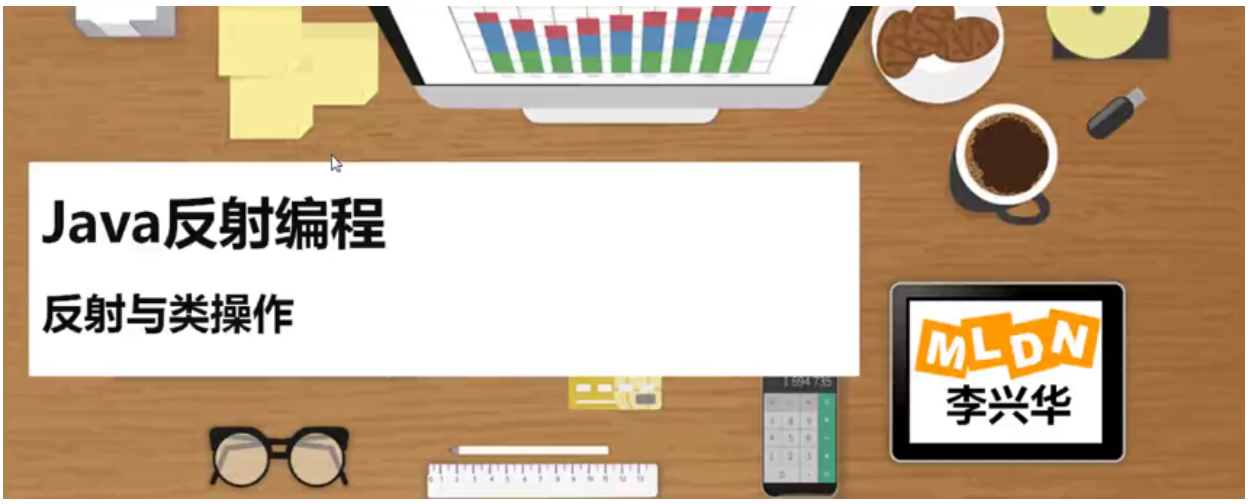


博客： https://www.cnblogs.com/HOsystem/p/14116443.html

2、具体内容

在反射机制的处理过程之中不仅仅只是一个实例化对象的处理操作，更多的情况下还有类的组成结构操作，任何一个类的基本组成结构：父类（父接口）、包、属性、方法（构造方法、普通方法）。

■获取类的基本信息

一个类的基本信息主要包括类所在的包名称、父类的定义、父接口的定义。

范例：定义一个程序类

```
package cn.mldn.service;
public interface IMessageService
{
	public void send() ;
}
```

```
package cn.mldn.service;
public interface
IChannelService {
	public boolean connect() ;
}
```

```
package cn.mldn.abs;
public abstract class
AbstractBase {
}
```

```
package cn.mldn.vo;
import cn.mldn.abs.AbstractBase;
import cn.mldn.service.IChannelService;
import cn.mldn.service.IMessageService;
public class Person extends AbstractBase implements IMessageService,IChannelService {
	@Override
	public boolean connect() {
		return true ;
	}

	@Override
	public void send() {
		if (this.connect()) {
			System.out.println("【信息发送】www.mldn.cn");
		}
	}
}
```

如果此时要想获得类的一些基础信息则可以通过Class类中的如下方法：

·获取包名称：public Package getPackage()；

·获取继承父类：public Class<? super T> getSuperclass()；

·获取实现父接口：public Class<?>[] getInterfaces()；

范例：获得包名称

```
package cn.mldn.demo;
import cn.mldn.vo.Person;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Class<?> cls = Person.class ; // 获取指定类的Class对象
		Package pack = cls.getPackage() ; // 获取指定类的包定义
		System.out.println(pack.getName());
	}
}
```

范例：获取父类信息

```
package cn.mldn.demo;
import cn.mldn.vo.Person;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Class<?> cls = Person.class ; // 获取指定类的Class对象
		Class<?> parent = cls.getSuperclass() ;
		System.out.println(parent.getName());
		System.out.println(parent.getSuperclass().getName());
	}
}
```

