

frank 的专栏

人类的一切智慧是包含在这四个字里面的：“等待”和“希望”。——《基督山伯爵》

☰ 目录视图

☰ 摘要视图

RSS 订阅

个人资料



frank909

+ 关注

✉ 发私信



访问：168863次

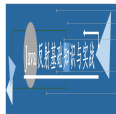
积分：2390

等级：**BLOG > 5**

排名：第15569名

原创：44篇 转载：0篇
译文：1篇 评论：360条

博客专栏



Java 反射基础知识与实战

文章：5篇
阅读：46556

文章分类

- Android笔记 (32)
- Android开发异常汇总 (5)
- Android FrameWork疑点难点Tips (2)
- Android实用开源库 (2)
- 开发工具使用技巧或疑难杂症 (1)
- Java 基础知识 (6)
- Kotlin 学习计划 (2)
- Android 自定义 View (3)
- Java 反射 3 板斧 (4)

文章存档

- 2017年08月 (2)
- 2017年07月 (4)
- 2017年06月 (5)
- 2017年05月 (4)

原 [置顶] 细说反射，Java 和 Android 开发者必须跨越的坎

标签：[android](#) [java](#) [反射](#) [Method](#)

2017-07-06 23:36 👁 12425人阅读 💬 评论(37) ☆ 收藏 ⚠ 举报

☰ 分类：

Java 基础知识 (5) Java 反射 3 板斧 (3)

❗ 版权声明：本文为博主原创文章，未经博主允许不得转载。

目录(?) [+]

写作是门手艺，笑对需要勇气。

写下这个题目的时候，我压力比较大，怕的是费力不讨好。因为反射这一块，对于大多数人员而言太熟悉了，稍微不注意就容易把方向写偏，把知识点写漏。但是，我已经写了注解和动态代理这两个知识点的博客，阅读量还可以，这两个知识点是属于反射机制中的，现在对于注解和动态代理息息相关的反射知识基础我倒是退缩了，所以说看起来很普通的东西，其实真的要一五一十地把它的门道说才方显功力。我们经常说一个人半吊子二把刀，说起来头头是道，做起来却不是那么一回事。

王阳明说知行合一，很多人只让自己停留在知的阶段，没有行，或者说行的能力薄弱，因为没有行来“事上练”，所以就没有办法不停检测自己的“知”是否正确，也就无法“致良知”，这就是王阳明心学，有兴趣的同学可以自行去阅读相关的书籍。听不懂的也没有关系，大体意思就是实践出真理，理论和实践相结合。对于 Java 反射这类基础知识，很多同学看了一遍就觉得懂了，其实很多时候还是没有懂，只是跟着书本被动阅读，你会产生一种错觉，这种错觉就是你以为你懂了，其实，你没有。如何检测呢？很简单，你在阅读某本书，某个章节之后，你合上书本，闭上眼睛，你试着回想一下，你刚才看过的内容，你能记住多少？别不信，你现在就可以找一本书试一试。

讲了这么多，我的观点其实很简单，就是认真对待你的一技之长，尽可能把每个知识点真正弄懂，带着自己的思考去学习新的概念，然后适时做一些练习来检测和巩固。

下面，让我们一起认真对待之前可能没有多在意的基础知识之一——Java 反射。

注意，这篇文章因为内容太多，所以篇幅非常长。中途受不了的同学可以回到目录跳转到感兴趣的小节进行学习。

- [向一个门外汉介绍反射](#)
- [反射入口](#)
 - [Class](#)
 - [Class 的获取](#)
 - [通过 ObjectgetClass](#)



阅读排行

一看你就懂，超详细java中的C...	(18200)
轻松学，Java 中的代理模式及...	(13155)
针对 CoordinatorLayout 及 ...	(12420)
细说反射，Java 和 Android ...	(12375)
秒懂，Java 注解（Annotatio...	(8709)
Android Framework中的线程...	(7739)
反射进阶，编写反射代码值得...	(7487)
轻松学，听说你还没有搞懂 D...	(6445)
通信协议之Protocol buffer(J...	(6211)
OKHTTP之缓存配置详解	(5913)

评论排行

秒懂，Java 注解（Annotatio...	(54)
一看你就懂，超详细java中的C...	(48)
细说反射，Java 和 Android ...	(37)
不再迷惑，也许之前你从未真...	(26)
长谈：关于 View Measure 测...	(20)
轻松学，Java 中的代理模式及...	(17)
反射进阶，编写反射代码值得...	(16)
轻松学，听说你还没有搞懂 D...	(14)
针对 CoordinatorLayout 及 ...	(14)
通信协议之Protocol buffer(J...	(13)

推荐文章

- * CSDN日报20170725——《新的开始，从研究生到入职亚马逊》
- * 深入剖析基于并发AQS的重入锁(Reentrant Lock)及其Condition实现原理
- * Android版本的"Wannacry"文件加密病毒样本分析(附带锁机)
- * 工作与生活真的可以平衡吗？
- * 《Real-Time Rendering 3rd》提炼总结——高级着色：BRDF及相关技术
- * 《三体》读后思考-泰勒展开/维度打击/黑暗森林

最新评论

- 一看你就懂，超详细java中的ClassLoade...
迷糊的悸动：写的真好，根据过程阅读研究了下，发下对class的加载机制理解不是那么模糊了
- Java 泛型，你了解类型擦除吗？
philhong：public void testSuper(Collection<? super Sub>...>
- RecyclerView探索之通过ItemDecoratio...
YaoWatson：佩服！
- OKHTTP之缓存配置详解
frank909：@doubi0511doubi:https 的情况 我没有研究过。不过，你说的情况我建议你自己去编...
- 细说 AppBarLayout.如何理解可折叠 Too...
abs625：清晰易懂，非常给力，支持博主
- Java 泛型，你了解类型擦除吗？
书生语：厉害
- 一看你就懂，超详细java中的ClassLoade...
Tonado_1：楼主写得也太棒了，不过我还有个疑问，classloader在加载一个类的时候会自动加载这个类的父类吗？...

- 通过 class 标识
- 通过 Class.forName 方法

• Class 内容清单

- Class 的名字
 - 当 Class 代表一个引用时
 - 当 Class 代表一个基本数据类型比如 int 的时候
 - 当 Class 代表的是基础数据类型的数组时 比如 int 这样的 3 维数组时
 - simpleName 的不同
- Class 获取修饰符
- 获取 Class 的成员
 - 获取 Filed
 - 获取 Method
 - 获取 Constructor
- Field 的操控
 - Field 类型的获取
 - Field 修饰符的获取
 - Field 内容的读取与赋值
- Method 的操控
 - Method 获取方法名
 - Method 获取方法参数
 - Method 获取返回值类型
 - Method 获取修饰符
 - Method 获取异常类型
 - Method 方法的执行
- Constructor 的操控
- 反射中的数组
 - 反射中动态创建数组
 - Array 的读取与赋值
- 反射中的枚举 Enum
 - 枚举的获取与设定

• 反射与自动驾驶

• 总结

向一个门外汉介绍反射

反射是什么？

官方文档上有这么一段介绍：

Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the **Java** virtual machine. This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language. With that caveat in mind,



微信关注CSDN
获得无限技术资源

快速回复

☆ 我要收藏

返回顶部

OKHTTP之缓存配置详解
doubi0511doubi : 还有一个问题，如果设置了缓存后一个请求被并发执行了多次（比如刷新token）。那么第二次会等待第一次...
OKHTTP之缓存配置详解
doubi0511doubi : 请教：此缓存机制对于https是否同样适用？
一看你就懂，超详细java中的ClassLoad...
lyt645774075 : 专门登录感谢，写得非常好，学习了

统计

reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible.

我来翻译一下：反射技术通常被用来检测和改变应用程序在 Java 虚拟机中的行为表现。它是一个相对而言比较高级的技术，通常它应用的前提是开发者本身对于 Java 语言特性有很强理解的基础上。值得说明的是，反射是一种强有力的技术特性，因此可以使得应用程序突破一些藩篱，执行一些常规手段无法企及的目的。

我再通俗概括一下：**反射是个很牛逼的功能，能够在程序运行时修改程序的行为。但反射是非常规手段，反射有风险，应用需谨慎。**

相信，大部分同学会有稍微清晰一点的概念了。但这还不是我的目的所在。

我的目的是想，**我如何向一个刚有一点点 Java 基础的初学者，或者说毫无 Java 基础的门外汉解释清楚反射这样一种东西？**

直接翻译官方文档，显然是不太行。因为那仍然是抽象的，所以，最好的方法仍然是通过类比或者是拟人，用生活场景中具体的事物与抽象的概念建立相关性。

把程序代码比作一辆车，因为 Java 是面向对象的语言，所以这样很容易理解，正常流程中，车子有自己的颜色、车型号、品牌这些属性，也有正常行驶、倒车、停泊这些功能操作。

