个人资料

frank909

关注 发私信

博客专家

访问： 168864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] 反射进阶，编写反射代码值得注意的诸多细节

2017-07-27 23:56 7536人阅读 评论(16) 收藏 举报

分类：

Java 基础知识（5） Java 反射 3 板斧（3）

目录(?) [+]

前面一段时间，我编写了一篇关于 **Java** 反射基础知识的博文，内容挺多的，涉及到了 Class 的获取，Field、Method、Constructor、Array 及 Enum 的获取与操作。如果学会了这些知识，就能阅读或者是编写大多数反射相关代码。

但是，因为反射这一块的内容实在是太多了，编写代码过程中难免会遭遇到各种各样的 Exception，对于一个刚熟悉反射基础知识的新手而言，往往会感到深深的挫败感。其实，反射仍然算不难，但需要耐心与细心地对待它，本文的目的是列举编写反射代码中值得注意的一些细节，大家可以针对下面的目录部分，结合自己对反射这一块的熟悉程序，进行选择性地阅读。

2017年04月 (4)

展开

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(Reetrant 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&gt...

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?...

编写反射代码时，一些常见的异常。

如果你反射的基本知识都没有掌握，建议先仔细阅读我这篇文章《细说反射，Java 和 Android 开发者必须跨越的坎》

获取不存在的对象

比如获取不存在的 Class 对象，比如获取不到一个类中并不存在的 Field、Method 或者是 Constructor。 而 Field、Method 和 Constructor 都是一个 Class 对象中的成员。

获取不到 Class 对象

我们知道获取 Class 对象有 3 种方式。

1. 通过一个对象的 getClass() 方法。
2. 通过 .class 关键字。
3. 通过 Class.forName()。

前两种方式，基本上是没有什么值得注意的地方，需要注意的是第 3 种，因为 `Class.forName()` 传递进去的参数是一个字符串类型，所以理论上你可以这样编写代码。

```
1  Class test = Class.forName("hello world");
```

显然，在虚拟机中并不会存在这样一个类，所以，Java 提供了一个异常用来在获取不到 Class 文件时进行抛出。这个异常是 ClassNotFoundException。我们应该对于这一块进行处理。

```
1
```

```
Class cls1 = new String("1").getClass();

Class cls2 = int.class;

try {
    Class test = Class.forName("hello world");
} catch (ClassNotFoundException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
    System.out.println("find class error:"+e1.getMessage());
}
```

可以看到，只有通过 `Class.forName()` 的方式获取 Class 的时候才要进行异常的捕获处理，如果查找不到这个 Class 那么程序就会抛出异常。

```
java.lang.ClassNotFoundException: hello world


    at java.net.URLClassLoader.findClass(Unknown Source)
    at java.lang.ClassLoader.loadClass(Unknown Source)
    at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
    at java.lang.ClassLoader.loadClass(Unknown Source)
    at java.lang.Class.forName0(Native Method)
    at java.lang.Class.forName(Unknown Source)
```

获取不到 Field

获取不到 Field 的情况分两种：

1. 确实不存在这个 Field

2. 由于修饰符导致的权限问题。

获取不存在的 Field

我们先定义一个类 Base。

```
public class Base {
    public String b;
}
```

我们知道，获取一个 Class 中 Field 方式有 4 种。

```

public Field getField(String name);

public Field getDeclaredField(String name);

public Field[] getFields();

public Field[] getDeclaredFields()
```

现在，我们可以用 getField() 方法获取 b 这个 Field。

```
Class clzBase = Base.class;
try {
    Field fieldBase = clzBase.getField("b");
} catch (NoSuchFieldException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
} catch (SecurityException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}
```

但是，我们却没有办法去获取一个不存在的 Field，如 d。

```
Field fielddBase = clzBase.getField("d");
```

它会导致程序抛出 NoSuchFieldException 异常。

```
java.lang.NoSuchFieldException: d
    at java.lang.Class.getField(Unknown Source)
```

Field 存在，但获取不到

```
public class Base {
    public String b;

    int d;
}

public class NonExistTest {

    public static void main(String[] args) {

        Field fieldBase = clzBase.getField("d");
        System.out.println(Base.class.getSimpleName()+" has a field "+ fieldBase.getNa

    }

}
```

可以发现程序报错了。

```
java.lang.NoSuchFieldException: d
    at java.lang.Class.getField(Unknown Source)
```