范例：获取父接口

```
package cn.mldn.demo;
import cn.mldn.vo.Person;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Class<?> cls = Person.class ; // 获取指定类的Class对象
		Class<?> clazz[] = cls.getInterfaces() ;
		for (Class<?> temp : clazz) {
			System.out.println(temp.getName());
		}
	}
}
```

当获取了一个类的Class对象之后就意味着这个对象可以获取类之中的一切继承结构信息。

■获取构造方法

在一个类之中除了有继承的关系之外最为重要的操作就是类中的结构处理了，而类中的结构里面首先需要观察的就是构造方法的使用问题，实际上在之前通过反射实例化对象的时

候就已经接触到了构造方法的问题了：

·实例化方法替代： clazz.getDeclaredConstructor().newInstance()；

所有类的构造方法的获取都可以直接通过Class类来完成，该类中定义有如下的几个方法：

·获取所有构造方法：public Constructor<?>[] getDeclaredConstructors()throws SecurityException；

·获取指定构造方法：public Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes) throws NoSuchMethodException,SecurityException；

·获取所有构造方法：public Constructor<?>[] getConstructors() throws SecurityException；

·获取指定构造方法：public Constructor<T> getConstructor(Class<?>... parameterTypes) throws NoSuchMethodException,SecurityException；

范例：修改Person类的定义

```
package cn.mldn.abs;
public abstract class AbstractBase {
    public AbstractBase() {}
    public AbstractBase(String msg) {}
}
```

```
package cn.mldn.vo;
import cn.mldn.abs.AbstractBase;
import cn.mldn.service.IChannelService;
import cn.mldn.service.IMessageService;
public class Person extends AbstractBase implements IMessageService,IChannelService {
    public Person() {}
    public Person(String name,int age) {}
    //其它操作略
}
```

范例：获取构造

```
package cn.mldn.demo;
import java.lang.reflect.Constructor;
import cn.mldn.vo.Person;
public class JavaAPIDemo {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Person.class ; // 获取指定类的Class对象
        Constructor<?>[] constructors = cls.getDeclaredConstructors() ; // 获取全部构造
        for (Constructor<?> cons : constructors) {
            System.out.println(cons);
        }
    }
}
```

此时获取的是类之中全部构造方法，但是也可以获取一个指定参数的构造，例如：现在的Person类之中提供有两个构造：

```
package cn.mldn.vo;
import cn.mldn.abs.AbstractBase;
import cn.mldn.service.IChannelService;
import cn.mldn.service.IMessageService;
public class Person extends AbstractBase implements IMessageService,IChannelService {
```

