclaredMethod获取方法（因为此方法可以获取类的私有方法，但是不能获取父类方法）

　　如何获取父类方法呢，上一个例子format方法其实就是父类的方法（获取的时候用到的是getMethod）

　　首先我们要知道，如何获取类的父亲：

　　比如有一个类，继承自Person

　　使用

[IMG_284](javascript:void(0);)

public class ReflectionTest {

@Test

public void testGetSuperClass() throws Exception{

String className = "com.atguigu.java.fanshe.Student";

Class clazz = Class.forName(className);

Class superClazz = clazz.getSuperclass();

System.out.println(superClazz);

}

}//结果是 “ class com.atguigu.java.fanshe.Person ”

[IMG_285](javascript:void(0);)

　　此时如果Student中有一个方法是私有方法method1(int age); Person中有一个私有方法method2();  
　　怎么调用

**定义一个方法，不但能访问当前类的私有方法，还要能父类的私有方法**

[IMG_286](javascript:void(0);)

/\*\*

\*

\* @param obj: 某个类的一个对象

\* @param methodName: 类的一个方法的方法名.

\* 该方法也可能是私有方法, 还可能是该方法在父类中定义的(私有)方法

\* @param args: 调用该方法需要传入的参数

\* @return: 调用方法后的返回值

\*/

public Object invoke2(Object obj, String methodName,

Object ... args){

//1. 获取 Method 对象

Class [] parameterTypes = new Class[args.length];

for(int i = 0; i < args.length; i++){

parameterTypes[i] = args[i].getClass();

}

try {

Method method = getMethod(obj.getClass(), methodName, parameterTypes);

method.setAccessible(true);

//2. 执行 Method 方法

//3. 返回方法的返回值

return method.invoke(obj, args);

} catch (Exception e) {

e.printStackTrace();

}

return null;

}

/\*\*

\* 获取 clazz 的 methodName 方法. 该方法可能是私有方法, 还可能在父类中(私有方法)

\* 如果在该类中找不到此方法，就向他的父类找，一直到Object类为止

　　　\* **这个方法的另一个作用是根据一个类名，一个方法名，追踪到并获得此方法**  
 \* @param clazz

\* @param methodName

\* @param parameterTypes

\* @return

\*/

public Method getMethod(Class clazz, String methodName,

Class ... parameterTypes){

for(;clazz != Object.class; clazz = clazz.getSuperclass()){

try {

Method method = clazz.getDeclaredMethod(methodName, parameterTypes);

return method;

} catch (Exception e) {}

}

return null;

}

[IMG_287](javascript:void(0);)

## ****3.2 如何描述字段-Field****

[IMG_288](javascript:void(0);)

@Test

public void testField() throws Exception{

String className = "com.atguigu.java.fanshe.Person";

Class clazz = Class.forName(className);

//**1.获取字段**

// 1.1 获取所有字段 -- 字段数组

// 可以获取公用和私有的所有字段，但不能获取父类字段

Field[] fields = clazz.getDeclaredFields();

for(Field field: fields){

System.out.print(" "+ field.getName());

}

System.out.println();

// 1.2获取指定字段

Field field = clazz.**getDeclaredField**("name");

System.out.println(field.getName());

Person person = new Person("ABC",12);

**//2.使用字段**

// 2.1获取指定对象的指定字段的值

Object val = field.**get**(person);

System.out.println(val);

// 2.2设置指定对象的指定对象Field值

field.**set**(person, "DEF");

System.out.println(person.getName());

// 2.3如果字段是私有的，不管是读值还是写值，都必须先调用setAccessible（true）方法

// 比如Person类中，字段name字段是公用的，age是私有的

field = clazz.getDeclaredField("age");

field.**setAccessible**(true);

System.out.println(field.get(person));

}

[IMG_289](javascript:void(0);)

　　但是如果需要访问父类中的（私有）字段：

[IMG_290](javascript:void(0);)

/\*\*

\* //创建 className 对应类的对象, 并为其 fieldName 赋值为 val

\* //Student继承自Person,age是Person类的私有字段/  
 public void testClassField() throws Exception{

String className = "com.atguigu.java.fanshe.Student";

String fieldName = "age"; //可能为私有, 可能在其父类中.