这是根据 API 的定义，getField() 只能获取 public 属性的 Field。这个时候改用 getDeclairedField() 方法就可以了。

```
Field fieldBase1 = clzBase.getDeclaredField("d");
```

getDeclaredField() 能够获取一个 Class 中被定义的 Field。

有同学可能会想，既然 getDeclaredField() 能够获取所有修饰符类型的 Field，那么为什么还要整出一个 getField() 方法。

其实，getDeclaredField() 并非万能的，有一种属性它是没有办法获取到的，那就是从父类继承下来的 Field。

```
public class Base {
    public String b;

    int d;
}

public class Sub extends Base{

}

Class clzBase = Sub.class;

Field fieldBase1 = clzBase.getDeclaredField("b");
System.out.println(Sub.class.getSimpleName()+" has a field "+ fieldBase1.getName());
```

通过 getDeclaredField() 方法去获取 Sub 的父类 Base 中的 Field b，程序会抛出异常。

```
java.lang.NoSuchFieldException: b
    at java.lang.Class.getDeclaredField(Unknown Source)
```

但是，通过 getField() 方法却可以获取。

```
Field fieldBase1 = clzBase.getField("b");

System.out.println(clzBase.getSimpleName()+" has a field "+ fieldBase1.getName());
```

打印结果是：

```
Sub has a field b
```

但是，getField() 也有它的局限性，因为它只能获取被 public 修饰的 Field。例如

```
Field fieldBase1 = clzBase.getField("d");

System.out.println(clzBase.getSimpleName()+" has a field "+ fieldBase1.getName());
```

上面代码中，试图用 getField() 方法去获取一个非 public 修饰的 Field，结果自然是获取不到的。

```
java.lang.NoSuchFieldException: d
    at java.lang.Class.getField(Unknown Source)
```

所以，我们可以得到一张表，这张表可以说明 getField() 方法和 getDeclaredField() 方法的能力范围。

方法	作用对象	
	本 Class	SuperClass
getField	public	public
getDeclaredField	public、protected、defalut、private	×
getFields	public	public
getDeclaredFields	public、protected、defalut、private	×

另外，getFields() 和 getDeclaredField() 与 getField() 和 getDeclaredFields() 作用类似，只不过它们返回的是所有的符合条件的 Field 数组。

如何获取一个 Class 中继承下来的非 public 修饰的 Field

先观察这两个类。

```
public class Base {
    public String b;

    protected int d;
}

public class Sub extends Base{

}
```

Base 是 Sub 的父类，但是通过 Sub.class 对象是没有办法获取到 Base.class 中的 Field d,因为 d 是默认的修饰符，对于这一点 getField() 只能获取 pulic 属性的 Field 如 b,getDeclaredField() 更是无能为力。

那么，问题来了。

如果非要获取一个 Class 继承下来的非 public 修饰的 Field 要怎么办？

答案是通过获取这个 Class 的 superClass。然后调用这个 superClass 的 getDeclaredField() 方法。

```
Class clzBase = Sub.class;
Class superClass = clzBase.getSuperclass();

Field fieldBase1 = superClass.getDeclaredField("d");
System.out.println(clzBase.getSimpleName()+" has a field "+ fieldBase1.getName());
```

获取不到 Method

Method、Constructor 和 Field 一样都是 Class 的成员。所以，有很多共性。因为上一小节讲过 Field 的诸多细节，所以类似的地方我会一笔带过。

获取本身就不存在的 Method

```
public class Base {
    public String b;

    protected int d;

    void testDefault0(){};

    public void testPublic0(){}

    protected void testProtected0(){}

    private void testPrivate0(){}
}
```

获取本身就不存在的 Method，程序会抛出一个 NoSuchMethodException 异常。

```
Class class1 = Base.class;

try {
    Method methodtest = class1.getDeclaredMethod("hellowworld");
} catch (NoSuchMethodException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (SecurityException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

Method 存在却获取不到。

同 Field 类似，这里给出一张表。

方法	作用对象	
	本 Class	SuperClass
getMethod	public	public
getDeclaredMethod	public、protected、defalut、private	×
getMethods	public	public
getDeclaredMethods	public、protected、defalut、private	×

因为参数类型不匹配而找不到。

```
public class Base {
    public String b;

    protected int d;

```

```
6
7       public Base() {
8           super();
9       }
10
11      void testDefault0(){};
12
13      public void testPublic0(){}
14
15      protected void testProtected0(){}
16
17      private void testPrivate0(){}
18
19      public void test(int a,float b){}
20  }
```

现在要获取 test() 对应的 Method。