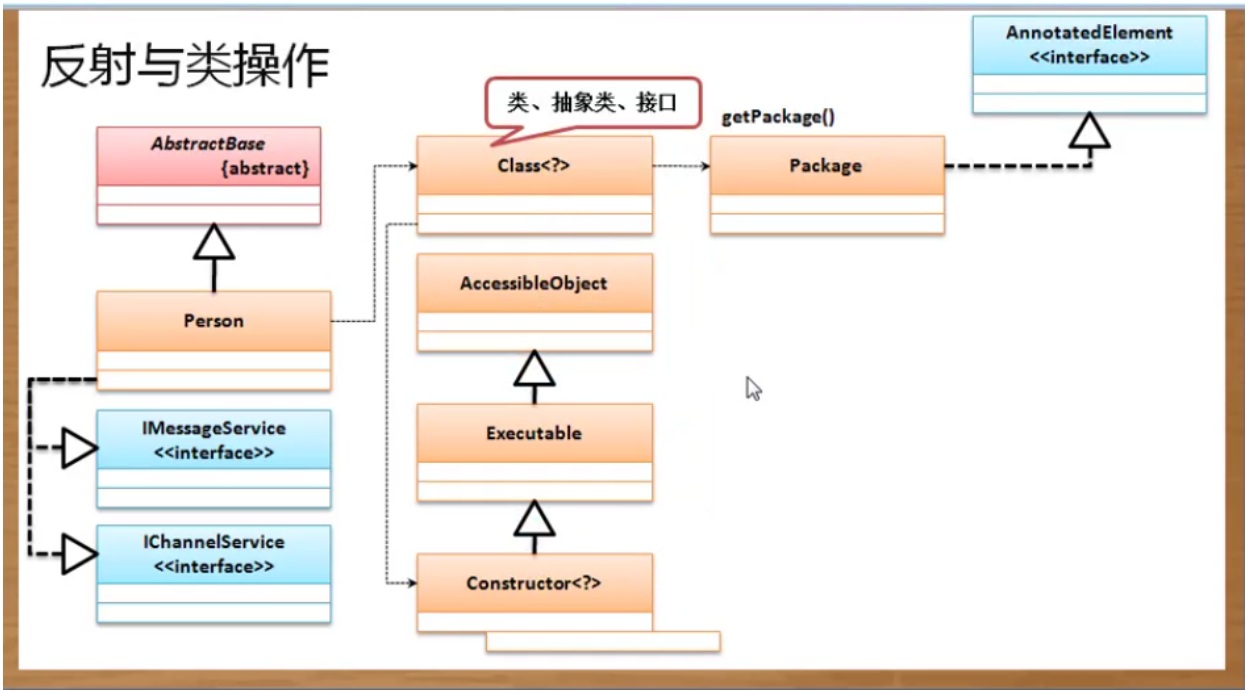
```
    private String name ;
    private int age ;
    public Person() {}
    public Person(String name,int age) {
        this.name = name ;
        this.age = age ;
    }
    @Override
    public String toString() {
        return "姓名：" + this.name + "、年龄：" + this.age ;
    }
    @Override
    public boolean connect() {
        return true ;
    }

    @Override
    public void send() {
        if (this.connect()) {
            System.out.println("【信息发送】www.mldn.cn");
        }
    }
}
```

此时程序打算调用Person类之中的有参构造方法进行Perso类对象的实例化处理，这个时候就必须指明要调用的构造，而后通过Constructor类之中提供的实例化方法操作：public T newInstance(Object... initargs) throws InstantiationException, IllegalAccessException,IllegalArgumentException, InvocationTargetException；

范例：调用指定构造实例化对象

```
package cn.mldn.demo;
import java.lang.reflect.Constructor;
import cn.mldn.vo.Person;
public class JavaAPIDemo {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Person.class ; // 获取指定类的Class对象
        Constructor<?> constructor = cls.getConstructor(String.class,int.class) ;
        Object obj = constructor.newInstance("小强",78) ; // 实例化对象
        System.out.println(obj);
    }
}
```

虽然程序代码本身允许开发者调用有参构造处理，但是如果从实际的开发来讲，所有的使用反射的类中最好提供有无参构造，因为这样的实例化可以达到统一性。



■获取方法

在进行反射处理的时候也可以通过反射来获取类之中的全部方法，但是需要提醒的，如果要想通过反射调用这些方法必须有一个前提条件：类之中要提供有实例化对象。

在Class类里面提供有如下的操作可以获取方法对象：

·获取全部方法：public Method[] getMethods()throws SecurityException；

·获取指定方法：public Method getMethod(String name,Class<?>... parameterTypes)throws NoSuchMethodException,SecurityException；