正常情况下，我们需要为车子配备一个司机，然后按照行为准则规范行驶。

那么反射是什么呢？反射是非常规手段，正常行驶的时候，车子需要司机的驾驶，但是，反射却不需要，因为它就是车子的——自动驾驶。



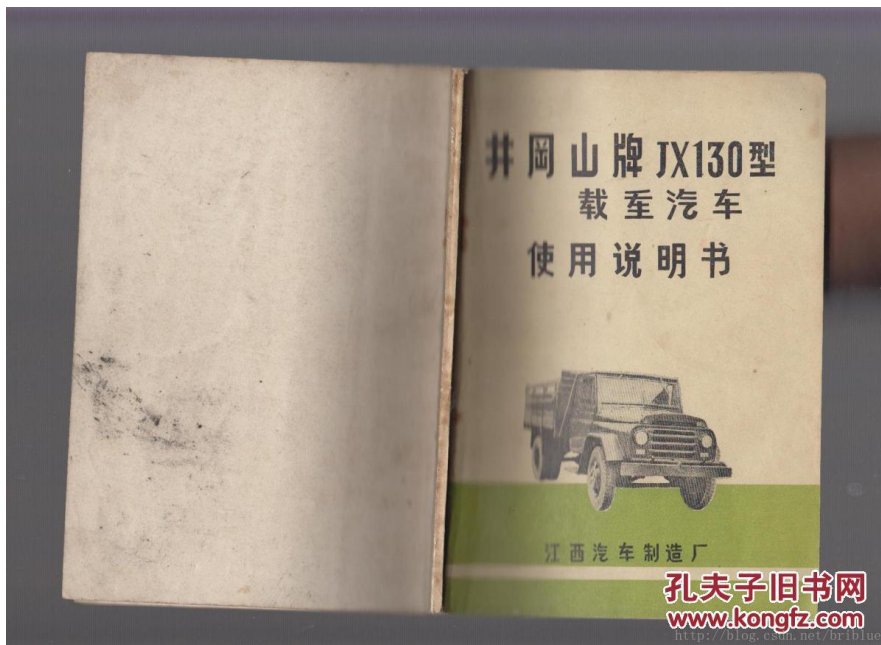
因为，反射牛逼，又因为反射非常规，所以，它风险未知，需要开发者极强的把控力。而汽车中的自动驾驶技术现在是热门，但是特斯拉都出过故障，所以同样在汽车领域，自动驾驶技术也需要车厂家有极牛逼的风险把控能力，这个基础就是要遵从汽车本身的结构与交通规则，不能因为运用了自动驾驶技术的汽车就不叫做汽车了，应用了反射技术的代码就不叫做代码了。

自动驾驶需要遵守基础规则，同样反射也需要，下面的文章就是介绍反射技术应该遵守的规格与限制。

反射入口

我们试想一下，如果自动驾驶要运用到一辆汽车之上，研发人员首先要拿到的是什么？

肯定是汽车的规格说明书。



同样，反射如果要作用于一段 Java 代码上，那么它也需要拿到一本规格说明书，那么对于反射而言，这本规格说明书是什么呢？

Class

因为 Java 是面向对象的语言，基本上是以类为基础构造了整个程序系统，反射中要求提供的规格说明书其实就是一个类的规格说明书，它就是 Class。

注意的是 Class 是首字母大写，不同于 class 小写，class 是定义类的关键字，而 Class 的本质也是一个类，因为在 Java 中一切都是对象。

```
1 public final class Class<T> implements java.io.Serializable,  
2                                     GenericDeclaration,  
3                                     Type,  
4                                     AnnotatedElement {}
```

Class 就是一个对象，它用来代表运行在 Java 虚拟机中的类和接口。

把 Java 虚拟机类似于高速公路，那么 Class 就是用来描述公路上飞驰的汽车，也就是我前面提到的规格说明书。

Class 的获取

反射的入口是 Class，但是反射中 Class 是没有公开的构造方法的，所以就没有办法像创建一个类一样通过 new 关键字来获取一个 Class 对象。

不过，不用担心，Java 反射中 Class 的获取可以通过下面 3 种方式。

1. 通过 Object.getClass()

对于一个对象而言，如果这个对象可以访问，那么调用 `getClass()` 方法就可以获取到了它的相应的 Class 对象。

```
1 public class Car {}
2
3 public class Test {
4
5     public static void main(String[] args) {
6
7         Car car = new Car();
8
9         Class clazz = car.getClass();
10    }
11 }
12 }
13 }
```

值得注意的是，这种方法不适合基本类型如 `int`、`float` 等等。

2. 通过 `.class` 标识

上面的例子中，`Car` 是一个类，`car` 是它的对象，通过 `car.getClass()` 就获取到了 `Car` 这个类的 Class 对象，也就是说通过一个类的实例的 `getClass()` 方法就能获取到它的 Class。如果不想创建这个类的实例的话，就需要通过 `.class` 这个标识。

```
1 public class Test {
2
3     public static void main(String[] args) {
4
5         Class clazz = Car.class;
6         Class cls1 = int.class;
7         Class cls2 = String.class;
8
9     }
10 }
```

3. 通过 `Class.forName()` 方法

有时候，我们没有办法创建一个类的实例，甚至没有办法用 `Car.class` 这样的方式去获取一个类的 Class 对象。

这在 **Android** 开发领域很常见，因为某种目的，**android** 工程师把一些类加上了 `@hide` 注解，所示这些类就没有出现在 SDK 当中，那么，我们要获取这个并不存在于当前开发环境中的类的 Class 对象时就没有辙了吗？答案是否定的，Java 给我们提供了 `Class.forName()` 这个方法。

只要给这个方法中传入一个类的全限定名称就好了，那么它就会到 Java 虚拟机中去寻找这个类有没有被加载。

```
1 try {
2     Class clz = Class.forName("com.frank.test.Car");
3 } catch (ClassNotFoundException e) {
4     // TODO Auto-generated catch block
5     e.printStackTrace();
6 }
```

“`com.frank.test.Car`” 就是 `Car` 这个类的全限定名称，它包括包名+类名。

如果找不到时，它会抛出 `ClassNotFoundException` 这个异常，这个很好理解，因为如果查找的类

没有在 JVM 中加载的话，自然要告诉开发者。

所以，上面 3 节讲述了如何拿到一个类的 Class 对象。

Class 内容清单

仅仅拿到 Class 对象还不够，我们感兴趣的是它的内容。

在正常的代码编写中，我们如果要编写一个类，一般会定义它的属性和方法，如：

```
1 public class Car {
2
3     private String mBand;
4
5     private Color mColor;
6
7     public enum Color {
8         RED,
9         WHITE,
10        BLACK,
11        BLUE,
12        YELLOR
13    }
14
15
16
17    public Car() {
18        super();
19        // TODO Auto-generated constructor stub
20    }
21
22
23    public Car(String mBand) {
24        this.mBand = mBand;
25    }
26
27
28    public void drive() {
29        System.out.println("di di di, 开车了!");
30    }
31
32    @Override
33    public String toString() {
34        return "Car [mBand=" + mBand + ", mColor=" + mColor + "]";
35    }
36
37
38 }
```

现在我们来——分解它。

Class 的名字

Class 对象也有名字，涉及到的 API 有：

```
1 Class.getName();
2
3 Class.getSimpleName();
4
5 Class.getCanonicalName();
```

现在，说说它们的区别。

因为 Class 是一个入口，它代表引用、基本数据类型甚至是数组对象，所以获取它们的方式又有一点不同。

先从 getName() 说起。

当 Class 代表一个引用时

getName() 方法返回的是一个二进制形式的字符串，比如 “com.frank.test.Car”。

当 Class 代表一个基本数据类型，比如 int.class 的时候

getName() 方法返回的是它们的关键字，比如 int.class 的名字是 int。

当 Class 代表的是基础数据类型的数组时 比如 int[][][] 这样的 3 维数组时

getName() 返回 [[I 这样的字符串。

为什么会这样呢？这是因为，Java 本身对于这一块制定了相应规则，在元素的类型前面添加相应数量的 [符号，用 I 的个数来提示数组的维度，并且值得注意的是，对于基本类型或者是类，都有相应的编码，所谓的编码大多数是用一个大写字母来指示某种类型，规则如下：

元素类型	编码
boolean	Z
byte	B
char	C
double	D
float	F
int	I
long	J
short	S

类或者接口 **L类名;**

需要注意的是类或者是接口的类型编码是 **L类名;** 的形式,后面有一个分号。

比如 String[].getClass().getName() 结果是 [Ljava.lang.String;。

我们来测试一下代码：

```
1 public class Test {
2
3     public static void main(String[] args) {
4
5         try {
6             Class clz = Class.forName("com.frank.test.Car");
7
8             Class clz1 = float.class;
9
10            Class clz2 = Void.class;
11
12            Class clz3 = new int[] {}.getClass();
13
14            Class clz4 = new Car[] {}.getClass();
15        }
```

```

16         System.out.println(clz.getName());
17         System.out.println(clz1.getName());
18         System.out.println(clz2.getName());
19         System.out.println(clz3.getName());
20         System.out.println(clz4.getName());
21
22
23     } catch (ClassNotFoundException e) {
24         // TODO Auto-generated catch block
25         e.printStackTrace();
26     }
27
28 }
29
30 }

```

上面代码的打印结果如下：

```

1 com.frank.test.Car
2 float
3 java.lang.Void
4 [I
5 [Lcom.frank.test.Car;

```

刚刚介绍的都是 `getName()` 的情况，那么 `getSimpleName()` 和 `getCanonicalName()` 呢？

`getSimpleName()` 自然是要去获取 `simplename` 的，那么对于一个 `Class` 而言什么是 `SimpleName` 呢？我们先要从嵌套类说起

```

1 public class Outer {
2
3     static class Inner {}
4
5 }

```

`Outer` 这个类中有一个静态的内部类。

```

1 Class clz = Outer.Inner.class;
2
3 System.out.println(" Inner Class name:"+clz.getName());
4 System.out.println(" Inner Class simple name:"+clz.getSimpleName());

```

我们分别打印 `Inner` 这个类的 `Class` 对象的 `name` 和 `simplename`。

```

1 Inner Class name:com.frank.test.Outer$Inner
2 Inner Class simple name:Inner