Object val = 20;

Object obj = null;

//1.创建className 对应类的对象

Class clazz = Class.forName(className);

//2.创建fieldName 对象字段的对象

Field field = getField(clazz, fieldName);

//3.为此对象赋值

obj = clazz.newInstance();

setFieldValue(obj, field, val);

//4.获取此对象的值

Object value = getFieldValue(obj,field);

}

public Object getFieldValue(Object obj, Field field) throws Exception{

field.setAccessible(true);

return field.get(obj);

}

public void setFieldValue(Object obj, Field field, Object val) throws Exception {

field.setAccessible(true);

field.set(obj, val);

}

public Field getField(Class clazz, String fieldName) throws Exception {

Field field = null;

for(Class clazz2 = clazz; clazz2 != Object.class;clazz2 = clazz2.getSuperclass()){

field = clazz2.getDeclaredField(fieldName);

}

return field;

}

[IMG_291](javascript:void(0);)

**3.3如何描述构造器-Constructor**

[IMG_292](javascript:void(0);)

@Test

public void testConstructor() throws Exception{

String className = "com.atguigu.java.fanshe.Person";

Class<Person> clazz = (Class<Person>) Class.forName(className);

//1. 获取 Constructor 对象

// 1.1 获取全部

Constructor<Person> [] constructors =

(Constructor<Person>[]) Class.forName(className).getConstructors();

for(Constructor<Person> constructor: constructors){

System.out.println(constructor);

}

// 1.2获取某一个，需要参数列表

Constructor<Person> constructor = clazz.getConstructor(String.class, int.class);

System.out.println(constructor);

//2. 调用构造器的 newInstance() 方法创建对象

Object obj = constructor.newInstance("zhagn", 1);

}

[IMG_293](javascript:void(0);)

**3.4 如何描述注解 -- Annotation**

　　定义一个Annotation

[IMG_294](javascript:void(0);)

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

@Retention(RetentionPolicy.RUNTIME)

@Target(value={ElementType.METHOD})public @interface AgeValidator {

public int min();

public int max();

}

[IMG_295](javascript:void(0);)

　　此注解只能用在方法上

@AgeValidator(min=18,max=35)

public void setAge(int age) {

this.age = age;

}

　　那么我们在给Person类对象的age赋值时，是感觉不到注解的存在的

@Test

public void testAnnotation() throws Exception{

Person person = new Person();

person.setAge(10);

}

　　必须通过反射的方式为属性赋值，才能获取到注解

[IMG_296](javascript:void(0);)

/\*\* Annotation 和 反射:

\* 1. 获取 Annotation

\*

\* getAnnotation(Class<T> annotationClass)

\* getDeclaredAnnotations()

\*

\*/

@Test

public void testAnnotation() throws Exception{

String className = "com.atguigu.java.fanshe.Person";

Class clazz = Class.forName(className);

Object obj = clazz.newInstance();

Method method = clazz.getDeclaredMethod("setAge", int.class);

int val = 6;

//获取指定名称的注解

Annotation annotation = method.getAnnotation(AgeValidator.class);

if(annotation != null){

if(annotation instanceof AgeValidator){

AgeValidator ageValidator = (AgeValidator) annotation;

if(val < ageValidator.min() || val > ageValidator.max()){

throw new RuntimeException("年龄非法");

}

}

}

method.invoke(obj, 20);

System.out.println(obj);

}

[IMG_297](javascript:void(0);)

　　**如果在程序中要获取注解，然后获取注解的值进而判断我们赋值是否合法，那么类对象的创建和方法的创建必须是通过反射而来的**

# ****4.反射与泛型****

　　定义一个泛型类

[IMG_298](javascript:void(0);)

public class DAO<T> {

//根据id获取一个对象 T get(Integer id){

return null;

}

//保存一个对象

void save(T entity){

}

}

[IMG_299](javascript:void(0);)

　　再定义一个子类，继承这个泛型类：

public class PersonDAO extends DAO<Person> {

}

　　父类中的泛型T，就相当于一个参数，当子类继承这个类时，就要给这个参数赋值，这里是把Person类型传给了父类

　　或者还有一种做法

public class PersonDAO<T> extends DAO<T> {

}

　　然后进行测试

[IMG_300](javascript:void(0);)

@Test

public void testAnnotation() throws Exception{

PersonDAO personDAO = new PersonDAO();

Person entity = new Person();

//调用父类的save方法，同时也把Person这个“实参”传给了父类的T personDAO.save(entity);

//这句的本意是要返回一个Person类型的对象

Person result = personDAO.get(1);

System.out.print(result);

}

[IMG_301](javascript:void(0);)

　　问题出来了。这里的get方法是父类的get方法，对于父类而言，方法返回值是一个T类型，当T的值为Person时，本该返回一个Person类型，但是必须用反射来创建这个对象（泛型方法返回一个对象），方法无非就是clazz.newInstance(); 所以关键点就是根据T得到其对于的Class对象。

　　那么首先，在父类中定义一个字段，表示T所对应的Class，然后想办法得到这个clazz的值

[IMG_302](javascript:void(0);)

public class DAO<T> {

private Class<T> clazz;

T get(Integer id){

return null;

}

}

[IMG_303](javascript:void(0);)

     如何获得这个clazz呢？

[IMG_304](javascript:void(0);)

@Test

public void test() throws Exception{

PersonDAO personDAO = new PersonDAO();

Person result = personDAO.get(1);

System.out.print(result);

}

[IMG_305](javascript:void(0);)

[IMG_306](javascript:void(0);)

public DAO(){

//1.