```
1
2   Method methodtest = class1.getDeclaredMethod("test");
```

如果不传入参数是获取不到的

```
1   java.lang.NoSuchMethodException: com.frank.test.notfound.Base.test()
2       at java.lang.Class.getDeclaredMethod(Unknown Source)
```

我们应该传入对应类型的参数

```
1   public Method getDeclaredMethod(String name, Class<?>... parameterTypes)
```

值得注意的是，后面接受的是可变参数，也就是参数的个数不定，并且类型都是 Class。

```
1   Method methodtest = class1.getDeclaredMethod("test",int.class,float.class);
```

当然，这样的形式也是可以的。

```
1   Method methodtest = class1.getDeclaredMethod("test",new Class[]{int.class,float.class}
```

如果一个 Method，参数过多的话，推荐使用后面一种方式。

获取 Method 的时候，参数一定要一一匹配，意思是参数的数目与类型都必须对应上，不然还是会抛出 NoSuchMethodException 异常，如下面：

```
1   Method methodtest = class1.getDeclaredMethod("test",int.class,double.class);
2
3   或者
4
5   Method methodtest = class1.getDeclaredMethod("test",int.class);
```

获取不到 Constructor

获取本身不存在的构造器。

```
1   Class class1 = Base.class;
2
3   Constructor constructor = class1.getConstructor(int.class);
```

需要注意的是，它仍然会抛出 NoSuchMethodException 这个异常。并没有一个 NoSuchConstructorException 的异常，其实想想也是，构造方法本质上其实也就是一个方法，只

不过在反射机制中，因为构造方法的特殊性和重要性，要以单独用 Constructor 把它与普通的 Method 区分开来了。

Constructor 存在，却获取不到。

同 Field 和 Method 一样，同样存在

```
getConstructor()

getDeclaredConstructor()

getConstructors()

getDeclaredConstructor()
```

方法	作用对象	
	本 Class	SuperClass
getConstructor	public	×
getDeclaredConstructor	public、protected、defalut、private	×
getConsturctors	public	×
getDeclaredConstructors	public、protected、defalut、private	×

不一样的是，getConstructor() 和 getConstructors() 也没有办法获取 SuperClass 的 Constructor。

参数不匹配的问题

这个同 Method 一样。

获取一个 Class 的内部类或者接口

```
public Class<?>[] getClasses()

public Class<?>[] getDeclaredClasses()
```

编写代码体会一下：

```
public class TestMember {

    class enclosingClass{};

    public interface testInterface{}

}


public class MemberTest {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Class clz = TestMember.class;

        Class[] members = clz.getDeclaredClasses();

        for ( Class c : members ) {
            System.out.println(c.toGenericString());
        }
    }

}
```

测试结果如下：

```
class com.frank.test.member.TestMember$enclosingClass
public abstract static interface com.frank.test.member.TestMember$testInterface
```

getInterfaces() 的作用

光看名字，大家可能都会觉得 getInterfaces() 的作用是获取一个类中定义的接口，但是其实不是的，getInterfaces() 获取的是一个类所有实现的接口。

```
public class A implements Runnable,Cloneable{

    @Override
    public void run() {

    }

}

public class MemberTest {

    public static void main(String[] args) {


        Class[] interfaces = A.class.getInterfaces();
        for ( Class c : interfaces ) {
            System.out.println(c.toGenericString());
        }
    }

}
```

结果是：

```
public abstract interface java.lang.Runnable
public abstract interface java.lang.Cloneable
```

反射中的权限问题

操纵非 public 修饰的 Field

```
public class TestPermission {

    private int value;



     public int getValue() {
        return value;
    }


    public void setValue(int value) {
        this.value = value;
    }


    public TestPermission() {
        super();
    }

```

```
}
```

TestPermission 这个类有一个 int 变量 value，现在用反射的手段获取它对应的 Field 然后操纵它的值。

```
public class AccessTest {

    public static void main(String[] args) {

        TestPermission test = new TestPermission();
        test.setValue(12);
        System.out.println(" value is :"+test.getValue());

        Class testclass = test.getClass();
        try {
            Field field = testclass.getDeclaredField("value");

            field.set(testclass, 30);

            System.out.println(" value is :"+test.getValue());
        } catch (NoSuchFieldException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        } catch (SecurityException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        } catch (IllegalArgumentException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