```

可以看到，因为是内部类，所以通过 `getName()` 方法获取到的是二进制形式的全限定类名，并且类名前面还有个 `$` 符号。

`getSimpleName()` 则直接返回了 `Inner`，去掉了包名限定。

打个比方，我的全名叫做 Frank Zhao，而我的 `simplename` 就叫做 `frank`，`simplename` 之于 `name` 也是如此。

simplename 的不同

需要注意的是，当获取一个数组的 `Class` 中的 `simplename` 时，不同于 `getName()` 方法，`simplename` 不是在前面加 `[]`，而是在后面添加对应数量的 `[]`。

```

1 Class clz = new Outer.Inner[][] {}.getClass();

```



```

2
3 System.out.println(" Inner Class name:"+clz.getName());
4 System.out.println(" Inner Class simple name:"+clz.getSimpleName());

```

上面代码打印结果是：

```

1 Inner Class name:[[Lcom.frank.test.Outter$Inner;
2 Inner Class simple name:Inner[] [] []

```

还需要注意的是，对于匿名内部类，`getSimpleName()` 返回的是一个空的字符串。

```

1 Runnable run = new Runnable() {
2
3     @Override
4     public void run() {
5         // TODO Auto-generated method stub
6     }
7 };
8
9
10 System.out.println(" Inner Class name:"+run.getClass().getName());
11 System.out.println(" Inner Class simple name:"+run.getClass().getSimpleName());

```

打印结果是：

```

1 anonymous Class name:com.frank.test.Test$1
2 anonymous Class simple name:

```

最后再来看 `getCanonicalName()`。

Canonical 是官方、标准的意思，那么 `getCanonicalName()` 自然就是返回一个 Class 对象的官方名字，这个官方名字 `canonicalName` 是 Java 语言规范制定的，如果 Class 对象没有 `canonicalName` 的话就返回 `null`。

`getCanonicalName()` 是 `getName()` 和 `getSimpleName()` 的结合。

- `getCanonicalName()` 返回的也是全限定类名，但是对于内部类，不用 \$ 开头，而用 ..
- `getCanonicalName()` 对于数组类型的 Class，同 `simplename` 一样直接在后面添加 []。
- `getCanonicalName()` 不同于 `simplename` 的地方是，不存在 `canonicalName` 的时候返回 `null` 而不是空字符串。
- 局部类和匿名内部类不存在 `canonicalName`。

```

1 Class clz = new Outter.Inner[] [] {}.getClass();
2
3 System.out.println(" Inner Class name:"+clz.getName());
4 System.out.println(" Inner Class simple name:"+clz.getSimpleName());
5 System.out.println(" Inner Class canonical name:"+clz.getCanonicalName());
6
7
8 //run 是匿名类
9 Runnable run = new Runnable() {
10
11     @Override
12     public void run() {
13         // TODO Auto-generated method stub
14     }
15 };
16
17
18 System.out.println(" anonymous Class name:"+run.getClass().getName());
19 System.out.println(" anonymous Class simple name:"+run.getClass().getSimpleName());

```

```

20 System.out.println(" anonymous Class canonical name:"+run.getClass().getCanonicalName());
21
22 // local 是局部类
23 class local{};
24
25
26 System.out.println("Local a name:"+local.class.getName());
27 System.out.println("Local a simplename:"+local.class.getSimpleName());
28 System.out.println("Local a canonicalname:"+local.class.getCanonicalName());

```

打印结果如下：

```

1 Inner Class name:[[Lcom.frank.test.Outter$Inner;
2 Inner Class simple name:Inner[] [] []
3 Inner Class canonical name:com.frank.test.Outter.Inner[] [] []
4
5 anonymous Class name:com.frank.test.Test$1
6 anonymous Class simple name:
7 anonymous Class canonical name:null
8
9 Local a name:com.frank.test.Test$1local
10 Local a simplename:local
11 Local a canonicalname:null

```

Class 去获取相应名字的知识内容就讲完了，仔细想一下，小小的一个细节，其实蛮有学问的。

好了，我们继续往下。

Class 获取修饰符

通常，Java 开发中定义一个类，往往是要通过许多修饰符来配合使用的。它们大致分为 4 类。

- 用来限制作用域，如 public、protected、private。
- 用来提示子类复写，abstract。
- 用来标记为静态类 static。
- 注解。

Java 反射提供了 API 去获取这些修饰符。

```

1 package com.frank.test;
2
3 public abstract class TestModifier {
4
5 }

```

我们定义了一个类，名字为 TestModifier，被 public 和 abstract 修饰，现在我们要提取这些修饰符。我们只需要调用 Class.getModifiers() 方法就是了，它返回的是一个 int 数值。

```

1 System.out.println("modifiers value:"+TestModifier.class.getModifiers());
2 System.out.println("modifiers :"+Modifier.toString(TestModifier.class.getModifiers()))

```

打印结果是：

```

1 modifiers value:1025
2 modifiers :public abstract

```

大家肯定会有疑问，为什么会返回一个整型数值呢？

这是因为一个类定义的时候可能会被多个修饰符修饰，为了一并获取，所以 Java 工程师考虑到了位运算，用一个 int 数值来记录所有的修饰符，然后不同的位对应不同的修饰符，这些修饰符对应的位都定义在 Modifier 这个类当中。

```
1 public class Modifier {
2
3     public static final int PUBLIC          = 0x00000001;
4
5
6     public static final int PRIVATE         = 0x00000002;
7
8
9     public static final int PROTECTED      = 0x00000004;
10
11
12     public static final int STATIC         = 0x00000008;
13
14
15     public static final int FINAL          = 0x00000010;
16
17
18     public static final int SYNCHRONIZED   = 0x00000020;
19
20
21     public static final int VOLATILE       = 0x00000040;
22
23
24     public static final int TRANSIENT      = 0x00000080;
25
26
27     public static final int NATIVE         = 0x00000100;
28
29
30     public static final int INTERFACE      = 0x00000200;
31
32
33     public static final int ABSTRACT       = 0x00000400;
34
35
36     public static final int STRICT         = 0x00000800;
37
38     public static String toString(int mod) {
39         StringBuilder sb = new StringBuilder();
40         int len;
41
42         if ((mod & PUBLIC) != 0)          sb.append("public ");
43         if ((mod & PROTECTED) != 0)       sb.append("protected ");
44         if ((mod & PRIVATE) != 0)         sb.append("private ");
45
46         /* Canonical order */
47         if ((mod & ABSTRACT) != 0)        sb.append("abstract ");
48         if ((mod & STATIC) != 0)          sb.append("static ");
49         if ((mod & FINAL) != 0)           sb.append("final ");
50         if ((mod & TRANSIENT) != 0)       sb.append("transient ");
51         if ((mod & VOLATILE) != 0)        sb.append("volatile ");
52         if ((mod & SYNCHRONIZED) != 0)    sb.append("synchronized ");
53         if ((mod & NATIVE) != 0)          sb.append("native ");
54         if ((mod & STRICT) != 0)          sb.append("strictfp ");
55         if ((mod & INTERFACE) != 0)       sb.append("interface ");
56
57         if ((len = sb.length()) > 0)      /* trim trailing space */
58             return sb.toString().substring(0, len-1);
59         return "";
60     }
61 }
62 }
```

调用 Modifier.toString() 方法就可以打印出一个类的所有修饰符。

当然，`Modifier` 还提供了一系列的静态工具方法用来对修饰符进行操作。

```
1 public static boolean isPublic(int mod) {
2     return (mod & PUBLIC) != 0;
3 }
4
5
6 public static boolean isPrivate(int mod) {
7     return (mod & PRIVATE) != 0;
8 }
9
10
11 public static boolean isProtected(int mod) {
12     return (mod & PROTECTED) != 0;
13 }
14
15
16 public static boolean isStatic(int mod) {
17     return (mod & STATIC) != 0;
18 }
19
20
21 public static boolean isFinal(int mod) {
22     return (mod & FINAL) != 0;
23 }
24
25
26 public static boolean isSynchronized(int mod) {
27     return (mod & SYNCHRONIZED) != 0;
28 }
29
30
31 public static boolean isVolatile(int mod) {
32     return (mod & VOLATILE) != 0;
33 }
34
35
36 public static boolean isTransient(int mod) {
37     return (mod & TRANSIENT) != 0;
38 }
39
40
41 public static boolean isNative(int mod) {
42     return (mod & NATIVE) != 0;
43 }
44
45
46 public static boolean isInterface(int mod) {
47     return (mod & INTERFACE) != 0;
48 }
49
50
51 public static boolean isAbstract(int mod) {
52     return (mod & ABSTRACT) != 0;
53 }
54
55
56 public static boolean isStrict(int mod) {
57     return (mod & STRICT) != 0;
58 }
```

这些代码的作用，一看就懂，所以不再多说。

获取 Class 的成员

一个类的成员包括属性（有人翻译为字段或者域）、方法。对应到 `Class` 中就是 `Field`、`Method`、

Constructor。

获取 Filed

获取指定名字的属性有 2 个 API

```
1 public Field getDeclaredField(String name)
2             throws NoSuchFieldException,
3             SecurityException;
4
5 public Field getField(String name)
6             throws NoSuchFieldException,
7             SecurityException
```

两者的区别就是 `getDeclaredField()` 获取的是 Class 中被 `private` 修饰的属性。 `getField()` 方法获取的是非私有属性，并且 `getField()` 在当前 Class 获取不到时会向祖先类获取。

获取所有的属性。

```
1 //获取所有的属性，但不包括从父类继承下来的属性
2 public Field[] getDeclaredFields() throws SecurityException {}
3
4 //获取自身的所有的 public 属性，包括从父类继承下来的。
5 public Field[] getFields() throws SecurityException {
```