System.out.println("DAO's Constrctor...");

System.out.println(this); //结果是：com.atguigu.java.fanshe.PersonDAO@66588ec0

//this：父类构造方法中的this指的是子类对象，因为此时是PersonDAO对象在调用

System.out.println(this.getClass()); //结果是：class com.atguigu.java.fanshe.PersonDAO

//2.

//获取DAO子类的父类

Class class1 = this.getClass().getSuperclass();

System.out.println(class1); //结果是：class com.atguigu.java.fanshe.DAO

//此时只能获的父类的类型名称，却不可以获得父类的泛型参数

//3.

//获取DAO子类带泛型参数的子类

Type type=this.getClass().getGenericSuperclass();

System.out.println(type); //结果是：com.atguigu.java.fanshe.DAO<com.atguigu.java.fanshe.Person>

//此时获得了泛型参数，然后就是把它提取出来

//4.

//获取具体的泛型参数 DAO<T>

//注意Type是一个空的接口，这里使用它的子类ParameterizedType，表示带参数的类类型（即泛型）

if(type instanceof ParameterizedType){

ParameterizedType parameterizedType = (ParameterizedType) type;

Type [] arges = parameterizedType.getActualTypeArguments();

System.out.println(Arrays.asList(arges)); //结果是：[class com.atguigu.java.fanshe.Person]

//得到的是一个数组，因为可能父类是多个泛型参数public class DAO<T,PK>{}

if(arges != null && arges.length >0){

Type arg = arges[0];

System.out.println(arg); //结果是：class com.atguigu.java.fanshe.Person

//获得第一个参数

if(arg instanceof Class){

clazz = (Class<T>) arg;

//把值赋给clazz字段 }

}

}

}

[IMG_307](javascript:void(0);)

　　所以就定义一个方法，**获得 Class 定义中声明的父类的泛型参数类型**

[IMG_308](javascript:void(0);)

public class ReflectionTest {

/\*\*

\* 通过反射, 获得定义 Class 时声明的父类的泛型参数的类型

\* 如: public EmployeeDao extends BaseDao<Employee, String>

\* @param clazz: 子类对应的 Class 对象

\* @param index: 子类继承父类时传入的泛型的索引. 从 0 开始

\* @return

\*/

@SuppressWarnings("unchecked")

public Class getSuperClassGenricType(Class clazz, int index){

Type type = clazz.getGenericSuperclass();

if(!(type instanceof ParameterizedType)){

return null;

}

ParameterizedType parameterizedType =

(ParameterizedType) type;

Type [] args = parameterizedType.getActualTypeArguments();

if(args == null){

return null;

}

if(index < 0 || index > args.length - 1){

return null;

}

Type arg = args[index];

if(arg instanceof Class){

return (Class) arg;

}

return null;

}

@SuppressWarnings("unchecked")

public Class getSuperGenericType(Class clazz){

return getSuperClassGenricType(clazz, 0);

}

@Test

public void testGetSuperClassGenricType(){

Class clazz = PersonDAO.class;

//PersonDAO.class

Class argClazz = getSuperClassGenricType(clazz, 0);

System.out.println(argClazz);

//结果是class com.atguigu.java.fanshe.Person }

}

[IMG_309](javascript:void(0);)

## ****反射小结****

 1. Class: 是一个类; 一个描述类的类.

　　封装了描述方法的 Method,

　　　　   描述字段的 Filed,

              描述构造器的 Constructor 等属性.  
   
 2. 如何得到 Class 对象:  
  　　2.1 Person.class  
  　　2.2 person.getClass()  
  　　2.3 Class.forName("com.atguigu.javase.Person")  
    
 3. 关于 Method:  
  　　3.1 如何获取 Method:  
  　　　　1). getDeclaredMethods: 得到 Method 的数组.  
  　　　　2). getDeclaredMethod(String methondName, Class ... parameterTypes)  
    
  　　3.2 如何调用 Method  
  　　　　1). 如果方法时 private 修饰的, 需要先调用 Method 的　setAccessible(true), 使其变为可访问  
  　　　　2). method.invoke(obj, Object ... args);  
    
  4. 关于 Field:  
  　　4.1 如何获取 Field: getField(String fieldName)  
  　　4.2 如何获取 Field 的值:   
  　　　　1). setAccessible(true)  
  　　　　2). field.get(Object obj)  
  　　4.3 如何设置 Field 的值:  
  　　　　field.set(Obejct obj, Object val)  
    
  5. 了解 Constructor 和 Annotation   
    
  6. 反射和泛型.  
  　　6.1 getGenericSuperClass: 获取带泛型参数的父类, 返回值为: BaseDao<Employee, String>  
  　　6.2 Type 的子接口: ParameterizedType  
  　　6.3 可以调用 ParameterizedType 的 Type[] getActualTypeArguments() 获取泛型参数的数组.