}
```

结果却报错了。

```
 value is :12
java.lang.IllegalAccessException:
Class com.frank.test.access.AccessTest
can not access a member of class com.frank.test.access.TestPermission with modifiers '
    at sun.reflect.Reflection.ensureMemberAccess(Unknown Source)
    at java.lang.reflect.AccessibleObject.slowCheckMemberAccess(Unknown Source)
    at java.lang.reflect.AccessibleObject.checkAccess(Unknown Source)
    at java.lang.reflect.Field.set(Unknown Source)
    at com.frank.test.access.AccessTest.main(AccessTest.java:19)
```

它抛出来的是一个 IllegalAccessException 异常。

修正方法也很简单。

```
Field field = testclass.getDeclaredField("value");

field.setAccessible(true);
```

可以看到结果正常了。

```
 value is :12
 value is :30
```

这里需要多说两句：我这里以 private 为例，其实 protected 和 default 也是一样的，但是它们不同于 private 的地方在于，它们在本 package 范围内是可见的，有兴趣的同学可以测试一下，测试代码在一个 package，而测试的类在另外一个 package。

操纵一个 final 类型的 Field

```
public class TestPermission {

    protected int value;

    public final int value1 = 0;

}
```

给 TestPermission 这个类新加一个字段 value1，但是它是一个被 final 修饰的属性，如果利用反射来修改它的值会怎么样呢？

```
TestPermission test = new TestPermission();


Class testclass = test.getClass();
try {
    Field field = testclass.getDeclaredField("value1");

    field.setInt(test,123);
    int a = field.getInt(test);

    System.out.println(" value1 is :"+a);

} catch (NoSuchFieldException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
} catch (SecurityException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
} catch (IllegalArgumentException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (IllegalAccessException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

结果却遭遇了异常。

```
java.lang.IllegalAccessException: Can not set final int field com.frank.test.access.Te
    at sun.reflect.UnsafeFieldAccessorImpl.throwFinalFieldIllegalAccessException(Unkno
    at sun.reflect.UnsafeFieldAccessorImpl.throwFinalFieldIllegalAccessException(Unkno
    at sun.reflect.UnsafeQualifiedIntegerFieldAccessorImpl.setInt(Unknown Source)
    at java.lang.reflect.Field.setInt(Unknown Source)
```

虽然，value1 是被 public 修饰，但是它同样被 final 修饰，这在正常的开发流程说明这个属性不能够再被改变。

如果要解决这个问题，同样可以使用 setAccessible(true) 方法。

```
Field field = testclass.getDeclaredField("value1");

field.setInt(test,123);
int a = field.getInt(test);

System.out.println(" value1 is :"+a);
```

操纵非 public 修饰的 Method

```
public class TestPermission {
```

```
2
3
4
5       public TestPermission() {
6           super();
7       }
8
9
10      private void justSay() {
11          System.out.println("just for test meothod");
12      }
13  }
14
15  public class AccessTest {
16
17      public static void main(String[] args) {
18          Class clz = TestPermission.class;
19
20          try {
21              Method method = clz.getDeclaredMethod("justSay");
22              method.invoke(clz.newInstance(),null);
23          } catch (NoSuchMethodException e) {
24              // TODO Auto-generated catch block
25              e.printStackTrace();
26          } catch (SecurityException e) {
27              // TODO Auto-generated catch block
28              e.printStackTrace();
29          } catch (IllegalAccessException e) {
30              // TODO Auto-generated catch block
31              e.printStackTrace();
32          } catch (IllegalArgumentException e) {
33              // TODO Auto-generated catch block
34              e.printStackTrace();
35          } catch (InvocationTargetException e) {
36              // TODO Auto-generated catch block
37              e.printStackTrace();
38          } catch (InstantiationException e) {
39              // TODO Auto-generated catch block
40              e.printStackTrace();
41          }
42      }
43
44  }
```

试图通过反射去操作一个 private 方法，结果报错了。

```
1  java.lang.IllegalAccessException: Class com.frank.test.access.AccessTest can not acces
2      at sun.reflect.Reflection.ensureMemberAccess(Unknown Source)
3      at java.lang.reflect.AccessibleObject.slowCheckMemberAccess(Unknown Source)
4      at java.lang.reflect.AccessibleObject.checkAccess(Unknown Source)
5      at java.lang.reflect.Method.invoke(Unknown Source)
6      at com.frank.test.access.AccessTest.main(AccessTest.java:13)
```

它抛出来的是一个 IllegalAccessException 异常。

修正方法也很简单。

```
1  Method method = clz.getDeclaredMethod("justSay");
2  method.setAccessible(true);
3  method.invoke(clz.newInstance(),null);
```