·获取本类全部方法：public Method[] getDeclaredMethods() throws SecurityException；

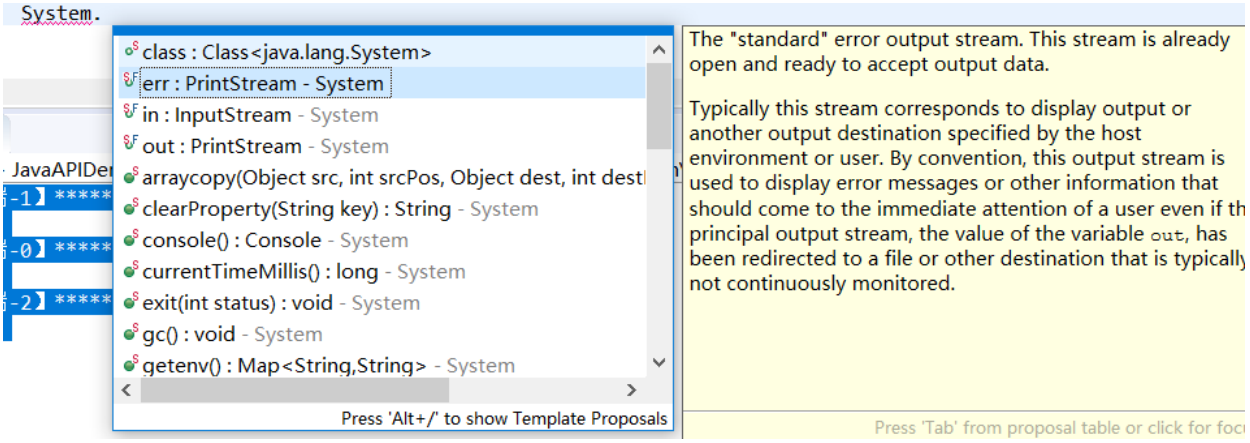
·获取本类指定方法：public Method getDeclaredMethod(String name,Class<?>... parameterTypes) throws NoSuchMethodException,SecurityException；

范例：获取全部方法

```
package cn.mldn.demo;
import java.lang.reflect.Method;
import cn.mldn.vo.Person;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Class<?> cls = Person.class ; // 获取指定类的Class对象
		{	// 获取全部方法（包括父类中的方法）
			Method methods [] = cls.getMethods() ;
			for (Method met : methods) {
				System.out.println(met);
			}
		}
```

```
            System.out.println("------------ 难以忘怀的愚人节的分割线 ---------------");
            {     // 获取本类方法
                  Method methods [] = cls.getDeclaredMethods() ;
                  for (Method met : methods) {
                        System.out.println(met);
                  }
            }
      }
}
```

但是需要注意的是，这个时候的方法信息的获取是依靠Method类提供的toString()方法完成的，很多时候也可以由用户自己来拼凑方法信息的展示信息。



范例：自定义方法信息显示

```
package cn.mldn.demo;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import cn.mldn.vo.Person;
public class JavaAPIDemo {
      public static void main(String[] args) throws Exception {
            Class<?> cls = Person.class; // 获取指定类的Class对象
            Method methods[] = cls.getMethods();
            for (Method met : methods) {
                  int mod = met.getModifiers() ; // 修饰符
                  System.out.print(Modifier.toString(mod) + " ");
                  System.out.print(met.getReturnType().getName() + " ");
                  System.out.print(met.getName() + "(");
                  Class<?> params [] = met.getParameterTypes() ; // 获取参数类型
                  for (int x = 0 ; x < params.length ; x ++) {
                        System.out.print(params[x].getName() + " " + "arg-" + x);
                        if (x < params.length - 1) {
                              System.out.print(",");
                        }
                  }
                  System.out.print(")");
                  Class<?> exp [] = met.getExceptionTypes() ;
                  if (exp.length > 0) {
                        System.out.print(" throws ");
                  }
                  for (int x = 0 ; x < exp.length ; x ++) {
```

```
                    System.out.print(exp[x].getName());
                    if (x < exp.length - 1) {
                        System.out.println(",");
                    }
                }
                System.out.println(); // 换行
            }
        }
}
```

这种代码你只需要清楚可以根据反射获取方法的结构即可，不需要做过多的深入理解，但是在Method类里面有一个致命的重要方法：public Object invoke(Object obj,Object... args) throws IllegalAccessException,IllegalArgumentException,InvocationTargetException；

