**利用反射调用构造**

**利用反射调用类中的其他结构（重点）**

Class类是反射中最重要的类，也是所有反射的源头，而对于每一个结构（构造，方法，成员）都将利用Class找到，所以下面将利用反射操作类的其他结构。

范例：操作类中的构造方法：

回顾开发原则，每一个简单的java类都要求提供有无参的构造方法。

**范例：错误的代码：**

**package** fanshe\_1;

**class** Book{

**private** String title;

**private** Double price;

**public** Book(String title) {

**super**();

**this**.title = title;

}

**public** Book(String title, Double price) {

**super**();

**this**.title = title;

**this**.price = price;

}

**public** String toString(){

**return** "书名：" + **this**.title + "价格：" + **this**.price;

}

}

**public** **class** TestGetGouZao {

**public** **static** **void** main(String[] args)**throws** Exception {

Class<?> cls = Class.*forName*("FanSheJiZhi.Book");

cls.newInstance(); //只能调用无参，所以会出错

}

}

这个时候由于类中没有无参构造所以用newinstance一定会出错。

利用Class 类的newInstance()方法可以进行对象的实例化操作，但是类中一定要存在无参构造。不然会报异常。

在Class类中存在以下操作方法：

1. 取得全部构造：public Constructor<?>[] getConstructors() throws SecurityException;
2. 取得指定参数类型的构造：public Constructor<T> getConstructor(Class<?>...parameterTypes) throws ........

在反射过程之中是一个只认类型不认具体对象的工具类，包括在进行方法重载的时候，认的也只是方法名称和参数类型。以上的两个方法返回的是java.lang.reflect.Constructor类的对象。这个类有如下的方法：

1. **取得构造方法名称：public String getConstructors()；**
2. **取得构造方法的修饰符：public int getModefiers();**
3. **取得参数类型：public Class<?> [] getParameterTypes();**

但是所有的修饰符都是通过数字编号取得的，也就是说如果要想取得具体内容必须将相应的数字变回可以读懂的关键字，所以必须使用一个类java.lang.reflectModifier完成，于是在这个类里面提供了一个简单的方法可以进行数字到关键字的还原:public static String toString(int mod);

**范例：取得全部构造方法信息：**

**package** fanshe\_1;

**class** Book{

**private** String title;

**private** Double price;

**public** Book(String title) {

**super**();

**this**.title = title;

}

**public** Book(String title, Double price) {

**super**();

**this**.title = title;

**this**.price = price;

}

**public** String toString(){

**return** "书名：" + **this**.title + "价格：" + **this**.price;

}

}

**public** **class** TestGetGouZao {

**public** **static** **void** main(String[] args)**throws** Exception {

Class<?> cls = Class.*forName*("FanSheJiZhi.Book");

cls.newInstance(); //只能调用无参，所以会出错

}

}

**取得全部构造方法的信息：**

**package** fanshe\_1;

**import** java.lang.reflect.Constructor;

**import** java.lang.reflect.Modifier;

**class** Book{

**private** String title;

**private** Double price;

**public** Book(String title) {

**super**();

**this**.title = title;

}

**public** Book(String title, Double price)**throws** Exception {

**super**();

**this**.title = title;

**this**.price = price;

}

**public** String toString(){

**return** "书名：" + **this**.title + "价格：" + **this**.price;

}

}

**public** **class** TestGetGouZao {

**public** **static** **void** main(String[] args)**throws** Exception {

Class<?> cls = Class.*forName*("fanshe\_1.Book"); //得到类名称

Constructor<?> cons[] = cls.getConstructors(); //取得所有的构造

**for**(**int** x = 0; x < cons.length;x++){ //输出构造方法名

//利用Modifier类的方法getModifier()将数字还原为修饰符

System.***out***.print(Modifier.*toString*(cons[x].getModifiers()) + " ");

//System.out.print(cons[x].getModifiers() + " "); //取得修饰符返回的是1，是public的编号

System.***out***.print(cons[x].getName() + "(");//取得构造方法名称

Class<?> params[] = cons[x].getParameterTypes();//取得全部参数类型

**for**(**int** y = 0;y < params.length;y++){ //取得构造函数的参数

System.***out***.print( params[y].getSimpleName() + " args" + y );

**if**(y < params.length -1){ //若有两个以上的参数则加逗号隔开

System.***out***.print(",");

}

}

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

//------------------------------------------------------

//取得全部异常

Class<?> exp[] = cons[x].getExceptionTypes(); //取得所有构造方法的异常

**if**(exp.length > 0){

System.***out***.print("throws ");

**for**(**int** y = 0;y < exp.length;y++){

System.***out***.println(exp[y].getSimpleName());

**if**(y < exp.length - 1){

System.***out***.println(",");

}

}

}

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

}

}

}

输出结果：public fanshe\_1.Book(String args0)

public fanshe\_1.Book(String args0,Double args1)

throws Exception

在实际的开发环境之中，所有的开发工具都具备这样的操作过程，但是如果采用这样的方式实现了随笔提示，这个代码的执行速度是很慢的。

但是在Constructor类里面提供有一个专门负责实例化对象的方法，这个方法可以传递指定的参数的具体内容。

1.实例化对象：public T newInstance(Object ...initargs)throws...