可以用一个例子，给大家加深一下理解。

```
1 public class Farther {
2
3     public int a;
4
5     private int b;
6
7 }
8
9 public class Son extends Farther {
10     int c;
11
12     private String d;
13
14     protected float e;
15 }
16
17
18 package com.frank.test;
19
20 import java.lang.reflect.Field;
21
22 public class FieldTest {
23
24     public static void main(String[] args) {
25         // TODO Auto-generated method stub
26
27         Class cls = Son.class;
28
29         try {
30             Field field = cls.getDeclaredField("b");
31
32         } catch (NoSuchFieldException e) {
33             // TODO Auto-generated catch block
34             e.printStackTrace();
35             System.out.println("getDeclaredField "+e.getMessage());
36         } catch (SecurityException e) {
37             // TODO Auto-generated catch block
38             e.printStackTrace();
39             System.out.println("getDeclaredField "+e.getMessage());
40         }
41     }
42 }
```

```

39     }
40
41     try {
42         Field field = cls.getField("b");
43
44     } catch (NoSuchFieldException e) {
45         // TODO Auto-generated catch block
46         e.printStackTrace();
47         System.out.println("getField "+e.getMessage());
48     } catch (SecurityException e) {
49         // TODO Auto-generated catch block
50         e.printStackTrace();
51         System.out.println("getField "+e.getMessage());
52     }
53
54
55
56
57
58     Field[] filed1 = cls.getDeclaredFields();
59
60     for ( Field f : filed1 ) {
61         System.out.println("Declared Field :"+f.getName());
62     }
63
64     Field[] filed2 = cls.getFields();
65
66     for ( Field f : filed2 ) {
67         System.out.println("Field :"+f.getName());
68     }
69
70 }
71
72 }

```

代码打印结果：

```

1 java.lang.NoSuchFieldException: b
2     at java.lang.Class.getDeclaredField(Unknown Source)
3     at com.frank.test.FieldTest.main(FieldTest.java:13)
4 java.lang.NoSuchFieldException: b.getDeclaredField b
5
6     at java.lang.Class.getField(Unknown Source)
7     at com.frank.test.FieldTest.main(FieldTest.java:26)
8 getField b
9
10 Declared Field :c
11 Declared Field :d
12 Declared Field :e
13
14 Field :a

```

大家细细体会一下，不过需要注意的是 `getDeclaredFiled()` 方法可以获取 `private`、`protected`、`public` 和 `default` 属性，但是它获取不到从父类继承下来的属性。

获取 Method

类或者接口中的方法对应到 Class 就是 Method。

相应的 API 如下：

```

1 public Method getDeclaredMethod(String name, Class<?>... parameterTypes)
2
3 public Method getMethod(String name, Class<?>... parameterTypes)
4
5 public Method[] getDeclaredMethods() throws SecurityException
6

```



```
7
8 public Method getMethod(String name, Class<?>... parameterTypes)
```

因为跟 Field 类似，所以不做过多的讲解。parameterTypes 是方法对应的参数。

获取 Constructor

Java 反射把构造器从方法中单独拎出来了，用 Constructor 表示。

```
1 public Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)
2
3 public Constructor<T> getConstructor(Class<?>... parameterTypes)
4
5 public Constructor<?>[] getDeclaredConstructors() throws SecurityException
6
7 public Constructor<?>[] getConstructors() throws SecurityException
```

仍然以前面的 Father 和 Son 两个类为例。

```
1 public class Farther {
2
3     public int a;
4
5     private int b;
6
7     public Farther() {
8         super();
9         // TODO Auto-generated constructor stub
10    }
11
12
13 }
14
15 public class Son extends Farther {
16     int c;
17
18     private String d;
19
20     protected float e;
21
22
23
24     private Son() {
25         super();
26         // TODO Auto-generated constructor stub
27     }
28
29
30
31     public Son(int c, String d) {
32         super();
33         this.c = c;
34         this.d = d;
35     }
36
37 }
38
39 public class ConstructorTest {
40
41     public static void main(String[] args) {
42         // TODO Auto-generated method stub
43
44         Class clz = Son.class;
45
46         Constructor[] constructors = clz.getConstructors();
47
48         for (Constructor c : constructors) {
49             System.out.println("getConstructor:" + c.toString());
```

```

50     }
51
52     constructors = clz.getDeclaredConstructors();
53
54     for ( Constructor c : constructors ) {
55         System.out.println("getDeclaredConstructors:"+c.toString());
56     }
57
58 }
59
60 }
61

```

测试程序代码的打印结果如下：

```

1  getConstructor:public com.frank.test.Son(int, java.lang.String)
2
3  getDeclaredConstructors:private com.frank.test.Son()
4  getDeclaredConstructors:public com.frank.test.Son(int, java.lang.String)

```

因为，Constructor 不能从父类继承，所以就没有办法通过 getConstructor() 获取到父类的 Constructor。

我们获取到了 Field、Method、Constructor,但这一是终点，相反，这正是反射机制中开始的地方，我们运用反射的目的就是为了获取和操控 Class 对象中的这些成员。

Field 的操控

我们在一个类中定义字段时，通常是这样。

```

1  public class Son extends Farther {
2      int c;
3
4      private String d;
5
6      protected float e;
7
8      Car car;
9
10 }

```

像 c、d、e、car 这些变量都是属性，在反射机制中映射到 Class 对象中都是 Field，很显然，它们也有对应的类别。

它们要么是 8 种基础类型 int、long、float、double、boolean、char、byte 和 short。或者是引用，所有的引用都是 Object 的后代。

Field 类型的获取

获取 Field 的类型，通过 2 个方法：

```

1  public Type getGenericType() {}
2
3  public Class<?> getType() {}

```

注意，两者返回的类型不一样，getGenericType() 方法能够获取到泛型类型。大家可以看下面的代码进行理解：

```

1  public class Son extends Farther {

```

```

2      int c;
3
4      private String d;
5
6      protected float e;
7
8      public List<Car> cars;
9
10     public HashMap<Integer,String> map;
11
12     private Son() {
13         super();
14         // TODO Auto-generated constructor stub
15     }
16
17
18
19     public Son(int c, String d) {
20         super();
21         this.c = c;
22         this.d = d;
23     }
24 }
25
26
27 public class FieldTest {
28
29     public static void main(String[] args) {
30         // TODO Auto-generated method stub
31
32         Class cls = Son.class;
33
34
35         Field[] filed2 = cls.getFields();
36
37         for ( Field f : filed2 ) {
38             System.out.println("Field :"+f.getName());
39             System.out.println("Field type:"+f.getType());
40             System.out.println("Field generic type:"+f.getGenericType());
41             System.out.println("-----");
42         }
43
44     }
45
46 }

```

打印结果：

```

1  Field :cars
2  Field type:interface java.util.List
3  Field generic type:java.util.List<com.frank.test.Car>
4  -----
5  Field :map
6  Field type:class java.util.HashMap
7  Field generic type:java.util.HashMap<java.lang.Integer, java.lang.String>
8  -----
9  Field :a
10 Field type:int
11 Field generic type:int
12 -----

```

可以看到 `getGenericType()` 确实把泛型都打印出来了，它比 `getType()` 返回的内容更详细。

Field 修饰符的获取

同 `Class` 一样，`Field` 也有很多修饰符。通过 `getModifiers()` 方法就可以轻松获取。

```

1  public int getModifiers() {}

```

这个与前面 Class 获取修饰符一致，所以不需要再讲，不清楚的同学翻看前面的内容就好了。

Field 内容的读取与赋值

这个应该是反射机制中对于 Field 最主要的目的了。

Field 这个类定义了一系列的 get 方法来获取不同类型的值。

```
1
2 public Object get(Object obj);
3
4 public int getInt(Object obj);
5
6 public long getLong(Object obj)
7     throws IllegalArgumentException, IllegalAccessException;
8
9 public float getFloat(Object obj)
10    throws IllegalArgumentException, IllegalAccessException;
11
12 public short getShort(Object obj)
13    throws IllegalArgumentException, IllegalAccessException;
14
15 public double getDouble(Object obj)
16    throws IllegalArgumentException, IllegalAccessException;
17
18 public char getChar(Object obj)
19    throws IllegalArgumentException, IllegalAccessException;
20
21 public byte getByte(Object obj)
22    throws IllegalArgumentException, IllegalAccessException;
23
24 public boolean getBoolean(Object obj)
25    throws IllegalArgumentException, IllegalAccessException;
```

Field 又定义了一系列的 set 方法用来对其自身进行赋值。

```
1 public void set(Object obj, Object value);
2
3 public void getInt(Object obj, int value);
4
5 public void setLong(Object obj, long value)
6     throws IllegalArgumentException, IllegalAccessException;
7
8 public void setFloat(Object obj, float value)
9     throws IllegalArgumentException, IllegalAccessException;
10
11 public void setShort(Object obj, short value)
12     throws IllegalArgumentException, IllegalAccessException;
13
14 public void setDouble(Object obj, double value)
15     throws IllegalArgumentException, IllegalAccessException;
16
17 public void setChar(Object obj, char value)
18     throws IllegalArgumentException, IllegalAccessException;
19
20 public void setByte(Object obj, byte b)
21     throws IllegalArgumentException, IllegalAccessException;
22
23 public void setBoolean(Object obj, boolean b)
24     throws IllegalArgumentException, IllegalAccessException;
```

可能有同学会对方法中出现的 Object 参数有疑问，它其实是类的实例引用，这里涉及一个细节。

Class 本身不对成员进行储存，它只提供检索，所以需要 Field、Method、Constructor 对象来

承载这些成员，所以，针对成员的操作时，一般需要为成员指定类的实例引用。如果难于理解的话，可以这样理解，班级这个概念是一个类，一个班级有几十名学生，现在有A、B、C 3个班级，将所有班级的学生抽出来集合到一个场地来考试，但是学生在试卷上写上自己名字的时候，还要指定自己的班级，这里涉及到的 **Object** 其实就是类似的作用，表示这个成员是具体属于哪个 **Object**。这个是为了精确定位。

