# 1、Filed

## 1、取得class

### 1、Class.forName()

#### 举例

Class<?> demo = Class.forName("com.reflect.Person2");

### 2、Person.class

Field[]fields=Person2.class.getDeclaredFields();

### 3、new 对象出来.class

Person pserson=new Person ();

Field[] fields= pserson.getClass().getDeclaredFields();

## 2、取得反射的字段内容，注意私有和公有public

<http://www.cnblogs.com/manuosex/p/3852716.html>

### 1.测试实体类，y公有 x私有

|  |
| --- |
| public class Point {    private int x;  public int y;  public Point(int x, int y) {  super();  this.x = x;  this.y = y;  }    } |

### 2.1、使用getField 取得公有的y（不能取得私有的）

|  |
| --- |
| Point p = new Point(3,5);    //得到一个字段  Field fieldY = p.getClass().getField("y"); //y 是变量名  System.out.println(fieldY.get(pt1)); // 5 |

### 2.2 如果是私有的getDeclaredField ，setAccessible

#### 1、使用上面的getField会报错，提示找不到NoSuchFieldException

#### 2、getDeclaredField，提示有，但是取不出来java.lang.IllegalAccessException: //Class com.ncs.ReflectTest can not access a member of class com.ncs.Point with modifiers "private"

#### 3、暴力反射2+setAccessible

|  |
| --- |
| Field fieldX = pt1.getClass().getDeclaredField("x");  //这个管你公的私的，都拿来  fieldX.setAccessible(true);//上面的代码已经看见钱了，开始抢了  System.out.println(fieldX.get(pt1));   //out 3 OK!! |

## 3、getType（） 取得字段类型

Field fieldX = pt1.getClass().getDeclaredField("x");

field.getType()== String.class

## 4.、getName，则表示该字段名字（String 类型）

field.getName()

## 4、set把一个类里所有String类型的字段里的b的值 变 成a的值

|  |
| --- |
| Field[] fields = pt1.getClass().getFields();       for(Field field : fields ) {       //我说的是String 类型的字段，当然要选择一下       //if(field.getType().equals(String.class)) {       //字节码都是一份的，干什么用equals啊       if(field.getType()== String.class) {          String oldValue = (String)field.get(pt1);          String newValue = oldValue.replace('b', 'a');           //改 对象的字段            field.set(pt1, newValue);                }          } |

## 5、从xml文件中读出内容利用field反射，放进实体对象中，实战，其中有省略

|  |
| --- |
| Field[] fields=eaheadvo.getClass().getDeclaredFields();  OAXMLHeadVO eaheadvo=new OAXMLHeadVO();  **while**(iteEle.hasNext()){  **for**(**int** i=0;i<fields.length;i++){  Field field=fields[i];  setFieldValue(field, eaheadvo, valueTxt);  } |

### 5.2实现方法如下，这样就会有实体对象出现，其内部也有相应的值

|  |
| --- |
| /\*\*  \* 为属性赋值  \* **@param** field 要赋值的属性  \* **@param** ins 属性所属对象  \* **@param** value 值  \*/  **private** **void** setFieldValue(Field field,Object ins,String value){  **try**{  **if**(field==**null** || ins==**null** || value==**null** || "".equals(value)){  **return**;  }  field.setAccessible(**true**);    String fieldType=field.getGenericType().toString();  **if**("class java.lang.String".equals(fieldType)){  field.set(ins, value);  }  **if**("class java.lang.Integer".equals(fieldType)){  Integer val=Integer.*valueOf*(value);  field.set(ins, val);  }  **if**("class java.math.BigDecimal".equals(fieldType)){  BigDecimal bde=**new** BigDecimal(value);  field.set(ins, bde);  }  **if**("class java.util.Date".equals(fieldType)){  SimpleDateFormat df=**new** SimpleDateFormat("yyyyMMdd");  Date date=df.parse(value);  field.set(ins, date);  }  }**catch**(Exception ex){  logger.error(ex);  **throw** **new** BusinessException("处理失败");  }    } |

## 6、[getDeclaredFields() 与getFields()的区别](http://blog.csdn.net/zhaofeiyue1234/article/details/50781005)

<http://blog.csdn.net/zhaofeiyue1234/article/details/50781005>

<https://yq.aliyun.com/articles/41923>

getFields()与getDeclaredFields()区别:getFields()只能访问类中声明为公有的字段,私有的字段它无法访问，能访问从其它类继承来的公有方法.getDeclaredFields()能访问类中所有的字段,与public,private,protect无关，不能访问从其它类继承来的方法

 getMethods()与getDeclaredMethods()区别:getMethods()只能访问类中声明为公有的方法,私有的方法它无法访问,能访问从其它类继承来的公有方法.getDeclaredFields()能访问类中所有的字段,与public,private,protect无关,不能访问从其它类继承来的方法

getConstructors()与getDeclaredConstructors()区别:getConstructors()只能访问类中声明为public的构造函数.getDeclaredConstructors()能访问类中所有的构造函数,与public,private,protect无关