**范例：调用指定构造实例化对象：**

**package** fanshe\_1;

**import** java.lang.reflect.Constructor;

**import** java.lang.reflect.Modifier;

**class** Book{

**private** String title;

**private** Double price;

**public** Book(String title) {

**super**();

**this**.title = title;

}

**public** Book(String title, Double price)**throws** Exception {

**super**();

**this**.title = title;

**this**.price = price;

}

**public** String toString(){

**return** "书名：" + **this**.title + "价格：" + **this**.price;

}

}

**public** **class** TestGetGouZao {

**public** **static** **void** main(String[] args)**throws** Exception {

Class<?> cls = Class.*forName*("fanshe\_1.Book");

Constructor<?> cons = cls.getConstructor(String.**class**,Double.**class**);

Book book = (Book)cons.newInstance("java开发",80.9);

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

}

}

通过以上的一段代码必须明确的是，类之中只有提供无参构造的时候才是最容易操作的一种，以上的代码如果是有参构造，那么在编写代码的时候麻烦程度是相当高的，而解决方法同样需要用户处理麻烦逻辑，最好的做法就是本着简单java类的标准开发原则进行。

**探究方法的调用（普通方法的调用）**

**1.操作类中的方法：**

清楚了构造方法的调用之后，下面进行普通方法的调用，在Class类里面提供了有两组普通方法的信息取得：

1. **第一组取得本类定义的方法：**
   1. **Public Method[] getDeclaredMethods() throws SecurityException;**
   2. **Public Method getDeclaredMethod(String name,Class<?>...parameterTypes) Throws .....**
2. **第二组取得本类的所有方法：**
   1. **Public Method[] getMethods() throws SecurityException**
   2. **Public Method getMethod(String name ,Class<?>...parameterTypes) throws.....**

范例:编写程序验证区别：

**package** fanshe\_2;

**import** java.lang.reflect.Method;

**interface** Message{

**public** **void** print();

}

**abstract** **class** Info{

**public** **abstract** **void** get();

}

**class** MessageImpl **extends** Info **implements** Message{

@Override

**public** **void** print() {}

@Override

**public** **void** get() {}

**public** **void** set(){}; //自己定义的方法

}

**public** **class** TestDemo\_1 {

**public** **static** **void** main(String[] args) **throws** Exception{

Class<?> cls = Class.*forName*("fanshe\_2.MessageImpl");

//Method met[] = cls.getMethods(); //得到全部的方法，包括继承object类中的方法

Method met[] = cls.getDeclaredMethods(); //得到本类中的全部的方法，不包括继承object类中的方法

**for**(**int** x = 0;x < met.length;x++){

System.***out***.println(met[x]);

}

}

}

以上代码是利用了Method类之中的toString()方法取得了每一个方法的信息，那么现在要求可以自己定义方法输出。

范例：输出全部方法：

**package** fanshe\_2;

**import** java.lang.reflect.Method;

**import** java.lang.reflect.Modifier;

**interface** Message{

**public** **void** print();

}

**abstract** **class** Info{

**public** **abstract** **void** get();

}

**class** MessageImpl **extends** Info **implements** Message{

@Override

**public** **void** print() {}

@Override

**public** **void** get() {}

**public** **void** set(){}; //自己定义的方法

}

**public** **class** TestDemo\_1 {

**public** **static** **void** main(String[] args) **throws** Exception{

Class<?> cls = Class.*forName*("fanshe\_2.MessageImpl");

Method met[] = cls.getMethods(); //得到全部的方法，包括继承object类中的方法

//Method met[] = cls.getDeclaredMethods(); //得到本类中的全部的方法，不包括继承object类中的方法

**for**(**int** x = 0;x < met.length;x++){

System.***out***.print(Modifier.*toString*(met[x].getModifiers()) + " "); //得到权限修饰符

System.***out***.print(met[x].getReturnType().getSimpleName() + " "); //得到简洁的返回值类型

System.***out***.print(met[x].getName()); //得到方法的名字

System.***out***.print("(");

Class<?> params[] = met[x].getParameterTypes();//得到参数的类型

**for**(**int** y = 0;y < params.length;y++){

System.***out***.print(params[y].getSimpleName() + "args" + y); //输出参数类型加参数

**if**(y < params.length - 1){

System.***out***.print(",");

}

}

System.***out***.print(")");

Class<?> exp[] = met[x].getExceptionTypes();//得到所有的异常

**if**(exp.length > 0){

System.***out***.print(" throws ");

**for**(**int** y = 0;y < exp.length;y++){

System.***out***.println(exp[y].getSimpleName()); //输出异常

**if**(y < exp.length -1){

System.***out***.println(",");

}

}

}

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

}

}

}

在之前取得了Constructor类的对象是为了明确调用类之中指定的参数的构造方法，而取得了Method对象不是要进行以上输出操作的。在Method类之中提供有一个种农药方法：

1. **调用本类方法：**public Object invoke(**Object obj,**Object...args)throws....