下面用代码来说明：

```
1 A testa = new A();
2 testa.a = 10;
3
4 System.out.println("testa.a = "+testa.a);
5
6 Class c = A.class;
7
8 try {
9     Field fielda = c.getField("a");
10
11     int ra = fielda.getInt(testa);
12
13     System.out.println("reflection testa.a = "+ra);
14
15     fielda.setInt(testa, 15);
16
17     System.out.println("testa.a = "+testa.a);
18
19 } catch (NoSuchFieldException e) {
20     // TODO Auto-generated catch block
21     e.printStackTrace();
22 } catch (SecurityException e) {
23     // TODO Auto-generated catch block
24     e.printStackTrace();
25 } catch (IllegalArgumentException e) {
26     // TODO Auto-generated catch block
27     e.printStackTrace();
28 } catch (IllegalAccessException e) {
29     // TODO Auto-generated catch block
30     e.printStackTrace();
31 }
```

打印结果如下：

```
1 testa.a = 10
2 reflection testa.a = 10
3 testa.a = 15
```

我们再来看看 **Field** 被 **private** 修饰的情况

```
1 public class A {
2
3     public int a;
4
5     private int b;
6
7     public int getB() {
8         return b;
9     }
10
11     public void setB(int b) {
12         this.b = b;
13     }
14
15 }
```

再编写测试代码

```
1 A testa = new A();
2 testa.setB(3);
3
4 System.out.println("testa.b = "+testa.getB());
5
6 Class c = A.class;
7
8 try {
9     Field fieldb = c.getDeclaredField("b");
10    int rb = fieldb.getInt(testa);
11
12    System.out.println("reflection testa.b = "+rb);
13
14    fieldb.setInt(testa, 20);
15
16    System.out.println("testa.b = "+testa.getB());
17
18 } catch (NoSuchFieldException e) {
19     // TODO Auto-generated catch block
20     e.printStackTrace();
21 } catch (SecurityException e) {
22     // TODO Auto-generated catch block
23     e.printStackTrace();
24 } catch (IllegalArgumentException e) {
25     // TODO Auto-generated catch block
26     e.printStackTrace();
27 } catch (IllegalAccessException e) {
28     // TODO Auto-generated catch block
29     e.printStackTrace();
30 }
```

打印的结果如下：

```
1 testa.b = 3
2 java.lang.IllegalAccessException: Class com.frank.test.FieldTest can not access a member
3   at sun.reflect.Reflection.ensureMemberAccess(Unknown Source)
4   at java.lang.reflect.AccessibleObject.slowCheckMemberAccess(Unknown Source)
5   at java.lang.reflect.AccessibleObject.checkAccess(Unknown Source)
6   at java.lang.reflect.Field.getInt(Unknown Source)
7   at com.frank.test.FieldTest.main(FieldTest.java:20)
```

抛异常了。这是因为在反射中访问了 `private` 修饰的成员，如果要消除异常的话，需要添加一句代码。

```
1 fieldb.setAccessible(true);
```

再看打印结果

```
1 testa.b = 3
2 reflection testa.b = 3
3 testa.b = 20
```

Method 的操控

Method 对应普通类的方法。

我们看看一般普通类的方法的构成。

```
1
2 public int add(int a, int b);
```


方法由下面几个要素构成：

- 方法名
- 方法参数
- 方法返回值
- 方法的修饰符
- 方法可能会抛出的异常

很显然，反射中 Method 提供了相应的 API 来提取这些元素。

Method 获取方法名

通过 getName() 这个方法就好了。

以前面的 Car 类作为测试对象。

```
1 public class MethodTest {
2
3     public static void main(String[] args) {
4         // TODO Auto-generated method stub
5         Car car = new Car();
6
7         Class clz = car.getClass();
8
9         Method methods[] = clz.getDeclaredMethods();
10
11         for ( Method m : methods ) {
12             System.out.println("method name:"+m.getName());
13         }
14     }
15
16 }
```

打印结果如下：

```
1 method name:toString
2 method name:drive
```

Method 获取方法参数

涉及到的 API 如下：

```
1 public Parameter[] getParameters() {}
```

返回的是一个 Parameter 数组，在反射中 Parameter 对象就是用来映射方法中的参数。经常使用的方法有：

Parameter.java

```
1 // 获取参数名字
2 public String getName() {}
3
4 // 获取参数类型
5 public Class<?> getType() {}
6
7 // 获取参数的修饰符
8 public int getModifiers() {}
```

当然，有时候我们不需要参数的名字，只要参数的类型就好了，通过 Method 中下面的方法获取。

Method.java

```
1 // 获取所有的参数类型
2 public Class<?>[] getParameterTypes() {}
3
4 // 获取所有的参数类型，包括泛型
5 public Type[] getGenericParameterTypes() {}
```

下面，同样进行测试。

```
1 public class Car {
2
3     private String mBand;
4
5     private Color mColor;
6
7     public enum Color {
8         RED,
9         WHITE,
10        BLACK,
11        BLUE,
12        YELLOR
13    }
14
15
16
17    public Car() {
18        super();
19        // TODO Auto-generated constructor stub
20    }
21
22
23    public Car(String mBand) {
24        this.mBand = mBand;
25    }
26
27
28    public void drive() {
29        System.out.println("di di di, 开车了!");
30    }
31
32    @Override
33    public String toString() {
34        return "Car [mBand=" + mBand + ", mColor=" + mColor + "]";
35    }
36
37    public void test(String[] paraa, List<String> b, HashMap<Integer, Son> maps) {}
38
39
40 }
41
42 public class MethodTest {
43
44    public static void main(String[] args) {
45        // TODO Auto-generated method stub
46        Car car = new Car();
47
48        Class clz = car.getClass();
49
50        Method methods[] = clz.getDeclaredMethods();
51
52
53
54        for ( Method m : methods ) {
55            System.out.println("method name:" + m.getName());
56
57            Parameter[] paras = m.getParameters();
58
59            for ( Parameter p : paras ) {
```

```

60         System.out.println(" parameter :"+p.getName()+" "+p.getType().getName());
61     }
62
63     Class[] pTypes = m.getParameterTypes();
64
65     System.out.println("method para types:");
66     for ( Class type : pTypes ) {
67         System.out.print(" "+ type.getName());
68     }
69     System.out.println();
70
71     Type[] gTypes = m.getGenericParameterTypes();
72     System.out.println("method para generic types:");
73     for ( Type type : gTypes ) {
74         System.out.print(" "+ type.getTypeName());
75     }
76     System.out.println();
77     System.out.println("=====");
78
79     }
80 }
81
82 }

```

打印结果如下：

```

1  method name:toString
2  method para types:
3
4  method para generic types:
5
6  =====
7  method name:test
8  parameter :arg0 [Ljava.lang.String;
9  parameter :arg1 java.util.List
10 parameter :arg2 java.util.HashMap
11 method para types:
12 [Ljava.lang.String; java.util.List java.util.HashMap
13 method para generic types:
14 java.lang.String[] java.util.List<java.lang.String> java.util.HashMap<java.lang.Integer,java.lang.String>
15 =====
16 method name:drive
17 method para types:
18
19 method para generic types:
20
21 =====

```

Method 获取返回值类型

```

1  // 获取返回值类型
2  public Class<?> getReturnType() {}
3
4
5  // 获取返回值类型包括泛型
6  public Type getGenericReturnType() {}

```

Method 获取修饰符

```

1  public int getModifiers() {}

```

这部分内容前面已经讲过。

Method 获取异常类型

```
1 public Class<?>[] getExceptionTypes() {}
2
3 public Type[] getGenericExceptionTypes() {}
```

Method 方法的执行

这个应该是整个反射机制的核心内容了，很多时候运用反射目的其实就是为了以常规手段执行 Method。

```
1 public Object invoke(Object obj, Object... args) {}
```

Method 调用 invoke() 的时候，存在许多细节：

- invoke() 方法中第一个参数 Object 实质上是 Method 所依附的 Class 对应的类的实例，如果这个方法是一个静态方法，那么 obj 为 null，后面的可变参数 Object 对应的自然就是参数。
- invoke() 返回的对象是 Object，所以实际上执行的时候要进行强制转换。
- 在对 Method 调用 invoke() 的时候，如果方法本身会抛出异常，那么这个异常就会经过包装，由 Method 统一抛出 InvocationTargetException。而通过 InvocationTargetException.getCause() 可以获取真正的异常。

下面同样通过例子来说明,我们新建立一个类，要添加一个 static 修饰的静态方法，一个普通的方法和一个会抛出异常的方法。

```
1 public class TestMethod {
2
3     public static void testStatic () {
4         System.out.println("test static");
5     }
6
7     private int add (int a,int b ) {
8         return a + b;
9     }
10
11     public void testException () throws IllegalAccessException {
12         throw new IllegalAccessException("You have some problem.");
13     }
14
15 }
```