在Person类里面为name属性追加有setter、getter方法。

```
public class Person extends AbstractBase implements IMessageService,IChannelService {
    private String name ;
    private int age ;
    public Person() {}

    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    //其它操作略
}
```

```
//完整的代码
package cn.mldn.vo;
import cn.mldn.abs.AbstractBase;
import cn.mldn.service.IChannelService;
import cn.mldn.service.IMessageService;
public class Person extends AbstractBase implements IMessageService,IChannelService {
    private String name ;
    private int age ;
    public Person() {}
    public Person(String name,int age) {
        this.name = name ;
        this.age = age ;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    @Override
    public String toString() {
        return "姓名：" + this.name + "、年龄：" + this.age ;
```

```
	}
	@Override
	public boolean connect() {
		return true ;
	}

	@Override
	public void send() {
		if (this.connect()) {
			System.out.println("【信息发送】www.mldn.cn");
		}
	}
}
```

随后需要通过反射机制来实现Person类之中的setter与getter方法的调用处理。

范例：在不导入指定类开发包的情况下实现属性的配置

```
package cn.mldn.demo;
import java.lang.reflect.Method;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Class<?> cls = Class.forName("cn.mldn.vo.Person") ; // 获取指定类的Class对象
		String value = "小强子" ; // 要设置的属性内容
		// 1、任何情况下如果要想保存类中的属性或者调用类中的方法都必须保证存在有实例化对象，既然不允许导入包，那么就反射实例化
		Object obj = cls.getDeclaredConstructor().newInstance() ;// 调用无参构造实例化
		// 2、如果要想进行方法的调用，那么一定要获取方法的名称
		String setMethodName = "setName" ; // 方法名称
		Method setMethod = cls.getDeclaredMethod(setMethodName, String.class) ; // 获取指定的方法
		setMethod.invoke(obj, value) ; // 等价于：Person对象.setName(value);
		String getMethodName = "getName" ;
		Method getMethod = cls.getDeclaredMethod(getMethodName) ; // getter没有参数
		System.out.println(getMethod.invoke(obj)); // 等价于：Person对象.getName()
	}
}
```

利用此类操作整体的形式上不会有任何的明确的类对象产生，一切都是依靠反射机制处理的，这样的处理避免了与某一个类的耦合问题。

■获取成员

类结构之中的最后一个核心的组成就是成员（Field），大部分情况下都会将其称为成员属性，对于成员信息的获取也是通过Class类完成的，在这个类中提供有如下两组操作方法：

·获取本类全部成员：public Field[] getDeclaredFields() throws SecurityException;

·获取本类指定成员：public Field getDeclaredField(String name) throws NoSuchFieldExceptio
n,SecurityException；

·获取父类全部成员：public Field[] getFields() throws SecurityException；

·获取父类指定成员：public Field getField(String name) throws NoSuchFieldException,
SecurityException；

范例：修改要操作的父类结构

```
package cn.mldn.abs;
public abstract class AbstractBase {
	protected static final String BASE = "www.mldn.cn" ;
	//protected修改为public观看现象
	private String info = "Hello MLDN" ;
	public AbstractBase() {}
	public AbstractBase(String msg) {}
}
```

```
package cn.mldn.service;
public interface IChannelService {
	public static final String NAME = "mldnjava" ;
	public boolean connect() ;
}
```

范例：获取类中的成员

```
package cn.mldn.demo;
import java.lang.reflect.Field;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Class<?> cls = Class.forName("cn.mldn.vo.Person") ; // 获取指定类的Class对象
		{	// 获取父类之中的公共成员信息
			Field fields [] = cls.getFields() ;	// 获取成员
			for (Field fie : fields) {
				System.out.println(fie);
			}
		}
		System.out.println("-------- 番茄酱与辣椒酱的分割线 ----------");
		{	// 获取子类中的定义的成员
			Field fields [] = cls.getDeclaredFields() ; // 获取成员
			for (Field fie : fields) {
				System.out.println(fie);
			}
		}
	}
}
```