**在使用以上的invoke()方法操作的时候，一定要保证已经存在了本类实例化对象，而方便的是这种实例化对象可以直接利用Object类代替，可以直接利用Class类反射实例化对象，而后通过Object类对象操作。就没有必要向下转型了。**

范例：利用反射调用方法：

**package** fanshe\_2;

**import** java.lang.reflect.Method;

**class** Message1{

**public** **void** print(String str){//无返回值有参数

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

}

}

//要调用普通方法一定要保证类有实例化对象

**public** **class** TestDemo\_2 {

**public** **static** **void** main(String[] args) **throws** Exception{

Class<?> cls = Class.*forName*("fanshe\_2.Message1");

Object obj = cls.newInstance(); //实例化对象

//已经明确的找到了Method(public void print(String str))的

Method met = cls.getMethod("print", String.**class**);//得到方法，分别传入方法名，和参数类型

met.invoke(obj, "hello word！"); //传入本类对象，和方法参数

}

}

**范例：利用反射调用方法：**

**package** fanshe\_2;

**import** java.lang.reflect.Method;

**class** Message1{

**public** **void** print(String str){

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

}

**public** String echo(String str){

**return** "ECHO" + str;

}

}

//要调用普通方法一定要保证类有实例化对象

**public** **class** TestDemo\_2 {

**public** **static** **void** main(String[] args) **throws** Exception{

Class<?> cls = Class.*forName*("fanshe\_2.Message1");

Object obj = cls.newInstance();

Method met = cls.getMethod("echo", String.**class**);

Object object = met.invoke(obj, "你好啊！"); //为方法传入参数并接受返回值

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

}

}

以上只是演示了Method类的执行操作，但是从实际的开发来讲，反射结合简单java类的情况是最多的，我们一直强调的原则是：简单java类之中的属性一定要进行封装，封装后的属性要按照要求编写setter，getter方法

**范例：利用反射实现getter,setter的调用**

**package** fanshe\_2;

**import** java.lang.reflect.Method;

**class** Message1{

**public** **void** print(String str){

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

}

**public** String echo(String str){

**return** "ECHO" + str;

}

}

//要调用普通方法一定要保证类有实例化对象

**public** **class** TestDemo\_2 {

**public** **static** **void** main(String[] args) **throws** Exception{

// Class<?> cls = Class.forName("fanshe\_2.Message1");

// Object obj = cls.newInstance(); //实例化对象

// //已经明确的找到了Method(public void print(String str))的

// Method met = cls.getMethod("print", String.class);//得到方法，分别传入方法名，和参数类型

// met.invoke(obj, "hello word！"); //传入本类对象，和方法参数

Class<?> cls = Class.*forName*("fanshe\_2.Message1");

Object obj = cls.newInstance();

Method met = cls.getMethod("echo", String.**class**);

Object object = met.invoke(obj, "你好啊！"); //为方法传入参数并接受返回值

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

}

}

**package** fanshe\_2;

**import** java.lang.reflect.Method;

**class** Dept{

**private** String dname;

**public** **void** setDname(String dname){

**this**.dname = dname;

}

**public** String getDname(){

**return** **this**.dname;

}

}

**public** **class** TestDemo\_3 {

**public** **static** **void** main(String[] args)**throws** Exception {

//默认会提供以下内容

String className = "fanshe\_2.Dept";

String property = "dname";

String value = "财务部";

Class<?> cls = Class.*forName*(className); //找到指定的类型

//以下一行代码有一个String.class 会造成耦合度问题

//提供方法名称和参数类型

Method setMet = cls.getDeclaredMethod("set" + StringUtils.*initcap*(property), String.**class**);

Method getMet = cls.getDeclaredMethod("get" + StringUtils.*initcap*(property));//得到getter方法

Object obj = cls.newInstance(); //实例化对象

setMet.invoke(obj, value); // 利用Invoke（）调用本类方法setter()方法 为变量赋值

System.***out***.println(getMet.invoke(obj)); //getter方法

}

}

以上代码就实现了对getter方法和setter方法的调用。

如果想要通过反射设置dname的属性内容，一般都会提供一下内容：

1. 要操作的类名称，提供的目的是取得Class类对象
2. 要操作的类中的属性名称，目的是为了找到与之对应的setter和getter。

要想找到setter，getter方法，那么必须将属性的单词首字母变为大写，这个类JDK一直没有提供。所以我们要自己定义一个工具类来实现这样的操作。

**范例：定义一个自己的工具类**

**package** fanshe\_2;

**public** **class** StringUtils { //将首字母转为大写

**public** **static** String initcap(String str){

**if**(str == **null**){

**return** **null**;

}

**return** str.substring(0,1).toUpperCase() + str.substring(1);

}

}

这个类以后就直接使用了，只要调用方法就可以实现属性名称的大写了