操纵非 public 修饰的 Constructor

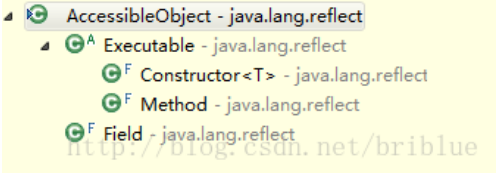
同前面两种，同样是通过 setAccessible(true) 来搞定。

所以，在反射中如果要操作被 private 修饰的对象，那么就必须调用它的 setAccessible(true)。

setAccessible() 的秘密

我们已经知道 Field、Method 和 Constructor 都有 setAccessible() 这个方法，至于是什么呢？

这是因为它们有共同的祖先 AccessObject。




```
public class AccessibleObject implements AnnotatedElement {
    public void setAccessible(boolean flag) throws SecurityException {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) sm.checkPermission(ACCESS_PERMISSION);
        setAccessible0(this, flag);
    }

    /* Check that you aren't exposing java.lang.Class.<init> or sensitive
       fields in java.lang.Class. */
    private static void setAccessible0(AccessibleObject obj, boolean flag)
        throws SecurityException
    {
        if (obj instanceof Constructor && flag == true) {
            Constructor<?> c = (Constructor<?>)obj;
            if (c.getDeclaringClass() == Class.class) {
                throw new SecurityException("Cannot make a java.lang.Class" +
                                            " constructor accessible");
            }
        }
        obj.override = flag;
    }

    /**
     * Get the value of the {@code accessible} flag for this object.
     *
     * @return the value of the object's {@code accessible} flag
     */
    public boolean isAccessible() {
        return override;
    }
}
```

可以看到，主要是设置内部一个 override 变量。

那么，我们再以 Method 的 invoke 方法为例。

```
public Object invoke(Object obj, Object... args)
        throws IllegalAccessException, IllegalArgumentException,
           InvocationTargetException
{
    if (!override) {
        if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {
            Class<?> caller = Reflection.getCallerClass();
            checkAccess(caller, clazz, obj, modifiers);
        }
    }
    MethodAccessor ma = methodAccessor;             // read volatile
    if (ma == null) {
        ma = acquireMethodAccessor();
    }
    return ma.invoke(obj, args);
```

```
}
```

如果一个 Method 的 overide 为 false 的话，它就会根据 Modifiers 判断是否具有访问权限。

[Reflection.java]

```
public static boolean quickCheckMemberAccess(Class<?> memberClass,
                                             int modifiers)
{
    return Modifier.isPublic(getClassAccessFlags(memberClass) & modifiers);
}
```

这个方法主要是简单地判断 modifiers 是不是 public，如果不是的话就返回 false。所以 protected、private、default 修饰符都会返回 false,只有 public 都会返回 true。

而不是 public 修饰的话会执行下面的代码

```
void checkAccess(Class<?> caller, Class<?> clazz, Object obj, int modifiers)
        throws IllegalAccessException
{
    if (caller == clazz) {  // quick check
        return;             // ACCESS IS OK
    }
    Object cache = securityCheckCache;  // read volatile
    Class<?> targetClass = clazz;
    if (obj != null
        && Modifier.isProtected(modifiers)
        && ((targetClass = obj.getClass()) != clazz)) {
        // Must match a 2-list of { caller, targetClass }.
        if (cache instanceof Class[]) {
            Class<?>[] cache2 = (Class<?>[]) cache;
            if (cache2[1] == targetClass &&
                cache2[0] == caller) {
                return;     // ACCESS IS OK
            }
            // (Test cache[1] first since range check for [1]
            // subsumes range check for [0].)
        }
    } else if (cache == caller) {
        // Non-protected case (or obj.class == this.clazz).
        return;             // ACCESS IS OK
    }

    // If no return, fall through to the slow path.
    slowCheckMemberAccess(caller, clazz, obj, modifiers, targetClass);
}

// Keep all this slow stuff out of line:
void slowCheckMemberAccess(Class<?> caller, Class<?> clazz, Object obj, int modifiers,
                           Class<?> targetClass)
    throws IllegalAccessException
{
    Reflection.ensureMemberAccess(caller, clazz, obj, modifiers);

    // Success: Update the cache.
    Object cache = ((targetClass == clazz)
                    ? caller
                    : new Class<?>[] { caller, targetClass });

    // Note:  The two cache elements are not volatile,
    // but they are effectively final.  The Java memory model
    // guarantees that the initializing stores for the cache
    // elements will occur before the volatile write.
    securityCheckCache = cache;         // write volatile
}
```