但是在Field类里面最为重要的操作形式并不是获得全部的成员，而是如下的三个方法：

·设置属性内容：public void set(Object obj,Object value) throws IllegalArgumentException,
IllegalAccessException

·获取属性内容：public Object get(Object obj) throws IllegalArgumentException,IllegalAccessE

xception；

所有的成员是在对象实例化之后进行空间分配的，所以此时一定要先有实例化对象之后才可以进行成员的操作：

范例：直接调用Person类中的name私有成员

```
package cn.mldn.demo;
import java.lang.reflect.Field;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Class<?> cls = Class.forName("cn.mldn.vo.Person") ; // 获取指定类的Class对象
		Object obj = cls.getConstructor().newInstance() ; // 实例化对象（分配成员空间）
		Field nameField = cls.getDeclaredField("name") ; // 获取成员对象
		nameField.setAccessible(true); // 没有封装了
		nameField.set(obj, "番茄强");  // 等价于：Person对象.name = "番茄强" ;
		System.out.println(nameField.get(obj)); // 等价于：Person对象.name
	}
}
```

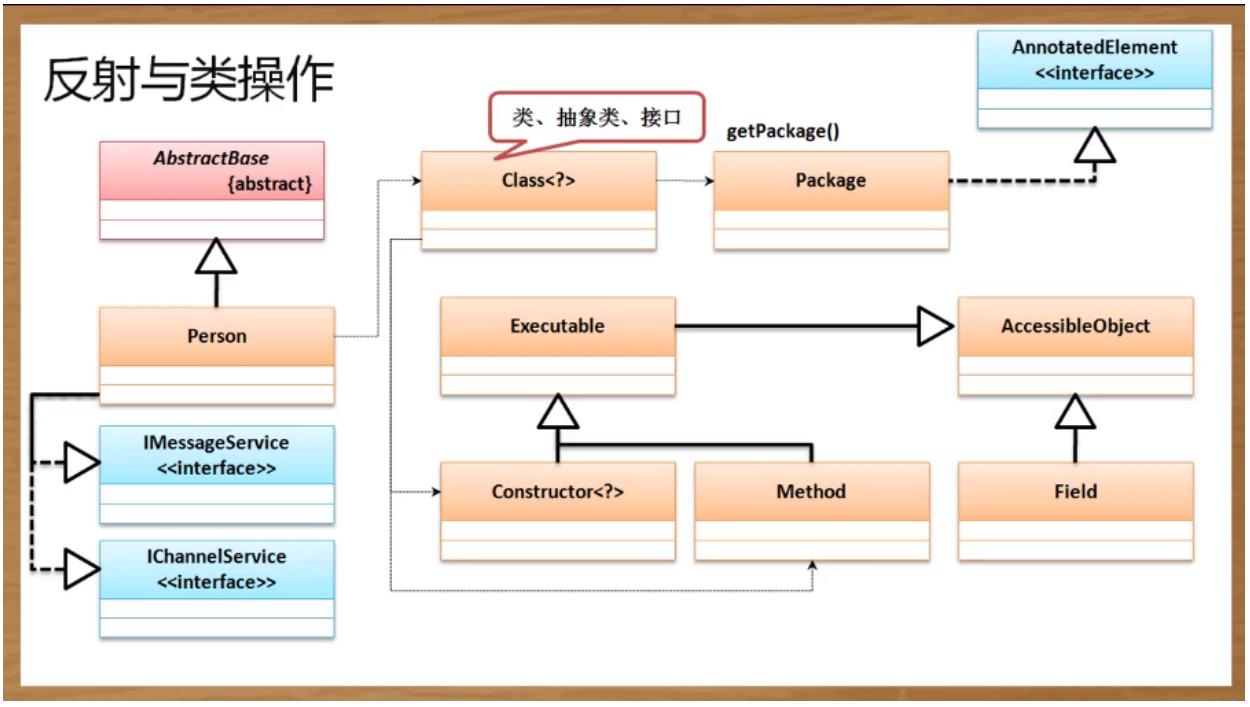
通过一系列的分析可以发现，类之中的构造、方法、成员属性都可以通过反射实现的调用，但是对于成员的反射调用很少这样直接处理，大部分操作都应该通过setter或getter处理，所以对于以上的代码只能够说是反射的特色，但是不具备有实际的使用能力，而对于Field类在实际开发之中只有一个方法最为常用：