我们编写测试代码：

```
1 public class InvokeTest {
2
3     public static void main(String[] args) {
4         // TODO Auto-generated method stub
5         Class testCls = TestMethod.class;
6
7         try {
8             Method mStatic = testCls.getMethod("testStatic",null);
9             // 测试静态方法
10            mStatic.invoke(null, null);
11        } catch (NoSuchMethodException e) {
12            // TODO Auto-generated catch block
13            e.printStackTrace();
14        } catch (SecurityException e) {
15            // TODO Auto-generated catch block
16            e.printStackTrace();
17        }
18    }
19 }
```

```

17     } catch (IllegalAccessException e) {
18         // TODO Auto-generated catch block
19         e.printStackTrace();
20     } catch (IllegalArgumentException e) {
21         // TODO Auto-generated catch block
22         e.printStackTrace();
23     } catch (InvocationTargetException e) {
24         // TODO Auto-generated catch block
25         e.printStackTrace();
26     }
27
28     TestMethod t = new TestMethod();
29
30     try {
31         Method mAdd = testCls.getDeclaredMethod("add", int.class, int.class);
32         // 通过这句代码才能访问 private 修饰的 Method
33         mAdd.setAccessible(true);
34         int result = (int) mAdd.invoke(t, 1, 2);
35         System.out.println("add method result: "+result);
36     } catch (NoSuchMethodException e) {
37         // TODO Auto-generated catch block
38         e.printStackTrace();
39     } catch (SecurityException e) {
40         // TODO Auto-generated catch block
41         e.printStackTrace();
42     } catch (IllegalAccessException e) {
43         // TODO Auto-generated catch block
44         e.printStackTrace();
45     } catch (IllegalArgumentException e) {
46         // TODO Auto-generated catch block
47         e.printStackTrace();
48     } catch (InvocationTargetException e) {
49         // TODO Auto-generated catch block
50         e.printStackTrace();
51     }
52
53     try {
54         Method testExcep = testCls.getMethod("testException", null);
55
56         try {
57             testExcep.invoke(t, null);
58         } catch (IllegalAccessException e) {
59             // TODO Auto-generated catch block
60             e.printStackTrace();
61         } catch (IllegalArgumentException e) {
62             // TODO Auto-generated catch block
63             e.printStackTrace();
64         } catch (InvocationTargetException e) {
65             // TODO Auto-generated catch block
66             //e.printStackTrace();
67
68             // 通过 InvocationTargetException.getCause() 获取被包装的异常
69             System.out.println("testException occur some error, Error type is :"+e.
70             System.out.println("Error message is :"+e.getCause().getMessage());
71         }
72
73
74     } catch (NoSuchMethodException e) {
75         // TODO Auto-generated catch block
76         e.printStackTrace();
77     } catch (SecurityException e) {
78         // TODO Auto-generated catch block
79         e.printStackTrace();
80     }
81 }
82
83 }

```

打印结果如下：

```
1 test static
2 add method result:3
3 testException occur some error,Error type is :java.lang.IllegalAccessException
4 Error message is :You have some problem.
```

Constructor 的操控

在平常开发的时候，构造器也称构造方法，但是在反射机制中却把它与 Method 分离开来，单独用 Constructor 这个类表示。

Constructor 同 Method 差不多，但是它特别的地方在于，它能够创建一个对象。

在 Java 反射机制中有两种方法可以用来创建类的对象实例：Class.newInstance() 和 Constructor.newInstance()。官方文档建议开发者使用后面这种方法，下面是原因。

- Class.newInstance() 只能调用无参的构造方法，而 Constructor.newInstance() 则可以调用任意的构造方法。
- Class.newInstance() 通过构造方法直接抛出异常，而 Constructor.newInstance() 会把抛出来的异常包装到 InvocationTargetException 里面去，这个和 Method 行为一致。
- Class.newInstance() 要求构造方法能够被访问，而 Constructor.newInstance() 却能够访问 private 修饰的构造器。

还是通过代码来验证。

```
1 public class TestConstructor {
2
3     private String self;
4
5     public TestConstructor() {
6         self = " Frank ";
7     }
8
9     public TestConstructor(String self) {
10        this.self = self;
11    }
12
13    @Override
14    public String toString() {
15        return "TestConstructor [self=" + self + "]";
16    }
17
18
19 }
```

上面的类中有 2 个构造方法，一个无参，一个有参数。编写测试代码：

```
1 public class NewInstanceTest {
2
3     public static void main(String[] args) {
4         // TODO Auto-generated method stub
5
6         Class clz = TestConstructor.class;
7
8         try {
9             TestConstructor test1 = (TestConstructor) clz.newInstance();
10
11             System.out.println(test1.toString());
12         } catch (InstantiationException e) {
13             // TODO Auto-generated catch block
14             e.printStackTrace();
15         }
16     }
17 }
```



```

15         } catch (IllegalAccessException e) {
16             // TODO Auto-generated catch block
17             e.printStackTrace();
18         }
19
20         try {
21             Constructor con = clz.getConstructor(String.class);
22
23             TestConstructor test2 = (TestConstructor) con.newInstance("Zhao");
24
25             System.out.println(test2.toString());
26
27         } catch (NoSuchMethodException e) {
28             // TODO Auto-generated catch block
29             e.printStackTrace();
30         } catch (SecurityException e) {
31             // TODO Auto-generated catch block
32             e.printStackTrace();
33         } catch (InstantiationException e) {
34             // TODO Auto-generated catch block
35             e.printStackTrace();
36         } catch (IllegalAccessException e) {
37             // TODO Auto-generated catch block
38             e.printStackTrace();
39         } catch (IllegalArgumentException e) {
40             // TODO Auto-generated catch block
41             e.printStackTrace();
42         } catch (InvocationTargetException e) {
43             // TODO Auto-generated catch block
44             e.printStackTrace();
45         }
46     }
47 }
48
49 }

```

分别用 `Class.newInstance()` 和 `Constructor.newInstance()` 方法来创建类的实例，打印结果如下：

```

1 TestConstructor [self= Frank ]
2 TestConstructor [self=Zhao]

```

可以看到通过 `Class.newInstance()` 方法调用的构造方法确实是无参的那个。

现在，我们学习了 `Class` 对象的获取，也能够获取它内部成员 `Filed`、`Method` 和 `Constructor` 并且能够操作它们。在这个基础上，我们已经能够应付普通的反射开发了。

但是，Java 反射机制还另外细分了两个概念：数组和枚举。

反射中的数组

数组本质上是一个 `Class`，而在 `Class` 中存在一个方法用来识别它是否为一个数组。

`Class.java`

```

1 public native boolean isArray();

```

为了便于测试，我们创建一个新的类

```

1 public class Shuzu {
2
3     private int[] array;
4
5     private Car[] cars;

```

其中有一个 int 型的数组属性，它的名字叫做 array。还有一个 cars 数组，它的类型是 Car，是之前定义好的类。当然，array 和 cars 是 Shuzu 这个类的 Field，对于 Field 的角度来说，它是数组类型，我们可以这样理解数组可以同 int、char 这些基本类型一样成为一个 Field 的类别。

我们可能通过一系列的 API 来获取它的具体信息，刚刚有提到它本质上还是一个 Class 而已。

```
1 getName();
2
3 getComponentType();
```

第二个方法是获取数组的里面的元素的类型，比如 int[] 数组的 componentType 自然就是 int。

按照惯例，写代码验证。

```
1 public class ArraysTest {
2
3     public static void main(String[] args) {
4         Class clz = Shuzu.class;
5
6         Field[] fields = clz.getDeclaredFields();
7
8         for (Field f : fields) {
9             // 获取 Field 的类型
10            Class c = f.getType();
11            // 判断这个类型是不是数组类型
12            if (c.isArray()) {
13                System.out.println("Type is " + c.getName());
14                System.out.println("Component type is : " + c.getComponentType());
15            }
16        }
17    }
18
19 }
```

打印结果如下：

```
1 Type is [I
2 ComponentType type is :int
3 Type is [Lcom.frank.test.Car;
4 ComponentType type is :class com.frank.test.Car
```

反射中动态创建数组

反射创建数组是通过 Array.newInstance() 这个方法。

Array.java

```
1 public static Object newInstance(Class<?> componentType, int... dimensions)
2     throws IllegalArgumentException, NegativeArraySizeException {}
```

第一个参数指定的是数组内的元素类型，后面的是可变参数，表示的是相应维度的数组长度限制。

比如，我要创建一个 int[2][3] 的数组。

```
1 Array.newInstance(int.class, 2, 3);
```

Array 的读取与赋值

首先，对于 Array 整体的读取与赋值，把它作为一个普通的 Field，根据 Class 中相应获取和设置就

好了。调用的是 Field 中对应的方法。

```
1 public void set(Object obj,
2                 Object value)
3     throws IllegalArgumentException,
4         IllegalAccessException;
5
6
7 public Object get(Object obj)
8     throws IllegalArgumentException,
9         IllegalAccessException;
```

还需要处理的情况是对于数组中指定位置的元素进行读取与赋值，这要涉及到 Array 提供的一系列 setXXX() 和 getXXX() 方法。因为和之前 Field 相应的 set、get 方法类似，所以我在下面只摘抄典型的几种，大家很容易知晓其它类型的怎么操作。

```
1 public static void set(Object array,
2                       int index,
3                       Object value)
4     throws IllegalArgumentException,
5         ArrayIndexOutOfBoundsException;
6
7
8 public static void setBoolean(Object array,
9                              int index,
10                             boolean z)
11     throws IllegalArgumentException,
12         ArrayIndexOutOfBoundsException;
13
14
15
16 public static Object get(Object array,
17                          int index)
18     throws IllegalArgumentException,
19         ArrayIndexOutOfBoundsException;
20
21
22 public static short getShort(Object array,
23                              int index)
24     throws IllegalArgumentException,
25         ArrayIndexOutOfBoundsException;
26
```