最终通过 Reflection 这个类的静态方法 ensureMemberAccess() 确认。

```
public static void ensureMemberAccess(Class<?> currentClass,
                                       Class<?> memberClass,
                                       Object target,
                                       int modifiers)
        throws IllegalAccessException
    {
        if (currentClass == null || memberClass == null) {
            throw new InternalError();
        }

        if (!verifyMemberAccess(currentClass, memberClass, target, modifiers)) {
            throw new IllegalAccessException("Class " + currentClass.getName() +
                                             " can not access a member of class " +
                                             memberClass.getName() +
                                             " with modifiers \"" +
                                             Modifier.toString(modifiers) +
                                             "\"");
        }
    }

```

如果没有访问权限，程序将会在此抛出一个 IllegalAccessException 的异常。

所以，如果通过反射方式去操作一个 Field、Method 或者是 Constructor，最好先调用它的 setAccessible(true) 以防止程序运行异常。

Class.newInstance() 和 Constructor.newInstance() 的区别

我们知道，通过反射创建一个对象，可以通过 Class.newInstance() 和 Constructor.newInstance() 两种。那么，它们有什么不同的地方吗？

总体而言，Class.newInstance() 的使用有严格的限制，那就是一个 Class 对象中，必须存在一个无参数的 Constructor，并且这个 Constructor 必须要有访问的权限。

```
package com.frank.test.newinstance;

public class TestCreate {

}

public class NewInstanceTest {

    public static void main(String[] args) {
        Class clz = TestCreate.class;

        try {
            Constructor[] constructors = clz.getDeclaredConstructors();
            for ( Constructor c : constructors ) {
                System.out.println(c.toString());
            }
            TestCreate obj = (TestCreate) clz.newInstance();

            System.out.println(obj.toString());
        } catch (InstantiationException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            // TODO Auto-generated catch block
```

```
            e.printStackTrace();
        }
    }

}
```

如上面的代码，我们试图通过 Class.newInstance() 的方法创建一个 TestCreate 对象实例。

```
public com.frank.test.newinstance.TestCreate()
com.frank.test.newinstance.TestCreate@15db9742
```

Java 默认会给每个类加上一个无参的构造方法，并且它的修饰符是 public。现在，我们作一个小小的变动。

```
public class TestCreate {
    private TestCreate() {}
}
```

给 TestCreate 添加相应的构造方法，但是是 private 的访问权限，我们再看测试结果。

```
private com.frank.test.newinstance.TestCreate()
java.lang.IllegalAccessException:
Class com.frank.test.newinstance.NewInstanceTest
can not access a member of class com.frank.test.newinstance.TestCreate with modifiers
    at sun.reflect.Reflection.ensureMemberAccess(Unknown Source)
    at java.lang.Class.newInstance(Unknown Source)
    at com.frank.test.newinstance.NewInstanceTest.main(NewInstanceTest.java:15)
```

但是，通过 Constructor.newInstance() 却没有这种限制。Constructor.newInstance() 适应任何类型的 Constructor,无论它们有参数还是无参数，只要通过 setAccessible() 控制好访问权限就可以了。

所以，一般建议优先使用 Constructor.newInstance() 去创建一个对象实例。

谨慎使用 Method.invoke() 方法

通过 Method 调用它的 invoke 方法，这应该是整个反射机制中的灵魂了。但是，正因为如此，我们就得小心处理它的很多细节。

静态方法和非静态方法的区别

```
public class TestMethod {

    static void test1() {
        System.out.println("test1");
    }

    void test2() {
        System.out.println("test1");
    }
}

public class MethodDetailTest {

    public static void main(String[] args) {
        TestMethod.test1();

        new TestMethod().test2();
```

```
    }

}
```

我们知道，在正常流程开发中，调用静态方法直接用 类.方法() 的形式就可以调用，而调用非静态的方法那就必须先创建对象，再通过对象调用相应的方法。

那么，对应到反射中如何正常调用静态方法和非静态方法呢？

```
try {
    Method method1 = clz.getDeclaredMethod("test1");

    method1.invoke(null);
} catch (NoSuchMethodException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (SecurityException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (IllegalAccessException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (IllegalArgumentException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (InvocationTargetException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

try {
    Method method2 = clz.getDeclaredMethod("test2");

    method2.invoke(new TestMethod());
} catch (NoSuchMethodException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (SecurityException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (IllegalAccessException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (IllegalArgumentException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (InvocationTargetException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

关键在于 Method.invoke() 的第一个参数，static 方法因为属于类本身所以不需要对象，那么非静态方法的就必须要传入一个对象了，而且这个对象也必须是与 Method 对应的。

我们可以测试一下，看看随便给 invoke() 方法传递一个对象会如何？