·**获取成员类型：public Class<?> getType()**；

范例：获取Person类中的name成员类型

```
package com.mdln.www;

package cn.mldn.demo;
import java.lang.reflect.Field;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Class<?> cls = Class.forName("cn.mldn.vo.Person") ; // 获取指定类的Class对象
		Field nameField = cls.getDeclaredField("name");  //獲取成員對象
		System.out.println(nameField.getType().getName());    //獲取完整類名稱"包.類"
		System.out.println(nameField.getType().getSimpleName());//獲取類名稱
	}
}
```

在以后开发中进行反射处理的时候，往往会利用Field与Method类实现类中的setter方法的调用。



■Unsafe工具类

反射是Java的第一大特点，一旦打开了反射的大门就可以有了更加丰富的类设计形式。除了JVM本身支持的反射处理之外，在Java里面也提供有一个sun.misc.Unsafe（不安全的操作），这个类的主要特点是可以利用反射来获取对象，并且直接使用底层的C++来代替JVM执行，即：可以绕过VM的相关的对象的管理机制，如果你一旦使用了Unsafe类，那么你的项目之中将无法继续使用JVM的内存管理机制以及垃圾回收处理。

但是如果要想使用Unsafe类首先就需要确认一下这个类之中定义的构造方法与常量问题

构造方法：	private Unsafe() {}
私有常量：	private static final Unsafe theUnsafe = new Unsafe();

但是需要注意的是，在这个Unsafe类里面并没有提供static方法，即：不能够通过类似于传统的单例设计模式之中提供的样式来进行操作，如果要想获得这个类的对象，就必须利用反射机制来完成。

```
Field field = Unsafe.class.getDeclaredField("theUnsafe") ; // 获取成员
field.setAccessible(true); // 解除封装
Unsafe unsafeObject = (Unsafe) field.get(null) ; // static属性不需要传递实例化对象
```

在传统的开发之中，一个程序类必须要通过实例化对象后才可以调用类中的普通方法，尤其是以单例设计模式为例。

范例：使用Unsafe类绕过实例化对象的管理

```
package com.mdln.www;
```

```
package cn.mldn.demo;
import java.lang.reflect.Field;
import sun.misc.Unsafe;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Field field = Unsafe.class.getDeclaredField("theUnsafe") ; // 获取成员
		field.setAccessible(true); // 解除封装
		Unsafe unsafeObject = (Unsafe) field.get(null) ; // 原本的Field需要接收实例化对象，但是Unsafe不需要
		//利用Unsafe類繞過了JVM的管理機制，可以在沒有實例化對象的情況下獲取一個Singleton類實例化對象
		Singleton instance = (Singleton) unsafeObject.allocateInstance(Singleton.class) ; // 获取对象，不受JVM控制
		instance.print();// 调用类中的方法
	}
}
class Singleton {
	private Singleton() {
		System.out.println("****** Singleton类的构造方法 ******");
	}
	public void print() {
		System.out.println("www.mldn.cn");
	}
}
```

Unsafe只能够说为我们的开发提供了一些更加方便的处理机制，但是这种操作由于不受JVM的管理所以如果不是必须的情况下不建议使用，而讲解这个类主要的目的是帮助大家巩固对于反射的理解，同时也帮助大家在笔试的时候如果有人问到单例设计模式的情况下，也可以追加一个Unsafe以加深你对这一概念的理解。