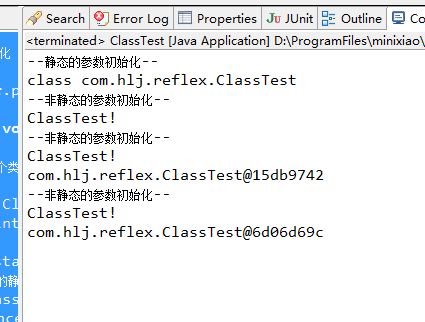
## 7、反射获取父类的字段

Class tempClass = obj .getClass() ;  
while (tempClass!= null && !tempClass.getName().toLowerCase().equals("java.lang.object")) { declaredFields.addAll(Arrays.asList(tempClass .getDeclaredFields()));  
 tempClass = tempClass.getSuperclass();  
}

# 2、Class

|  |
| --- |
| **package** com.hlj.reflex;  **public** **class** ClassTest {  /\*\*  \* 构造函数  \*/  **public** ClassTest() {  System.***out***.println("ClassTest!");  }  // 静态的参数初始化 //只会执行一次  **static** {  System.***out***.println("--静态的参数初始化--");  }  // 非静态的参数初始化 //动态 new 或者 newInstance 实例化对象的时候执行，可执行多次  {  System.***out***.println("--非静态的参数初始化--");  }  **public** **static** **void** main(String[] args) **throws** Exception {  /\*\*  类已加载并且这个类已连接，这是正是class的静态方法forName（）完成的工作。  \*/  Class clazz = Class.*forName*("com.hlj.reflex.ClassTest"); //打印 --静态的参数初始化--  System.***out***.println(clazz); //打印 class com.hlj.reflex.ClassTest  /\*\*  \* 1、使用newInstance可以解耦。使用newInstance的前提是，类已加载并且这个类已连接，  \* 这是正是class的静态方法forName（）完成的工作。newInstance实际上是把new 这个方式分解为两步，  \* 即，首先调用class的加载方法加载某个类，然后实例化  2、newInstance: 弱类型。低效率。只能调用无参构造。 new: 强类型。相对高效。能调用任何public构造。  3、 A a = (A)Class.forName("com.hlj.reflex.ClassTest ").newInstance();  这和下面的是一样的结果  A a = new A()\*/  clazz.newInstance() ;//打印 --非静态的参数初始化-- + ClassTest!  ClassTest classDemo = (ClassTest) clazz.newInstance(); //打印 --非静态的参数初始化-- + ClassTest!  System.***out***.println(classDemo);//com.hlj.reflex.ClassTest@15db9742  //等于上面两行代码  System.***out***.println(**new** ClassTest() ); //打印 --非静态的参数初始化-- + ClassTest! + com.hlj.reflex.ClassTest@6d06d69c    } |

## 打印结果



## Java动态加载类和静态加载类的区别

1.Java动态加载类和静态加载类的区别

new创建对象的方式称作为静态加载，而使用Class.forName("XXX")称作为动态加载，它们俩本质的区别在于静态加载的类的源程序在编译时期加载（必须存在），而动态加载的类在编译时期可以缺席（源程序不必存在）。

2.为什么需要动态加载类

对于我自己的理解，动态加载类增加了程序的灵活性。比如一个程序中有50个功能，但你可能只会使用其中的一个，如果你用的是静态加载的方式，你必须在编译前提供100个功能的所有定义，否则无法编译通过，若你使用的是动态加载机制，则不需要如此大费周章，用哪一个就定义哪一个即可

# 3、Constructor

## 1、构造器类

|  |
| --- |
| **public** **class** ConstructorPerson {  String str ;  **public** String getStr() {  **return** str;  }  **public** **void** setStr(String str) {  **this**.str = str;  }  **public** ConstructorPerson() {  System.***out***.println("无参构造器");  }  **public** ConstructorPerson(String str) {  System.***out***.println("有参 String str 构造器");  **this**.str = str;  }  /\*\*  \* 私有构造函数  \* **@param** str1  \* **@param** str2  \*/  **private** ConstructorPerson(String str1,**int** n) {  System.***out***.println("有参 私有构造器");  **this**.str = str1;  }    } |

## 2、测试类

|  |
| --- |
| //解剖类的构造函数，创建类的对象  **public** **class** ConstructorTest {      **public** **static** **void** main(String[] args) **throws** Exception {  //无参构造器 test1() ;  //有参构造器 test2();  //私有构造器  *test3*();  } |

### 1、无参构造器

|  |
| --- |
| //反射构造函数：public Person() 无参构造器  **public** **static** **void** test1() **throws** Exception{    Class clazz = Class.*forName*("com.hlj.reflex.ConstructorPerson");  Constructor c = clazz.getConstructor(); //一样  // Constructor c = clazz.getConstructor(null);    System.***out***.println(c); //打印 public com.hlj.reflex.ConstructorPerson()    } |