```
Method method2 = clz.getDeclaredMethod("test2");

method2.invoke(new String("134"));
```

随便给它传递一个字符串，结果报错了。

```
java.lang.IllegalArgumentException: object is not an instance of declaring class
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
```

```
    at java.lang.reflect.Method.invoke(Unknown Source)
    at com.frank.test.method.MethodDetailTest.main(MethodDetailTest.java:39)
```

抛出了一个 IllegalArgumentException 的异常，提示说 object 不是声明的 class 的对象实例。

Method.invoke() 参数的秘密

```
public Object invoke(Object obj, Object... args)
        throws IllegalAccessException, IllegalArgumentException,
           InvocationTargetException
```

第一个 Object 参数代表的是对应的 Class 对象实例，这在上面一节已经见识到了。而后面的参数就是可变形参了，它接受多个参数。我们考虑一种特殊情况。

```
public class TestT<T>{

    public  void test(T t){}

}
```

这是一个泛型类，T 表示接受任意类型的参数。

```
public class MethodDetailTest {

    public static void main(String[] args) {

        Class clzT = TestT.class;
        try {
            Method tMethod = clzT.getDeclaredMethod("test",Integer.class);
            tMethod.setAccessible(true);
            try {
                tMethod.invoke(new TestT<Integer>(),1);
            } catch (IllegalAccessException | IllegalArgumentException
                    | InvocationTargetException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        } catch (NoSuchMethodException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        } catch (SecurityException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
    }
}
```

结果却报错。

```
java.lang.NoSuchMethodException: com.frank.test.method.TestT.test(java.lang.Integer)
    at java.lang.Class.getDeclaredMethod(Unknown Source)
    at com.frank.test.method.MethodDetailTest.main(MethodDetailTest.java:14)
```

提示找不到这个方法。原因是**类型擦除**。

当一个方法有泛型参数时，编译器会自动向上转型，T 向上转型是 Object。所以实际上是

```
void test(Object t);
```

上面的代码试图去找 `test(Integer t)` 这个方法，自然是找不到。

```
Method tMethod = clzT.getDeclaredMethod("test",Object.class);
tMethod.setAccessible(true);
```

```
3
4  tMethod.invoke(new TestT<Integer>(),1);
```

这样编写代码才正常。

Method 中处理 Exception

我们平常开发中，少不了与各种异常打交道。

```
1  public class TestMethod {
2
3      public static class Super{}
4
5      public static class Sub extends Super{}
6
7      static void test1(Sub sub) {
8          System.out.println("test1");
9      }
10
11     void test2() throws IllegalArgumentException {
12         System.out.println("test2");
13         throw new IllegalArgumentException("只是用来测试异常");
14     }
15
16 }
```

为了便于测试，给 test2() 方法添加了一个异常。在 Java 反射中，一个 Method 执行时遭遇的异常会被包装在一个特定的异常中，这个异常就是 InvocationTargetException。

```
1  try {
2      Method method2 = clz.getDeclaredMethod("test2");
3
4      method2.invoke(clz.newInstance());
5  } catch (NoSuchMethodException e) {
6      // TODO Auto-generated catch block
7      e.printStackTrace();
8  } catch (SecurityException e) {
9      // TODO Auto-generated catch block
10     e.printStackTrace();
11 } catch (IllegalAccessException e) {
12     // TODO Auto-generated catch block
13     e.printStackTrace();
14 } catch (IllegalArgumentException e) {
15     // TODO Auto-generated catch block
16     e.printStackTrace();
17 } catch (InvocationTargetException e) {
18     // TODO Auto-generated catch block
19     e.printStackTrace();
20 } catch (InstantiationException e) {
21     // TODO Auto-generated catch block
22     e.printStackTrace();
23 }
```

它会出现异常。