进行代码测试：

```
1 public class ArraysTest {
2
3     public static void main(String[] args) {
4         Class clz = Shuzu.class;
5
6         try {
7             Shuzu shu = (Shuzu) clz.newInstance();
8
9             Field arrayF = clz.getDeclaredField("array");
10            arrayF.setAccessible(true);
11
12            Object o = Array.newInstance(int.class, 3);
13            Array.set(o, 0, 1);
14            Array.set(o, 1, 3);
15            Array.set(o, 2, 3);
16
17            arrayF.set(shu, o);
18
19            int[] array = shu.getArray();
20
21            for (int i = 0; i < array.length; i++) {
22                System.out.println("array index "+i+" value:"+array[i]);
23            }
24        } catch (Exception e) {
25            e.printStackTrace();
26        }
27    }
28 }
```

```

23         }
24
25     } catch (InstantiationException e) {
26         // TODO Auto-generated catch block
27         e.printStackTrace();
28     } catch (IllegalAccessException e) {
29         // TODO Auto-generated catch block
30         e.printStackTrace();
31     } catch (NoSuchFieldException e) {
32         // TODO Auto-generated catch block
33         e.printStackTrace();
34     } catch (SecurityException e) {
35         // TODO Auto-generated catch block
36         e.printStackTrace();
37     }
38
39 }
40
41 }
42 }
43

```

打印结果如下：

```

1 array index 0 value:1
2 array index 1 value:3
3 array index 2 value:3

```

反射中的枚举 Enum

同数组一样，枚举本质上也是一个 Class 而已，但反射中还是把它单独提出来了。

我们来看一般程序开发中枚举的表现形式。

```

1 public enum State {
2     IDLE,
3     DRIVING,
4     STOPPING,
5
6     test();
7
8     int test1() {
9         return 0;
10    }
11 }

```

枚举真的跟类很相似，有修饰符、有方法、有属性字段甚至可以有构造方法。

在 Java 反射中，可以把枚举看成一般的 Class，但是反射机制也提供了 3 个特别的 API 用于操控枚举。

```

1 // 用来判定 Class 对象是不是枚举类型
2 Class.isEnum()
3
4 // 获取所有的枚举常量
5 Class.getEnumConstants()
6
7
8 // 判断一个 Field 是不是枚举常量
9 java.lang.reflect.Field.isEnumConstant()

```

枚举的获取与设定

因为等同于 Class，所以枚举的获取与设定就可以通过 Field 中的 get() 和 set() 方法。

需要注意的是，如果要获取枚举里面的 Field、Method、Constructor 可以调用 Class 的通用 API。

用例子来加深理解吧。

```
1 public enum State {
2     IDLE,
3     DRIVING,
4     STOPPING,
5
6     test();
7
8     int test1() {
9         return 0;
10    }
11
12 }
13
14 public class Meiju {
15
16     private State state = State.DRIVING;
17
18     public State getState() {
19         return state;
20     }
21
22     public void setState(State state) {
23         this.state = state;
24     }
25 }
26
27 public static void main(String[] args) {
28
29     Class clz = State.class;
30
31     if ( clz.isEnum() ){
32         System.out.println(clz.getName()+" is Enum");
33
34         System.out.println(Arrays.asList(clz.getEnumConstants()));
35         // 获取枚举中所有的 Field
36         Field[] fs = clz.getDeclaredFields();
37
38         for ( Field f : fs ) {
39             if ( f.isEnumConstant() ){
40                 System.out.println(f.getName()+" is EnumConstant");
41             }else {
42                 System.out.println(f.getName()+" is not EnumConstant");
43             }
44         }
45
46         Class cMeiju = Meiju.class;
47         Meiju meiju = new Meiju();
48
49         try {
50             Field f = cMeiju.getDeclaredField("state");
51             f.setAccessible(true);
52
53
54             try {
55                 State state = (State) f.get(meiju);
56
57                 System.out.println("State current is "+state);
58
59                 f.set(meiju, State.STOPPING);
60
61 }
```

```
62         System.out.println("State current is "+meiju.getState());
63
64     } catch (IllegalArgumentException e) {
65         // TODO Auto-generated catch block
66         e.printStackTrace();
67     } catch (IllegalAccessException e) {
68         // TODO Auto-generated catch block
69         e.printStackTrace();
70     }
71
72     } catch (NoSuchFieldException e) {
73         // TODO Auto-generated catch block
74         e.printStackTrace();
75     } catch (SecurityException e) {
76         // TODO Auto-generated catch block
77         e.printStackTrace();
78     }
79 }
80
81 }
82
83 }
```

打印结果如下：

```
1  com.frank.test.State is Enum
2  [IDLE, DRIVING, STOPPING, test]
3  IDLE is EnumConstant
4  DRIVING is EnumConstant
5  STOPPING is EnumConstant
6  test is EnumConstant
7  ENUM$VALUES is not EnumConstant
8  State current is DRIVING
9  State current is STOPPING
```

到这里，反射的所有知识基本上讲完了。下面进行模拟实战。

反射与自动驾驶

文章开头，我用自动驾驶的技术来比喻反射，实际上的目的是为了给初学者一个大体的印象和一个模糊的轮廓，实际上反射不是自动驾驶，它是什么取决于你自己对它的理解。

下段代码的目标是为了对比，先定义一个类 AutoDrive，这个类有一系列的属性，然后有一系列的方法，先用普通编码的方式来创建这个类的对象，调用它的方法。然后用反射的机制模拟自动驾驶。

汽车开动的步骤，以手动档为例。

1. 空档发动。
2. 打左转向灯。
3. 踩离合挂一档。
4. 起步松手铰。

现在代码模拟

```
1  public class AutoDrive {
2
3      public enum Color {
4          WHITE,
5          REN,
```



```

6      BLUE
7    }
8    private String vendor;
9
10   private Color color;
11
12   public AutoDrive(String vendor, Color color) {
13       super();
14       this.vendor = vendor;
15       this.color = color;
16   }
17
18   public AutoDrive() {
19       vendor = "Nissan";
20       color = Color.WHITE;
21   }
22
23   public void drive() {
24
25       boot();
26
27       turnOnLeftLight();
28
29       cailiheguayidang();
30
31       songshousha();
32
33
34       tips();
35
36   }
37
38   private void tips() {
39       System.out.println("您正在驾驶 "+color+" "+vendor+" 汽车，小心行驶。");
40   }
41
42   private void songshousha() {
43       // TODO Auto-generated method stub
44       System.out.println("起步松手刹。");
45   }
46
47   private void cailiheguayidang() {
48       // TODO Auto-generated method stub
49       System.out.println("踩离合器，挂一档");
50   }
51
52   private void turnOnLeftLight() {
53       // TODO Auto-generated method stub
54       System.out.println("打左转向灯");
55   }
56
57   private void boot() {
58       // TODO Auto-generated method stub
59       System.out.println("空档发动汽车");
60
61   }
62
63 }
64

```

我们只要创建一个 AutoDrive 的对象，调用它的 drive() 方法就好了。

```

1 public class DriveTest {
2
3
4     public static void main(String[] args) {
5         AutoDrive car = new AutoDrive();
6
7         car.drive();
8     }
9

```

```
9
10 }
```

结果如下：

```
1 空档发动汽车
2 打左向灯
3 踩离合器，挂一档
4 起步松手刹。
5 您正在驾驶 WHITE Nissan 汽车，小心行驶。
```

我们现在要使用自动驾驶技术，具体到代码就是反射，因为非常规嘛。

```
1 public class DriveTest {
2
3
4     public static void main(String[] args) {
5         AutoDrive car = new AutoDrive();
6
7         car.drive();
8
9         Class cls = AutoDrive.class;
10        try {
11            Constructor cons = cls.getConstructor(String.class, AutoDrive.Color.class);
12
13            // 利用反射技术创建 AutoDrive 对象
14            AutoDrive autoDrive = (AutoDrive) cons.newInstance("Tesla", AutoDrive.Color
15
16            // 获取能够驱动汽车的 drive 方法
17            Method method = cls.getMethod("drive");
18
19            System.out.println("=====\n自动驾驶马上开始\n=====");
20            // 通过反射调用 Method 方法，最终车子跑起来
21            method.invoke(autoDrive, null);
22
23
24        } catch (NoSuchMethodException e) {
25            // TODO Auto-generated catch block
26            e.printStackTrace();
27        } catch (SecurityException e) {
28            // TODO Auto-generated catch block
29            e.printStackTrace();
30        } catch (InstantiationException e) {
31            // TODO Auto-generated catch block
32            e.printStackTrace();
33        } catch (IllegalAccessException e) {
34            // TODO Auto-generated catch block
35            e.printStackTrace();
36        } catch (IllegalArgumentException e) {
37            // TODO Auto-generated catch block
38            e.printStackTrace();
39        } catch (InvocationTargetException e) {
40            // TODO Auto-generated catch block
41            e.printStackTrace();
42        }
43    }
44
45 }
```

最后，打印结果：

```
1 空档发动汽车
2 打左向灯
3 踩离合器，挂一档
4 起步松手刹。
5 您正在驾驶 WHITE Nissan 汽车，小心行驶。
6 =====
```

```
7  自动驾驶马上开始
8  =====
9  空档发动汽车
10 打左向灯
11 踩离合器，挂一档
12 起步松手刹。
13 您正在驾驶 RED Tesla 汽车，小心行驶。
```

总结

1. Java 中的反射是非常规编码方式。
2. Java 反射机制的操作入口是获取 Class 文件。有 Class.forName()、.class 和 Object.getClass() 3 种。
3. 获取 Class 对象后还不够，需要获取它的 Members，包含 Field、Method、Constructor。
4. Field 操作主要涉及到类别的获取，及数值的读取与赋值。
5. Method 算是反射机制最核心的内容，通常的反射都是为了调用某个 Method 的 invoke() 方法。
6. 通过 Class.newInstance() 和 Constructor.newInstance() 都可以创建类的对象实例，但推荐后者。因为它适应于任何构造方法，而前者只会调用可见的无参数的构造方法。
7. 数组和枚举可以被看成普通的 Class 对待。