### 2、有参构造器

|  |
| --- |
| //反射构造函数：public Person(String name)  **public** **static** **void** test2() **throws** Exception{    Class clazz = Class.*forName*("com.hlj.reflex.ConstructorPerson");  Constructor c = clazz.getConstructor(String.**class**);  System.***out***.println(c); //打印 public com.hlj.reflex.ConstructorPerson(java.lang.String)  ConstructorPerson person = (ConstructorPerson)c.newInstance("测试成功"); //打印 有参 String str 构造器    System.***out***.println(person.str); //打印 测试成功  } |

### 3、私有构造器，暴力反射

|  |
| --- |
| //反射私有的构造函数：private Person(String str1 ,int n )  **public** **static** **void** test3() **throws** Exception{  Class clazz = Class.*forName*("com.hlj.reflex.ConstructorPerson");  Constructor c = clazz.getDeclaredConstructor(String.**class**,**int**.**class**);    c.setAccessible(**true**);//暴力反射    ConstructorPerson p = (ConstructorPerson) c.newInstance("私有构造器",2);    System.***out***.println(p.str);  } |

# 4、Method

## 1、方法类,注意下面的toString 有不同的使用方式

|  |
| --- |
| **public** **class** MethodPerson {  **private** String name;//名字  **private** String type;//类型  **private** **int** camp;//0,近卫；1，天灾  **public** MethodPerson(){}  **public** MethodPerson(String name, String type, **int** camp) {  **super**();  **this**.name = name;  **this**.type = type;  **this**.camp = camp;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  p**ublic** String getType() {  **return** type;  }  **public** **void** setType(String type) {  **this**.type = type;  }  **public** **int** getCamp() {  **return** camp;  }  **public** **void** setCamp(**int** camp) {  **this**.camp = camp;  }    @Override  **public** String toString() {  **return** "MethodPerson [\n name=" + name + ", \n type=" + type + ", \n camp=" + camp + "\n]";  }    } |

## 2、获取全部的公开的方法

**private** **static** **void** getMethods() {

Class<?> methodPersonClass = MethodPerson.**class**;

Method[] methods = methodPersonClass.getMethods();

**for** (Method method : methods) {

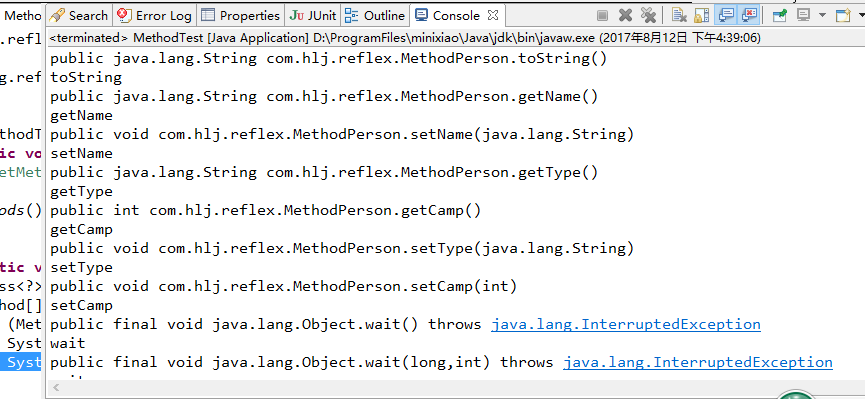
System.***out***.println(method);

System.***out***.println(method.getName());

}

}

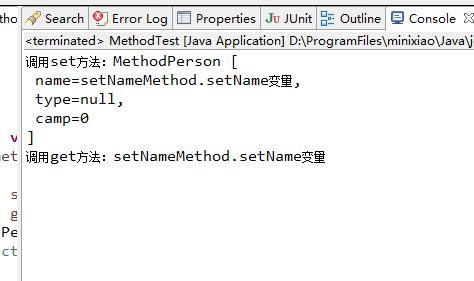
### 2.2打印结果



## 3、获取某个方法，并执行

|  |
| --- |
| **private** **static** **void** getMethodSome() {  Class<?> methodPersonClass = MethodPerson.**class**;  **try** {  Method setNameMethod = methodPersonClass.getMethod("setName",String.**class**);  Method getNameMethod = methodPersonClass.getMethod("getName");  MethodPerson methodPerson = (MethodPerson)methodPersonClass.newInstance();  // Object methodPerson = herosClass.newInstance(); //同上    setNameMethod.invoke(methodPerson,"setNameMethod.setName变量");  /\*\*  \* 1、如果 MethodPerson 类中 没有 toString 方法，则会打印toSring 内容，奇怪吧  \*  \* 打印： 调用set方法：com.hlj.reflex.MethodPerson@6d06d69c  \*  \* 2、如果 MethodPerson 类中 有 toString 方法，则会打印ToString（）方法中的内容  \* 打印： 调用set方法：MethodPerson [  name=setNameMethod.setName变量,  type=null,  camp=0  ]  \*/  System.***out***.println("调用set方法："+methodPerson);  /\*\*  \* 直接打印出getNname的结果值 ，上面已经赋值了，所以打印结果即可  \* 打印：调用get方法：setNameMethod.setName变量  \*/  System.***out***.println("调用get方法："+getNameMethod.invoke(methodPerson));  } **catch** (Exception e) {  e.printStackTrace();  } |

### 3.2 打印结果



# 5、