```
1  test2
2  java.lang.reflect.InvocationTargetException
3      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
4      at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
5      at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
6      at java.lang.reflect.Method.invoke(Unknown Source)
7      at com.frank.test.method.MethodDetailTest.main(MethodDetailTest.java:63)
8  Caused by: java.lang.IllegalArgumentException: 只是用来测试异常
9      at com.frank.test.method.TestMethod.test2(TestMethod.java:15)
10     ... 5 more
```

我们如果需要在代码中手动获取被包装在 InvocationTargetException 中的异常，就要通过它的方法手动获取。

```
catch (InvocationTargetException e) {
    // TODO Auto-generated catch block
    //e.printStackTrace();
    Throwable cause = e.getCause();
    System.out.println(cause.toString());

}
```

通过调用 InvocationTargetException 对象的 getCause() 方法，会得到 Throwable 对象，原始的异常就包含在里面。打印结果如下：

```
java.lang.IllegalArgumentException: 只是用来测试异常
```

总结

反射牛逼的地方，在于它可以绕过一定的安全机制，比如操纵 private 修饰的 Field、Method、Constructor。并且它还有能力改变 final 属性的 Field。

但是反射头痛的地方就是它有太多的 Exception 需要处理。

异常名称	原因
ClassNotFoundException	1. Class.forName() 传入的包名有误 2. Class 本身不存在
NoSuchFieldException	1. Field 名称不正确 2. getDeclaredField 和 getField() 方法使用不当。
NoSuchMethodException	1. 方法本身不存在 2. 传入的参数的类型不匹配 3. 传入的参数个数不匹配
IllegalAccessException	1. 访问非 public 修饰的对象如 Field、Method、Consturctor。 2. 操作 final 修饰的 Field
IllegalArgumentException	1. Method.invoke 中参数匹配 2. Field 操作时设置的值不匹配 3. Constructor.newInstance() 传入的参数不匹配
InvocationTargetException	1. Method 运行时产生异常 2. Constructor.newInstance() 作用时产生异常
InstantiationException	Class.newInstance() 或者 Constructor.newInstance() 异常

总之，还是那句话：反射有风险，编码需谨慎。

顶 13 踩 1

- 上一篇 轻松学，听说你还没有搞懂 Dagger2
- 下一篇 Java 泛型，你了解类型擦除吗？

相关文章推荐

- 10分钟从源码编译到部署ceph环境
- Ceph 多节点quick部署
- 浅谈虚拟机的基本原理
- 细说反射，Java 和 Android 开发者必须跨越的坎
- 一款能够提高工作效率的小软件
- Server JRE 简介
- android应用开发-从设计到实现 4-4版本管理
- Android: Android Thumbnail 攻略
- Java学习之整数类型最大值最小值
- Tortoise SVN使用方法，简易图解（转）

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

mcg-helper 10楼 5天前 10:37发表

分享一款代码生成器，拖拽式组件结合流式处理，很容易的访问数据库、http、文件读写操作等等，支持编写javascript、java、freemaker，功能丰富强大，编制规则，可生成一切想要的代码，详见：http://blog.csdn.net/LoginandPwd/article/details/76944900，更多资讯：http://blog.csdn.net/LoginandPwd/article

Codeoftrip 9楼 2017-08-02 16:08发表

反射中的权限问题

操纵非 public 修饰的 Field 这个案例中运行后的结果有点小问题

frank909 Re: 2017-08-02 19:53发表

回复Codeoftrip：有什么问题，你能说详细点吗?

郭二关 8楼 2017-08-02 11:16发表

看过你很多blog，写得很好，关注了

frank909 Re: 2017-08-02 19:53发表

回复郭二关：谢谢你的关注。

zejian_ 7楼 2017-08-01 11:39发表

回复naskado：京东还加班少？骗谁啊？哈哈哈，工资几乎是大公司中比较低的吧。。。

赵尽朝 6楼 2017-07-31 17:52发表

很详细的教程

赵尽朝 5楼 2017-07-31 17:52发表

总结的非常详细，谢谢分享

frank909 Re: 2017-07-31 20:27发表

回复赵尽朝：谢谢支持。

12期-焦玉丽 4楼 2017-07-31 17:11发表

感谢分享 积累

frank909 Re: 2017-07-31 20:27发表

回复12期-焦玉丽：谢谢支持。

aiming66 3楼 2017-07-31 17:00发表

总结的非常的棒。

frank909 Re: 2017-07-31 20:27发表

回复aiming66：谢谢你的肯定。

aiming66 2楼 2017-07-31 17:00发表

谢谢分享。

android_jiajia 1楼 2017-07-31 16:44发表

看完了，写的很好。

frank909 Re: 2017-07-31 20:27发表

回复android_jiajia：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