最后，需要注意的是。

反射是非常规开发手段，它会抛弃 Java 虚拟机的很多优化，所以同样功能的代码，反射要比正常方式要慢，所以考虑到采用反射时，要考虑它的时间成本。另外，就如无人驾驶之于汽车一样，用着很爽的同时，其实风险未知。

洋洋洒洒已经 2000 多行了，本来还有东西没有写完，因为这一块内容实在太多了。只能另外写一篇文章了，讲得是反射中一些常见的细节和容易出错的地方。不过，这篇文章的内容已经足够应付平常开发中所需要的反射知识了。

只是，在日常开发中，利用反射飙车的时候，记得提醒自己一句：老哥，稳住。

关于反射更多细节，可以阅读这篇博文 [《反射进阶，编写反射代码值得注意的诸多细节》](#)

顶

29

踩

1

- [▲ 上一篇](#) 轻松学，Java 中的代理模式及动态代理
- [▼ 下一篇](#) 轻松学，浅析依赖倒置（DIP）、控制反转(IOC)和依赖注入(DI)

相关文章推荐

- Java 标准日志工具 Log4j 的使用（附源代码）
- 7 款基于 HTML5 Canvas 的超炫 3D 动画效果
- AJAX学习笔记05
- dedecms 的这个dede:arclist里怎么调用全局变量...
- AJAX学习笔记01
- jdbcType与javaType的对应关系
- 数据库类型与JDBC TYPE 和Java类型对应关系
- 使用idea+springboot+Mybatis搭建web项目
- 类名.class 类名.this 详解
- 类名.this 的使用

猜你在找

- 【直播】机器学习&数据挖掘7周实训--韦玮
- 【直播】3小时掌握Docker最佳实战-徐西宁
- 【直播】计算机视觉原理及实战--屈教授
- 【直播】机器学习之矩阵--黄博士
- 【直播】机器学习之凸优化--马博士
- 【套餐】系统集成项目管理工程师顺利通关--徐朋
- 【套餐】机器学习系列套餐（算法+实战）--唐宇迪
- 【套餐】微信订阅号+服务号Java版 v2.0--翟东平
- 【套餐】微信订阅号+服务号Java版 v2.0--翟东平
- 【套餐】Javascript 设计模式实战--曾亮

查看评论



philhong

21楼 6天前 15:30发表

有笔误？

Field 又定义了一系列的 set 方法用来对其自身进行赋值。

```
public void set(Object obj, Object value);
```

```
public void getInt(Object obj,int value);
```

```
public void getLong(Object obj,long value)
throws IllegalArgumentException, IllegalAccessException;
```

```
public void getFloat(Object obj,float value)
throws IllegalArgumentException, IllegalAccessException;
```

```
public void getShort(Object obj,short value)
throws IllegalArgumentException, IllegalAccessException;
```

```
public void getDouble(Object obj,double value)
throws IllegalArgumentException, IllegalAccessException;
```

```
public void getChar(Object obj,char value)
throws IllegalArgumentException, IllegalAccessException;
```

```
public void getByte(Object obj,byte b)
throws IllegalArgumentException, IllegalAccessException;
```

```
public void getBoolean(Object obj,boolean b)
throws IllegalArgumentException, IllegalAccessException;
```

只有第一个方法是set，其余的都是get呀。。。



赫家旺

20楼 2017-08-05 09:53发表

感谢



沉在水中的鱼

19楼 2017-08-04 10:08发表

楼主写的很好，很细致。学习了



暗夜ノ使者

18楼 2017-07-31 19:37发表

反射的学习



秋毅

17楼 2017-07-31 17:55发表

文章很好,指的收藏



zhangwenhaojf40it

16楼 2017-07-21 14:12发表 评论

nice! 会继续关注的



qq_32127251

15楼 2017-07-14 15:47发表 评论

写的浅显易懂，仔细读完了感觉有收获，谢谢博主分享,很赞!



qq_35845339

14楼 2017-07-11 18:22发表 评论

在运行时判断任意一个对象所属的类；在运行时构造任意一个类的对象；在运行时判断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；生成动态代理。这句话对吗？



frank909

Re: 2017-07-11 21:34发表 评论

回复qq_35845339：动态代理和静态代理本质上是一样的，它同样要实现和被代理类相同的接口。只不过静态代理的代码是我们手动编写在 IDE 中，而动态代理的代码是借助于 Proxy 和 InvocationHandler 在 Java 系统的帮助下动态生成的。具体细节，你可以看下我另外一篇相关的博文。



qq_28432625

13楼 2017-07-10 18:31发表 评论

你在文章中的关于getDeclaredFields方法的描述不是很正确，根据官方文档的解释是可以获取到public，protected,default,private的属性，但不包括继承的属性，感谢你，提供了这么好的文章。

附：

Returns an array of `Field` objects reflecting all the fields declared by the class or interface represented by this `Class` object. This includes public, protected, default (package) access, and private fields, but excludes inherited fields. The elements in the array returned are not sorted and are not in any particular order. This method returns an array of length 0 if the class or interface declares no fields, or if this `Class` object represents a primitive type, an array class, or void.



frank909

Re: 2017-07-10 19:55发表 评论

回复qq_28432625：谢谢你的指出。现在已经更正。一时疏忽失误了，造成错误的表述。不好意思。



RogueZww

Re: 2017-07-10 19:32发表 评论

回复qq_28432625：同问这个问题



玩家六

12楼 2017-07-10 17:23发表 评论

getDeclaredFields可以获取public属性的，只是不能获取父类属性



frank909

Re: 2017-07-10 19:55发表 评论

回复jslcyly: 谢谢你的指出。现在已经更正。



qq_28432625

Re: 2017-07-10 18:33发表 评论

回复jslcyly: 是的，getDeclaredFields 可以获取所有的属性，但是不包括继承的属性



qq_34048982

11楼 2017-07-10 17:23发表 评论

收获很大。希望还能继续看到更新。



frank909

Re: 2017-07-10 19:57发表 评论

回复qq_34048982: 有更新的, 下一篇大概会在下周某个时候编写, 内容是大概是编写反射代码时可能遇到的问题。如果, 状态和时间允许的话我会再写另外一篇, 演示如何利用反射实战, 编写 Android 上的数据库框架。



qq_34048982

10楼 2017-07-10 17:22发表 评论

收获很大。希望还能继续看到更新。



神鸟自然

9楼 2017-07-10 16:30发表 评论

引用“Lee_vi”的评论:

有一个问题楼主知道吗? 我们公司有很多工具类, 传的参数都是类名.class, 请问这样写有什么用处...

自己去看看.readExcel()的传参类型啊.class的作用是实例化对象



Lee_vi

Re: 2017-07-10 16:49发表 评论

回复q975583865: 看了, 有的是实例化bean, 有的是实例化service, 有一点不懂的是实例化service后, 打了这个service的断点, 确实是跑了这个service里的方法, 但是不知道是怎么进这个方法。。我看了一下也没有用反射调方法的办法啊



神鸟自然

Re: 2017-07-10 16:55发表 评论

回复Lee_vi: 这个不清楚



神鸟自然

8楼 2017-07-10 16:27发表 评论

+1



雪吖头

7楼 2017-07-10 11:47发表 评论

很不错的分享。



frank909

Re: 2017-07-10 19:58发表 评论

回复u013035612: 谢谢你的肯定。



木子二月鸟

6楼 2017-07-10 10:44发表 评论

我就完全符合楼主文章的目标用户——门外汉自学JAVA, 非常受用, 收藏了, 谢谢楼主码这么多字~



frank909

Re: 2017-07-10 19:58发表 评论

回复muzierueniao: 谢谢你的肯定。



Lee_vi

5楼 2017-07-10 09:38发表 评论

有一个问题楼主知道吗? 我们公司有很多工具类, 传的参数都是类名.class, 请问这样写有什么用处吗?
例如:
//导入

ExcelReaderUtil.readExcel(Hotel_rplan_excel.class);



frank909

Re: 2017-07-10 20:01发表

回复Lee_vi: 我不知道你的具体代码,但是我大概能够想到你那个工具类的意图。

传入.class文件名的目的应该是生成POJO数据类。不同的class代表不同的数据类型,比如Hotel_rplan_excel.class应该指的是与hotel相关的数据实体,通过从excel中读取数据然后生成相应的数据实体。



Lee_vi

Re: 2017-07-11 08:40发表

回复bribblue: 好的,我再研究研究,谢谢!



冯尧

4楼 2017-07-09 21:26发表

讲解很详细,谢谢博主分享。



frank909

Re: 2017-07-09 21:31发表

回复u013038861: 谢谢你的肯定



盖丽男

3楼 2017-07-08 21:24发表

反射一开始真的很难理解



frank909

Re: 2017-07-09 21:30发表

回复u013036688: 不知这篇文章有没有给你加深一些理解。



菜鸡小王子

2楼 2017-07-07 23:12发表

很好,谢谢楼主



frank909

Re: 2017-07-09 21:31发表

回复qq_35181209: 谢谢你的肯定



疯行

1楼 2017-07-07 17:57发表

沙发,楼主很棒



frank909

Re: 2017-07-07 20:43发表

回复u011389515: 谢谢你的肯定

发表评论

用户名: qq_36596145





评论内容:



提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

[公司简介](#) | [招贤纳士](#) | [广告服务](#) | [联系方式](#) | [版权声明](#) | [法律顾问](#) | [问题报告](#) | [合作伙伴](#) | [论坛反馈](#)

 [网站客服](#)  [杂志客服](#)  [微博客服](#)  webmaster@csdn.net  400-660-0108 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 |

江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2017, CSDN.NET, All Rights Reserved 